Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

46

F. L. Bauer, P. Brinch-Hansen, E. W. Dijkstra,
A. Ershov, D. Gries, M. Griffiths, C. A. R. Hoa
G. Seegmüller, W. A. Wulf

Language Hierarchies and Interfaces

International Summer School

Edited by F. L. Bauer and K. Samelson



Springer-Verlag
Berlin · Heidelberg · New York 1976

Editorial Board
P. Brinch Hansen · D. Gries · C. Moler · G. Seegmüller · J. Stoer
N. Wirth

Editors
Prof. Dr. Dr. h. c. Dr. Friedrich L. Bauer
Prof. Dr. Klaus Samelson
Institut für Informatik
der Technischen Universität
Arcisstraße 21
8000 München 2/BRD

Library of Congress Cataloging in Publication Data
Main entry under title:

Language hierarchies and interfaces.

(Lecture notes in computer science ; 46)
"The international summer school took place from July 23 to August 2, 1975, in Marktoberdorf ... and was sponsored by the NATO Scientific Affairs Division under the 1975 Advanced Study Institutes programme."
Includes bibliographical references and index.
1. Electronic digital computers--Programming--Congresses. 2. Programming languages (Electronic computers) --Congresses. I. Bauer, Friedrich Ludwig, 1924- II. Samelson, Klaus, 1918- III. North Atlantic Treaty Organization. Division of Scientific Affairs.

IV. Series.
QA76.6.L335 001.6'42 76-54339

AMS Subject Classifications (1970): 68-02, 68A05
CR Subject Classifications (1974): 4.12, 4.20, 4.22, 4.30, 4.31, 4.32, 4.34, 5.24

ISBN 3-540-07994-7 Springer-Verlag Berlin · Heidelberg · New York
ISBN 0-387-07994-7 Springer-Verlag New York · Heidelberg · Berlin

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks.

Under § 54 of the German Copyright Law where copies are made for other than private use, a fee is payable to the publisher, the amount of the fee to be determined by agreement with the publisher.

© by Springer-Verlag Berlin · Heidelberg 1976
Printed in Germany
Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.

PREFACE

The International Summer School took place from July 23 to August 2, 1975, in Marktoberdorf. This Summer School was organized under the auspices of the Technical University Munich, and was sponsored by the NATO Scientific Affairs Division under the 1975 Advanced Study Institutes Programme. Partial support for this conference was provided by the European Research Office and the National Science Foundation.

Contents

INTRODUCTION

E. W. Dijkstra — ON THE TEACHING OF PROGRAMMING, I.E. ON THE TEACHING OF THINKING 1

CHAPTER 1.: CONCURRENCY

C.A.R. Hoare — PARALLEL PROGRAMMING: AN AXIOMATIC APPROACH 11

1. Introduction 12
2. Concepts and Notations 12
3. Disjoint Processes 14
4. Competing Processes 17
5. Cooperating Processes 20
6. Communicating Programs 24
7. Colluding Processes 29
8. Machine Traps 34
9. Conclusion 36
References 38
Appendix 40

E. W. Dijkstra — ON-THE-FLY GARBAGE COLLECTION: AN EXERCISE IN COOPERATION 43

Introduction 44
Preliminary Investigations 46
A Coarse-grained Solution 48
A Solution with a Fine-grained Collector 52
A Solution with a Fine-grained Mutator as well 53
In Retrospect 54
History and Acknowledgements 55
References 55
Appendix 55

D. Gries — AN EXERCISE IN PROVING PARALLEL PROGRAMS CORRECT 57

1. Introduction 58
2. Definition and Use of the Language 59
3. On-the-fly Garbage Collection 63
4. Proof of Correctness of the Mutator-Collector System 69
4.1. Proof Outline for the Main Program 71
4.2. Proof Outline for the Marking Phase 71
4.3. Proof Outline for the Collecting Phase 73
4.4. Proof of Properties of the Mutator 73

4.5. Showing Non-interference 75
5. Concluding Remarks 78
References 81

P. Brinch Hansen — THE PROGRAMMING LANGUAGE CONCURRENT PASCAL 82

1. The Purpose of Concurrent Pascal 84
1.1. Background 84
1.2. Processes 84
1.3. Monitors 85
1.4. System Design 88
1.5. Scope Rules 93
1.6. Final Remarks 95
2. The Use of Concurrent Pascal 96
2.1. Introduction 96
2.2. Processes 96
2.3. Monitors 100
2.4. Queues 103
2.5. Classes 104
2.6. Input/Output 105
2.7. Multiprocess Scheduling 106
2.8. Initial Process 108
Acknowledgements 110
References 110

CHAPTER 2.: PROGRAM DEVELOPMENT

E. W. Dijkstra — GUARDED COMMANDS, NON-DETERMINACY AND A CALCULUS FOR THE DERIVATION OF PROGRAMS 111

1. Introduction 111
2. Two Statements made from Guarded Commands 112
3. Formal Definition of the Semantics 114
3.1. Notational Prelude 114
3.2. The Alternative Construct 116
3.3. The Repetitive Construct 118
4. Formal Derivation of Programs 119
5. Concluding Remarks 122
Acknowledgements 123
References 124

M. Griffiths — PROGRAM PRODUCTION BY SUCCESSIVE TRANSFORMATION 125

1. Introduction 126
2. Successive Transformation 127
2.1. The Problem 127
2.2. Solution by Invariants 128
2.3. Solution by Successive Transformation 129
2.4. Discussion 131
3. Transformation Methods 133
3.1. Recursion and Iteration 133
3.2. Introduction of a Variable 134
3.3. Function Inversion and Counting 136
3.4. Changes in Data Structure 138
3.5. Program Schemes and Automatic Transformation 139

	3.6. Discussion	140
	4. Some Implications	142
	4.1. Language Design	142
	4.2. System Structure	143
	4.3. The Multi-Language Problem	144
	4.4. Efficiency	145
	4.5. Use of Static Information	146
	5. Conclusion	147
	5.1. Associated Research	147
	5.2. Final Remarks	148
	References	149
	Acknowledgements	152
F. L. Bauer	PROGRAMMING AS AN EVOLUTIONARY PROCESS	153
	First Lecture: METAMORPHOSES	155
	Styles of Programming	155
	Properties Defining Recursion and their Derivation	158
	Second Lecture: TECHNIQUES	162
	Transition between Recursion and Iterative Notation	162
	The COOPER Transformation as an Example for the Recursion Removal	165
	Function Inversion	167
	Third Lecture: DANGEROUS CORNERS	172
	Sequentialization and the Danger of Destruction	172
	Sharing of Variables	174
	The Method of Invariants	176
	Conclusion: PROGRAMMING AS A PROCESS	179
	References	181
C. A. R. Hoare	PROOF OF CORRECTNESS OF DATA REPRESENTATION	183
	1. Introduction	183
	2. Concepts and Notations	183
	3. Example	184
	4. Semantics and Implementation	186
	5. Criterion of Correctness	186
	6. Proof Method	187
	7. Proof of Smallintset	188
	7.1. Initialisation	188
	7.2. Has	188
	7.3. Insert	189
	7.4. Remove	189
	8. Formalities	190
	9. Extensions	191
	9.1. Class Parameters	191
	9.2. Dynamic Object Generation	192
	9.3. Remote Identification	192
	9.4. Class Concatenation	192
	9.5. Recursive Class Declaration	192
	References	193
F. L. Bauer	APPENDIX: A PHILOSOPHY OF PROGRAMMING	194
	First Lecture: A Unified, Conceptual, Basis of Programming	196

Second Lecture:
The Role of Structuring in Programming 204
Third Lecture:
System Uniformity of Software and Hardware 216
Final:
Our Responsibility 227
Literature 229
Appendix:
Variables Considered Harmful 230
Procedures and their Parameters 231
Building Procedures from Primitives 233
Result Parameters 235
Variables 237

CHAPTER 3.: OPERATING SYSTEMS STRUCTURE

C.A.R. Hoare — THE STRUCTURE OF AN OPERATING SYSTEM 242

1. *Introduction* 243
2. *A Class with Inner* 244
3. *A Nested Class Declaration* 246
4. *Compile Time Checking* 247
5. *Multilevel Structuring* 248
6. *A Third Level* 249
7. *Error Control* 250
8. *Accounting* 252
9. *The Top Level* 253
10. *Protection* 254

Conclusion 256
Acknowledgements 256
References 265

G. Seegmüller — LANGUAGE ASPECTS IN OPERATING SYSTEMS 266

1. *The Role of Language in Operating Systems* 268
1.1. *Language und Function* 268
1.2. *Language and People* 269
1.3. *Language and Computing Systems* 270
1.4. *Language and System Construction* 274
2. *Are there Special Requirements for Systems Programming* 277
3. *A Remark on Current Systems Programming Languages* 279
4. *How Does the Successful Systems Programmer Survive* 280
5. *Design Criteria for an Operating System Programming Language* 281
6. *Language Mechanisms Assisting in the Construction of Structured Systems* 282
7. *Example: The Language System ASTRA* 285
8. *Concluding Remarks* 289
9. *Acknowledgements* 290
10. *Literature* 290

W. A. Wulf — STRUCTURED PROGRAMMING IN THE BASIC LAYERS OF AN OPERATING SYSTEM 293

Introduction 294
A Personal View of Structure, Programs, and Programming 295
Comments on "Hierarchy" 301
Layers of Operating Systems 304
The "Basic" Layers - Some Assumptions 311
A High Level Model of a Hydry-like System 314
The Lowest Level of Hydra — Machine Assumptions 318
A Bottom-Up-Presentation 320
Some Concluding Remarks and Caveats 342

E. W. Dijkstra — A TIME-WISE HIERARCHY IMPOSED UPON THE USE OF A TWO-LEVEL STORE 345

Introduction 346
The Role of the Replacement Algorithm in a Multiprogramming Environment 348
About the Ideal Window Size 350
About the Degree of Multiprogramming 351
About the Adjustment of Window Size 352
Monotonic Replacement Algorithms 353
The Time-wise Hierarchy 354
Efficiency and Flexibility 355
Temptations to be Resisted 356
Analyzing the Mismatch between Configuration and Workload 357
Acknowledgements 357

CHAPTER 4.: PROGRAMMING SYSTEMS STRUCTURE

A. P. Ershov — PROBLEMS IN MANY-LANGUAGE SYSTEMS 358

Lecture 1:
1. Introduction and Preview of the BETA System 361
1.1. Introduction 361
1.2. Brief Overview of the System 363
1.3. Plan of the Course 366
1.4. Example 366
Lecture 2:
2. Internal Language of the BETA System 368
2.1. Design Concepts 368
2.2. INTEL Program Scheme 369
2.3. INTEL Objects 370
2.4. INTEL Statements 373
2.5. Transput 376
2.6. Parallelism 376
2.7. Discussion 377

Lecture 3:
3. Decomposition and Synthesis in the BETA System 380
3.1. Introduction 380
3.2. Lexical Analysis. Executive Procedures 382
3.3. Syntactic Analysis and Parsing 384
3.4. Semantic Analysis and Synthesis 384
3.5. Lexical Information 385
3.6. Syntactic Information 385
3.7. Semantic Information 386
3.8. Information for Synthesis and Code Generation 387
3.9. Discussion 388
Lecture 4:
4. Optimization and Code Generation in the BETA System 391
4.1. Collection of the Optimising Transformations 391
4.2. Analysis 393
4.3. Factorization 394
4.4. Preliminary Code Generation 394
4.5. Memory Allocation 395
4.6. The Coding of Subroutines and Procedures 396
4.7. Final Code Generation 397
Lecture 5:
5. Compiler Writing Systems as a Factor in Unification and Comparison of Programming Languages 398
5.1. Introduction 398
5.2. Universal Executive Procedures for Synthesis 390
5.3. Criteria for Evaluation of Programming Languages 402
5.4. Data Types 403
5.5. Name Declarations 405
5.6. Resume of the Comparison 407
5.7. Conclusion 407
Acknowledgements 409
References 410
Index of Terms 411

INTRODUCTION

On the teaching of programming, i.e. on the teaching of thinking.

E. W. Dijkstra, Burroughs, Nuenen, Netherlands

It is said that the murderer must return to the place of his crime. I feel myself in the murderer's position, for I found in my files an unfinished manuscript, of some seven years ago, with the ambitious title: "On the orginazation of one's intellect in scientific research". I quote its first paragraph:

> "This is a ridiculous project. It is the kind of project for which one must be slightly drunk, before one dares to undertake it, all the time knowing that one will be sober thrice, before it will have been finished. My only excuse is, that I know its ridiculousness."

Honesty forces me to admit that since the above was written, I have been sober far more than three times and that the manuscript is still as unfinished as I had left it.

* * *

Before starting with the real subject of this talk, viz. the teaching of thinking, I must dwell for a short while at the "i.e." in the title that equates the teaching of programming to the teaching of thinking. We still find the opinion that the teaching of programming boils down to the teaching of programming languages. This misconception is not restricted to the organizers of the vocational training courses in the field and their victims. On many an application for a research fellowship I find the applicant in his curriculum proudly advertizing his fluency in, or at least experience with, a wild variety of instruction codes, programming languages and systems. An organization that in my country claims a central role in the education of programming professionals gives two courses in programming: its introductory course covers FORTRAN and some ALGOL 60, its advanced course deals with .... COBOL! We call such courses "driving lessons". The student's attention is almost entirely absorbed by becoming fully familiar with the ideosyncrasies of those various languages and is made to believe that the more of those ideosyncrasies he understands, the better a programmer he will be. But no one tells him that all those bells and whistles --those so-called "powerful features"-- belong more to the problem set than to the solution set. Nobody tells him that at best he will become an expert coder of trivial algorithms, and that his difficulties will never arise from the problem itself, but always from the funny way the (usually given) algorithm has to be coded.

After a number of such courses, the really tough problems are way beyond his mental horizon. His difficulties are not the ones I think worthwhile to discuss, and I shall confine my attention to the difficulties of solving intrinsically hard problems, problems that come only within reach by the time that we use an adequate, elegant mini-language and our vision is no longer blurred by the wealth of mutually conflicting "powerful features" of the more baroque programming languages. Addressing an audience of computing scientists, I assume that I don't need to belabour this point any further: by the time that all knowledge of one's programming language can conveniently be formulated on a single sheet of paper and can be gotten across the limelight in about twenty minutes lecturing, it becomes clear what "programming" really amounts to, viz. designing algorithmic solutions, and that activity requires the ability to think effectively more than anything else. Hence the "i.e." in my title.

* * *

Talking about "thinking" and, even worse, its teaching is regarded by many as something very ambitious, on the verge of foolishness, a feeling which I shared seven years ago. As the reasons for such a feeling are so obvious, some disclaimers and restrictions of the subject matter seem indicated.

First of all, I am not going to speculate how the brain works, nor am I going to suggest models for it --be they neurological, physiological, philosophical or what have you--. My interest is in <u>using</u> the facility and the fact that some writers of detective stories insist on referring to their hero's "grey matter" has alway struck me as confusing and as a symptom of bad taste; for after all, what has its colour to do with it?

Secondly, "thinking" is a word covering such a wild variety of vague activities that we should restrict it for the purpose of our discussion. (It nearly covers non-activity as well, because it is what we are doing when doing nothing while awake!) If I were addressing a PEN-congress I might try to capture aspects of what goes on in my head when I am writing poetry, but I would like to make clear that I am aware of the fact that I am now addressing computing scientists interested in programming and its teaching: my subject is restricted accordingly.

Thirdly, we should not be deterred by the circumstance that "thinking about thinking" has a markedly incestuous flavour, full of hungry snakes that satisfy their appetite by starting at each other's, or even worse: their own, tails for dinner. Unsatisfactory as this may seem to the orderly mind, we should recognize that this world of our understanding is full of such circularities. Surface tension is explained in terms of the properties of molecules, which are explained in terms of properties of atoms, which are explained in terms of electrons and a nucleus, which is explained in terms of protons and neutrons, in the so-called "drop model" held together by some sort of surface tension. But if the circle is long enough, no one notices it and no one needs to bother: each understands his part of the chain.

In order to disguise the incestuous nature of "thinking about thinking" I shall introduce different names for distinguishable thinking activities.

* * *

I intend to use the (hopefully sufficiently descriptive) term "reasoning" for all manipulations that are formalized --or could readily be so-- by techniques such as arithmetic, formula manipulation or symbolic logic, etc. These techniques have a few common characteristics.

First of all, their application is straightforward in the sense that as soon as it has been decided in sufficient detail, what has to be achieved by them, there is no question anymore how to achieve it. And whenever such a technique has been applied, the question whether this has been done correctly is undebatable.

Secondly --and this is not independent of the first characteristic-- we know how to teach the techniques: arithmetic is taught at primary school, formula manipulation is taught at the secondary school and symbolic logic is taught at the university.

Thirdly, we are very good at doing modest amounts of reasoning. When large amounts of it are needed, however, we are powerless without mechanical aids. To multiply two two-digit numbers is something we all can do, for the multiplication of two five-digit numbers most of us would prefer the assistance of pencil and paper, the multiplication of two hundred-digit numbers is a

task that, even with the aid of pencil and paper, most of us would not care to undertake.

The amount of reasoning that we need when undertaking a non-trivial problem is often our stumbling block and I am therefore tempted to relate the effectiveness with which we have arranged our thoughts to the extent that we have been able to reduce the demands on our reasoning powers. Depending on how we attack a problem --and this is, of course, one of the main points in my whole argument-- the amount of reasoning needed may vary a great deal. Before proceeding, I would like to show you one simple example to drive home that message.

On a flat earth with a constant acceleration of gravity a cannon ball is shot away under a given angle and with a given initial velocity. Ignoring air resistance we may ask the question: at what distance from the cannon will the ball hit the ground again?

I have seen more than one professional mathematician solving this problem by deriving --with or without differential equations-- the equation of the trajectory and then solving it for x after substitution of 0 for y. Compared to the length of their calculations, "to their amount of reasoning", a relatively simple answer was produced.

I have seen schoolboys derive the same answer in a quite different manner. They were familiar with the first principles of mechanics but had the advantage of knowing nothing about differential equations, nor about analytical geometry and parabolas. The argument is then that, because the horizontal speed v_x of the cannon ball is constant (because gravity works vertically) the required answer is

$$t * v_x \quad ,$$

where t is the time that the cannon ball travels. Conservation of energy then requires that the ball that has left the cannon with an upward speed v_y returns to the surface of the earth with a downward speed v_y, i.e. after a change in vertical speed of $2 * v_y$. The acceleration g of gravity

being given, the amount of time needed for that change equals

$$2 * v_y / g \quad .$$

But that was the time that the ball had travelled, and therefore the required distance is

$$\frac{2 * v_x * v_y}{g}$$

And this was the end of their derivation, completed without any computation of the trajectory, the maximum height, etc. (It was typical that all professional mathematicians that produced the latter solution without hesitation had already discovered it in their schooldays!)

Compared with the first solution, the second one is so much simpler that it is worthwhile to try to analyse, how this simplification could be achieved. The first solution uses the parabola that describes the trajectory, i.e. it gives the full connection between the horizontal and the vertical movement. The second solution, however, consists of two isolated arguments, a consideration about the horizontal movement and a consideration about the vertical movement, the two being connected by what we computer scientists would call "a thin interface", viz. the total travelling time. It is an example of a "modularized" argument; it is apparently characterized by concise modules connected via thin interfaces. (It is worth noticing that in our little cannon ball example the interface was expressed in terms of "time", a concept that hardly occurred in the problem statement! Refusal to introduce this concept or --what amounts to the same thing--: its immediate elimination, leads immediately to the timeless aspects of the problem, to the shape of the trajectory, i.e. to the clumsy first solution.)

I have used the computing term "modularization" intentionally. Too often in the past the justification of "modularization" of a large software project has been that it allowed a number of groups to work in parallel on different parts of the same project. And we also know from sad experience that unless the interfaces are indeed sufficiently thin, communication problems become prohibitively severe and that trying to solve these problems by improving the communication facilities between the groups is fighting the symptoms instead of the cause. Here, however, we see in all its clarity a quite different and more fundamental purpose of modularization, viz. to reduce the total

reasoning (irrespective of whether thereafter it will be done by different groups in parallel, or by a single one in succession).

How we choose our "modules" is apparently so critical that we should try to say something about it in general. History can supply us with many illuminating examples, I shall just pick one of them, viz. Galileo's discovery of the laws of the falling body. Since Aristotle, who had observed feathers, leaves and pebbles, heavy bodies fell faster than light ones. It was a first-rate discovery to separate two influences, the pull of gravity and the resistance of the air, and to study the influence of gravity in the hypothetical absence of air resistance, regardless of the fact that, due to the "horror vacui" this absence was not only unrealistic but philosophically unimaginable to the late medieval mind. Galileo could also have chosen the other abstraction, viz. to study the influence of air resistance in the equally unimaginable absence of gravity's pull, but he did not and the reason is quite obvious: he would not have known what to say about it!

The moral of the story is two-fold and, after some consideration, clear. Firstly, whenever an intellectual subuniverse, such as that of Galileo's falling bodies, is created, it is only justified to the extent that you can say something about that subuniverse, about its laws and their consequences. Secondly, the subuniverse must be concise and, therefore, we must have a "thin" interface. Without assumptions, laws or axioms or how you call them, one can say nothing about the subuniverse, so something must be taken into account. But it should be the minimum with the maximum gain, for the more is dragged into the picture, the less concise our reasoning that relies on it: at some stage in the game, some sort of Law of Diminishing Returns comes into action and that is the moment to stop extending the module of our reasoning.

The relevance of this type of considerations to our own trade of programming I can easily illustrate with an example from my own experience, viz. the introduction of the concept of cooperating sequential processes as a way of coming to grips with the problem of operating system design. It is one thing to decide to try to parcel out what should happen under control of an operating system as what happens under control of a set of cooperating se-

quential processes. It is quite another thing to decide to regulate their cooperation without any assumptions about their different speed ratios, not even between such processes for which such speed ratios are known well enough. But the decision is easily justified: only by refusing to take such analogue data into account could we restrict our subuniverse to one for which discrete reasoning was sufficient. The gain was two-fold: firstly, our more general considerations were applicable to a wider class of instances, secondly, the simplicity of our subuniverse made the study of phenomena such as deadlock and individual starvation feasible. Furthermore it showed the great benefits that could be derived from the availability of primitives catering for the so-called "mutual exclusion".

This idea of trying to reduce the demands on our reasoning powers is very close to a philosophical principle that since many centuries is known as "Occam's Razor": if two competing hypotheses explain the same phenomenon, the one that embodies the weaker assumptions should be preferred. (And we must assume that William of Occam knew how to compare the relative weaknesses of competing hypotheses.)

Of one thing we should always be aware. Our reasoning powers get strained by a case-analysis in which we have to distinguish between a great number of different cases. When the number of cases to be distinguished between builds up multiplicatively, they get quickly strained beyond endurance and it is almost always a clear indication that we have separated our concerns insufficiently. (From this point of view it is not surprising that many, who have thought, feel that the technique of the so-called "Decision Tables" is self-defeating. In view of their tendency towards exponential growth, the advice to use them seems hardly justified.)

In the mean time we have encountered a very different kind of thinking activity. Besides the reasoning that actually solves the problem, we have the --hopefully preliminary!-- thinking that reduces that amount of reasoning. Let us call it "pondering", thereby indicating that, ideally, it is done before the actual reasoning starts. It is, however, intended to include as well the "supervision" during the reasoning, the on-going "efficiency control".

The ability to "ponder" successfully is absolutely vital. When we encounter a "brilliant, elegant solution", it strikes us, because the argument, in its austere simplicity, is so shatteringly convincing. And don't think that such a vein of gold was struck by pure luck: the man that found the conclusive argument was someone who knew how to ponder well.

Reasoning, we know, can be taught. Can we also teach "pondering"? We certainly can teach pondering, as long as we do not flatter ourselves with the hope, that all our students will also learn it. But this should not deter us, for in that respect that subject "pondering" is no different from the subject "reasoning", for also for the latter subject holds that some students will never learn it.

Among mathematicians I have encountered much skepticism about the teachability of pondering, but the more I see of the background of that skepticism, the less discouraging it becomes. Sometimes the skepticism is no more than expressing the justified doubt, whether anyone can learn how to ponder well, but as just stated, that need not deter us: let us focus our attention on that part of the population that could learn how to ponder provided that they are taught how to ponder. I see no reason why that part of the population should be empty. Sometimes the skepticism is the result of the inability to form a mental picture of how such pondering lessons would look like, but that inability should not deter us either, for it is so easily explained. Today's mathematical culture suffers from a style of publication, in which the results and the reasoning justifying them are published quite explicitly but in which all the pondering is rigorously suppressed, as if the need to ponder were a vulgar infirmity about which we don't talk in civilized company. (And if the author has not already suppressed it, there is a fair chance that the editor will do so for him!) In the nineteenth century --read Euler, who quite openly and apparently with great delight mentioned all his hunches with what is now a surprising frankness!-- we did not suffer from such a cramped style. And as a result of this fashion to regard explicit reference to one's pondering as "unscientific", many contemporary mathematicians even lack the vocabulary in which they could describe it and this lack makes them very unconscious about their own methodology. Their way of pondering being unknown to themselves, it becomes some-

thing "unknowable" and highly personal, it becomes regarded as a gift with which someone must be "born". And here we find the third source of skepticism: the mere suggestion that pondering could be taught is experienced as a threat upon their ego.

To those who have never tried to visualize how lectures in pondering could look like and, therefore, doubt their feasibility, I can only give one advice. Suppose that you stop teaching results and solutions, but start to solve problems in the lecture room and that you try to be as explicit as possible about your own pondering. What will happen? The need to get some sort of verbal grip on your own pondering will by sheer necessity present your ponderings as something in which, as time progresses, patterns will become distinguishable. But once you have established a language in which to do your own pondering, in which to plan and to supervise your reasoning, you have presented a tool that your students could use as well, for the planning and supervision of their reasoning. In all probability it will have your own personal flavour --I hope that it will, I am tempted to add-- but this does by no means exclude that it won't help some of your students: you would not be the first to found a school of thought! They will learn in your way never to embark unnoticed on an extensive new line of reasoning; then they will learn in your way never to do so without a prior evaluation of the chance of simplification versus the chance of further complication, etc. And, eventually when they grow up , they will learn to ponder in their own way, although the traces of your teaching will probably remain visible all through their lives.

* * *

In the above I have presented "pondering" as an optimization problem, viz. how to get the maximum result out of the minimum amount of reasoning. In doing so I followed a quite respectable tradition that presents certain forms of "laziness" as an indispensable mathematical virtue. This may surprise all those who know my profound distrust regarding the simplifications that underly the "Homo Economicus" as a picture of Man, a picture in which the self-sacrifying person (if admitted as a human being at all!) is classified as a non-interesting fool. To correct a wrong impression I may have made, let me quantify my considerations. If you would like to think effectively (because you like doing so), then you had better try to become an expert at

it! But, if you would prefer to dream (because you like having beautiful dreams), then you have better become an expert at dreaming! (Remarks to which we can add, that the two facilities do not exclude each other.)

There is a third, indispensable mental activity which, for lack of a better name, I indicate with "revolting". This is what has to take place when, no matter what we try, our maximized output remains nil, i.e. when we are stuck in an impossible task. The only effective reactions are either to change the problem until it becomes manageable, or to throw it away and to turn to something else. Obvious as this may seem, I must mention it explicitly, because experience has shown that these are difficult decisions to consider, and that, therefore, we must teach the courage to revolt.

The decision is always difficult because it has the connotation of failure --usually without justification, for most tasks are impossible anyhow--, it is, however, particularly difficult if the impossibility of the task is politically impalatable to the society of which one is a member. In serious cases the victim who has embarked on a popular, but impossible task, can hardly afford to admit even to himself its impossibility and the resulting inner conflicts may form a serious threat to one's integrity, one's health and one's sanity. For those who doubt their courage to revolt, it seems a safe precaution to stay away from popular projects, particularly if they work in an intolerant environment. For those who have learned to revolt well, the act is always one of liberation: it is an ability shared by most innovators of science.

13th February 1975
NUENEN

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

CHAPTER 1.: CONCURRENCY

C. A. R. Hoare

The Queen's University of Belfast
Belfast, Northern Ireland

Parallel Programming: An Axiomatic Approach

Summary

This paper develops some ideas expounded in [1]. It distinguishes a number of ways of using parallelism, including disjoint processes, competition, cooperation, communication and "colluding". In each case an axiomatic proof rule is given. Some light is thrown on traps or ON conditions. Warning: the program structuring methods described here are not suitable for the construction of operating systems.

Work on this paper has been supported in part by ARPA under contract SD-183 and NSF under contract GJ-36473X.
The views expressed are those of the author.

1. Introduction

A previous paper [1] summarizes the objectives and criteria for the design of a parallel programming feature for a high level programming language. It gives an axiomatic proof rule which is suitable for disjoint and competing processes, but seems to be inadequate for cooperating processes. Its proposal of the "conditional critical region" also seems to be inferior to the more structured concept of the class [2] or monitor [3]. This paper introduces a slightly stronger proof rule, suitable for cooperating and even communicating processes. It suggests that the declaration is a better way of dealing with competition than the resource. It then defines a slightly different form of parallelism more suitable for non-deterministic algorithms, and finally adapts it to deal with the vexed problem of machine traps.

2. Concepts and Notations

We shall use the notation [1]

$$Q_1 // Q_2$$

to denote a parallel program consisting of two processes Q_1 and Q_2 which are intended to be executed "in parallel". The program $Q_1 // Q_2$ is defined to terminate only if and when both Q_1 and Q_2 have terminated.

The notation

$$P\{Q\}R$$

asserts that if a propositional formula P is true of the program variables before starting execution of the program statement Q, then

the propositional formula R will be true on termination of Q , if it ever terminates. If not, $P\{Q\}R$ is vacuously true.

The notation

$$Q_1 \sqsubseteq Q_2$$

asserts that the program statements Q_1 and Q_2 have identical effects under all circumstances on all program variables, provided that Q_1 terminates. The notation $Q_1 \equiv Q_2$ means $Q_1 \sqsubseteq Q_2$ & $Q_2 \sqsubseteq Q_1$, i.e., they terminate together, and have identical effects when they do. The theory and logic of the $\sqsubseteq$ relation are taken from Scott [4].

The notation

$$\frac{A \quad B}{C}$$

denotes a proof rule which permits the deduction of C whenever theorems of the form A and B have been deduced.

The notations for assignment $(x := e)$ and composition of statements $(Q_1;Q_2)$ have the same meaning as in ALGOL 60, but side-effects of function evaluation are excluded.

As examples of proof rules whose validity follows fairly directly from these definitions we give:

$$\frac{P\{Q_1\}S \qquad S\{Q_2\}R}{P\{Q_1;Q_2\}R}$$ Rule of Composition

$$\frac{Q_1 \sqsubseteq Q_2 \qquad P\{Q_2\}R}{P\{Q_1\}R}$$ Rule of Containment

We will use the word "process" to denote a part of a program intended to be executed in parallel with some other part; and use the phrase "parallel program" to denote a program which contains or consists of two or more processes. In this paper we will talk in terms of only

two processes; however all results generalize readily to more than two.

3. Disjoint Processes

Our initial method of investigation will be to enquire under what circumstances the execution of the parallel program $Q_1 // Q_2$ can be guaranteed to be equivalent to the sequential program $Q_1 ; Q_2$. Preferably these circumstances should be checkable by a purely syntactic method, so that the checks can be carried out by a compiler for a high level language.

The most obvious case where parallel and serial execution are equivalent is when two processes operate on disjoint data spaces, in the same way as jobs submitted by separate users to a multiprogramming system. Within a single program, it is permissible to allow each process to access values of common data, provided none of them update it. In order to ensure that this can be checked at compile time, it is necessary to design a language with the decent property that the set of variables subject to change in any part of the program is determinable merely by scanning that part. Of course, assignment to a component of a structured variable must be regarded as changing the whole variable, and variables assigned in conditionals are regarded as changed, whether that branch of the conditional is executed or not.

Given a suitable syntactic definition of disjointness, we can formulate the proof rule for parallel programs in the same way as that for sequential ones:

$$\frac{P\{Q_1\}S \qquad S\{Q_2\}R}{P\{Q_1 /\!/ Q_2\}R}$$

Asymmetric Parallel Rule

provided that Q_1 and Q_2 are disjoint.

The proof of this (if proof it needs) may be based on the commutivity of the basic units of action performed in the execution of Q_1 and Q_2. Consider an arbitrary assignment $x_1 := e_1$ contained in Q_1 and an arbitrary assignment $x_2 := e_2$ contained in Q_2. Since Q_1 and Q_2 are disjoint, e_2 does not contain x_1 and e_1 does not contain x_2. The values of expressions are independent of the values of the variables they do not contain, and consequently they are unaffected by assignment to those variables. It follows that:

$$(x_1 := e_1 ; x_2 := e_2) \equiv (x_2 := e_2 ; x_1 := e_1) ,$$

i.e., these units of actions commute.

Consider now any interleaving of units of action of Q_1 and Q_2. If any action of Q_2 precedes any action of Q_1, the commutivity principle (together with substitution of equivalents) may be used to change their order, without changing the total effect. Provided both Q_1 and Q_2 terminate, this interchange may be repeated until all actions of Q_1 precede all actions of Q_2. But this extreme case is just the effect of executing the whole of Q_1 followed by the whole of Q_2. If one or both of Q_1 and Q_2 fails to terminate, then both $Q_1;Q_2$ and $Q_1 /\!/ Q_2$ equally fail to terminate.

Thus we have proved that

$$Q_1 /\!/ Q_2 \equiv Q_1;Q_2$$

and consequently their correctness may be proved by the same proof rule.

Of course, this justification is still very informal, since it is based on the assumption that parallel execution is equivalent to an arbitrary interleaving of "units of action". It assumes, for example, that two "simultaneous" accesses of the same variable will not interfere with each other, as they might if one access got hold of half the variable and the other got hold of the other half. Such ridiculous effects are in practice excluded by the hardware of the computer or store. On a multiprocessor installation the design of the store module ensures that two accesses to the same (in practice, even neighboring) variables will exclude each other in time, so that even if requests arrive "simultaneously", one of them will be completed before the other starts. This concept of exclusion together with commutivity will assume greater importance in what follows.

In [1] the proof rule for disjoint processes was given in the more symmetric form:

$$\frac{P_1\{Q_1\}R_1 \qquad P_2\{Q_2\}R_2}{P_1 \,\&\, P_2\{Q_1 /\!/ Q_2\}R_1 \,\&\, R_2}$$

Symmetric Parallel Rule

provided that P_1, Q_1, R_1 are disjoint from P_2, Q_2, R_2. This proof rule may be simpler to use for systematic or automatic program construction than the asymmetric rule given above, in cases where the desired result of a program is of the form $R_1 \,\&\, R_2$, and the program is not intended to change any variable common to R_1 and R_2. The symmetric form of the rule can be derived from the asymmetric form, by showing that every proof using one could also have used the other. Assume $P_1\{Q_1\}R_1$ and $P_2\{Q_2\}R_2$ have been proved. The disjointness of R_1 and Q_2 and the disjointness of P_2 and Q_1 ensure the truth of $P_2\{Q_1\}P_2$ and $R_1\{Q_2\}R_1$; hence

$$P_1 \,\&\, P_2\{Q_1\}R_1 \,\&\, P_2$$

and $$R_1 \,\&\, P_2\{Q_2\}R_1 \,\&\, R_2 \quad .$$

One application of the asymmetric parallel rule gives:

$$P_1 \,\&\, P_2\{Q_1 /\!/ Q_2\}R_1 \,\&\, R_2$$

which is the same conclusion as the symmetric rule.

In [1] it was shown that disjoint parallelism permits the programmer to specify an overlap between input/output operations and computation, which is probably the main benefit which parallelism can offer the applications programmer. In contrast to other language proposals, it does so in a secure way, giving the user absolute compile-time protection against time-dependent errors.

4. Competing Processes

We shall now explore a number of reasons why the rule of disjointness may be found unacceptably restrictive, and show in each case how the restriction can be safely overcome.

One important reason may be that two processes each require occasional access to some limited resource such as a line-printer or an on-line device for communication with the programmer or user. In fact, even mainstore for temporary working variables may be a limited resource: certainly an individual word of mainstore can be allocated as local workspace to only one process at a time, but may be reallocated (when that process has finished with it) to some other process that needs it.

The normal mechanism in a sequential programming language for making a temporary claim on storage during execution of a block of program is

the declaration. One of the great advantages of the declaration is that the scope of use of a variable is made manifest to the reader and writer; and furthermore, the compiler can make a compile-time check that the variable is never used at a time when it is not allocated. This suggests that the declaration would be a very suitable notation by which a parallel process may express the acquisition and relinquishment of other resources, such as lineprinters. After all, a lineprinter may be regarded as a data structure (largely implemented in hardware) on which certain operations (e.g., print a line) are defined to be available to the programmer. More accurately, the concept of a line printer may be regarded as a type or class of variable, new instances of which can be "created" (i.e., claimed) and named by means of declaration, e.g., using the notation of PASCAL [14]:

```
begin managementreport: lineprinter; ...
```

The individual operations on this variable may be denoted by the notations of [2]:

```
managementreport.output(itemline);
```

which is called from within the block in which the managementreport is declared, and which has the effect of outputing the value of "itemline" to the lineprinter allocated to managementreport.

This proposal has a number of related advantages:

(1) The normal scope rules ensure that no programmer will use a resource without claiming it, --

(2) Or forget to release it when he has finished with it.

(3) The same proof rule for declarations (given in [7]) may be used for parallel processes.

(4) The programmer may abstract from the number of items of resource actually available.

(5) If the implementer has available several disjoint items of a resource (e.g. two line printers), they may be allocated simultaneously to several processes within the same program.

These last three advantages are not achieved by the proposal in [1].

There are also two disadvantages:

(1) Resource constraints may cause deadlock, which an implementation should try to avoid by compile-time and/or run-time techniques [1,5]. The proposal here gives no means by which a programmer can assist in this.

(2) The scope rules for blocks ensure that resources are released in exactly the reverse order to that in which they are acquired. It is sometimes possible to secure greater efficiency by relaxing this constraint.

Both these disadvantages may reduce the amount of parallelism achievable in circumstances where the demand on resources is close to the limit of their availability. But of course they can never affect the logical correctness of the programs.

It is worthy of note that the validity of sharing a resource between two processes, provided that they are not using it at the same time, also depends on the principle of commutivity of units of action. In this case, the entire block within which a resource is claimed and used must be regarded as a single unit of action, and must not be interleaved with execution of any other block to which the same resource is allocated. The programmer presumably does not mind which of these

two blocks is executed first; for example, he does not mind which of the two files is output first on the lineprinter, because he is interested in them only after they have been separated by the operator. Thus as far as he is concerned, the two blocks commute as units of action; of course he could not tolerate arbitrary interleaving of lines from the two files.

5. Cooperating Processes

Hitherto, parallel programming has been confined to disjoint and competing processes, which can be guaranteed by a compile-time check to operate on disjoint data spaces. The reason for insisting on disjointness is that this is an easy way for the compiler to check that the units of action of each process will commute. In the next two sections we shall investigate the effects of relaxing this restriction, at the cost of placing upon the programmer the responsibility of proving that the units of action commute. Processes which update one or more common variables by commutative operations are said to cooperate.

One consequence of the commutivity requirement is that neither process can access the value of the shared variable, because this value will in general be different whether it is taken before or after updating by the other process. Furthermore, the updating of a shared variable must be regarded as a single unit of action, which occurs either wholly before or wholly after another such updating. For these reasons, the use of normal assignment for updating a variable seems a bit misleading, and it seems better to introduce the kind of notation

used in [6], for example:

n :+ 1 in place of n := n+1 .

One useful commutative operation which may be invoked on a shared set is that which adds members to that set, i.e., set union:

s :∪ t (s := s ∪ t) ,

since evidently s :∪ t ; s :∪ t' ≡ s :∪ t' ; s :∪ t for all values of t and t' . A similar commutative operation is set subtraction:

s :- t .

As an example of the use of this, consider the primefinding algorithm known as the sieve of Eratosthenes. An abstract parallel version of this algorithm may be written using traditional set notations:

```
sieve := {i | 2 ≤ i ≤ N};
p1 := 2; p2 := 3;
while p1^2 ≤ N do
   begin {remove multiples of (p1) // remove multiples of (p2)};
      if p2^2 < N then p1 := min{i | i > p2 & i ∈ sieve}
                  else p1 := p2;
      if p1^2 < N then p2 := min{i | i > p1 & i ∈ sieve}
   end;
```

The validity of the parallelism can be assured if the only operation on the sieve performed by the procedure "remove multiples of (p) " is set subtraction:

```
procedure remove multiples of (p: 2..N);
   begin i: 2..N;
      for i := p^2 step p until N do sieve :- {i}
   end;
```

Of course, when a variable is a large data structure, as in the example given above, the apparently atomic operations upon it may in practice require many actual atomic machine operations. In this case an implementation must ensure that these machine operations are not interleaved with some other operation on that same variable. A part of a program which must not be interleaved with itself or with some other part is known as a critical region [5]. The notational structure suggested in [2] seems to be a good one for specifying updating operations on variables, whether they are shared or not; and the proof rules in the two cases are identical. The need to set up an exclusion mechanism for a shared variable supports the suggestion of Brinch Hansen [9] that the possibility of sharing should be mentioned when the variable is declared.

It is worthy of note that the validity of a parallel algorithm depends only on the fact that the abstract operations on the structured variable commute. The actual effects on the concrete representation of that variable may possibly depend on the order of execution, and therefore be non-deterministic. In some sense, the operation of separating two files of line printer paper is an abstraction function, i.e., a many-one function mapping an ordered pair onto a set. Abstraction may prove to be a very important method of controlling the complexity of parallel algorithms.

In [1] it was suggested that operations on a shared variable s should be expressed by the notation

with s do Q ,

where Q was to be implemented as a critical region, so that its execution would exclude in time the execution of any other critical region with the same variable s . But the present proposal is distinctly superior:

(1) It uses the same notations and proof rules as sequential programs;

(2) It recognizes the important rôle of abstraction.

(3) The intended effect of the operation as a unit of action is made more explicit by the notation.

(4) The scope rules make deadlock logically impossible.

Finally, the proof rule given in [1] is quite inadequate to prove cooperation in achieving any goal (other than preservation of an invariant)

A useful special case of cooperation between parallel processes which satisfies the commutivity principle is the use of the "memo function" suggested by Michie [10]. Suppose there are certain values which may or may not be needed by either or both processes, and each value requires some lengthy calculation to determine. It would be wasteful to compute all the values in advance, because it is not known in advance which of them will be needed. However, if the calculation is invoked from one of the cooperating processes, it would be wasteful to throw the result away, because it might well be needed by the other process. Consequently, it may pay to allocate a variable (e.g. an array A) in advance to hold the values in question, and set it initially to some null value. The function which computes the desired result is now adapted to first look at the relevant element of A . If this is not null, the function immediately returns its value without further computation. If not, the function computes the result and stores it in the variable. The proof of the correctness of such a technique is based on the invariance of some such assertion as:

$$\forall i(A[i] \neq \text{null} \supset A[i] = f(i)) \quad ,$$

where A is the array (possibly sparse) in which the results are stored,

and f is the desired function. The updating of the array A must be a single unit of action; the calculation of the function f may, of course, be reentrant. This technique of memo functions may also be used to convey results of processes which terminate at an arbitrary point (see Section 7).

6. Communicating Programs

The commutivity principle, which lies at the basis of the treatment of the preceding sections, effectively precludes all possibility of communication between processes, for the following reason. The method that was used in Section 3 to prove

$$Q_1 // Q_2 \equiv Q_1 ; Q_2$$

can also be used to prove

$$Q_1 // Q_2 \equiv Q_2 // Q_1 \quad .$$

It follows that a legitimate implementation of "parallelism" would be to execute the whole of Q_1 and then the whole of Q_2, or to do exactly the reverse. But if there were any communication between Q_1 and Q_2, this would not be possible, since it would violate the principle that a communication cannot be received before it has been sent.

In order to permit communication between Q_1 and Q_2 it is necessary to relax the principle of commutivity in such a way that complete execution of Q_2 before starting Q_1 is no longer possible. Consider an arbitrary unit of action q_1 of Q_1, and an arbitrary unit of action q_2 of Q_2. We say that q_1 and q_2 <u>semicommute</u> if:

$$q_2 ; q_1 \sqsubseteq q_1 ; q_2 \quad .$$

If all q_1 and q_2 semicommute, we say that Q_1 and Q_2 are communicating processes, and that Q_1 is the producer process, and Q_2 is the consumer [5].

The effect of semicommutivity is that some interleavings of units of action may be undefined; but moving actions of Q_2 after actions of Q_1 will never give a different result or make the interleaving less well defined; consequently the execution of the whole of Q_1 before starting Q_2 is still a feasible implementation, in fact the one that is most defined:

$$Q_1 // Q_2 \sqsubseteq Q_1 ; Q_2$$

Thus it is still justified to use the same proof rule for parallel as for sequential programs.

If assertional proof methods are used to define a programming language feature, it is reasonable to place upon an implementor the injunction to bring a program to a successful conclusion whenever it is logically feasible to do so (or there is a good engineering reason not to, e.g., integer overflow; and it is not logically possible to terminate a program of which "false" is provably true on termination). In the case of communicating programs, termination can be achieved by simply delaying an action of Q_2 where necessary until Q_1 has performed such actions as make it defined, which will always occur provided $Q_1 ; Q_2$ terminates.

The paradigm case of semicommutative operations are input and output of items to a sequence. Output of an item x to sequence s will be denoted:

```
s.output(x);
```

it is equivalent to

$s := s \cap \langle x \rangle$;

where $\cap$ is the symbol of concatenation, and $\langle x \rangle$ is the sequence whose only item is x. This operation appends the item x to the end of the sequence and is always defined. Input of the first item from a sequence s to the variable y will be denoted:

s.input(y)

which is equivalent to a unit of action consisting of two operations:

y := first(s); s := rest(s);

where first maps a sequence onto its first item and rest maps a sequence onto a shorter sequence, namely the sequence with its first item removed. The removal of an item from an empty sequence is obviously undefined; on a non-empty sequence it is always defined. A sequence to which an item has just been output is never empty. Hence

s.input(y) ; s.output(x) $\sqsubseteq$ s.output(x) ; s.input(y)

i.e., these operations semicommute. Consequently a sequence may be used to communicate between two processes, provided that the first only performs output and the second only performs input. If the second process tries to input too much, their parallel execution does not terminate; but neither would their sequential execution. Processes communicating by means of a sequence were called coroutines by Conway [11], who pointed out the equivalence between sequential and parallel execution.

In practice, for reasons of economy, the potentially infinite sequence used for communication is often replaced by a bounded buffer, with sufficient space to accommodate only a few items. In this case, the operation of output will have to be delayed when the buffer is full, until input has created space for a new item. Furthermore the program

may fail to terminate if the number of items output exceeds the number of items input by more than the size of the buffer. And finally, since either process may have to wait for the other, purely sequential execution is in general no longer possible, because it would not terminate if the total length of the output sequence is larger than the buffer (which it usually is). Thus the parallel program is actually more defined than the corresponding sequential one, which may seem to invalidate our proof methods.

The solution to this problem is to consider the relationship between the abstract program using an unbounded sequence and the concrete program using a bounded buffer representation for the sequence. In this case, the concrete program is the same as the abstract one in all respects except that it contains an operation of concrete output (to the buffer) whenever the abstract program contains abstract output (to the sequence), and similarly for input. Concrete output always has the same effect as abstract output when it is defined, but is sometimes undefined (when the buffer is full), i.e.:

$$\text{concrete output} \sqsubseteq \text{abstract output} \quad .$$

The replacement of an operation by a less well defined one can never change the result of a program (by the principle of continuity [4]), so the concrete program is still contained in the abstract one

$$\text{concrete} \sqsubseteq \text{abstract} \quad .$$

This justifies the use of the same proof rule for the concrete as for the abstract program. The abstract sequence plays the role of the "mythical" variables used by Clint [12]; here again, abstraction proves to be a vital programming tool.

In order to implement a concrete data representation for a variable which is being used to communicate between processes, it is necessary to have some facility for causing a process to "wait" when it is about to perform an operation which is undefined on the abstract data or impossible on its current representation. Furthermore, there must be some method for "signalling" to wake up a waiting process. One method of achieving this is the condition variable described in [3]. Of course, if either process of the concrete program can wait for the other, it is possible for the program to reach deadlock, when both processes are waiting. In this case it is not reasonable to ask the implementor to find a way out of the deadlock, since it would involve a combinatorial investigation, where each trial could involve backtracking the program to an earlier point in its execution. It is therefore the programmer's responsibility to avoid deadlock. The assertional proof methods given here cannot be used to prove absence of deadlock, which is a form of non-termination peculiar to parallel programs.

A natural generalization of one-way communication is two-way communication, whereby one process Q_1 uses a variable s_1 to communicate to Q_2, and Q_2 uses a variable s_2 to communicate with Q_1. Communication is achieved, as before, by semicommutative operations. It is now impossible to execute Q_1 and Q_2 sequentially in either order; and it is plain that the proof rule should be symmetric. Furthermore, the correctness of Q_1 may depend on some property S_2 of s_2 which Q_2 must make true, and similarly, Q_2 may need to assume some property S_1 of s_1 which Q_1 must make true. Hence we derive the rule:

$$\frac{P_1 \,\&\, S_2\{Q_1\}S_1 \,\&\, R_1 \qquad P_2 \,\&\, S_1\{Q_2\}S_2 \,\&\, R_2}{P_1 \,\&\, P_2\{Q_1 \,//\, Q_2\}R_1 \,\&\, R_2}$$

Rule of two-way communication

where P_1, Q_1, R_1, S_1 are disjoint from P_2, Q_2, R_2, S_2 except for variables s_1, s_2, which are subject only to semicommutative operations in Q_1 and Q_2, as explained above; and P_1, S_1, R_2 may contain s_1 (but not s_2) and P_2, S_2, R_1 may contain s_2 (but not s_1). The informal proof of this is complex, and is included in an appendix.

7. Colluding Processes

In certain combinatorial and heuristic applications, it can be difficult for the programmer to know which of two strategies is going to be successful; and an unsuccessful strategy could run forever, or at least take an uncontrollable or unacceptable length of time. For example, a theorem-checker might attempt to find a proof and a counter-example in parallel, knowing that if one attempt succeeds, the other may not terminate. In such cases, Floyd [13] has suggested the use of "non-deterministic" algorithms: both strategies are executed in parallel, until one of them succeeds; the other is then discontinued. In principle, this can be very wasteful, since all effort expended on the unsuccessful strategy is wasted, unless it has cooperated in some way with the successful one. Processes which implement alternative strategies we will call colluding.

Colluding processes require a completely new notation and proof rule, representing the fact that only one of them has to terminate.

These will be taken from Lauer [8], who uses the form

$$Q_1 \underline{\text{ or }} Q_2$$

to denote a program involving execution of either Q_1 or Q_2, where the programmer either does not know or care which one is selected. The proof rule is adapted from the symmetric rule for disjoint processes:

$$\frac{P_1\{Q_1\}R_1 \qquad\qquad P_2\{Q_2\}R_2}{P_1 \& P_2\{Q_1 \underline{\text{ or }} Q_2\}R_1 \vee R_2}$$

where P_1, Q_1, R_1 are disjoint from P_2, Q_2, R_2.

Note the continued insistence on disjointness, which was not made in [8]. This has the advantage of permitting a genuine parallel implementation. It has the even greater advantage that it does not require an implementation to undo (backtrack) the effects of the unsuccessful process. For suppose Q_1 was successful, and therefore R_1 is true on completion of the program. R_1 does not mention any variable changed by Q_2, so the programmer cannot know anything of the values or properties of these variables at this point; and so the fact that Q_2 has changed these values does not matter. However the values are not formally undefined -- for example, they can still be printed out. Furthermore, if Q_2 has used something like the memo function technique described in Section 5, it is possible to use the results of its calculations, even after it has been terminated at an arbitrary point in its execution.

However, it must not be a wholly arbitrary point; a process must not be stopped in the middle of one of its "units of action", i.e., in the middle of updating a structured variable non-local to the process. If it were so stopped, the invariant of the data structure might no longer

be true, and any subsequent attempt to access that variable would be disastrous. The need to inhibit termination during certain periods was recognized by Ashcroft and Manna [15].

Sometimes a colluding process can detect that it will never succeed, and might as well give up immediately, releasing its resources, and using no more processor time. To do this, Floyd suggested a basic operation

failure;

the proof rule for this may be simply modelled on that for the jump:

true {failure} false

which permits failure to be invoked in any circumstances (true), and which states that failure always fails to terminate. If all processes fail, the program fails, and no property of that program will hold after the failure. The situation is the same as that of a sequential program, artificially interrupted by expiry of some time limit.

In order to ensure that time is not excessively wasted on an unsuccessful process, the programmer should exert every endeavor to ensure that a process usually detects whether it is going to fail as early as possible. However, it may be that a process sometimes discovers that although failure is quite likely, it is not yet certain, and it may take a longer time to decide than was originally hoped. In this case, it would be wise to delay continuation of the current process but without precluding the possibility of later continuation. To achieve this, I suggest a primitive scheduling statement:

wait;

this is intended to cause immediate suspension of the calling process,

allowing the processor to concentrate attention on the other processes, until either

(1) one of them succeeds: the waiting process is then abandoned in the normal way;

(2) all of them fail: the waiting process is then resumed in the normal way as the last remaining hope;

(3) all non-failed processes have themselves invoked a wait: then the longest waiting process is resumed.

(If several processors are available, the above remarks require adaptation.)

If greater sophistication in scheduling is desired, a process which is exceptionally unpromising should indicate this fact by passing a parameter to the wait:

```
wait(t);
```

where t is an indication of how many times the calling process is willing to be overtaken by more promising processes. The implementation of this is accomplished most easily by maintaining a pseudo-parallel time queue, as in SIMULA. For wise scheduling, t should be proportional to an estimate of the expense required by the current process before it comes to a decision on its own success or failure. Of course, a process should try to avoid waiting while it is in possession of expensive resources. Since every process retains some allocation of storage and overhead during a wait, waiting should be used sparingly. Nevertheless, it gives the programmer a useful degree of control in specifying a "breadth first" or "depth first" search of a tree of alternatives.

It hardly seems worthwhile to seek more sophisticated scheduling methods for colluding processes. One great advantage of the wait is

that each process can schedule itself at a time when its resource occupation is low; furthermore it can do so successfully without knowing anything about the purpose, logic, progress, or even the name of any other process. This is the secret of successful structuring of a large program, and suggests that self-scheduling by a wait is a good programming language feature, and surely preferable to any feature which permits one process to preempt or otherwise schedule another at an arbitrary point in its progress.

But perhaps the strongest argument in favor of a wait is that the insertion of a wait has no effect whatsoever on the logic of a process, and in a proof of correctness it may be ignored. It is equivalent to an empty statement, and has the delightful proof rule:

$$R\{\text{wait}(t)\}R$$

for any assertion R .

On completion of the program Q_1 or Q_2 , it can be quite difficult to find out which of them has in fact succeeded. Suppose, for example, the purpose of the program is to find a z satisfying $R(z)$. Suppose processes Q_1 and Q_2 satisfy

$$P_1\{Q_1\}R(y_1)$$

$$P_2\{Q_2\}R(y_2) \quad .$$

It is now possible to prove

$$P_1 \,\&\, P_2\{(Q_1 \text{ or } Q_2);\ \text{if } R(y_1) \text{ then } z := y_1 \text{ else } z := y_2\}R(z) \quad .$$

But $R(y_1)$ may be expensive or impossible to compute, and something better is required. A possible solution is based on the "protected tail" described in [15]. In this, a colluding process has two parts

$$Q \text{ then } Q'$$

where Q is the part that may fail to terminate, and Q' is initiated only when Q has terminated. However all parallel colluding processes are stopped before Q' starts. That is why Q' has the name "protected tail". Since a protected tail is never executed in parallel, the rule of disjointness may be somewhat relaxed, permitting the protected tails to update the same variables, e.g.:

$$Q_1 \text{ then } z := y_1 \text{ or } Q_2 \text{ then } z := y_2 \quad .$$

The appropriate proof rule is:

$$\frac{P_1\{Q_1\}R_1 \qquad R_1\{Q_1'\}R \qquad P_2\{Q_2\}R_2 \qquad R_2\{Q_2'\}R}{P_1 \& P_2\{Q_1 \text{ then } Q_1' \text{ or } Q_2 \text{ then } Q_2'\}R}$$

where P_1 , Q_1 , R_1 , are disjoint from P_2 , Q_2 , R_2 .

The construction Q_1 or Q_2 is something like the least upper bound [4] of two functions $f_1 \sqcup f_2$. However $f_1 \sqcup f_2$ is inconsistent if f_1 and f_2 both terminate and have different results; and it is not possible to guarantee against this inconsistency either by a compile time or a run time check (which could go on forever if the functions are consistent). The or construction is still well-defined (at least axiomatically), in spite of the fact that the effects of Q_1 and Q_2 are nearly always different.

8. Machine Traps

Dijkstra has expressed the view [16] that one of the main values of parallel programming ideas is the light that they shed on sequential programming. This section suggests that the idea and proof method for

colluding programs may be used to deal with the problem of machine traps that arise when a machine cannot perform a required operation due to overflow or underflow, and either stops the program or jumps to some trap routine specified by the programmer. At first sight such a jump seems to be even more undisciplined than a go to statement invoked by the program, since even the source of the jump is not explicit. But the main feature of such a jump is that it signals failure of the machine to complete the operations specified by the program Q_1 ; if the programmer is willing to supply some alternative "easier" but less satisfactory program Q_2 , the machine will execute this one instead, just as in the case of colluding processes.

However, there are two great differences between this case and the previous one.

(1) The programmer would very much rather complete Q_1 than Q_2 .

(2) Parallel execution of Q_1 and Q_2 is not called for. Q_2 is invoked only when Q_1 explicitly fails.

For these reasons it would be better to introduce a different notation, to express the asymmetry:

$$Q_1 \text{ otherwise } Q_2$$

Also, because parallelism is avoided, the rule of disjointness can be relaxed considerably:

$$\frac{P_1\{Q_1\}R_1 \qquad\qquad P_2\{Q_2\}R_2}{P_1 \,\&\, P_2\{Q_1 \text{ otherwise } Q_2\}R_1 \vee R_2}$$

where Q_1 is disjoint from P_2 ; this states that Q_2 may not assume anything about the variables changed by Q_1 . However Q_2 is still

allowed to print out these variables, or take advantage of any memo functions computed.

It is necessary to emphasize again the impermissibility of stopping in the middle of an operation on a variable non-local to a process. If failure occurs or is invoked in the middle of such an operation, it is the smallest process lexicographically enclosing the variable that must fail. This can be assured by the normal scope rules, provided that the critical regions are declared local to the variable, as in monitors and data representations, rather than being scattered through the program which uses them, as in [1].

This proposal provides the programmer with much of the useful part of the complex PL/I [17] ON-condition and prefix mechanisms. The other intended use of the ON-condition is to extend machine arithmetic by supplying programmer-defined results for overflowing operations. For this I would prefer completely different notations and methods.

This proposal also provides the programmer with a method for dealing with transient or localized failure of hardware at run time, or even (dare I mention it?) with programming error. The need for a means to control such failures has been expressed by d'Agapeyeff [18].

9. Conclusion

In conclusion it is worth while to point out that the parallel composition of programs has pleasant formal properties, namely // and or are associative and commutative, with fixed point "do nothing" and "failure" respectively; and otherwise is associative with fixed point "failure". These facts are expressed by the equivalences:

$Q_1 // Q_2 \equiv Q_2 // Q_1$

Q_1 or $Q_2 \equiv Q_2$ or Q_1

$(Q_1 // Q_2) // Q_3 \equiv Q_1 // (Q_2 // Q_3)$

$(Q_1$ or $Q_2)$ or $Q_3 \equiv Q_1$ or $(Q_2$ or $Q_3)$

$(Q_1$ otherwise $Q_2)$ otherwise $Q_3 \equiv Q_1$ otherwise $(Q_2$ otherwise $Q_3)$

$(Q //$ do-nothing$) \equiv Q$

Q or failure $\equiv Q$

failure otherwise $Q \equiv Q$ otherwise failure $\equiv Q$.

References

[1] C. A. R. Hoare. "Towards a Theory of Parallel Programming," in Operating Systems Techniques, ed. C. A. R. Hoare and R. H. Perrot. Academic Press, 1972.

[2] C. A. R. Hoare. "Proof of Correctness of Data Representations," Acta Informatica 1, 271-281 (1972).

[3] C. A. R. Hoare. "Monitors: an Operating System Structuring Concept." Seminar delivered to I.R.I.A., May 11, 1973.

[4] D. Scott. "Outline of a Mathematical Theory of Computation," PRG-7. Programming Research Group, Oxford University.

[5] E. W. Dijkstra. "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys. Academic Press, 1968.

[6] C. A. R. Hoare. "Notes on Data Structuring," in Structured Programming, by E. W. Dijkstra, O. J. Dahl, C. A. R. Hoare. Academic Press, 1972.

[7] C. A. R. Hoare. "Procedures and Parameters: an Axiomatic Approach," in Symposium on Semantics of Algorithmic Languages, ed. E. Engeler. Springer-Verlag, 1972.

[8] P. E. Lauer. "Consistent Formal Theories of the Semantics of Programming Languages," Ph.D. thesis, Queen's University, Belfast. TR.25.121 IBM Laboratory, Vienna, Nov. 1971.

[9] P. Brinch Hansen. Operating System Principles. Prentice-Hall, 1973.

[10] D. Michie. "Memo functions: a language feature with 'rote learning' properties," MIP-R-29, Edinburgh University, (November 1967).

[11] M. E. Conway. "Design of a Separable Transition Diagram Compiler," Comm. ACM 6, 396-408, (1963).

[12] M. Clint. "Program Proving: Coroutines," Acta Informatica 2, 50-63, (1973).

[13] R. W. Floyd. "Nondeterministic Algorithms," J.ACM 14, 4, pp. 636-644, (1967).

[14] N. Wirth. "The Programming Language PASCAL," Acta Informatica 1, 1 (1971), pp. 35-63.

[15] E. A. Ashcroft, Z. Manna. "Formalization of Properties of Parallel Programs," A.I.M. 110, Stanford University, February 1970

[16] E. W. Dijkstra. Private communication.

[17] Formal Definition of PL/I. IBM Laboratory, Vienna, TR.25.071 (1967).

[18] A. d'Agapeyeff. Private communication.

Appendix: Proof of rule of two-way communication.

The informal proof of this depends on a mythical reordering of units of action, where a unit of action is defined as an assignment of a constant to a variable, or the performance of an operation with constant parameters to a variable. Thus for example, an operation in Q_1

s_2.input(y);

would appear every time in a computation of Q_1 as

y := 17;

s_2.truncate;

where 17 happens to be the value of the first item of s_2 at the time, and the "truncate" operator removes the first item from a sequence.

Consider a particular interleaved execution of $Q_1 // Q_2$. Sort the computation into the order

$$E_{21} ; E_1 ; E_{22} \quad ,$$

where E_{21} is the sequence of all operations of Q_2 on s_2 ,

E_1 is the sequence of all operations of Q_1 ,

E_{22} is the sequence of all other operations of Q_2 .

This is feasible, because operations on one variable commute with operations on all other variables, and operations of Q_2 on s_2 semicommute with operations of Q_1 on s_2 , so the rearranged sequence can only be more defined than the original interleaving.

Define

$\bar{P}_2$ as the result of replacing all occurrences of s_2 in P_2 by the initial value of s_2 ,

and $\bar{S}_2$ as the result of replacing in S_2 all occurrences of variables changed by Q_2 by their final values, i.e., after executing E_{22}.

We will assume that the premises of the rule are valid, and hence we assert informally (i.e., not by showing it to be deducible):

$$P_1 \& S_2\{E_1\}S_1 \& R_1 \tag{1}$$

$$P_2 \& S_1\{E_{21};E_{22}\}S_2 \& R_2 \tag{2}$$

We will prove three lemmas.

(I) $P_1 \& P_2\{E_{21}\}P_1 \& \bar{P}_2 \& \bar{S}_2$

(II) $P_1 \& \bar{P}_2 \& \bar{S}_2\{E_1\}S_1 \& R_1 \& \bar{P}_2$

(III) $S_1 \& R_1 \& \bar{P}_2\{E_{22}\}R_1 \& R_2$

The conclusion of the rule follows directly by the rule of composition.

Lemma I

The only variable free in $\bar{S}_2$ is s_2, which is not changed by E_{22}. Its truth after E_{22} implies its truth before. Hence from (2) we get

$$P_2 \& S_1\{E_{21}\}\bar{S}_2$$

The only variable mentioned in E_{21} is s_2, which is not mentioned in S_1 Provided that there exist values satisfying S_1, it follows that

$$P_2\{E_{21}\}\bar{S}_2$$

(If S_1 were unsatisfiable, Q_1 would not terminate under any circumstances; and neither would $Q_1//Q_2$, which would make any conclusion about $Q_1//Q_2$ vacuously true). Since s_2 is not mentioned in P_1

$$P_1 \& P_2\{E_{21}\}\bar{S}_2 \& P_1 \quad .$$

The truth of $\bar{P}_2$ after E_{21} follows from the truth of P_2 before.

Lemma II

In (1), S_2 is the only part containing variables subject to updating by Q_2. By instantiating these variables, we can get:

$$P_1 \& \bar{S}_1\{E_1\}S_1 \& R_1$$

Since $\bar{P}_2$ contains no variable subject to change in E_1, the lemma follows immediately.

Lemma III

Since S_1 and $\bar{P}_2$ do not mention s_2, they are true after E_{22} if and only if they are true before. Hence from (2)

$$\bar{P}_2 \& S_1\{E_{22}\}R_2 \quad .$$

Since R_1 does not mention any variable subject to change in E_{22}, Lemma III is immediate.

On-the-fly garbage collection: an exercise in cooperation.

by

Edsger W.Dijkstra *)
Leslie Lamport **)
A.J.Martin ***)
C.S.Scholten ****)
E.F.M.Steffens ***)

*) Burroughs, Plataanstraat 5, NL-4565 NUENEN, The Netherlands

**) Massachusetts Computer Associates Inc., 26 Princess Street, WAKEFIELD, Mass. 01880, U.S.A.

***) Philips Research Laboratories, EINDHOVEN, The Netherlands

****) Philips-Electrologica B.V., APELDOORN, The Netherlands

Abstract. A technique is presented which allows nearly all of the garbage detection and collection activity to be performed by an additional processor, operating concurrently with the processor carrying out the computation proper. Exclusion and synchronization contraints between the processors have been kept weak.

Key Words and Phrases: garbage collection, multiprocessing, cooperation between sequential processes with minimized mutual exclusion, program correctness for multiprocessing tasks.

CR Categories: 4.32, 4.34, 4.35, 4.39, 5.23.

20th of October 1975

On-the-fly garbage collection: an exercise in cooperation.

Introduction.

In any large-scale computer installation today, a considerable amount of time of the (general purpose) processor is spent on "operating the system". With the emerging advent of multiprocessor installations the question arises to what extent such "housekeeping activities" can be carried out concurrently with the computation(s) proper. Because the more intimate the interference, the harder the organization of the cooperation between the concurrent processes, the problem of garbage collection was selected as one of the most challenging --and, hopefully, most instructive!-- problems. (Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap that we could possibly fall into.) In our treatment we have tried to blend a condensed design history --in order not to hide the heuristics completely-- with a rather detailed justification of our final solution. Whether the following solution, which is the result of many iterations, is of any economic significance, is a question beyond the scope of this paper.

We tackled the problem as it presents itself in the traditional implementation environment for pure LISP (and shall describe our solution in the usual terminology, leaving the natural generalizations to the reader). The data structure to be stored consists of a directed graph in which each node has at most two outgoing edges, more precisely: may have a left-hand outgoing edge and may have a right-hand outgoing edge. In the original problem statement, either of them or both could be missing; for the sake of homogeneity we follow the --not unusual-- practice of introducing a special purpose node called "NIL" and we represent an originally missing outgoing edge by an edge with the node called NIL as its target. As a result, each node has now always exactly two outgoing edges; the outgoing edges from NIL point to NIL itself. At any moment in time all the nodes must be "reachable" (via a directed path along the directed edges) from one or more fixed nodes --called "the roots"-- with a constant place in memory. The storage allocated to each node is constant in time and equal in size, viz. sufficient to accommodate two pointers --one for each outgoing edge-- pointing to the node's immediate successors. Given (the address of) a node, finding (the

address of) its left- or right-hand successor node can be regarded as an atomic, primitive action; finding its predecessor nodes, however, would imply a search through memory.

In the original problem statement, for a reachable node an outgoing edge could be deleted, changed or added. The effect of the special node NIL is that now all three modifications of the data structure take the same form, viz. the change of an outgoing edge of a reachable node. Note that such a change may turn a number of formerly reachable nodes into unreachable ones: they then become what is called "garbage". Changing an edge may direct the new edge towards a target node that was already reachable or towards a new node that has to be added to the data structure; such a new node --which upon creation has only NIL as successor node-- is taken from the so-called "free list", i.e. a linearly linked list of nodes that are currently not used for storing a node of the data structure. By linking the "free" nodes linearly --via their left-hand outgoing edge, say-- and introducing a special root pointing to the begin node of the free list, also the nodes of the free list can be regarded as reachable. By furthermore declaring that also the node called NIL is a root, we achieve our next homogenizing simplification: a change redirects for a reachable node one of its outgoing edges to a reachable node. (See Appendix.)

Garbage may arise anywhere in store, and it is the purpose of the so-called "garbage collector" to detect such disconnected and therefore obsolete nodes and to append them to the free list. In classical LISP implementations the computation proceeds until the free list is exhausted (or nearly so). Then the computation proper comes to a grinding halt, during which the processor is devoted to garbage collection. Starting from the roots, all reachable nodes are marked --because we have made the nodes of the free list reachable from a special root, nodes of the free list (if any) will in our case be marked as well-- . Upon completion of this marking phase, all unmarked nodes can be concluded to be garbage and are appended to the free list, after which the computation proper is resumed.

The minor disadvantage of this arrangement is the central processor time spent on the collection of garbage; its major disadvantage is the unpredictability of these garbage collecting interludes, which makes it hard

to design such a system so as to meet real time requirements as well. It was therefore tempting to investigate whether a second processor --called "the collector"-- could collect garbage on a more continuous basis, concurrently with the activity of the other processor --for the purpose of this discussion called "the mutator"-- which would be dedicated to the computation proper. We have imposed upon our solution a few constraints (compare [2]). The interference between collector and mutator should be minimal --i.e. no highly frequent mutual exclusion of elaborate activities, as this would defy our aim of concurrent activity-- , the overhead on the activity of the mutator (as required for the cooperation) should be kept as small as possible, and, finally, the ongoing activity of the mutator should not impair the collector's ability to identify garbage as such as soon as logically possible. (One synchronization measure is evidently unavoidable: when needing a new node from the free list, the mutator may have to be delayed until the collector has appended some nodes to the free list. This is the now traditional producer/consumer coupling; in the context of this article it must suffice to mention that this form of synchronization can be achieved without any need for mutual exclusion.)

Preliminary investigations.

A counterexample taught us that the goal "no overhead for the mutator" is unattainable. Suppose that nodes A and B are permanently reachable via a constant set of edges, while node C is reachable only via an edge from A to C . Suppose furthermore that from then on the mutator performs with respect to C repeatedly the following sequence of operations:

1) making an outgoing edge from B point to C
2) deleting the edge from A to C
3) making an outgoing edge from A point to C
4) deleting the edge from B to C .

The collector, which observes nodes one at a time, will discover that A and B are reachable from the roots, but never needs to discover that C is reachable as well: while A is observed by the collector, C may be reachable via B only, and the other way round. We may therefore expect that the mutator may have to mark in some way target nodes of changed edges.

Marking will be described in terms of colours. When we start with all

nodes white, and, furthermore, the combined activity of collector and mutator can ensure that eventually all reachable nodes become black, then all white nodes can be identified as garbage. For each repetitive process --and the marking process certainly is one-- we have always two concerns (see [1]): firstly we must have a monotonicity argument on which to base our proof of termination, secondly we must find an invariant relation which, initially true and not being destroyed, will still hold upon termination. For the monotonicity argument we suggest (fairly obviously)

during marking each node will darken monotonically.

For the invariant relation --a relation which must be satisfied both before and after the marking cycle-- we must generalize initial and final state of the marking process and our first guess was (perhaps less obvious, but not unnatural)

P1: during marking there will be no edge pointing from a black node to a white one.

Additional action is then required from the mutator when it is about to introduce an edge from a black node to a white one: just placing it would cause a violation of P1 . The monotonicity requirement tells us, that the black source node of the new edge has to remain black, and, therefore, P1 tells us that the target node of the new edge cannot be allowed to remain white. But the mutator cannot make it just black, because that could cause a violation of P1 between that new target node and its immediate successors. For that reason grey has been introduced as intermediate colour and the overhead considered for the mutator was

A1: when introducing an edge, the mutator shades its target node.

Note 1. Shading a node is defined to make a white node grey and to leave the colour of a grey or a black node unchanged. (End of note 1.)

The choice of the invariant relation P1 has been sufficient for a solution --not published here-- with a rather coarse grain of interleaving (in which, for instance, A1 was assumed to be available as a single, indivisible action). We could not use it, however, as a stepping stone towards a solution that allowed a finer grain of interleaving, because total absence

of an edge from a black node to a white one was a stronger relation than we managed to maintain. We could, however, retain the notion "grey" as "semi-marked", more precisely, as representing our unfulfilled marking obligation: as before, the marking activity of the collector remains localized at grey nodes and their possibly white successors.

A coarse-grained solution.

In our unpublished solution we made essential use of the fact that after the collector had initialized the marking phase by shading all roots, the validity of P1 allowed us to conclude that the existence of a white reachable node implied the existence of a grey node (even of a grey reachable node, but the reachability of such an existing grey node was not essential). A weaker relation from which the same conclusion can be drawn is

P2: during the marking cycle (that the collector has initialized by shading all roots) there exists for each white reachable node a so-called "propagation path", leading to it from a (not necessarily reachable) grey node, and consisting solely of edges with white targets (and, as a consequence, without black sources).

Note 2. In the absence of edges from a black node to a white one, relation P2 is clearly satisfied. (End of note 2.)

The existence of edges from a black node to a white one is restricted by

P3: during the marking cycle only the last edge placed by the mutator may lead from a black node to a white one.

Note 3. In the absence of black nodes, P3 is trivially satisfied. (End of note 3.)

When the mutator redefines an outgoing edge of a black node, it may direct it towards a white node: this new edge from a black node to a white one is permitted by P3 , but because the previously placed one could still exist and be of the same type, we consider for the mutator the following indivisible action:

A2: shade the target of the edge previously placed by the mutator and redirect for a reachable node one of its outgoing edges towards an already reachable node.

Note 4. For the very first time the mutator changes an edge we can assume that, for lack of a previously placed edge, the shading will be suppressed or an arbitrary reachable node will be shaded; the choice does not matter for the sequel. (End of note 4.)

Action A2 has been carefully chosen in such a way that it leaves P3 invariant; it leaves, however, the stronger relation P2 and P3 invariant as well.

Proof. The action A2 cannot introduce new reachable nodes; it, therefore, does not introduce new white ones for which extra propagation paths must exist. If the node whose successor is redefined is black, its outgoing edge that may have disappeared as a result of the change was not part of any propagation path, and the edges of the old propagation paths will be sufficient to provide the new propagation paths. (Possibly we don't need all of them as a result of the shading and/or white reachable nodes having become unreachable.) If the node whose successor is redefined was white or grey to start with, it will become at most grey and the resulting graph has no edge from a black node to a white one --if one existed, it has been removed by the shading and the change has not introduced a new one-- and (see Note 2) P2 holds upon completion. (End of proof.)

We have now reached the stage where we can describe our first collector, which repeatedly performs the following program. (Our bracket pairs "if...fi" and "do...od" delineate our alternative and repetitive constructs respectively (see [1]), comments have been inserted between braces and labels have been inserted for the discussion.) The program has two local integer variables i and k ; the nodes in memory are assumed to be numbered from 0 through M - 1 .

```
marking phase:
begin {there are no black nodes}
  C1: "shade all the roots" {P2 and P3};
      i:= 0; k:= M;
  marking cycle:
      do k > 0 → {P2 and P3}
            if C2: "node nr. i  is grey" →
                        k:= M;
                  C3: "shade the successors of node nr. i  and make node
                        nr. i  black" {P2 and P3}
             [] C2: "node nr. i  is not grey" →
                        k:= k - 1 {P2 and P3}
            fi {P2 and P3};
            i:= (i + 1)mod M
      od {P2 and P3 and there are no grey nodes, hence all white nodes
          are garbage}
end;
appending phase:
begin i:= 0;
      do i < M → {a node with a number  < i  cannot be black;
                  a node with a number  ≥ i  cannot be grey,
                  and is garbage, if white}
            if C2: "node nr. i  is white" →
                  C4: "append node nr. i  to the free list"
             [] C2: "node nr. i  is black" →
                  C5: "make node nr. i  white"
            fi;
            i:= i + 1
      od {there are no black nodes}
end
```

The indivisible actions of the collector --between the execution of which actions A2 of the mutator may occur-- are

1) "shading of a single root" (from which C1 is composed: the order in which the roots are shaded is irrelevant)

2) establishing the current colour of node nr. i (labeled "C2").

3) the total actions C3, C4 (see, however, the Appendix) and C5.

Remark 1. With a more elaborate administration local to the collector --a list of grey or possibly grey nodes-- a probably much more efficient marking phase could have been designed. For the sake of simplicity we have not done so. (End of remark 1.)

We observe that (even independent of the colour of node nr. i !) action C3: "shade the successors of node nr. i and make node nr. i black" can never cause a violation of P2 and P3 : the shading of the successors can never do any harm, as a result of the shading the outgoing edges of node nr. i are no longer needed for a propagation path, and making node nr. i black maintains the existence of the propagation paths needed without introducing an edge from a black node to a white one.

The state characterized by the absence of grey nodes, which implies on account of P2 that all white ones are garbage and that all reachable ones are black, is stable, because the absence of white reachable nodes prevents the mutator from introducing grey nodes, and the absence of grey nodes prevents the collector from doing so. Because, when a grey node is encountered, k is reset to M , the marking cycle can only terminate with a scan past all nodes, during which no grey node is encountered. Because the mutator leaves grey nodes grey, no grey node can have existed at the beginning of such a scan, i.e. the stable state must have been reached at that moment. Termination of the marking cycle is guaranteed because of the monotonicity of the colouring history of each node and because of the fact that resetting k to M is always accompanied by effective darkening of at least one node (nr. i to be precise).

When the appending phase starts, all reachable nodes are black and all white nodes are garbage. Note that the existence of black garbage is not excluded. The appending phase deals with each node in turn: as long as it has not been dealt with (i.e. has a number $\geq$ i) it cannot change colour: if black, it remains black because the mutator can only shade it, and if it is white, it is garbage and, by definition, the mutator won't touch it. As soon as it has been dealt with (i.e. has a number $<$ i), it has been white and can at most have been shaded by the mutator. Black garbage at the beginning of the appending phase will not be appended during that appending phase, it will only be made white; during the next marking phase it will remain white,

and the next appending phase will indeed append it. Therefore, no garbage, once created, will escape being collected.

A solution with a fine-grained collector.

We would like to break C3 open as a succession of five indivisible subactions, say (m1 and m2 being local variables of the collector):

C3.1: m1:= number of the left-hand successor of node nr. i ;
C3.2: shade node nr. m1 ;
C3.3: m2:= number of the right-hand successor of node nr. i ;
C3.4: shade node nr. m2 ;
C3.5: make node nr. i black

None of the actions C3.1 , C3.2 , C3.3, and C3.4 can cause violation of P2 and P3 . The actions C3.1 and C3.3 cannot do so because they leave no trace in common memory, and the actions C3.2 and C3.4 cannot do so because shading cannot do so. Besides that, because shading of a node commutes with any number of actions A2 , we have, by the time that the collector starts with C3.5, a state as if the shading of node nr. m1 had been part of C3.1 and the shading of node nr. m2 had occurred simultaneously with C3.3 . Without loss of generality we can continue our discussion as if "shade left-hand successor" and "shade right-hand successor" are available as indivisible actions. The problem, however, lies with C3.5 : can we safely make node nr. i black? Note that neither m1 , nor m2 needs still to be one of its successors: m1 and m2 even never need to have been its left- and right-hand successor simultaneously! A more thorough study of the mutator, however, reveals that it is safe.

Proof. During the marking phase we define a changing set of edges to which we give --in order to avoid false connotations-- the rather meaningless name "dodo-edges". (Note that we only define the set of dodo-edges for our benefit. The mutator and collector would have a hard time if they had to update it explicitly: in the jargon the term "ghost variable" is sometimes used for such an entity.) The set of dodo-edges is defined as follows as a function of the evolving computations:

1) at the beginning of the marking phase the set of dodo-edges is initialized with all the edges with a grey target

2) each time a white node becomes grey, all its incoming edges (that were

not already a dodo-edge) are added to the set of dodo-edges

3) when the action A2 , seen as a replacement of an outgoing edge, removes a dodo-edge --or an edge that, according to the second rule, would have become one as a consequence of A2's shading act-- the new edge that replaces it is also a dodo-edge: it "inherits the dodo-ness" of the edge it replaces.

The above rules imply that a dodo-edge is never needed for a propagation path. The last one all by itself implies that once the left-hand outgoing edge of a node is a dodo-edge, it will remain so, no matter how often redirected by the mutator, until the end of the marking phase, and that the same holds for the right-hand outgoing edge. In short: when, since the beginning of the marking phase, a given node has had a grey left-hand successor and has had a grey right-hand successor, it has two outgoing dodo-edges and making it black will never cause violation of P2 . It won't violate P3 either: if it has a white successor, the corresponding edge must have been the last one placed by the mutator (it can therefore have at most one white successor) and that edge from a black node to a white one is the one explicitly allowed by P3. (End of proof.)

The above argument sheds another light upon the action C3. Instead of waiting until it has seen both successors of a node to be non-white, it forces termination of that waiting process by shading its successors itself. It refrains from shading the successors of a white node, as that would defeat garbage detection, it also refrains from shading the successors of a black node (although such a black node could have a white successor) because that is unnecessary. It is in this sense that the grey nodes represent our unfulfilled marking obligation.

Note 5. In breaking up C3 we have placed C3.5 "make node nr. i black" at the end. As making a node black commutes with all other actions A2 and C3.1 through C3.4 , we could also have placed it at the beginning, before dealing (in some order) with the successors; P2 and P3 could then be violated temporarily. (End of note 5.)

A solution with a fine-grained mutator as well.

From the above it is obvious that no harm is done if at random moments a daemon would shade a reachable node. We now assume a very friendly daemon

that between any two successive actions A2 of the mutator shades the target node of the last placed edge. For the initial state of an action A2 during a marking cycle, we can now assert (besides P2 and P3) the absence of an edge from a black node to a white one, regardless of the question whether the last shading by the friendly daemon took place during the current marking phase, or earlier. As a result, the proof that A2 leaves P2 and P3 invariant is now also valid if A2 does not shade at all! Thanks to the daemon, it does not need to do so anymore! We can therefore replace A2 by the succession of the following two separate indivisible subactions:

"redirect for a reachable node an outgoing edge towards a reachable node" ;
"shade the target of the edge just placed".

Remark 2. The detailed implementation of what we have described as "a grain of interleaving" falls very definitely outside the scope of this paper: many techniques --even allowing concurrent access to the same unit of information-- are possible (see [3], [4]). (End of remark 2.)

In retrospect.

It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked --sometimes even for weeks and to many people-- like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug. Work has been done on formal correctness proofs ([5], [6]), but a shape that would make them fit for print has, to our tastes, not yet been reached. Hence our informal justification (which we do not regard as an adequate substitute for a formal correctness proof!). Whether its stepwise approach --which this time seems to have been successful in reducing the case analyses-- is more generally applicable, is at the moment of writing still an open question.

When it is objected that we still needed rather subtle arguments, we can only agree whole-heartedly: all of us would have preferred a simpler argument! Perhaps we should conclude that constructions that give rise to such tricky problems are not to be recommended. One firm conclusion, however, can be drawn: to believe that such solutions can be found without a very careful justification is optimism on the verge of foolishness.

History and acknowledgements. (As in this combination this is our first exercise in international and inter-company cooperation, some internal credit should be given as well.) After careful consideration of a wider class of problems the third and the fifth authors selected and formulated this problem and did most of the preliminary investigations; the first author found a first solution during a discussion with the latter, W.H.J.Feijen and M.Rem. It was independently improved by the second author --to give the free list a root and mark its nodes as well, was his suggestion-- and, on a suggestion made by Jack Mazola, by the first and the third author. The first and the fourth merged these embellishments, but introduced a bug that was found by N.Stenning and M.Woodger [7]. The final version and its justification are the result of a joint effort of the four authors in the Netherlands. The active and inspiring interest shown by David Gries is mentioned in gratitude.

References.

1. Dijkstra, Edsger W., Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Comm. ACM 18, 8 (Aug. 1975), 453-457.
2. Steele Jr.,Guy L., Multiprocessing Compactifying Garbage Collection. Comm. ACM 18, 9 (Sep. 1975), 495-508.
3. Lamport, Leslie. On Concurrent Reading and Writing. (Submitted to the Comm. ACM.)
4. Scholten, C.S., Private Communication
5. Gries, David, An Exercise in Proving Parallel Programs Correct. (Submitted to the Comm.ACM.)
6. Lamport, Leslie, Report CA-7508-0111, Massachusetts Computer Associates, Inc
7. Woodger, M., Private Communications.

Appendix.

Here we give an example of how the free list and the operations such as taking a node from or appending a node to the free list can be implemented. We consider the nodes of the free list ordered according to "age". For each node in the free list, the right-hand successor is NIL, the left-hand successor is NIL for the youngest node and is the next-younger one for the others. We have a root called TAKE , its left-hand successor and its right-hand successor are both the oldest free node; we have a second root called APP , whose left-hand and right-hand successor are both the youngest free node.

Taking a free node --and making it the left-hand successor of some reachable node X , say-- can be done in the following steps (shown in a hopefully self-explanatory notation):

```
X.left:= TAKE.left;                    (All four actions should follow
TAKE.left:= TAKE.right.left;            the shading convention chosen.)
TAKE.right.left:= NIL;
TAKE.right:= TAKE.left
```

To append, say, node Y --in action C4-- could be done by:

```
Y.left:= NIL; Y.right:= NIL;
APP.left:= Y;
APP.right.left:= Y;
APP.right:= APP.left
```

When a minimum of two free nodes is maintained, the collector that appends is certain only to deal with nodes that are left alone by the mutator, and the action C4 need not be regarded as a single, indivisible action, but is trivially allowed to be broken up in the above subactions. The synchronizati guaranteeing the lower bound for the length of the free list is here supposed to be implemented by other, independent means.

20th of October 1975

D. Gries

Cornell University and Technical University Munich

An Exercise in Proving Parallel Programs Correct *

Abstract:

A parallel program, Dijkstra's on-the-fly garbage collector, is proved correct using a proof method developed by Owicki. The fine degree of interleaving in this program makes it especially difficult to understand, and complicates the proof greatly. Difficulties with proving such parallel programs correct are discussed.

* This research was partially supported by the National Science Foundation under grant GJ-42512.

1. Introduction

At the NATO international Summer School on Language Hierarchies and Interfaces, Marktoberdorf, 1975, Dijkstra presented an "on-the-fly" garbage collector. Dijkstra and his colleages had tackled this problem "as one of the more challenging - and hopefully instructive - problems" in parallel programming. Indeed, the high degree of interleaving of the processors' actions made his solution, and the arguments about its correctness, difficult to understand. The major difficulty was the lack of necessary tools, the lack of any systematic method for understanding parallelism. I had recently worked with Susan Owicki and read her thesis [3] on methods for proving properties of parallel programs, and as Dijkstra presented his solution at the summer school, it struck me that with Owicki's techniques I could perhaps provide a better understanding of the program. With some help from Dijkstra and Tony Hoare, I was able to arrive at an outline of a proof of correctness of the garbage collector, and to present it a few days later at the Summer School.

A fully detailed, complete proof, however, took me much longer, partly because I was not adept enough yet at applying the techniques, but also because proving properties of parallel programs is so much harder than proving properties of sequential programs.

Owicki's proof techniques deserve further study, and this paper attempts to describe them and their use in the context of Dijkstra's garbage collector. In section 2 we present and discuss some of Owicki's proof techniques. In section 3, we describe the garbage collection problem and give the solution, along with an informal discussion of its correctness. Section 4 is devoted to more formally establishing its correctness.

This exercise has convinced me that parallel programming is much harder than sequential programming, and that we must use more systematic techniques if we are to master it. The reader might want to study section 5, the conclusions, after looking at the solution but before reading its correctness proof, in order to fully understand the problems of parallelism.

This exercise could not have been possible without a good knowledge of Owicki's thesis, and I am grateful for the privilege of supervising her thesis work. I am grateful to Edsger Dijkstra for showing us the garbage collection problem, and to Dijkstra and Hoare for help in developing the proof in its various stages. I thank the members of IFIP working group WG 2.3 (programming methodology) for the opportunity to present and discuss this material at the September 1975 meeting. Finally, thanks go to Mrs. Heilmann for her excellent job of typing.

2. Definition and use of the language

Let S be a statement and P and Q assertions about variables. In [2], Hoare introduces notation like {P} S {Q} to informally mean:
if P is true before execution of S then Q will be true when S terminates.
This is a statement of partial correctness; termination of S must be established by other means. Hoare introduces axioms similar to the following for a fragment of ALGOL

(2.1) null $\quad\quad$ {P} skip {P}

(2.2) assignment $\quad\quad$ $\{P_e^x\}$ x := e {P}

where P_e^x represents the result of substituting (e) for each free occurence of x in P . e.g. if P is $(a>0 \wedge b=1)$, then P_{a+b}^a is $(a+b>0 \wedge b=1)$. It should be recognized that this axiom holds only when variable x has no other name used in e or P ; otherwise the axiom may not be consistent.

(2.3) alternation $\quad\quad$ $\dfrac{\{P \wedge B\}\ S1\ \{Q\},\ \{P \wedge \neg B\}\ S2\ \{Q\}}{\{P\}\ \text{if}\ B\ \text{then}\ S1\ \text{else}\ S2\ \{Q\}}$

(2.4) iteration $\quad\quad$ $\dfrac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while}\ B\ \text{do}\ S\ \{P \wedge \neg B\}}$

(2.5) composition $\quad\quad$ $\dfrac{\{P0\}\ S1\ \{P1\},\ \{P1\}\ S2\ \{P2\},\ \ldots,\ \{Pn-1\}\ Sn\ \{Pn\}}{\{P0\}\ \text{begin}\ S1\ ;\ \ldots\ ;\ Sn\ \text{end}\ \{Pn\}}$

(2.6) consequence $\quad\quad$ $\dfrac{\{P1\}\ S\ \{Q1\},\ P \vdash P1,\ Q1 \vdash Q}{\{P\}\ S\ \{Q\}}$

Let us now briefly discuss proofs of properties of sequential programs. When we write {P} S {Q}, this implies the existence of a proof {P} S {Q} using the axioms and inference rules (2.1)-(2.6). For example suppose we have already proved
$\{P1 \wedge e\}$ S1 {Q1} and $\{P1 \wedge \neg e\}$ S2 {Q1}, and suppose we have

S ≡ begin x := a ; if e then S1 else S2 end

Then a proof of {P} S {Q} might be

(1) $\{P1^x_a\}$ x := a {P1} — assignment

(2) $\dfrac{\{P1^x_a\}\ x := a\ \{P1\},\quad P \vdash P1^x_a}{\{P\}\ x := a\ \{P1\}}$ — consequence

(3) $\dfrac{\{P1 \wedge e\}\ S1\ \{Q1\}\quad \{P1 \wedge \neg e\}\ S2\ \{Q1\}}{\{P1\}\ \underline{\text{if}}\ e\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \{Q1\}}$ — alternation

(4) $\dfrac{\{P1\}\ \underline{\text{if}}\ e\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \{Q1\},\quad Q1 \vdash Q}{\{P1\}\ \underline{\text{if}}\ e\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \{Q\}}$ — consequence

(5) $\dfrac{\{P\}\ x := a\ \{P1\},\quad \{P1\}\ \underline{\text{if}}\ e\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \{Q\}}{\{P\}\ \underline{\text{begin}}\ x := a;\ \underline{\text{if}}\ e\ \underline{\text{then}}\ S1\ \underline{\text{else}}\ S2\ \underline{\text{end}}\ \{Q\}}$ — composition

This proof can be outlined more compactly and understandably by interleaving statements and assertions:

(2.7)

```
{P}
begin  {P}
       {P1^x_a}
       x := a
       {P1}
       if e then {P1 ∧ e}  S1  {Q1}
            else {P1 ∧ ¬e}  S2  {Q1}
       {Q1}
       {Q}
end
{Q}
```

In a proof outline, two adjacent assertions {P1} {P2} denote the use of the rule of consequence, where $P1 \vdash P2$. Secondly, each statement S is preceded directly by an assertion called the precondition of S, written pre(S). The precondition of x := a above is $P1^x_a$.

Owicki [3] introduces two statements for parallel processing. The cobegin statement indicates that processes are to be executed in parallel; the await statement provides synchronization and mutual exclusion. The await statement has the form

await B then S

where B is a Boolean expression and S a statement. Execution of the await is delayed until B is true. At this time, S is executed as an indivisible operation - no other process may execute while S is executing, or during the time that B is found to be true and execution of S is begun, since this might falsify B. If two delayed processes have their corresponding Booleans B come true at the same time, one of them is further delayed while the other executes. The scheduling algorithm for determining which process is allowed to proceed does not concern us here.

The formal definition of the await statement is:

(2.8) await $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{await}\ B\ \text{then}\ S\ \{Q\}}$

Before introducing the cobegin statement, let us explain what it means for two paralle processes to be interference-free.

(2.9) Definition. Given $\{P\}\ S\ \{Q\}$, let T be some other assignment or await statement with precondition pre(T). We say that T does not interfere with the proof of $\{P\}\ S\ \{Q\}$ if

(a) $\{Q \wedge \text{pre}(T)\}\ T\ \{Q\}$

(b) for each statement S' of S which is not within an await, $\{\text{pre}(S') \wedge \text{pre}(T)\}\ T\ \{\text{pre}(S')\}$

Thus, execution of T can not affect the truth of the preconditions and result condition used in the proof of S and hence the proof $\{P\}\ S\ \{Q\}$ still holds, even if T is executed in parallel with S .

(2.1o) Definition. $\{P1\}\ S1\ \{Q1\}$ and $\{P2\}\ S2\ \{Q2\}$ are interference-free if each assignment statement of S2 (which does not occur within an await) and each await of S2 does not interfere with the proof of $\{P1\}\ S1\ \{Q1\}$, and vice versa.

If S1 and S2 are intereference-free as just defined, then execution of S2 leaves valid all the arguments used in the proof $\{P1\}\ S1\ \{Q1\}$, and therefore the proof still holds in the face of parallel execution. This allows us to define the parallel cobegin statement as follows:

(2.11) parallelism $\dfrac{\{P1\}\ S1\ \{Q1\}\ ,\ \{P2\}\ S2\ \{Q2\}\quad \text{interference-free}}{\{P1 \wedge P2\}\ \text{cobegin}\ S1\ //\ S2\ \text{coend}\ \{Q1 \wedge Q2\}}$

In any operational model consistent with this and the other axioms, statements S1 and S2 are executed in parallel, and execution of a cobegin terminates only when both S1 and S2 have terminated. No assumptions about the relative speeds of the processes S1 and S2 are made. Evaluation of any expression or execution of any assignment, however, must be performed as an indivisible operation which cannot be interrupted. But we can lift even this restriction if we adhere to the following (which this paper does):

(2.12) Any expression e in process Si may contain at most one reference to at most one variable changed in the other process Sj. If variable x of x := e in Si is referenced by process Sj , then e can contain no references to x or to a variable changed in Sj .

With this convention, the only indivisible action need be the memory reference. Suppose process Si changes variable (location) A while process Sj , $j \neq i$, is referencing A . The memory must have the property that the value received for A by process Sj be either the value of A before, or after, the assignment, but it may not be garbage caused by fluctuation of the state of memory during the assignment to A . Thus the methods described here can be used to prove properties of programs executing on any reasonable machine, with as fine a grain of interleaving as one could imagine. Dijkstra's on-the-fly garbage collector takes advantage of such a fine grain of interleaving.

One often must be able to delete variables from (or add variables to) a program in order to effect a proof. The following definitions allow this.

(2.13) Definition. Let AV be a set of variables which appear only in assignments x := e , where $x \in AV$. Then AV is an auxiliary variable set for S .

(2.14) Definition. Let AV be an auxiliary variable set for S' . S is a reduction of S' if it is obtained from S by one of the operations
(1) Delete all assignments x := e where $x \in AV$, or
(2) Replace await true then x := e by x := e , provided x := e satisfies (2.12).

(2.15) Auxiliary variable axiom. Let AV be an auxiliary variable set for S' , S a reduction of S' with respect to AV , and P and Q assertions which do not contain variables from AV . Then

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

We now have a system for proving partial correctness of parallel programs. We shall see that we cannot use it completely and formally, because the processes we deal with may not even terminate! But we can use the insight gained to informally prove properties of parallel programs.

The formalization teaches us to understand parallel processes in two steps. First, prove the properties of each parallel process S1 and S2 as sequential programs, disregarding completely parallel execution. Secondly, show that execution of S2 does not destroy the proof of S1's properties, and vice versa, for if parallel execution of S2 does not invalidate the proof, it cannot destroy the desired properties!

This is an important step forward in understanding parallelism. Earlier work has often tried to show that execution of S2 does not interfere with the execution of S1. By concentrating more on the proof, we turn our attention to a more static object which is easier to handle. Of course, the sequential proofs may turn out to be harder, because we must often weaken or change the arguments so that they will not be destroyed by parallel activity. This will become clear later.

We shall subsequently apply this technique. We will not prove that subparts of a sequential programs work correctly if it is obvious; we shall use proof outlines as in (2.7), and we will often leave out implications $P \vdash Q$ if they can be discerned by the reader. We shall also use other statement notations where clearer, and will make program transformations without a formal proof rule, if the transformations are obviously correct. The assertions themselves will often be at a high, informal level, in an attempt to be clear without having to resort to too much formalism.

3. On-the-fly Garbage Collection

The data structure used in a conventional implementation of LISP is a directed graph in which each node has at most two outgoing edges (either of which may be missing): an outgoing left edge and an outgoing right edge. At any moment all nodes of the graph must be reachable (via a directed path along directed edges) from a fixed root which has a fixed, known place in memory. The storage allocated for each node is constant in size and can accomodate two pointers, one for each outgoing edge. A special value nil denotes a missing edge.

For any reachable node, an outgoing edge may be deleted, changed or added. Deletion and change may turn formerly reachable nodes into unreachable nodes which can no longer be used by the program (henceforth called the mutator). These unreachable nodes are therefore called garbage. Nodes not being used by the mutator are stored on a

free list maintained as a singly linked list. The mutator may delete the first node from the free list and insert it into the directed graph by placing an edge to it from a reachable node.

If the free list becomes empty, computation halts and a process called "garbage collection" is invoked. Beginning with the roots, all reachable nodes are marked; upon completion of this marking phase, all unmarked nodes are known to be garbage and are appended to the free list. Computation then resumes.

A major disadvantage of this arrangement is the unpredictability of the garbage collection interludes. Dijkstra and his colleagues therefore investigated the use of a second processor, the collector, which would collect garbage on a more continuous basis, concurrently with the action of the mutator. The constraints imposed on their solution were:

> the "interference between collector and mutator should be minimal ..., the overhead on the activity of the mutator (as required for cooperation) should be kept as small as possible, and finally, the ongoing activity of the mutator should not impair the collector's ability to identify garbage as such as soon as possible."

Their solution satisfies these criteria, and we make no improvement on it at all; we are concerned only with the proof and description of their solution. Overhead on the mutator is one or two simple assignments per changed or added edge, the only indivisible action need be the memory reference, and the only synchronization is when the mutator must wait for the free list to have more than one node before taking a node from it.

We now turn to the algorithm itself. The collector has two phases: marking reachable nodes, and collecting unmarked, unreachable nodes. For marking, we must use three colors: white represents unmarked, black marked, and gray an "inbetween" color needed for cooperation between collector and mutator. To see the need for the third color, suppose we use only black and white, and let nodes N and M be as depicted in state 1 of Fig. 1. Now let the mutator repeatedly perform the following sequence of actions:

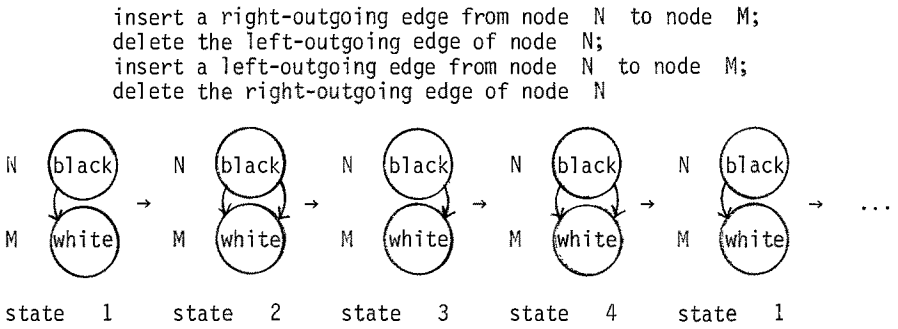


Fig. 1. non-cooperation when using only two colors

M is thus always reachable from N . If M is white, the collector must notice that M is N's successor, and blacken M . But the collector might never see this, for it might always check N's left-outgoing edge when it is nil (i.e. in state 3), and might always check N's right-outgoing edge when it is nil (i.e. in state 1).

Thus, the mutator must cooperate in some fashion, and does so by graying a white node when it draws an edge to it.

We now come to the representation of the graph of nodes. We use an array m[0:N] for the nodes. nil is represented by 0 , and thus the mutator itself may never reference node m[0] . This is not necessary, but makes presentation of the collector easier. We shall often speak of "node i" or just "i", instead of using the longer term "node m[i]".

Each node has three subfields which are of interest to us:

```
(3.1)   m[i].color   current color of node (white, gray or black)
        m[i].left    node i's left successor (0 if none)
        m[i].rite    node i's right successor (0 if none)
```

The following indivisible actions are used to color nodes:

```
(3.2)   whiten(i):        m[i].color := white
        blacken(i):       m[i].color := black
        atleastgray(i):   if m[i].color=white then m[i].color := gray
```

Note that a black node is not made gray by operation atleastgray. These operations could be implemented using two bits, with white = 00, gray = 01 and black = 11. The operation atleastgray(i) would consist of "oring" the pattern 01 into m[i].color; this is a single instruction "or to memory" on many machines.

Two nodes m[ROOT] and m[FREE] are in fixed, constant places in the array m[0:N]. m[ROOT] is the single root of the mutator's graph, while m[FREE] is used to indicate where the free list begins. Within the collector, we do not distinguish between these two roots (and nil), and thus the free list and node m[0] will be marked and unmarked just as the mutator's graph is.

The free list is maintained using an extra integer variable ENDFREE to mark the end of the free list. Fig. 2 illustrates the free list, while the following definition Ifree describes it more exactly; Ifree is invariantly true throughout execution.

(3.3) Ifree ≡

(a) the free list contains $j \geq 1$ nodes with distinct indices
m[FREE].left = m[FREE].left^1 ≠ 0,
m[m[FREE].left].left = m[FREE].left^2 ≠ 0,
...
m[FREE].left^j ≠ 0;

(b) m[FREE].left^{j+1} = 0;

(c) ENDFREE = m[FREE].left^{j-1} ∨ ENDFREE = m[FREE].left^j;

(d) all nodes on the free list have no right successors.

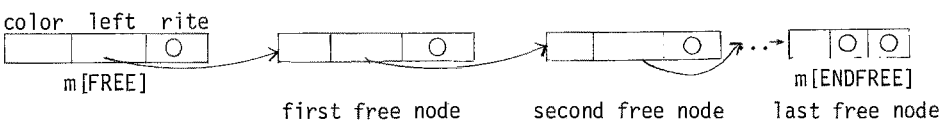


Fig. 2. The free list

The mutator has at its disposal two procedures to add edges from one node to another:

(3.4) {Add a left outgoing edge from m[k] to m[j]}
proc addleft(k,j); begin m[k].left := j; atleastgray(j) end;
{Add a right outgoing edge from m[k] to m[j]}
proc addrite(k,j); begin m[k].rite := j; atleastgray(j) end

The mutator we write in (3.5) as a never-ending, nondeterministic guarded command [4]. The only operations to appear are those which any mutator must use to change the structure of the graph. The while loops are used to make the mutator wait until the free list has two or more nodes, before taking one off it. Variables k , j and f are local to the mutator; the collector also does not change FREE .

```
(3.5)  mutator:  do true ⇒
         Let  k, j be indices of nodes reachable from ROOT (k ≠ 0, j ≠ 0);
         if true ⇒ m[k].left := 0
          [] true ⇒ m[k].rite := 0
          [] true ⇒ addleft(k,j)
          [] true ⇒ addrite(k,j)
          [] true ⇒ Take first free node as  k's  left successor:
                    f := m[FREE].left;
                    addleft(k,f) ;
                    while f = ENDFREE do skip;
                    addleft(FREE, m[f].left);
                    m[f].left := 0
          [] true ⇒ Take first free node as  k's  right successor:
                    f := m[FREE].left;
                    addrite(k,f);
                    while f = ENDFREE do skip;
                    addleft(FREE, m[f].left);
                    m[f].left := 0
         fi
       od
```

The collector is given below in (3.6). When first studying it, remember the insight gained from the formalism and treat it as an independent, sequential program under no parallel influence.

At the beginning of each execution of the body of the collector's main loop, there are no black nodes. Execution of the first section of the body grays the roots, so that any reachable white node is reachable from a gray node (without going through a black node). After execution of the second section Blacken gray nodes ... (we will look at this in detail subsequently), all reachable nodes are black, so that all white nodes are garbage. The third section labeled Collect then searches through the nodes, appending white ones to the free list and whitening black nodes, in preparation for the next iteration.

```
(3.6)  Collector: do true ⇒
         Make roots at least gray:
           atleastgray(ROOT); atleastgray(FREE); atleastgray(NIL);
         Blacken gray nodes and nodes reachable from gray nodes:
           i := 0;
           do i ≤ N and m[i].color ≠ gray ⇒ i := i + 1
           [] i ≤ N and m[i].color = gray ⇒ atleastgray(m[i].left); *
                                            atleastgray(m[i].rite);
                                            blacken(i);
                                            i := 0
           od;
         Collect: Put white nodes on free list and whiten black nodes:
           for i := 0 step 1 until N do
             if m[i].color = white ⇒ Append i to free list:
                                      m[i].left := 0;
                                      m[i].rite := 0;
                                      m[ENDFREE].left := i;
                                      ENDFREE := i
             [] m[i].color = black ⇒ whiten(i)
             [] m[i].color = gray  ⇒ skip
             fi
           od
```

The second section labeled Blacken gray nodes ... searches through all nodes, both reachable and unreachable ones. Upon encountering a gray node, it grays its successors (if white), and then blackens it. Thus, every reachable white node is always reachable from a gray node, which we express as

(3.7) i white and reachable ⇒ ∃ path $(k_1,\ldots,k_p,i)$ where k_1 is gray and $k_2,\ldots,k_p$ are white

The effect is that, beginning with the gray nodes, all reachable nodes are first grayed and then blackened, in waves spreading out from the roots. If a gray node becomes unreachable (because of mutator interaction), it nevertheless is blackened, along with its successors.

Each time a gray node is found and blackened, the collector begins checking the nodes again from the beginning. If no gray node is found during a complete traversal, then all nodes are black or white. From (3.7) and the absence of gray nodes we conclude that all reachable nodes are black, and that all white nodes are garbage and can be collected.

---

* This should be written as "t := m[i].left; atleastgray(t)" where t is a local variable. Since the mutator never tests the color of a node and only grays a node using also atleastgray, the single statement atleastgray(m[i].left) is equivalent under parallel operation to this sequence of two operations.

The node traversal algorithm in Blacken gray nodes ... has been made simple and inefficient in order to simplify the correctness proof. Any traversal algorithm can be used which makes a final pass through all nodes without finding a gray node; this last pass is necessary because of interaction by the mutator.

The collector, as an independent algorithm, doesn't do much. It alternately blackens all white nodes - including the free list - and then makes them white again. The free list is repeatedly marked and unmarked, and there is never any garbage to collect!

Let us now consider interaction by the mutator. Suppose the mutator takes a node M from the free list, grays it (using procedure addleft or addrite), and then deletes an edge so as to make M unreachable. Thereafter, the mutator may not reference M until it has been put on the free list. During the subsequent marking phase the collector blackens M, and during the following collecting phase it whitens M. During the next marking phase M remains white and unreachable, so that the following collection finally appends M to the free list. Thus, any reachable node is appended to the free list within two collecting phases after it becomes unreachable.

This discussion does not prove correctness, for it does not take into account the mutator interaction. The main problem is with mutator interference when the collector is about to blacken node i. The collector assumes that i's successors are nonwhite. However, just before the operation blacken(i) the mutator might interrupt and change i's successor to a white node. Blackening i could then lead to a black-to-white edge and destruction of the important invariant (3.7). (3.7) is needed in order to be able to prove that all reachable nodes have been blackened, or marked. We must show that (3.7) always holds during marking even in the face of mutator interaction.

The reader might wish to think about the following problem in light of this discussion: should procedure addleft(k,j) be written as

```
                m[i].left := j;   atleastgray(j)
or as           atleastgray(j);   m[i].left := j ?
```

4. Proof of correctness of the mutator-collector system

We give proof outlines of the main program, the collector's marking phase, the collector's collection phase, and the mutator, in that order. We follow this with a discussion of the interference-free property. We will use the following notation:

(4.1) $\mathcal{L}n$ ≡ m[n].left -- n's current left successor
$\mathcal{R}n$ ≡ m[n].rite -- n's current right successor
reach(n) ≡ ∃ path(ROOT,...,n) or a path(FREE,...,n)
reachR(n) ≡ ∃ path(ROOT,...,n)
gray-reachable(n) ≡ ∃ path(k_1,...,k_p,n) where k_1 is gray and k_2,...k_p are white

We use two auxiliary variables. Variable mark indicates whether the collector is currently marking; it is referred to in the mutator's assertions but not in the mutator itself. Variable add indicates whether the mutator is currently executing in procedure addleft of addrite. These variables are needed only for the proof; by virtue of axiom (2.15) they may be deleted without disturbing the correctness of the system.

The mutator has its set of reasonable states under which it can cooperate; we describe these in the following assertions:

(4.2) Mfree ≡ Ifree ∧ no free list node is reachable from ROOT
Mgraph ≡ add=0 ∧ (mark ⇒ there is no black-to-white edge)

The mutator actions are simple enough to understand without a formal proof outline, but we must use these assertions and a proof outline in order to show non-interference

The collector also has its set of reasonable states which concern the free list and the colors of nodes at various stages of execution. We group them here to facilitate referencing them later; the reader might wait until they are referenced in a proof outline before studying them.

(4.3) Cfree ≡ Ifree ∧ ENDFREE=0 ∧ NIL=$\mathcal{L}$NIL=$\mathcal{R}$NIL=0
Cmark ≡ mark ∧ ROOT,FREE and NIL are nonwhite ∧
(n white and reachable ⇒ gray-reachable(n))
Cm(i) ≡ 0≤i≤N+1 ∧ (m[0:i-1] contains a gray node implies m[i:N] contains a gray node)
C$\mathcal{L}$ ≡ $\mathcal{L}$i white ⇒ i=k=add≠0 ∧ ∃path(k_1,...,k_p,$\mathcal{L}$i) with k_1 gray, k_2,...,k_p white, and k_1=i implies k_2=$\mathcal{R}$i
C$\mathcal{R}$ ≡ $\mathcal{R}$i white ⇒ i=k=-add≠0 ∧ ∃path(k_1,...,k_p,$\mathcal{R}$i) with $k_1$gray; k_2,...,k_p white, and k_1=i implies k_2=$\mathcal{L}$i
Ccoll(i)≡ ¬ mark ∧ (0≤n≤i ⇒ n nonblack)
∧ ((i<n≤N ∧ n white) ⇒ n is not reachable)

We are finally ready to give the proof outlines of the various sections. Look upon each as a sequential program, and the proof outlines should not be hard to understand. Difficulties arise only because assertions have been weakened in order to show non-interference later on.

4.1 Proof outline for the main program

Note that there is no terminating condition; whether the system halts depends on the particular mutator being executed. We are interested only in proving properties which hold as long as the mutator is executing.

```
(4.1.1)  {Mfree ∧ Cfree ∧ ¬ mark ∧ add = 0 ∧ no black nodes}
         cobegin {Cfree ∧ ¬ mark ∧ no black nodes}
                 collector:  do true  ⇒
                   {Cfree ∧ ¬ mark ∧ no black nodes}
                   Make roots at least gray;
                   Blacken gray nodes and nodes reachable from grays;
                   {Cfree ∧ ¬ mark ∧ all white nodes are unreachable}
                   Put white nodes on free list and whiten black nodes
                   {Cfree ∧ ¬ mark ∧ no black nodes}
                  od
           //    {Mfree ∧ Mgraph}
                 mutator
         coend
```

4.2 Proof outline for the marking phase

This phase consists of atleast graying the roots and blackening gray nodes...; Its input-output assertions are given in (4.1.1). Some comments are in order concerning the assertions used in the proof outline (4.2.2), which have been weakened in order to prove non-interference later on. A case in point is assertion Cm(i), given in (4.3). We would have liked to use the assertion

$$0 \leq i \leq N + 1 \wedge m[0:i-1] \text{ contains no gray node}$$

However, the mutator can gray a node in the mentioned partition. This forced us to use assertion Cm(i) instead. In the same way, instead of using the simple assertion "$\mathfrak{L}i$ is nonwhite", we are forced to use the more complicated assertion C$\mathfrak{L}$.

Note that from Cmark ∧ CR ∧ CL we can conclude that at most one of i's successors is white. Secondly, if, say Li is white, then it is gray-reachable from a node other than i, and hence blackening i will not destroy the assertion Cmark.

We must show that the loop of (4.2.2) terminates. Consider the following integer function f :

(4.2.1) f ≡ 3·N·(no. of white nodes) + 2·N·(no. gray nodes) + N + 1 - i

Each execution of the loop body reduces f by at least 1. Furthermore f ≥ 0. Hence after a finite number of iterations the loop terminates.

```
(4.2.2) {Cfree ∧ ¬ mark ∧ no black nodes}
        Make roots ROOT, FREE and NIL at least gray;
        {Cfree ∧ ¬ mark ∧ no black nodes ∧ roots are nonwhite}
        mark := true;
        {Cfree ∧ Cmark}
        i := 0
        {Cfree ∧ Cmark ∧ Cm(i)}
        do i ≤ N ∧ m[i].color≠gray ⇒ {Cfree ∧ Cmark ∧ Cm(i) ∧ i nongray}
                                     {Cfree ∧ Cmark ∧ Cm(i+1)}
                                     i := i + 1
                                     {Cfree ∧ Cmark ∧ Cm(i)}
        ▯  i ≤ N ∧ m[i].color=gray ⇒ {Cfree ∧ Cmark ∧ i gray}
                                     atleastgray(m[i].left);
                                     {Cfree ∧ Cmark ∧ i gray ∧ Li nonwhite}
                                     {Cfree ∧ Cmark ∧ i gray ∧ CL}
                                     atleastgray(m[i].rite);
                                     {Cfree ∧ Cmark ∧ i gray ∧ CL ∧ Ri nonwhite}
                                     {Cfree ∧ Cmark ∧ i gray ∧ CL ∧ CR}
                                     blacken(i)
                                     {Cfree ∧ Cmark ∧ i black}
                                     i := 0
                                     {Cfree ∧ Cmark ∧ Cm(i)}
          {Cfree ∧ Cmark ∧ Cm(i)}
        od;
        {Cfree ∧ Cmark ∧ Cm(N+1)}
        {Cfree ∧ Cmark ∧ all white nodes are unreachable}
        mark := false
        {Cfree ∧ ¬ mark ∧ all white nodes are unreachable}
```

4.3 Proof outline for the collecting phase

(4.3.1) Collect:

```
{Cfree ∧ ¬ mark ∧ all white nodes are unreachable}
{Cfree ∧ Ccoll(-1)}
for i := 0 step 1 until N do
      {Cfree ∧ Ccoll(i-1)}
      if m[i].color=white ⇒
          {Cfree ∧ Ccoll(i) ∧ ¬ reach(i)}
          m[i].left := 0; m[i].rite := 0;
          {Cfree ∧ Ccoll(i) ∧ ¬ reach(i) ∧ ℒi = ℛi = 0}
          m[ENDFREE].left := i;
          {Ifree ∧ Ccoll(i) ∧  ENDFREE = i ≠ 0 ∧ NIL = ℒNIL = ℛNIL = 0}
          ENDFREE := i
          {Cfree ∧ Ccoll(i) ∧ ENDFREE = i}
      ▯ m[i].color=black ⇒ {Cfree ∧ Ccoll(i-1) ∧ i black}
                           whiten(i)
                           {Cfree ∧ Ccoll(i)}
      ▯ m[i].color=gray  ⇒ skip   {Cfree ∧ Ccoll(i)}
      fi
      {Cfree ∧ Ccoll(i)};
{Cfree ∧ Ccoll(N)}
{Cfree ∧ ¬ mark ∧ no black nodes}
```

4.4 Proof of properties of the mutator

We begin with two lemmas about procedures addleft and addrite. The extra auxiliary variable add is needed later to show non-interference; by the auxiliary variable axiom, assignments to add, as well as the awaits, can be deleted. Implied in these lemmas is that these procedures have no effect except as stated.

(4.4.1) Lemma

```
{Mgraph ∧ reach(k) ∧ reach(j)}          {Mgraph ∧ reach(k) ∧ reach(j)}
addleft(k,j)                     and   addrite(k,j)
{Mgraph ∧ reach(k) ∧ ℒk = j}            {Mgraph ∧ reach(k) ∧ ℛk = j}
```

Proof outlines

```
proc addleft(k,j);
 begin  {Mgraph ∧ reach(k) ∧ reach(j)}
        await true then begin m[k].left := j;  add := k  end;
        {k = add ≠ 0 ∧ reach(k) ∧ ℓk = j
        ∧ ( mark ⇒ the  only possible black-to-white edge is (k,ℓk)}
        await true then begin atleastgray(j);  add := 0  end
        {Mgraph ∧ reach(k) ∧ ℓk = j}
 end

proc addrite(k,j) ;
 begin  {Mgraph ∧ reach(k) ∧ reach(j)}
        await true then begin m[k].rite := j;  add := -k  end;
        {k = -add ≠ 0 ∧ reach(k) ∧ ℛk = j
        ∧ ( mark ⇒ the only possible black-to-white edge is (k,ℛk) }
        await true then begin atleastgray(j);  add := 0  end
        {Mgraph ∧ reach(k) ∧ ℛk = j}
 end
```

We are now ready to give the proof outline of the mutator. We show only three operations -- those dealing with the left successor of a node; the other three operations dealing with the right successor are similar. When showing non-interference, we shal also deal only with these three operations.

```
(4.4.4)  {Mfree ∧ Mgraph}
         mutator :  do true  ⇒
           {Mfree ∧ Mgraph}
           Let k, j be indices of nodes reachable from ROOT;
           {Mfree ∧ Mgraph ∧ reachR(k) ∧ reachR(j)}
           if true ⇒ {Mfree ∧ Mgraph ∧ reachR(k)}
                     m[k].left := 0
                     {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℓk = 0}
           ▯ true ⇒ {Mfree ∧ Mgraph ∧ reachR(k) ∧ reachR(j)}
                     addleft(k,j)
                     {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℓk = j}
           ▯ true ⇒ Take first free node as k's left successor:
                       {Mfree ∧ Mgraph ∧ reachR(k)}
                       f := m[FREE].left;
                       {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℓFREE = f ≠ 0}
                       addleft(k,f);
                       {Ifree ∧ Mgraph ∧ reachR(k) ∧ ℓFree = ℓk = f ≠ 0  ∧
                       every path from ROOT to freelist uses edge (k,ℓk)  }
```

```
                while f = ENDFREE do skip ;
                {Ifree ∧ Mgraph ∧ reachR(k) ∧ ℒFREE = ℒk = f ≠ 0 ∧ ℒf ≠ 0 ∧
                 every path from ROOT to freelist uses edge (k,ℒk)}
                addleft(FREE, m[f].left);
                {Ifree ∧ Mgraph ∧ reachR(k) ∧ ℒFREE = ℒf ∧ ℒk = f ∧ ℛf = 0
                 every path from ROOT to freelist uses edge (f,ℒf)}
                m[f].left := 0
                {Mfree ∧ Mgraph ∧ reachR(k) ∧ ℒk = f ∧ ℒf = ℛf = 0}
            fi
            {Mfree ∧ Mgraph}
        od
```

4.5 Showing non-interference

We must show that the precondition of each statement S of the collector cannot be falsified by execution of the mutator, and vice versa. We must also show that the function f (see (4.2.1)) remains a decreasing function under parallel execution, in order to show that the marking phase of the collector still terminates. It will help to handle the assertions in separate classes: first, those assertions which deal only with the structure of the graph - like Ifree and reachR(k) - and secondly, those that also deal with the coloring of nodes.

Non-interference of assertions dealing only with structure

Note first that Ifree is true throughout execution of both processes. Now consider the collector (3.6). The only statements that change the graph structure occur in Append i to free list. Here, the successors of an unreachable node i are deleted and i is appended to the free list. Hence, the collector changes successors only of unreachable nodes, node ENDFREE, and the last free list node.

On the other hand, the mutator changes successors only of reachable nodes, and never of node ENDFREE or of the last free list node. The mutator and collector work with disjoint subsets of the nodes in this respect.

With this insight, we now scan the collector's proof outline and make a list of those assertions which are obviously not falsified by the mutator:

(4.5.1) Ifree, Cfree, NIL = ℒNIL = ℛNIL = 0
¬ reach(i), ¬ reach(i) ∧ ℒi = ℛi = 0,
ENDFREE = i, ℒENDFREE = i, mark, ¬ mark

In the same manner, we list the mutator assertions about structure which are not falsified by the collector:

(4.5.2) Ifree, Mfree, assertions dealing with the reachability of nodes and who their successors are, such as reach(k), reachR(k) $\wedge$ $\mathfrak{L}k = f \wedge \mathfrak{L}f = \mathfrak{R}f = 0$

We are able to handle the non-interference of these assertions informally, because the sets of nodes which each process can work with (with respect to graph structure) are well separated.

Non-interference of the other mutator preconditions

Three other mutator assertions must not be falsified by the collector, all dealing with the existence of black-to-white edges:

(4.5.3) Mgraph $\equiv$ add = 0 $\wedge$ (mark $\Rightarrow$ there is no black-to-white edge),
k = add $\neq$ 0 $\wedge$ (mark $\Rightarrow$ only possible black-to-white edge is $(k,\mathfrak{L}k)$,
k =-add $\neq$ 0 $\wedge$ (mark $\Rightarrow$ only possible black-to-white edge is $(k,\mathfrak{R}k)$.

We shall deal only with the first two. A scan of the collector yields the following three assignments, with relevant preconditions, which might falsify one of these assertions:

(4.5.4)

{no black nodes}	mark := true
{$\neg$ mark}	whiten(i)
{Cmark $\wedge$ C$\mathfrak{L}$ $\wedge$ C$\mathfrak{R}$}	blacken(i)

The only assignment where non-interference is not obvious is the third: blacken(i). Consider assertion Mgraph. We have

{Mgraph $\wedge$ Cmark $\wedge$ C$\mathfrak{L}$ $\wedge$ C$\mathfrak{R}$} $\Rightarrow$ {$\mathfrak{L}$i and $\mathfrak{R}$i are nonwhite}

Hence, blackening node i under these conditions leaves Mgraph true.
Next, consider the second assertion of (4.5.3). From this assertion and the precondition of blacken(i) we can prove that $\mathfrak{R}$i is nonwhite. Furthermore, if $\mathfrak{L}$i is white, then i = k and the edge $(i,\mathfrak{L}i)$ is the same edge as $(k,\mathfrak{L}k)$. Thus blackening i leaves this assertion true.

Non-interference of the other collector preconditions

We note that the only coloring action of the mutator is to gray a reachable white node. Hence the mutator cannot falsify the assertions: n nonblack, n black, n nonwhite, n gray, ¬ reach(n) ∧ n white, for any arbitrary node n. We now scan the collector and make a list of the remaining assertions which must not be interfered with (see (4.3) for definitions):

(4.5.5) Cmark
Cmark ∧ C$\mathfrak{L}$ (and similarly Cmark ∧ C$\mathfrak{R}$)
Cm(i) (for $0 \le i \le N+1$)
Ccoll(i) (for $-1 \le i \le N$; this includes the assertion "all white nodes are unreachable")
f does not increase (see (4.2.1) for the definition of f)

We will show non-interference only of the two interesting ones: Cmark, and Cmark ∧ C$\mathfrak{L}$. We note that

Mgraph ∧ roots nonwhite ∧ mark ⇒ Cmark

Since the mutator does not whiten nodes, it can falsify Cmark only by falsifying Mgraph also. Mgraph is false only in two places, in procedures addleft and addrite. We consider only one case. Suppose then that execution of the first await of addleft leaves a black-to-white edge (k,$\mathfrak{L}$k), under the conditions

{Mgraph ∧ reach(k) ∧ reach(j) ∧ k black ∧ j white ∧ Cmark}
await true then begin m[k].left := j ; add := k end

Consider a white, reachable node i before execution of the await. Since Cmark is true, there exists a path $(k_1,\ldots,k_p,i)$ where k_1 is gray and $k_2,\ldots,k_p$ are white. Since node k is black, execution of m[k].left := j cannot destroy this path. Hence i remains gray-reachable after executing the await. Thus Cmark is also true after execution of the await.

Now consider assertion Cmark ∧ C$\mathfrak{L}$. If node $\mathfrak{L}$i mentioned in C$\mathfrak{L}$ is white, then the mutator is currently between the two awaits of procedure addleft, because the mutator's variable add > 0. Execution of the second await grays node j , which in this case is node $\mathfrak{L}$i , and hence C$\mathfrak{L}$ remains true after execution of the second await.

On the other hand, suppose Cmark ∧ C$\mathfrak{L}$ is true, and that node $\mathfrak{L}$i is nonwhite. The mutator can falsify Cmark ∧ C$\mathfrak{L}$ only by making i's left successor white, and can do this only by changing i's left successor through execution of the first await statement in procedure addleft. For this to happen, the following must be true before execution of the await

$$\{\text{Cmark} \wedge C\mathfrak{L} \wedge \mathfrak{L}i \text{ nonwhite} \wedge \text{reach}(k) \wedge \text{reach}(j) \wedge j \text{ white} \wedge i = k\}$$

From this we can conclude that there is a path $(k_1,\ldots,k_p,j)$ where k_1 is gray and $k_2,\ldots,k_p$ are white. Further, if node i lies on this path, its successor on the path can not be $\mathfrak{L}$i, which is nonwhite, and must therefore be $\mathfrak{R}$i. This path is not disturbed by execution of the await, and is exactly the path described in C$\mathfrak{L}$ as the necessary condition if $\mathfrak{L}$i is white. Hence Cmark ∧ C$\mathfrak{L}$ is not falsified by execution of this statement either.

5. Concluding remarks

It took 12 typewritten pages to introduce the topic, describe the proof techniques, give the solution and describe it informally. Another 9 pages to present a more detailed, formal proof of correctness. The complexity of parallel programs with such a fine degree of interleaving seems to be an order of magnitude greater than the complexity of corresponding sequential programs. While another person might be able to improve on the style and presentation to make it appear simpler, I maintain that we must use such systematic methods to master and control the complexity. To support my view, thus far I have seen three purported solutions to this problem, either in print or ready to be submitted for publication. All used informal reasoning and had roughly the same mistake. Informal reasoning was just not adequate to handle the problem.

The main difficulty with the proof method used here is that assertions for one process have to be designed in the light of knowledge of the other processes. In the worst case, if one process has n statements and the other m statements, the proof method requires work proportional to m·n. I suspect that no general proof method for parallel programs can improve on this worst case bound.

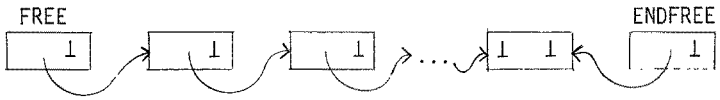
When we write a procedure to be used in a sequential setting, once written and proved correct we can view it as a black-box operation, and use it over and over again without having to look in the black box. We worry only about what it does. In a parallel setting, however, we must analyze the procedure each time we wish to use it, to make sure that the parallelism does not disturb its proof of correctness. And each change in the other process forces us to reanalyze the procedure again.

The only way to avoid this is to make the procedure an indivisible operation through the use of synchronization and mutual exclusion primitives.

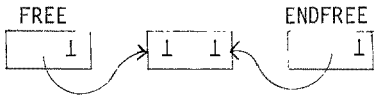
The on-the-fly garbage collector is very fragile and susceptible to such changes. Slight changes which would seem innocent in a sequential setting are disasterous in a parallel context. Two examples of this would prove enlightening.

First of all, consider a free list where ENDFREE is also a node and not just an index, perhaps described as in (5.1). (5.2) shows a free list consisting of a single node.

(5.1)



(5.2)



Suppose the free list always contains at least one element. The mutator could use either of the following two waiting tests before taking a node from the free list:

$$m[FREE].left \neq m[ENDFREE].left \quad \text{or} \quad m[m[FREE].left].left \neq 0$$

In sequential programming these might be equivalent, but in parallel programming the use of one or the other could lead to an error. The process which adds a node to the free list cannot change $\mathfrak{L}ENDFREE$ and $\mathfrak{L}(\mathfrak{L}ENDFREE)$ at the same time, so that only one of the following holds:

$$m[FREE].left \neq m[ENDFREE].left \Rightarrow m[m[FREE].left].left \neq 0$$

or

$$m[m[FREE].left].left \neq 0 \Rightarrow m[FREE].left \neq m[ENDFREE].left$$

Which one holds can only be determined by looking at the process which actually appends a node to the free list.

The second point illustrates the advantages of formal and systematic reasoning over informal reasoning. The last "bug" found in Dijkstra's program, which has appeared in three "solutions" that I have seen, was in the mutator's procedure addleft(k,j). This was first written as

```
atleastgray(j);   m[k].left := j
```

Informal reasoning had led to the conclusion that j should be grayed first, so that no black-to-white edge would exist. Only much later did informal reasoning by Mike Woodger find the error. Suppose the mutator is stopped after graying j . The collector begins marking and blackens k and j , begins collecting and whitens k and j , and then begins marking and blackens k . The collector pauses for a breath. The mutator executes m[k].left := j and then deletes all other edges ending in j. The only way to reach j is through the black-to-white edge (k,j). At this point, the collector finishes marking, leaving j white, and then collects the white reachable node j , appending it to the free list.

Informal reasoning alone can never hope to think of all such torturous execution sequences; only systematic method based on a formal method can expect to cope with the complexity. Using Owicki's techniques, the error would have been easy to find. In making up the list of mutator's assertionsthat may not be falsified we would have listed {mark ⇒ j nonwhite}. Clearly execution of the collector's statement mark := true can falsify this assertion.

Having been warned of this error by Dijkstra, I cannot argue that I found and corrected it myself. But I venture to say that if I had used Owicki's methods carefully enough (one makes errors in proofs too), I would have found it easily.

Of course, executing half of the sequence m[k].left := j ; atleastgray(j) leaves a black-to-white edge momentarily, and it was Dijkstra who led me to use instead the invariant

{every reachable white node is reachable from a gray node}

in the collector. The assertion "there are no black-to-white edges" had to be weakened because of the mutator's actions.

This exercise has convinced me that first building a program with little regard to correctness, and then debugging it to find errors, is even more folly for parallel programs than it is for sequential programs. In either case, it is not sufficient. The goal of every programmer should be to make the detection of an error during testing the exception rather than the rule, as it is now. This means that the more complicated the problem and resulting program, the more systematically and formally the problem must be investigated in order to control the complexity.

References

1. Dijkstra, E.W., et al. On-the-fly garbage collection: an exercise in co-operation. In this volume.

2. Hoare, C.A.R. An axiomatic basis for computer programming. CACM 12 (Oct 1969), 576-583.

3. Owicki, S. Axiomatic Proof Techniques for Parallel Programs. TR 75-251, Dept. of Computer Science, Cornell University, July 1975 (PhD thesis).

4. Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. CACM 18 (Aug 1975), 453-457

P. Brinch Hansen

California Institute of Technology, Pasadena
USA

The Programming Language Concurrent Pascal

Copyright 1975 by the Institute of Electrical and Electronic Engineers, Inc.
Reprinted with permission from IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 199-2o7.

THE PROGRAMMING LANGUAGE CONCURRENT PASCAL

Per Brinch Hansen

Information Science
California Institute of Technology

February 1975

Key Words and Phrases: structured multiprogramming, concurrent programming languages, hierarchical operating systems, concurrent processes, monitors, classes, abstract data types, access rights, scheduling.

Abstract

The paper describes a new programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent programming tools called processes and monitors. Part 1 of the paper explains these concepts informally by means of pictures illustrating a hierarchical design of a simple spooling system. Part 2 uses the same example to introduce the language notation. The main contribution of Concurrent Pascal is to extend the monitor concept with an explicit hierarchy of access rights to shared data structures that can be stated in the program text and checked by a compiler.

1. THE PURPOSE OF CONCURRENT PASCAL

1.1. BACKGROUND

Since 1972 I have been working on a new programming language for structured programming of computer operating systems. This language is called Concurrent Pascal. It extends the sequential programming language Pascal with concurrent programming tools called processes and monitors [1, 2, 3].

This is an informal description of Concurrent Pascal. It uses examples, pictures, and words to bring out the creative aspects of new programming concepts without getting into their finer details. I plan to define these concepts precisely and introduce a notation for them in later papers. This form of presentation may be imprecise from a formal point of view, but is perhaps more effective from a human point of view.

1.2. PROCESSES

We will study concurrent processes inside an operating system and look at one small problem only: How can large amounts of data be transmitted from one process to another by means of a buffer stored on a disk ?

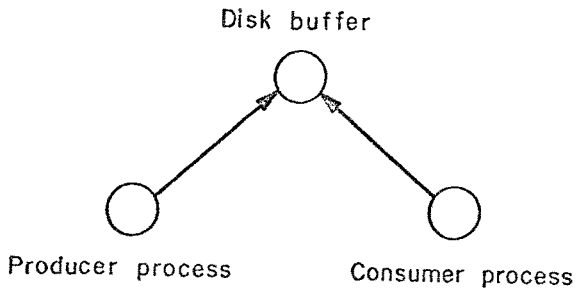


Fig. 1. Process communication

Figure 1 shows this little system and its three components: a process that produces data, a process that consumes data, and a disk buffer that connects them.

The circles are system components and the arrows are the access rights of these components. They show that both processes can use the buffer (but they do not show that data flows from the producer to the consumer.) This kind of picture is an access graph.

The next picture shows a process component in more detail (Fig. 2).

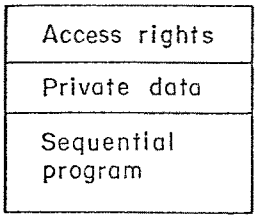


Fig. 2. A process

A process consists of a private data structure and a sequential program that can operate on the data. One process cannot operate on the private data of another process. But concurrent processes can share certain data structures (such as a disk buffer). The access rights of a process mention the shared data it can operate on.

1.3. MONITORS

A disk buffer is a data structure shared by two concurrent processes. The details of how such a buffer is constructed are irrelevant to its users. All the processes need to know is that they can send and receive data through it. If they try to operate on the buffer in any other way it is probably either a programming mistake or an example of tricky programming. In both case, one would like a compiler to detect such misuse of a shared data structure.

To make this possible, we must introduce a language construct that will enable a programmer to tell a compiler how a shared data structure can be used by processes. This kind of system component is called a monitor. A monitor can synchronize concurrent processes and transmit data between them. It can also control the order in which competing processes use shared, physical resources. Figure 3 shows a monitor in detail.

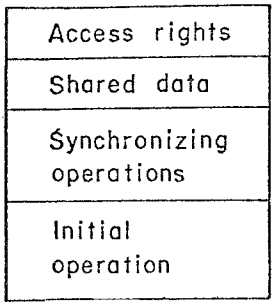


Fig. 3. A monitor

A monitor defines a shared data structure and all the operations processes can perform on it. These synchronizing operations are called monitor procedures. A monitor also defines an initial operation that will be executed when its data structure is created.

We can define a disk buffer as a monitor. Within this monitor there will be shared variables that define the location and length of the buffer on the disk. There will also be two monitor procedures, send and receive. The initial operation will make sure that the buffer starts as an empty one.

Processes cannot operate directly on shared data. They can only call monitor procedures that have access to shared data. A monitor procedure is executed as part of a calling process (just like any other procedure).

If concurrent processes simultaneously call monitor procedures that operate on the same shared data these procedures must be executed strictly one at a time. Otherwise, the results of monitor calls will be unpredictable. This means that the machine must be able to delay processes for short periods of time until it is their turn to execute monitor procedures. We will not be concerned about how this is done, but will just notice that a monitor procedure has exclusive access to shared data while it is being executed.

So the (virtual) machine on which concurrent programs run will handle short-term scheduling of simultaneous monitor calls. But the programmer must also be able to delay processes for longer periods of time if their requests for data and other resources cannot be satisfied immediately. If, for example, a process tries to receive data from an empty disk buffer it must be delayed until another process sends more data.

Concurrent Pascal includes a simple data type, called a queue, that can be used by monitor procedures to control medium-term scheduling of processes. A monitor can either delay a calling process in a queue or continue another process that is waiting in a queue. It is not important here to understand how these queues work except for the following essential rule: A process only has exclusive access to shared data as long as it continues to execute statements within a monitor procedure. As soon as a process is delayed in a queue it loses its exclusive access until another process calls the same monitor and wakes it up again. (Without this rule, it would be impossible for other processes to enter a monitor and let waiting processes continue their execution.)

Although the disk buffer example does not show this yet, monitor procedures should also be able to call procedures defined within other monitors. Otherwise, the language will not be very useful for hierarchical design. In the case of a disk buffer, one of these other monitors could perhaps define simple

input/output operations on the disk. So a monitor can also have access rights to other system components (see Fig. 3).

1.4. SYSTEM DESIGN

A process executes a sequential program - it is an active component. A monitor is just a collection of procedures that do nothing until they are called by processes - it is a passive component. But there are strong similarities between a process and a monitor: both define a data structure (private or shared) and the meaningful operations on it. The main difference between processes and monitors is the way they are scheduled for execution.

It seems natural therefore to regard processes and monitors as abstract data types defined in terms of the operations one can perform on them. If a compiler can check that these operations are the only ones carried out on the data structures then we may be able to build very reliable, concurrent programs in which controlled access to data and physical resources is guaranteed before these programs are put into operation. We have then to some extent solved the resource protection problem in the cheapest possible manner (without hardware mechanisms and run time overhead).

So we will define processes and monitors as data types and make it possible to use several instances of the same component type in a system. We can, for example, use two disk buffers to build a spooling system with an input process, a job process, and an output process (Fig. 4). I will distinguish between definitions and instances of components by calling them system types and system components. Access graphs (such as Fig. 4) will always show system components (not system types).

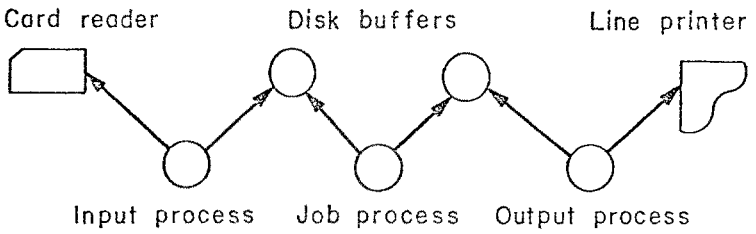


Fig. 4. A spooling system

Peripheral devices are considered to be monitors implemented in hardware. They can only be accessed by a single procedure io that delays the calling process until an input/output operation is completed. Interrupts are handled by the virtual machine on which processes run.

To make the programming language useful for stepwise system design it should permit the division of a system type, such as a disk buffer, into smaller system types. One of these other system types should give a disk buffer access to the disk. We will call this system type a virtual disk. It gives a disk buffer the illusion that it has its own private disk. A virtual disk hides the details of disk input/output from the rest of the system and makes the disk look like a data structure (an array of disk pages). The only operations on this data structure are read and write a page.

Each virtual disk is only used by a single disk buffer (Fig. 5). A system component that cannot be called simultaneously by several other components will be called a class. A class defines a data structure and the possible operations on it (just like a monitor).

The exclusive access of class procedures to class variables can be guaranteed completely at compile time. The virtual machine does not have to schedule simultaneous calls of class procedures at run time, because such calls cannot occur. This makes class calls considerably faster than monitor calls.

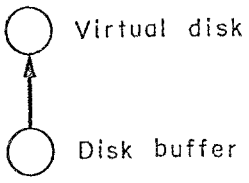


Fig. 5. Buffer refinement

The spooling system includes two virtual disks but only one real disk. So we need a single disk resource monitor to control the order in which competing processes use the disk (Fig. 6). This monitor defines two procedures, request and release access, to be called by a virtual disk before and after each disk transfer.

It would seem simpler to replace the virtual disks and the disk resource by a single monitor that has exclusive access to the disk and does the input/output. This would certainly guarantee that processes use the disk one at a time. But this would be done according to the built-in short-term scheduling policy of monitor calls.

Now to make a virtual machine efficient, one must use a very simple short-term scheduling rule (such as first-come, first-served) [2]. If the disk has a moving access head this is about the worst possible algorithm one can use for disk transfers. It is vital that the language make it possible for the programmer to write a medium-term scheduling algorithm that will minimize disk head movements [3]. The data type queue mentioned earlier makes it possible to implement arbitrary scheduling rules within

a monitor.

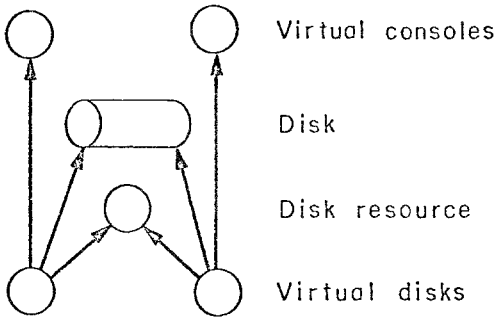


Fig. 6. Decomposition of virtual disks

The difficulty is that while a monitor is performing an input/output operation it is impos ible for other processes to enter the same monitor and join the disk queue. They will automatically be delayed by the short-term scheduler and only allowed to enter the monitor one at a time after each disk transfer. This will, of course, make the attempt to control disk scheduling within the monitor illusory. To give the programmer complete control of disk scheduling, processes should be able to enter the disk queue during disk transfers. Since arrival and service in the disk queueing system potentially are simultaneous operations they must be handled by different system components as shown in Fig. 6.

If the disk fails persistently during input/output this should be reported on an operator's console. Figure 6 shows two instances of a class type, called a virtual console. They give the virtual disks the illusion that they have their own private consoles.

The virtual consoles get exclusive access to a single, real console by calling a console resource monitor (Fig. 7). Notice that we now have a standard technique for dealing with virtual devices.

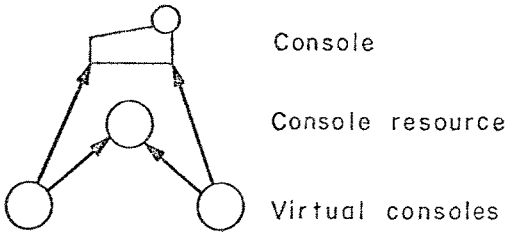


Fig. 7. Decomposition of virtual consoles

If we put all these system components together, we get a complete picture of a simple spooling system (Fig. 8). Classes, monitors, and processes are marked C, M, and P.

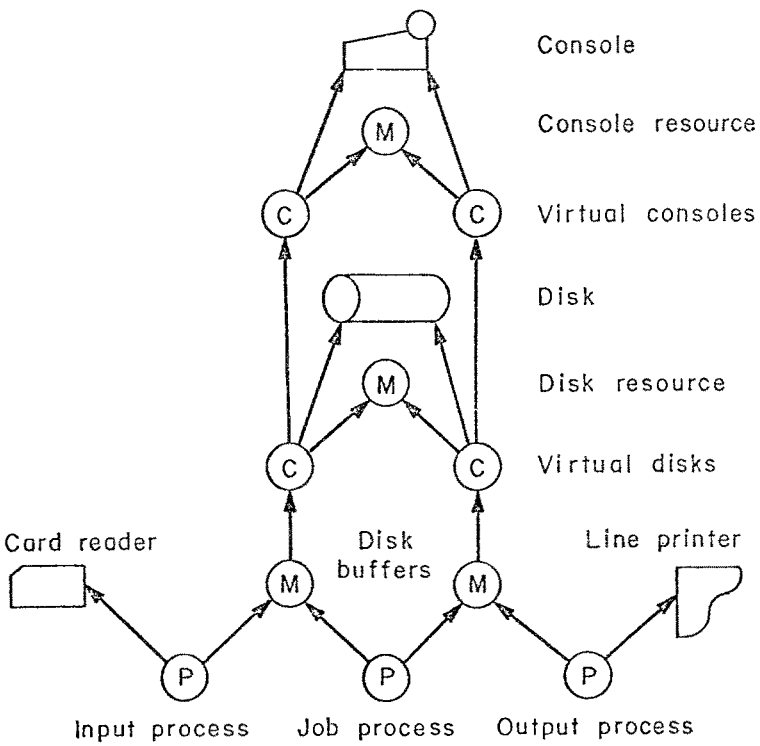


Fig. 8. Hierarchical system structure

1.5. SCOPE RULES

Some years ago I was part of a team that built a multiprogramming system in which processes can appear and disappear dynamically [4]. In practice, this system was used mostly to set up a fixed configuration of processes. Dynamic process deletion will certainly complicate the semantics and implementation of a programming language considerably. And since it appears to be unnecessary for a large class of real-time applications, it seems wise to exclude it altogether. So an operating system written in Concurrent Pascal will consist of a fixed number of processes, monitors, and classes. These components and their data structures will exist forever after system initialization. An operating system can, however, be extended by recompilation. It remains to be seen whether this restriction will simplify or complicate operating system design. But the poor quality of most existing operating systems clearly demonstrates an urgent need for simpler approaches.

In existing programming languages the data structures of processes, monitors, and classes would be called "global data". This term would be misleading in Concurrent Pascal where each data structure can be accessed by a single component only. It seems more appropriate to call them permanent data structures.

I have argued elsewhere that the most dangerous aspect of concurrent programming is the possibility of time-dependent programming errors that are impossible to locate by program testing ("lurking bugs") [2, 5, 6]. If we are going to depend on real-time programming systems in our daily lives, we must be able to find such obscure errors before the systems are put into operation.

Fortunately, a compiler can detect many of these errors if processes and monitors are represented by a structured notation in a high-level programming language. In addition, we must exclude low-level machine features (registers, addresses, and interrupts) from the language and let a virtual machine control them. If we want real-time systems to be highly reliable, we

must stop programming them in assembly language. (The use of hardware protection mechanisms is merely an expensive, inadequate way of making arbitrary machine language programs behave almost as predictably as compiled programs.)

A Concurrent Pascal compiler will check that the private data of a process only are accessed by that process. It will also check that the data structure of a class or monitor only is accessed by its procedures.

Figure 8 shows that access rights within an operating system normally are not tree-structured. Instead they form a directed graph. This partly explains why the traditional scope rules of block structured languages are inconvenient for concurrent programming (and for sequential programming as well). In Concurrent Pascal one can state the access rights of components in the program text and have them checked by a compiler.

Since the execution of a monitor procedure will delay the execution of further calls of the same monitor, we must prevent a monitor from calling itself recursively. Otherwise, processes can become deadlocked. So the compiler will check that the access rights of system components are hierarchically ordered (or, if you like, that there are no cycles in the access graph).

The hierarchical ordering of system components has vital consequences for system design and testing [7]:

A hierarchical operating system will be tested component by component, bottom up (but could, of course, be conceived top down or by iteration). When an incomplete operating system has been shown to work correctly (by proof or testing), a compiler can ensure that this part of the system will continue to work correctly when new untested program components are added on top of it. Programming errors within new components cannot cause old components to fail because old components do not call new components, and new components only call old components through well-defined procedures that have already been tested.

(Strictly speaking, a compiler can only check that single monitor calls are made correctly; it cannot check sequences of monitor calls, for example whether a resource is always reserved before it is released. So one can only hope for compile time assurance of partial correctness.)

Several other reasons besides program correctness make a hierarchical structure attractive:

1) A hierarchical operating system can be studied in a stepwise manner as a sequence of abstract machines simulated by programs [8].

2) A partial ordering of process interactions permits one to use mathematical induction to prove certain overall properties of the system (such as the absence of deadlocks) [2].

3) Efficient resource utilization can be achieved by ordering the program components according to the speed of the physical resources they control (with the fastest resources being controlled at the bottom of the system) [8].

4) A hierarchical system designed according to the previous criteria is often nearly-decomposable from an analytical point of view. This means that one can develop stochastic models of its dynamic behavior in a stepwise manner [9].

1.6. FINAL REMARKS

It seems most natural to represent a hierarchical system structure, such as Fig. 8, by a two-dimensional picture. But when we write a concurrent program we must somehow represent these access rules by linear text. This limitation of written language tends to obscure the simplicity of the original structure. That is why I have tried to explain the purpose of Concurrent Pascal by means of pictures instead of language notation.

The class concept is a restricted form of the class concept of Simula 67 [10]. Dijkstra suggested the idea of monitors [8]. The first structured language notation for monitors was proposed in [2] and illustrated by examples in [3]. The queue variables needed by monitors for process scheduling were suggested in [5] and modified in [3].

The main contribution of Concurrent Pascal is to extend monitors with explicit access rights that can be checked at compile time. Concurrent Pascal has been implemented at Caltech for the PDP 11/45 computer. Our system uses sequential Pascal as a job control and user programming language.

2. THE USE OF CONCURRENT PASCAL

2.1. INTRODUCTION

In Part 1 the concepts of Concurrent Pascal were explained informally by means of pictures of a hierarchical spooling system. I will now use the same example to introduce the language notation of Concurrent Pascal. The presentation is still informal. I am neither trying to define the language precisely nor to develop a working system. This will be done in other papers. I am just trying to show the flavor of the language.

2.2. PROCESSES

We will now program the system components in Fig. 8 one at a time from top to bottom (but we could just as well do it bottom up).

Although we only need one input process, we may as well define it as a general system type of which several copies may exist:

```
type inputprocess =
process(buffer: diskbuffer);
var block: page;
cycle
  readcards(block);
  buffer.send(block);
end
```

An input process has access to a buffer of type diskbuffer (to be defined later). The process has a private variable block of type page. The data type page is declared elsewhere as an array of characters:

```
type page = array (.1..512.) of char
```

A process type defines a sequential program - in this case, an endless cycle that inputs a block from a card reader and sends it through the buffer to another process. We will ignore the details of card reader input.

The send operation on the buffer is called as follows (using the block as a parameter):

```
buffer.send(block)
```

The next component type we will define is a job process:

```
type jobprocess =
process(input, output: diskbuffer);
var block: page;
cycle
  input.receive(block);
  update(block);
  output.send(block);
end
```

A job process has access to two disk buffers called input and output. It receives blocks from one buffer, updates them, and sends them through the other buffer. The details of updating can be ignored here.

Finally, we need an output process that can receive data from a disk buffer and output them on a line printer:

```
type outputprocess =
process(buffer: diskbuffer);
var block: page;
cycle
  buffer.receive(block);
  printlines(block);
end
```

The following shows a declaration of the main system components

```
var buffer1, buffer2: diskbuffer;
    reader: inputprocess;
    master: jobprocess;
    writer: outputprocess;
```

There is an input process, called the reader, a job process, called the master, and an output process, called the writer. Then there are two disk buffers, buffer1 and buffer2, that connect them.

Later I will explain how a disk buffer is defined and initialized. If we assume that the disk buffers already have been initialized, we can initialize the input process as follows:

```
init reader(buffer1)
```

The init statement allocates space for the private variables of the reader process and starts its execution as a sequential

process with access to buffer1.

The access rights of a process to other system components, such as buffer1, are also called its parameters. A process can only be initialized once. After initialization, the parameters and private variables of a process exist forever. They are called permanent variables.

The init statement can be used to start concurrent execution of several processes and define their access rights. As an example, the statement

```
init reader(buffer1), master(buffer1, buffer2), writer(buffer2)
```

starts concurrent execution of the reader process (with access to buffer1), the master process (with access to both buffers), and the writer process (with access to buffer2).

A process can only access its own parameters and private variables. The latter are not accessible to other system components. Compare this with the more liberal scope rules of block structured languages in which a program block can access not only its own parameters and local variables but also those declared in outer blocks. In Concurrent Pascal, all variables accessible to a system component are declared within its type definition. This access rule and the init statement make it possible for a programmer to state access rights explicitly and have them checked by a compiler. They also make it possible to study a system type as a self-contained program unit.

Although the programming examples do not show this, one can also define constants, data types, and procedures within a process. These objects can only be used within the process type.

2.3. MONITORS

The disk buffer is a monitor type:

```
type diskbuffer =
monitor(consoleaccess, diskaccess: resource;
        base, limit: integer);
var disk: virtualdisk; sender, receiver: queue;
    head, tail, length: integer;

procedure entry send(block: page);
begin
  if length = limit then delay(sender);
  disk.write(base + tail, block);
  tail:= (tail + 1) mod limit;
  length:= length + 1;
  continue(receiver);
end;

procedure entry receive(var block: page);
begin
  if length = 0 then delay(receiver);
  disk.read(base + head, block);
  head:= (head + 1) mod limit;
  length:= length - 1;
  continue(sender);
end;

begin "initial statement"
  init disk(consoleaccess, diskaccess);
  head:= 0; tail:= 0; length:= 0;
end
```

A disk buffer has access to two other components, consoleaccess and diskaccess, of type resource (to be defined later). It also has access to two integer constants defining the base address and limit of the buffer on the disk.

The monitor declares a set of shared variables: The disk is declared as a variable of type virtualdisk. Two variables of type queue are used to delay the sender and receiver processes until the buffer becomes nonfull and nonempty. Three integers define the relative addresses of the head and tail elements of the buffer and its current length.

The monitor defines two monitor procedures, send and receive. They are marked with the word entry to distinguish them from local procedures used within the monitor (there are none of these in this example).

Receive returns a page to the calling process. If the buffer is empty, the calling process is delayed in the receiver queue until another process sends a page through the buffer. The receive procedure will then read and remove a page from the head of the disk buffer by calling a read operation defined within the virtual disk type:

```
disk.read(base + head, block)
```

Finally, the receive procedure will continue the execution of a sending process (if the latter is waiting in the sender queue).

Send is similar to receive.

The queuing mechanism will be explained in detail in the next section.

The initial statement of a disk buffer initializes its virtual disk with access to the console and disk resources. It also sets the buffer length to zero. (Notice, that a disk buffer does not use its access rights to the console and disk, but only passes them on to a virtual disk declared within it.)

The following shows a declaration of two system components of type resource and two integers defining the base and limit of a disk buffer:

```
var consoleaccess, diskaccess: resource;
    base, limit: integer;
    buffer: diskbuffer;
```

If we assume that these variables already have been initialized, we can initialize a disk buffer as follows:

```
init buffer(consoleaccess, diskaccess, base, limit)
```

The init statement allocates storage for the parameters and shared variables of the disk buffer and executes its initial statement.

A monitor can only be initialized once. After initialization, the parameters and shared variables of a monitor exist forever. They are called permanent variables. The parameters and local variables of a monitor procedure, however, exist only while it is being executed. They are called temporary variables.

A monitor procedure can only access its own temporary and permanent variables. These variables are not accessible to other system components. Other components can, however, call procedure entries within a monitor. While a monitor procedure is being executed, it has exclusive access to the permanent variables of the monitor. If concurrent processes try to call procedures within the same monitor simultaneously, these procedures will be executed strictly one at a time.

Only monitors and constants can be permanent parameters of processes and monitors. This rule ensures that processes only communicate by means of monitors.

It is possible to define constants, data types, and local procedures within monitors (and processes). The local procedures of a system type can only be called within the system type. To prevent deadlock of monitor calls and ensure that access rights are hierarchical the following rules are enforced: A procedure must be declared before it can be called; Procedure definitions cannot be nested and cannot call themselves; A system type

cannot call its own procedure entries.

The absence of recursion makes it possible for a compiler to determine the store requirements of all system components. This and the use of permanent components make it possible to use fixed store allocation on a computer that does not support paging.

Since system components are permanent they must be declared as permanent variables of other components.

2.4. QUEUES

A monitor procedure can delay a calling process for any length of time by executing a delay operation on a queue variable. Only one process at a time can wait in a queue. When a calling process is delayed by a monitor procedure it loses its exclusive access to the monitor variables until another process calls the same monitor and executes a continue operation on the queue in which the process is waiting.

The continue operation makes the calling process return from its monitor call. If any process is waiting in the selected queue, it will immediately resume the execution of the monitor procedure that delayed it. After being resumed, the process again has exclusive access to the permanent variables of the monitor.

Other variants of process queues (called "events" and "conditions") are proposed in [3, 5]. They are multi-process queues that use different (but fixed) scheduling rules. We do not yet know from experience which kind of queue will be the most convenient one for operating system design. A single-process queue is the simplest tool that gives the programmer complete control of the scheduling of individual processes. Later, I will show how multi-process queues can be built from single-process queues.

A queue must be declared as a permanent variable within a monitor type.

2.5. CLASSES

Every disk buffer has its own virtual disk. A virtual disk is defined as a class type:

```
type virtualdisk =
class(consoleaccess, diskaccess: resource);
var terminal: virtualconsole; peripheral: disk;

procedure entry read(pageno: integer; var block: page);
var error: boolean;
begin
  repeat
    diskaccess.request;
    peripheral.read(pageno, block, error);
    diskaccess.release;
    if error then terminal.write('disk failure');
  until not error;
end;

procedure entry write(pageno: integer; block: page);
begin "similar to read" end;

begin "initial statement"
  init terminal(consoleaccess), peripheral;
end
```

A virtual disk has access to a console resource and a disk resource. Its permanent variables define a virtual console and a disk. A process can access its virtual disk by means of read and write procedures. These procedure entries request and release exclusive access to the real disk before and after each block transfer. If the real disk fails the virtual disk calls its virtual console to report the error.

The initial statement of a virtual disk initializes its virtual console and the real disk.

Section 2.3 shows an example of how a virtual disk is declared and initialized (within a disk buffer).

A class can only be initialized once. After initialization, its parameters and private variables exist forever. A class procedure can only access its own temporary and permanent variables. These cannot be accessed by other components.

A class is a system component that cannot be called simultaneously by several other components. This is guaranteed by the following rule: A class must be declared as a permanent variable within a system type; A class can be passed as a permanent parameter to another class (but not to a process or monitor). So a chain of nested class calls can only be started by a single process or monitor. Consequently, it is not necessary to schedule simultaneous class calls at run time - they cannot occur.

2.6. INPUT/OUTPUT

The real disk is controlled by a class

```
type disk = class
```

with two procedure entries

```
read(pageno, block, error)
write(pageno, block, error)
```

The class uses a standard procedure

```
io(block, param, device)
```

to transfer a block to or from the disk device. The io parameter is a record

```
var param: record
             operation: iooperation;
             result: ioresult;
             pageno: integer
           end
```

that defines an input/output operation, its result, and a page number on the disk. The calling process is delayed until an io operation has been completed.

A virtual console is also defined as a class

```
type virtualconsole =
class(access: resource);
var terminal: console;
```

It can be accessed by read and write operations that are similar to each other:

```
procedure entry read(var text: line);
begin
  access.request;
  terminal.read(text);
  access.release;
end
```

The real console is controlled by a class that is similar to the disk class.

2.7. MULTIPROCESS SCHEDULING

Access to the console and disk is controlled by two monitors of type resource. To simplify the presentation, I will assume that competing processes are served in first-come, first-served order. (A much better disk scheduling algorithm is defined in [3]. It can be programmed in Concurrent Pascal as well but involves more details than the present one.)

We will define a multiprocess queue as an array of single-process queues

```
type multiqueue = array (.0..qlength-1.) of queue
```

where qlength is an upper bound on the number of concurrent processes in the system.

A first-come, first-served scheduler is now straightforward to program:

```
type resource =
monitor
var free: boolean; q: multiqueue;
    head, tail, length: integer;

procedure entry request;
var arrival: integer;
begin
  if free then free:= false else
  begin
    arrival:= tail;
    tail:= (tail + 1) mod qlength;
    length:= length + 1;
    delay(q(.arrival.));
  end;
end;
```

```
procedure entry release;
var departure: integer;
begin
  if length = 0 then free:= true else
  begin
    departure:= head;
    head:= (head + 1) mod qlength;
    length:= length - 1;
    continue(q(.departure.));
  end;
end;

begin "initial statement"
  free:= true; length:= 0;
  head:= 0; tail:= 0;
end
```

2.8. INITIAL PROCESS

Finally, we will put all these components together into a concurrent program. A Concurrent Pascal program consists of nested definitions of system types. The outermost system type is an anonymous process, called the initial process. An instance of this process is created during system loading. It initializes the other system components.

The initial process defines system types and instances of them. It executes statements that initialize these system components. In our example, the initial process can be sketched as follows (ignoring the problem of how base addresses and limits of disk buffers are defined):

```
type
  resource = monitor ... end;
  console = class ... end;
  virtualconsole =
    class(access: resource); ... end;
  disk = class ... end;
  virtualdisk =
    class(consoleaccess, diskaccess: resource); ... end;
  diskbuffer =
    monitor(consoleaccess, diskaccess: resource;
            base, limit: integer); ... end;
  inputprocess =
    process(buffer: diskbuffer); ... end;
  jobprocess =
    process(input, output: diskbuffer); ... end;
  outputprocess =
    process(buffer: diskbuffer); ... end;
var
  consoleaccess, diskaccess: resource;
  buffer1, buffer2: diskbuffer;
  reader: inputprocess;
  master: jobprocess;
  writer: outputprocess;
begin
  init consoleaccess, diskaccess,
    buffer1(consoleaccess, diskaccess, base1, limit1),
    buffer2(consoleaccess, diskaccess, base2, limit2),
    reader(buffer1),
    master(buffer1, buffer2),
    writer(buffer2);
end.
```

When the execution of a process (such as the initial process) terminates, its private variables continue to exist. This is necessary because these variables may have been passed as permanent parameters to other system components.

Acknowledgements

It is a pleasure to acknowledge the immense value of a continuous exchange of ideas with Tony Hoare on structured multiprogramming. I also thank my students Luis Medina and Ramon Varela for their helpful comments on this paper.

The project is now supported by the National Science Foundation under grant number DCR74-17331.

References

1. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

2. Brinch Hansen, P. Operating system principles. Prentice-Hall, July 1973.

3. Hoare, C. A. R. Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-57.

4. Brinch Hansen, P. The nucleus of a multiprogramming system, Comm. ACM 13, 4 (Apr. 1970), 238-50.

5. Brinch Hansen, P. Structured multiprogramming. Comm. ACM 15, 7 (July 1972), 574-78.

6. Brinch Hansen, P. Concurrent programming concepts. ACM Computing Reviews 5, 4 (Dec. 1974), 223-45.

7. Brinch Hansen, P. A programming methodology for operating system design. Proc. IFIP 74 Congress, (Aug. 1974), 394-97.

8. Dijkstra, E. W. Hierarchical ordering of sequential processes. Acta Informatica 1, 2 (1971), 115-38.

9. Simon, H. A. The architecture of complexity. Proc. American Philosophical Society 106, 6 (1962), 468-82.

10. Dahl, O.-J., and Hoare, C. A. R. Hierarchical program structures. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured programming. Academic Press, 1972.

CHAPTER 2.: PROGRAM DEVELOPMENT

Guarded commands, non-determinacy and a calculus for the derivation of programs.

by Edsger W.Dijkstra *)

*) Author's address: BURROUGHS
Plataanstraat 5
NUENEN - 4565
The Netherlands.

Abstract. So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow non-deterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be shown.

Keywords. programming languages, sequencing primitives, program semantics, programming language semantics, non-determinacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology.

CR-category: 4.20, 4.22.

Guarded commands, non-determinacy and a calculus for the derivation of programs.

1. Introduction.

In section 2, two statements, an alternative construct and a repetitive construct will be introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called "guarded command", a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential non-determinacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact tha

now programs can be derived in a more systematic manner than before.

In section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs will be given, together with two theorems for each of the constructs (without proof).

In section 4, it will be shown, how upon the above a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present "an algorithm" for the derivation of programs: we have used the term "a calculus" for a formal discipline --a set of rules-- such that, if applied successfully

1) it will have derived a correct program
2) it will tell us that we have reached such a goal.

(In choosing the term "calculus" we have been inspired by the "integral calculus" and the "propositional calculus" where we have a very similar situation.)

2. Two statements made from guarded commands.

If the reader accepts "other statements" as indicating, say, assignment statements and procedure calls, we can give the relevant syntax in BNF [2]. In the following we have extended BNF with the convention that the braces "{...}" should be read as: "followed by zero or more instances of the enclosed".

< guarded command > ::= < guard > → < guarded list >
< guard > ::= < boolean expression >
< guarded list > ::= < statement > {; < statement >}
< guarded command set > ::= < guarded command > {[] < guarded command >}
< alternative construct > ::= **if** < guarded command set > **fi**
< repetitive construct > ::= **do** < guarded command set > **od**
< statement > ::= < alternative construct > | < repetitive construct >
"other statements" .

The semicolons in the guarded list have the usual meaning: when the guarded list is selected for execution its statements will be executed successively in the order from left to right; a guarded list will only be

selected for execution in a state such that its guard is true. Note that a guarded command by itself is not a statement: it is component of a guarded command set from which statements can be constructed. If the guarded command set consists of more than one guarded command, they are mutually separated by the separator "[]" ; our text as then an arbitrarily ordered enumeration of an unordered set, i.e. the order in which the guarded commands of a set appear in our text is semantically irrelevant.

Our syntax gives two ways for constructing a statement out of a guarded command set. The alternative construct is written by enclosing it by the special bracket pair: "if ... fi". If in the initial state none of the guards is true, the program will abort, otherwise an arbitrary guarded list with a true guard will be selected for execution.

Note. If the empty guarded command set were allowed "if fi" would be semantically equivalent to "abort" . (End of note.)

An example --illustrating the non-determinacy in a very modest fashion-- would be the program that for fixed x and y assigns to m the maximum value of x and y :

```
if x ≥ y → m:= x
 [] y ≥ x → m:= y
fi      .
```

The repetitive construct is written down by enclosing a guarded command set by the special bracket pair "do ... od" . Here a state in which none of the guards is true will not lead to abortion but to proper termination; the complementary rule, however, is that it will only terminate in a state in which none of the guards is true: when initially or upon completed execution of a selected guarded list one or more guards are true, a new selection for execution of a guarded list with a true guard will take place, and so on. When the repetitive construct has terminated properly, we know that all its guards are false.

Note. If the empty guarded command set were allowed "do od" would be semantically equivalent to "skip" . (End of note.)

An example --showing the non-determinacy in somewhat greater glory-- is the program that assigns to the variables q1, q2, q3 and q4 a permutation of the values Q1, Q2, Q3 and Q4, such that $q1 \leq q2 \leq q3 \leq q4$. Using concurrent assignment statements for the sake of convenience, we can program

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do q1 > q2 → q1, q2 := q2, q1
 [] q2 > q3 → q2, q3 := q3, q2
 [] q3 > q4 → q3, q4 := q4, q3
od
```

To conclude this section we give a program where not only the computati but also the final state is not necessarily uniquely determined. The program should determine k such that for fixed value n $(n > 0)$ and a fixed function $f(i)$ defined for $0 \leq i < n$, k will eventually satisfy:

$$0 \leq k < n \ \underline{\text{and}} \ (\forall i: 0 \leq i < n: f(k) \geq f(i)) \quad .$$

(Eventually k should be the place of a maximum.)

```
k:= 0; j:= 1;
do j ≠ n → if f(j) ≤ f(k) → j:= j + 1
           [] f(j) ≥ f(k) → k:= j; j:= j + 1
           fi
od .
```

Only permissible final states are possible and each permissible final state is possible.

3. Formal definition of the semantics.

3.1. Notational prelude.

In the following sections we shall use the symbols P , Q and R to denote (predicates defining) boolean functions defined on all points of the state space; alternatively we shall refer to them as "conditions", satisfied by all states for which the boolean function is true. Two special predicates that we denote by the reserved names "T" and "F" play a special role: T denotes the condition that, by definition, is satisfied by all states, F denotes, by definition, the condition that is satisfied

by no state at all.

The way in which we use predicates (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by Hoare [1], the main difference being that we have tightened things up a bit: while Hoare introduces sufficient pre-conditions such that the mechanisms will not produce the wrong result (but may fail to terminate), we shall introduce necessary and sufficient --i.e. so-called "weakest"-- pre-conditions such that the mechanisms are guaranteed to produce the right result.

More specifically: we shall use the notation "wp(S, R)" , where S denotes a statement list and R some condition on the state of the system, to denote the weakest pre-condition for the initial state of the system such that activation of S is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the post-condition R . Such a "wp" --which is called "a predicate transformer", because it associates a pre-condition to any post-condition R -- has, by definition, the following properties.

1) For any S , we have for all states

$$wp(S, F) = F$$

(the so-called "Law of the Excluded Miracle").

2) For any S and any two post-conditions, such that for all states

$$P \Rightarrow Q$$

we have for all states

$$wp(S, P) \Rightarrow wp(S, Q) \quad .$$

3) For any S and any two post-conditions P and Q we have for all states

$$(wp(S, P) \text{ and } wp(S, Q)) = wp(S, P \text{ and } Q) \quad .$$

4) For any S and any two post-conditions P and Q we have for all states

$$(wp(S, P) \text{ or } wp(S, Q)) \Rightarrow wp(S, P \text{ or } Q) \quad .$$

Together with the rules of propositional calculus and the semantic definitions to be given below, the above four properties take over the

role of the "rules of inference" as introduced by Hoare [1].

We take the position that we know the semantics of a mechanism S sufficiently well if we know its predicate transformer, i.e. can derive wp(S, R) for any post-condition R .

Note. This position is taken in full acknowledgement of the fact that in the case of non-deterministic mechanisms, the knowledge of the predicate transformer does not give a complete description: for those initial states that do not necessarily lead to a properly terminating activity, the knowledge of the predicate transformer does not give us any information about the final states in which the system might find itself after proper termination. (End of note.)

Example 1. The semantics of the empty statement, denoted by "skip", are given by the definition that for any post-condition R , we have

$$wp(\text{"skip"}, R) = R \quad .$$

Example 2. The semantics of the assignment statement "x:= E" are given by

$$wp(\text{"x:= E"}, R) = R^x_E \quad ,$$

in which R^x_E denotes a copy of the predicate defining R in which each occurence of the variable "x" is replaced by "(E)".

Example 3. The semantics of the semicolon ";" as concatenation operator are given by

$$wp(\text{"S1; S2"}, R) = wp(S1, wp(S2, R)) \quad .$$

3.2. The alternative construct.

In order to define the semantics of the alternative construct we define two abbreviations.

Let "IF" denote $\underline{\text{if}}\ B_1 \rightarrow SL_1 \ [] \ \ldots \ [] \ B_n \rightarrow SL_n\ \underline{\text{fi}}$;

let "BB" denote $(\exists i: 1 \leq i \leq n: B_i)$;

then, by definition

$$wp(IF, R) = (BB \text{ and } (\forall i: 1 \leq i \leq n: B_i \Rightarrow wp(SL_i, R)) \quad .$$

(The first term "BB" requires that the alternative construct as such will not lead to abortion on account of all guards false, the second term requires that each guarded list eligible for execution will lead to an acceptable final state.) From this definition we can derive --by simple substitutions--

Theorem 1. From

$$(\forall i: 1 \leq i \leq n: (Q \text{ and } B_i) \Rightarrow wp(SL_i, R)) \quad \text{for all states}$$

we can conclude that

$$(Q \text{ and } BB) \Rightarrow wp(IF, R) \quad \text{holds for all states} \; .$$

Let "t" denote some integer function, defined on the state space, and let "wdec(S, t)" denote the weakest pre-condition such that activatio of S is guaranteed to lead to a properly terminating activity leaving the system in a final state such that the value of t is decreased by at least 1 (compared to its initial value). In terms of "wdec" we can formulate the very similar

Theorem 2. From

$$(\forall i: 1 \leq i \leq n: (Q \text{ and } B_i) \Rightarrow wdec(SL_i, t)) \quad \text{for all states}$$

we can conlude that

$$(Q \text{ and } BB) \Rightarrow wdec(IF, t) \quad \text{holds for all states.}$$

Note (which can be skipped at first reading). The relation between "wp" and "wdec" is as follows. For any point X in state space we can regard

$$wp(S, t \leq t_0)$$

as an equation with t_0 as the unknown. Let its smallest solution for t_0 be tmin(X). (Here we have added the explicit dependence on the state X.) Then tmin(X) can be interpreted as the lowest upper bound for the final value of t if the mechanism S is activated with X as initial state. Then, by definition,

$$wdec(S, t) = (tmin(X) \leq t(X) - 1) = (tmin(X) < t(X)) \quad .$$

(End of note.)

3.3. The repetitive construct.

As is to be expected, the definition of the repetitive construct

$$\underline{\text{do}}\ B_1 \rightarrow SL_1\ [\!]\ \ldots\ [\!]\ B_n \rightarrow SL_n\ \underline{\text{od}}$$

that we denote by "DO" , is more complicated.

Let $H_0(R) = (R\ \underline{\text{and}}\ \underline{\text{non}}\ BB)$

and for $k > 0$: $H_k(R) = (wp(IF, H_{k-1}(R))\ \underline{\text{or}}\ H_0(R))$

(where "IF" denotes the same guarded command set enclosed by "if fi")

then, by definition

$$wp(DO, R) = (\exists k: k \geq 0: H_k(R)) \quad .$$

(Intuitively, $H_k(R)$ can be interpreted as the weakest pre-condition guaranteeing proper termination after at most k selections of a guarded list, leaving the system in a final state satisfying R .) Via mathematical induction we can prove

Theorem 3. From

$$(P\ \underline{\text{and}}\ BB) \Rightarrow (wp(IF, P)\ \underline{\text{and}}\ wdec(IF, t)) \quad \text{for all states}$$

and $$P \Rightarrow (t \geq 0) \quad \text{for all states}$$

we can conclude that we have for all states

$$P \Rightarrow wp(DO, P\ \underline{\text{and}}\ \underline{\text{non}}\ BB) \quad .$$

Note that the antecedent of Theorem 3 is of the form of the consequents of Theorems 1 and 2.

Because T is the condition by definition satisfied by all states, $wp(S, T)$ is the weakest pre-condition guaranteeing proper termination for S . This allows us to formulate an alternative theorem about the repetitive construct, viz.

Theorem 4. From

$$(P\ \underline{\text{and}}\ BB) \Rightarrow wp(IF, P) \quad \text{for all states,}$$

we can conclude that we have for all states

$$(P\ \underline{\text{and}}\ wp(DO, T)) \Rightarrow wp(DO, P\ \underline{\text{and}}\ \underline{\text{non}}\ BB) \quad .$$

In connection with the above theorems "P" is called "the invariant relation

and "t" is called "the variant function".

4. Formal derivation of programs.

The formal requirement of our program performing "m:= max(x, y)" --see above-- is that for fixed x and y it establishes the relation

R: $(m = x \text{ or } m = y) \text{ and } m \geq x \text{ and } m \geq y$.

Now the Axiom of Assignment tells us that "m:= x" is the standard way of establishing the truth of "m = x" for fixed x, which is a way of establishing the truth of the first term of R. Will "m:= x" do the job? In order to investigate this, we derive and simplify

$$\begin{aligned} wp(\text{"m:= x"}, R) &= (x = x \text{ or } x = y) \text{ and } x \geq x \text{ and } x \geq y \\ &= x \geq y \quad . \end{aligned}$$

Taking this weakest pre-condition as its guard, Theorem 1 tell us that

$$\text{if } x \geq y \rightarrow m := x \text{ fi}$$

will produce the correct result if it terminates succesfully. The disadvantage of this program is that $BB \neq T$, i.e. it might lead to abortion; weakening BB means looking for alternatives which might introduce new guards. The obvious alternative is the assignment "m:= y" with the guard

$$wp(\text{"m:= y"}, R) = y \geq x \quad ;$$

thus we are led to our program

$$\begin{aligned} &\text{if } x \geq y \rightarrow m := x \\ &\ [] \ y \geq x \rightarrow m := y \\ &\text{fi} \end{aligned}$$

and by this time $BB = T$ and therefore we have solved the problem. (In the mean time we have proved that the maximum of two values is always defined, viz. that R considered as equation for m has always a solution.)

As an example of the deriviation of a repetitive construct we shall derive a program for the greatest common divisor of two positive numbers, i.e. for fixed, positive X and Y we have to establish the final relatio

$$x = \gcd(X, Y) \quad .$$

The formal machinery only gets in motion, once we have chosen our invariant relation and our variant function. The program then gets the structure

"establish the relation P to be kept invariant";
do "decrease t as long as possible under invariance of P" od .

Suppose that we choose for the invariant relation

P: $\gcd(X, Y) = \gcd(x, y)$ and $x > 0$ and $y > 0$

a relation that has the advantage of being easily established by

$$x := X;\ y := Y \quad .$$

The most general "something" to be done under invariance of P is of the form

$$x, y := E1, E2$$

and we are interested in a guard B such that

$$(P \text{ and } B) \Rightarrow wp("x, y := E1, E2", P)$$
$$= (\gcd(X, Y) = \gcd(E1, E2) \text{ and } E1 > 0 \text{ and } E2 > 0) \quad .$$

Because the guard must be a computable boolean expression and should not contain the computation of $\gcd(X, Y)$ --for that was the whole problem! we must see to it that the expressions E1 and E2 are so chosen, that the first term

$$\gcd(X, Y) = \gcd(E1, E2)$$

is implied by P , which is true if

$$\gcd(x, y) = \gcd(E1, E2) \quad .$$

In other words we are invited to massage the value pair (x, y) in such a fashion that their gcd is not changed. Because --and this is the place where to mobilize our mathematical knowledge about the gcd-function--

$$\gcd(x, y) = \gcd(x - y, y)$$

a possible guarded list would be

$$x := x - y \quad .$$

Deriving

$$wp("x:= x - y", R) = (gcd(X, Y) = gcd(x - y, y) \ \underline{and} \ x - y > 0 \ \underline{and} \ y > 0)$$

and omitting all terms of the conjunction implied by P we find the guard

$$x > y$$

as far as the invariance of P is concerned. Besides that we must require guaranteed decrease of the variant function t . Let us investigate the consequences of the choice

$$t = x + y \quad .$$

From

$$wp("x:= x - y", t \leq t_0) =$$
$$wp("x:= x - y", x + y \leq t_0) = (x \leq t_0)$$

we conclude that

$$tmin = x \quad ;$$

therefore

$$wdec("x:= x - y", t) = (x < x + y) = (y > 0) \quad .$$

The requirement of monotonic decrease of t imposes no further restriction of the guard because wdec("x:= x - y", t) is fully implied by P and we come at our first effort

```
x:= X; y:= Y;
do x > y → x:= x - y od .
```

Alas, this single guard is insufficient: from P and non BB we are not allowed to conclude x = gcd(X, Y). In a completely analogous manner, the alternative y:= y - x will require as its guard y > x and our next effort is

```
x:= X; y:= Y;
do x > y → x:= x - y
 [] y > x → y:= y - x
od                    .
```

Now the job is done, because with this last program non BB = (x = y) and (P and x = y) => (x = gcd(X, Y) because gdc(x, x) = x .

Note. The choice of t = x +2y and the knowledge of the fact that the gcd is a symmetric function could have led to the program

```
x:= X; y:= Y;
do x > y → x:= x - y
 [] y > x → x, y := y, x
od                    .
```

The swap "x, y := y, x" can never destroy P : the guard of the last guarded list is fully caused by the requirement that t is effectively decreased.

In both cases the final game has been to find a large enough set of such guarded lists that BB , the disjunction of their guards, was sufficiently weak: in the case of the alternative construct the purpose is avoiding abortion, in the case of the repetitive construct the goal is getting BB weak enough such that P and non BB is strong enough to imply the desired post-condition R .

5. Concluding remarks.

The research, the outcome of which is reported in this article, was triggered by the observation that Euclid's Algorithm could also be regarded as synchronizing the two cyclic processes "do x:= x - y od" and "do y:= y - x od" in such a way that the relation x > 0 and y > 0 would be kept invariantly true. It was only after this observation that we saw that the formal techniques we had already developed for the derivation of the synchronizing conditions that ensure the harmonious co-operation of (cyclic) sequential processes, such as can be identified in the total activity of operating systems, could be transferred lock, stock and barrel to the development of sequential programs as shown in this article. The main difference is that while for sequential programs the situation "all guards false" is a desirable goal --for it means termination of a repetitive construct-- , one tries to avoid it in operating systems --for there it means deadlock.

The second reason to pursue these investigations was my personal desire to get a better appreciation --among other things in order to be able to evaluate how realistic some claims towards "automatic programming" were-- which part of the programming activity can be regarded as formal routine and which part of it seems to require "invention". While the design of an alternative construct now seems to be a reasonably straight-

forward activity, that of a repetitive construct requires what I regard as "the invention" of an invariant relation and a variant function. For me, the main value of the calculus shown in section 4 is that it has strenghtened my skepticism about some of the claims or goals of "automatic programming"; me presenting this calculus should not be interpreted as me suggesting that all programs should be developed that way: it just gives us another handle.

The calculus does, however, explain my preference for the axiomatic definition of programming language semantics via predicate transformers above other definition techniques: the definition via predicate transformers seems to lend itself most readily to being forged into a tool for the goal-directed activity of program composition.

Finally I would like to say a word or two about the role of the potential non-determinacy. I quote in this connection C.A.R.Hoare: "A system which permits user programs to become non-deterministic presents dreadful problems to the maintenance engineer: it is not a "facility" to be lightly granted." (This is particularly true in the absence of self-checking hardware.) I myself had to overcome a considerable mental resistance before I found myself willing to consider non-deterministic programs seriously. It is, however, fair to say that I could not have discovered the calculus shown before having taken that hurdle and I leave it to the environment whether the non-determinacy is eventually resolved by human intervention or mechanically, in a reproducible manner or not. (It is only in an environment in which all programs should be deterministic, where non-reproducible behaviour is interpreted as machine malfunctioning: I can easily think of an environment in which non-reproducible user program behaviour is quite naturally and almost always correctly taken as an indication that the user in question has written a non-deterministic program!)

Acknowledgements.

In the first place my acknowledgements are due to the members of the IFIP Working Group W.G.2.3 on "Programming Methodology": it was, as a matter of fact, during the Blanchland meeting of this Working Group in October 1973 that the guarded commands were both born and shown to the

public. In connection with this effort its members R.M.Burstall, C.A.R. Hoare, J.J.Horning, J.C.Reynolds, D.T.Ross, G.SeegmÜller, N.Wirth and M. Woodger deserve my special thanks. Besides them, W.H.J.Feijen, D.E.Knuth, M.Rem and C.S.Scholten have been directly helpful in one way or another. I should also thank the various audiences --in Albuquerque (courtesy NSF), in San Diego and Luxembourg (courtesy Burroughs Corporation)-- that have played their role of critical sounding board beyond what one is entitled to hope.

[1] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, Comm.ACM 12 (Oct. 1969) 576 - 583.

[2] Naur, Peter, (Ed.) Report on the Algorithmic Language ALGOL 60, Comm.ACM 3 (May 1960) 299 - 314

26th June 1974
NUENEN

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

M. Griffiths

Université de Grenoble *
France

Program Production by Successive Transformation

ow at the Univerity of Nancy, France

I. INTRODUCTION

Over the last few years, the computing community, using terms like software engineering, program proving or structured programming, has been concerned with the technology of producing programs in which the user may have a certain confidence. This technology, well-described in the literature in works like (Dahl, 72), leaves the programmer with a certain number of sound tools to be applied during the program production process. We will suggest, during this set of lectures, another tool which may be used with the same aims. The idea is not to replace any existing techniques, but to give another handle which may be useful in certain cases.

The technique proposed is that of successive transformation of the abstract specification of the algorithm towards an efficient program as proposed by F. L. Bauer in (1973). This will often involve the production of a recursive version of the program, to be followed by the elimination of the recursion in order to obtain an efficient, iterative text. Such transformations have been studied theoretically, and a developping body of knowlegde allows us to lay down guidelines for attacking particular cases which occur. Before writing programs, let us also consider the hoped - for status of a program text.

At the present time, large programs are built up of basic components which are simple instructions, and the result can only be used in its entirety. This means that when a similar problem is tackled later, programmers start again. In long term, this state of affairs must evolve, since it makes economic nonsense. It must become standard practice to re-use, and to communicate, individual algorithms as procedures, these being the middle-level components which are necessary. Parallels may be found in electronic circuitry, or in any form of mechanical engineering. The individual components should be capable of composition to produce larger units. A few attempts at global structures of this kind have been made, in particular for mathematical function libraries.

In order to put these ideas into practice, several problems need to be solved. First of all the programmers themselves have to be convinced. They must also have available suitable tools and techniques. It is with these that we are concerned. The existence of useful blocks of program requires, in the first instance, a much higher degree of formalisation than that to which we are accustomed at the

specification level. The form of the specification should probably be functional, and not include the notion of variable. It is here that a lot of work needs to be done, since everything hinges on this form. We foresee that the text of the program should be deduced, by successive transformation, from the specification, which should remain the only form known to the outside world. The next chapter shows an example of the derivation of an efficient program text from an abstract, functional definition.

Another topic which will require research is that of the development of methods of composition of functional specifications. We do not, at the present time, have any serious results in this direction, but, given the fact that the formalisation is so close to classical mathematical statements, it is supposed that it will be possible to apply well-known, traditional methods.

So far, this discussion has led to considerations as to the way in which the programmer does his work. We should also consider what would be the impact of this programming style on languages and software support. We will examine each of these topics in subsequent chapters. However, anything which may be presented in this direction will remain speculative until sufficient experience has been gained in practical applications. The tools must follow the programmer's real needs.

2. SUCCESSIVE TRANSFORMATION

The following example was suggested by David Gries at a course in Munich in March 1975. It is in fact a student problem, and the first solution, by invariants, is that which one could expect a student to produce. This solution will be followed by a second solution, based on successive transformation, which was developped subsequently. Both solutions are presented in the manner in which they were obtained, which makes the presentation heavier than is necessary, but we consider that it is important to analyse the thought patterns which were really followed. The example, together with another, more complicated one, is discussed in (Gries 1975).

2.1. The problem

Consider an ordered set of integers in a vector a [1:n]. It is required to find the integer which occurs most often in a. If the maximum is attained by more than one value, either solution may be given.

For example, consider the vector

(1 1 3 3 3 6 7 7)

The value of the mode is 3, and this value occurs three times.

2.2. Solution by Invariants

This method of construction requires us to start with an idea of the algorithm which we wish to produce. The program will be a loop which examines the elements successively, retaining the current values of the mode, the number of occurrences of the mode, and the number of occurrences of the most recently seen value.
We ask the question 'is the current value the same as its predecessor, and, if so, does this change our current view of the mode, and/or of its number of occurrences ?' Thus we define two cases as we go round the loop, depending on whether the current integer is the same as, or different from, the preceeding one.
The invariant of the loop is :

P :

Value = the mode of a [1:i]
Number = the number of occurrences of the mode of a [1:i]
Occ = the number of occurrences of a [i] in a [1:i]

We require a loop of the form :

while B do S

P must be true before and after the loop (that is to say that it is invariant under S), and of course B will be false when we leave the loop. The end condition of the loop (which is not B) is :

$\neg B : i \geq n$

Thus we have B : $i < n$
The loop must be :

while i < n do
comment increase i by 1 keeping P true ;

To have P true before the loop we write the initialisation :

i ← 1 ; value ← a[1] ; number ← 1 ; occ ← 1 ;

Increasing i by 1 while keeping P true leads to :

```
i ← i + 1 ;
if a[i] = a[i-1]
then begin occ ← occ + 1 ;
     if occ > number
     then begin value ← a[i] ;
               number ← occ
          end
     end
else occ ← 1
```

The reader should go slowly through this last part, confirming that it does leave P invariant. We get the following program :

```
i ← 1 ; value ← a[1] ; number ← 1 ; occ ← 1 ;
while i < n do
begin i ← i+1 ;
     if a[i] = a[i - 1]
     then begin occ ← occ + 1 ;
          if occ > number
          then begin value ← a[i] ;
                    number ← occ
               end
          end
     else occ ← 1
end
```

This is not exactly the same solution as that of David Gries, but this is of no importance to the present discussion. Improvements were suggested to the program, but nobody realised (there were 100 people present) that :

- the test a[i] = a[i-1] is unnecessary
- the problem needs only two variables and not three.

2.3. Solution by Successive Transformation

As a first attempt, we try to write down a mathematical case analysis to do the same thing, which gives the following :

X = mode (a,n)
Mode is a two-valued function, giving value (mode (a,n)) and number (mode (a,n))

1 $n = 1 \Rightarrow X = (a[1], 1)$

2 $n > 1, a[n] \neq a[n-1] \Rightarrow X = \text{mode}(a, n-1)$

3 n > 1, a[n] = a[n-1], value (mode (a, n-1)) = a[n-1]
X = (value (mode (a, n-1)), number (mode (a, n-1)) + 1)

4 n > 1, a[n] = a[n-1], value (mode (a, n-1)) ≠ a[n-1],
a[n] = a[n-number (mode (a, n-1))] ⇒ X = (a[n], number (mode (a, n-1)) + 1

5 n > 1, a[n] = a[n-1], value(mode (a, n-1)) ≠ a[n-1],
a[n] ≠ a[n-number (mode (a, n-1))] ⇒ X = mode (a, n-1).

The idea is to write conditions which represent all cases and which can lead to recursive procedures. Since there are no variables, the difficulty was to discover whether a[n] was a mode or not, which forced, after reflection, the test :

a[n] = a[n-number]

The function must be two-valued, and attempts to make it one-valued failed.
This is what led to two variables (although I admit that I did not realise it at the time).
These conditions can be easily transformed to the following recursive procedure, which is also a two-valued one (purists will perhaps make an effort to excuse the as yet unsatisfactory formalism) :

```
mode (a, n) :
    if n = 1
    then (a[1], 1)
    else if a[n] ≠ a[n - 1]
        then mode (a, n-1)
        else if value (mode (a, n-1)) = a[n-1]
            then (value (mode (a, n-1)), number (mode (a, n-1)) + 1)
            else if a[n] = a[n-number (mode (a, n-1))]
                then (a[n], number (mode (a, n-1)) + 1)
                else mode (a, n-1)
```

Fortunately, it was at this point in time that I took a critical look at things. The procedure returns essentially the same solution in different cases. These correspond to the fact that, in the mathematical formulation, cases 2 and 5 can be joined into one condition, as can cases 3 and 4. We obtain the following :

```
n = 1 ⇒ X = (a[1], 1)
n > 1, a[n] = a[n-number (mode (a, n-1))]
        ⇒ X = (a[n], number (mode (a, n-1)) + 1)
n > 1, a[n] ≠ a[n-number (mode (a, n-1))]
        ⇒ X = mode (a, n-1)
```

At the moment of writing this, the idea was to shorten the code, not to make it more efficient in terms of time. Transformation to recursive procedure form now gives :

```
Mode (a, n) :
     if n = 1
     then (a[1], 1)
     else if a[n] = a[n-number (mode (a, n-1))]
          then (a[n], number (mode (a, n-1)) + 1)
          else mode (a, n-1)
```

Further transformation removes the frequent recalculation of mode (a, n-1) :

```
Mode (a, n) :
     if n = 1
     then (a[1], 1)
     else begin (p, q) ← mode (a, n-1) ;
            if a[n] = a[n] - q
            then (a[n], q+1)
            else (p, q)
     end
```

We have introduced variables for the first time ! The transformation to iterative form is :

```
value ← a[1]; number ← 1 ;
for i ← 2 step 1 until n do
begin if a[i] = a[i-number]
      then begin value ← a[i] ; number ← number + 1 end
end
```

This is a much better program than the preceeding version, needing only two variables and one test per element. This result was unexpected and was the reason for taking the method more seriously and trying to apply it to other cases.

2.4. Discussion

What do we learn from this example ? First of all, that the approach via mathematical formulation, recursive specification and transformation to iterative form was more successful than the combined efforts of some good programmers using methods which are thought to be advanced. This single success is of course not a guarantee that improvements will always be made, but, in our view, it shows itself worthy of further study. The important properties are the form of the mathematical

specification, the lack of variables, the recursive formulation and, in this case, the simplicity of transformation. We will discuss these points individually.

Mathematical formulation allows us to remain much further from the computer than would otherwise be the case, and in this context any programming language is already too near. This means that the information is in a static and very dense form. In the above example this allowed us to improve the formulation by purely static considerations, but this is not the only advantage. We should also find it easier to compose several such functions in order to create a new one, since the application of functions is a well-understood mathematical tool. It will be useful to try and mirror the required function composition rules into some algorithmic language.

Lack of variables is the factor which allows us to stay away from dynamic processes as long as possible. The idea is not original (Bauer, 74) but has not yet been exploited on a large enough scale. Variables are merely concerned with the efficient implementation of algorithms, and should in many cases not be part of the construction method.

The fact that the algorithmic formulation was recursive in the first instance is typical of the method, but not indispensable. It is true that in some cases the mathematical formulation translates directly to an iterative program. However, since the recursive formulation seems to happen more often, perhaps because of a bias towards a certain type of problem, we must consider general methods for its elimination. One thing the author is trying to see is the relation between the form of the recursive program and the number of variables which are necessary.

The short version of the program is not the best solution in terms of the number of operations at run-time. Improved, but more complicated versions have been presented by E.W. Dijkstra, B. Krieg-Bruckner and the author. An important point about these solutions was that none of them were presented using the recursive formulation, which it would have been difficult to apply to them, but all use the basic idea of testing using a 'stride'. We would suggest that different methods may be interesting on different occasions, which could make life difficult when the choice has to be made. (Bauer, 75b) shows that, in some cases, the functional specification can either be transformed or used as an invariant, thus showing some kind of equivalence.

3. TRANSFORMATION METHODS

The problem of eliminating recursion has been variously studied, and we possess a certain number of recipes which are of help in many cases. To do a reasonable job, we need to study program schemes, and it may become useful to further develop automatic transformation. This has always been studied within the framework of the optimisation of text produced by a compiler, which is its natural place. However, we believe that it would be useful, under certain circumstances, to show the transformed version of his program to the user.

It is, of course, unreasonable to aim at complete automation of transformations, as it is also unreasonable to wish to eliminate all recursion. It is simply, at the present state of computer development, more efficient with most compilers to use recursion sparingly. To what extent present compiler inefficiency is justified is subject to discussion.

3.1. Recursion and Iteration

There exists a well-known equivalence between the most simple recursive situation and the while-loop, which is expressed by the equivalence of the following pair of statements :

```
while c(x) do x ← s(x)
proc f(x) ; if c(x) then f(s(x)) ;
```

We will assume that c(x) does not change the value of x (for convenience) and that s(x) has a value different from x (necessary). An example of this construction is the application of a function to each member of a list. We transform the following procedure into a corresponding loop :

```
proc apply p (list) :
     if ¬ empty (list)
     then begin p (first (list)) ;
                apply p (follow (list))
          end ;


while  ¬ empty (list) do
begin p (first (list)) ;
      list ← follow (list)
end
```

Although trivial, this transformation applies to a certain number of practical programming situations.

3.2. Introduction of a variable

This is the method which we use for the traditional case of factorial :

```
fact(n) : if n = 1
          then 1
          else (n*fact (n-1))
```

In iterative form we have :

```
fact(n) : begin int total ← 1 ;
                while n>1 do
                begin total ← total*n ;
                      n ← n-1
                end ;
                total
          end
```

This is one of the crucial steps in the production of a program, and can have considerable influence on the clarity and efficiency of the resulting text. As we see in (Bauer, 74) there is a close relationship between the parameters and the result of functions in the recursive formulation and the number of variables in the iterative one.

The real problem is that of execution order, which, instead of being implicit, as in the recursive text, becomes explicit in the iterative one. The variables will have their correct values in the new version when the operations carried out possess certain well-known, mathematical properties. Some particular cases of this type are studied in (Darlington,73) and we will return to this topic. In the example of factorial, the operation is multiplication, which is associative, commutative, and even has an inverse, and the explicit ordering of operations causes no problem. We note that, if multiplication is replaced by division, we may consider two new functions, which do not give the same result :

```
f(n) : if n = 1
       then 1
       else (n/f(n-1))
g(n) : if n = 1
       then 1
       else (g(n-1)/n)
```

Most of us would have to stop and think to get rid of the recursion in these examples, and there are sound mathematical reasons for this.

In many cases, programmers introduce variables as indications of their intuitive understanding of problem structure, and this is a normal, and useful, method of working. This may also happen when working by transformation, as in the case of the following example, suggested by Bauer :

We consider the case of the calculation of a^b with the classic optimisation which allows squaring when b is even, as in the following formulation :

$x = power\ (a, b)$
$b = 1 \Rightarrow x = a$
$b = 2*y\ (y\ integer) \Rightarrow x = a^y * a^y$
$b = y + 1 \Rightarrow x = a * a^y$

This gives the simple recursive procedure :

```
power (a, b) :
      if b = 1
      then a
      else if even (b)
             then (power (a, b/2) * power (a, b/2))
             else (a * power (a, b-1))
```

Improvement comes from not re-evaluating the same call twice, and realising that if b is odd then b-1 is even, thus obtaining, leaving out declarations :

```
if b = 1
then a
else begin if odd (b)
           then begin b ← b-1 ; mult ← a end
           else mult ← 1 ;
           y ← power (a, b/2) ;
           mult * y * y
     end
```

The best iterative version we have found uses a second variable :

```
power (a, b) :
      total ← 1 ;
      factor ← a ;
      while b > 1 do
      begin if odd (b)
            then begin total ← total*factor ;
                       b ← b - 1
                 end ;
            b ← b/2 ;
            factor ← factor*factor
      end ;
      power ← total * factor
```

In this particular case, the author worked intuitively, using a new variable to resolve the problem of evaluation order. The decision was based on programming experience. However, Bauer shows in his lectures on this course that it is possible to proceed without this type of intuition. It would seem that one of the more important areas which require reflection is that of to what degree we require intuitive logic of the sort that we are used to, and to what degree we can replace this by mathematical transformation. The difference between the two is seen when we try to prove the equivalence of the recursive and iterative programs above, which requires considerable mental effort, whereas in the more formal approach, the demonstration is implied by the transformation.

3.3. Function inversion and counting

Let us consider a general form of the simple kind of recursion we have been treating :

```
proc f(x) ;
     if g(x)
     then h(x)
     else begin α ; f(k(x)) ; β end
```

α and β represent the operations accomplished before and after the recursive call. We suppose that α , β and g do not change x, and that k is a function which produces different, successive values (thus avoiding looping).

We see that the result of this program is to apply α successively to the values
$x, k(x), k^2(x), \ldots, k^n(x)$
then to produce $h(k^{n+1}(x))$

and then β to $k^n(x), k^{n-1}(x), \ldots, k(x), x$

where $g(k^i(x)), 0 \leq i \leq n$ is false

and $g(k^{n+1}(x))$ is true.

The first operations can be obtained by :

```
while ¬ g(x) do
begin α ; x ← k(x) end ;
h(x)
```

Remains the execution n times of β . Different conditions, pointed out in (Kröger, 74) and (Veillon, 74) may allow this. For example, if β does not use the values of x produced by k, we merely need to count, either by k itself or with some other counter :

```
    repeat the same number of times β
or  x ← x0 ;
    while ¬ g(x) do
    begin β ; x ← k(x) end
```

x_0 is the original value of x. In the first version we would declare an artificial counter. In the second version, β operates in the presence of each relevant value of x, but in the opposite order from that required. This may not matter, but to obtain the right order, we may use the inverse of k (if it exists). We obtain :

```
proc f(x) ;
begin x0 ← x ;
      while ¬ g(x) do
      begin α ; x ← k(x) end ;
      h(x) ;
      while x ≠ x0 do
      begin x ← k-1(x) ; β end
end
```

It is when k has no inverse and the order of application of the different calls of β is important that we may need to stack the values of $k^i(x)$.

Note that the example of factorial can be considered as having α empty (h(x) is total ← 1), and using the function k(i ← i-1) or its inverse (i ← i+1) as producers of successive values. The two versions of factorial are :

```
fact(n) : i ← n ; total ← 1 ;
          while i > 1 do
          begin total ← total * i ; i ← i-1 end
fact(n) : i ← 1 ; total ← 1 ;
          while i < n do
          begin i ← i+1 ; total ← total * i end
```

3.4. Changes in Data Structure

If we try to summarise the reasons for taking some of the lines of approach which we choose, they are essentially dual. Firstly we wish to provide techniques which will help programmers in solving problems, and secondly to improve educational processes by making students more conscious of the way in which the programming task is performed. These aims are not new, and our suggestions form a part of a wave of preaching, teaching and persuasion of various sorts. Both the above aims are involved in thinking about how algorithms are invented, as in the following example, again due to Veillon.

Consider the case of the routine which marks words in a list system based on LISP-like pointers. The central part of the algorithm has the form :

```
proc search(x) ;
     if¬atom(x)
     then begin search (hd(x)) ;
                search (tℓ(x))
          end
```

We have left out problems of physically placing and testing the mark. The second recursive call is easily eliminated by the standard device

```
x ← tℓ(x)
```

followed by a repeat of the process. The first call is more difficult, since hd(x) has no inverse. Under certain circumstances we may wish to remedy this by putting reverse pointers into the structure, so as to be able to climb back up. If we consider carefully which pointers are required when it is ultimately possible to find the best of the known marking algorithms, which was proposed by Schorr and Waite and is described in (Knuth, 73).

This teaching technique should force the student to discover the solution himself. We obtain this result by making him conscious of the particular properties that the program should possess, in the case of this example the existence of an inverse for a given function. The example also shows the necessity to consider each of the aspects of a program, algorithm, control structure, program properties and data structure.

3.5. Program schemes and automatic transformation

Elimination of recursion has been studied by various authors, some of whom have studied the automation of the processes involved. The theoretical work involved is based on program schemes (see(Manna, 74), (Garland, 71), (Cooper, 66), (Strong, 70)) in which the relevant control structures and conditions are codified. Practical implementations of the theory are rare, being confined to particular cases considered by optimisating compilers. An exception is that described in (Burstall, 75) and (Darlington, 73), in which a practical, working implementation is applied to LISP-like programs. We give below an example from (Darlington,73), translated into an ALGOL-like form for readability, in order to give the flavour of this work. The method is to define equivalent recursive and iterative schema, together with the conditions which must be satisfied if the transformation is to be allowed. The example is that which generalises the case of factorial :

recursive form : $f(x) = \begin{cases} a \rightarrow b \\ \underline{\text{not}}\ a \rightarrow h\ (d,\ f(e)) \end{cases}$

```
iterative form : f(x)=  if a then result ← b
                        else begin result ← d ;
                                   x ← e ;
                                   while not a do
                                   begin result ← h (result, d) ;
                                         x ← e
                                   end ;
                                   result ← h (result, b)
                             end
```

conditions : h is associative, x does not occur free in h.

If we again think about the teaching of programming, and in particular the necessity of treating programs as objects which have properties, the habit of working with generalised program schemes is most useful. It also encourages students to realise that there exist solutions to classes of problems, illustrated, for example, by the fact that the above scheme applies both to factorial and to the inversing of the order of elements in a list (pointed out in (Cooper, 66) :

$$fact(x) = \begin{cases} x = 0 \rightarrow 1 \\ x \neq 0 \rightarrow mult\ (x,\ fact(n-1)) \end{cases}$$

$$reverse(x) = \begin{cases} null(x) \rightarrow nil \\ \neg\ null(x) \rightarrow concat(reverse(t\ell(x)),\ cons(hd(x),\ nil)) \end{cases}$$

The necessity of recursion removal is purely economic, and the interest of the topic will be discussed in the next section. Whether or not we wish to remove recursion in practice, the author considers that the teaching of program schemes and their equivalence to be very useful.

3.6. Discussion

This discussion has two main objectives, the first being to repeat the reasons which motivate work of this type, and the second to try to put it into perspective by indicating what has been achieved and to project towards the future.

The revolution which programming is going through as a result of the realisation that programs are susceptible to mathematical treatment, and indeed verification, has led to a number of interesting lines of research, for example the study of automatic proving techniques (previously regarded as being an interesting toy), or the practical study of abstract program schemes. In the practical world, the main difficulties encountered in the application of these attractive techniques are those of the need to produce both a static description of a problem in terms of invariants and its dynamic equivalent, which is an executable program text ; the need to educate suitably trained programmers is also important. This inspires us to consider ways of automating the transformation of static descriptions into programs, thus eliminating the necessity of a separate proof phase. Of course, as we all know, modern techniques already lead us to the parallel construction of the program and of its proof.

Based on the experience of an admittedly small number of examples, we suggest that the method is immediately applicable to some percentage of cases, and as such, should be generally taught to, and studied by, programmers in order to add another tool to what is at the moment a limited set. To whatever proportion of programs the method represents a help, and this is a very speculative question, we also believe that the study of the involved mechanisms is of a nature to improve the capacities and understanding of students of programming, and that teaching should be influenced.

In the future, we must continue to develop along the lines which are already clear, and we identify three such domains :

- Examples. We must try a very much larger number of examples, including cases which are full-sized. Students could help here.
- Formal work on program schemes. Projects like that of Burstall and Darlington, previously quoted, indicate that it should be possible to consider many

more cases. We should also consider whether these cases can be taught intuitively.

- Hand transformation. We are in the process of discovering a certain number of guidelines, including, for example, indications of the number of necessary variables, those properties which it is useful to discover in the functions (associativity, possession of an inverse, ...) and the identification of critical sections. These guidelines are, at the moment, difficult to give to somebody else, and we must find how to present them clearly.

It is only after intensive use and analysis that we can decide on the real importance of different tools.

4. SOME IMPLICATIONS

Perhaps the lesson that we are trying to get across is that, whatever algorithmic language we choose, it will be at too low a level to serve as a specification language for a problem. Current specification languages tend to be higher level algorithmic languages, and this may well be a mistake. We consider that a specification should be largely mathematical and less computer oriented. From such a specification, we should, however, be capable of producing a program acceptable to some compiler. It may well be necessary to pass through several transformations in order to arrive at a program which is acceptable in terms of efficiency, but the important factor remains the established equivalence of successive versions, which avoids program proving.

Some details emerge from this which are worthy of discussion. The preceeding chapters have assumed that the specification transforms most easily into a recursive structure. This will often be so, but not always. In particular, many numerical problems are already cast in a form which gives rise to iterative loops ; after all, FORTRAN was developped for good reasons !

4.1. Language design

Most of the language requirements for the mechanisms we suggest are those which are already considered to encourage good programming. One difference is that the criteria of clean programming do not need confirmation, since they happen automatically. The independence, for example, of the different parameters and variables is assured by the derivation from mathematical text. Side effects are not possible in this structure.

One detail is the use of multi-valued functions. The method of successive transformation shows up the limitations of current languages in that they allow an unlimited number of parameters, but only one result. There is clearly a need for multi-valued results, defined in the same way as the parameters. This need was already apparent in the first example, where we wished to write :

$$(p, q) \leftarrow \text{mode}\ (a, n-1).$$

The procedure delivers two values, which should be associated by their order, in the same way as parameters. We know of no language which allows this. In defense of existing languages, we should point out that the difference is merely one of syntactic sugar.

There is considerable consensus of opinion on the fact that, at our present level of knowledge and understanding, the problems of language design are not those of control structure, but rather of data structure, type and addressing mechanisms. We will not consider these problems in this study, which means that, within the limits we impose, present programming languages are relatively satisfactory, even if they are far too general and allow too many silly things.

4.2. System structure

The main purpose of this paper is to try to persuade users of computers to see the process of producing programs in a different light, and in particular to consider programs much more as manipulable objects which exist in different forms, and even in different languages, but which do well-defined things. Whichever version of a program we take, we should get exactly the same result for any given set of data. Algorithms are to be seen as mathematical functions and manipulated as such.

If this sermon is to have any effect, it will primarily be on the habits of programmers, and not as much on available software. However, we may well ask the question of what this may mean in terms of automatic system support. A first conclusion that we draw is that the different versions of the algorithm should be stored in the computer, the most important being the original specification. This is not just to encourage the programmer to follow the rules, but also that the highest level is the most re-usable and adaptable. The only hope for a sane and reasonable future in programming technology lies in the re-use of previous work, thus avoiding the continual re-writing and debugging of many examples of the same program. We have so far been tackling this problem on a low level by writing emulators, trying to standardise languages, and so on. In the author's opinion, we should be seeing the problem at a more abstract level.

The idea of program libraries is not new, and even has practical applications in certain restricted fields, like some aspects of numerical mathematics. These have been one-language libraries, since we do not possess ways of going from one language to another. For the future, we suggest that it may be better to create specification libraries, formally described, which may be translated into different programming languages. It is this specification level which is re-usable, whatever compiler the user happens to possess.

We reach the conclusion that, although additional system software may be useful, it is not the main point that we should attack. We must concentrate our efforts on developping practical abstraction mechanisms which conform to the following

criteria :

- Bear sufficient resemblance to mathematical functions to allow some of the useful mechanisms to be applied on a static level
- Be able to be transformed into dynamic form (program) by the introduction of variables
- Be understandable to a wider audience than present-day theoretical presentations
- Be susceptible to composition to make bigger blocks from small ones.

Although the author feels that techniques can be, and are begining to be, developped, there is still a lot of work to be done in this field.

4.3. The multi-language problem

Once upon a time, a group of computer scientists decided to create a single intermediate language, UNCOL, which could serve for different source languages and different machine languages (Steel, 61). (Ershov, 75) describes a modern equivalent in another chapter of this book. In the technical sense, UNCOL was a failure, since it did not lead to numerous compilers for different languages on particular machines. As a research project, it was in at least one sense a success, since an analysis of the reasons for its failure is still applicable today. These reasons are concerned with purely technical problems in the design of languages and of computers. We will discuss only those problems concerned with languages.

The differences encountered between different, classical languages like FORTRAN, ALGOL 60 or PL/1 are well-known. Let us consider the loop statements in these languages. As we have already noted, control statements are easier to analyse. ALGOL 60 re-evaluates the step and the limit each time round the loop, PL/1 evaluates them at the start and keeps those values. FORTRAN, at least in the first versions, restricted depth of nesting to three in order to use the three index registers on the 700 series of computers. This sort of detail makes life impossible for multiple-language implementers, without, in our view, corresponding benefits for users, or even for program efficiency. Use of particular language details of this sort is often held to be a sign of unclean programming. A previous discussion of this point is to be found in (Griffiths, 73).

When these program characteristics were identified as not being conducive to good programming, the initial reaction was to teach programmers not to use these

possibilities in the language. This was a step forward, but not the root of the problem. We now realise that programmers should be taught to write programs in such a way that they would not be tempted to write programs using these possibilities. From that point to their suppression from the languages is a simple step, but one which we seem unwilling to take.

The interesting thing for this discussion is that the particular loop form chosen in your favorite programming language is irrelevant, since the simple form which results from good programming technique can be expressed in any language, even if we use if and goto. This attitude, of considering the programming language which is accepted by the computer as being a last, minor detail, has been very successful in teaching (Courtin, 75). It cannot be quite as successful in developping programs for practical applications, in view of the problems encountered with data structures.

We suggest, then, that the previous approaches to portability and the multi-language problem have been situated at too low a level. Our effort should be towards the creation of higher-level descriptions of algorithms which can then be transformed into different language systems. This point of view is, in fact, a criticism of present languages, which contain far too many special cases in situations where these have no interest. By using more mathematical methods of program construction, we avoid the use of these special possibilities, the program being recoded in any new language as the need arises.

4.4. Efficiency

The transformations which may be carried out are of two kinds, those which create a program from a mathematical text, and those which improve the efficiency of a program, usually in its execution time. They are consecutive, and essentially independent steps. The need to remove recursion is an example of a transformation carried out in the interests of efficiency. Many such transformations are available as a result of work on optimising compilers (Rustin, 72) and of hand-improvement (Knuth, 74). The need for these transformations depends on the frequency of execution of the program and on the architecture of the computer system, both hardware and software. The figures given in (Darlington, 73) of two orders of magnitude of improvement in execution time indicate the poverty of some compiler implementation. They are, of course, particular to the language and the compiler considered. What is more, future hardware or software development may completely change the situation.

4.5. Use of static information

There are many different kinds of static information available when programmers use modern methods of program construction. Since the term has sometimes led to confusion, in this context we mean by static information that information which lays down permanent relationships between program objects ; items of static information are not executable by a computer.

A preliminary list of types of static information might include the following, where individual types may be present or absent from a given program following the programming methodology applied :

- Invariants. Those relationships which a proof will show to be constantly true under the different transformations described by the program.
- Input conditions. Conditions which limit the domain of values of the input data.
- Mathematical specification of the algorithm. See the original example. Under some conditions this specification is the same as the set of input conditions plus invariants.
- Deductions. Information drawn from the program text. For example the deduction that, at termination of

  while B do S ;

  the condition B is false.
- Assertions. Relations which like deductions are to be true at a given point of the program, but which need to be shown to be true as part of the proof process. They are deduced from input conditions, deductions and formal logic.

What use should be made of this static information ? We consider to be axiomatic that the information should be kept in the program documentation, since it represents the 'understanding' of a program. However, it may be useful to go further, and to use such static statements as compile-time or run-time checks. In particular, input conditions should be checked by run-time routines, and deductions can be controlled by the compiler. Good programmers have, of course, always checked their input, but there is a very strong case for special language features. We need two, one of which is to be applied at compile-time, and the other at run-time. Syntactic sugar will depend on the host language.

To what degree the relationships between invariants, mathematical descriptions and programs may be related is still in the field of research, as is the degree to which assertions may be usefully tested automatically by proving mechanisms. The categorisation given above allows us to distinguish some of the subjects which require further work.

5. CONCLUSION

Our theme, that improvements in programming security, performance and education depend on a higher degree of formalisation and the use of static information, is one which lies in a long line of research projects and publications, each of which illuminates some part of this basic truth.

The method that we have chosen to illustrate is that of the use of recursive descriptions, which correspond to a class of algorithms which can be defined in terms of recurrence relations. This class is larger than one might think, since it includes many problems for which iterative solutions are often given directly, with the help of 'intuition' or 'inspiration'. This point of view was foreseen in (Barron, 68) amongst others.

5.1. Associated research

Perhaps the first effort at programming by static assertions instead of by dynamic instructions was the result of work at the Computer Research Group of the University of Aberdeen (Elcock, 68, 71 ; Foster, 68, 69). The basic idea is that the interpreter of the language should be able to resolve sets of constraints, called assertions, by making calculations, in some order, to find the unknowns. To give a trivial example, consider the following assertions :

a + b = 7
b + 1 = 3

By inversion of the two addition operators, the interpreter deduces, successively, that b has the value 2 and, hence, a has the value 5. These values are fixed. There is no assignment, or equivalent mechanism, since there is no specified order.

Along with the possibility of allowing constraints on numerical values, data structures can also be specified in the same way. This is a most important part of the work, and had been in some sense the starting point. We take an example from (Foster, 69). Consider the LISP-like assignment :

Z2 ← cons(cons(hd(Z1), cons(hd(tℓ(Z1)), nil)), cons(cons(hd(Z1),
cons(hd(tℓ(Z1))), nil), nil))

The following assertions represent the same relationship between Z1 and Z2 without specifying which list should be deduced from which :

Z1 = [a ; [b ; c]]
Z2 = [[a ; b] ; [a ; c]].

This, of course, is a good example for the method, since the assignments are so ugly. The comparison would be less severe with a more modern language.

Work on the compilation and interpretation of such languages has at least indicated the limits which can be approached by static programming, and also those points at which a knowledge of execution order is essential.

Better known research which helps us to see the relationship between static information and programs can be found in projects which aim at improving language definition (Hoare, 69 ; Knuth, 68, 71). There is a close parallel between the relationship that ties a language definition to its compiler and that which binds the specification of an algorithm to its corresponding program.

5.2. Final remarks

The programming revolution which depends on (Floyd, 67) and which has led to the methods of (Dahl, 72) is not yet over. The impetus given to our thinking is leading to changes which will only take place over a number of years. The ideas are to be absorbed in teaching, the programming industry and even in the realm of mathematics. We must continue to add new tools, while working towards some unification among different tendencies.

REFERENCES

D.W. BARRON
Recursive Techniques in Programming
Macdonald Computer Monographs, 1968

F.L. BAUER
Variables Considered Harmful
Seminar, TU Munich, 1974

F.L. BAUER
A Philosophy of Programming
TU Munich, 1975

F.L. BAUER

Marktoberdorf, 1975 (b)

R.M. BURSTALL, J. DARLINGTON
Some Transformations for Developping Recursive Programs
SIGPLAN Notices 10, 6, June 1975

D.C. COOPER
The Equivalence of Certain Computations
Comp. Jnl. May 1966

J. COURTIN, J. VOIRON
Introduction à l'Algorithmique et aux structures de données
Université de Grenoble, 1975

O.J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE
Structured Programming
Academic Press, 1972

J. DARLINGTON, R.M. BURSTALL
A System which Automatically Improves Programs
3rd Int. Conf. on Artificial Intelligence, Stanford, Aug. 1973

E.W. ELCOCK
Descriptions
in Machine Intelligence, 3, Edinburgh University Press, 1968

E.W.ELCOCK, J.M. FOSTER, P.M.D. GRAY, J.J. MCGREGOR, A.M. MURRAY
ABSET-A Programming Language Based on Sets : Motivation and Examples
in Machine Intelligence, 6, Edinburgh University Press, 1971

A.P. ERSHOV
Problems in Many-Language Systems
Marktoberdorf, 1975

R.W. FLOYD
Assigning Meanings to Programs
Proc. Symposium in Appl. Maths.,AMS, New York, 1967

J.M. FOSTER
Assertions : Programs Written Without Specifying Unnecessary Order
in Machine Intelligence, 3, Edinburgh University Press, 1968

J.M. FOSTER, E.W. ELCOCK
ABSYS 1 : An Incremental Compiler for Assertions ; An Introduction
in Machine Intelligence, 4, Edinburgh University Press, 1969

S.J. GARLAND, D.C. LUCKHAM
Program Schemes, Recursion Schemes and Formal Languages
UCLA-ENG-7154, University of California, 1971

D. GRIES
Recursion as a Programming Technique
Cornell, April 1975

M. GRIFFITHS
Relationship between Language Definition and Implementation
in Advanced Course in Software Engineering, Springer-Verlag, 1973

C.A.R. HOARE
An Axiomatic Basic for Computer Programming
CACM, 1969

D.E. KNUTH
Semantics of Context-Free Languages (a)
Math. Sys. Theory, Vol. 2, n° 2, 1968

D.E. KNUTH
Semantics of Context-Free Languages (b)
Math. Sys. Theory, Vol. 5, n° 1, 1971

D.E. KNUTH
The Art of Computer Programming, Vol. 1
Addison Wesley, 1973

D.E. KNUTH
Structured Programming with GOTO Statements
ACM Comp. Surveys, Dec. 1974

H. KRÖGER
Bemerkungen zur Auflösung von Rekursionen
Seminar, TU Munich, 1974

Z. MANNA
Mathematical Theory of Computation
McGraw Hill, 1974

R. RUSTIN (ed.)
Design and Optimization of Compilers
Prentice Hall, 1972

T.B. STEEL
UNCOL : The Myth and the Fact
Ann. Rev. in Aut. Prog., 2, 1961

H.R. STRONG
Translation of Recursion Equations into Flowcharts
JCSS, June 1971

G. VEILLON
Transformation de Programmes Récursifs
Grenoble, 1975

ACKNOWLEDGEMENTS

Original impulsion came from discussions with F.L. Bauer ; the author has also benefited from contact and discussion with J. Courtin, D. Gries, C.H.A. Koster, G. Louis, J.P. Paillard, P.C. Scholl, G. Veillon, J. Voiron.

F. L. Bauer

Technical University Munich
Germany

Programming as an Evolutionary Process

Over the last years, the Marktoberdorf Summer School has been a place where people tried to uncover the mysteries of programming. Programming on the way from the dilettantism of a home-made 'trickology' [1] to a scientific discipline: this has been the general theme of many of the lectures by Dijkstra, Hoare, Dahl, Perlis, Brinch Hansen, Randell, Wirth and others [2].

Discipline means 'a method of teaching'. Programming has to be learned. Programming as a scientific discipline means: Programming can be taught, programming is to be taught. Programming is not trivial. Taking into account on the one hand the vast masses of unskilled programmers that the hasty building of the computer industry has raised, on the other hand a typical pure mathematicians attitude "I think programming has no problems", we have to face the fact that education (of students and society) is our most important duty.

Programming needs discipline. This is what we should have learned from the 'software crisis'. More precisely: Programming as a process can be done in an orderly way, step by step, applying rules safely - under control of intuition. The programming process leads safely from the problem description (the "contract"), to the fully operative, operationally ameliorated form that is handed over finally to the compiler, i.e. to the computing system. This has been demonstrated by the discipleship of 'Structured programming' and of 'Proving program correctness'. The following lectures should contribute more universally to these efforts.

---

1) The term was coined by W.L. van der Poel 1962 [1]

2) In particular, [2]...[13] and the lectures in this volume.

First Lecture: METAMORPHOSES

Styles of Programming

Assume, a master in computer science (having just got his degree) is asked to produce a program for the factorial. It can be expected that the word factorial rings a bell in his intellectual apparatus. Quite likely, the program he produces looks like this [1)]

```
                proc fact = (nat n) nat :
                   ⌜ var nat y := n , var nat z := 1 ;
(I)                  while y > 0 do  z := z × y; y := y-1 od;
                     z                                      ⌟
```

A mathematician, given the same order, may produce something like

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ fact(n-1) \times n & \text{otherwise} \end{cases}$$

Why the difference?

The mathematician does not know how to program in the style of (I), and he may not even realize that he has written a program, which reads in ALGOL 68 notation

```
proc fact = (nat n) nat :
              if n = 0 then  1
                       else  fact(n-1)  × n   fi
```

which is essentially the same as

```
                proc fact = (nat n) nat :
(M)                   if  n > 0  then  fact (n-1) × n
                                 else  1              fi
```

---

1) For the sake of comparison, a notation similar to ALGOL 68 is used everywhere. nat denotes non-negative integers, ⌜ and ⌟ denote begin and end. Other non-orthodox constructs like simultaneous assignments are used somewhere. var is used in the sense of PASCAL.

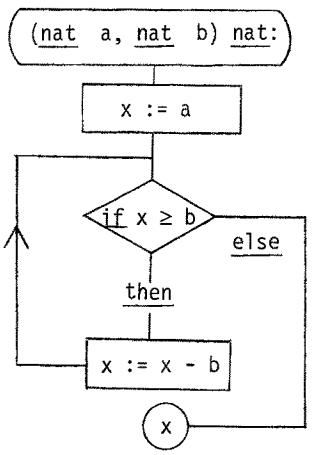
What is now the difference? Is there any substantial difference between (I) and (M)? Before going into this, we discuss a few other examples:

If the problem is posed:

"determine the remainder of the division of a by b , where b is nonzero

somebody may arrive at the flow-chart

(T)



and somebody else will write

(U)

```
proc mod = (nat a , nat b) nat :
    ⌈var nat x := a ;
     while x ≥ b do x := x - b od ;
     x                                  ⌋
```

and a third one the following

(V)

```
proc mod = (nat a , nat b) nat :
    if a ≥ b then mod (a - b, b)
             else a              fi
```

and still a fourth will arrive at

"the number x , $0 \leq x < b$ such that $a = m \times b + x$
for some number m "

which would be formalized to [1]

```
proc mod = (nat a, nat b) nat :
(W)          ι nat x : (∃ nat m : a = m × b + x ∧ 0 ≤ x < b)
```

What does the difference mean? It means that a programmer trained in the use of flow-charts reads the problem description in the following way: "Take b away from a if this is possible, and do this again". In doing so he assumes a 'program variable' to hold the intermediate values. So does the classical ALGOL programmer, who reads the problem description as follows:
"x is initially a. As long as x is greater or equal b , subtract b from it".
Version (V) could be produced by a mathematician who is trained in reducing a solution to a simpler case of the same problem. Version (W) finally, would be found in most modern textbooks of elementary algebra or number theory, it is a formal definition of the problem.

This definition may at first glance not seem to be operational, although it is: The interval $[0, b - 1]$ contains a finite number of elements, and no m greater than a will fulfill the condition; thus a search machine will find the solution x once it is established that x is uniquely defined.

But such an operational solution is usually considered to be too cumbersome: Once it has been shown that the version (V) is equivalent to (W), a HERBRAND-KLEENE machine [2] will produce the solution more efficiently than the search machine.

---

1) The symbol ι means "this one ..., which fulfills...". Its use is only allowed if the element is uniquely defined ($b \neq 0$ is necessary and sufficient!). The symbol was introduced by ACKERMANN, ZUSE used it 1945 in his PLANKALKÜL [14].

2) Instead of KLEENE's definition of 1936 [15], it suffices to take the simpler machine using the so-called 'full substitution rule', see for example [16].

Again, the recursion in (V) is of such a special form that it can be transformed equivalently into the iterative form (U).

The examples show: Preoccupation and knowledge about operative solutions, even knowledge of preferred machine functions - which are effects of training - influence programming style. Someone who finds the style of (T) or (U) suitable to express directly what he thinks he understood to be the problem, is fully justified in using this style. But somebody who says: "Let me think: I would use a variable and subtract repeatedly ..." is not formalizing the problem, he is formalizing a hazy solution to the problem - which may be wrong.

The counsel is therefore: *Start with the problem and formalize it by defining it to a sufficient degree of precision, and do not try to jump into a solution.*

But I hear people saying: How do we proceed to a solution? A program written in the style of (W) will not be acceptable to any compiler: even a program written in the style of (V), I have been taught, is recursive and should be avoided. The answer is: There are safe ways leading from (W) to (V) and from (V) to (U) - and, of course, from (U) to (T). The latter is obvious. We shall, however, discuss in some detail the first two steps.

Properties Defining Recursion and their Derivation

In order to proceed from (W) to (V), we first have to establish the property

$$a \geq b \text{ implies } \mathrm{mod}(a,b) = \mathrm{mod}(a-b, b)$$

This can be done quite formally:

$$\begin{aligned}
\mathrm{mod}(a-b, b) &= \iota\, \underline{\mathrm{nat}}\ x : \exists\, \underline{\mathrm{nat}}\ m : a-b = m \times b + x \wedge 0 \leq x < b \\
&= \iota\, \underline{\mathrm{nat}}\ x : \exists\, \underline{\mathrm{nat}}\ m : a = (m+1) \times b + x \wedge 0 \leq x < b \\
&= \iota\, \underline{\mathrm{nat}}\ x : \exists\, \underline{\mathrm{nat}}\ n : a = n \times b + x \wedge 0 \leq x < b \\
&= \mathrm{mod}(a,b).
\end{aligned}$$

Secondly, we have to prove termination: Since $b \neq 0$, we have $b > 0$ and thus, the Archimedian property of the numbers guarantees that $a - m \times b < b$ for some m. Usually, however, one does this proof intuitively.

Formalizing the problem description does not seem to be of great importance when doing the step from (W) to (V). The gain lies in the choice of the property we derive, and this choice is done intuitively. After formalization, however, the actual confirmation can be reduced to mechanical checking. This shows already the essential cooperation between intuition and mechanical performance in the programming process. The gain becomes more obvious when we are able to derive from (W) another property, namely

$$\text{mod}\ (a,b) = \text{mod}\ (\ \text{mod}\ (a,\ 2 \times b),\ b)$$

It takes a few more steps (which are left to the reader) to demonstrate this result. Termination will also be shown, and we arrive at the program [1]

(VV)

```
proc mod = (nat a , nat  b) nat :
              if  a ≥ 2 × b  then  mod ( mod (a, 2 × b), b)
              elsf  a ≥ b    then  mod (a-b, b)
                             else  a                          fi
```

Operationally, this is considerable gain over (V): a reduction in the number of repetitions from m to the order $^2\log m$. Could we have derived this improved version as easily from (V) directly, i.e. without explicitly using the properties expressed in (W)? It seems to be essential to start with the version (W).

The counsel is: *When amelioration is sought for, it should not be tried at too low a level.* At still lower levels, this is more generally known: Nobody would try to obtain this amelioration from (U) or on the flow diagram level. Thus, even when a problem seems to be stated "naturally" in the style of flow-chart programming or classical ALGOL, it may be worthwhile to formulate it 'on higher levels', in order to be able to come to more efficient solutions.

---

[1] due to P. PEPPER

The Euclidean algorithm, for example, may be formulated: "Starting with a and b, form a sequence by adjoining repeatedly the remainder of the penultimate element divided by the last element, until zero is reached". Although the use of a data structure 'sequence' would more directly correspond to this description, an iterative program

```
proc gcd = (nat a, nat b) nat :
(A)        ⌈var nat u := a, var nat v := b ;
            while v > 0 do (u,v) := (v, mod (u,v)) od ;
            u                                          ⌋
```

seems to be appropriate. However,

```
proc gcd = (nat a, nat b) nat :
(B)         if b > 0 then gcd (b, mod (a,b))
                     else a                    fi
```

does not only show much more simply the essence of the algorithm, it also has no sequential hazards:

In (A), most programmers feel somewhat uneasy about the result: Is it u or is it v, that gives the result upon termination? Why is it not v? Most people try to decide this by simulating some steps of the sequential process. (Of course, it can not be v, since v is zero upon termination, by definition.)

Why should we in spite of this continue with the descriptive form

```
proc gcd = (nat a, nat b) nat :
(C)        ι nat x : x|a ∧ x|b ∧
                  (∀ z : z|a ∧ z|b) : x ≤ z ⇒ x = z
```

It shows immediately the commutativity of a and b in gcd(a,b) and the associativity of a, b and c in gcd(a, gcd(b,c)); while the first could also be derived easily from (B), to derive the second from (B) would be a difficult task.

Moreover, it produces variants like

(BB)
```
proc gcd = (nat a, nat b) nat :
    if   a > b  then  gcd (a-b, b)
    elsf a < b  then  gcd (a, b-a)
                else  a          fi
```

(which still shows the symmetry, but fails to terminate for a = 0 or b = 0!) or

(BBB)
```
proc gcd = (nat a, nat b) nat :
    if b > 0  then if  a ≥ b  then gcd (a-b,b)
                              else gcd (b, a) fi
              else a                          fi
```

(which works also for a = 0 or b = 0 and is not so far from (B)).

This demonstrates that the derivation of recursive programs from a descriptive form like (C) has to be done with care also as far as termination is concerned. (Replacing in (BBB) the condition a ≥ b by the condition a > b is deadly!) More protection against mistakes can be obtained by using the idea of Dijkstra's 'guarded commands' (see his lectures in this volume):

```
proc gcd = (nat a, nat b) nat :
    if  a > b ∧ b > 0  then  gcd (a-b,b)
    ▯     b > a        then  gcd (b,a)
    ▯   a = b ∨ b = 0  then  a          fi
```

Second Lecture: TECHNIQUES

Transition between Recursive and Iterative Notation

The transition between recursive and iterative notation is the story of looming and vanishing of 'program variables'. Certain selfrecursive definitions like (V) or (B) allow direct transition to iterative notation, introducing as many program variables as there are parameters and using a simultaneous assignment modelled after the actual parameter expressions in the recursive definition. For example, (V) is equivalent (by definition!) to

```
proc mod = (nat a, nat b) nat :
        ⌜(var nat x, var nat y) := (a,b) ;
         while x ≥ y do (x,y) := (x - y, y) od;
         x                                    ⌟
```

and (B) equivalent to

```
proc gcd = (nat a, nat b) nat :
        ⌜(var nat u, var nat v) := (a,b) ;
         while v > 0 do (u,v) := (v, mod (u,v)) od ;
         u                                         ⌟
```

from which (U) and (A), resp., differ only in details: The initialization is broken up. Moreover, in (U), no variable is needed for the second parameter which remains constant under recursion.

The general scheme for this transformation for procedures without parameters and results is (see DIJKSTRA [13])

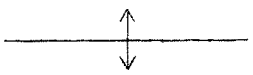
```
proc F = : if B then S ; F fi
```



```
proc F = : while B do S od
```

For procedures with parameters and results, one obtains

```
proc F = (λ A) ρ :
     if B[A] then S ; F (A')
             else p (A)      fi
```



```
proc F = (λ A) ρ :
    [var λ Y := A ;
     while B[Y] do S ; Y := Y' od ;
     p(Y)                          ]
```

where λ and ρ are (composite) types and A the (composite) parameter, A' means an expression in A , Y' the same expression in Y.

The iterative normal form thus consists of three parts: initialization, repetition, result indication. Note that every variable introduced in this way is initialized. The occurrence of uninitialized variable declarations frequently indicates bad programming style.

Our program (M), however, does not allow this transition. Instead, we consider a related program:

```
      proc fact = (nat n) nat: fac(n,1),
      proc fac  = (nat n, nat m) nat :
(MM)                if n > 0 then fac(n-1, m × n)
                             else m              fi
```

It allows transition to the following iterative notation

```
proc fact = (nat n) nat :
        ⌈ (var nat y, var nat z) := (n,1) ;
(II)      while y > 0 do (y,z) := (y-1, z × y) od ;
          z                                        ⌋
```

From this, (I) will be obtained by breaking up initialization and by sequentializing the repeated assignment: Note that the order of the two single assignments specified in (I) is the only correct one!

The transition above falls under the general scheme

```
proc F = (λ A) ρ : G (A") ,
proc G = (μ Y) ρ :
         if B[Y] then S ; G (Y')
                 else p (Y)    fi
```



```
proc F = (λ A) ρ :
       ⌈ var μ Y := A" ;
         while B[Y] do S ; Y := Y' od ;
         p (Y)                          ⌋
```

What, however, is the connection between (M) and (MM)? To establish it, we have to do a silly thing, called embedding: We complicate the definition (M) of the factorial by introducing additional parameters which are finally kept fixed. Thus, we may write

(M')
```
proc fact = (nat n) nat : fac(n,1) ,
proc fac  = (nat n, nat m) nat :
            if n > 0 then fac(n-1,m) × n
                     else m              fi
```

The only difference now is that

	fac(n-1,m) × n	in (M')
is replaced by	fac(n-1, m × n)	in (MM)

Is this allowed? In our special case, we may verify it easily. There is, however, a more general rule, found by COOPER in 1966 [17] [1]:

Let σ denote a binary operation on natural numbers in infix notation. Then

```
proc f1 = (nat n, nat m) nat :
          if n > 0 then f1(n-1,m) σ n
                   else m                fi
```

and

```
proc f2 = (nat n, nat m) nat :
          if n > 0 then f2(n-1, m σ n)
                   else m                fi
```

---

1) DARLINGTON and BURSTALL used a variant in 1973 [18], MORRIS found (independently) a special case in 1971 [19].

are equivalent if and only if σ is right-commutative:

$$(a \sigma b) \sigma c = (a \sigma c) \sigma b \ .$$

Since + and x are commutative and associative, they are right-commutative But - is so, too:

$$(a - b) - c = (a - c) - b$$

and so is / :

$$(a / b) / c = (a / c) / b \ .$$

The COOPER transformation is one among several methods to transform (at the expense of additional parameters) a recursive definition to a form which is equivalent to an iterative form. Not all recursive definitions allow this with the introduction of a finite number of additional parameters. Intrinsically recursive procedures can only be treated with the introduction of stacks instead of variables.

Function Inversion

When the COOPER transformation is not applicable, another method may sometimes be applied. A recursive procedure without formal parameters of the form

proc F = : if B then α ; F ; β fi

where α and β are statements, can be interpreted as

proc F = : ⌈ while B do α od ;
dojustasoften β od ⌋

(STRONG 1970 [20], KRÖGER 1974 [21]).

This leads to the following implementation: With the help of an arbitrary unary operator q on objects of some mode ν , which is of order infinity for some 'initial value' i_0 ,

$$\forall\ n,m \in \mathbb{N},\ n \neq m\ :\ q^n\ i_0 \neq q^m\ i_0$$

(and thus has an inverse q^{-1} on $\{q^n\ i_0,\ n \geq 1\}$),

the number of iterations in the while-part is 'counted' and the dojustasoften-part is repeated with inverse counting until the initial value is reached again:

proc F = : ⌈ var ν i := i_0 ;
while B do (α, i := q(i)) od ;
while i ≠ i_0 do (β, i := q^{-1}(i)) od ⌋

Frequently used operators q are : counting by one (up or down), counting by some integer, halving or doubling. In particular, q can frequently be chosen to coincide with some counting operation already existing in the program (for example, in primitive recursive functions). For procedures with parameters and results, we may derive the following transition scheme (note that h does not depend on A!):

```
proc F = (λ A) ρ :
         if   B[A]  then  h( F(A'))
                    else  p(A)     fi
```



```
proc F = (λ A) ρ :
         ⌜ var λ Y := A , var ν i := i0 ;
           while  B[Y]  do  Y := Y' , i := q(i)  od ;
           ⌜ var ρ Z := p(Y) ;
             while  i ≠ i0  do  Z := h(Z) , i := q^-1(i)  od ;
             Z                                                ⌟⌟
```

Typically, the iterative form contains now two repetitions and the initialization of a 'result variable' before entering the second repetition.

As a simple example we use (with $b \neq 0$!)

```
proc bip = (nat a, nat b) nat :
           if  b > 1  then  bip(a, b-1) ↑ 2
                      else  a              fi
```

This gives at first (note that a remains constant)

```
proc bip = (nat a, nat b) nat :
           ⌜ var nat y := b , var ν i := i0 ;
             while  y > 1  do  y := y-1,  i := q(i)  od ;
             ⌜ var nat z := a ;
               while  i ≠ i0  do  z := z ↑ 2,  i := q^-1(i)  od ;
               z                                              ⌟
```

Now putting ν = nat , q(i) = i-1 , i_0 = b ;

we can identify i with y and obtain with $q^{-1}(i)$ = i+1

```
proc bip = (nat a, nat b) nat :
           ⌈ var nat i := b ;
             while i > 1 do i := i-1 od ;
             ⌈ var nat z := a ;
               while i ≠ b do z := z ↑ 2, i := i+1 od ;
               z                                          ⌋⌋
```

In this special case, a further simplification is possible: the first two lines in the body of the procedure above can be replaced by

```
var nat i := 1
```

and thus

```
proc bip = (nat a, nat b) nat :
           ⌈var nat i := 1 ;
            ⌈var nat z := a ;
             while i ≠ b do z := z ↑ 2, i := i+1 od ;
             z
```

A more difficult, however, quite interesting example is given by (VV). Here mod occurs twice in the body of the definition. We shall see that one occurrence can be eliminated: From the definition (W) it is clear that

$$0 \leq \mathrm{mod}(a, 2\times b) < 2\times b$$

and thus easy to show that

```
mod(a, b) = if mod(a, 2×b) ≥ b then mod(a, 2×b) - b
                                else mod(a, 2×b)    fi .
```

Using this in a recursive definition similar to (VV), we have

(VVV)

```
proc mod = (nat a, nat b) nat :
    if a ≥ b then ⌈ if mod(a, 2×b) ≥ b
                    then mod(a, 2×b) - b
                    else mod(a, 2×b)   fi ⌋
             else a                        fi
```

Here our last transition scheme is not applicable, and we need the following generalization:

```
proc F = (λ A) ρ :
    if B[A] then h( F(A'), A)
            else p(A)        fi
```

(where $A' = d(A)$ and d is injective, d^{-1} being the inverse on the image of d)



```
proc F = (λ A) ρ :
    ⌈ var λ Y := A ;
      while B[Y] do Y := d(Y) od ;
      ⌈ var ρ Z := p(Y) ;
        while Y ≠ A do Y := d⁻¹(Y) ;
                       Z := h(Z, Y) od ;
        Z                              ⌋⌋
```

Note: We made use of d instead of q for the function inversion and identified $\nu = \lambda$, $i = Y$, $i_0 = A$.

The restrictions on d, however, are stronger than the ones we had put on q.

Using this transition scheme, we obtain from (VVV)

```
proc mod = (nat a, nat b) nat :
           ⌈ var nat r := a, var nat dd := b ;
             while r ≥ dd do r := r ; dd := 2 × dd od
             ⌈ var nat z := r ;
               while (r,dd) ≠ (a,b) do dd := dd/2 ;
                                    z := if z ≥ dd then z - dd
                                                   else z fi od ;
             z
```

This can be simplified:

```
proc mod = (nat a, nat b) nat :
           ⌈ var nat r := a, var nat dd := b ;
             while r ≥ dd do dd := 2 × dd od ;
(WW)         while dd ≠ b do dd := dd/2 ;
                             if r ≥ dd then r := r - dd fi od ;
             r                                                  ⌋
```

This is the usual program for division using binary number representation (cf. Dijkstra 1969 (EWD-249)), derived in a systematic way.

Third Lecture: DANGEROUS CORNERS

Sequentialization and the Danger of Destruction

The transition schemes between certain recursive and certain iterative notations show that the one form is as good as the other, apart from its appearance. When nevertheless programming with variables is considered to be dangerous (Variables considered harmful [22]), then it has reasons : Programming with variables is usually done in complete sequentialization, and sequentialization is considered harmful.

Sequentialization means replacing simultaneous assignments by a succession of simple assignments. Using auxiliary variables, this can always be done. Sometimes, however, some or all auxiliary variables can be saved, if the simple assignments are done in the right sequence. For example,

$$(y,z) := (y-1,\ y \times z)$$

can be rewritten, partly collaterally

$$(h_1 := y-1,\ \ h_2 := y \times z);\ (y := h_1,\ \ z := h_2)\ .$$

The sequence

$$h_1 := y-1;\ \ h_2 := y \times z;\ \ z := h_2;\ \ y := h_1$$

can be changed into

$$h_2 := y \times z;\ \ z := h_2;\ \ h_1 := y-1;\ \ y := h_1$$

and thus shortened to

$$z := y \times z\ ;\ \ y := y-1\ .$$

The sequence

$$h_2 := y \times z\ ;\ h_1 := y-1;\ y := h_1;\ \ z := h_2$$

can be shortened to

$$h_2 := y \times z;\ \ y := y-1;\ \ z := h_2\ .$$

Complete sequentialization is to be done with great care; and should be done mechanically. *Many programming mistakes occur when sequentialization is done hastily by mental work without mastering its complexity.* Sequentialization also destroys structure. Simultaneous assignments preserve structure [1]. This is particularly seen when a subprocedure is inserted as a macro in a repetition: If (U) is inserted into (A), we obtain

```
                proc gcd = (nat a, nat b) nat :
                    ⌈ var nat u := a , var nat v := b ;
                      while v > 0
(A_U)                 do (u,v) := (v, ⌈var nat x := u ;
                                      while x ≥ v do x := x-v od ;
                                      x                          ⌋) od ;
                      u
```

If this is completely sequentialized to a standard ALGOL form

```
                proc gcd = (nat a, nat b) nat :
                    ⌈ nat u , v , x ;
                      u := a ;
                      v := b ;
(A')                  while v > 0
                      do x := u ;
                         while x ≥ v
                         do x := x-v od ;
                         u := v ;
                         v := x          od ;
                      u                      ⌋
```

---

1) Transforming an arbitrary, completely sequential flow chart program into recursive form amounts to finding a simultaneous assignment comprising all assignments existing in one loop.

then some quite relevant structure is gone, structure that would be helpful in understanding the program.

Moreover, the declaration of x is moved to the begin, which means that x cannot be initialized there. It would be much better to keep and show at least some collaterality, for example

```
    :
while  v > 0
do  x := u ;
    (u := v, while  x ≥ v  do  x := x - v  od) ;
    v := x                                      od ;
u
```

By the way, (A') should be compared with the partially sequentialized iterative version of (BBB):

```
proc gcd  =  (nat  a, nat  b) nat :
          ⌈ nat  u, v ;
            u := a ;
            v := b ;
(A")        while   v > 0
            do  if   u ≥ v   then  u := u - v
                             else  (u,v) := (v,u)  fi  od ;
            u                                          ⌋
```

Sharing of Variables

Procedures may have variables as parameters. Related to the procedure (U) is the following procedure with one variable parameter x which holds the result:

```
proc mod* = (var nat x, nat b) :
              while x ≥ b do x := x-b od
```

and we get for (U)

```
proc mod = (nat a, nat b) nat :
          ⌈var nat x := a ;
           mod* (x , b); x ⌋
```

Now, since mod (u,v) occurs in (A) in a simultaneous assignment to (u,v), it may be replaced by ⌈mod* (u,v) ; u⌋ , thus we obtain

```
proc gcd = (nat a, nat b) nat :
          ⌈var nat u := a ; var nat v:= b ;
(A''')     while v > 0
              do (u, v) := (v, ⌈mod* (u, v) ; u ⌋) od ;
           u                                          ⌋
```

Inserting now mod* , we obtain in partially sequentialized form

```
proc gcd = (nat a, nat b) nat :
          ⌈nat u , v ;
           u := a ;
           v := b ;
(A'''')    while v > 0
           do while u ≥ v
              do u := u-v od ;
              (u, v) := (v, u) od ;
           u
```

While (A') uses three variables, (A'''') has only two: mod* shares one variable with gcd. The reader should note how subtle the insertion process is to be handled when sharing of variables is aimed at. Sharing of variables, if done with insufficient care, is likely to be the source of mistakes, too.

By the way, (A'''') should be compared with (A")!

The Method of Invariants

Proving correctness of programs involving programming variables can be done with the help of suitable invariants. This now classical method (FLOYD 1967 [23], DIJKSTRA 1968 [24], HOARE 1969 [32], NAUR 1969 [25], KING 1969 [26]) can be demonstrated with the examples (I), (U), (A): In (I), z×fact (y) remains invariant under z := z × y, y := y-1 and is 1×fact(n) originally, z×fact(0) finally; thus z equals fact(n) and yields the result. But what are we using in this proof? We use the fact that

$$y \times fact(y-1) \text{ equals } fact(y) \text{ and that}$$
$$fact(0) \text{ equals } 1$$

This is nothing than the definition which is expressed in the program (M).

Likewise, for (U), mod(x,b) remains invariant under x := x-b and is mod(a,b) originally; since finally x < b, mod(x,b) is finally x , which yields the result. Again, this proof uses that

$$\text{for } x \geq b : \quad mod(x-b,b) \text{ equals } mod(x,b)$$
$$\text{for } x < b : \quad mod(x,b) \text{ equals } x$$

which is the essence of the recursive definition (V). Similarly (A) can be treated. (We should also consider 'the invariant of the repetition' to be quite generally a predicate (an 'assertion'), in our case the predicates z×fact(y) = fact(n) and mod(x,b) = mod(a,b), resp.)

Looking at these examples, and knowing that there is a deep connection between iterative and recursive forms, we should not be surprised that finding an invariant means finding the appropriate embedding (see (M')) of the recursively defined procedure, and proving correctness of the iterative form amounts to using the recursive definition. (The example of the factorial shows, that a COOPER transformation involved in the derivation of the procedure makes no extra difficulties for the proof process.)

Anyhow, proving program correctness is a useful and quite practical instrument, *provided a program is given that can be assumed to be correct.* We know, however, that this assumption can not be expected to be true in the usual situation: even 'small mistakes', like writing $<$ instead of $\leq$ or y instead of x, usually make a program totally wrong. What will happen with such a program when the method of invariants is tried? Looking for 'the' invariant is now a difficult task, since the 'right' invariant will not be invariant and an expression that is invariant indeed may be hard to find - and if so, is quite useless.

Moreover, proving program correctness *is easy only if the invariant is not hard to discover.* Given a complicated program without any additional information, finding the invariant may not be feasible with reasonable effort.

On the other hand the more complicated the repetition structure of a program is, the more it is necessary to prove its correctness.

This seems to produce a practical conflict. We should, however, keep in mind that we normally do not start with a program that falls from heaven. *We must develop the program and the correctness proof together.* After all, if an assertion is found to be an invariant of an iterative program this means that a recursive definition is known and, in effect, is used as we have seen above. This recursion could be used from the very beginning *and the corresponding iterative form could as well be obtained by a series of program transformations* - correctly, if no clerical mistake is made. The method of invariants can, however, be used to check the correctness of the transformed result of a transformation.

It is more important to be able to derive a correct program than to prove its correctness, although the moral value of a direct correctness proof should not be neglected.

A last remark about finding an invariant: If the program (WW) for binary division is given without its systematic derivation, finding an invariant for proving its correctness is much simplified if the program is supplemented by a variable q and suitable assignments

```
proc mod = (nat a, nat b) nat :
        ⌈ var nat r := a, var nat dd := b, var nat q := 0 ;
          while r ≥ dd do dd := 2 × dd od ;
(WW')     while dd ≠ b do dd := dd/2 , q := 2 × q ;
                          if r ≥ dd then r := r - dd , q := q+1 fi od ;
          r
```

In this case $r + q \times dd$ is the invariant, equal to a initially and $r + q \times b$ finally, with $0 \leq r < b$. The additional variable q , of course, gives the quotient! Simple, but only if you know it beforehand.

Conclusion: PROGRAMMING AS A PROCESS

Programming starts with a problem and ends with an efficient solution. 'To have an idea' frequently amounts to finding a solution irrespective of how inefficient it is, and it is hard to say what is more difficult: finding a solution or refining it.

Programming is program development. Development goes in small steps: Programming goes stepwise. Independent of the architecture of the storage-controlled machine, program development is refinement. Thus, programming is done by stepwise refinement.

Refinement may mean: substructuring the operations. It may also mean: substructuring the objects, introducing 'data structures'. Frequently, it is done by joint refinement. *Thus, programming means structuring.*

Programming as a process that is done in steps can be easily understood if it preserves the structure that exists already: Structured programming *is structure-preserving programming and substructuring at the same time. It is done by* stepwise program composition, *it is* programming by action clusters, *it leads to a* system hierarchy.
Starting with the problem and ending with an efficient solution means going from the problem, which is considered top, to the bottom of the machine. Programming is top-down program development, *which sometimes needs backtracking: in complicated cases, programming means an* iterative multi-level modelling *of the problem.*

The lines above comprise the catchwords which mark the revolution in programming that started ten years ago (see also [27]). A programming methodology had to be developped. We tried in the foregoing lectures to follow this doctrine and to illustrate it.

The revolution in programming was motivated by the 'software crisis' of the sixties, the fact that large software systems showed alarming deficiencies Analyzing the reasons, it was found that new ways were needed in teaching programming. Teaching should not start with the machine. Teaching should be done in the same direction as the program development goes: it should start with concepts necessary in the problem definition, and should add step by step the concepts needed on the way to more and more efficient programs. This *top-down teaching* has been used with great success at the Technical University Munich since 1964 [28] and is today widely replacing the former teaching style 'upwards from the bit'.

The next logical step is to support the programming process by the machine. The compiler is only the *ultima ratio*, programming is a transformation process that can be done rationally, according to rules, without intellectual hazards. Rational steps, however, can be supervised by the machine. Program documentation, supported by the machine, will list all sorts of intermediate versions of the program to be developed.

Programming can be done, as we have seen in the first lecture, in different styles. Not only does it create aversions if style is squeezed into the narrowness of a language, it is typical of good programming that refinement is done locally to different depths. Rather than having transitions between fixed language layers, one needs computer-aided, intuition-controlled programming in *one simple programming language*, which allows the widest possible spectrum of style. In the foregoing lectures we tried to show what this would look like, and what particular difficulties are to be expected. It is essential that programming has found a unified, conceptual basis [29] despite hundreds of so-called programming languages [30]. Moreover, programming is neither a software nor a hardware domain, there exists a uniformity of software and hardware systems [31].

Computer-aided, intuition-controlled programming is the next goal. Intuition is helpful, if not mandatory, if efficiency is aimed at. Intuition, however, is dangerous if it is not controlled by ratiocination. The computer can perform mechanical operations more safely than the human being. Thus, human intuition can be freed from the burden of clerical work. This should not only turn out to be of great help in education, it should in the end also contribute to professional programming.

References

[1] W.L. van der Poel — Micro-programming and Trickology. In: Digitale Informationswandler. Braunschweig 1962

[2] J.J. Horning, B. Randell — Structuring Complex Processes. Marktoberdorf Summer School 197o

[3] O.-J. Dahl — Hierarchical Program Structures. Marktoberdorf Summer School 197o

[4] C.A.R. Hoare — Data Structures. Marktoberdorf Summer School 197o

[5] R. Bayer — On the Structure of Data and Application Programs. Marktoberdorf Summer School 1971

[6] A. Wang, O.-J. Dahl — Coroutine Sequencing in a Block Structured Environment Marktoberdorf Summer School 1971

[7] C.A.R. Hoare — Program Proving. Marktoberdorf Summer School 1971

[8] E.W. Dijkstra — Hierarchical Ordering of Sequential Processes. Marktoberdorf Summer School 1971

[9] P. Brinch Hansen — Concurrent Programming Concepts. Marktoberdorf Summer School 1973

[1o] R. Gnatz — Sets and Predicates in Programming Languages. Marktoberdorf Summer School 1973

[11] C.A.R. Hoare, N. Wirth — An Axiomatic Definition of the Programming Language PASCAL. Marktoberdorf Summer School 1973

[12] W.M. Turski — Morphology of Data. Marktoberdorf Summer School 1973

[13] E.W. Dijkstra — A Simple Axiomatic Basis for Programming Language Constructs. Marktoberdorf Summer School 1973

[14] K. Zuse — Der Plankalkül. Veröffentlicht durch Gesellschaft für Mathematik und Datenverarbeitung, Bericht Nr. 63, Bonn 1972

[15] S.C. Kleene — General Recursive Functions of Natural Numbers. Math. Ann. 112, 727-742 (1936)

[16] Z. Manna, S. Ness, J. Vuillemin — Inductive Methods for Proving Properties of Programs. Comm. ACM 16, 491-5o2 (1973)

[17] D.C. Cooper — The Equivalence of Certain Computations. Comp. J. 9, 45-52 (1966)

[18] J. Darlington, R.M. Burstall — A System which Automatically Improves Programs. Proc. Third Internat. Conf. on Artif. Intell., Stanford 1973, 479-485

[19] J.H. Morris — Another Recursive Induction Principle. Comm. ACM 14, 351-354 (1971)

[2o] H.R. Strong — Translating Recursive Equations into Flow Charts. In: Proc. 2nd Annual ACM Symposium on Theory of Computing, New York 197o, 184-197. Also J. CSS 5, 254-285 (1971)

[21] H. Kröger — Bemerkungen zur Auflösung von Rekursionen. In: Seminar über Methodik des Programmierens, TU München, Interner Bericht, 1974, 45-62

[22] F.L. Bauer — Variables Considered Harmful. Appendix to A Philosophy of Programming. TUM Report 7513, 1975 (and in this volume)

[23] R.W. Floyd — Assigning Meanings to Programs. In: Schwartz, J.T. (ed.): Mathematical Aspects of Computer Science. Proc. Symposia in Applied Mathematics 19. Providence (R.I.): Amer. Math. Soc. 1967, 19-32

[24] E.W. Dijkstra — A Constructive Approach to the Problem of Program Correctness. BIT 8. 174-186 (1968)

[25] P. Naur — Programming by Action Clusters. BIT 9, 25o-258 (1969)

[26] J.C. King — A Program Verifier. Ph.D. Thesis, Carnegie-Mellon University, Pittsb.(Pa.), 1969

[27] F.L. Bauer (ed.) — Software Engineering - An Advanced Course. Lecture Notes in Computer Science 30, Springer 1975

[28] F.L. Bauer, G. Goos — Informatik - Eine einführende Übersicht. 2 Vols. (2nd ed.) Springer 1973, 1974

[29] F.L. Bauer — A Unified, Conceptual Basis of Programming. In: A Philosophy of Programming (London Lectures 1973) TUM Report 7513, 1975 (and in this volume)

[3o] P.J. Landin — The Next 7oo Programming Languages. Comm. ACM 9, 157-164 (1966)

[31] F.L. Bauer — System Uniformity of Software and Hardware. In: A Philosophy of Programming (London Lectures 1973) TUM Report 7513, 1975

[32] C.A.R. Hoare — An Axiomatic Basis for Computer Programming. Comm. ACM 12, 576-58o, 583 (1969)

C. A. R. Hoare

The Queen's University of Belfast

Belfast, Northern Ireland

Proof of Correctness of Data Representations

Summary. A powerful method of simplifying the proofs of program correctness is suggested; and some new light is shed on the problem of functions with side-effects.

1. Introduction

In the development of programs by stepwise refinement [1–4], the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" program. A similar suggestion was made more formally in algebraic terms in [5], which gives a general definition of simulation. However, a more restricted definition may prove to be more useful in practical program proofs.

If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorising it in this way. Furthermore, the two parts of the proof correspond to the successive stages in program development, thereby contributing to a constructive approach to the correctness of programs [6]. Finally, it must be recalled that in the case of larger and more complex programs the description given above in terms of two stages readily generalises to multiple stages.

2. Concepts and Notations

Suppose in an abstract program there is some abstract variable t which is regarded as being of type T (say a small set of integers). A concrete representation of t will usually consist of several variables $c_1, c_2, \ldots, c_n$ whose types are directly (or more directly) represented in the computer store. The primitive operations on the variable t are represented by procedures $p_1, p_2, \ldots, p_m$, whose bodies carry out on the variables $c_1, c_2, \ldots, c_n$ a series of operations directly (or more directly) performed by computer hardware, and which correspond to meaningful operations on the abstract variable t. The entire concrete representation of the type T can

be expressed by declarations of these variables and procedures. For this we adopt the notation of the SIMULA 67 [7] class declaration, which specifies the association between an abstract type T and its concrete representation:

class T;
begin ... declarations of $c_1, c_2, \ldots, c_n \ldots$;
procedure p_1 ⟨formal parameter part⟩; Q_1;
procedure p_2 ⟨formal parameter part⟩; Q_2;
. . . . . (1)
procedure p_m ⟨formal parameter part⟩; Q_m;
Q
end;

where Q is a piece of program which assigns initial values (if desired) to the variables $c_1, c_2, \ldots, c_n$. As in ALGOL 60, any of the p's may be functions; this is signified by preceding the procedure declaration by the type of the procedure.

Having declared a representation for a type T, it will be required to use this in the abstract program to declare all variables which are to be represented in that way. For this purpose we use the notation:

var(T) t;

or for multiple declarations:

var(T) $t_1, t_2, \ldots$;

The same notation may be used for specifying the types of arrays, functions, and parameters. Within the block in which these declarations are made, it will be required to operate upon the variables $t, t_1, \ldots$, in the manner defined by the bodies of the procedures $p_1, p_2, \ldots, p_m$. This is accomplished by introducing a compound notation for a procedure call:

$$t_i \cdot p_j \text{ ⟨actual parameter part⟩};$$

where t_i names the variable to be operated upon and p_j names the operation to be performed.

If p_j is a function, the notation displayed above is a function designator; otherwise it is a procedure statement. The form $t_i \cdot p_j$ is known as a *compound identifier*.

These concepts and notations have been closely modelled on those of SIMULA 67. The only difference is the use of **var**(T) instead of **ref**(T). This reflects the fact that in the current treatment, objects of declared classes are not expected to be addressed by reference; usually they will occupy storage space contiguously in the local workspace of the block in which they are declared, and will be addressed by offset in the same way as normal integer and real variables of the block.

3. Example

As an example of the use of these concepts, consider an abstract program which operates on several small sets of integers. It is known that none of these sets ever has more than a hundred members. Furthermore, the only operations

actually used in the abstract program are the initial clearing of the set, and the insertion and removal of individual members of the set. These are denoted by procedure statements

$$s \cdot \text{insert}(i)$$

and

$$s \cdot \text{remove}(i).$$

There is also a function "$s \cdot \text{has}(i)$", which tests whether i is a member of s.

It is decided to represent each set as an array A of 100 integer elements, together with a pointer m to the last member of the set; m is zero when the set is empty. This representation can be declared:

```
class smallintset;
begin integer m; integer array A [1:100];

        procedure insert(i); integer i;
          begin integer j;
            for j:=1 step 1 until m do
              if A[j]=i then go to end insert;
              m:=m+1;
              A[m]:=i;

    end insert: end insert;

        procedure remove(i); integer i;
          begin integer j, k;
            for j:=1 step 1 until m do
              if A[j]=i then
                begin for k:=j+1 step 1 until m do A[k-1]:=A[k];
                  comment close the gap over the removed member;
                  m:=m-1;
                  go to end remove
                end;

end remove: end remove;

        Boolean procedure has (i); integer i;
          begin integer j;
                has:=false;
                for j:=1 step 1 until m do
                  if A[j]=i then
                    begin has:=true; go to end contains end;

end contains: end contains;

    m:=0; comment initialise set to empty;

end smallintset;
```

Note: as in SIMULA 67, simple variable parameters are presumed to be called by value.

4. Semantics and Implementation

The meaning of class declarations and calls on their constituent procedures may be readily explained by textual substitution; this also gives a useful clue to a practical and efficient method of implementation. A declaration:

$$\textbf{var}(T)t;$$

is regarded as equivalent to the unbracketed body of the class declaration with **begin** ... **end** brackets removed, after every occurrence of an identifier c_i or p_i declared in it has been prefixed by "$t\cdot$". If there are any initialising statements in the class declaration these are removed and inserted just in front of the compound tail of the block in which the declaration is made. Thus if T has the form displayed in (1), **var**$(T)t$ is equivalent to:

... declarations for $t \cdot c_1, t \cdot c_2, \ldots, t \cdot c_n \ldots$;
procedure $t \cdot p_1(\ldots)$; Q_1';
procedure $t \cdot p_2(\ldots)$; Q_2';
.....
procedure $t \cdot p_m(\ldots)$; Q_m';

where $Q_1', Q_2', \ldots, Q_m', Q'$ are obtained from $Q_1, Q_2, \ldots, Q_m, Q$ by prefixing every occurrence of $c_1, c_2, \ldots, c_n, p_1, p_2, \ldots, p_m$ by "$t\cdot$". Furthermore, the initialising statement Q' will have been inserted just ahead of the statements of the block body.

If there are several variables of class T declared in the same block, the method described above can be applied to each of them. But in a practical implementation, only one copy of the procedure bodies will be translated. This would contain as an extra parameter an address to the block of $c_1, c_2, \ldots, c_n$ on which a particular call is to operate.

5. Criterion of Correctness

In an abstract program, an operation of the form

$$t_i \cdot p_j(a_1, a_2, \ldots, a_{nj}) \tag{2}$$

will be expected to carry out some transformation on the variable t_i, in such a way that its resulting value is $f_j(t_i, a_1, a_2, \ldots, a_{nj})$, where f_j is some primitive operation required by the abstract program. In other words the procedure statement is expected to be equivalent to the assignment

$$t_i := f_j(t_i, a_1, a_2, \ldots, a_{nj});$$

When this equivalence holds, we say that p_j *models* f_j. A similar concept of modelling applies to functions. It is desired that the proof of the abstract program may be based on the equivalence, using the rule of assignment [8], so that for any propositional formula S, the abstract programmer may assume:

$$S^{t_i}_{f_j(t_i, a_1, a_2, \ldots, a_{nj})} \{t_i \cdot p_j(a_1, a_2, \ldots, a_{nj})\} S.^1$$

1 S^x_y stands for the result of replacing all free occurrences of x in S by y: if any free variables of y would become bound in S by this substitution, this is avoided by preliminary systematic alteration of bound variables in S.

In addition, the abstract programmer will wish to assume that all declared variables are initialised to some designated value d_0 of the abstract space.

The criterion of correctness of a data representation is that every p_j models the intended f_j and that the initialisation statement "models" the desired initial value; and consequently, a program operating on abstract variables may validly be replaced by one carrying out equivalent operations on the concrete representation.

Thus in the case of smallintset, we require to prove that:

$$\begin{aligned}
&\textbf{var}(i)t \text{ initialises } t \text{ to } \{\ \} \text{ (the empty set)}\\
&t \cdot \text{insert } (i) \equiv t := t \cup \{i\}\\
&t \cdot \text{remove}(i) \equiv t := t \cap \neg\{i\}\\
&t \cdot \text{has}(i) \equiv i \in t.
\end{aligned} \tag{3}$$

6. Proof Method

The first requirement for the proof is to define the relationship between the abstract space in which the abstract program is written, and the space of the concrete representation. This can be accomplished by giving a function $\mathscr{A}(c_1, c_2, \ldots, c_n)$ which maps the concrete variables into the abstract object which they represent. For example, in the case of smallintset, the representation function can be defined as

$$\mathscr{A}(m, A) = \{i : \text{integer} \mid \exists k (1 \leqq k \leqq m \,\&\, A[k] = i)\} \tag{4}$$

or in words, "(m, A) represents the set of values of the first m elements of A". Note that in this and in many other cases $\mathscr{A}$ will be a many-one function. Thus there is no unique concrete value representing any abstract one.

Let t stand for the value of $\mathscr{A}(c_1, c_2, \ldots, c_m)$ before execution of the body Q_j of procedure p_j. Then what we must prove is that after execution of Q_j the following relation holds:

$$\mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{nj})$$

where $v_1, v_2, \ldots, v_{nj}$ are the formal parameters of p_j.

Using the notations of [8], the requirement for proof may be expressed:

$$t = \mathscr{A}(c_1, c_2, \ldots, c_n)\ \{Q_j\}\ \mathscr{A}(c_1, c_2, \ldots, c_n) = f_j(t, v_1, v_2, \ldots, v_{nj})$$

where t is a variable which does not occur in Q_j. On the basis of this we may say: $t \cdot p_j(a_1, a_2, \ldots, a_n) \equiv t := f_j(t, a_1, a_2, \ldots, a_n)$ with respect to $\mathscr{A}$. This deduction depends on the fact that no Q_j alters or accesses any variables other than c_1, $c_2, \ldots, c_n$; we shall in future assume that this constraint has been observed.

In fact for practical proofs we need a slightly stronger rule, which enables the programmer to give an invariant condition $I(c_1, c_2, \ldots, c_n)$, defining some relationship between the constituent concrete variables, and thus placing a constraint on the possible combinations of values which they may take. Each operation (except initialisation) may assume that I is true when it is first entered; and each operation must in return ensure that it is true on completion.

In the case of smallintset, the correctness of all operations depends on the fact that m remains within the bounds of A, and the correctness of the remove operation is dependent on the fact that the values of $A[1], A[2], \ldots, A[m]$ are all different; a simple expression of this invariant is:

$$\text{size}(\mathscr{A}(m, A)) = m \leqq 100. \qquad (I)$$

One additional complexity will often be required; in general, a procedure body is not prepared to accept arbitrary combinations of values for its parameters, and its correctness therefore depends on satisfaction of some precondition $P(t, a_1, a_2, \ldots, a_n)$ before the procedure is entered. For example, the correctness of the insert procedure depends on the fact that the size of the resulting set is not greater than 100, that is

$$\text{size}(t \cup \{i\}) \leqq 100$$

This precondition (with t replaced by $\mathscr{A}$) may be assumed in the proof of the body of the procedure; but it must accordingly be proved to hold before every call of the procedure.

It is interesting to note that any of the p's that are functions may be permitted to change the values of the c's, on condition that it preserves the truth of the invariant, and also that it preserves unchanged the value of the abstract object $\mathscr{A}$. For example, the function "has" could reorder the elements of A; this might be an advantage if it is expected that membership of some of the members of the set will be tested much more frequently than others. The existence of such a concrete side-effect is wholly invisible to the abstract program. This seems to be a convincing explanation of the phenomenon of "benevolent side-effects", whose existence I was not prepared to admit in [8].

7. Proof of Smallintset

The proof may be split into four parts, corresponding to the four parts of the class declaration:

7.1. Initialisation

What we must prove is that after initialisation the abstract set is empty and that the invariant I is true:

$$\textbf{true}\ \{m := 0\}\ \{i \mid \exists k (1 \leqq k \leqq m \,\&\, A[k] = i)\} = \{\}$$
$$\&\ \text{size}(\mathscr{A}(m, a)) = m \leqq 100$$

Using the rule of assignment, this depends on the obvious truth of the lemma

$$\{i \mid \exists k (1 \leqq k \leqq 0 \,\&\, A[k] = i\} = \{\} \,\&\, \text{size}(\{\}) = 0 \leqq 100$$

7.2. Has

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I\{Q_{\text{has}}\}\mathscr{A}(m, A) = k \,\&\, I \,\&\, \text{has} = i \in \mathscr{A}(m, A)$$

where Q_{has} is the body of has. Since Q_{has} does not change the value of m or A, the truth of the first two assertions on the right hand side follows directly from their truth beforehand. The invariant of the loop inside Q_{has} is:

$$j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j, A)$$

as may be verified by a proof of the lemma:

$$\begin{aligned} & j < m \,\&\, j \leqq m \,\&\, \text{has} = i \in \mathscr{A}(j, A) \\ \supset\ & \textbf{if } A[j+1] = i \textbf{ then } \big(\textbf{true} = i \in \mathscr{A}(m, A)\big) \\ & \qquad\qquad \textbf{else } \text{has} = i \in \mathscr{A}(j+1, A). \end{aligned}$$

Since the final value of j is m, the truth of the desired result follows directly from the invariant; and since the "initial" value of j is zero, we only need the obvious lemma

$$\textbf{false} = i \in \mathscr{A}(0, A)$$

7.3. Insert

What we must prove is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I\{Q_{\text{insert}}\}\mathscr{A}(m, A) = (k \cup \{i\}) \,\&\, I,$$

where $P \equiv_{\text{df}} \text{size}\big(\mathscr{A}(m, A) \cup \{i\}\big) \leqq 100$.

The invariant of the loop is:

$$P \,\&\, \mathscr{A}(m, A) = k \,\&\, I \,\&\, i \,\tilde{\in}\, \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \tag{6}$$

as may be verified by the proof of the lemma

$$\begin{aligned} & \mathscr{A}(m, A) = k \,\&\, i \,\tilde{\in}\, \mathscr{A}(j, A) \,\&\, 0 \leqq j \leqq m \,\&\, j < m \supset \\ & \qquad \textbf{if } A[j+1] = i \textbf{ then } \mathscr{A}(m, A) = (k \cup \{i\}) \\ & \qquad\qquad \textbf{else } 0 \leqq j+1 \leqq m \,\&\, i \,\tilde{\in}\, \mathscr{A}(j+1, A) \end{aligned}$$

(The invariance of $P \,\&\, \mathscr{A}(m, A) = k \,\&\, I$ follows from the fact that the loop does not change the values of m or A). That (6) is true before the loop follows from $i \,\tilde{\in}\, \mathscr{A}(0, A)$.

We must now prove that the truth of (6), together with $j = m$ at the end of the loop, is adequate to ensure the required final condition. This depends on proof of the lemma

$$j = m \,\&\, (6) \cup \mathscr{A}(m+1, A') = (k \cup \{i\}) \,\&\, \text{size}\big(\mathscr{A}(m+1, A')\big) = m+1 \leqq 100$$

where $A' = (A, m+1 : i)$ is the new value of A after assignment of i to $A[m+1]$.

7.4. Remove

What we must prove is

$$\mathscr{A}(m, A) = k \,\&\, I\{Q_{\text{remove}}\}\mathscr{A}(m, A) = (k \cap \neg\{i\}) \,\&\, I.$$

The details of the proof are complex. Since they add nothing more to the purpose of this paper, they will be omitted.

8. Formalities

Let T be a class declared as shown in Section 2, and let $\mathscr{A}$, I, P_j, f_j be formulae as explained in Section 6 (free variable lists are omitted where convenient). Suppose also that the following $m+1$ theorems have been proved:

$$\textbf{true}\ \{Q\}\, I \,\&\, \mathscr{A} = d_0 \tag{7}$$

$$\mathscr{A} = t \,\&\, I \,\&\, P_j(t)\ \{Q_j\}\, I \,\&\, \mathscr{A} = f_j(t) \quad \text{for procedure bodies } Q_j \tag{8}$$

$$\mathscr{A} = t \,\&\, I \,\&\, P_j(t)\ \{Q_j\}\, I \,\&\, \mathscr{A} = t \,\&\, p_j = f_j(t) \quad \text{for function bodies } Q_j. \tag{9}$$

In this section we show that the proof of these theorems is a sufficient condition for the correctness of the data representation, in the sense explained in Section 5.

Let X be a program beginning with a declaration of a variable t of an abstract type, and initialising it to d_0. The subsequent operations on this variable are of the form

(1) $t := f_j(t, a_1, a_2, \ldots, a_{nj})$ if Q_j is a procedure

(2) $f_j(t, a_1, a_2, \ldots, a_{nj})$ if Q_j is a function.

Suppose also that $P_j(t, a_1, a_2, \ldots, a_{nj})$ has been proved true before each such operation.

Let X' be a program formed from X by replacements described in Section 4, as well as the following (see Section 5):

(1) initialisation $t := d_0$ replaced by Q'

(2) $t := f_j(t, a_1, a_2, \ldots, a_{nj})$ replaced by $t \cdot p_j(a_1, a_2, \ldots, a_{nj})$

(4) $f_j(t, a_1, a_2, \ldots, a_{nj})$ by $t \cdot p_j(a_1, a_2, \ldots, a_{nj})$.

Theorem. Under conditions described above, if X and X' both terminate, the value of t on termination of X will be $\mathscr{A}(c_1, c_2, \ldots, c_n)$, where $c_1, c_2, \ldots, c_n$ are the values of these variables on termination of X'.

Corollary. If $R(t)$ has been proved true on termination of X, $R(\mathscr{A})$ will be true on termination of X'.

Proof. Consider the sequence S of operations on t executed during the computation of X, and let S' be the sequence of subcomputations of X' arising from execution of the procedure calls which have replaced the corresponding operations on t in X. We will prove that there is a close elementwise correspondence between the two sequences, and that

(a) each item of S' is the very procedure statement which replaced the corresponding operation in S.

(b) the values of all variables (and hence also the actual parameters) which are common to both "programs" are the same after each operation.

(c) the invariant I is true between successive items of S'.

(d) if the operations are function calls, their results in both sequences are the same.

(e) and if they are procedure calls (or the initialisation) the value of t immediately after the operation in S is given by $\mathscr{A}$, as applied to the values of c_1, c_2, ..., c_n after the corresponding operation in S'.

It is this last fact, applied to the last item of the two sequences, that establishes the truth of the theorem.

The proof is by induction on the position of an item in S.

(1) *Basis.* Consider its first item of S, $t:=d_0$. Since X and X' are identical up to this point, the first item of S' must be the subcomputation of the procedure Q which replaced it, proving (a). By (7), I is true after Q in S', and also $\mathscr{A}=d_0$, proving (c) and (e). (d) is not relevant. Q is not allowed to change any non-local variable, proving (b).

(2) *Induction step.* We may assume that conditions (a) to (e) hold immediately after the $(n-1)$-th item of S and S', and we establish that they are true after the n-th. Since the value of all other variables (and the result, if a function) were the same after the previous operation in both sequences, the subsequent course of the computation must also be the same until the very next point at which X' differs from X. This establishes (a) and (b). Since the only permitted changes to the values of $t \cdot c_1$, $t \cdot c_2$, ..., $t \cdot c_n$ occur in the subcomputations of S', and I contains no other variables, the truth of I after the previous subcomputation proves that it is true before the next. Since S contains *all* operations on t, the value of t is the same before the n-th as it was after the $(n-1)$-th operation, and it is still equal to $\mathscr{A}$. It is given as proved that the appropriate $P_j(t)$ is true before each call of f_j in S. Thus we have established that $\mathscr{A}=t\&I\&P_j(t)$ is true before the operation in S'. From (8) or (9) the truth of (c), (d), (e) follows immediately. (b) follows from the fact that the assignment in S changes the value of no other variable besides t; and similarly, Q_j is not permitted to change the value of any variable other than $t \cdot c_1$, $t \cdot c_2$, ..., $t \cdot c_n$.

This proof has been an informal demonstration of a fairly obvious theorem. Its main interest has been to show the necessity for certain restrictive conditions placed on class declarations. Fortunately these restrictions are formulated as scope rules, which can be rigorously checked at compile time.

9. Extensions

The exposition of the previous sections deals only with the simplest cases of the Simula 67 class concept; nevertheless, it would seem adequate to cover a wide range of practical data representations. In this section we consider the possibility of further extensions, roughly in order of sophistication.

9.1. Class Parameters

It is often useful to permit a class to have formal parameters which can be replaced by different actual parameters whenever the class is used in a declaration. These parameters may influence the method of representation, or the identity

of the initial value, or both. In the case of smallintset, the usefulness of the definition could be enhanced if the maximum size of the set is a parameter, rather than being fixed at 100.

9.2. Dynamic Object Generation

In Simula 67, the value of a variable c of class C may be reinitialised by an assignment:

$$c := \textbf{new}\ C\ \langle\text{actual parameter part}\rangle;$$

This presents no extra difficulty for proofs.

9.3. Remote Identification

In many cases, a local concrete variable of a class has a meaningful interpretation in the abstract space. For example, the variable m of smallintset always stands for the size of the set. If the main program needs to test the size of the set, it would be possible to make this accessible by writing a function

integer procedure size; size:=m;

But it would be simpler and more convenient to make the variable more directly accessible by a compound identifier, perhaps by declaring it

public integer m;

The proof technique would specify that

$$m = \text{size}\,(\mathscr{A}(m, A))$$

is part of the invariant of the class.

9.4. Class Concatenation

The basic mechanism for representing sets by arrays can be applied to sets with members of type or class other than just integers. It would therefore be useful to have a method of defining a class "smallset", which can then be used to construct other classes such as "smallrealset" or "smallcarset", where "car" is another class. In SIMULA 67, this effect can be achieved by the class/subclass and virtual mechanisms.

9.5. Recursive Class Declaration

In Simula 67, the parameters of a class, or of a local procedure of the class, and even the local variables of a class, may be declared as belonging to that very same class. This permits the construction of lists and trees, and their processing by recursive procedure activation. In proving the correctness of such a class, it will be necessary to assume the correctness of all "recursive" operations in the proofs of the bodies of the procedures. In the implementation of recursive classes, it will be necessary to represent variables by a null pointer (**none**) or by the *address* of their value, rather than by direct inclusion of space for their

values in block workspace of the block to which they are local. The reason for this is that the amount of space occupied by a value of recursively defined type cannot be determined at compile time.

It is worthy of note that the proof-technique recommended above is valid only if the data structure is "well-grounded" in the sense that it is a pure tree, without cycles and without convergence of branches. The restrictions suggested in this paper make it impossible for local variables of a class to be updated except by the body of a procedure local to that very same activation of the class; and I believe that this will effectively prevent the construction of structures which are not well-grounded, provided that assignment is implemented by copying the complete value, not just the address.

I am deeply indebted to Doug Ross and to all authors of referenced works. Indeed, the material of this paper represents little more than my belated understanding and formalisation of their original work.

References

1. Wirth, N.: The development of programs by stepwise refinement. Comm. ACM. **14**, 221–227 (1971).
2. Dijkstra, E. W.: Notes on structured programming. In Structured Programming. Academic Press (1972).
3. Hoare, C. A. R.: Notes on data structuring. *Ibid.*
4. Dahl, O.-J.: Hierachical program structures. *Ibid.*
5. Milner, R.: An algebraic definition of simulation between programs. CS 205 Stanford University, February 1971.
6. Dijkstra, E. W.: A constructive approach to the problem of program correctness. BIT. **8**, 174–186 (1968).
7. Dahl, O.-J., Myhrhaug, B., Nygaard, K.: The SIMULA 67 common base language. Norwegian Computing Center, Oslo, Publication No. S-22, 1970.
8. Hoare, C. A. R.: An axiomatic approach to computer programming. Comm. ACM. **12**, 576–580, 583 (1969).

Prof. C. A. R. Hoare
Computer Science
The Queen's University of Belfast
Belfast BT 71 NN
Northern Ireland

F. L. Bauer

Technical University Munich
Germany

A Philosophy of Programming

A Course of 3 Lectures
given at the
Imperial College of Science and Technology
University of London
on 3, 4 and 5 October 1973

2nd printing, revised and supplemented by an Appendix:

Variables considered harmful

It is an old experience that a gap frequently exists between theory and practice. One should therefore not be astonished to find such a situation in the field of programming. Indeed, both in the environments of Los Angeles and Novosibirsk, a balance has to be found and maintained between scientific and economic considerations. Moreover a third aspect, the formation of programmers, must be taken into account, though its importance has, so far, been unrecognised. Thus, my attempt to lecture on a philosophy of programming must be seen against a background of rivalry between scientific, economic and educational considerations.

Fortunately, I have been exposed in my life to all these aspects, and I could, from intimate knowledge, give an ardent plea for each one of them. Instead, I shall take it for granted that a compromise must be established. I am not necessarily saying that the right balance does already exist, or in fact, I consider that in some places more scientific considerations dominate the scene, while at other places economic considerations (guess where!), and, of course, also educational ones may be given undue weight. This is thus the problem: That undue weight might be given to any one of the three aspects mentioned.

Of course, this can happen only if something protects an institution that does so: The ivory tower of the research establishment allows scientists to look down with disdain upon economic, and sometimes also upon educational questions.[1] Likewise, financial power enables the economist - at least the shortsighted one - to get away with neglecting science[2] and with no more than lip-service paid to education. Manpower shortages, finally, gives the educator a chance to neglect scientific and economic factors.

Assuming, that a compromise is not only in the best interest of all parties, but that it even may be reached one day, I am aware and ready to bear the reproach of a possible accusation that I am an incorrigible idealist. Indeed, what I am going to say will bring me repeatedly into this danger, it is an idealist's philosophy of programming, and it is somewhat programmatic in character, intended to stimulate further research.

---

[1] In a touch of self-irony, ALAN PERLIS has characterized this as follows: "Computer Science is able to generate its own problems, understood and appreciated by only its own people ... so the thing is quite healthy in the academic sense."

[2] As JERRY FELDMAN said: "Practical people just do not read the literature, even though there are sometimes real solutions to their problems to be found there."

First Lecture:

A unified, conceptual, basis of programming

The theme of this lecture raises the question: Why do we need a unified, conceptual, basis of programming? The answer is, that we can give scientific, economic and educational reasons. This will be elaborated in the sequel, which will include a discussion of the (more difficult) question whether such a unified, conceptual, basis can exist.

First, however, we should clarify the situation. Difference of notation does not necessarily mean difference of concepts. This may trivially be so right down to the choice of characters. The figure >>one<< , handwritten, looks in some countries like | , in others 1 , and the figure >>seven<< may appear as 7 or as 7 . Nevertheless, there does not appear to be a different concept of numbers behind the different notations. Even notations like 63 and LXIII, although using different number representation systems, m e a n the same, i.e. the same number >>sixty-three<< . In fact, we have a common or unified conceptual basis underlying the Arabic and Roman number systems. We have on earth somehow a single conceptual basis for numbers, whether or not we possess a knowledge of Peano's axioms. We may find in mathematics other things that are conceptually universal, despite different notations, for example multiplication is denoted by × , · or * . It may be annoying for the standards organization that some people use the comma instead of the point to mark the fractional part of a number, but it brings no conceptual complication.

Thus, maybe there is no problem of the universality of concepts in programming, but experience in other fields shows that we should not take this for granted. Different political systems sometimes imply absolutely incommensurable concepts with one and the same word. Democracy as used, for example, in "Democratic party" in the USA and "German Democratic Republic" in Germany, does not mean the same. This may, of course, be a result of the propagandistic use of the word and among "equal" people, the feeling usually prevails that they have a common basis of concepts-all attempts of language translation are tacitly based on this assumption. Whether, on the other hand, from linguistic points of view, certain Polynesian languages have indeed a conceptual basis in common with English, is irrelevant here, since the cultural differences result in a very small overlapping of the abstract concepts.

It suffices to state the thesis that ATHANASIUS KIRCHER and GOTTFRIED WILHELM LEIBNIZ proclaimed: There is independent of

notation and language a common conceptual basis for the events that man considers and creates.

This thesis obviously includes, amongst other creations of man[1], the world of mathematics, and whatever difference existed between the world of GIORDANO BRUNO and of Pope CLEMENS VIII, of ROBESPIERRE and of METTERNICH, of MARX and of BISMARCK - and, for the genius loci, of HAROLD WILSON and HAROLD McMILLAN - it has not produced different mathematices. A strange aberration under the Hitler Regime, called "Deutsche Mathematik", was mathematically nothing more then the lack of good taste.

Thus we may assume that in programming, too, a unique conceptual basis exists. If we would try to defeat this thesis, we could of course mention a number of so-called programming languages with constructions so incommensurable that one would loose all hope to find a common basis. Yet, in so far as these languages have been implemented - some, alas, have not even found a posteriori a definition other than by a particular implementation - we may, for the sake of simplicity, assume that they were created for computers of a certain series of a certain manufacturer. Moreover, we could do it in principle with a Turing Machine. Thus, in terms of these machines, there is a common ground and a unique conceptual basis. Moreover, if somebody would come forward with a carefully defined programming language and the claim that it can not be implemented on a machine of the aforesaid manufacturer, he would make himself look foolish.

Nevertheless, for a number of scientific, educational and even a few economic reasons, we - and you - would not like to make the afore said machine immortal in the sense of basing all our programming conceptually on it. Thus, the question really is not so much

Does a unique conceptual basis exist

but

How do we attain a unified conceptual basis which stands up well to scientific, economic and educational questioning.

If we are not too far from this goal, the advantages are so obvious that to our first question: "Why" a sufficient answer can be expected.

Indeed, existing programming languages show more in common,

---

[1] KRONECKER said: Die ganzen Zahlen hat der liebe Gott geschaffen, alles andere ist Menschenwerk.

sometimes hidden under notation, than one would believe at first glance. We may take "noble" languages like ALGOL 68 or PASCAL, or "vulgar" ones like PL1, add exotic ones like APL or white dwarfs like BCPL, and of course, we may keep ALGOL 60 in view and even look with nostalgia down at FORTRAN. There is, as we shall see, a lot in common, in fact so much that ERSHOV could start a project to build a package of several compilers for several languages with one and the same backing run-time system. The differences remaining after removal of trivial notational and irrelevant syntactic discrepancies are such that they can be explained by arbitrary selection from the available stock.

There are many reasons for doing this, ranging from taste to voluntary austerity. NIKLAUS WIRTH does not like to bother with a variety of types - and has good reasons - so in EULER he is on the side of type-free languages. This does not, however, mean that types do not exist. They exist also for NIKLAUS WIRTH, and in discussions with his colleagues he is apt to use the concept "type".[1] In PASCAL, after all, types are used. ALGOL 60 is intended for numerical problems - thus no handling of names seemed appropriate.

But does the existing variety of programming languages already provide us with a complete conceptual basis for programming? Fifteen years after the Zurich ALGOL conference, where for the first time general discussions within an international group of first-class experts on language features were carried on, one would be tempted to say "yes". But a sceptic like me would say, that it is probably too early. We have followed, to some extent, wrong paths set up by national prejudice as much as by existing machinery. In fact, we shall see below, that certain features of KONRAD ZUSE's Plankalkül, designed in 1945, have not yet shown up in to-days programming languages, and since the Plankalkül was taken notice of only by very few people, this means that, while most of the Plankalkül ideas have been rediscovered - a few came into the open through RUTISHAUSER - some went unrediscovered so far.

At this moment, from scientific, economic and, in particular, educational considerations, we would be extremely interested to explore the full range of fundamental concepts of programming, exhaustively. But it should be clear that, in contrast to the

---

[1] In fact, I have only once found two of my honorable colleagues absolutely unable to communicate over a certain feature. It turned out later that the barrier was a misunderstanding coming from imprecise definition.

above-mentioned, widely used machine of a leading manufacturer, a neat conceptual base should be looked for. In particular, we would not want a mere collection of unrelated special cases. We therefore postulate:

(i) It should be possible to compose new concepts from more basic ones by simple and well-defined composition rules.

These rules will, in many cases, allow a recursive formation of new concepts. It is one of the greater merits of A.VAN WIJNGAARDEN that he presented in ALGOL 68 a rather complete set of concepts, built up recursively - some people, looking only at the syntactic decor, do not recognize this essential role of the upper level of his two-level grammar.

In this formation of new concepts, we have to have something to start with. This will be a smaller or larger set of primitive concepts. We will not be attracted if this set has thousands of elements - just like a mathematician would dislike an algebraic structure which has 500 axioms. We therefore require that

(ii) The primitive concepts, from which to start, should be general and powerful enough to allow their number to be small.

Requirements of this form are not restricted to Computer Science, they would do good to any scientific field. Moreover, there are ample examples in engineering using prefabricated goods. In programming, however, these requirements are probably more vital than anywhere else. Programming, in its productive side, resembles engineering very much. In contrast to classical engineering, however, the raw materials are now cheep, they cost practically nothing. Paper may be used for sketches, but finally, if somebody is able to borrow a magnetic tape or magnetic cards, he may store his program on it, may use it, reproduce it, and may finally return the borrowed program carrier with practically no loss. In this situation, the immense material costs that classical works of engineering involve, disappear. What remains, the labour of setting up the program, converges relatively to zero the more use the program finds. Thus, software engineering allows much larger constructions than classical engineering did and still does. The complexity of these constructions reaches the border line of human comprehension.[1] Thus it is absolutely necessary

[1] A.P.ERSHOV has treated this subject in great detail in a lecture entitled "Aestetics and the human factor in programming [1], at the 1972 Spring Joint Computer conference.

to have sound methods for the formation of sufficiently complex composite concepts that underly the design of large software packages. Thus, in programming, the above mentioned requirements are to be taken very seriously.

In order to be explicit, we shall give a short account of some research work carried out in Munich at the moment, where the investigation of concepts used in scientific literature is being studied in connection with documentation problems. In documentation, if the results of retrieval and selection are to be worthwhile, the conceptual ties that exist between the key words used to denote concepts are most important. Studies of the group, led by Dr. BRAUN, show that only very few rules need to be used to form new concepts from those already available; these rules follow patterns which are universal in all natural languages and allow the treatment of such concepts.

For example, assume that the concepts >>semigroup<<, >>group<< and >>commutative<< already exist, whereby we know that >>group<< is a subconcept of >>semigroup<< and that >>commutative<< is applicable to >>semigroup<<, hence >>commutative semigroup<< exists. Then >>commutative<< is applicable to >>group<< and the new concept >>commutative group<< can be derived. Besides, we infer that >>commutative group<< is a subconcept both of >>group<< and of >>commutative semigroup<<.

Moreover, assume that the concepts >>lattice group<< and >>subgroup<< already exist, whereby we know that >>lattice group<< is a subconcept of >>group<<, and that >>subgroup<< is applicable to >>of a group<<, hence >>subgroup of a group<< exists. Then >>subgroup<< is applicable to >>of a lattice group<< and the new concept >>subgroup of a lattice group<< can be derived. Besides, we infer that it is a subconcept of >>subgroup of a group<<.

If additionally >>normal<<, as applicable to >>subgroup<<, exists, >>normal subgroup<< exists and is a subconcept of >>subgroup<<, thus >>normal subgroup<< is applicable to >>of a group<< and the new concept >>normal subgroup of a group<< can be derived, which we can infer to be a subconcept of >>subgroup of a group<<.

In order to be brief, let it suffice that such rules using attributes and prepositions, together with an adverb - adjective construct (>>partially ordered<<, >>linearly independent<<) are, cum grano salis, the only ones actually used in forming mathematical concepts - and most likely not restricted to this domain.

This result shows a striking analogy to the metanotion formalism of A.VAN WIJNGAARDEN, mentioned above, in the description of ALGOL 68. Apart from trivial enumerations, the construction of types ('modes') and procedures in ALGOL 68 with the help of the particles

long... reference to... row of...
structured with... union of... procedure yielding...
procedure with... parameter yielding...

are of the aforementioned sort.

From this observation, we are encouraged to add a third requirement to our list:

(iii) The rules for the formation of new concepts should be universal. The primitive concepts to start out with should also be universal.

In this way, from a few, universal concepts, by universal rules, new concepts can be formed as desired for any use in any programming language.

It should be noted that this universality is on a meta-level, as compared with the "universal programming languages" that were under consideration in the early days of UNCOL and ALGOL. Universality certainly holds today for languages describing the syntax of programming languages, with essentially BNF productions, possibly shortened by the use of replicators. Universality should also be aimed at on the semantic side as far as the concepts involved in certain constructs are concerned. Freedom of design will then hold in the actual syntax of a programming language and in the corresponding semantic meaning. Needless to say, such a design could always lend itself to natural generalization, a worthwhile tendency which has sometimes been called 'the ALGOL spirit'.

It may be appropriate to give an indication of what the set of primitive concepts might look like. Experience with modern top-down teaching in programming shows that to start with, it should certainly include the concepts of procedure and of procedure parameter, thus trivially embracing operations and their objects, the operands. It should also include the concept of free identification, permitting one to denote something described in some notation, by a freely chosen notation.

Another fundamental concept is that of references, introducing objects which have the meaning of refering to other objects. Cha-

racteristically, these can be presented with the call as parameters of procedures although they only deliver results after the procedure has finished. Although ALGOL 68, PASCAL and BCPL seem to differ greatly in this respect, I would venture the view that they all are based on a common reference concept and that (apart from notation) they differ only in their referencing mechanism, more or less under the depressing influence of existing hardware.

Next comes the introduction of sets of objects, important since many operations may be naturally restricted to operands of certain sets. The domain of a procedure, for example, is expressed by restrictions of its parameters to certain sets. References to a certain set may themselves also form a set. Together with the concept of set there is then also the fundamental operation of testing whether an object belongs to a set.

The family of parameters of a procedure may be looked at as a new object of a new, composite class - we have the concept of structured objects - the slight generalization of C.A.R.HOARE's records. New objects, selectors, that have the meaning of selecting components of a structured object are, however, necessary. Also since the component itself may be structured, their composition has to be defined. As a trivial case, the use of numerical subscripts may be mentioned. Furthermore, one has to take into account how referencing and selecting may cooperate - for example, they commute in some programming languages. I have the definite feeling that these concepts are sufficient to describe ALGOL 68 or rather its revised form; in particular a more explicit introduction of selectors would have given light even to some of the remaining dark corners. The last mentioned concept is in any case just emerging, and one or two more may be required, perhaps in connection with the actual handling of sets characterized by arbitrary predicates.[1] In any case, it is a pretty small number of basic concepts, and these fundamental concepts should be so simple that they can be taught to children. Whether or not this is possible, is the crucial test.

In this sense, we hope that the development of programming languages, which at the moment is a playground, will become more rational and, in the end, strictly functional. Remembering frustrating IFIP WG 2.1 working sessions, where very intelligent people discussed misunderstandings on the basis of partial ignorance, a miraculous possibility opens here: Future programming

[1] R.GNATZ [2]

languages, or programming languages of the future, can be the result of rational discussion among people who have - irrespective of whether they are a manufacturers employee, a research laboratory worker or a teacher - a common basis for their reasoning, which puts them all on an equal footing: A thesaurus of common concepts and the ability to use them rationally.

Second lecture:

The role of structuring in programming

While the previous lecture discussed some aspects of design principles for programming languages, this one concentrates on their rational use. It should be kept in mind, however, that a powerful programming language allows a great variety of style in the way it is used. Some user can go to the extreme in breaking down procedures into subprocedures, which in turn are broken down into subprocedures and so on; another user may hardly use procedures at all. Or, relatedly, one user may break down any expression into three-address-like assignments to explicitly introduced auxiliary variables, another one does not do this. More seriously, one user may use jumps without any hesitation, while another one will carefully try to avoid them. In fact each one of these users, by writing in his own style, defines for himself his particular programming language as a subset of the given one. He may even use several of these, according to varying circumstances – different ones for numerical work and for symbol handling work, different ones for programs which will run only a few times and for programs which will chew time - or different ones for the morning and for the afternoon, according to the amount of concentration available. An experienced programmer will subconsciously do so.

There can be no question that the idea of a "universal programming language" can only mean a language that permits the experienced programmer full freedom in the choice of his style. So far, existing attempts at universality could not give the programmer this choice. One and only one language from the point of view of definition (and thus, from the point of view of the compiler) yet an infinity of languages, with varying style, to please the programmer, that is the goal - probably unreachable - that should be aimed at. Since experience has shown so far, that the future demands of programmers can hardly be predicted - we say in Bavaria "The appetite grows while we eat" - the solution against the background of a common conceptual basis, can only be expected in the direction that was aimed at with extensible languages, a field that, to my opinion, has not yet prospered well enough. We shall in this lecture not go into technicalities, but we will shortly return to this problem.

For the moment, we will concentrate on the question: Although some programmer may want to make extensive use of jumps, it could happen that I do not want to let him do so. In other words: when a language, that is a natural language, permits the use of alternative expressions in the sense of different styles,

we cannot prevent somebody from using a bad style, or even from using an abusive language. We may not want a programmer to use a certain style, if such a style leads to a program, the correctness of which is hard to verify. We may dislike styles which stimulate people to take risks - risks that lead to mistakes. We prefer styles that permit us to read and understand, without solving riddles, someone elses programs. The use of jumps is an excellent example of how to transgress all these sentiments, and it is for this reason that jumps are considered harmful by some people.

One could try to restrict programming languages so that everybody would be forced to program in a self-defensive and self-explicable way, to mention only the two virtues discussed above, but which could possibly be supplemented by a few others. At the present stage of our knowledge, this does not seem to be possible. Moreover, it may not even be wise to want such restrictions, since they are very likely to have technical sideeffects for which one would pay more than the gain is worth. Restrictions have also the further disadvantage that they are, to some degree, a matter of taste.

It seems to be better, to educate programmers to make non-objectionable use of a programming language, and even better, to teach them how to make the best use of a programming language, as a function of the situation.

Alarmed by signs of system-inherent inefficiency in the programming efforts of manufacturers (the 'software crisis'), which was quite contrary to his own experience with a middle-sized system programming effort (the T.H.E. operating system), E.W.DIJKSTRA studied the programming habits of our generation and found, that in the long course of development from programming the von Neumann machine, something was lost which one would consider a very natural technique: the structured design of a program. Clearly, a machine program for the von Neumann machine shows nothing whatsoever of the structure of the original problem. But the use of flow-diagrams to support such programs also does not usually bring back the original structure. On the contrary, it may obscure it even more. It stands to reason that our knowledge of the ultimate goal, machine programming, has influenced very much the way we use the so-called high level languages, like ALGOL 60. It has made us not only forget well-structured design even at that level, but also to propagate this to our pupils. I cannot explain in any other way the Saul-Paul effect that I have observed many times when someone started thinking about 'structured programming' or, as an American colleague recently said, "you read the whole literature with a different eye after you have read DIJKSTRA".

"Structured programming"[3] - other catchwords used are "stepwise refinement"[4], "iterative multi-level modelling"[5], "system hierarchy"[6], "stepwise program composition"[7], "programming by action clusters"[8], "top-down program development"[9] - is a technique which allows verification of the correctness of all steps in the design process and, thus, automatically leads to a *self-explicable* and *self-defensive* programming style.

The result of structured programming is not just one program, but a whole sequence of programs such that any one can immediately be deduced from its predecessor.[1] Structured programming in this strict sense is very paper-consuming. In practice, one does not always rewrite the whole program, it frequently suffices to follow a path of stepwise detailization sometimes here, sometimes there. Clearly, pencil and paper are not very ideal tools for structured programming, and the systematic use of an eraser would destroy earlier versions and thus exclude somebody else not only from verifying correctness, but even from understanding the program fully. Sometimes, rudiments of earlier stages of the development are carried in the so-called *comments* allowed in most programming languages. Without question, structured programming can be executed very efficiently with the help of a file keeping system and a display. The whole design process can then be recorded and, if necessary, be played back. A very promising attempt has been made by BRIAN RANDELL with the PEARL system. After sufficient experience, efficient solutions for the problem of presenting this full material, without loss of clarity, also in printed form, will be developed.

In order to demonstrate that structured programming is not restricted to the level of certain so-called high-level languages, we give a very simple example at a relatively low level: Consider the problem to program the assignment

$$a := (b + c \times d)/(c + d)$$

in three-address-like assignments close to machine language.

---

[1] An aside is appropriate: The order relation which is the starting point for DANA SCOTTs theory, is just the relation >>more detailed than<< which is relevant here.

We have

(I) $a := (b + c \times d)/(c + d)$

and from this[1)]

(II) $h_1 := c \times d$;
$a := (b + h_1)/(c + d)$

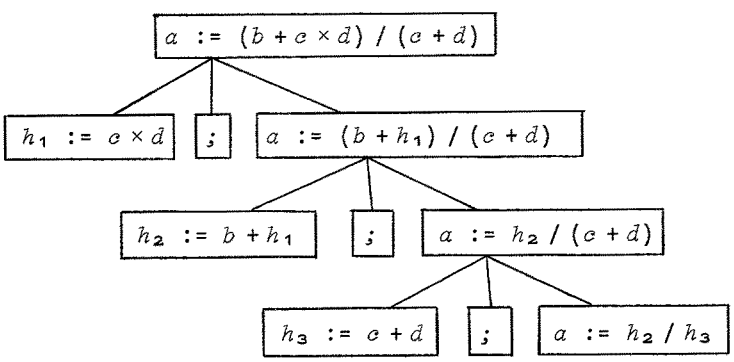
Then

(III) $h_1 := c \times d$;
$h_2 := b + h_1$;
$a := h_2/(c + d)$

and finally

(IV) $h_1 := c \times d$;
$h_2 := b + h_1$;
$h_3 := c + d$;
$a := h_2/h_3$.

Anyone will be able to verify the correctness of the design. Of course, since we have always modified only the last line, a shorter writeup would reflect the tree-like structure of our development.



1) We assume here that the auxiliary variables $h_1, h_2, \ldots$ are declared elsewhere.

I could also imagine a problem in which something may be specified about the type of some of the parameters in the formula, for example b being real, c and d being integer. Then a would be real in general, and we would derive in steps, which we omit now[1].

$$\begin{aligned} \underline{int}\ k_1 &:= c \times d\ ; \\ \underline{real}\ l_1 &:= b + k_1; \\ \underline{int}\ k_2 &:= c + d\ ; \\ \underline{real}\ a &:= l_1/k_2 \end{aligned}$$

The type-correctness may be verified immediately.

It goes without saying that in a lecture no example that extends over more than a few lines can and should be given. I will comply with this fact which excludes problems starting at a verbal level and assume that you will agree in the end that for large design jobs, it will be even more important to apply these techniques.

But for small examples also, an attempt to prove correctness can give deeper insight.

To give a beautiful example: Today, everybody knows the Euclidean algorithm for the determination of the greatest common divisor of two (natural) numbers; this is, what we may use to start from in our example:

"Given a,b , determine the remainder of the division of a by b . Repeat this process with b and the remainder".

This rather imprecise formulation probably means, that a be replaced by b , b by the remainder of the division of a by b , and that with the new pair (a,b) the process be repeated. Nothing is said about stopping, but since we can not divide by zero, we have to stop in any case when the remainder becomes zero. Hurray! then the divisor, the (old) b - that is the new a - is the greatest common divisor - we forgot to mention this above.

Note: From an incorrectly stated problem, the correct program can be obtained by guessing, possibly also by meditation or by witchcraft, but certainly not by verifyable modification. Structured programming is of no help when mistakes are inherent.

[1] We assume here that the auxiliary variables are declared (of the specified type) and initialized as given.

After this diversion, we start now with a more formal presentation. Strictly speaking, we have recursively

```
(*)  proc gcd = (int a , int b): if b ≠ 0 then gcd(b,a mod b)
                                 else result := a   fi
```

This can be modified to

```
proc gcd = (ref int a , ref int b) :
                  if b ≠ 0 then (a,b) := (b, a mod b);
                                gcd(a,b)
                       else result := a            fi
```

and now the parameters can be freed:

```
proc gcd = (ref int a, ref int b) :
                  ⌈while b ≠ 0 do (a,b) := (b, a mod b)  od;
                   result := a
```

Omitting now, for brevity, the parameter part, we obtain

```
(**)  while b ≠ 0 do (a,b) := (b, a mod b) od;
       a
```

in which the final a (to which the last non-zero b has been assigned) yields the desired result.

The simultaneous assignment of the (old) value of b and the value of the remainder of (the old) a by (the old) b to a and b respectively, occurs between the *do* and the *od*. A pair of values is formed, and this pair is assigned to the pair of variables. Can this be further refined? Can it be replaced by

```
a := b;  b := a mod b
```

But why should the two single assignments be done in this order? Should not

```
b := a mod b;  a := b
```

do it as well? The answer is that neither of the two versions is a correct modification, in fact the first piece uses the new a instead of the old a in the second assignment, while the second piece uses the new b instead of the old one. Not only can a correct modification be checked, an incorrect one will be discovered in many cases quite easily. In fact, any single modification

should be such that any failure to verify its correctness with a given repertoire of rules implies that the modification is incorrect.

The repertoire of rules just mentioned must of course be such that it could be applied mechanically, i.e. that a program could be or is written to apply it. Then the above restriction of the step size means that no heuristic means have to be applied in order to prove or disprove the correctness of a modification - a requirement that seems to be necessary in view of the broken promises of artificial intelligence.

Another step in the refinement process could be, to replace the remainder operation by a procedure using only addition. In fact, everybody knows now that "the remainder of the division of a by b is obtained, if b is repeatedly subtracted from a until, for the first time, something less than b is left."

Warned by the previous example, we may write more formally

$r := a;$
while $r \geq b$ *do* $r := r-b$ *od;*
r

We can now replace a *mod* b by this piece, obtaining the version

$(***)$ *while* $b \neq 0$ *do* $(a,b) := (b,$ [$r := a;$ *while* $r \geq b$ *do* $r := r-b$ *od;* r]
a

Next, we go a step further and make use of the variable r that somebody has sent us from heaven (or which otherwise has to be declared internally) and obtain the version:

$(****)$ *while* $b \neq 0$ *do* [$r := a;$ [$a := b,$ *while* $r \geq b$ *do* $r := r-b$ *od;*] $b := r$] *od;*
a

I must leave it to you to discover the formal rules which have permitted this (correct!) modification.

At this moment should there be someone among you who now has doubts about everything, including whether the euclidean algorithm actually gives the greatest common divisor. To him I give the problem again in the genuine form "Find the greatest common divisor of a and b"[1] and ask him, to agree to let $f(a,b)$ denote the greatest common divisor of a and b. Next, I ask him to confirm that this operation is symmetric, that is:

$$f(a,b) = f(b,a),$$

also that for $a > b$:

$$f(a-b,b) = f(a,b)$$

and finally to concede that:

$$f(b,b) = b.$$

Thus, a seemingly more direct way of calculation would be

```
(†)  (r,s) := (a,b);
     while r ≠ s do  ┌─────────────────────┐
                     │ if r > s then r := r-s │
                     │         else s := s-r fi │ od;
                     └─────────────────────┘
     r
```

That not only looks somewhat similar to the previous solution, it turns out to perform exactly the same sequence of operations if fed with the same integers. But intuitive assurance is not what is asked for. Again, I must leave it to you to establish step by step the modification that leads from the one version to the other. HINT: The above piece can first be modified to

```
(††) (r,s) := (a,b);
     while r ≠ s do ┌──────────────────────┐
                    │ while r > s do r := r-s od; │
                    │ while s > r do s := s-r od │ od;
                    └──────────────────────┘
     r
```

It should have become clear by now that structured programming actually means well-structured programming. In fact, according to DIJKSTRA "A major function of the structuring of the program is

[1] The example is taken from E.W.DIJKSTRA [7]

to keep a correctness proof feasible".

Note: The art in programming is to find the right structure.

No doubt, this requires intuition. But intuition is not enough: A program that counts once too often (to whom has this not happened in the days of machine programming) is wrong. Nothwithstanding the fact that it is nearly right, it wastes as much time as if it had counted wrong by two or twelve. Structured programming exposes the intuition to the crucial test of correctness proof. "How to solve it" now becomes "How to prove it". Computer-aided programming? We shall see that in its last consequences, structured programming leads exactly to the use of the computer as a correctness-prover, and that this is a natural continuation of the role that is played by compilers today. Thus, at the very end, problems will really be solved with the help of computers, and not, as is now, the computer only following the programs that are expected, hoped for, and believed to give solutions of a given problem, or rather programs that are suspected, known or even proven not to give solutions of the given problem.

Let us first investigate the compiler-related characteristics of structured programming. If we return for a moment to the first example we have given: It is conceivable that we would have given that expression to a compiler and obtained the final sequence of three-address-form assignments from it. Moreover, we would be quite convinced that the compiler would have done the intermediate steps in the same way as we did. In fact, the compiler would have selected - among several possibilities - just the one we have chosen, being programmed to do so. Even more, a program of a size quite comparable to the compiler would be able to verify modification of the form we have made.

On the other hand, a compiler that could pick up the version (†) above and produce the version (††) does not exist today. A compiler that could possibly produce the version (**) from version (††) is out of the reach of our present mastery of the technicalities. Thus, while present compilers do the modification down to some machine-close level, in an absolute dictatorial way, the compiler for computer-aided, intuition-controlled programming would investigate any verification step it is offered, would reject the new version if it could not be proven to be correct, would even go down into the domain of what compilers accomplish now if guided, but would finally try its own straightforward way close to machine level as soon as the green light were given. With this view high-level and low-level programming become somewhat less distinguishable, certainly for the satisfaction of

those to whom the two words express the difference between nobility and proletariat.

The mechanical validity check for certain modifications requires a formalized description of the semantics of essential programming constructs. Thus it was only consistent that DIJKSTRA also entered this domain [10]. Starting with McCARTHY in 1962, followed by FLOYD and HOARE, one has now reached a state that, as far as some of the most important constructs (including alternatives and repetitions) are involved, mechanical validity checks embracing the handling of pre- and postconditions, i.e. boolean expressions, can be envisaged. From the list of indispensable primitive concepts, discussed in the first lecture, compared with DIJKSTRA's paper we see that there is still something lacking, mainly in connection with structured objects and selectors. It will certainly be a long time before structured programming, fully supported by mechanical validity checks, will be available.

Structured programming - we have used the word in the widest sense - involves a spectrum of possibilities of formulation that is wider than any existing programming language provides for, ranging from very high language levels, in particular those of a verbal type, to machine-close levels. Since structured programming should permit any mixture of levels in order to be absolutely free in intuitive guidance, one would need wide-spectrum programming languages. It is conceivable that ALGOL 68 in its revised form, or rather a subset of it that has expelled some of the remaining monstrosities, can be augmented in such a way that its spectrum can be extended in the downward direction, permitting in particular more explicit name-manipulating operations that exhibit, if required, the address-character of names, in conjunction with the explicit introduction of selectors (instead of the "descriptors" used in ALGOL 68 semantics) that also permit the formulation of structured names with particular applications to stacks, tables and queues. But this would not be enough. In the upward direction, one would at least need conditions, (i.e. boolean expressions) and also quantifiers like "for all..." and "this one, which..." as genuine, manipulatable objects[1] - constructs that KONRAD ZUSE already had in his Plankalkül of 1945, constructs which are essential for the aboriginal formulation of problems.

While further accomplishments in this direction should be awaited, another approach can lead more quickly to some results.

[1] R.GNATZ [2]

Imagining a set of, say, a dozen programming languages layered so that any construction in any language can directly be translated into the language of the next lower level, and with compilers doing this transformation at hand - the uppermost level could be something like ALGOL 68, the lowest a certain assembly language, structured programming could go through these layers. The constraint of being forced to use these layers would be rewarded by disentangling the whole problem into eleven simpler problems. Each compiler would be replaced by a validity checker that would test the modification offered to it, would reject it if incorrect, accept it if correct and, while truly following all these recommended steps, doing the rest of the compilation to the best of its ability. This validity-checker would degenerate into a genuine compiler, if nothing is offered to it and would no longer be a compiler, if all possible modifications were described.

The different layers of languages could also be considered in their totality as a single wide-spectrum language. In particular, if they are subset-wise connected, then the uppermost layer defines the spectrum. Practicality may, however, be held against the complete subset relation.

Such an approach could be of particular interest in connection with the design of very large software systems, which requires the use of particular tools of software engineering [11].

Finally, it should be remarked that not all programming languages are equally suitable for structured programming. Some, like APL, are positively hostile to structured programming. The fact that this particular language is believed to be enormously useful in connection with "interactive programming" for example demonstrates that the form of "interactive programming" in question is quite contrary to structured programming. It is to be hoped that this misuse of terminals, and other wonder lamps that a leading manufacturer has set up for the misguidance of its customers, will die out and that mere text-editing will be replaced by sound computer assistance to programming. Languages like BASIC are likewise of no use in structured programming. The fact that certain economic situations favor the introduction of BASIC in high-schools by short-sighted teachers is deplorable. Countermeasures must be taken which convince the people in question that structured programming is a better way to follow. And whatever defects they may have, languages like PASCAL, BCPL, BLISS, proposals like GEDANKEN, modifications like PL/C, as well as good old ALGOL 60 and even brand new ALGOL 68 Revised are the result of serious attempts to provide adequate tools for the programmer's immense job of mastering complexity. The best programming language, however, can usually not stop people from misusing it, from using

it badly or just from inadequate use. Education is the only thing that can help. The programming language, however, should be a guide to good style. To quote DIJKSTRA again, in connection with certain products which are supported by a leading manufacturer "one of my implicit morals will be that such programming languages, each in their own way, are vehicles inadequate to guide our thoughts. If FORTRAN has been called an infantile disorder, PL/1 must be classified as a fatale disease".
Needless to add that programming habits can not be better than the languages used for programming. It is certainly possible to write in FORTRAN clean style. But it is rarely found.

We can not close this lecture without mentioning a deep connection between structured programming and the modern top-down way of teaching programming. This teaching method starts with the most general concepts and adds, step by step, new ones that allow more detailed formulation until one arrives at formulations close to machine level. Teaching progresses in the same direction that later is used in actual structured programming (*top-down-programming*), thus strongly motivating this technique. On the contrary, the classical bottom-up teaching of programming, starting with a machine ('our universal model computer') and sometimes only climbing half-way up to the hills of ALGOL 60, is in essential contradiction to a structured programming approach. It is to be reasoned that hitherto the predominance of this teaching technique is partly responsible for the bad programming habits of the generation which went through the 'software crisis'.

Third lecture:

System uniformity of software and hardware[1)]

It could be argued that the examples of structured programming we have given in the second lecture are insignificant because of their restricted size. To quote DIJKSTRA once more, "the examples will be 'small' programs, while the need for a discipline becomes really vital in the case of 'large' programs. Dealing with small examples in an ad-hoc fashion gives the student not the slightest clue as to how to keep the construction of a large program under his intellectual control. In illustrating how we can avoid uncontrolled complexity, I hope to deal with small examples in such a fashion, that methodological extrapolation to larger tasks is feasible. The very last thing we can do is to show the reader how to organize his thoughts when composing, say, page-size programs. If in the following the reader thinks that I am too careful, he should bear the chapter-size programs in mind. (If he is still unconvinced he should study a single page program made by a messy programmer: He will then discover that even a single page has room enough for a disgusting and intellectually unhealthy amount of unmastered complexity!)".

Unmastered complexity, however, is not the prerogative of software. Hardware also suffers from it from time to time. Looking at some of the more remarkable hardware design defects that have come to general knowledge, we find parallels with software. Nowadays, when unmastered complexity has become the nightmare of software engineers, methods developed to escape from this danger can also be applied to hardware design.

The design of existing hardware is inappropriate anyhow. Storage organization carries the load of earlier wrong decisions, which can not be revolved so easily, due to compatibility reasons. Another area of insufficient hardware organization is concerned with programmable priority and interrupt devices. The problems involved have come into the open with attempts to design large operating systems. It was only then that abstract studies of sufficient depth were undertaken. A number of software based approaches have developed: semaphores are used for synchronization by mutual exclusion (*P*- and *V*-operations, DIJKSTRA 1965) and for event synchronization; BRINCH-HANSEN (1970) uses message primitives, DAHL (1970) uses coroutines, WULF (1969) has process

---

1) This lecture theme was stimulated by a paper given by R.WIEHLE [12].

creation and termination operations. So far, the development has, however, not reached the hardware side.

In order to illustrate the problem we take a very simple situation: two pieces of program, one denoted by $\mathcal{A}$, one by $\mathcal{B}$, are collateral, in plain words "side by side", i.e. neither has any effect on the other.[1] We then write $\mathcal{A},\mathcal{B}$, as in

$$a := 1,\ b := 2$$

$\mathcal{A}$ and $\mathcal{B}$ may both use a common procedure, as in

$$a := gcd(3,5),\ b := gcd(74,122)$$

two distinguishable, independent incarnations of the procedure are meant. But both $\mathcal{A}$ and $\mathcal{B}$ may not alter the same variable, since this may mean a mutual effect. This is clear from the example

$$(a := gcd(3,5);\ c := c+1),\ (b := gcd(74,122);\ c := c\times 2).$$

Similarly the example

$$(a := gcd(3,5);\ c := c+1),\ (b := gcd(74,122);\ c := c+1)$$

will also be excluded, the two pieces not being truly collateral, although no harm is done: The two additions of 1 commute.

Should this latter construct, therefore, be allowed? No, since the two assignments can still come into conflict, if they are tried simultaneously - or, more realistically, within a tiny interval of time - a conflict which would not always be resolved by the hardware. Thus, we could save the situation in the second example if we prevented a clash of assignments. In the early days of railroads, when most lines were single track, a clever device was introduced: Whenever a train entered a track, a semaphore went up and signaled to another train - arriving usually from the other side - that it had to wait until the first train had left the section and correspondingly the semaphore was lowered. Thus, we introduce semaphores, i.e. binary variables, variables with two states, 1 and 0, or *up* and *down*, which may be set to either state, and with the (tacit) assumption that the same semaphore may be set from each of the collateral

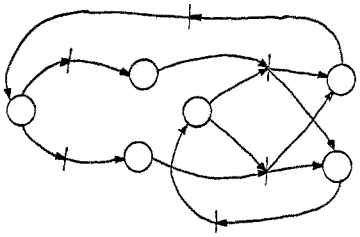
[1] This is a clarification of the definition in the ALGOL 68 (and ALGOL 68 Revised) Report.

pieces. If such a semaphore may also be tested to determine whether its state is 1 or 0 , and if corresponding conditions may be formulated, then the railroad semaphore example leads up to the collateral construct

```
(a:=gcd(3,5); label1: if sema=1 then [sema:=0; c:=c+1; sema:=1]
                      else goto label1      fi          ),
(b:=gcd(74,122); label2: if sema=1 then [sema:=0; c:=c+1; sema:=1]
                      else goto label2      fi          )
```

where *sema* denotes a semaphore. Unfortunately, the example is still wrong, unless special assumptions, not inherent in nature of a conditional statement, are made. Besides, alas, the example contains a *goto* . DIJKSTRA shows how to replace this clumsy (and incorrect) construction for mutual exclusion by a more elegant one, using two synchronizing primitives, the operations *P* and *V* mentioned above, at the beginning and at the end of the critical section.

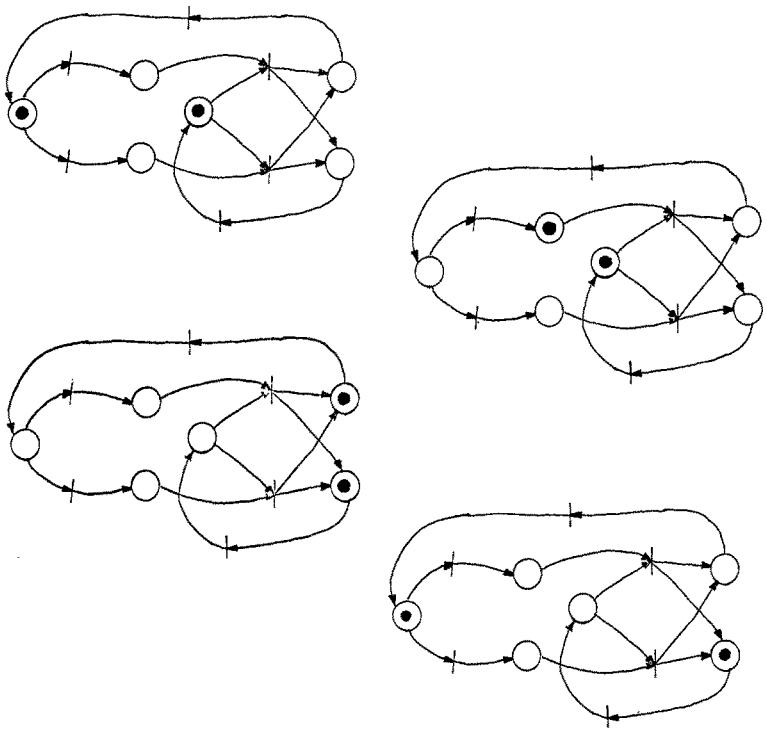
Interestingly enough, C.A.PETRI published a very fundamental scheme of asynchronous cooperation in 1961, which has been rediscovered after several years. In its general form we have a directed graph with nodes of two types, those called places and marked by circles, others called hurdles and marked by bars. The following may illustrate it:[1)]



1) From J.DENNIS [13].

And now a solitaire game is played: Each place is supplied with some number (possibly zero) of tokens. A hurdle is called "enabled", if all places from which an arc leads to the hurdle, are supplied with at least one token. One enabled hurdle may be chosen to fire. This amounts to removing one token from each of the places that have an arc to the hurdle and to adding one token to each of the places to which an arc leads from the hurdle.

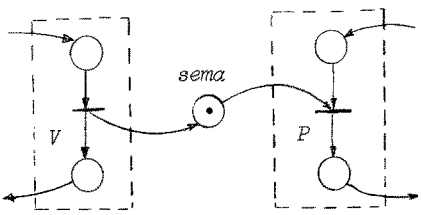
The following shows a sequence of situations in the Petri net above.



State transition diagrams ('flow diagrams') can easily be expressed by a special class of Petri nets, where only one token is always present (the location of the token corresponds to the number in the program counter of a conventional machine). Petri nets of this class belong to programs in the classical sense of

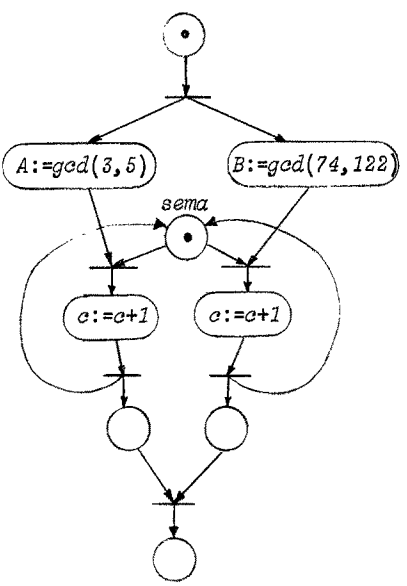
the word. More generally, we speak of processes if the corresponding Petri net is not necessarily a state transition diagram.

In particular, program synchronization may be treated with the help of Petri nets. In the following scheme, the synchronizing primitives P and V, mentioned above, are the subgraphs encircled at the left and the right, the place in the center plays the role of the semaphore,



P : blocks
V : releases

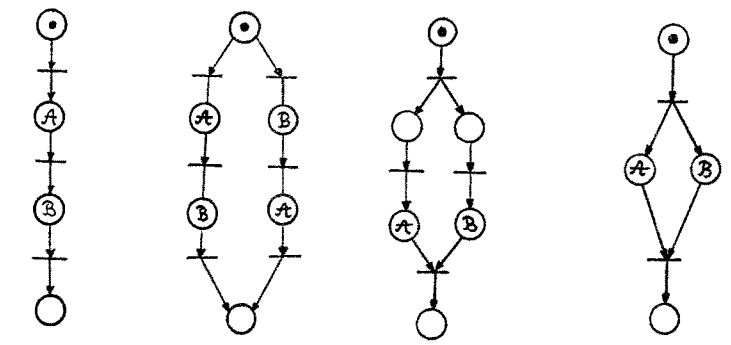
and our attempts above to master a concurrent situation are successfully expressable with the following Petri net.



Although Petri nets are much too primitive to be suitable for practical work – they resemble in this respect Turing machines –, they are an excellent basis to which some other, if not all of the synchronizing devices mentioned above, can be reduced. In particular, they lend themselves easily to the generalization of programs to processes, where work on several programs is done concurrently, when more than one stream of actions exist. They can thus be used to express even different processes that are allowed in a collateral construction. The construction

$$\mathcal{A}, \mathcal{B}$$

allows essentially four different Petri nets



serially sequential | pseudo-collaterally sequential | genuine collateral | parallel

These are called the serially sequential realization, the pseudo-collaterally sequential realization, the genuine collateral realization and the parallel realization. The two sequential realizations are programs, they are based on state transition diagrams. The parallel realization is actually a process and no longer a program, the same is true for the genuine collateral realization.

Having thus explained the fundamental role, Petri nets play in questions of synchronization, in particular for processes, as exemplified in pure software terms, it should be mentioned that C.A.PETRI devised them originally in connection with hardware

considerations. Computer engineers will immediately see their relevance for the complicated architecture of asynchronous units, in particular in connection with I/O devices and in real time application. Petri nets, however, are absolutely neutral as to whether the world they describe is soft or hard, they are on such a level of abstraction such that the difference between software and hardware disappears. The fact that this is possible for 'flow' structures, is a first example of what we may call the system uniformity of software and hardware.
Is there more of this sort? Indeed, if at elementary school a problem in arithmetic is given in the form

$$3 \mid \times 6 \mid -2 \mid \div 8 \mid +5$$

Does software or hardware underly this? The child on being asked this question, would not know how to answer, for lack of terminology, of course. But if we were to try to answer instead, should we say: 'soft' if the calculation is done mentally, 'hard' if a desk calculator is used. There is a difference, but is this difference of any importance, or is it just accidental? And if we see

$$(33 \times (5 + 2) + 8) \times 17$$

is an association with 'soft' more justified than one with 'hard'? It depends on what machine you have been brought up with. If you have been familiar with the machine of a smaller American manufacturer, or if you have recently discovered minicomputers with algebraic input, and if you have not been indoctrinated by some other sources, you may have the 'hard' feeling in connection with any arithmetic expression. What about a full program in one of the more respectable programming languages? People who have written an interpreter for such a language may feel very 'hard' about the constructions of this language, but people who have written a compiler for it should have no reason to feel 'softer'. In general people who have learned programming in top-down teaching and who are used to do structured programming, should at any moment know what the construct they have written down means operationally.

Thus, truly operational programming languages, if taught the right way, should not leave any room for the foolish distinction between hard and soft.

After all, microprogramming is becoming more and more interesting. Take the problem of forming the n-th power of a number, say a real number. There is the trivial way (for non-negative exponents)

```
(real a, int b) real:
     ⌈real x := a, int y := b, real z := 1;
      while y > 0 do y := y - 1, z := z × x od;
      z                                        ⌋,
```

but there is also the more subtle way, where

```
y := y - 1, z := z × x
```

is replaced by

```
⌈if odd y then y := y - 1, z := z × x fi
 y := y / 2, x := x × x                 ⌋.
```

The number of steps this segment has to be performed is, for not too small n, considerably decreased. Is it more economical to use the variant? This depends on the hardware. We may safely assume that squaring takes about the same time as multiplication. But how does halving compare with counting down by 1, and how expensive is the test, whether y is odd? Even at a rather high level of programming, information is needed about hardware functions and, moreover, about the compiler. (I am sure that certain compilers do halving by floating point division, using a floating point representation of 2, and I would not be surprised if this representation would even be incorrect in the last bit. I don't dare to think how odd y will be accomplished - on a certain minicomputer which I found otherwise quite agreeable, there is no other way of doing it than by 2 × entier(y / 2) ≠ y.)

Clearly, if y exists in binary representation, then odd y will be done by inspecting the last bit, and halving by a right shift. If y is not in binary representation, then its binary representation will be formed anyhow during the process.

If this is not enough to underline the fact that software and hardware can not be looked at separately, then we will modify the problem slightly in order to demonstrate that software and hardware can even be indistinguishable. We observe that the method does not only work for repeated multiplication, but for every repeated associative dyadic operation, provided z is initialized by the neutral element

of this operation. In particular, we may replace multiplication of reals by addition of integers and have the algorithm

```
(int a, int b) int:
  ⌈int x := a, int y := b, int z := 0;
   while y > 0 do ⌈if odd y then y := y - 1, z := z + x fi;
                  y := y / 2, x := x + x                  ⌋ od;
   z                                                        ⌋
```

for the multiplication of a by b . Apart from the fact that it is an algorithm with a remarkable history [1], we may now discover that it is the multiplication algorithm for binary number representation of x also: With AC (for accumulator) instead of z , lastbit y ≠ 0 for odd y , rightshift y for y := y / 2 and leftshift x for x := x + x, we have the classical multiplication control

```
AC := 0;
while y > 0 do ⌈if lastbit y ≠ 0 then AC := AC + x fi;
               rightshift y, leftshift x              ⌋ od .
```

(Note that y := y - 1 can be dispensed with, if rightshift drops the last bit.)

Thus, we have "high level" programming and the multiplication control in an arithmetic unit in peaceful coexistence. The conclusion is that microprogramming can be and should be a smooth continuation of "high level" programming.

This is even more so, if we start at a still higher level:

```
proc mult = (int a, int b) int:
            if b > 0 then a + mult(a,b - 1)
                     else 0                fi
```

---

[1] The algorithm is said to go back to ancient Egypt, it was much in use in medieval times. It is illustrated by the example

25	43
12	~~86~~
6	~~172~~
3	344
1	688
	1075

Numbers at the right are crossed out if numbers on the left are even. The remaining numbers are added up.

is (for non-negative integers) the natural definition of multiplication. It can be operationally shortened to

```
proc mult1 = (int a, int b) int:
             if b > 0 then if odd b then a + mult1(a,b-1)
                                    else mult1(a+a,b/2)  fi
                      else 0                                fi
```

Embedding multiplication in accumulating multiplication, we obtain

```
proc mult1 = (int a, int b) int:
             accmult(a,b,0)

proc accmult = (int x, int y, int z) int:
               if y > 0 then if odd y then accmult(x,y-1, z+x)
                                      else accmult(x+x,y/2,z) fi
                        else z                                  fi
```

which coincides with the repetitive algorithm above. A recursive definition of the multiplication unit of a computer? Yes, and in this general description it is not even of any bearing whether multiplication is done serially (recursive in the time scale) or in parallel (recursive in space, i.e. in circuitry) by additions.

This system uniformity in arithmetic can be more dramatically demonstrated by addition. Addition of non-negative integers is defined in terms of the successor and predecessor functions by

```
proc add = (int a, int b) int:
           if b > 0 then succ(add(a,pred(b)))
                    else a                    fi
```

This can be shortened operationally to

```
proc add1 = (int a, int b) int:
   if b > 0 then
             if odd b then
                      if odd a then 2 × succ(add1((a-1)/2,(b-1)/2))
                               else succ(2 × add1(a/2,(b-1)/2))   fi
                      else
                      if odd a then succ(2 × add1((a-1)/2,b/2))
                               else 2 × add1(a/2,b/2)                fi
                                                                      fi
             else a                                                     fi
```

This is equivalent to

```
proc add2 = (int a, int b) int:
   if b > 0 then
            if odd b ∧ odd a then 2 × succ(add2((a-1)/2,(b-1)/2))
          elsf odd b∧even a then succ(2 × add2(a/2,(b-1)/2))
          elsf even b∧odd a then succ(2 × add2((a-1)/2,b/2))
                            else 2 × add2(a/2,b/2)              fi
            else a                                              fi
```

which shows clearly the "logical" functions to be performed, if additon is done in a radix 2 system and again leaves open whether the recursion is done in time scale or in circuitry.

The last example has also shown, that (binary) case cascades and "logical" functions have a common root and that simplification of case cascades and simplification of logical expressions go in parallel. Euler knew that, and so did Venn (1880), but it has been rediscovered frequently by laymen, for example in circuit theory or recently for "decision tables". System uniformity is a point of view, that abolishes this duplication in efforts.

We could continue with examples:
Recognition algorithms can be used to define automata and formal languages and at the same instance to define sequence-controlled recognition circuits. The natural rowwise extension of operations to rows of operands is an instrument of language extension and also a famous hardware trick for gaining operational speed.

We know that the border line between software and hardware depends on matters of taste as well as on the rapidly varying production costs. At a moment, when for the first time in 20 years smaller computers have a chance to attain a better price/performance ratio than the big ones, a shift of the border line can be envisaged.

Does that change our programming problems, our programming tools, or rather should it? Insofar as it would also enable us to do away with outdated storage organization, we could hope for a change.

Bad hardware design replaced by similarly bad software design is no good, but it can be 'repaired' - programs are this alterable. However, replacing bad software design by bad hardware design, as the trend would go, is deadly; it is a nightmare to everybody hoping for progress.

Thus, to be prepared mentally for coming changes, it is necessary to see system uniformity everywhere. This is again an educational problem, a problem which is not yet as much worked out as would seem to be necessary.

The programming style of tomorrow should be independent of where the border line between software and hardware is situated, it should not refer at all to that distinction. Hardware and software should be considered as different technological realizations of programming concepts and nothing else.

Final:

Our responsibility

This series of lectures is necessarily incomplete. It was impossible to go into some technical details, but my references to the literature may show the way. It was impossible to give equal weight to all relevant aspects: The title "A philosophy of programming" already expresses that I am aware of some subjective bias. Nevertheless, I would hope for consensus on a few points. First there is our relation to complexity: As much as I believe that programming deals today with the most complex artifacts of mankind, I am convinced that it has a very simple conceptual basis - much simpler, by the way, than mathematics - but what can be built out of these simple elements easily outgrows the mental capability of an individual. We have to learn to master the complexity which we are allowed to tackle: A unique conceptual basis, well-structured design and the knowledge of the system uniformity of hardware and software should help us in overcoming the difficulties.

PERLIS said once: "Our problems arise from demands, appetites and our exuberant optimism. They are magnified by the unevenly trained personnel with whom we work."

Here is our point: Education. Little can be done about people who have entered the computing profession some time ago with uneven training. But we have a responsibility for the future.

There is the manufacturer's responsibility. His educational approach is understandably, but unfortunately, directed on a short range basis towards increased productivity, reliability and loyalty of his employee. In full accordance with this goal, the manufacturer has a tendency to keep the employee dependent. This may include a certain reserve in allowing the employee too much intellectual activity. All this is not healthy, and will in the long run work against the manufacturer who acts this way. We do not have to discuss it further.

There is the responsibility of the educator which is not restricted to the level of Ph.D.candidates. Programming concepts are simple enough to be taught to children, if this is done with proper regard to the childs more playing attitude and no complicated problems involving great endurance are given. Already at the age of fifteen, some ambition to solve 'attractive' problems can be aroused. My own attempts in this direction have led to a booklet "Andrei and the monster"[14], an English translation of which is in preparation.

There is the responsibility of the programming guild: This we should take more seriously and here we should include the many people doing practical work, aside from scientific workers and educators. The programming guild of the future should not only have sound knowledge of the state of the art when obtaining the diploma, it should have sufficient formal training that it can cope with further development. In particular, it will not be sufficient that informaticians can communicate with each other, they have to be able to communicate in a very precise way. Our present inefficiencies in this respect have been held up to ridicule recently ('Wouldn't it be nice if we could write computer programs in ordinary English') by I.D.HILL[15]. He comes to the conclusion, that we should start to teach people - not only computer people - to communicate with each other in a more precise way, at least when giving instructions. Let us hope that progress in programming will enable us to do so.

Literature:

[1] A.P.ERSHOV, Aestetics and the Human Factor in Programming. The Computer Bulletin 16, No.7, 352-355 (1972)

[2] R.GNATZ, Sets and Predicates in Programming Languages. Lectures given at the Marktoberdorf Summer School 1973.

[3] E.W.DIJKSTRA, Notes on Structured Programming. (Report EWD 249) 1969

[4] N.WIRTH, Program Development by Stepwise Refinement. Bericht ETH Zürich, Fachgruppe Computer Wiss.Nr.2, 1971

[5] F.W.ZURCHER and B.RANDELL, Proceedings IFIP Congress 1968 North Holland , Amsterdam 1969

[6] E.W.DIJKSTRA, Comm ACM 11, 341-346 (1968)

[7] E.W.DIJKSTRA, A Short Introduction to the Art of Programming. (Report EWD 316) 1971

[8] P.NAUR, Programming by Action Clusters. BIT 9, 250-258 (1969)

[9] R.CONWAY and D.GRIES, An Introduction to Programming. Cambridge, Mass. 1973

[10] E.W.DIJKSTRA, A Simple Axiomatic Basis for Programming Language Constructs. (Report EWD 372). Lectures, given at the Marktoberdorf Summer School 1973

[11] F.L.BAUER (editor), Software Engineering. Lecture Notes in Economics and Mathematical Systems, vol.81, Springer Verlag 1973

[12] R.WIEHLE, Looking at Software as Hardware. Lectures, given at the Marktoberdorf Summer School 1973

[13] J.DENNIS, Concurrency. In:[11]

[14] F.L.BAUER, Andrei und das Untier. Bayer. Schulbuch-Verlag, München 1972

[15] I.D.HILL, Wouldn't it be nice if we could write computer programs in ordinary English. The Computer Bulletin 16, No.6 (1972)

Appendix:

Variables considered harmful *

Variables (in the sense of programming languages, for example ALGOL 60) seem to belong to the most often used concepts in programming. This is reflected in teaching; for example in the introductory book BAUER-GOOS, Informatik, they are presented quite in the beginning. We are not so sure today whether variables deserve this treatment. We would rather think that they belong in one class with go to statements: they lead to remote connections (of data) like gotos do (of commands) and they open the way therefore to dangerous programming, to programming mistakes. They are to be considered harmful, too. Alas, they are not fully superfluous, which is also true of gotos. In any case, whether or not to use variables is a question of the level on which programming is done. My feeling is that programmers have a tendency, a habit to introduce variables much too early during the development process of a program. This is probably due to the prevailing education. In fact, just good programmers think, when asked to write a routine for the faculty, they have to start in a way like, in ALGOL 68 notation,

```
proc fac = (int n) int :
           (int y := n, int z := 1;
            while y > 0 do z := z×y; y := y-1 od;
            z)
```

and not simply

```
proc fac = (int n) int : if n = 0 then 1
                         else n×fac (n-1) fi
```

because the latter form could even be written down by a mathematician, and they have a professional pride to know a better way. True so! But in a mathematical problem, it is more safe to s t a r t with the mathematicians formulation, like it is advisable to start any problem in the language of the problem. From this starting point, program development towards efficient solution then has to go in r a t i o n a l steps.

In fact, quite a substantial expressive power is available w i t h o u t variables, using only procedures with parameters and branching on conditions. We will therefore start with this level. We will in particular show that parameters can be fully motivated without reference to mathematics. The parameter concept is much wider, it is a fundamental concept of systematic reasoning. So is case distinction, i.e. branching on conditions.

---

* Lecture, presented in honour of A.S. Householder, University of Tennessee at Knoxville, April 24, 1974

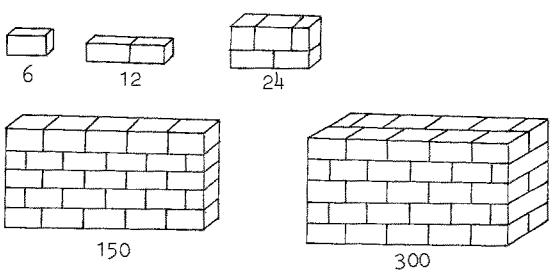
Procedures and their parameters

If we look at a formula describing a classical calculation, like

$$V = \frac{\pi}{3}(r_1^2 + r_1 r_2 + r_2^2)h,$$

we find a difference in the meaning of π (or rather of $\frac{\pi}{3}$), of r_1, r_2 and of h. π denotes a certain constant (or rather $\frac{\pi}{3}$ does! Compare to $\hbar$ in physics!), while r_1, r_2 and h may vary. They are the parameters of the calculation. In fact, in some applications, h may be fixed forever; then r_1, r_2 only are parameters.

Above, we started from an already existing calculation formula and found that it has parameters. Let us now begin with a problem that may originally be defined verbally, like: "A wall is built from bricks of the size $2 \times 5 \times 10$ inches. How much is its weight?" There are many walls, and we could of course determine somehow the weight of a one-brick wall, of a two-brick wall, of a four-brick wall and of some others. We would do this once for all and include it in a catalogue



This would be of great help for people who have not the faintest idea of the multiplication tables (the homo noncalculans does exist!). Still, there would not be enough room in a handy book to keep all the walls that might be built, and furthermore, as we know (but the homo noncalculans has not yet been confronted with the shocking idea), no catalogue could keep them all: they are more than any number. Thus, the British Museum method does not apply, we have to parametrize and, for example, to take the number of bricks to be the parameter.

Parametrization needs the use of an object - in our case a number - for which certain operations are defined. There may be very well non-numerical objects found in the role of parameters.

In our case, every number, every object of the kind 'number' may be the actual value of the parameter. We normally use denotations like 2 5 8 to denote these objects. Moreover, we have to have denotations for the formal parameter. Any marks will do it, in our example we may use n in accordance with modern mathematical usage - in ancient mathematics, verbal constructions were used. We also have to say what the parameters are, in order to distinguish them from other marks, even from letters that are used for different purposes. Thus, for a formula like the one at the beginning of this section, we may differentiate by writing with CHURCH's $\underline{\lambda}$

$$(\underline{\lambda}r_1,\underline{\lambda}r_2,\underline{\lambda}h)\colon \frac{\pi}{3} \times (r_1^{\,2} + r_1 \times r_2 + r_2^{\,2}) \times h$$

or

$$(\underline{\lambda}r_1,\underline{\lambda}r_2)\colon \frac{\pi}{3} \times (r_1^{\,2} + r_1 \times r_2 + r_2^{\,2}) \times h\,.$$

Such a thing, or the recipe

> 'Assuming the n u m b e r of bricks is given, take this n u m b e r six times and you obtain the weight'

is a calculation procedure, where we use calculation in the wide sense of LEIBNIZ ("reckoning" - in the sense of German „ich rechne damit, daß es heute regnet").

We may also allow, for the purpose of abbreviation, freely chosen denotations for such calculation procedures, like *tcc* in the example above or "recipe for weight of walls", and we write then

$\underline{\text{exec}}$ *tcc* (37,43,806) or

"use recipe for weight of walls, for twenty-four bricks". The call, as it is called, brings the identification of the formal parameters with the actual values of the parameters.

Special numerical procedures are: the addition of two numbers, the multiplication of two numbers, the squaring of a number. They have abbreviated standard denotations like + × ² . For notational convenience, we write

3+5 instead of $\underline{\text{exec}}$ +(3,5) or
8^2 instead of $\underline{\text{exec}}$ ²(8)

but this is nothing more than notation.

An essential point, however, is that we allow denotations for formal parameters in the position of actual parameters of some internal procedures. Using triadic addition and multiplication, we have strictly speaking

$$tcc = (\underline{\lambda}r_1, \underline{\lambda}r_2, \underline{\lambda}h) :$$

$$\times \left(\frac{\pi}{3}, +(^2(r_1), \times (r_1, r_2), {}^2(r_2)), h\right) .$$

Thus, we can use procedures within procedures.

Building procedures from primitives

In this way, we may even build up some of our mostly used numerical operations from more primitive ones. We may start with

$$(\underline{\lambda}a, \underline{\lambda}b) : \ll \text{the sum of } a \text{ and } b \gg$$

where the verbal definition is to be understood to be a primitive concept, or with

$$(\underline{\lambda}a, \underline{\lambda}b) : \ll \text{the difference of } a \text{ and } b \gg$$

Introducing an abbreviation for this procedure, for example the freely chosen identifier *diff*,

$$diff = (\underline{\lambda}a, \underline{\lambda}b) : \ll \text{the difference of } a \text{ and } b \gg$$

allows us to write (we omit the exec, since the presence of parameters, in parentheses, indicates the call)

$$diff(8, 3) \quad \text{or}$$
$$diff(a, b)$$

where a,b are formal parameters. We can now define the operation sum, based on the operation difference as a primitive operation, namely

$$(\underline{\lambda}a, \underline{\lambda}b) : diff(a, diff(\ll zero \gg, b))$$

where ≪ zero ≫ is a constant, an operation with no argument. Usually the denotation 0 is introduced for the argumentless operation, the constant ≪ zero ≫

$$0 = \ll zero \gg$$

and with a denotation *sum*, we have

$$sum = (\underline{\lambda}a, \underline{\lambda}b) : diff(a, diff(0, b))$$

where $diff = (\underline{\lambda}a, \underline{\lambda}b) : \ll \text{the difference of } a \text{ and } b \gg$
and $0 = \ll zero \gg$.

We learn from this example that 'primitive' is a relative characterization: somebody might indeed want to consider *sum* as well as *diff* a primitive operation, when working with a computer.

To have a more serious example, we take the greatest common divisor of two numbers

$$(\underline{\lambda}a,\underline{\lambda}b) : \ll \text{the greatest common divisor of } a \text{ and } b \gg$$

If division is available and in particular the remainder of the division of a by b can be obtained, say

$$mod = (\underline{\lambda}a,\underline{\lambda}b) : \ll \text{the remainder of the division of } a \text{ by } b \gg ,$$

then a simple operation for obtaining the greatest common divisor which goes back to EUCLID, can be described as

$$(*) \quad gcd = (\underline{\lambda}a,\underline{\lambda}b) : \underline{\text{if}}\ b \neq 0\ \underline{\text{then}}\ gcd(b, mod(a,b))\ \underline{\text{else}}\ a\ \underline{\text{fi}}$$

If, however, *mod* is not accepted as a primitive, then it can be described by

$$(**) \quad mod = (\underline{\lambda}a,\underline{\lambda}b) : \underline{\text{if}}\ a \geq b\ \underline{\text{then}}\ mod(diff(a,b),b)\ \underline{\text{else}}\ a\ \underline{\text{fi}}$$

It is to be noted that this procedure will not work if $b = 0$, since it then leads to an infinite recursion. Thanks God, in *gcd*, *mod*(a,b) is only used if $b \neq 0$ - what else would we expect from an algorithm that is ascribed to the well-reputed EUCLID.

But now, someone might come and deny that *diff* is primitive. We then have to go a step deeper: based on a primitive operation *succ* with one parameter, we have

$$diff = (\underline{\lambda}a,\underline{\lambda}b) : \underline{\text{if}}\ a \neq b\ \underline{\text{then}}\ succ(diff(a,succ(b)))\ \underline{\text{else}}\ 0\ \underline{\text{fi}}$$

for the difference that is defined within non-negative (natural) numbers. (Obviously, from ≪ zero ≫ and ≪ the successor of . ≫, one can not expect to obtain negative numbers). Again, in *mod*, *diff* is only used in this situation, thanks to the condition $a \geq b$.

To be honest, one should add that the conditions like $b \neq 0$, $a \neq b$ or $a > b$ are also to be considered as primitive operations. In daily life, we are immediately able to perform these comparisons, but this is due to the fact that we are used to work with numbers in radix notation, and the comparison is made in the same way as names are sorted in a directory: by lexicographic comparison. For a computer, this may be too messy a way. Still, $a > b$ can be defined with the help of other primitives:

greater = (λa,λb) : if b ≠ 0 then
if a ≠ 0 then *greater*(*pred*(a),*pred*(b))
else false fi
else true fi

where *pred* = (λa) : *diff*(a,*succ*(0))

Summarizing, we can say: We have allowed definition of a procedure (possibly recursively) with the help of a branching on condition

if >predicate< then >procedure1< else >procedure2< fi

and the use of other procedures. Whether the objects are numbers or something else, is irrelevant. In any case, however, do we need some primitive operations to start with, and there is for example one universal primitive that yields a truth value and can therefore be used in procedures: the comparison

notequ = (λa,λb) : ≪a is not equal b≫

For numbers, we have seen that we can build from this primitive and a few specific ones, comparable to the Peano axiom, in only three steps the Euclidean algorithm; (one knows from formal logic that we have encompassed the general recursive functions).

Result parameters

So far, rather for the sake of simplicity, a procedure has had only one result. Since it has one or more parameters, we could also allow more than one result. Development of mathematical notation was here too inflexible, but we could imagine that *intdiv* denotes formation of the ordered pair of quotient and remainder in integer division, or

intdiv = (λa,λb) : (*div*(a,b) , *mod*(a,b))

A comma may indicate here, as in the parameter list, the collateral occurrence of *div*(a,b) and *mod*(a,b).

However, it may be convenient to be able to r e f e r to some result of a calculation, in particular if many results come out and are to be used in different following procedures - or even some are to be used several times. This leads to new objects, which have the sole meaning of referring to other objects - we call them references.

The seeming paradox, that use of a procedure means specification of all its parameters p r i o r to the entry, and that therefore a result parameter must be specified at the very beginning, makes clear that only a reference to a result can be an actual parameter.

For references, we need also denotations (which again are freely chosen) and a special symbol that connects a reference and the object to which it refers, like

x ≺ 17 (ZUSE)
x := 17 (ALGOL)
x = 17 (misleadingly simplified in some proletaric programming languages)

We then have for example

$$(\underline{\lambda}a,\underline{\lambda}b,\underline{\lambda}\xi,\underline{\lambda}\eta) : \lceil \xi \prec div(a,b),\ \eta \prec mod(a,b) \rfloor$$

Clearly, in any procedure, a formal parameter of the kind reference may only once be found in the ‚body' and this to the left of a ≺ sign, to the right of which a parameter meaning a primitive object or a procedure call delivering such an object stands.

We may also write

$$(\underline{\lambda}a,\underline{\lambda}b,\underline{\lambda}\xi,\underline{\lambda}\eta) : \lceil (\xi,\eta) \prec (div(a,b), mod(a,b)) \rfloor$$

and this should be nothing else than a notational variant.

Furthermore, it adds notationally to the clarity if we write <ξ> whenever that object, to which ξ refers, is meant. Clearly, one will find systems where the explicit use of <•> or of an operator cont can be avoided whenever the circumstances uniquely allow to fill in one. This area, which is the field of notational shorthand, has nothing to do with conceptual bases, although in many programming languages, particularly of the proletaric sort, the lack of conceptual clarity is covered by notational tricks.

Last not least, we mention that references should be of particular value if the same object is referred to from many different places rather than having many occurrences of equivalent objects. This is the same system that is used in references in books - rather than to have in an appendix all relevant articles and books reprinted.

It is clear that references can also be used if a result is needed more than once: while $(a+b)\times(a+b)$ means that the same call $+(a,b)$ is executed twice, $\mu \prec a+b\ ;\ <\mu>\times<\mu>$ contains certainly only one call.
To use the semicolon as an additional mark that determines the order of execution becomes now a necessity.

Variables

All the examples we have considered do not have variables[1], and we have seen that quite complicated examples can be formulated without variables. How do they come into the play?

References may, apart from their result role, have an additional input function:
Since we may write

$$perm = (\underline{\lambda}a,\underline{\lambda}b) :\ (b,a)$$

we should also be able to write

$$perm^* = (\underline{\lambda}\xi,\underline{\lambda}\eta) :\ (\xi,\eta) \prec (<\eta>,<\xi>) .$$

We should also allow things like

$$count = (\underline{\lambda}\xi) :\ \xi \prec <\xi> + 1 .$$

What we have here, are the so called transient parameters.

Now we may also rewrite *gcd* ((*), p.5) with transient parameters

$$gcd^* = (\underline{\lambda}\xi,\underline{\lambda}\eta) :\ \underline{\text{if}}\ <\eta> \neq 0\ \underline{\text{then}}\ (\xi,\eta) \prec (<\eta>, mod(<\xi>,<\eta>));\ gcd(\xi,\eta)\ \underline{\text{else}}\ <\xi>\ \underline{\text{fi}}$$

We are left with a repeated assignment and ξ,η become ‚variable'. For this prize, we have formally eliminated recursivity and replaced by simple repetition. The procedure above is usually written like

[1] 'Variable' is used here in the sense the terminus has in programming languages. We could also say, there is no assignment. We cannot discuss here the theoretical question, whether variables could be totally eliminated.

gcd^* = ($\underline{\lambda}\xi$,$\underline{\lambda}\eta$): ⌈$\underline{\text{while}}$ <η> ≠ 0 $\underline{\text{do}}$ (ξ,η) ≺ (<η>,mod(<ξ>,<η>)) $\underline{\text{od}}$;
<ξ> ⌋

Likewise, we may rewrite mod ((**), p.5)

mod^* = ($\underline{\lambda}\xi$,$\underline{\lambda}\eta$) : ⌈$\underline{\text{while}}$ <ξ> ≥ <η> $\underline{\text{do}}$
(ξ,η) ≺ ($diff$(<ξ>,<η>) ,<η>) $\underline{\text{od}}$;
<ξ> ⌋

which can, of course, be simplified to

mod^* = ($\underline{\lambda}\xi$,$\underline{\lambda}\eta$): ⌈$\underline{\text{while}}$ <ξ> ≥ <η> $\underline{\text{do}}$ ξ ≺ $diff$(<ξ>,<η>) $\underline{\text{od}}$;
<ξ> ⌋

Since repetitions clearly are done in sequence (this comes from the recursive definition), we may allow things done in sequence also elsewhere by using the semicolon as a sequencing sign. Then things such as

a ≺ a+1 ; ~~~~~~ ; a ≺ a-1

are permissible and we arrive also at constructions like

s ≺ 1 ; $\underline{\text{for}}$ i $\underline{\text{from}}$ 2 $\underline{\text{to}}$ 100 $\underline{\text{do}}$ s ≺ s + $\frac{1}{i}$ $\underline{\text{od}}$.

Problems of numerical analysis are mathematically frequently written in a notation that leads itself easily to the use of repetition and variables. This may be one reason for the favorite role variables play in ALGOL 60, while at the same time McCARTHY had developed LISP for non-numerical use.

But now we have the question: How can the original procedures gcd and mod be based on gcd^* and mod^* ?

We may introduce first new procedures $gcd^\times$ and $mod^\times$ which have the original object parameters and additionally reference parameters: These procedures can be based on gcd^* and mod^* :

$gcd^\times$ = ($\underline{\lambda}$a,$\underline{\lambda}$b,$\underline{\lambda}\xi$,$\underline{\lambda}\eta$): ⌈(ξ,η) ≺ (a,b); gcd^*(ξ,η)⌋
$mod^\times$ = ($\underline{\lambda}$a,$\underline{\lambda}$b,$\underline{\lambda}\xi$,$\underline{\lambda}\eta$): ⌈(ξ,η) ≺ (a,b); mod^*(ξ,η)⌋

Inserting in the body of $mod^\times$ the explicit formulation of mod^* above, we obtain

$mod^{\times}$ = (λa,λb,λξ,λη) : ⌈(ξ,η) ≔ (a,b);
while <ξ> ≥ <η> do
ξ ≔ *diff*(<ξ>,<η>) od;
<ξ>
⌋

and we see that the reference parameter η is completely unnecessary, we may introduce the simpler procedure where <η> is replaced by b throughout and thus η can be omitted:

$mod^{\times\times}$ = (λa,λb,λξ) : ⌈ξ ≔ a;
while <ξ> ≥ b do
ξ ≔ *diff*(<ξ>,b) od;
<ξ>
⌋

But we cannot dispense of ξ in the same way, nor can we simplify $gcd^{\times}$ in this way: We should remember that the reference parameters have been introduced in order to come from the recursive form of *gcd* and *mod* to a repetitive form, where the reference parameters serve as 'variables' for the repetition. We cannot get rid of these references, if they are truly variable and not, like η in *mod**, constant. We need them in the body of the procedures, but we do not need them as result parameters: In both cases <ξ> is the 'left-over' result and thus ξ as result parameter gives nothing new. In $gcd^{\times}$, <η> is finally zero and thus η delivers only a trivial result.

Thus, the only role of these result parameters is to make references available in the body. As long as we have them in the parameter list, we may use as actual parameter any reference. We can therefore delete them from the parameter list and introduce them in the body by identification with arbitrary references, i.e. we modify $gcd^{\times}$ and $mod^{\times}$ to

gcd = (λa,λb) : ⌈(λξ,λη) = (any ref, any ref);
(ξ,η) ≔ (a,b);
while <η> ≠ 0 do
(ξ,η) ≔ (<η>, *mod*(<ξ>,<η>)) od;
<ξ>
⌋

and

```
mod = (λa,λb) : ⌈λξ = any ref;
                 ξ ≺ a;
                 while <ξ> ≥ b do
                 ξ ≺ diff(<ξ>,b) od;
                 <ξ>                    ⌋
```

where both *gcd* and *mod* are now equivalent to the original recursive definitions (*), (**).

We see: Variables originate from parameters used in recursive situations. The introduction of references allows in certain cases even formally to go from the recursive formulation to a repetitive situation. If we want to eliminate the reference parameters, we introduce variables explicitly in the body. Their nature is still the same, and this gives a natural definition of scope and range of variables, independent of ad hoc definitions.

The introduction of variables, however, necessitates the sequential notation and thus is a dangerous tool [1]. This is in particular so, if for notational simplicity variables are 'global' to a procedure, i.e. are not listed among the parameters since their denotations can be kept fixed [2].

It is interesting to compare what we get from inserting *mod* in *gcd* with what we have when inserting *mod** in *gcd**. In the former case, we obtain (replacing the denotation ξ of the bound variable in *mod* by α)

```
gcd = (λa,λb) : ⌈(λξ,λη) = (any ref, any ref);
                (ξ,η) ≺ (a,b);
                while <η> ≠ 0 do ⌈λα = any ref;
                                  α ≺ <ξ>;
                                  while <α> ≥ <η> do
                                  α ≺ diff(<α>,<η>) od;
                                  (ξ,η) ≺ (<η>,<α>)  ⌋ od;
                <ξ>                                  ⌋
```

On the other hand, we can transform *gcd** equivalently to

```
gcd* = (λξ,λη) : ⌈while <η> ≠ 0 do
                          (ξ,η) ≺ (<η>, mod*(ξ,η)) od;
                 <ξ>                                 ⌋
```

---

[1] The idea to program without variables has been advocated recently, among others, by J. BACKUS: Reduction Languages and Variable Free Programming. IBM Research, RJ 1010, April 7, 1972, San José, CA.,USA

[2] See W. WULF and M. SHAW, Global Variable Considered Harmful. SIGPLAN Notices 8, No. 2, 28-34 (1973).

(replacing *mod*(<ξ>,<η>) by *mod**(ξ,η) is a nontrivial step and needs justification!) and obtain

```
gcd* = (λξ,λη) : ⌈while <η> ≠ 0 do
                    ⌈while <ξ> ≥ <η> do
                        ξ ≔ diff(<ξ>,<η>) od;
                    (ξ,η) ≔ (<η>,<ξ>)      ⌋ od;
                <ξ>                          ⌋
```

Thus, the introduction of references and the insertion of procedures within procedures do not commute: they produce in general different, however equivalent programs.

CHAPTER 3.: OPERATING SYSTEMS STRUCTURE

C. A. R. Hoare

The Queen's University of Belfast
Belfast, Northern Ireland

The Structure of an Operating System

Draft May 1975

Key words and phrases:
Programming Languages, Implementation Languages, Macros, Operating Systems, Protection, Structure.

Abstract:
This paper describes the use of the class and inner concepts of SIMULA 67 to express the multi-level structure of an operating system. A comparison is drawn between compile-time checking and run-time protection.

1. Introduction.

Dijkstra [4] has proposed that an operating system should be structured as a series of levels, each of which uses the previous lower levels to implement a more pleasant abstract or virtual machine for the benefit of the subsequent higher levels. The lowest level of the hierarchy is the bare hardware, and the highest level is the eventual user program. This paper suggests that the class and inner features of SIMULA 67 [1] provide an appropriate programming language notation for describing such a structure. The suggestion is illustrated by the stepwise development (bottom-up) of parts of a very simple pseudo-offlining system. Emphasis is placed on compile-time security checking, which enforces the structuring discipline between parts of the operating system; and an analogy is drawn with the run-time protection methods which impose a similar discipline upon the user program, and thereby protect the system against its possible misbehaviour.

The notations of this paper are taken from SIMULA 67, with the PASCAL [11] form of variable declaration to emphasise the absence of references,and to suggest the use of normal stack methods of dynamic storage allocation. They are not necessarily the best notations for an operating system structuring language.

The class method of design and implementation of a representation for abstract data is explained in [6]. The main innovations of this paper are:

(1) The introduction of an asterisk in the declaration of identifiers which are to be accessible to the subsequent user of a class. Identifiers declared without an asterisk may be used only within the class declaration; they are hidden from the outside, as recommended in [10,12].

(2) The introduction of an inner statement to mark the position at which (conceptually) the appropriate block of the user program will be executed. This permits the designer of a class to complement the compulsory initialisation of local variables of a class by the appropriate compulsory finalisation.

(3) The distinction between a class which may have multiple instances, from one which can have only a single instance at a time.

(4) The nested use of class declarations; this is not strictly an innovation to the language; but its effects are certainly surprising and useful.

2. A class with inner.

A primary prupose of an operating system is to share the equipment it controls among a number of users. In a multiprogramming system, several processes may be in concurrent execution, and each of them may from time to time require exclusive access to a particular item of equipment, for example, a single lineprinter. In this section we shall construct a class which implements a separate "virtual" lineprinter for each process, taking advantage of the fact that each process requires its virtual lineprinter only for isolated periods. Such a period is represented within a process by a block (fig. 1b) in which a new variable or "instance" of class lineprinter is declared and given a name (e.g. report). Output to this virtual lineprinter is then effected by calls on the "compound" procedure identifier report.output. A compound identifier consists of the name of a variable (report), followed by a dot, followed by the name of a procedure (output) invoking some operation on that variable, and followed by any actual parameters of the procedure. Such a call can occur only within the scope of the declaration of the variable concerned.

In order to prevent chaotic interleaving of output from several processes, we need some synchronisation mechanism whereby a process can delay itself when it needs a lineprinter currently in use by another process. Because of its familiarity, we shall here adopt the semaphore for this purpose, and declare a global instance &pmutex of type semaphore, initialising it to 1. (Fig 1a). The declaration of the class lineprinter, which implements the virtual lineprinter concept, has the same form as a procedure declaration. It has a local procedure output (with a machine code body) which carries out actual output to the actual lineprinter. This procedure is marked with an asterisk to show that it can be called as the second part of a compound identifier, from within a block in which

the first part has been declared. The compound tail of the class body consists of three statements:

(1) P(&pmutex) which "initialises" each variable of the class by claiming its exclusive use of the actual lineprinter,

(2) V(&pmutex) which "finalises" each variable of the class by releasing the actual lineprinter for use by other processes,

(3) these are separated by an inner statement, which represents the block of user program in which an instance of the class is locally declared, and from which calls on the procedure output may be made by an appropriate compound identifier.

The effect of the declaration of a class instance can be explained by a copy rule similar to that for a procedure call in ALGOL 60.

0. Multiple declarations in a single block are treated as if they were in nested blocks.

1. Take a fresh copy of the body of the class declaration.

2. Turn every starred identifier into a compound identifier by prefixing to it the name of the instance being declared. Such compound identifiers then have the same ALGOL scope rules as other identifiers.

3. Make sure that all unstarred identifiers of the class body are unique, for example by systematically appending to the identifier a sufficiently high numeral.

4. Replace the inner of the class body by a copy of the remainder of the block in which the declaration of the class instance occurs.

5. If the programming language allows jumps out of a block, every such jump must be preceded by a copy of the finalisation code of the class. However, for simplicity, in the remainder of this paper we shall assume that the language does not allow jumps; we shall find that we do not miss them.

The application of these rules to the examples of figures 1a and 1b is given in figure 1c.

These copy rules show that the introduction of classes does not require the introduction of references into the language or into its implementation. Furthermore, the normal efficient stack methods of storage allocation are fully adequate for class instances, as for other declared variables. There is no reason why an implementation should not use closed subroutines in place of literal substitution in the source text, as in the normal implementation of ALGOL procedures. Nevertheless, literal macro-substitution may be in many cases the most efficient method, especially if it is followed by certain simple local optimisations, such as referring local workspace of a block to the activation record of an enclosing block. In effect, the whole class concept can be used as a well-disciplined method of macro-processing in any programming language, even assembly code. I am grateful to Brinch Hansen for pointing this out.

However, it is very unfortunate if the designer or user of a class persists in understanding it solely in terms of literal substitution. The purpose of the class, like that of a procedure, is to express and implement some useful abstraction. The user of a class should regard it as an abstract data type, which can be used to declare new variables, on which certain primitive operations (i.e., the accessible procedures) can then be performed. The details of the implementation method can and should be ignored.

3. A nested class declaration.

In the example of the previous section, there is a close logical and structural connection between the global semaphore &pmutex, its initialisation to unity, and its use in the lineprinter class. It would be nice to represent this close connection by a single textually contiguous program structure; and this section suggests that the class is also a suitable structure for this purpose. To show this, we declare a class &pallocator (Fig 2a), in which both &pmutex and the class lineprinter are locally declared identifiers. Note that &pmutex is

unstarred, and therefore remains inaccessible from within the inner of the &pallocator, which will be the whole of the rest of the system in which the instance &pa:&pallocator is declared. However the class lineprinter will be accessible by the compound identifier &pa.lineprinter; and the procedure output is accessible (as before) only from within a block in which an instance of that class is declared. (Fig. 2b). Thus the effect of the program in Fig.2 will be identical to that of Fig.1, as may be shown by application of the copy rule to the declaration of &pa.

4. Compile time checking.

The usefulness of a program structuring method can be greatly enhanced if the programming disciplines required to uphold the structure can be enforced by a compile-time check. For example, part of the value of the procedure of ALGOL 60 or the SUBROUTINE of FORTRAN as methods for expressing abstractions, is that compile-time checkable scope rules prevent the user from accessing or interfering with local variables which should be of no concern to him. The value of compile-time checks in avoiding error is even greater when an already written program has to be slightly modified.

It is therefore illuminating to list some examples of the kind of error which might cause disaster in an operating system.

(1) A process may output to the lineprinter without first acquiring it by P(&pmutex).

(2) A process may release the lineprinter by V(&pmutex) without having acquired it.

(3) A process may fail to release the lineprinter when finished with it.

(4) A process may attempt to acquire a second lineprinter when only one is available.

(5) A process may declare its own separate instance of an &pallocator.

The first three errors can all be averted by a compile-time checking of the scope rules associated with classes. The only way of bypassing these checks is by direct use of machine code; we must therefore find some other way of controlling the use of machine code, for example by a compile-time switch which can be switched off but never on.

The next two errors (4) and (5) can be averted by making a new rule that no instance of a class can be declared within the scope of another instance of the same class. This rule can be enforced by a compile-time check (except possibly in the case of recursive procedures, which can profitably be omitted from a language designed for implementing operating systems). However if there are several lineprinters, the declaration of multiple instances should be allowable, perhaps by prefixing the class declaration by the word multiple.

Of course, there are many other programming errors that can never be checked at compile time. Nevertheless, the compile-time elimination of trivial errors makes the detection of substantial errors much easier.

5. Multilevel Structuring.

The &pallocator is a simple module which on the basis of a single hardware lineprinter creates up to one "virtual" lineprinter for each process which wants one; and it achieves this effect by ensuring that only one process is using the hardware at a time. But it is rather inadequate for sharing a lineprinter among separate jobs submitted to a batch processing system; for example, it gives no indication of where the output of one job ends and another one begins. It doesn't even ensure that each new job starts its output on a fresh page; so it would be impossible to separate the output for distribution to its owners.

We therefore introduce the concept of a linefile, which by definition always starts at the top of a fresh page with a complete line of asterisks, and ends at the bottom of the last page with a complete line of asterisks. The operator can therefore see where to separate the files of different users. To prevent confusion,

we also stipulate that a linefile must not contain a complete line of asterisks.

The declaration and use of a linefile in a user process is identical to the previous case, and is illustrated in Fig 3b. The concept of a linefile is implemented as a class local to a class &fallocator, which has a unique instance &fa. The only apparent difference in the user process is the use of different names; and even this change can be avoided if desired by systematic use of the name lineprinter instead of linefile and &pa instead of &fa.

The &fallocator class can, of course, be programmed from the beginning as a separate class, or it could be derived by inserting additional statements into the &pallocator class. Instead of doing this, we show how the &fallocator can be programmed, taking advantage of the previous &pallocator class to administer the lower level aspects of exclusion and physical output.

To accomplish this, the unique instance &pa of the &pallocator class is declared within the &fallocator class; and this makes it possible for the linefile class to declare an instance &p of the line printer class to carry out the actual output of lines to the actual lineprinter. However, the user of a linefile cannot call the procedure &p.output directly; he can call only the output procedure of the linefile. Consequently, there is no way in which he can avoid the run-time check against outputting a complete line of asterisks.

The application of the copy rule (Fig. 3c) to the declaration of &pa shows that the required interleaving of actions from the two classes can be achieved automatically. However, it would be wrong for the programmer to think solely in terms of the interleaving; he should simply regard the &fallocator as a higher level concept implemented on the basis of the lower level &pallocator.

6. A third level.

A further example of the power of the class and inner methods for multilevel structuring is the implementation of the familiar concept of a pseudo-offlined

output. In order to achieve this, we need a filing system, with the structure shown in Fig. 4(a). It provides a class scratchfile, which is intended to have multiple instances. Each scratchfile starts off as an output file, accepting output commands. A rewind repositions the file at the beginning, and enables it to carry out input instructions for as long as the Boolean function more indicates that there are further lines to be read.

The class pseudolinefile is shown in Fig. 4(b); it is also capable of multiple instances. The output procedure which it provides to the user in fact outputs its line parameter to a scratch file s acquired for this purpose. The acquisition of an actual lineprinter for output of the file is delayed until the finalisation section, i.e. after the user has finished all his output.

After declaration of p&f:pseudooffline output it is thenceforward impossible for a process to acquire a linefile for direct output to the lineprinter, since he can no longer refer to &fa, which is the unique instance of the &fallocator class. However, if the operating system designer wishes to preserve the option of direct output, he can in principle do so by attaching an asterisk to the declaration of *&fa. Then the user can still access it by a doubly compound identifier, for example:

report: p&f.&fa.linefile;

The design of a filing system and of a series of classes for input (i.e., cardreader, cardfile, and pseudocardfile) is left as an exercise for the enthusiastic reader.

7. Error Control.

One of the most important functions of an operating system is to ensure that the "virtual" facilities provided to its users shall be more reliable than the actual hardware used to implement these facilities. This is best ensured by designing each level of the system to deal with all errors which it can at its own level; and to report other failures to some higher level which might be able to take more global action.

For example, the procedure output of the lineprinter class should be able to recover from certain hardware conditions (e.g. lineprinter switched off) which can be detected before the line has been output. But there are certain errors which may have actually caused an incorrect line to be printed. For this reason, the lineprinter class needs a local variable

*spoiled: Boolean;

which is set false initially, and is set true if and when the possibility of spoiled output is detected. There is nothing that can be done at the level of the lineprinter to conceal the occurrence of such an error.

At the level of linefiler, the concept of page separation becomes significant; and at this level it is possible to deal with the "paper low" condition on the lineprinter by instructing the operator to load a new pile, and by waiting until he does.

At the level of the pseudo linefile, it is feasible to deal with the spoiled line conditions, detected at the lowest level. For example, the pseudo linefile may instruct the operator to scrap an entire page on which a spoiled line occurs, and may repeat output of its contents. In the case of persistent failure, it can call the maintenance engineer, carry out diagnostic procedures to help him find the error, and then output the whole file again from the beginning.

The phenomenon of error and failure is very pervasive in the design of an operating system, and it is not always possible to deal with it except by modifying or even disrupting the overall structure. Furthermore, it does not seem often possible to implement error handling as a class separate from that whose errors it controls. And the problem would seem not to be simplified by the use of jumps. A hardware designer who genuinely wishes to help the software will exert his best endeavours to maximise reliability.

8. Accounting.

Another important function of an operating system is to maintain a record of significant events on a log file which is printed out whenever the system is closed down. This log should also keep an account of the cost of each job, so that each user may later be correctly charged. But the user must be able to limit his liability by declaring the maximum cost he is prepared to pay for each job; and the operating system must be permitted to terminate a job which exceeds the limit. We assume that this limit is written together with the identity of the user, on a job card at the head of the user's deck of cards, and this card is passed as a parameter to each instance of the useraccount class, shown in Fig. 5. This class provides a procedure *charge which will be used by higher levels of the operating system to record costs incurred, and a Boolean variable overlimit which may be tested at appropriate intervals to determine whether the user job should be continued.

An interesting problem arises with the sharing of the single log file between many processes: when one process calls the procedure log.output, all other processes must be prevented from entering that procedure for the duration of the call; otherwise the interleaved updating of variables local to the log would in general cause chaos. For this reason, Brinch Hansen [2] has suggested that declarations of variables which are to be updated by several processes should be preceded by the word shared; and in implementation, an exclusion mechanism such as a semaphore must be introduced to prevent reentrancy of the procedures which update the variable.

In figure 6 the user account class is used in an input output class, which implements the concept of system input/output for a very simple pseudo-offlining system. Note how the separately programmed concepts of pseudoofflining and accounting are brought together only at this high level. The price that is paid for such a degree of isolation and structure is the possible extra overheads of calls on the input and output procedures. It is sometimes desirable that an implementation should use macro-substitution techniques both for classes and for

procedures to reduce this overhead, even though this removes some of the carefully designed structure of the system.

9. The top level.

The top level of an extremely simple pseudo-offlining system can be expressed as a collection of virtual machines (Fig. 7), each of which is a separate process, executing a stream of user jobs, and taking turns to read the next job from the actual cardreader, to execute a user program in actual main store, and to print its results on the actual lineprinter.

Inside its perpetual loop, each virtual machine declares an instance of the input output class to carry out its user's input and output. This requires acquisition of a pseudo cardfile, which involves first the acquisition of an actual card reader, and then the copying of the user's input file to a scratchfile. The actual card reader is then released, and another scratchfile is obtained for pseudooffline output.

Next the declaration

store: userstorage

is intended to acquire an area of main store to hold the user program, and initialise it to contain some standard service routine, for example, a compiler or a job control language interpreter (which may subsequently load a compiler of the user's choice). The user's program is then executed inside a loop which periodically tests whether it ought to be terminated, for example because of exceeding the cost limit, occurrence of a parity error in the user store, or even voluntary ending of the program. Finally, all resources will be released by exit from the block.

When the user program requires to perform input and output, it will do so by calls on the procedures system.input and system.output, in exactly the same way as if the user program were executed from within the inner of a class in which the variable system has been declared with an asterisk. Thus the operating system

may regard the user program as simply the innermost inner, representing the highest level of abstraction. This analogy is pursued in the next section.

10. Protection.

It is very attractive and convenient to take a dump of the compiler after compiling the operating system, and to use this version as the standard initial program in each user's store, and to insist that the computer is programmed exclusively in this language, and using this compiler. If the language is secure and correctly implemented, the compile time checks associated with classes, procedures, and parameters will ensure that no user program expressed in the language can adversely affect the operating system or any other user in any way. And a system in which all attempts to violate operating system protection can be detected and averted at compile time would be exceptionally efficient and exceptionally convenient, as is shown by experience with the Burroughs MCP system.

Unfortunately it is not always good enough for a job-shop operating system.

(1) The risks of hardware error during execution of user program should be taken into account.

(2) No currently fashionable programming language has the requisite property of security.

(3) A compiler is too large an item of software to put so much trust in.

For these reasons, it is necessary to supplement or replace the compile time security checks by protection at run time. The necessary criterion of a run time protection system is that it will deal safely with a "user program" which has been submitted by a random number generator; and no lesser achievement is acceptable!

In theory, all run-time protection schemes are equivalent to direct interpretive execution of the instructions of the user program. Each instruction is analysed to check whether it violates protection; if so, the program terminates;

otherwise the instruction is executed. Fortunately, the hardware of a multi-programming computer usually possesses a facility whereby the hardware itself will interpret long series of user program instructions which do not violate some obvious protection rule (like access outside certain store bounds); and will return control to a software interpreter in the operating system only on encountering a suspicious instruction. The software interpreter then analyses whether the suspicious instruction should be interpreted as a call on a procedure of the operating system, and whether its parameters are valid. Thus the only instructions which suffer the overhead of software interpretation are user calls on operating system functions.

By suitable design of hardware protection mechanisms (for example, ENTER capabilities [9]), it may be possible to reduce even these overheads. However, the requirements for checking parameter validity, and for implementing the scope rules for classes, are quite complicated; and even some of the more elaborate capability proposals would not seem to be adequate for this purpose.

It is also desirable to allow the protection mechanisms and the associated hardware interpreter to be called in a nested fashion, by the user program itself. Thus for example it would be possible for an in-core compiler to transfer control to a program it has just compiled, and have it executed at full speed with hardware protection of the integrity of the compiler itself and its run-time library. Such a hardware nested protection scheme could also in principle be used to establish protection between one level of an operating system and another; but this would seem to be distinctly inferior in efficiency and safety and convenience to the compile-time checking techniques advocated in this paper.

The suggestion that I would like to emphasise is that the design of hardware and software protection mechanisms should be oriented towards the implementation of an analogue of the scope rules of a high level language; and furthermore, the protection requirements of an operating system can be best expressed in a high level language by the implicit scoping of procedures and classes, and not by explicit and error-prone instructions for loading protection registers, switching mode bits, and so on.

Conclusion.

A picture of the overall structure of a simple pseudoofflining system is shown in Fig.8. The relationship denoted by the lines is that the higher module is implemented with the aid of the lower module, and therefore depends upon the working of that module. Note that this view of the structure is radically different from the view obtained by identifying a module with a process, related to other processes by a stream of messages.

The real criterion of a well-structured program is not merely aesthetic elegance, which is undoubtedly less than that of a Greek temple; but rather the criterion that small changes in the problem should lead to small changes in the program. The reader may evaluate the merits of the proposed operating system structure by attempting a number of small extensions or modifications, for example, to cater for addition of a second cardreader or lineprinter. If it is easy to find which modules to change, and easy to ensure that the changes have no unexpected interactions with other modules, then the labour of finding and maintaining the structure has not been wasted.

Acknowledgements.

The ideas of this paper are currently being tested in a pilot project for the implementation of a model operating system, which enjoys the support of the Science Research Council of Great Britain.

This paper also owes much to IFIP W.G.2.3 (the working group on programming methodology) and to its individual members for many conversations and discussions leading to the development and refinement of the ideas expressed; to R.M. McKeag, D. Bustard and J. Clarke, who are engaged on the pilot project mentioned above, and to J. Elder, who has investigated the concept of a class by an implementation in the framework of PASCAL [5].

```
ℓpmutex:semaphore;
class lineprinter;
   begin procedure* output (ℓ:line);
      begin "machine code" end
      P(ℓpmutex);
      inner;
      V(ℓpmutex)
   end;
ℓpmutex:=1;
```

1a. The lineprinter class.

```
begin report:lineprinter;
      . . .;
      report.output ("CONTROL LIMIT EXCEEDED");
      . . .
end;
```

1b. Use of lineprinter from within a process.

```
begin procedure report.output (ℓ:line);
      begin "machine code" end;
      P(ℓpmutex);
      . . .;
      report.output ("CONTROL LIMIT EXCEEDED");
      . . .;
      V(ℓpmutex)
end;
```

1c. Application of copy rule.

```
class &pallocator;
begin &pmutex:semaphore;
      class *lineprinter;
      begin procedure *output (&:line);
            begin " machine code "  end;
            P(&pmutex);
            inner;
            V(&pmutex);
      end lineprinter;
      &pmutex:=1;
      inner
end;
```

2a. The &pallocator.

```
&pa:&pallocator; comment global to all processes;
. . .
begin report:&pa.lineprinter;
      . . .;
      report.output ("CONTROL LIMIT EXCEEDED");
      . . .
end;
```

2b. Use of &pa from within a process.

```
class &fallocator;
begin &pa:&pallocator;
      class *linefile;
         begin &p:&pa.lineprinter;
               procedure *output (&:line);
                  begin if & = "all asterisks" then &p.output ("error")
                                               else &p.output (&)
                  end;
               "throw to top of next page";
               &p.output ("all asterisks");
               inner;
               "throw to bottom of this page";
               &p.output ("all asterisks")
         end linefile;
      inner
end &fallocator;
```

3a. The &fallocator class.

```
&fa:&fallocator; comment global to all processes;
. . .
    begin report:&fa.linefile;
          . . .;
          report.output ("CONTROL LIMIT EXCEEDED");
          . . .
    end;
```

3b. Use of &fa.linefile from within a process.

```
class ℓfallocator;
begin ℓpmutexl:semaphore;
      class *linefile;
      begin procedure ℓp.output (ℓ:line);
            begin "machine code" end;
            procedure *output (ℓ:line);
            begin if ℓ = "all asterisks" then ℓp.output ("error")
                                         else ℓp.output (ℓ);
            P(ℓpmutexl);
            "throw to the top of next page";
            ℓp.output ("all asterisks");
            inner;
            "throw to bottom of this page";
            ℓp.output ("all asterisks")
            V(ℓpmutexl)
      end linefile;
      ℓpmutexl:=1;
      inner;
end ℓfallocator;
```

Fig. 3c. Application of copy rule.

```
class filing system;
begin ...
    multiple class *scratch file;
    begin procedure *output (ℓ:line); ...
          procedure *rewind; ...
          procedure *input (result ℓ:line); ...
          function *more:Boolean; ...
          ...
    end scratch file;
    ...
end filing system;
fs:filing system;
```

4a. Outline of filing system.

```
class pseudo offline output;
begin ℓfa:ℓfallocator;
    multiple class *pseudolinefile;
    begin s:fs.scratchfile;
          procedure *output (ℓ:line);
              begin s.output (ℓ) end;
          inner;
          begin ℓ:ℓfa.linefile; t:line;
                while s.more do
                    begin s.input (t);
                          ℓ.output (t)
                    end;
          end
    end pseudolinefile;
    inner
end;
plf:pseudooffline output;
```

4b. Pseudooffline output.

```
class accounting;
begin log:pseudolinefile;
      class useraccount (jobcard:card);
      begin cost,costlimit:integer;
           *overlimit:Boolean;
            procedure *charge (amount:integer);
                begin cost:=cost+amount;
                      overlimit:=cost>limit
                end;
            cost:=joboverhead;
            costlimit:="appropriate field of jobcard";
            overlimit:=cost>limit;
            log.output ("START OF followed by jobcard and time");
            inner;
            if overlimit then cost:=costlimit;
            log.output ("END OF followed by jobcard, time and cost");
      end useraccount;
      log.output ("START OF LOG followed by date and time");
      inner;
      log.output ("END OF LOG followed by date and time")
end;
acc:accounting;
```

Figure 5.

```
class input output;
begin cr:pcf.pseudocardfile;
      ℓp:plf.pseudolinefile;
     *ua:acc.useraccount (cr.jobcard);
      procedure *input (c:card);
          begin  cr.input(c); ua.charge (cardcost) end;
      procedure *output (ℓ:line);
          begin ℓp.output(ℓ); ua.charge (linecost) end;
      ℓp.output ("START OF followed by cr.jobcard");
      inner;
      if ua.overlimit then ℓp.output ("overlimit message");
      ℓp.output ("END OF followed by cr.jobcard")
      if cr.more then "output unread cards"
end inputoutput;
```

Figure 6.

```
process virtual machine;
    while¬closedown do
       begin system:input output;
       store: user storage;
       while   "termination condition has not occurred"
          do   "execute a portion of user program";
       end
```

Figure 7.

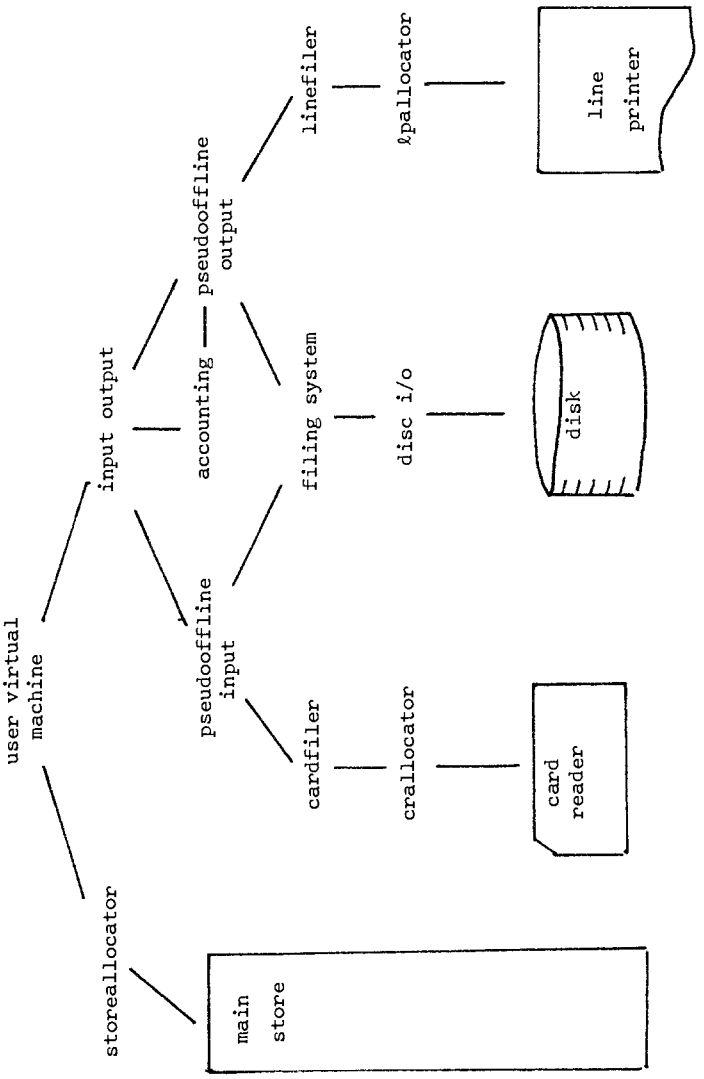


Fig. 8. Structure of the system.

References

1. G. Birtwhistle et al. SIMULA begin.
Student Litteratur and Auerbach, 1974.

2. P. Brinch Hansen. Structured Multiprogramming.
Comm. ACM, 15, 7 (July 1972), 574 - 577.

3. O.-J. Dahl. Hierarchical Program Structures.
in Structured Programming, Academic Press, 1972.

4. E.W. Dijkstra. Structure of the T.H.E. Multiprogramming System.
Comm. ACM 11 (1968), 341.

5. J. Elder, Ph.D. thesis, Queen's University, Belfast.

6. C.A.R. Hoare. Proof of correctness of data representations.
Acta Informatica 1 (1972), 271 - 281.

7. C.A.R. Hoare. Monitors: an Operating System Structuring Concept,
Comm. ACM, 17, 10 (October 1974), 549 - 557.

8. P. Naur (ed.). Report on the Algorithmic Language ALGOL 60.

9. R. Needham. Protection and Process Management in the CAP computer.
in Protection in Operating Systems, IRIA 1974.

10. D. Parnas. Information distribution aspects of software methodology
Proceedings of IFIP Congress, 1974.

11. N. Wirth. The Programming Language PASCAL.
Acta Informatica 1, 1 (1971), 35 - 63.

12. J. Palme. Protected Program Modules in SIMULA 67.
FOA P report C8372 Sept.1973.

G. Seegmüller

versity of Munich and Leibniz Computing Center
Germany

Language Aspects in Operating Systems

Table of Contents

1. The Role of Language in Operating Systems 268

1.1 Language and Function 268

1.2 Language and People 269

1.3 Language and Computing Systems 270

1.4 Language and System Construction 274

2. Are there Special Requirements for Systems Programming 277

3. A Remark on Current Systems Programming Languages 279

4. How Does the Successful Systems Programmer Survive 280

5. Design Criteria for an Operating System Programming Language 281

6. Language Mechanisms Assisting in the Construction of Structured Systems 282

7. Example: The Language System ASTRA 285

8. Concluding Remarks 289

9. Acknowledgements 290

10. Literature 290

1. The Role of Language in Operating Systems

If not stated otherwise, we use the term Operating System in the wider sense, i. e. it is understood to include all programs also on the user level such as compilers, linking programs, utilities and user written programs.

1.1 Language and Function

An operating system offers to its users a set of functional primitives. By means of these primitives a user may create new functions the representations of which he may wish to be manipulated. Finally he may want that these functions will be executed.

All this is impossible without the use of some language. It is obvious that the properties and the quality of such a language are among the decisive elements concerning success or failure of the user's projects. In this paper we shall discuss some of these language properties.

There are different ways how to represent a function by means of languages. For instance, it may be done in an axiomatic manner which is in contrast to the algorithmic way. Similarly we may speak of static versus dynamic representations or of implicit versus explicit methods.

A language may be suited for processing by a machine only, or it may be limited to usage between human beings, or it is suitable for both purposes. There are relatively well-established standards as to how programs of a certain language can be processed by a machine, i. e. how well they can be represented, manipulated, analysed, translated and executed. The body of knowledge about the suitability of a language for usage by human beings, however, is still rather poor. Later on we shall discuss some issues in this area.

One language may be designed for the sole purpose of formulating and executing action sequences. Another language may only permit the description of operands (or data). Most languages offer both facilities.

Languages differ widely in their degree of explicitness with which statements about objects may be made. In almost all cases there is a certain connection between this degree of explicitness (level of detail) of a language and its domain of practical applicability. This domain indicates how the set of constructable functions is limited, in a practical sense. By this limitation the algorithmic exploration of new fields may be furthered or hindered. A language may be "blind" for a certain direction of thought. On the other hand it may encourage the development of a new area.

The properties of "higher" languages compared with the properties of "lower" languages indicate that there is a price for a better adaptation to a class of problem solutions. Usually there is a reduction in universality. Better structuring possibilities seem to be tied to less generality, a higher "quality" of formulation seems to reduce the domain of practical applicability, greater construction safety is consequential for the size of the set of functional primitives which may be offered by a language.

There is a strategic aspect with the introduction of a new language. It may influence the suppression or stimulation of activities. (Platon:... nothing is more dangerous for the states than the introduction of a new music ...).

1.2 Language and People

There are typical intuitive categories of judgement of the ways in which functional primitives are represented in a language. Constructs may be found attractive or repulsive,

suggestive of their meaning or misleading, readable or not readable.

In most cases there is an objective as well as a subjective component to such a judgement. We can observe an evolution of styles and traditions which is often different in different camps or schools of thought. There are favourite languages which are maintained and developed further. Doubtlessly, the use of a certain language forms certain thinking habits.

There is the phenomenon of strong emotional engagement, of unbelievable irrational eruptions of otherwise very reasonable people as soon as certain language matters are touched upon. This very fact has a deep sociological root. It has to do with the overall role of language in general. There is this primitive feel of pleasure of the full understanding of all shades of a code, the feel of unity, concord and safety with the "code brothers". Threatening phenomena are banned by an explanation in terms of the own guild or a new term is invented. Rarely, a term is adopted from another guild. One sticks together. This saves energy and causes minimal disturbance. In addition, this helps to protect own dominance.

The successful introduction of a language is an excellent strategy for long-term domination, or influence at least. It is an old theme of mankind to push languages. The Babylonians were severely punished by the confusion of their language.

1.3 Language and Computing Systems

The communication with computing systems happens by means of programs or program pieces, by data (which are not programs or program pieces), and by signals which stand for the activation or deactivation of programs, program pieces and data which are coded in a special way.

There is an inevitable involvement of language in all communication with computing systems. The properties and the quality of the languages involved do influence the quality of the communication. Languages form the frame of the interface with the computing system.

The above is equally true for the communication between functional subunits of computing systems themselves, be it hardware or software or any combination thereof.

There are very different uses of language with a large computing system. For the running system we may distinguish:

a. User languages for the solution of user problems. They range from so-called general purpose languages to very special languages. This collection may include languages for the manipulation of user objects in the system, as, e. g., command languages, transaction languages and also languages for the control of technical processes.

b. Operator languages for the system operator's information and for redirecting the system. This is an area which is almost always in a poor state. It is often confined to a clumsy human access mechanism to the system store. There is no or only an insufficient high level mechanism for system redirection (e. g., one would like to say "from now on, at most 75 % of processor time should be spent for time-sharing", or "from now on, for two hours, absolute priority for the terminal group C"). Instead, operators have to implement such wishes by assigning new values to a dozen "system parameters". In most cases an operator is overtasked by such a wish.

c. System measuring languages for effectuating general as well as specially directed system measuring. It is an almost undeveloped field.

d. User-administration languages for administering access-rights and resources for users. The ultimate garbage collector function of a system is part of the semantics of such languages. Again, this is an underdeveloped field.

e. Software-maintenance languages for maintaining the system's software. They are languages for the last phase of operating system projects. They do not seem to exist. The maintenance personnel has to implement maintenance wishes through typical user job functions and even then, in most cases, the system has to be empty in order to function correctly.

f. Hardware-maintenance languages for changing the active configuation and for carrying out tests. They are, in a sense, languages for the last phase of hardware projects. Currently, only a few scattered functions seem to be available on some systems.

While an operating system or a larger application system is developed we may distinguish:

g. Specification languages for the first design stages. The emphasis is more on the what than on the how. Objects of the design are described rather implicitly.

h. Construction languages for later design stages. Levels of increasing detail have to be considered, hardware limitations play an important role. The emphasis is more on the "how".

i. Implementation languages for the coding of the system.

j. Test languages for preparing test environments and carrying out actual tests.

k. Project management languages for the administration of the development process and for the control of the actual system building from proper components.

There were attempts in the past to have one unified system encompassing all of the above five areas. Most of these projects seem to have failed. The temptation to incorporate too much seems to be irresistible.

Within any larger software or hardware system itself we may identify interface languages, e. g.:

l. Virtual machine languages formed by the operation and operand sets of the conceptual virtual machines of the system and by their composition rules.

m. Intermediate languages between system components like, e. g., translators, linkers, loaders and service routines. These languages describe admissible parameter structures of system components.

The importance of languages in connection with computing systems is also a growing economic factor. This factor is strongly influenced by the quality and the number of languages which have to be absorbed. Typical activities are the design of languages, the production of translators, the marketing of programming systems together with the necessary accompanying material, the use of the languages - this includes very often the tacit and subconscious adoption of their "philosophy" - and the maintenance of the products.

Experience has shown that only the limiting factors of processor speed, storage capacity and the number of programmers delineate the domain of problems which are actually attacked and which are put on the back of national economies. Often, problem complexity, implementation difficulties and the abilities of programmers are only of minor concern. Depending on the quality of the languages employed in such a project the hour of truth will come soon, late or never.

A sizeable waste of resources is a possible consequence, caused by massive accumulation of overheads, by late project failures or by catastrophic or sneaking malfunction of systems in use.

1.4 Language and System Construction

The development of a convincing structure is a necessary condition for a successful system. Learning and understanding the construction process is one of the central topics of computer science. A convincing construction seems to be infeasible with an "unsuitable" language.

A program is to describe a class of correct executions. The very same program may be the "carrier" of radically differing executions. During the construction and implementation phase we have only static program texts of increasing explicitness. We wish, as clearly as possible, to understand from these texts the class of potential executions. We want "to tame" the executions by working on the static text. In view of the overwhelming possibilities of executional complexity we must establish a very simple correspondence between executional entities, constructional units, and language constructs. This task generates quite an atmosphere of intellectual tension which is fascinating and scientifically fertile. Results in this field may become very important also for other areas of human efforts. In the last few years, some important progress has been made. We are, however, still far away from having come near to the ultimate goal.

There is one general principle in language design which has turned out to be of some help in coming closer to the above goal.

It is the reduction of (unnecessary) execution variability as far as possible. In particular, this may be achieved, e. g., by introducing data types (i. e. sets of values together with specific sets of operations) and typed variables,

by introducing simple and suggestive control structures where "simplicity" is judged, e. g., by inspection of the axiom set for the control structure, and by introducing text structures which visually correspond to execution entities. It is also very beneficial to have at each point in the text interfaces with the state spaces of the possible executions which are as narrow as possible. This may be approximated by introducing a narrow procedure parameter mechanism and by appropriate scope rules, e. g., objects declared in surrounding textual constructs are not inherited automatically. Also, the functions available at a certain program point should always belong to one conceptual level only. It is detrimental to the understandability of, e. g., a higher language program, if program and data addresses may be used without some authorization enforcement. The general tendency of design is, therefore, to restrict automatic accessibility of objects and automatic availability of operations to a minimum. Going beyond this minimum has to be done overtly by declarative means.

Another useful step is the introduction of language elements which constitute suitable abstractions of entities which occur in wide classes of applications. E. g., in higher languages one might think of introducing concepts like classes and processes, where classes may be used by the programmer to build data types which are adequate for the solution of the current problem. The introduction of a language causes programmers, after a while, to think in categories of its data types. Provisions have to be made in order to counter this "fencing-in" effect.

Languages should allow to the programmer a balanced distribution of the problem solution complexity between program and data structures. Often we find complicated programs which use only very simple data structures. In many of these cases

a simpler program which uses more elaborate data structures would constitute a more convincing solution. Concerning this aspect, e. g., Pascal is superior to Algol60.

Constructs in languages may be of different structural levels. Typically we may distinguish, e. g.,

a. Process, message, resource, monitor, class, ...

b. Procedure, block, segment, ...

c. if statement, repetition statement, ...

d. Assignment statement, swap statement, ...

e. Constant, variable, addition operator, ...

f. Address part, operation part, ...

There is a strong correlation between the "level of a language" and its richness with elements of one or more of the above categories.

An analysis of widely used higher languages which are designed for programming at the user-program execution level of typical operating systems shows that these languages consist of elements of the following kinds:

i. Elements which are only available at the user-level of the system or, in addition, at very few neighbouring levels. Examples are file types and operations. Strangely enough, these elements are system-dependent.

ii. Elements which could also be made available at any other system level without any safety, machine-dependence, or overhead problems. Examples are arithmetic and boolean data types and many control statements.

iii. Elements which could be made available at most system levels, however there are safety problems. Examples are pointers and storage allocation functions. There is a certain degree of system dependence.

What is typically missing with higher languages in order to use them universally in operating systems are elements of the above categories a. and some very computer-model dependent functions normally only in category f. as, e. g., load processor or start I/O. However, this drawback is not so serious as another one. Neither these languages nor their known implementations have mechanisms for the controlled use, i. e. for specifying the admissibility, of language elements which do not belong to the above category ii. We shall deal with this problem later on in this paper.

2. Are there Special Requirements for Systems Programming

Often a distinction is made between so-called system and application software. These are relative terms. If designed and programmed properly, one man's application may be the other man's system. For instance, components of a set of application programs which are not specific to the application in question and which are or could be used by several applications may be viewed as system programs. In larger computing systems we have operating systems as minimum system software. This minimum is supplemented by user built system software which appears as an application relative to the software which is

already available. Speaking about system software is always meant with a certain application field on top of it in mind.

In an operating system every program is in uses-relation or in used by-relation to other programs, or hardware components or human observers. Interface "peculiarities" may therefore exist at the lower and upper ends of the hierarchy as it is normally introduced by the above relation.

Interfaces are no problem for the simple-applications designer and programmer. He suppresses the human interface or he does (often subconsciously) some input/output-formatting. The interface of his application program to the system is given implicitly by the conventions of the programming language and its implementation. Concurrency problems do not exist or they are suppressed, in most cases.

The situation is seemingly different for a system designer and programmer. There is, at the lower end, a complicated hardware interface. There is the need to design interfaces between the system components. All these interfaces have to be created by the designer. In addition, concurrency considerations are quite normal.

In the case of more complicated applications we have a situation which is roughly in between the two above extremes.

In order to give an answer to the question asked in the heading of this section we may conclude that in systems programming there is a higher probability of being forced to face and to solve concurrency problems. There is, partly as a consequence, an increased need for higher modularity, precise interfaces, and very thorough structuring considerations. One has to face greater complexity, there is a stronger compulsion for perspicuity and overview. There is no way to avoid

possibly ugly (machine) interfaces. All this indicates that there is no essential difference to "normal" programming. It is very difficult, if not impossible, to determine properties or requirements which are specific to systems but not to applications. Aspects of producing system software turn out to be aspects of producing software in general. All problems mentioned for systems may also occur in applications. In systems programming, however, they are of greater imminence and frequency.

Results from research in systems programming may have positive effects on programming quality and style in general.

3. A Remark on Current Systems Programming Languages

We may distinguish between machine-oriented and higher systems programming languages. The former were primarily contributing to better readability - in comparison with assembler programs - and understandability of coded algorithms. With some rare exceptions, restrictive devices, as mentioned at the end of section 1.4, were not among the primary design objectives. The latter are well suited to programming those components of a system which run on a virtual machine that implements processes and system operations available to them. Adherence to interface rules may be checked at compile time. However, the system kernel must be implemented in another language. The code generated by the higher language translator must properly interface with the code of the kernel.

There is quite an activity in the area of designing new systems programming languages. However, also here ontogeny seems to repeat phylogeny. The actual progress is very slow.

4. How Does the Successful Systems Programmer Survive

Even at the times of prevalence of assembler language there were notable successes of systems. Almost always such a success could be traced back to one or two "good" systems programmers. What was or is the difference between those people and the majority of programmers?

Very often one could find them thinking about their tasks "day and night". They were executing algorithms of the system in their heads - and finding principal or incidental errors long before the first try on an actual machine! By this effort they tried to compensate for the lack of understanding the dynamics of the system from assembler programs. This "method" seems to work up to a certain size of systems. Today most systems tend to be far beyond this limit. The times of this sort of programming star are over. However, there are other habits of these people which deserve attention.

When setting out for a programming task, the good programmer makes always a selection from the repertoire of his tools and language elements. This choice is based on what can be safely used in the case at hand and what could be dangerous. The programmer adheres to a discipline in the choice of tools.

When, for instance, designing interfaces, choosing denotations, deciding on the use of a language element in a particular case, the good programmer will always try to do this with as much uniformity as possible. This helps him and his readers to avoid unnecessary differences in details. The programmer adhers to a discipline of analogous decisions in analogous situations.

On the whole, we observe a defensive attitude, an adherence to self-imposed restrictions. Even with traditional higher languages there is no way to document these rules. They are in the head or in some scribbled notes of the programmer.

The product tends to disintegrate as soon as the man leaves the project.

The key point is that a better language should enforce restrictions to a tolerable degree and at the same time it should allow to declare additional self-imposed restrictions as part of the program text.

5. Design Criteria for an Operating System Programming Language

The set of requirements given below is to be understood as an addition to typical wishes for "good control structures", data structuring methods, data types, etc., as they are found, e. g. in Pascal.

a. System structuring entities which have turned out to be advantageous (e. g. modifications of the Simula class) should be easily implementable or directly be mirrored in the language by suitable abstractions.

b. The language should be applicable to all levels of typical systems. However, by its very own structure the language should not impose any specific layering which is alien to the product.

c. Programs written in the language should clearly exhibit several abstraction levels.

d. Information hiding aspects of program decomposition should be supported.

e. Access rights (and modes) should be clearly visible (and checkable).

f. The use of "level-sensitive" and of unsafe language elements must be shown by declaration or some other clear indication. (This would also be a means for the effective exclusion of the use of incompatible language elements alongside each other.)

g. The machine interface should be made available (and be subject to the mechanism mentioned under f.)

h. The language implementation should accept a "project specification" (e. g. stating access rights and admissibility of language elements) and insure the validity of the specification with respect to the product.

i. As an ideal, all checking should be doable before execution.

j. There should be no larger hidden operations (e. g. type conversions).

k. Optimization should be unnecessary except in case that the target machine provides for extensive local parallelism.

6. Language Mechanisms Assisting in the Construction of Structured Systems

Some conceivable language approaches to the solution of the problem of system structuring are the following.

A special language is written, tailored to one product. The design of the language is closely tied to the design of the system. This is an enormous amount of work for one system, but it enforces a very thorough design phase. By its structure and its elements the language mirrors exactly the system.

The disciplining effect of language design and use is beneficial to the system design. However, there is a disadvantage. The language may have to be changed many times while the system is developed and modified.

In a second approach a base language is used with a very good extension mechanism. By means of the extension mechanism an adaption of the language to the future product is made. The extended language should meet the requirements described in section 5. The state of the art prohibits such an approach.

In a third approach a base language is used with a relatively large set of primitive operations. The base language may consist of the following parts:

a. A machine-independent set of safe elements with simple implementability.

b. A machine-independent set of elements which are unsafe and/or more complicated to implement.

In addition, a machine supplement is used which consists of

c. a set of elements which are characteristic of a certain type of machine hardware and which are believed to be necessary for certain layers or components of program systems on that type of machine. Typically these elements are unsafe. They are needed only at very few places in a system.

Finally, in this approach, there may be a systems supplement which consists of

d. a set of elements which may only be used in certain components of certain systems. The use of such an element in a wrong layer or in the wrong system is unsafe.

In the total language consisting of the base and the two supplements the use of elements of the above categories b., c. and d. must be controlled by a restriction mechanism. It restricts in a declarative and "compile-time" checkable way for certain kinds of program units the set of admissible operations and the access relations to other program and data units.

One advantage of the latter approach is the controlled separation of machine and system dependence (for system dependence this is fully true only if special system properties are not buried in algorithms exclusively written by means of elements of the above category a.) Another advantage is the fully documented and machine-checkable control over unsafe or costly elements. Moreover, there is full control over the selection of layer or component-adequate operations. The designer has to be quite explicit about some of the properties of his building blocks.

Disadvantages are that there is the danger of a large language and that the textual representation of an abstraction level may not always be close to the desirable textual image of that level. In fact, the textual representation is homogeneous accross levels.

Such a restriction meachanism appears technically as a privileging mechanism if one follows the "almost everything is forbidden" - philosophy. It may be understood as a systematic use of the idea which is behind the "privileged instruction" mechanism in hardware. However, checking is by text inspection and not at run-time.

In view of the above considerations future "General Purpose Languages" could achieve a much wider range of useful applicability, if they were structure-supporting in the described sense.

7. Example: The Language System ASTRA

As far as they are of interest here, the main constituents of the ASTRA language system are:

a. A kernel language (KL) which is a higher programming language in the Pascal style plus syntactic mechanisms for

   - declaring access relations,
   - privileging the use of language elements,

   plus "syntactic frames" for certain language objects which may be defined only in connection with a specific target machine.

b. An implementation supplement which completes the language specification for a certain target machine and certain systems. It describes

   - capacity restrictions for operations and operands of the KL
   - permissible parameters for syntactic frames of the KL
   - further language elements (all of them are privileged)
   - libraries the elements of which may be used in connection with certain systems
   - information about hardware properties as, e. g.,

numbers and kinds of processors,
channels, devices,

addressing mechanisms and their use
in target programs,

processor modes, etc.

- the "standard character set".

The implementation supplement contains the elements which were described in section 6 as constituents of a machine supplement and of a systems supplement.

c. A project supplement which completes the language specification for a certain programming project. It describes

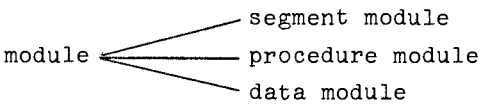
- the names of the program modules and data modules of the projected product,
- the access rights of each program module,
- the privileges of each module.

A project supplement usually undergoes a stepwise refinement process.

In general, an ASTRA implementation will consist of a translator and linking system which also uses the project supplement for checking purposes.

The most important language elements may be sketched as follows:

A program consists of modules:



A program contains at least one segment module or procedure module.

A segment module consists of

- access specifications
- private data
- private procedures
- public procedures
- public segments
  - main segments
  - hardware accessed segments

Private objects are not accessible from outside. Public objects are accessible from outside to the degree stated in access specifications.

A procedure module consists of

- access specifications
- private data
- private procedures
- public procedures

A data module consists of

- access specifications
- public data (including macro definitions)

A procedure may be a

- static procedure (static storage allocation)
- dynamic procedure (programmed storage allocation; the effects of recursive and reentrant procedures may easily be achieved)

A segment is similar to an Algol60 block. However, if the segment is not part of a dynamic procedure, then its local data are allocated statically. A segment consists of

- macro definitions
- type definitions
- constant declarations
- variable declarations
- procedure forward declarations
- procedure declarations
- event forward declarations
- a sequence of statements

A segment is itself a statement.

A segment module may access

- public procedures of procedure modules (by call statements),
- main segments of segment modules (by gomain statements),
- data of data modules (by data references).

A segment module may also have hardware accessed segments which get control via interrupts.

A procedure module may access

- public procedures of procedure modules,
- data of data modules.

A data module may use

- data of data modules.

Details of the language may be found in the reference manual.

A class as used by Brinch Hansen and Hoare maps directly into an ASTRA procedure module with static or recursive procedures.

8. Concluding Remarks

Good compile time checking possibilities make hardware checking to some extent unnecessary. There are cases, however, where a particular hardware provision prevents a less costly software solution. Assume, for example, that a semaphore P-operation is to be implemented and that the expensive transition into system mode is only to be made when it has become clear that the processor has to be assigned to another process. If the update and test operation on the semaphore is only available in system mode, then no cheap solution is possible!

A good textual structure of a system of programs, so written for good intelligibility by human readers, may have the consequence of many expensive environment changes during execution. It might be necessary to have mechanisms (e. g. open procedure techniques) which effectively generate target programs with much less environment transitions while the original program text with its good properties remains untouched.

A positive aspect of such a design is the strong pressure to think about global structuring at an early design stage. In addition, more static checks are possible in the areas of accessibility and of component-sensitive or otherwise unsafe language elements. Doubtlessly, there is an improved project control and a much better overview. This is still true during the maintenance phase of the product.

On the negative side we have the relatively large size of the language and the verbosity of the notation. It is also difficult to develop an access description which allows the adequate degree of refinement.

9. Acknowledgements

The author is indebted to Messrs. C. Correll, P. Gonser, H.-G. Hegering, R. Moll, H. Richter, D. Schneider, and A. Schwald, who have designed and are currently implementing the ASTRA system.

The influence of the discussions among the members of the IFIP Working Group 2.1 on Programming Methodology is also gratefully acknowledged.

10. Literature

ASTRA Reference Manual (to appear)

BRINCH HANSEN, P.
The Programming Language Concurrent Pascal
Information Science, California Institute of Technology, Pasadena, February 1975

CLARK, B.L., HAM, F.J.B.
The Project SUE System Language Reference Manual
Techn. Report CSRG-42, Univ. of Toronto, Sept. 1974

CORRELL, C. et al
A Survey of the Language System ASTRA, a Tool which Aids in Designing, Programming and Controlling System Software.
Leibniz Computing Center, Munich, Report 7503/1
June 1975

ICHBIAH, J.D., RISSEN, J.P., HELLIARD, J.C.
Spécifications de définition de LIS
Doc. No. STG-0-59-T, CII Louveciennes France,
October 1972

KNUTH, D.E., ZAHN, C.T.
Ill-Chosen Use of "Event"
CACM 18 (1975), p. 360

LISKOV, B.
A Note on CLU
MIT, MAC, Comp. Struct. Group Memo 112, Nov. 1974

De REMER, F., KRON, H.
Programming-in-the-Large versus Programming-in-the-Small. Proc. Realiability Conference, Los Angeles, April 1975

HEGERING, H.-G., SCHNEIDER, D., SCHWALD, A., SEEGMUELLER, G.
Systems Programming Elements of the Language ASTRA
Proc. of the European Computing Conference on Software Engineering (EUROCOMP) (to appear), London Sept. 1976

WIRTH, N.
The Programming Language PASCAL (Revised Report)
Bericht der Fachgruppe Computer-Wiss. Nr. 5, Eidg. Technische Hochschule Zürich, Nov. 1972

WULF, W., SHAW, M.
Global Variable Considered Harmful
ACM-Sigplan Notices, Febr. 1973

ZAHN, C.T.
A Control Statement for Natural Top-down Structured Programming
Lecture Notes in Computer Science, Vol. 19, Springer Verlag 1974.

W. A. Wulf

Carnegie-Mellon University, Pittsburgh
USA

Structured Programming in the Basic Layers of an Operating System

STRUCTURED PROGRAMMING IN THE BASIC LAYERS OF AN OPERATING SYSTEM

William A. Wulf

June 1975

Introduction

Professor Bauer suggested the title of these lectures nearly two years ago. At the time, and without much thought, it seemed reasonable enough. As I began to write these notes, however, I began to appreciate both the subtlety and the number of implicit assumptions in it. I'm afraid that in the past few years terms such as 'structured programming' and 'hierarchy' have come to have an almost religious significance. They are intoned with great reverence; faithful adherence to the doctrine is the only true path to forgiveness for past programming sins. I certainly do not mean to imply that there is anything wrong with these terms, but as with many religious concepts, I suspect that many more people believe in them than agree on their meaning. Just consider two of the terms in the title:

> 'Structured Programming': At the very best the title presumes that we all know and agree upon what that is. More deeply, the 'ing' ending presumes that it is the activity of programming that is to be structured; one might presume that the product of this activity, the program, need not be.
>
> 'Basic Layers': Not only does this presume that the programs we are talking about are layered, i.e., hierarchical, but that there is some agreed-upon criteria for determining these layers.

More globally, the whole title suggests, perhaps, that there is something more difficult about structuring the "basic layers of an operating system." I'm not at all sure that's true.

The more I think about the title, the more I like it; anything so apparently innocent, yet so semantically loaded, can only be the product of a great mind. So, if you'll allow me, I'd like to begin these notes with some remarks about the key phrases in the title.

A Personal View of Structure, Programs, and Programming

The term 'structured programming' is certainly popular in the current literature, but I'm not at all sure that there is common agreement on what the term means. Thus, this section is intended to present my own view. I do not necessarily believe this view is more correct, better, or even different from that of others; I include it only to assure some consistency between my intended meaning in the subsequent sections and that which may be perceived by the reader.

By now it is almost a cliche to say that there is a "software crisis." Nearly everyone recognizes that software costs more than hardware, and that the imbalance is projected to increase. Nearly everyone recognizes that software is seldom produced on schedule -- and worse, that the typical software product, costing more and delivered later than originally planned, seldom meets its performance goals; it's bigger, slower, and vastly more error-prone than was originally anticipated. The aggregated cost of a failure to meet performance goals, measured in additional resources, time, and reconstruction of data lost due to an error, may vastly outweigh the initial development cost.

Another component of the software crisis is less commonly recognized, but, in fact, is often more costly than any of those listed above -- namely, the extreme difficulty encountered in attempting to modify an existing program. Even though we frequently believe that we know what we will want a piece of software to do, and will be able to specify it precisely, it seems to be invariably true that after we have it we know better and would like to change it. Examination of the history of almost every major software

system shows that so long as it's used it's being modified! Evolution stops only when the system is dead. The cost of such evolution is almost never measured, but, in at least one case, it exceeds the original development cost by a factor of 100.

Assuming we all agree that there is a software crisis, then, what is the cause of the problem and what are we to do about it? One can find many answers to both questions in the published literature, but to this author the answer to the first, at least, is clear: complexity. Large programs are among the most complex creations of the human intellect; I know of few other artificial objects as complex as, for example, a modern operating system.

Complexity per se is not the culprit, of course; rather, it is our human limitations in dealing with complexity. Dijkstra said it extremely well when he spoke of

"our human inability to do much."

It is our human limitations, our inability to deal simultaneously with all the relations and ramifications of a complex situation, which lie at the root of our software problems. Perhaps it's too obvious to say, but if we really understood a program, we would understand why it is correct or incorrect; we would know why it runs as long as it does, and how it must be modified to improve its performance or incorporate a desirable feature.

Many suggestions have been made to "solve" the software crisis. The most recent of these, and the most promising, are the various programming methodologies: goto-less programming, top-down design, step-wise refinement, modular decomposition, and so on. This collection of methodologies is generally referred to as "structured programming."

Although these methodologies are different, they each have in common that they place restrictions on programs or on the process of creating them. The purpose of these restrictions is the same for all methodologies -- to achieve a match between the apparent complexity of the program and our human ability to deal with that complexity.

The phrase "apparent complexity" in the previous paragraph is a key one. It is intended to suggest that the real complexity of a program might be greater than its apparent complexity. For example, there might be ways of expressing the computation to be performed by a program which hide some of its "grubby details," while simultaneously highlighting its essential, major ideas. By ignoring the details and focusing only on the major ideas, the "apparent complexity" of the program is reduced, and hence should be more amenable to human comprehension. Moreover, once the major ideas of the program are appreciated and the intent of the "grubby details" established, understanding those details becomes a subproblem of exactly the same kind.

Before proceeding I must state another basic premise of these notes, namely, that the property of understandability must be a property of the program text itself!

I do not believe that (large) programs are designed and written. I believe that the initial development of the program is merely the first step in an evolutionary process which will persist until the program is no longer useful. I do not believe that this situation, which is certainly true now, is the result of an imperfect design methodology. Rather, I believe it to be inevitable; the same human limitations which prevent us from dealing with the full complexity of a system also prevent us from anticipating all the ways in which it will be used -- and hence the features they should have.

Given my strong belief in the inevitability of evolutionary modification, I am forced to the conclusion that it is paramount that whatever property of a program makes it understandable must be a property of the program text itself. Of course, I won't receive much argument over the assertion that we ought to make the text of our programs understandable, but the implications of the assertion are often not fully appreciated.

Let's consider, for example, the proposals for "top-down-design" or "step-wise refinement." These are both names for a methodology for writing programs. The methodology involves starting with a high-level, abstract program to perform some task. By definition, the level of abstraction is chosen such that the program is short,

understandable, and "obviously correct." Usually this level will be such that the resulting program cannot be expressed directly in one of the extant programming languages. Hence the methodology advocates a deliberate expansion of the program's primitive concepts into "lower-level" ones; in general this process may involve several levels of abstraction, but finally results in something which can be expressed in executable form.

The strength of the methodology, of course, is that, if done properly, the degree of complexity at each step of the process will be within our human ability to cope with it. In effect, it is a technique for decomposing a complex task into subtasks, each of which is manageable in isolation. Unfortunately, practiced in isolation th methodolog has two weaknesses.

First, and most important in the present context, the various abstraction and refinement steps are not necessarily present in the final program. Thus the methodology serves well during initial development, but fails to help the future program modifier. This is not to say that programs developed in this way are as hard to understand as ones developed in a more ad hoc fashion. On the contrary, they are usually much more understandable. It is a matter of degree, and this author believes that the methodology alone is insufficient.

Second, blindly practiced, a top-down methodology ignores an essential aspect of the engineering of program design -- namely, the search for commonality. A pure top-down methodology results in a tree of design decisions of abstractions and their abstractions. In such a design each line of code would be traceable through a unique set of decisions to the root program. In almost all cases this is undersirable. If, for example, abstractions exist for queues and sets, and both are implemented in terms of linked lists, it is probably preferable to use a common package of list manipulation routines. We shall examine the issue of commonality in greater detail later.

Let me briefly recap the discussion to this point. It has two essential components:

(1) The "software crisis" is the result of our human limitations in dealing with complexity.

(2) To "solve" the problem we must reduce the "apparent complexity" of programs, and this reduction must occur in the program text.

Under this view it is the nature of the program, rather than the methodology used to create it, which is central. A methodology is useful precisely to the extent that it leads to understandable programs, but if a program is understandable, the methodology used to create it is irrelevant. For this reason, I much prefer the term "structured programs" to "structured programming."

The next natural question is, "ok -- what makes a program understandable?" Unfortunately, the question is similar to "what makes a mathematical proof elegant," or "what makes a painting aesthetically pleasing." The properties which make something understandable, or elegant, or beautiful are related to psychological factors as well as the training, intelligence, and perhaps even taste of the beholder. It is unlikely that we will find a precise answer.

On the other hand, we do know some things which are hard to understand, e.g., "bowl-of-spaghetti" flow of control, and we can avoid these. More importantly, we know some properties which seem to be common to understandable programs, and we can strive to incorporate these. These latter properties include: minimise the assumptions between portions of a program, keep each portion (reasonably) small, and so on.

Finally, and most important of all, we know something about the way humans have traditionally dealt with understanding complex problems -- in fact, the way we are trained to deal with them -- and we can try to mold the expression of a program so that it facilitates these techniques. Among the most powerful of these techniques are those of abstraction and structuring. In practice these two notions are often intertwined, but by abstraction we mean the process of ignoring detail and dealing instead with a

generalized, idealized model of a complex entity; by structure we mean the relation between the parts of a whole.

Focus on the word "structure" for a moment: the relation between the parts of a whole. Both the notion of parts and the relation between them are important. Every program has some sort of structure -- some parts and some relation between them -- even if it is the vacuous structure consisting of a single part and the empty relation. If, however, each of the parts of the program is conceptually simple in isolation and the relation between them is also simple, then the whole program will be easy to comprehend.

But notice that if either the parts or the relation are difficult to understand, the whole will also be incomprehensible. Thus the process of constructing a well-structured program involves the choice of both of these, and neither can be considered less important than the other. This is the reason that simplistic rules such as 'avoid goto's' or 'subroutine-ize' do not necessarily lead to well-structured, understandable programs.

Abstraction may play a role in making either the parts or the relation between them simpler, but the more common case is probably related to the "parts." Functional abstraction via procedures and data abstractions via type or mode definitions are both examples of the use of abstraction to reduce the apparent complexity of some portion of a program.

At this point it is probably worth reviewing the tools we have available for structuring and abstracting programs -- and here we should include both linguistic and conceptual tools. Structuring tools are generally composition mechanisms of various flavors. For example, in the control domain they include

- compound statements, blocks, and procedures
- conditional and selective execution
- looping constructs of various kinds.

In the data domain they include arrays, records, lists, sequences, sets, unions, queues, and so on.

Abstraction tools are less often found in contemporary programming languages. Except for the class mechanism of Simula, and to some extent the type (mode) definitions in languages such as PASCAL and Algol '68, only procedures, and sometimes macro's, have been available. Conceptually, however, the notions of type, function, process, as well as all of those mathematics, are available.

Before leaving this personal view of what is usually called "structured programming," I would like to be sure to draw a distinction which I feel is often missed: namely, the difference between an abstraction and an instance of it.

An abstraction is an idealized model of a class of objects. An instance is a particular object in the class. "Computer" is an abstraction; the particular computer on which this text is being prepared is an instance of the abstraction. The distinction between an abstraction and an instance is trivially obvious, but we will rely on it heavily in the next section.

Comments on "Hierarchy"

As noted in the preceeding paragraphs, the structure of a program consists of a division into parts with some relation between them. In this section we shall consider several possible divisions and relations which seem especially appropriate for operating systems.

A particularly simple kind of relation, and hence one which is attractive as an aid to understanding, is a hierarchy. As noted by Parnas [P74], a system composed of parts $\alpha 1,\ldots,\alpha n$ may be said to be hierarchical with respect to a relation, or predicate $R(\alpha i,\alpha j)$, if we may define a set of levels by

(1) Level 0 is the set of parts, α, such that there does not exist a part β such that $R(\alpha,\beta)$.

(2) Level i is the set of parts, α, such that

(a) there exists a β on level i-1 such that $R(\alpha,\beta)$

(b) $R(\alpha,x)$implies that x is on level i-1 or lower.

Any relation, R, which satisfies these criteria may be represented by a directed graph without cycles.

As Parnas also points out, the use of the word 'hierarchy' has not been consistent in the literature, but the above definition seems to be the simplist which covers them all. Specifically, the difference between uses seems to be primarily one of the relation R. We do not wish to repeat Parnas' full explanation, but he notes in particular at least the following relations have been used:*

THE SYSTEM:

The relation "uses," in the sense of a subroutine call, as well as the relation "gives work to," in the sense of process cooperation, are defined. The THE system is hierarchical with respect to both these relations, and, as a matter of fact, they define the same hierarchy.

RC4000:

The RC4000 system imposed a hierarchy on the ownership and allocation of resources among processes. The same hierarchy was

---

*Note that authors have not generally stated the relation; Parnas is inferring it from context. See [P74] for a more detailed discussion of these relations.

imposed on protection, that is, parent processes were "more privledged than" their descendents.

MULTICS:

The "ring" structure of MULTICS imposes a hierarchy upon protection; that is, the relation is "may be accessed by."

Many other examples of such relations could be extracted from other papers, but, hopefully, the point has been made that the phrase "system X is hierarchical" is meaningless until the relation by which its parts are ordered is defined. Two supposedly "hierarchical" systems may have very different structures.

There are also many questions we might ask about the general notion of hierarchy, for example:

(a) Given that there might be more than one relation between the parts, is it desirable, indeed is it feasible, that precisely the same hierarchy should be imposed with respect to each relation?

(b) Are all non-hierarchical structures, i.e., cyclic ones, "bad"? Certainly at the level of programming languages, loops are common, and certain styles of looping are considered "well-structured." Even though we could replace loops by recursion, and hence a hierarchy could be imposed, we seldom do. We do this not just because recursion is more expensive, but because the iterative form is often clearer.

The division of a program into "parts" and the choice of the relation between those parts are not independent decisions, and comprehensibility of both are required in order to achieve comprehensibility of the whole. It is conceivable that an a priori choice of the relation could adversely affect the choice of the 'parts.'

(c) Given a relation, what parts should be "above" others? That is, is there an obvious preferance to the order between, say, the file system and the virtual memory system?

(d) Is there any advantage to imposing an even stricter definition of hierarchy than that given by Parnas? For example, by altering (2b) to read:

$R(\alpha,x)$ implies that x is (only) on level i-1.

None of these questions can be answered in isolation. Hierarchy is not, or at least should not be, an end in itself. Rather, hierarchy is one particular structure which might exist between the various parts of a system. Although it is frequently a useful one, and frequently a comprehensible one, we must avoid the trap of believing it's the only one, or even the "right" one, under all circumstances. Our objective must be clarity and correctness rather than any particular structural relation, no matter how fashionable that relation may be.

Lest the reader be mislead by inferring either too much or too little from the previous remarks, let me clearly state my personal opinion on hierarchies: A hierarchical relation is simple and intuitively appealing; it is often the "right" one. However: (1) they are not <u>always</u> "right," and may in fact lead to obfuscation in a few cases, and (2) attempting to impose the <u>same</u> hierarchy with respect to the several relations which exist between the parts of a large system is almost always <u>wrong</u>! Later I will also try to amplify these remarks in connection with the relations which may exist between abstractions, and those which exist between instances of these abstractions.

Layers of Operating Systems

One conventional view of an operating system is that it provides a "virtual machine" on which user programs execute. This virtual machine is often similar to the physical machine on which it is implemented, but some of the unpleasant features of the physical

machine have been eliminated and some useful features have been added. Among the unpleasant features often eliminated by the virtual machine are interrupts, device-dependent i/o, the size of the physical memory, etc. The features added usually include files, directories, and so on.

The virtual machine view ignores the issues related to the fact that the physical machine may be being used simultaneously by several parties -- and that there is an associated problem of resource allocation and adjudication of competing demands. Thus, an alternate veiw of operating systems is that they provide this allocation function and, in addition, do so both "fairly" and so as to maximally utilize the physical resources of the machine.

Neither of the above views explicitly accounts for the possibility that two or more of the users of the system may wish to share information and/or resources. So long as there is no sharing, each virtual machine may be viewed as an isolated entity and all resource rquests are "competing"; as soon as sharing is permitted, neither of these assumptions is valid. Thus a third view of an operating system is that it provides protection in the presence of controlled sharing.

In fact, of course, no one of these views is wholly appropriate and there may be many others; it is not my intent to suggest that any one deserves more emphasis than another. However, the relative emphasis between the three will affect the structure of a system, and we shall see examples of this in a moment.

In the United States there is a saying: "what's good for the goose is good for the gander." Instantiating this abstraction in the current context we have "if <virtual machines, resource allocation, protected sharing> are good for the user, then the same thing(s) are good for the operating system." That is, the same arguments which lead us to believe that the user would prefer to write programs in an environment which is more hospitable than that provided by the basic hardware also apply to most parts of the operating system itself. For example, many parts of the operating system operate asynchronously on behalf of various users; it is conceptually simpler, hence has "better

structure," if these portions may be thought of as executing on independent "virtual processors."

This discussion suggests a natural relation upon which a hierarchy might be based, namely, "uses" or "depends upon." This is the hierarchy of the THE system (which is shown below).

Level	Abstraction Provided
5	Operator (not part of the system)
4	Independent User Programs
3	Buffered input/output
2	Message Console
1	Segment Controller (virtual memory)
0	Process/Processor Multiplexing

Thus in the THE system, for example, everything above level 0 uses the abstraction of a virtual processor -- i.e., it may be an independent process. Above level 1 everything may have a private 'virtual memory.' And so on.

It is worthy of special note that each level in this hierarchy both provides facilities and hides certain (possibly unpleasant) properties of the basic machine.

Level	Hidden
5	--
4	--
3	Timing dependencies and other idiosyncracies of the input/output hardware.
2	The fact that there is a fixed number of operator consoles; several logical message streams are multiplexed onto these.
1	The fixed size of memory, physical core locations, and whether a particular word is in core or backing store.
0	The number and relative speed of the processors.

A more elaborate layering may be found in the SRI Security Kernel [R75] which also uses the 'uses' or 'depends upon' relation, and which, like the THE system, imposes a linear order on the components. I won't reproduce or discuss this system here; for the moment it is sufficient to note that the 'basic' or 'lowest' levels of these systems are those which abstract from the physical resources of the machine.

The THE system provides a nice example of the problem, or intellectual challenge if you prefer, of designing a hierarchical system. In particular, consider levels 1, 2, and 3, and notice that all three levels perform input/output operations. It would be desirable if all i/o were performed by a single level -- thus one might be tempted to suggest that levels 1 and 3 should be interchanged. If this were done, or so it seems, both the segment controller (level 2) and the message handler could perform "virtual i/o." Unfortunately the input/output level "needs" the message controller -- for example, to

inform the operator of unrecoverable device failures and, presumably, to request tapes to be mounted, etc. Also, inverting layers 1 and 3 would prevent the input/output level from running in a virtual memory, thus buffer allocation and so on would have to be done in the physical address space, and at least some sort of physical address space allocation would have to be available to the i/o level (as level 0.5?). Thus there is a certain amount of "chicken and egg" phenomena in choosing levels.

A related problem is discussed by Saxena and Bredt [SB75]. The THE system was designed to support a small, fixed number of concurrent processess. But suppose that a large number were to be supported; in fact, suppose that the number were large enough that their state information could not all be held simultaneously in primary memory. Saxena and Bredt suggest a solution in which the four lowest levels are:

Level	Abstraction
4	Virtual memory management for a large and variable number of processes.
3	Scheduling and synchronization of a large and variable number of processes.
2	Virtual memory management for a small and fixed number of processes.
1	Scheduling and synchronization for a small and fixed number of processes.

In this scheme levels 1 and 2 correspond to levels 0 and 1 of the THE system; specifically, the state information for the processes handled by these levels are assumed to be core resident. They are "fixed processes" in the sense that they persist for the life of the system. The levels 3 and 4 are very similar except that they multiplex the actual user processes, including address spaces, onto (some of) the fixed

processes provided by the lower levels. Again i/o is not treated as an explicit level and the i/o performed by the virtual memory levels is physical. Detailed programs are given for each of the levels in PASCAL.

I had three motives for introducing the Saxena and Bredt decomposition: first, it is simply another example of layering; second, it illustrates again the "chicken and egg" phenomenon; third, it illustrates what I percieve to be a failure to recognize the important distinction between an abstraction and an instantiation of that abstraction.

In the THE system, and most of those described as being hierarchical, there is only one instance of each of the named abstractions; thus the issue doesn't arise. In the Saxena and Bredt design we see two instances of each of the abstractions "scheduler" and "virtual memory." The two schedulers in theis scheme are in fact identical except for naming; the virtual memory systems are only slightly different, and some reorganization could also make them identical, so lets treat them as though they were for the moment. ·

Now we see that the (modified) Saxena and Bredt scheme is hierarchical with respect to the relation "uses an instance of" but not hierarchical with respect to the relation "uses the abstraction." The abstraction "scheduler" uses the abstraction "virtual memory manager" and vice versa, thus these two may not be ordered with respect to each other. On the other hand, the hierarchy shown earlier still obtains between the instantiations of the abstractions.

Given the distinction between abstraction and instance, we might venture an alternative structure for a THE-like system. (I do this with great tripidation given that EWD is the audience, but we Americans tend to be brash.) Let's hypothesize five basic abstractions:

(a) scheduler: process scheduling and multiplexing

(b) alloc: storage allocation from an "address space"

(c) i/o: buffered i/o

(d) console: multiplexed communication with the operator

(e) virtual memory: the THE 'segment controller'

Now we might propose that the lower levels, at least, of our hypothetical system appear as:

Level	Abstraction Instance
...	
7	io (other user devices)
6	alloc (from a virtual memory)
5	virtual memory
4	io (drum)
3	console
2	io (teleprinter)
1	alloc (from physical memory)
0	scheduler

The intent here, of course, is that the various instances of an abstraction invoke precisely the same code, only the context in which they operate are different. Thus, for example, the two instances of "alloc" are different only in that one executes in a

"virtual" address space while the other operates in the physical address space of the machine. Similarly, the instances of "io" on levels 7, 4, and 2 differ in that the first, when it uses alloc (on level 6) will obtain buffers from a private virtual memory while those on levels 2 and 4 will obtain buffers from physical memory.

I hope the advantage of considering the relation between instances rather than abstractions is obvious; we would like to be able to characterize an abstraction and prove things about its properties -- usually relative to properties of other abstractions -- and then be able to instantiate that abstraction in more than one place, and, in particular, at more than one 'level.'

The "Basic" Layers - Some Assumptions

For the remainder of these notes I am going to make an assumption which need not be true -- namely, that the 'basic layers' are those which abstract from the major physical resources of the machine -- processors, i/o, and memory. It seems plausible to make such an assumptrion only because we are not talking about any particular operating system -- and, generally speaking, these resources are among the first which one wishes to abstract from. But it need not be so! Specifically, an operating system whose primary concern is protection in the presence of sharing, might well place the protection mechanism, or some portion of it, on the "lowest" levels. This is true, for example, of the SRI security kernel design [R75]. Thus the choice of what belongs on the 'lowest' levels actually depends upon the highest level image of the system presented to the user.

I shall also presume that the system to be constructed at the highest level is a "conventional" multi-access one. Thus the lowest levels should support an arbitrary, but moderate, number of processes; memory may be shared, but the policy which controls sharing is not the concern of the lowest level mechanism, and so on. Similarly, I presume the underlying hardware is "conventional" in the sense that there is no especially elaborate hardware mechanism (e.g., beyond page mapping, interrupts, and some indivisible instruction of the "test-and-set" flavor) to support these goals.

Further, I shall presume the relation chosen to establish the hierarchy is "uses an instance of" -- that is, we shall define various abstractions and instantiate them for various purposes. We shall be concerned about the relation between the uses of the instantiations, but not necessarily about the corresponding relation between the abstractions themselves.

A note on linguistic assumptions: In the sequel I shall use an invented language to express algorithms -- the language itself is somewhat a hybrid of Simula and PASCAL. Hopefully, the meaning of most constructs will be clear from context, but I will try to make clarifying comments where appropriate. The reader should note that this language will not contain relatively high-level notions such as that of a "monitor" [H74] -- it is precisely such constructs we are attempting to define at the lowest levels, where it would be unfair to use them.

The examples used are taken from the Hydra system [W74], although the details have been changed to improve the exposition. From a gross viewpoint the lowest level provides scheduling and synchronization of a variable number of incore processes (the decision of which processes are to be made core-resident, and hence which are to be scheduled, is made at a much higher level; in fact in Hydra this decision is made by user-level programs). The next higher level supports i/o to the swapping media -- drums and disks. The highest level we shall consider supports a virtual memory consisting of several "pages"; pages may be shared among several processes. At a finer level of detail, each of the levels will be further decomposed into several abstractions.

Let us consider for a moment the 'special problems' of structuring these basic levels. In at least one sense there aren't any. The problem is just like that of structuring any other program -- we seek a decomposition which will consist of a number of abstract parts and a relation between these parts which will be clear, understandable, and obviously correct.

In another sense, structuring the basic levels of an operating system shares problems with structuring the lowest levels of other kinds of programs. Specifically, the very lowest level abstraction is fixed -- its the one provided to us by the language and/or machine. Certain things are reasonable (i.e., efficient) on this lowest level and others are not. Thus in designing our abstractions we must keep in mind both the abstraction to be provided as well as the basic tools with which we may fabricate the implementation of the abstraction.

Finally, there art two classes of problems which are not generally faced when structuring a normal user program:

(1) **The problems of parallelism.** A great deal has been written about these problems, and the need for mutual exclusion and deadlock prevention, if not the best means for achieving them, is well understood.

(2) **Policies.** The presence of multiple, simultaneous users implies the need for one or more policies for adjudicating between their competing demands for the finite resources. Both the choice of the policies to be used and the placement of their implementation in the hierarchy may significantly impact both the performance of the system as a whole (i.e., throughput) and the performance percieved by individual users. Some of the policies are often implemented in the lowest levels -- e.g., scheduling -- where there is, or should be, relatively little information about the image presented to the user. Thus, there arises an additional constraint on the structure - namely, that policies should, if possible, be deferred to the higher levels where more knowledge is available. Achieving this is sometimes referred to as "policy/mechanism separation." That is, separating policy decisions from the mechanisms which may be used to implement any of several policies.

A High Level Model of a Hydra-like System

As the reader may have detected, I am having some difficulty getting down to the primary topic of these notes -- structuring the basic levels. My first problem was one of terminology. My current problem is that I would like to make the following remarks concrete rather than "motherhood", and the best way I know to do that is by example. To use an example I need to define the "basic levels" at least in general terms; you need to know what abstraction they are intended to support. But to do that I must say a bit more about the levels above them -- and you see what that traps me into. The example I want to use is based on some of the ideas in Hydra [W74], and so I must tell you something (not too much hopefully) about the high level of Hydra -- at least as it relates to processes and paging. To avoid telling you too much, I'm afraid I shall have to lie a bit in one or two places; I hope you'll forgive me.

The basic objects we shall be concerned with are a process and a page. A process is just what you would expect; it is the smallest unit which may be scheduled for independent execution. A page is a logical unit of storage in which there may be either program or data. A process may have an arbitrary number of pages in its address space,* but only a fixed and finite number of them may be in primary memory at any moment; the set of pages associated with a process which must be in primary memory when the process is running is called its cps (core page set). The membership of the cps is explicity determined by the process. That is, operations are provided for including and/or deleting pages from the cps and the user must perform these operations by explicit calls on Hydra; the system does not attemp t automatically deduce the cps from the behavior of the program. Reference to a page which is not in the cps is prohibited.

---

*"address space" is not a Hydra concept; I use the term here only to suggest the analogous notion in other systems.

We may think of the cps-manipulation operations available to the user as:

cps-include (p:page) which inserts the specified page into the cps of the process executing the operation.

cps-delete (p:page) which correspondingly deletes the specified page.

Also of concern to us is the fact that medium-to-long term scheduling policies, as well as parameters which control short-term policies, are set by user-level software, not by the basic levels. This is realized in the following way.*

(1) Each process is in one of the following states:

S1: its cps is not in primary memory and it may not be scheduled by the basic levels.

S2: its cps is in primary memory, but it may not be scheduled by the basic levels.

S3: its cps is in primary memory, and it is permissible for the basic levels to schedule it.

(2) The short-term behavior of a process in state S3 is determined by the basic levels but is controlled by the following parameters:

priority: scheduling priority

---

*Again, this characterization is not completely faithful to Hydra.

**In the actual implementation of Hydra there may be a number of policy modules.

slice: size of a time slice

number of slices: a number of time slices

cpsmax: the maximum number of pages the process is permitted to have in its cps.

(3) The parameters in (2), and to some extent the state in (1), are determined by a distinguished process** called a policy module (or PM). We may think of a PM as capable of performing the following operations on a process:

setparm(P:process,priority,slice,numslice,cpsmax;integer) which sets the parameters of the process P.

makpresent(P:process) which makes the cps of a process resident in primary memory.

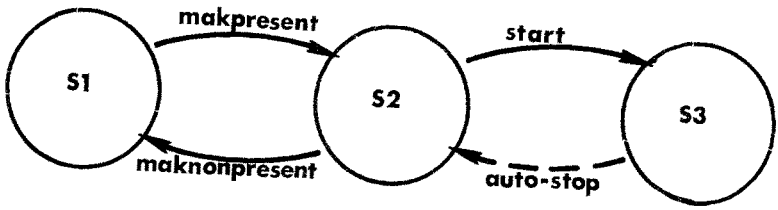
start(P:process)which allows the process to be scheduled by the basic layers.

maknonpresent(P:process)which allows the basic layers to remove the cps of the process from primary memory.

In addition, the basic layers will automatically perform a "stop" under certain circumstances. The PM may determine which processes have been so stopped by performing

wait-for-stop(P:process) which will suspend the PM until some process has been stopped and then return the process in P.

The relation between the states of a process and the operations which may be performed by a PM is characterized by the following state-transition diagram (the dotted lines in this diagram denote the "automatic stop" performed by the basic layers).


S1
makpresent
S2
start
S3
maknonpresent
auto-stop

Transitions not shown are illegal.

The "automatic stops" alluded to above arise when a process exceeds one of the parameters which the PM has set to control its resource allocation. In the simplified model we have presented there are two cases of this:

(1) **processor allocation:** the process has exceeded the time quantum allocated to it, namely, slice * number-of-slices.

(2) **memory allocation:** the process has attempted, by performing a cps-include, to increase the size of its cps beyond that allowed by cpsmax.

Thus the 'high-level' model of Hydra, the one we must be cognizant of as we discuss the lowest levels, is that the medium-to-long term policies governing the behavior of processes is controlled by a user level program -- the PM. The PM also influences short-term policies by appropriately setting the parameters of a process. The mechanism used to effect this control is the set of operations informally described above.

The Lowest Level of Hydra -- Machine Assumptions

At the other extreme from the high-level model presented above is the hardware architecture of the machine on which it is implemented. Since this also impacts the structuring issues, we must make a few comments about its properties (but no details).

First, C.mmp (the machine which Hydra is implemented on) is a multiprocessor. It may have as many as 16 processors, although at the moment it has 5.

Second, there is an inter-processor interrupt mechanism which permits one processor to interrupt any subset of the others at any of several priority levels. We shall denote this mechanism by an operation.

IPI (level,mask)

where

IPI	is short for "inter-processor interrupt"
level	specifies the priority level of the interrupt
mask	specifies the processors to be interrupted; processors are correlated with bit positions in the mask so that if bit k is set in the mask, processor k will be interrupted.

Third, it is possible to set the processor into a state in which it is "blind" to interrupts. We shall denote this linguistically by the construct

blind (<statement list>).

Fourth, there are two machine instructions which indivisibly test and alter a memory location. The instructions are

INC x	which increments the contents of location x by one and leaves the "condition codes" set to reflect the sign of the new value.
DEC x	which decrements the value of location x by one and leaves the "condition codes" set to reflect the sign of the new value.

Fifth, each processor has a small amount of private memory. Thus access of the "same address" on each of the machines may yield a different value. For example, one such location is called "ME" and contains a unique identifying number for each processor.

A Bottom-Up-Presentation

It has always been a mystery to me whether top-down or bottom-up presentations have the greatest pedagogical merit. The choice is between explaining the highest level, most abstract, concepts in terms of as yet undefined primitives if one goes top-down, or building a vocabulary without letting the reader know where he's being lead if one goes bottom up. Just for a change I think I'll try it bottom up. Just keep in mind that the ultimate target is the set of process and cps mainpulation operations presented previously.

(1) **Locks**

We shall begin by designing several abstractions of general utility; the first of these is a primitive syncronization mechanism called a lock

A lock may be thought of as having two states 'locked' and 'unlocked.' Two operations are provided on locks, LOCK and UNLOCK; in combination these are used to achieve mutually exclusive access to the data structure(s) protected by the lock. The effect of these operations may be defined by:

Operation	State	Action
LOCK	unlocked	Change state to 'locked,' allow the process to proceed.
LOCK	locked	Block the processor.
UNLOCK	unlocked	Illegal!
UNLOCK	locked	If no other processor is blocked on this lock, set the state to unlocked; otherwise, unblock one processor.

In reality the effect is achieved by a more cooperative interaction. The lock is actually a simple data structure and the two operations operate on instances of such a structure. If a process attempts to LOCK a lock that is already locked, the processor running the process (1) sets a bit in the lock structure to indicate that it is waiting on the lock, (2) disables all but a specific processor interrupt, and (3) enters a physical wait state. If the UNLOCK procedure finds that some processor is waiting on the lock it is unlocking, it will send a special interprocessor interrupt to all such processors. The interrupted processors then try for the lock again. In the following example, as later, we first list some relevant global variables and data structures presumed in the procedures themselves.

memask — A variable in the local memory of each processor. Bit N of memask is 1 iff the processor number corresponding to the local memory is N.

The implementation of the abstraction itself may be described by a Simula-like class:

```
class   lock (inval:integer) =
    begin
    rep   lkv,sublk:integer,lkmask:word;
    initialize   begin   sublk←lkmask←0;lkv←ival end;

    proc   LOCK(l:lock) =
           if( DEC l.lkv )<0 then
                  begin
                  l.lkmask←l.lkmask v memask;
                  do waitforlockinterrupt while ( DEC l.lkv )<0;
                  l.lkmask←l.lkmask ∧ not(memask);
                  end;

    proc   UNLOCK(l:lock) =
           if   ( INC l.lkv )≤0 then
                  begin
```

```
        l.sublk←1;
        while l.lkmask = 0 do;
        IPI(lockinterrupt, l.lkmask);
        end;

    end;
```

First let's dispense with some of the notational details. The class declaration is defining a new abstract data type, "lock," and we will be able to declare instances of these later. The only two operations defined on this type will be the proc's LOCK and UNLOCK. Each time that a lock is declared the actual storage allocated will be two integers (lkv and sublk) and a "word" (lkmask); a "word" is boolean vector of convenient length to which bit-wise operations may be applied (i.e., it's a machine "word"). The initialize portion of the class is executed at the time an instantiation of the class is created. Note that the lock is initialized from a parameter of the class.

The correctness of the machine language versions of these primitives has been proven [Ari73]; the details of this proof are beyond the scope of this paper, but it is instructive to reason about them in an informal way. Three assumptions are made: first, that the primitives are executed in a blind, that is non-interruptable, state; second, that the calls on the primitives are properly paired (each call on lock implies a subsequent call on unlock for the same lock structure); and third, that only these procedures manipulate lock structures.

The interesting case is when the lock is locked (lkv≤0) and another processor attempts to perform a lock operation. It will decrement the lkvk field, with the indivisible "subtract from memory' instruction, and observe that the result is negative. Because the processor is non-interruptable (assumed above), after some finite time it will set a bit in the 'lkmask' field corresponding to the processor on which the lock happens to be executing. It will then execute the 'waitforlockinterrupt' which involves enabling one interrupt, the lock interrupt, and then putting the processor into the idle state.

Independently, on some other processor, the unlock operation will be performed on the same lock structure. This is guaranteed to happen eventually by the second assumption above. The unlock operation increments the 'lkv' field, also indivisibly, and will observe the result to be non-positive (otherwise the first process would not have blocked). In such a case it sets 'sublk' to one and then attempts to generate an interrupt to all blocked processors.

There are two important observations at this point. First, this is the only situation in which 'sublock'>0; simple inspection shows this. Second, it's possible that 'lkmask'=0. Suppose the processor executing the lock operation has decremented 'lkv,' to -1 but has not yet set its bit into 'lkmask'; then 'lockmask' will be zero. By the previous reasoning, however, the locking processor will set its bit after some finite time. Thus the loop in the UNLOCK procedure

```
while l.lkmask=0 do;
```

is guaranteed to terminate. It will then execute the IPI to interrupt at least one, but possibly several, of the blocked processors.

Once the interprocessor interrupt is generated, one or more processors which were blocked will awaken from the 'waitforlockinterrupt' procedure into the loop:

```
do waitforlockinterrupt while ( DEC l.lkv )<0;
```

Decrementing the sublk field and observing the sign of the result is an indivisible operation. Thus precisely one of the processors awakened by the interrupt will observe a zero value and will exit the loop. The remaining will loop back and wait for the next UNLOCK operation to signal completion.

The purists will recognize the potential for 'individual starvation' in this scheme. That is, given enough contention for a single lock, it's theoretically possible for one of the processors to remain blocked forever -- to always loose the race for the sublock.

The possibility is infintesimal, however, and we choose to permit its possibility rather than unnecessarily complicate the mechanism.

Finally, in order to clarify some of the following we shall presume a macro facility; in the current case we define

```
macro   CRITICAL(l:lock,stmt) = begin LOCK(l); stmt; UNLOCK(l) end;
```

Thus the statement, 'stmt', is made invisible with respect to the lock "l."

(2) **Lists**

The next abstraction we should like to define is that of a list; in this particular case a doubly-linked circular list. The notion of a list is again defined by a class:

```
class   list =
        begin
        rep   next,prev:ref list;
        initialize   next←prev←@self;

        proc   linkafter(old,new:list) =
               begin
               new.next←old.next; new.prev←@old;
               old.next.prev←@new; old.next←@new;
               end;

        proc   delink(X:list) returns (y:ref(list)) =
               begin
               X.prev.next←X.next; X.next.prev←X.prev;
               y←X.next←X.prev←@X;
               end;
```

```
        end;
```

Hopefully this should not require explanation.

(4) **Freespace**

The next abstraction we shall need is that of a free-stroage allocation. I won't present the allocator in detail for two reasons. First the particular allocation strategy is of little interest to this discussion. Second, there is a linguistic problem; the notation I have chosen to use is strongly typed while a storage allocator is by its nature concerned with untyped storage. However, we may think of our allocator as having the general form

```
class   storage(N:integer) =
        begin
        rep   fd:lock(1), fh:list, fs:array[1:N] of word;
        comment   initialization depends upon storage allocation algorithm;

        proc   get(n:integer) returns (address) =
               CRITICAL(fl,begin...end);

        proc   free(a:address) =
               CRITICAL(fl,begin...end);

        end;
```

Here we see our first example of the use of one (actually two) abstraction in the implementation of another. The storage allocation uses a lock to insure mutually exclusive access to the pool of free storage and a list to keep track of the blocks of storage which are free.

To mesh the strongly-typed language with the untyped allocator we shall assume two magic operators

new(T(<parameters>)), where T is a type (class name) calls the allocator for enough storage for the type and then converts the "address" returned from the allocation into a "ref T."

free(t:ref T) calls the "free" routine in the allocator after having converted the ref T into an address.

(3) **Queues**

The next abstraction we shall define is that of a queue -- a collection of objects for which there is a defined insertion and deletion disipline. We shall be interested in several potential disciplines: fifo, priority-ordered, and reverse-priority ordered. The nature of the discipline for a particular queue will be "hidden" in the implementation of the queue.

The abstraction "queue" is again defined by a class. It is my intention that the abstraction be defined so that aribtrary types of objects may be placed in a queue; again this causes a bit of a linguistic problem* in that the queue definition must presume the existence of certain field names in these objects -- extant languages won't allow me to express this very well. We will plunge on undaunted.

The queue definition consists of the following:

---

*There are solutions to each of the linguistic problems mentioned; I simply don't want to be distracted by them.

```
class   queue (C: class, discipline:integer) =
     begin
     rep   head:list, d:integer;
     initialize   d←discipline;

     class   qelement(C:class) =
          begin
          rep   l:list, r:ref(C);
          proc   next = l.next;
          proc   pred = l.pred;
          end;

     proc   enq (Q:queue, x:C) =
          begin
          local n,t:ref(qelement(C));
          n←new(qelement(C));
          n.r←@x;
          case   Q.d of
               begin
               1:   linkafter (Q.head, n.l);
               2:   begin
                    t←Q.head.next;
                    while   t≠Q.head and x.priority≤t.r.priority
                              do   t←t.next;
                    linkafter(t.pred, n.l)
                    end;
               otherwise:   begin
                    t←Q.head.pred;
                    while   t≠Q.head and x.priority≥t.r.priority
                              do t←t.pred;
                    linkafter(t.next, n.l);
                    end;
               end;
```

```
proc   deq(Q:queue) returns (t:ref C) =
       begin
       if Q.head.next=Q.head then ERROR;
       t←Q.head.next.r;
       free(delink(Q.head.next));
       end;

proc   first(Q) returns (x:ref C) = x←Q.head.next.r;

proc   req(Q1,Q2:queue) = enq(Q1,deq(Q2));

end;
```

Examination of this abstraction will show that the instantiation parameter, "discipline," may have the values 1,2, or "something else." By introducing the manifest constants

```
constant   FIFO=1, PRIOR=2, REVPRIOR=3;
```

we have names which suggest the meaning of the various values of this parameter. Thus the declaration

```
own   X:queue(integer,FIFO)
```

will create a FIFO queue of (refernces to) integers. Subsequent use of the functions

```
enq(X,-) and deq(X,-)
```

will preserve the FIFO order of the queue X.

(4) **Process Context**

The representation of the machine "state" of a process generally consists of the contents of the registers and perhaps the contents of the mapping hardware which support the virtual memory. In any case, the precise representation is highly machine specific, and we shall not attempt to "fake" one here. Let it suffice to say that there is a conceptual class which is capable of saving and restoring such context.

```
class   processcontext =
        begin
        rep   pc:array[1:M] of word, plock:lock(1);
        . . .
        proc   contextswap(p1,p2:processcontext) =
               begin
               . . .
               end;
        end;
```

The meaning of the proc "contextswap" is, perhaps, a bit slippery and we should spend a few words on it.

At some moment of time assume that the processor is executing some process A, and further that for some reason or another we choose to change context to that of process B. The procedure "countextswap" will do this for us. But notice that the call on the procedure occurs in the context of A, and when it returns it will be in the context of B. Someplace in the middle the context has changed.

Now, since the only way that we could have left the context of B at some earlier moment was to have called contextswap, the return from contextswap in the context of B will be to a point at which that earlier call was made, that is, to a point fully expecting the return.

Got that? The only important thing to remember is that a call on contextswap always returns to the caller eventually -- it just may take a long time.

(5) Semaphores

In this section we shall define the abstraction of a "semaphore"; these are essentially identical to those originally defined by Dijkstra and we shall not motivate them further. However, we wish to explicitly decouple (short-term) scheduling decisions from synchronization, and to do this we will depart slightly from our otherwise bottom-up presentation. Specifically we shall presume the existence of a "scheduler" which provides

(a) a variable "running" which is in the local memory of each processor and which contains a reference to the process currently running on that processor.

(6) a procedure "selectess" which will select the next process to run on the processor from which it is invoked.

(c) a procedure "selectsor" which will select a processor to run a specified process.

Given these, we can define the notion of a semaphore as follows:

```
class   semaphore (initval:integer) =
    begin
    rep   cnt:integer, l:lock, Q:queue(process,FIFO);
    initialize   cnt←initval;

    proc   P(S:semaphore) =
        begin   local p:process;
        BLIND(
            CRITICAL(S.l,
                if(S.cnt←S.cnt-1)<0
                    then(enq(S.Q,running);p←selectess);
                    else p←null;
                );
            if p≠null then swapto (running,P);
```

```
                );
        end;

proc V(S:semaphore) =
        begin   local p:process;
        BLIND(
                CRITICAL(S.l,
                        if (S.cnt←S.cnt+1)≤0
                            then p←deq(S.Q)
                            else p←null;
                        );
                if p≠null then selectsor(p);
                );
        end;
end;
```

These definitions are essentially similar to those which have appeared in the literature previously and should not require detailed explanation. There are a few points, however, which deserve elaboration.

(a) Note the use of the nested BLIND and CRITICAL. Together these insure that the P and V operations are indivisible with respect to a single semaphore -- of course simultaneous P and/or V operations may be performed on distinct semaphores.

(b) The queue of processes on a semaphore is pure FIFO.

(c) Note that if a process is blocked as the result of a P operation, the procedure 'selectess' is called to choose the next process to be run, and that the last act of P in such a case will cause a context-swap to this process. When the process is unblocked (by some other process executing a V) control will resume just after the 'swapto' call and hence will then return to the caller. That is, the

'swapto' may be viewed as an 'exchange jump' co-routine linkage; it is a higher-level version of the "context-swap" operation discussed previously.

(d) The case of a V operation which unblocks a process is interesting. The V operation calls 'selectsor' with this process to attempt to find a processor to run the process. We shall examine this procedure in the next section, but note that this involves a scheduling decision, or at least some options for one, not found in a uniprocessor system.

(6) **Scheduling**

At last we will begin to see some relation between the low-level abstractions we have been building and the high-level model presented some time ago. First we shall need the abstraction of a "process" and then consider a possible scheduling algorithm. A "process" will be defined in the next section, but for now it is simply the place where the process parameters set by a PM are kept; in addition there is a field for the identity of the processor, if any, currently executing the process. Thus for the moment we may think of a process as

```
class   process
        begin
        rep   priority,slice,number-of-slices,cpsmax:integer,pnum:word;
         . . .
        proc swapto(p1,p2:process)= begin . . . end;
        end;
```

We might invent any of several scheduling algorithms at this point, but the following is a simple one which illustrates some of the options in a multiprocessor. Note, by the way, that the following is not a class simply because there is only one instantiation of it.

```
own   feasible:queue(process,PRIOR),
      processors:queue(process,REVPRIOR),
      feaslock,psorslock:lock(1);
```

```
proc   selectsor(p:process) =
     begin   local pl:ref(process),s:word;
     CRITICAL(feaslock,enq(feasible,p));
     CRITICAL(psorslock,
                    begin
                    pl←first(processors);
                    if pl.priority<p.priority then S←pl.pnum else S←0;
                    end;
                   );
     if S≠0 then IPI(ipsched,S);
     end;

proc   selectess returns (ref(process)) =
     begin   local p:ref(process);
     CRITICAL(feaslock, p←deq(feasible));
     return p
     end;
```

Note that "selectess" presumes that the feasible queue will never be empty -- thus there must be at least as many "idle jobs" as there are processors.

Now, we also need two procedures which handle interrupts -- the "ipsched" interrupt which signals that a processor is to re-schedule itself, and the clock. Here we'll define the inter-processor interrupt and leave the clock for a bit later.

```
interrupt proc   IPSCHED =
     begin   local p:ref(process);
     BLIND(
     CRITICAL (feaslock,
                    begin
                    p←deq(feasible);enq(feasible,running);
                    end);
     swapto(running,p))
```

```
        end;
```

(7) The Highest "Basic" Layer

Once again it seems appropriate, that is, pedagogically advantageous, to break with the bottom-up approach we have been following. This time we shall skip to the top of our basic layers and consider the abstraction of a process as it was presented informally before. Again we may consider a process to be a class:

```
class   process =
    begin
    rep   priority,slice,number-of-slices,cpsmax,state,tick,nts:integer,
          cntx:process-context,  rl:lock(1),
          c:cps,pnum;word;
    initialize   state←1;
    proc   setparm(P:process,pr,sl,ns,cm:integer) =
          begin
          if p.state≠1 then ERROR;
          P.priority←pr;
          P.number-of-slices←ns;
          P.slice←sl;
          P.cpsmax←cm; setmax(c,cm);
          end;

    proc   makpresent(P:process) =
          begin
          if   P.state≠1 then ERROR;
          getpages(P.c);
          P.state←2;
          end;

    proc   start(P:process) =
          begin
```

```
        if   P.state≠2 then ERROR;
        P.tick←P.slice; P.nts←P.number-of-slices;
        selectsor(P);
        P.state←3;
        end;

proc    maknonpresent(P:process) =
        begin
        if   P.state≠2 then ERROR;
        putpages(P.c);
        P.state←1;
        end

proc    wait-for-stop returns(p:ref(process)) =
        begin
        p←receive(stopnotice);
        p.state←2;
        end;

comment the following proc is not visable to users:

proc swapto(p1,p2:process) =
        begin
        LOCK(p2.rl);
        lastrun←running;   running←@p2;
        contextswap(p1.cntx, p2.cntx);
        UNLOCK(lastrun.rl);
        end;

end;
```

This is all quite straightforward except for two undefined notions -- (1) a 'cps' and the associated operations "getpage," "setmax," and "putpages," and (2) the object "stopnotice" and the associated operation "receive" done on it. We'll deal with this second object first and clear up the handling of the clock interrupt along the way.

(8) Mailboxes

Up to this point we have not needed the notion of asynchronous communication between processes; simple synchronization has been enough. But in order for the abstraction of a 'process' presented in the last section to function properly it must be able to receive information about stopped processes. Thus in this section we shall introduce the notion of a "mailbox" -- a bounded buffer of messages (actually processes in our case).

```
class   mailbox (size:integer,C:class) =
    begin
    rep   mutex:semaphore(1),limit:semaphore(size),num:semaphore(0),
          Q:queue(C,FIFO);

    proc   send(M:mailbox,X:C) =
          begin
          P(M.limit); P(M.mutex);
          enq(M.Q,x);
          V(M.mutex); V(M.num);
          end;

    proc   receive (M:mailbox) returns (x:refC) =
          begin
          P(M.num); P(M.mutex);
          x←deq(M.Q);
          V(M.mutex); V(M.limit)
          end
```

```
end;
```

This is the usual bounded-buffer mailbox. Now, using this we can explain the handling of the clock interrupt. First we'll declare a mailbox which may be used to pass stopped processes to the process class.

```
own   stopnotice:mailbox(K,process);
```

Then the clock interrupt handler is simply

```
interrupt   proc CLOCK =
    begin
    BLIND(
    if  (running.tick←running.tick-1)≤0 then
        begin
        if   (running.nts←running.nts-1)≤0
             then   send(stopnotice,running)
             else   CRITICAL(feaslock,
                    enq(feasible,running);
                    running.tick←running.slice);
        swapto(selectess)
        end;
    ) end;
```

Note, by the way, that the time slice is expressed in terms of some number of clock-ticks.

(9) The CPS

We are finally ready to examine the cps abstraction. The cps is simply a set of pages, of course, and for convenience we shall assume that there is some reasonable upper-bound, MX, on the number in any particular cps. Hence we can write

```
class   cps =
     begin
     rep   max,cursize:integer, ca:array[1:MX]of ref(page), s:semaphore(1);
     initialize   cursize←max←0;

     comment   first we define the user-visable operations;

     proc   cpsinclude(c:cps,p:ref(page)) =
            begin
            P(c.s);
            if   C.cursize≥C.max then ERROR;
            C.cursize←C.cursize+1;
            getpage(p);
            for   i from 1 to MX do
                  if   c.ca[i]=null then(c.ca[i]←p; V(c.s); return);
            V(c.s);
            end;

     proc   cpsdelete(c:cps,p:ref(page)) =
            begin
            P(c.s);
            for i from 1 to MX do
                  if   c.ca[i]=p then
                        begin
                        putpage(p); c.cursize←c.cursize-1; V(c.s); return
                        end;
            V(c.s);
            ERROR
            end;

     comment   now we define the operations used by the process abstraction;
```

```
proc   getpages(c:cps) =
       begin
P(c.s);
       for   i from 1 to MX do
             if   c.ca[i]≠null then getpage(c.ca[i]);
V(c.s);
       end;

proc   putpages(c.cps) =
       begin
       P(c.s);
       for   i from 1 to MX do
             if   c.ca[i]≠null then putpage(c.ca[i];
       V(c.s);
       end;

proc   setmax(c:cps,m:integer) =
       begin
       P(c.s); c.max←m; V(c.s);
       end;

end;
```

Note that if the maximum cps size is reduced below its current size, the user will only be allowed to delete pages until the size is below the permitted maximum.

Now, all that this abstraction has really done is to pass-the-buck to the page abstraction. We'll look at that in the next section.

(10) The Page Abstraction

You will recall from our discussion of much earlier that we want to allow sharing of pages among several processes. Thus the operations "putpage" and "getpage" used in the definition of cps, and which appear to require swapping the page out or in respectively, actually may or may not do this -- depending upon whether the page is being shared by some other process.

We can characterize a single page as being in one of three primary states: (0) non-existant, (1) in core, (2) on drum; of course, they may also be written to drum, or read from the drum, and we will have to synchronize these. Also, of course, it may be in transient state -- that is, some process may be requesting an operation on the page and its internal status is being altered. Thus, we must protect this status with appropriate mutual exclusion.

In describing this abstraction we shall have to do i/o. Since these notes are probably already too long, I shall simply assume that two page-oriented i/o operations are available.

READPAGE(coreaddress,drumaddress,semaphore);

and

WRITEPAGE(coreaddress,drumaddress,semaphore);

and assume that these operations will perform a V operation on the semaphore when the specified i/o operation is complete.

```
class   page =
    begin
    rep   coreloc,drumloc,nrefs,state:integer,
          mutex:semaphore(1), iosem:semaphore(0);
    initialize   (nrefs←state←0);
```

```
proc   getpage(p:page) =
       begin
       P(p.mutex); p.nrefs←p.nrefs+1;
       case   p.state of
              begin
              0: (p.coreloc←get(PAGESIZE); p.state←1)
              1: ;
              2: begin
                   p.coreloc←get(PAGESIZE);
                   READPAGE(p.coreloc,p.drumloc,p.iosem);
                   P(p.iosem);
                   p.state←1
                   end;
              end;
       V(p.mutex);
       end;

proc   putpage(p:page) =
       begin
       P(p.mutex); p.nref←p.nref-1;
       case   p.state of
              begin
              0:   (p.drumloc←allocdrum;p.state←2);
              1:   if p.nref=0 then
                     begin
                     p.drumloc←allocdrum;
                     WRITEPAGE(p.coreloc,p.drumloc,p.iosem);
                     P(p.iosem);
                     p.state←2
                     end;
              2:   ERROR
              end;
       V(p.mutex);
```

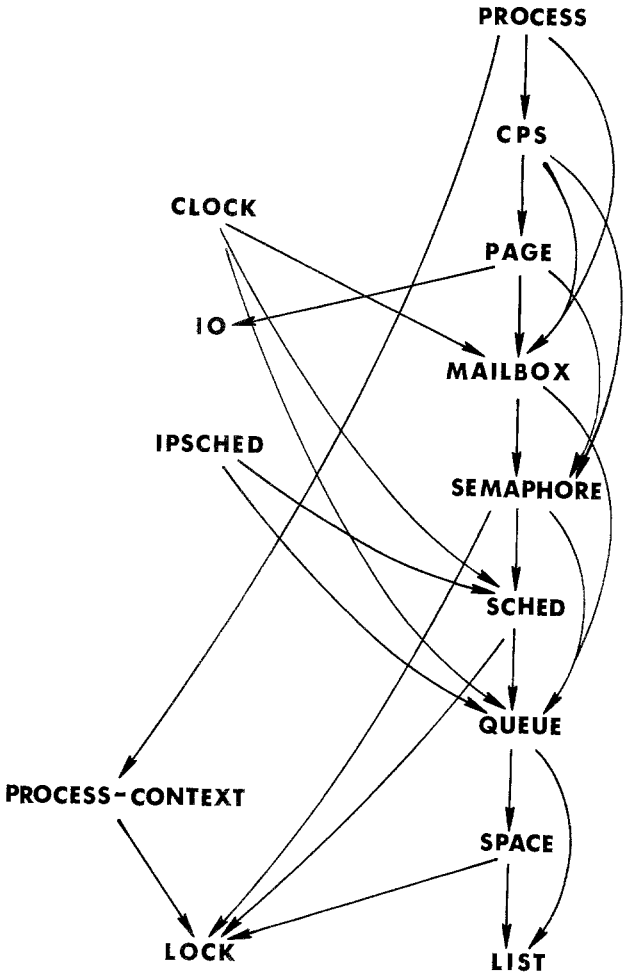
```
        end;
    end;
```

Some Concluding Remarks and Caveats

The last few pages have presented a good deal of code; such a thing is always dangerous in a set of lecture notes. It's bound to contain errors! There are probably at least two kinds. The ones of lesser importance are syntactic and/or typographical; I was inventing language as I went along and I'm sure there are inconsistencies. I did not want to be sidetracked by such issues however, and choose to plunge ahead. Second, I have not attempted to verify these programs and thus they are likely to contain conceptual errors as well. I would like to believe they will be minimal because of the decomposition, but ultimately human frailty will come to the surface; we shall see...

However, it was not my intent to produce an actual operating system for you -- either in terms of the facilities provided or in terms of the precise code. Rather, the intent was to illustrate the fine-grain decomposition of the most basic levels of an operating system. If the example alchieves that, I am satisfied.

Now it is time to sit back and reflect a moment on what I have tried to do and the "structure of the result. I have tried to start with an essentially "bare" machine and build up to a reasonable process and virtual memory mechanism; more sophisticates facilities, eg. a file system, could then be built using these basic mechanisms. The process of construction consisted of selecting a few useful concepts, eg. "queue", and defining them so that: (1) they were textually isolated, (2) their dependencies on other concepts were explicitly invoked through "operations" provided by the other concepts, and (3)trying to "hide" information about both the implementation and policies of a concept within its "class" definition. The following diagram illustrates the "depends upon" relation between the various abstractions; its a mess!


PROCESS
CPS
CLOCK
PAGE
IO
MAILBOX
IPSCHED
SEMAPHORE
SCHED
QUEUE
PROCESS-CONTEXT
SPACE
LOCK
LIST

I will, in the best tradition of course notes, leave to the reader to demonstrate that the dependency between **instances** of these abstractions is very simple.

Having achieved the decomposition, our next task is to verify each of the implementations of the various abstractions. That is the subject of a paper I shall distribute at the school.

A time-wise hierarchy imposed upon the use of a two-level store.

by Edsger W.Dijkstra.

Authors address:
BURROUGHS
Plataanstraat 5
NUENEN - 4565
The Netherlands

Abstract: Following general design principles a paging system has been developed in which has been aimed at high efficiency and a strong separation between store management and processor scheduling and a minimal influence of the program mix upon the system's performance. It is, furthermore, described how some dedicated hardware can be expected to contribute effectivel to memory management and the prevention of thrashing. Finally, the properties of the system should be such that a mismatch between configuration and workload gives a clear indication as to what reconfigurations seem indicated.
Key Words and Phrases: demand paging, window size, thrashing control, smoothness virtual store, two-level store, operating systems, design, reconfiguration, separation of concerns.
C.R.Categories: 4.32, 4.34, 6.21, 6.34, 6.39.

NUENEN, 6th December 1974

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

INTRODUCTION

A time-wise hierarchy imposed upon the use of a two-level store.

This paper is really two articles, merged into one. On the one hand it deals with a general design principle, on the other hand it deals with the design of a virtual storage system, a design to which that principle has been applied. Although the first aspect is the more general one, the title refers only to the second aspect, firstly because its elaboration occupies most of the space, and, secondly, because the virtual storage system to be developed below seems to be new and not without attractive properties.

The design principle in its most general form is that, whenever we have to design a mechanism meeting certain requirements, it does not suffice to design something of which we hope that it meets the requirements, on the contrary: we must design it in such a way that we can establish that it meets the requirements. As far as program correctness is concerned, this design principle has led to a programming methodology that is becoming more and more widely accepted: instead of making the program first and trying to establish its correctness afterwards --which may be near to impossible-- correctness proof and program are now developed hand in hand. (As a matter of fact, the development of the correctness proof is often slightly leading: as soon as the next argument in the proof has been chosen, a program part is designed so as to meet the proof's requirements.) Besides the mathematical requirement of correctness, we have the engineering requirement of "reasonable preformance' as well: this time the principle tells us, that it does not suffice to design a mechanism of which we hope that it will perform "reasonably well", but that we should (at least try to) design it in such a way that we can predict a priori how well it will perform. If we ask very precise questions about the performance, these questions may become very hard to answer: to predict that the computation time for the Horner scheme grows linearly with the degree of the polynomials is not hard, the estimation of the computation time needed for iterative computation of eigenvalues and eigenvectors of a symmetric matrix, however, is harder and probably most easily expressed in terms of the separation of the eigenvalues, i.e. in terms of part of the answer; then this dependance is something that we should try to derive and prove! Often we have to be content with "worst case" bounds (which in contrast to averages have at least the advantage of not depending on the usually unknown input population).Sometimes we have even to be content with still vaguer definitions

of what "reasonable performance" means: yet this is no licence to design, for instance, a mechanism whose performance is occasionally surprisingly bad.

The actual performance of a machine with a virtual storage system is dependent of what is usually denoted as "the workload characteristics". In the name of the predictability of that performance we shall try to design the system such as to make that dependence as simple as possible: in particular we require that a mismatch between configuration and workload does not only make itself manifest in the form of poor performance, but will in addition give a clear indication what type of change --if any-- of the configuration would improve the performance.

In order not to complicate the discussion unduly at the start, we shall make a few simplifying assumptions about the hardware. At the end we can reconsider these assumptions; some may be weakened easily, of others, however, we may come to the conclusion that if our hardware does not allow such idealizations, the scheduling problem will be "complified" seriously, perhaps even beyond our comprehension and control. In the latter case we don't need to feel having failed "to cope with the problem", on the contrary: the identification of seriously "complifying" hardware characteristics seems in the light of the present state of the art a valuable discovery.

As primary store we assume a random access store as randomly accessible as, say, a core store. As secondary store we assume a device with the characteristics of, say, a drum or a head-per-track disc, such that

1) place of information in secondary store need not influence decisions to change the contents of primary store, i.e. that page-wise it can be regarded as a random access store;

2) the processor speed is sufficiently slow and/or the cycle time of the primary store is sufficiently small and/or the transfer rate between primary and secondary store is sufficiently low that any slowing down of the processor as a result of cycle stealing by the channel (to all intents and purposes) can be ignored;

3) transport between the two storage levels is taken care of by a single, dedicated channel.

Furthermore I assume

4) a single processor

5) demand paging with fixed-size pages

6) such a modest amount of processor-status information (registers included!) that the time needed to switch the processor from one process to another can (to all intents and purposes) be ignored in view of an upper bound on the frequency with which these switchings may have to take place

7) no page-sharing between user programs (for instance possible on account of a common procedure library).

Remark 1. The above asumptions are --or at least: were-- not unrealistic. We shall later discuss some of the temptations that should be resisted when they are only partly fulfilled. (End of remark 1.)

Remark 2. Assumption 6 means that as far as scheduling processor time is concerned, we can regard the total processor time as the sum of the periods of time devoted to actual program process, and are at any time free to grant the processor to what is concerned as the most urgent task. If the price of switching the processor from one task to another has to be regarded as high, one is faced with the often conflicting aim to grant the processor to the task with the maximum expectation value for the period of time for which full-speed progress is possible. (End of remark 2.)

The role of the replacement algorithm in a multiprogramming environment.

The idea of demand paging is that processing proceeds at full speed as long as the information is present in primary store. Upon a so-called "page fault" --i.e. the detected desire to access a page that is currently not in main store-- the missing page must be brought in from secondary store. (The program causing the page fault has to wait until the channel has completed that transport; in a multiprogramming environment the processor is in the meantime available for other programs.) Besides bringing in the missing page, another page has to be dumped. It is the task of the so-called "replacement algorithm" to choose that victim; its goal is to keep the interesting pages in primary store. Obviously, with each reasonable replacement algorithm, permanently unreferenced pages have a tendency to disappear sooner or later from primary store.

The ideal replacement algorithm embodies clairvoyance: it kicks out the page that in view of future needs can be missed best. Clairvoyance, however, is hard to implement, and actual replacement algorithms are based upon, essentially, three different ideas. (We shall later see that for our purposes the first two have to be rejected.)

1) With a (quasi-)random number generator an "arbitrary" page residing in primary memory is chosen as the victim. It is reasonable in the sense that permanently unreferenced pages have indeed a tendency to disappear from primary store, it is simple and its performance is not half as bad as might be expected.

2) In an effort to speed up the disappearance of permanently unreferenced pages the machine keepts track of the order in which the pages currently residing in primary store came in, and the older ones are given a greater probability of being chosen as the victim. In the extreme case, always the oldest is chosen and the algorithm becomes a FIFO ("First-In-First-Out") rule

3) Predicting tomorrow's weather according to the principle "the same as today", the machine keeps to a certain extent track of the order in which pages currently in primary store have been accessed, and pages which for a relatively long time have not been accessed are given a greater probability of being chosen as the victim. In the extreme case we get the so-called LRU-algorithm ("Least Recently Used").

Note 1. In the case of cyclic access to $n+1$ pages with room for only n, both FIFO and LRU give the worst possible choice. As purely periodic access patterns are not unrealistic, it has been suggested to incorprate always a randomizing element in the page replacement algorithm, so as to reduce the probability of such a "disastrous resonance" to nearly nil. (End of note 1.)

We shall resume the discussion of the replacement algorithm later, because in a multiprogramming environment a more crucial decision has to be taken first. When a new victim has to be chosen, there are two alternatives:

1) either we regard primary store as a homogeneous pool of page frames and the victim is chosen on account of the total history in core, independent of the identity of the program that caused the page fault;

2) or we regard the page fault as a private occurence of the program in

which it happened, only the history of the pages of this program is taken into account and one of its own pages will be selected as the victim.

In the design of the THE-multiprogramming system in the early sixties I have chosen the first alternative and I remember the (opportunistic) arguments in favour of that decision: firstly it removed the obligation to keep track of which page frames were occupied by which programs --an administration that would have been complicated by the presence of shared library pages--, secondly it would automatically see to it that a program idling for other reasons would not continue to occupy page frames, as its then permanently non-accessed pages would disappear via the normal mechanism (which was LRU, related to the total history). This paper is a peccavi in the sense that --as I hope to demonstrate convincingly in the sequel-- this decision has been more than a mistake: it was a sin against proper design. (One of its unattractive features was that a large high-vagrancy program always lost its pages, and, as a result, suffered from very slow progress.) In the meantime we know that "separation of concerns" should be one of our dearest goals, and in the case of choice 1 the page faults caused by a single program are dependent both on its fellow-programs and on the relative speeds with which they are allowed to proceed. In the case of choice 2, however, were each program has its own, fixed number of page frames at its disposal, the generation of page faults is each program's private business, only dependent on that number of page frames, its access pattern and its(!) replacement algorithm. The mistake we made ten years ago was to allow a hardly controllable fine-grained interference between fellow programs that had been independently conceived but found themselves by accident mixed, instead of maintaining for these mutually independent programs to a much courser grain of time the mutual independency between their computational histories.

In the following we make the weak assumption about the replacement algorithm(s) used, that the average frequency of a program's page fault generation is a non-increasing (and usually even: a decreasing) function of its so-called "window size", i.e. the number of page frames allocated to it.

About the ideal window size.

In this section we shall describe how we propose to exploit our first three assumptions. After having observed that it is the function of the re-

placement algorithm to try to reduce --with a given window size-- the number of page faults caused by that program and, therefore, the total amount of time the channel is busy for the benefit of that program, our next purpose is to keep the channel nicely busy.

For each program we can introduce the total time C the processor has performed "computation" for that program, and also the total time T the channel has been occupied with "transports" between storage levels as a result of page faults caused by that program, both times C and T being recorded for that program since the same moment. When deciding how to allocate page frames to programs, i.e. when deciding the window size for each program, we seem to be managing three resources, viz. processor, channel and primary store. In this management problem, general dimension considerations tell us that the dimensionless quantity C/T must be significant. The point is, that processor and channel are resources doing something at a certain speed, but we cannot change the "speed" with which something is kept in store (no more than we are able to wait twice as fast for something).

Under the (temporary) assumption that for each program such a window size exists, we define for each program the "ideal" window size as the one that would give rise to a ratio $C/T = 1$, i.e. the window size that would cause on the average equal demands on processor time and channel time, the reason being that then processor and channel can be scheduled as a single resource. The result of demand paging is that a program has no use for the processor during the period of time that the channel is busy for it; as a result no program can occupy more than 50 percent of this combined resource, and if we want to keep the latter busy, we conclude that our degree of multiprogramming should at least be equal to two. This degree will usually not suffice (see below).

About the degree of multiprogramming.

In this section we assume that for each program the vagrancy characteristics are such that for each program a constant --and known-- window size can be considered as ideal.

In order to keep the combined resource constantly busy, individual C/T-ratios close to 1 is in general not enough. Suppose that the one program

generates its page faults --when executed all by itself-- quite regularly, one at a time, while the other program would generate under the same circumstances with half the frequency bursts of two page faults at a time: the combination would not fit and both processor and channel could be busy for at most 80 percent of the time. With a third program (of either type) full occupation is possible and an arbitrary program can use the maximum 50 percent. The typical purpose of multiprogramming is clear as far as utilization of the active resources is concerned: to absorb the bursts in which programs may generate page faults. After some consideration --and in analogy to other statistical phenomena-- it becomes hard to believe that the desire to absorb the bursts would ever give rise to a degree of multiprogramming exceeding 4 or 5 .

About the adjustment of window sizes.

We have introduced the notion of the "ideal" window size as the one by which program progress implies on the average equal loads C and T for processor and channel respectively. As a result the question whether for a given program the actual window has the ideal size or not, is meaningless unless it is related to a sufficiently large section of computation history, in which the increase of C + T is an order of magnitude larger than the T-increase caused by a single page fault (say: 20 times). Up till now, we have done as if during each computation the access pattern was sufficiently constant so that from beginning to end a single window size could be regarded as "ideal" for it, and also that for each program this size was known. In usual practice neither of these two conditions is fulfilled and, therefore, the system is required to discover for each computation what the ideal window size is, and to adjust for each program the window size when needed. For each program reconsideration (and possibly adjustment) of the window size should only take place with a frequency which is an order of magnitude smaller than that of the target frequency of page fault generation: it is pointless to be willing to vary a program's window size so rapidly that the periods during which it is by definition constant are so short that the question as to whether it was "ideal" becomes meaningless!

Let us assume therefore that for each program the system reconsiders its window size each time when that program has increased its C + T by a certain amount (equal to, say, 20 times the T-increase corresponding to a

single page fault.) When since the previous reconsideration of the window size C has increased much more than T , a smaller window might be more adequate, when T has increased much more than C , a larger window might be more adequate. We could think of a simple negative feedback, based upon the quotient of the observed increases of C and T , say decreasing the window size by one page frame when that quotient exceeds 1.1 and increasing the window size by one page frame when that quotient is less than 0.9 . Such a simple negative feedback, however, will not do the job, because even if our replacement algorithm is such that we can prove that a larger window would never lead to more page faults, the program might be such that a larger window would not lead to fewer page faults either!

A computation with high-frequency access to two fixed (program) pages and random access to 10,000 other (data) pages will not perform any better with a window of 100 frames (our maximum say) than with a window of 3 . If it has a window of 3 and its C/T ratio is too small, there is no point in increasing the window size. The simple negative feedback would continue to increase it and (like a young cuckoo) this program would eventually push the other programs out of primary store. This cuckoo effect cannot be remedied without penalty by suppressing growth of the window --although desirable on account of C/T-- as soon as no improvement is observed, and the reason is the following. A program with high-frequency access between 12 pages may perform equally poor with windows up to 11 frames and beautifully with a window of 12 frames, and this is something we would like to be discovered when its current window happens to be 4 . In other words: it is not enough to know the C/T- ratio caused by the current window size, we should also know it for other ones!

Monotonic replacement algorithms.

There is an important class of replacement algorithms --LRU is one of them, RANDOM and FIFO are not-- which we might call "monotonic", and are characterized by the following property. Considering two synchronized executions of the same program but with different window sizes, we call the replacement algorithm "monotonic" if at all times all pages contained in the smaller window will be contained in the larger window as well, provided that this was true at the beginning. As a result, in the computation with the larger window no page fault occurs that does not occur in the other computation as well.

Therefore, if a program is executed with a monotonic replacement algorithm and an actual window size w, it cannot cost much to record how many page faults would have occurred if the window size had been $w + 1$, $w + 2$ .. up to the maximum: it would only be a minor overhead on the actual page faults and would, therefore, be negligible. This information can be used to prevent the growth of a cuckoo, it does not cater for the detection of an existing cuckoo, i.e. a program whose window size can be decreased without any ill effects.

To record the page faults that would have occurred with window sizes smaller than the actual ones, additional hardware seems indicated. The knowledge of the number of page faults that would have occurred with smaller sized windows (particularly for the size $w - 1$) is so attractive to have, that the additional hardware seems justified. (In the latter case it can probably also take care of the recording of the number of page faults corresponding to window sizes larger than w.) Plotting page-fault frequency against window size it is not uncommon that this curve has a very sharp bend: we may expect programs that for a given window size w will give a ratio $C/T > 1$, while with a size $w - 1$ the ratio C/T would drop down unacceptably close to zero. With the simple feedback mechanism the effort at window size adjustment would lead to thrashing half the time --a nasty property of that feedback mechanisms that has been used as an argument against virtual storage systems as such-- . If additional hardware counts the virtual page faults that would have occurred with window sizes smaller than the actual one, the thrashing half the time is easily avoided.

In view of the above it is doubtful whether the introduction of a randomizing element in the page replacement algorithm in order to avoid "disastrous resonance" --see Note 1-- is still desirable: most desastrous resonances occur when the window size is a few frames too small. But now we can detect this and know how to remedy it, it seems better not to obscure the detection by the noise of a randomizer.

The time-wise hierarchy.

At our lowest level we have the individual access: the recording of its having taken place (for the sake of the replacement algorithm) and the test whether it causes a (virtual or actual) page fault are obvious

candidates for dedicated hardware.

At the next level we have the actual page faults, which occur several orders of magnitude less frequently. Taken in isolation they only influence the program in which they occur.

At the next level, but again an order of magnitude less frequent, the window size is reconsidered. In the decision to increase or decrease the window size a threshold should be introduced so as to increase the probability that the result of reconsidering the window size will be the decision to leave it as it stands. Furthermore, if available information suggests a drastic change in window size, follow this suggestion only partly --halfway, say-- : either the suggestion is "serious" and the total change will be effectuated within two or three adjustments anyhow, or the suggestion is not "serious", because the access pattern is so wild, that the notion of an "ideal" window size is (temporarily or permanently) not applicable to that program. In the latter case it is better to allow this program to contribute unequal loads to the processor and the channel --if it only occupies one tenth of that combined resource, it can only bring the two total loads mildly out of balance-- .

At the last level, but again at a lower frequency, change of window sizes may have to influence the degree of multiprogramming: growing window sizes may force load shedding, shrinking window sizes can allow an increase of the degree of multiprogramming.

As a result of past experience, the fact that these different levels (each with their own appropriate "grain of time") can be meaningfully distinguished in the above design, gives me a considerable confidence in its smoothness, in its relative unsensibility to workload characteristics.

Efficiency and flexibility.

The purpose of aiming at C/T-ratios close to 1 was to achieve for the active resource (i.e. processor and channel combined) a duty cycle close to a 100 percent, to a large extent independent of the program mix. This freedom can still be exploited in various ways. A program needing a large window on account of its vagrancy can be given the maximum 50 percent

of the active resource in order to reduce the time integral of its primary storage occupation. Alternatively we can grant different percentages of the active resource in view of (relatively long range) real time obligations: to allocate a certain percentage of the active resource to a program means to guarantee a certain average progress speed. (This seems to me more meaningful than "priorities" which, besides being a relative concept, can only be understood in terms of a specific queueing discipline that users should not need to be aware of at all!)

Remark 3. When a producer and a consumer are coupled by a bounded buffer, operating system designers prefer to have the buffer half-filled: in that state they have maximized the freedom to let one partner idle before it affects the other, thus contributing to the system's smoothness. Granting no program more than 50 percent of the active resource is another instance of consicously avoiding extreme of "skew" system states! (End of remark 3.)

Temptations to be resisted.

If we enjoy the luxury of a full duplex channel, the page being dumped and the page being brought in can be transported simultaneously (possibly at the price of one spare page frame). Usually, however, such a page swap between the two storage levels takes twice as much time as only bringing in a page. If the channel capacity is relatively low, it is therefore not unusual to keep track of the fact whether a page has been (or: could have been) written into since it was lastly brought in: if not, the identical information still resides in secondary store and the dumping transport can be omitted. This gain should be regarded as "statistical luck" which no strategy should try to increase and which should never be allowed to influence one's choice of the victim (quite apart from the fact that it is hard to reconcile with the monotinicity of the replacement algorithm, as the monotonic replacement algorithm is defined for all window sizes simultaneously, independent of the size of the actual window).

We have also assumed the absence of page sharing. But this was not essential: if program A wants to access a page from the common library which at that moment happens to reside in program B's window, a transport can be suppressed by allowing the windows to overlap on that page frame.

Both programs keep, independently of eachother, track of their own usage of that page for the sake of their own replacement algorithm and the page only disappears from main store when it is no longer in any window at all. Again, this gain should be regarded as "statistical luck" which should never be allowed to influence our strategies. Such pressure should be resisted, yielding to it would be terrible!

Analyzing the mismatch between configuration and workload.

If the channel achieves a duty cycle close too 100 percent , but the processor does not, a faster channel (or more channels) or a slower processor may be considered. If the processor achieves a duty cycle close to 100 percent , but the channel does not, a faster processor (or more processors) or a slower channel may be considered. (With two processors and one channel each program has the target C/T-ratio = 2 .)

Note 2. A change in the quotient of processing capacity and transport capacity will give rise to other window sizes. With the built-in detection of virtual page faults as well, a user can determine himself what effect on the window sizes the change in that capacity ratio would have for his workload, without changing the actual window sizes. He should do so before deciding to change the configuration. (End of note 2.)

If neither processor, nor channel achieves an acceptable duty cycle, we either have not enough work, or are unable to buffer the bursts. If we have enough independent programs, a larger primary store could be considered so as to increase the degree of multiprogramming. Otherwise we should consider the attraction of more work, or reprogramming --so as to change vagrancy characteristics--, or a completely different installation (e.g. with very different secondary store characteristics). Or we may decide to do nothing about it at all and live with it.

Acknowledgements. Collective acknowledgements are due to the members of the IFIP Working Group W.G.2.3 on "Programming Methodology" and to those of the Syracuse Chapter of the ACM. Personal acknowledgements are due to the latter's Chairman, Jack B.Cover, to Steve Schmidt from Burroughs Corporation, to John E.Savage from Brown University and Per Brinch Hansen from the California Institute of Technology.

CHAPTER 4.: PROGRAMMING SYSTEMS STRUCTURE

A. P. Ershov

Computing Center
Siberian Division of the
USSR Academy of Sciences, Novosibirsk
USSR

Problems in Many-Language Systems

Table of contents

List of figures 360

Lecture 1. Introduction and preview of the BETA system 361
1.1. Introduction 361
1.2. Brief overview of the system 363
1.3. Plan of the course 366
1.4. Example 366

Lecture 2. Internal language of the BETA system 368
2.1. Design concepts 368
2.2. INTEL program scheme 369
2.3. INTEL objects 370
2.4. INTEL statements 373
2.5. Transput 376
2.6. Parallelism 376
2.7. Discussion 377

Lecture 3. Decomposition and synthesis in the BETA system 380
3.1. Introduction 380
3.2. Lexical analysis. Executive procedures 382
3.3. Syntactic analysis and parsing 384
3.4. Semantic analysis and synthesis 384
3.5. Lexical information 385
3.6. Syntactic information 385
3.7. Semantic information 386
3.8. Information for synthesis and code generation 387
3.9. Discussion 388

Lecture 4. Optimization and code generation in the BETA system 391
4.1. Collection of the optimising transformations 391
4.2. Analysis 393
4.3. Factorization 394
4.4. Preliminary code generation 394
4.5. Memory allocation 395
4.6. The coding of subroutines and procedures 396
4.7. Final code generation 397

Lecture 5. Compiler writing systems as a factor in unification and comparison of programming languages 398
5.1. Introduction 398
5.2. Universal executive procedures for synthesis 390
5.3. Criteria for evaluation of programming languages 402
5.4. Data types
5.5. Name declarations 403
5.6. Resume of the comparison 405
5.7. Conclusion 407

Acknowledgements 407
References 409
Index of terms 410
411

List of figures

1. Compiler writing system
2. Compilation scheme in the BETA-compiler
3. Source program and result of the lexical phase
4. Parsing string
5. Parsing tree
6. Attributed parsing tree
7. INTEL program
8. Shadows and optimization attributes
9. Object program
10. Lexical information
11. Regularized C-F grammar
12. Relation between a grammar and parsing tree data base
13. Example of EP for semantic analysis
14. Example of EP for synthesis
15. Sequence of optimization passes
16. Alternative code generation

LECTURE 1

INTRODUCTION AND PREVIEW OF THE BETA SYSTEM

1.1. Introduction

At the Working IFIP Conference "ALGOL 68 implementation" held five years ago in Munich, the start of work on a multi-language programming system was announced [1]. The system has got an acronym "the *BETA system*". The work has been organized as a project ("the *BETA project*"). Though the work on the research and experimental part of the project is not completed, some important goals have already been achieved and problems still to be solved have been identified and described clearly enough. We therefore take the liberty to present you this intermediate status report, since some of the goals of the BETA project are closely connected with the theme of this summer school.

Actually, the essence of the BETA project is to develop a compiler writing system (CWS) and, using it as a tool, to implement several widespread algorithmic languages.

We understand a *compiler writing system* as an integrated collection of techniques, working procedures and programs which, being applied to any algorithmic language from some class of source languages, produce a compiler implementing the given language for some object computer.

Actually, a CWS is a combination of people, programs and computers (Fig. 1). *Designers* develop a CWS starting with some hypothesis on a class of languages. A *linguist* chooses an input language from the class and prepares its description for implementation. The *Metaprocessor* is an operational part of the CWS which "processes" the language description into a compiler implementing it. Metaprocessor functions are performed partially by an *instrumental computer* and partially by an *implementor*.

The class of languages on which our research is oriented definitely contains such languages as

FORTRAN
COBOL
ALGOL 60
ALPHA (Input language)
SIMULA
PL/I
SIMULA 67
ALGOL 68
PASCAL

It is not so easy to outline this class precisely and we only will say that we would not include into the list such languages as SNOBOL or LISP and would consider BLISS as a marginal language.

We formulate a general scientific and technological problem faced by the BETA project as follows:

+ to find out a real commonality in widely used programming languages, compilation methods for programs expressed in these languages, and the execution of these programs on computers;
+ to develop, on a basis of discovered commonality, a technology of implementation of different source languages, by means of a unified multi-language programming system suitable for a variety of object computers;
+ to ensure that the effort and complexity of preparation of a language description as input for the meta-processor are considerably less than conventional methods of "tailored" implementation, and to do so without heavy losses in efficiency.

Now, we are in a position to say that we have some positive outcome with respect to the first two problems and this enables us to express a hope of a positive solution of the third problem.

The commonality which we mean, in very general terms, can be expressed in the following categories:

phrase structure
block localization
common base statements
similar decomposition rules
expressions
data types
operational character of semantics
separation of compilation from execution

Such, as we believe, a very real commonality supporting these lists of languages and concepts allows us to enforce unifying properties of the CWS by organizing it around a *universal multi-language compiler* (UMLC).

The UMLC concept in a CWS design means that, for each language from a considered class, a single compilation theme is selected. This scheme is constructively embodied as a common data base in which different compilation phase algorithms are embedded. Some of these algorithms are invariant with respect to source languages; others depend partially on them, and a third set is completely defined by a language. What is important, is that there exists a "template" of the UMLC which is implemented in advance, has some integrity in itself; and thus implements a unification of compilation algorithms.

1.2. Brief overview of the system

The general compilation scheme adopted for the universal multi-language compiler in the BETA system (the *BETA compiler*) is shown at Fig. 2.

Let us make a few explanatory remarks. Unification of an input string consists of standardization of the input alphabet, classification of lexemes and techniques for symbol table construction. Unification of a parsing tree consists of a choice of uniform representation for tree arcs (Polish notation or list structure) and, what is most important, of

a uniform approach to the representation of attributes of tree vertices formed by language notions (nonterminal symbols).

The internal language of the BETA system (the *INTEL language*) plays a fundamental role in the research part of the BETA project as well as in the BETA system proper. In distinction from the previous language levels acting as a metaform, the INTEL language provides a universal means to:

- concrete specification of source language semantics
- performing optimizing transformations
- uniform specification of code generation for different object computers

Repeating a previous analogy, INTEL objects and statements should be greatest common divisors of source language constructions and lowest common factors of object computer instructions. In addition to this INTEL constructions have to explicate the information and control flow in a program.

The BETA compiler algorithms abstract compilation properties of algorithmic languages. Lexical analysis and parsing are "automata type" algorithms which are entirely defined by a source language grammar. There exists a single metaform (for all languages) which is a base for grammatical metaprocessors automatically (or with an implementor's assistance) generating corresponding compiler modules.

Semantic analysis and internal program synthesis algorithms are libraries of so-called executive (semantic) procedures which, in general, are specially written for each source language. The content of the libraries is defined by a list of grammatical elements of a language (lexemes, notions) as well as by specification of their semantic attributes. The libraries are unified by their structure as well by their functions. Structurally, the libraries are embedded into some common data base and are subjected to a uniform system of procedure calls, parameters passing and data access routines. Roughly speaking it is possible to say that an occurrence of a grammatical element in the parsing tree causes a call of the corresponding executive procedure. A functional unifica-

tion of executive procedures is related to so called "universal grammatical elements", i.e. such notions and lexemes which we try to recognize in every algorithmic language. An identification of universal grammatical elements and their proper parametrization enables us to write unified executive procedures in a generalized form which would be appropriate for implementation of several languages.

A general form of optimization and generation algorithms is naturally provided by a representation of all compiled programs as INTEL programs.

Let us now briefly describe a current status of the BETA system (July 1975). The designers are now engaged in what we call a trial implementation of the BETA compiler. In the trial implementation, we are trying to realize all our knowledge of compilation algorithms and methods of an incorporation of a language into the system, but we do not aim at ultimate efficiency or the full range of production features (check out, error messages, extended run-time libraries etc.).

The integration of the BETA compiler is being made over an implementation of the Pascal language. This language is now becoming a kind of a model for experimentation with some or other novelties. The YARMO language is used as an implementation language. YARMO is a home-made high-level machine oriented language which slightly resembles BLISS in contents and ALGOL 68 lexically. A BESM-6 computer (10^6 op/sec, 32 K 48-bit words) is used as object, compiling and instrumental computer. We hope to get the first Pascal programs compiled by the BETA compiler to the end of this year. Implementations of PL/I, ALGOL 68 and SIMULA 67 are being conducted in parallel. Student projects for an experimental implementation of BLISS, FORTRAN and ALGOL 60 are initiated. The metaprocessors are still non-formalized and are being designed gradually following the accumulation of an experience in the work with concrete languages.

1.3. Plan of the course

We shall devote the remaining part of the first lecture to a demonstration of an example of a simple Pascal program. Its representation during different compilation phases will be shown and some hints about compilation mechanisms will be given.

A review of the INTEL language and its use as a semantical basis of the source languages of the BETA system will be presented in the second lecture.

The third lecture will be devoted to the decomposition and synthesis phases of the BETA compiler. This part is most tightly connected with source language specifications, and a discussion of rules for source language description will occupy a considerable part of the lecture.

The fourth lecture will be completely devoted to a description of optimization and generation mechanisms in the BETA compiler. Special attention will be paid to the problem of analysis of INTEL programs and reduction of complexity of global optimization algorithms.

In the fifth and last lecture we shall analyze compiler writing systems as a very interesting and, in our opinion, very powerful impetus towards unification of programming languages. Somewhat anticipating our presentation, we shall note that a joint implementation of several languages happens to be an appropriate basis for a sound comparative analysis of contemporary algorithmic languages. It seems to us that such an analysis could be useful in search for an ideal programming language, Utopia 84, advocated by Don Knuth in his recent paper (7).

1.4. Example

The example presents a series of intermediate forms of a simple program at different stages of the compilation proc-

ess (Fig. 3 - 9). The direct comments are minimal. The actual forms are, certainly, edited to make the program more readable. A simplicity of the example does not provide enough work for certain essential phases of the compilation, especially, optimization. Many details will become more understandable during a second reading this material.

LECTURE 2

INTERNAL LANGUAGE OF THE BETA SYSTEM

2.1. Design concepts

The internal language is of fundamental importance in the BETA system. It provides a complete representation of any compiled program in any source language during a considerable part of the whole compilation process, from parsing — through optimization — up to object code generation. It indicates the universality of the BETA compiler in its optimization and generation phases and serves as a basis of a construction of metaprocessors parametrized by source language characteristics. In accordance with this the following design goals for a specification of the internal language of the BETA system (INTEL) have had to be achieved:

+ Universality of INTEL, i.e. that it be possible to obtain an adequate and reasonaly direct INTEL representation of source programs expressed in the different high level languages. Possible inefficiency of expressing irregular or defective features of source languages must not influence an efficient representation of typical and regular features of the languages.

+ Convenience of performing optimizing transformations over INTEL programs. Any result of an optimizing transformation has to preserve the semantics of the INTEL program. Analysis of the applicability of an optimizing transformation as well as the transformation itself have to be of appropriate algorithmic complexity.

+ Transparency of the generation of an object program. INTEL features have to reflect more or less directly the more typical properties of contemporary computers.

2.2. INTEL program scheme

Instead of following the conventional phrase structure of source languages, INTEL programs are composed in terms of different graph (or list) structures which constitute an INTEL program scheme. These structures are as follows (Fig. 7a,b):

2.2.1. *Mode graph*. Each vertex of the graph corresponds to a mode. Terminal vertices correspond to plain modes. A non-terminal vertex V corresponds to a compound mode. Its components are those modes whose corresponding vertices are reached directly by arcs leading from V. An equivalence relation is defined over vertices of the mode graph.

2.2.2. *Control flow graph*. Its vertices are INTEL program statements and its arcs denote transfer of control. Statements are labelled by there arguments and results, which, in turn, are either labelled by names of variables or are involved in a binary relation which directly shows an *information connection* from a result to an argument. Information connections are allowed only within a linear component of the control flow graph, each argument and result being involved in no more than one connection.

Some subgraphs of a control flow graph are identified as *fragments* of some class.

The following classes of fragments are distinguished

block
procedure
subroutine (non-recursive procedure without local variables)
task (in parallel computations)
co-routine (as in SIMULA 67)
procedure call
actual parameter evaluation (*thunk*)

2.2.3. The *static nesting graph* is a tree presenting the static nesting of bloks in an INTEL program.

2.3. INTEL objects

2.3.1. *Memory*. For the description of INTEL semantics we introduce the following internal objects:

memory (consisting of memory elements of different kinds)

generation (as a set of memory elements)

dynamic enclosure graph (presents the current nesting of activated fragments during a program execution)

Among memory elements we distinguish those for:

plain mode values

references

array and structure descriptors

prodecure descriptors

event, interrupt, task descriptors.

This distinction is entirely functional. All identified INTEL objects (except files) are allocated in a common direct-access memory consisting of words which can be grouped in indexed arrays. Each word can be considered as being composed of several bytes. In addition to the common memory there is a *stack* which is used for control and information connections (anonymous variables) already mentioned, which can be implemented as a register or as a push-down memory.

Each INTEL object is characterized by a size, mode and memory class. Some modes (actual modes) automatically prescribe the size of a variable of this mode; as for some other modes the size is taken from its descriptor. Size is calculated in words, bytes or bits.

2.3.2. Modes. In the following list of INTEL object modes, names are self-explanatory and require no comment:

literal

Boolean

integer

floating real

floating complex

nonqualified reference

virtual procedure (formal parameter of a formal procedure)
string
binary string
long integer
long floating real
long floating complex
variable label
procedure without parameters and result
format
file
short integer
short real
short complex
event
general procedure
fixed string
byte
gecimal integer
fixed point real
segment
qualified reference
area (as in PL/I)
static array
array descriptor
flexible array
variantless structure
variant structure
static array of structures
descriptor of an array of structures
word

2.3.3. Memory classes. There exist the following memory classes
static
automatic
semiautomatic

formal parameter
controlled
based
stack
information connection

Static objects are allocated to memory addresses by the compiler. Automatic objects are allocated on execution of the BEGIN statement when entering a block and their memory is released at any exit from the block. In the case of semi-automatic objects BEGIN allocates memory for their descriptors. Evaluation of the size of object, together with the allocation of actual memory for them, is done by FILL statements. Memory release is similar to that for automatic objects.

Memory for formal parameters is allocated by the LOAD (parameter) statement and released by any exit from the procedure body.

Memory for a new generation of controlled (based) objects is allocated and released by ALCON (ALBASE) and RECON (REBASE), respectively.

Memory allocation and release in the stack is done as a side-effect of various control statements which use the stack

2.3.4. Constants

Constants of the following modes are allowed in INTEL programs

Boolean
integer
fixed point real
real
complex
string
binary

2.4. INTEL statements

2.4.1. Memory control

BEGIN - allocation of automatic memory
END - release of automatic and semiautomatic memory
FILL - allocation of semiautomatic memory
ALCON - allocation of controlled memory
RECON - release of controlled memory
ALBASE - allocation of based memory
REBASE - release of based memory

2.4.2. Data access

Array components allocated in separate memory locations (words) are called aligned components. Array components occupying a part of a word can be allocated in the memory as packed byte-components

MOD - displacement from an array base indicated by a value given by an index.
PACK - packing or unpacking an object depending on a Boolean argument.
REBYTE - read a byte-component
WRIBYTE - write a byte-component

2.4.3. Control statements

GOTOC - go to a constant label
GOTOV - go to a variable label
SWITCH - go to a switch
IF - conditional statement
FOR - regular for statement
CONTINUE - loop delimiter

2.4.4. Procedures

ENTRY - procedure entry (INTEL allows multi-entry procedures)
RETURN
CALLSTART - start of a procedure call. This statement deals with differences between recursive, parallel and multi-entry procedures.

LOAD - loading an actual parameter
CALL - procedure call proper

2.4.5. Assignments
The assignment statement has a conventional structure

:= <destination> <source>

The source can be any object, data access operation or so called *multi-operation* which is a construction, which originated in the ALPHA system [9]. In the general case a multi-operation unites a sequence of commutative infix binary operations for which there exist (in some or other sense) an inverse operation:

conventional expression	multi-operation
a + b - c + d - e	$+_{00101}$abcde
a × b × c/d/e	$\times_{00011}$abcde
$\bar{x} \vee \bar{y} \vee z \vee u$	$\vee_{1100}$xyzu
$x\bar{\bar{y}}\bar{\bar{z}}u$	$\wedge_{0110}$xyzu

There exist the following basic arithmetic operations (in parenthesis - inverse operations):
addition (argument with inverse sign)
disjunction (negated argument)
addition by mod 2 (negated argument)
multiplication (inverse argument)
integer multiplication (integer quotient)
conjunction (negated argument)
reminder (first argument with inverse sign)
integer power (argument with inverse sign)
VALUE - dereferencing operation
All operations are extended over compound objects componentwise.

2.4.6. Relations
= = equality of references
=/= nonequality of references

= equal
≠ nonequal
< less
> greater
≤ less or equal
≥ greater or equal

All relations are conventionally extended over compound objects.

2.4.7. Non-interpreted operations

There is one general statement COMMAND whose operation is not "understood" by the INTEL language and is expanded at the generation phase. Special attributes, which can be necessary for optimization, are stored in special fields of the COMMAND statement.

2.4.8. Interrupt handling

CHECK a condition
MASK a condition
SIGNAL a condition
SET a reaction on a condition

All conditions are numbered and are defined by an operational environment of the system. Among others, the following conditions are distinguished:

division by zero
overflow
exponent overflow
loss of significant digits

2.4.9. Other statements

EMPTY - empty statement
HALT - halting statement

2.5. Transput

INTEL means of transput are oriented towards computers with non-protected memory for control words used in transput operations. This means that there is no need to have INTEL' special operations concerning file descriptors and similar objects.

Three kinds of transput are considered:

+ a sequential stream of alphanumeric data
+ record transput
+ transput controlled by data

Such means of transput identification as channel number, special names etc. do not belong to INTEL and are interpreted by special machine-oriented procedures.

There exist the following operations on files:

CREATE a file
OPEN a file
TAKE from a sequential file
READ from a direct access file by a key
PUT in a sequential file
WRITE on a direct access file by a key
CLOSE a file
LOOSE a file
DESTROY a file

2.6. Parallelism

An independent and symmetric organization of tasks is adopted. PL/I non-symmetry is implemented by the possibil ity of selecting a leading task. Tasks are synchronized by events (WAIT and POST statements) and by testing combined with setting a tag (TEST - and - SET statement).

There exist the following statements

PAR - to create a task
KILL - to liquidate a task

WAIT - to wait for an event
TEST-SET
POST - to make an event have happen
COROUT - to create a coroutine
RESUME - to resume coroutine execution
CURTASK - to get a reference to a current task from inside that task
BASE - to get a reference to a generation of storage space local to the current block.

2.7. Discussion

2.7.1. General properties of the INTEL language are a very appropriate subject for a discussion on language hierarchies and interfaces. There is a natural hierarchy of languages in the BETA system: the family of source languages - the INTEL language - the family of object machine languages. This central position of the INTEL language characterizes its role as an interface between source languages, object languages and compilation algorithms. Positive factors which make INTEL specification easier are as follow:

- its internal character: INTEL programs are automatically synthesized and not written by a programmer
- as a consequence, INTEL is free from many problems related to human factors: static mode checking, reliability, elegant lexics, conciseness, textual representation
- a comparative uniformity of fundamental properties of different hardware models: the properties of BESM, CDC, IBM 360, Burroughs are in a reasonable extent, integrated in INTEL features.

2.7.2. Interface with source languages. It is appropriate to stress that we are not satisfied with what is only in principle a representation of source language semantics.

We have to achieve reasonable efficiency, if CWS generat-

ed compiles are to be competitive with "language tailored" compilers, at least, with respect to the quality of object programs, and a source language implementation would be a task of reasonable complexity for a "linguist" and "implementor".

There is a natural classification of the descriptive methods of the source languages.

a) unique features which are not observed in most other languages (usually, too special or abnormal), e.g. quasi-parallelism in SIMULA 67, multi-entry procedures in PL/I, switches in ALGOL 60.

b) features which are similar pragmatically but not comparable in details of specification (for example the "length" specification in Pascal, PL/I and ALGOL 68) or implementation (flexible arrays in ALGOL 68 and structures with REFER OPTION in PL/I). Another typical example is transput formats and operations.

c) those features which could be treated as universal grammatical elements.

Following this remark we distinguish the following approaches to non-trivial interfaces with semantics of the source languages.

2.7.3. Simple association of a unique feature with other features of the INTEL language (for example, multi-entry procedures). This is the least desirable method of providing universality.

2.7.4. Decomposition in the sense of representing a source language construct by INTEL program portions composed from more primitive but more general statements. This method is the most natural and widespread. It reduces essentials instead of multiplying them as in the previous case. A decomposed representation sometimes improves the selective power of the optimization and generation algorithms. Decomposing has, however, its limits: some "actions" of source language addressing the operating system have to preserve

its integrity "up to the end"; similarly, some complicate "combined" machine instructions loose points of applicability to a compiled program.

2.7.5. Underdefinition consists in refusing to interpret a source language construction completely but replacing it by a formal symbol supplemented at a synthesis phase by some attributes providing the necessary interface with the optimization process (e.g. presence of side-effect, terminating linear component). Implementation of such construction is postponed up to generation phase. This approach is also less desirable and recommended only in well-restricted constructions (see above COMMAND statement).

2.7.6. Parametrization of the INTEL language and compilation algorithms is considered to be a most effective aid to control machine-oriented features of source languages and to reduce the need for underdefinition. Parametrization incorporates into INTEL, as well as into the BETA-compiler, an enquiry mechanism which supplies attributes of INTEL statements with appropriate machine-dependent values, or even selects, if necessary, special compilation subroutines, completely oriented toward an object computer. Parametrization makes INTEL a machine-oriented language in which it is, in principle, impossible to write a machine-independent program but which allows the development of general compilation algorithms applicable to any pair taken from the "direct product" of the class of source languages and the set of object computers. Parametrization is applied to many features of source languages (precision, transput, memory control, representation of objects, etc).

LECTURE 3

DECOMPOSITION AND SYNTHESIS IN THE BETA SYSTEM

3.1. Introduction

We shall outline below a sketch of a set of rules which prescribe to linguists how to prepare a source language for its implementation in the BETA system. These rules suggest those language patterns which the linguist has to recognize in the language, and how to present them in the required stylistic arrangement. On one hand, these rules prescribe to a linguist to recognize given universal constructions in a language, and to identifie possible differences between the complete and general universal construction and its particular use in the language being considered. On the other hand, some metaformalism is recommended which enables the linguist to describe any unique language features which cannot be recognized as of one of the given universal constructions.

3.1.1. Grammatical elements. A source language is desribed over a set of grammatical elements, their attributes and relations between elements. *Grammatical elements* consist of notions, positions, lexemes and symbols. When speaking of a grammatical element with respect to some program we actualy speak of various occurences of the grammatical element in the program.

We take the liberty not to redefine such well-known concepts as *notion* (as in Algol 68), *lexeme*, terminal productions of notions and lexemes as substrings of a source program text, and the relation "to be a (direct) constituent of". Concepts of a (name) scope, defining and applied occurences of a lexeme are also supposed to be known. A few

words about *positions*: a position in a production rule for a notion is a named "place" which can be replaced by a direct constituent of that notion. This constituent of the notion is called a *substituent* of the position. The relation between positions and their substituents is similar to that of field selectors and modes of "their" values in ALGOL 68 structures.

Another considered relation between notions is the "*brotherhood*" relation which connects neighboring elements of any uniform list of notions.

Given a program, finding all notion and lexeme occurences, and their terminal productions is called program parsing. A program text in which each terminal production is embraced by brackets identifying its corresponding notion is called *parsing string* (Fig. 4). The parsing tree concept is introduced in a conventional way; with the only exception that the brotherhood relation is represented by arcs of a special kind (Fig. 5).

Grammatical elements may have *attributes*. Each attribute is a set, each element of that set is called an attribute value.

A *semantic lexeme* is a lexeme to which the language description prescribes finding its defining occurence in some scope in the program. The defining occurence of a semantic lexeme is a source of attribute values of that lexeme. By definition, these attributes are attributed to all applied occurences of the semantic lexeme. As in ALGOL 68 we call this process *identification*.

For an occurence of a notion in a program (parsing tree) the language rules may specify values of its attributes as a function of attributes of its constituents. This inductive process is called *semantic induction*. Identification and semantical induction together are called semantic analysis.

A parsing tree with completely defined values of all attributes for all its vertices is called an attributed parsing tree (Fig. 6).

3.1.2. Decomposition and synthesis passes. Decompositior and synthesis is accomplished in the following five passes. The first pass - lexical analysis:

- lexical table construction
- elimination of lexeme productions from the program text
- parentheses balance checking
- (preliminary) scope identification

The second pass - syntactic analysis:

- parsing string construction

The third pass - parsing:

- conversion of a parsing string into a parsing tree

The fourth pass - semantic analysis:

- finding defining occurences of semantic lexemes
- identification
- semantic induction
- parsing tree rearrangement (if necessary)

The fifth pass - *synthesis*:

- conversion of an attributed parsing tree into an INTEL program

Each pass is characterized by some universal mechanism of processing compiled program. If a mechanism allows, then the mentioned "obligatory" pass functions can be loaded by some "looking ahead" additional actions thus shifting some functions of subsequent passes to (so called *advancing*) previous ones. This makes all the compilation process more efficient.

3.2. Lexical analysis. Executive procedures

3.2.1. Lexical automaton. The lexical analysis is performed by a *lexical automaton*. This is a deterministic finite automaton. Its input alphabet is the terminal symbol alphabet of the language; its output alphabet is the set of possible lexemes. The automaton performes proper and

attached functions. *Proper functions* are state transitions, reading input and writing output symbols. *Attached functions* are additional operations on tables and a context. A *context* is a collection of named memory registers accessible to the compiler and used globally when writing executive procedures (see below).

Some attached functions appear "automatically" in the BETA compiler (standard operations on tables, paired delimiters checking etc.). Other functions reflect specific properties of a language and are written by the linguist in the form of executive procedures.

3.2.2. The *executive procedure* (EP) is the main form of describing compilation properties of a source language, so guide-lines for their stylistic arrangement will be given in more detailes. An EP is written for a grammatical element of a language and is used in some phase of the BETA-compiler. The EP is responsible for processing the information related to any occurrence of the corresponding element. Some EPs are related not to element occurrences but to the element itself. These EPs process corresponding tables. Finding an occurrence of an element in the program initializes a call and execution of the corresponding EP. The element's attributes play the role of actual parameters of the EP. Besides attributes, context registers and constituent attributes are usually accessible to the EP. As a rule, EPs operate locally and are non-recursive.

EPs related to a pass are combined in libraries. Organisation of the library and EP calls is an organic part of the corresponding compilation mechanism of the pass. There exist universal EPs which are written for a family of languages and compose a part of the BETA compiler. However, unique EPs can be written specifically for a particular source language.

In the lexical phase, EPs can appear only for optional actions connected with finding defining occurences. All

other attached functions are assimilated by the synthesizer of the lexical automaton.

3.3. Syntactic analysis and parsing

3.3.1. *Grammatical structure.* Syntactic analysis is performed by a grammatical structure which is a deterministic automaton with external and internal states and a memory for a so-called hidden alphabet. The input alphabet is the lexical alphabet; the output alphabet is the union of the lexical alphabet and an alphabet of paired delimiters marked by language notions.

The meaning of *internal states* and the *hidden alphabet* are as follows. Each internal state is connected with a concrete memory element (counter, push-down or queue) from which a hidden alphabet letter can be read. If the automaton is in an internal state its function is defined by the hidden alphabet letter read from the memory element corresponding to the internal state. Proper functions of the grammatical structure are state transitions, storing hidden alphabet letters into memory elements, reading input symbols and writing output symbols, respectively. Attached functions are context processing and possible EPs for optional actions.

The grammatical structure is synthesized by a multistep metaprocessor from the initial context-free grammar of the language.

3.3.2. Parsing is performed by a universal procedure for conversion from a parsing string into a parsing tree.

3.4. Semantic analysis and synthesis

3.4.1. Semantic analysis is almost entirely based on EP libraries. Table processing EPs work for finding semantic

lexemes. Lexeme EPs perform identification. Notion EPs provide semantic induction; and special notion EPs are used if some parsing tree rearrangement is required.

3.4.2. Synthesis, as well as the previous pass, is completely based on an EP library. Its specific feature is the existence of large scale iterative or recursive EPs connected with a table rearrangement or processing large fragments of the parsing tree (e.g., flattening the expressions).

Below, a brief review of source language description rules will be given. This sketch will be illustrated by fragments of the Pascal language description which are useful for the demonstrational example from the first lecture.

3.5. Lexical information

This part of a language description contains all information which is necessary for a lexical automaton synthesis. Most of the information is given in list form (Fig. 10):

language basic symbol alphabet
list of paired delimeters
key-words list
reserved identifiers list
lexemes classification and rules for selecting lexeme terminal productions from a program text
scope finding rules
list of semantic lexemes and their attributes

3.6. Syntactic information

The source language syntax is described in a form of a regularized context-free grammar, which uses special notations for positions, optional positions (with possible empty substituents) and uniform notion lists. Such regularization

retains only essential notions in a parsing tree and makes it easier to convert from the grammar into formats for representation of parsing trees (Fig. 11).

Each notion production rule consists of a *leading definition* supplemented by a notion *position expansion*. The left side of the leading definition consists of the defined notion; the right side is a word composed of positions, possibly separated by delimiters. Positions show the number and order of notion constituents. Positions may have empty substituents but no alternatives are allowed in the leading definition.

A position expansion contains in the left side the expanded position and in the right side a list of alternative notions being substituents of this position. Each alternative may contain terminal symbols but only one notion.

Pseudonotions can be used as contraction rules. A pseudonotion is a name for a list of alternative notions. Pseudonotions do not appear in a parsing tree.

An example presented in Fig. 12 demonstrates a relation between a fragment of a language grammar and a corresponding fragment of a data base for parsing tree presentation; the node declaration are given in ALGOL 68.

An alternative in the right side of a position expansion can be a uniform list of notions which will appear in the parsing tree as a chain of notions constituting a brotherhood.

There are two versions of a language grammar, each corresponding to an initial parsing tree and an attributed (possibly rearranged) parsing tree prepared for synthesis.

3.7. Semantic information

3.7.1. Notion attributes specification. There is a distinction between attribute notations, notations for their values, and values themselves. There is also a distinction

between attributes which accompany each notion occurence and attributes which are stored "in one copy" in tables. All this "format" information is arranged as a list attached to each notion.

3.7.2. Executive procedures of the semantic analysis. For each grammatical element occurring in the initial parsing tree an executive procedure is written. It defines all attributes of all occurrences and prepares the attributed parsing tree for synthesis.

Possible actions:

- Finding a defining occurrence of a semantic lexeme. Extraction of attribute information from constituents of a declaration and storing it in a table. Parsing tree reduction (Fig. 13).
- Identification. Scope finding. Searching in a corresponding table of semantic lexemes. Copying attributes and transferring them into an applied occurrence.
- Semantic induction. Computation of values of a notion attribute as a function of constituent attributes. Checking consistency of position and substituent attributes.
- Coersion. Insertion of new vertices into the parsing tree as a result of a coersion.
- Generalization of notions. Notions n_1, n_2 become a more general notion $g(N)$ with an attribute N with values n_1 and n_2, $g(n_1)$, $g(n_2)$.

3.8. Information for synthesis and code generation

3.8.1. Executive procedures for synthesis. For each notion occurring in the attributed tree and for some lexemes (Fig. 14), an executive procedure is written:

Possible actions:

- Expansion of a notion as a piece of INTEL program
- Selection of a most appropriate INTEL form as a func-

tion of notion attributes (selective synthesis)
- Forming fragments in an INTEL program
- Constructing a block nesting graph
- Constructing a mode graph

3.8.2. Non-interpreted INTEL operations. Special EPs synthesize COMMAND statements in formats prescribed by a list of non-interpreted operations.

3.8.3. The catalog of the run-time library contains formats of call sequences and general information on access to the library file and, if necessary, residentation rules.

3.8.4. Executive procedures of the generation of non-interpreted operations. Each contains a template of machine coding which expands a given operation; are supplied with predicats which are evaluated over values of attributes and provide a selective generation.

A production version of a language description has to be supplemented with precompilation editing facilities, error message system and compilation rules for processing a syntactically incorrect text (emergency syntax).

3.9. Discussion

The rules of language description just presented are based on a series of principles which could be proved or disputed only by experience. Nevertheless, in any case it is worthwhile to identify some of them.

In general, a language description is strongly divided into decomposition rules (analysis) for a source text and synthesis rules for an INTEL program; they "communicate" through a parsing tree. The culmination of a language description is the synthesis EPs. The choice of language notions and their attributes has to be subjected to providing

a flexible and selective synthesis of an INTEL text which implements an occurrence of a given notion into a program.

Synthesis rules are of an operational character. They prescribe what INTEL primitives (and in what combination) should be used in order to implement a given notion. Another very important part of synthesis EPs are the already mention ed predicates which are evaluated over attributes values.

The operational character of EPs (or transducers in theoretical terminology) corresponds better to a compilation technique and gives more flexibility in balancing compilation and meta-compilation processes during the implementation of a language in the system.

A key problem for a linguist is a choice of language notions and their attributes. This choice predetermines the contents and functions of the EP libraries. The relations between notions and attributes are quite flexible; a notion N with an attribute A = {a,b,c} can be converted into three notions, Na, Nb, Nc without attribute A.

An inverse conversion is also possible and is actually used to fit some universal synthesis EPs better. The treatment of a language concept as a notion characterizes its stability, separability and generality. On the other hand, the variety and richness of attributes of a notion characterize a variablity of the notion in its occurrences and in different languages. We would, however, look at notions differently. A notion is something that can be found in a text - be separated from a string. A notion is a compound object which have constituents and itself can be a constituent. The more these synthetic and analitical approaches coincide, the more phrase structure of a language coincides with its semantic structure. Possible differences between these two approaches to the choice of notions are smoothed in the BETA system by the semantic analysis phase, where a rearrangement of the parsing tree is possible.

This remark confirms one more principle of an organization of information on a language and decomposition phase.

While reducing the variety of grammatical elements to a minimum and attempting to make compilation mechanisms as uniform as possible we, nevertheless, retain at appropriate points of the BETA compiler "elastic" constructions which assimilate peculiarities and provide interfaces: optional advancing the compilation process, context registers, memory and hidden alphabet in the grammatical structure and just mentioned tree rearrangement.

The partition of the decomposition and synthesis phases into five passes (distinguished not only by functions but also by mechanisms of information processing) is done in the trial implementation of the BETA project mainly for methodological purposes. We wanted to identify and purify a uniform processing mechanism as such. On the other hand there are some chances to believe that such an approach can be technologically well motivated. Two passes with uniform mechanisms in each can be more efficient than one pass with complicated mechanics of alternating processing functions. If, however, a combining of functions causes no difficulties it can always be done by a metaprocessor by means of already mentioned compilation advancing.

LECTURE 4

OPTIMIZATION AND CODE GENERATION IN THE BETA SYSTEM

4.1. Collection of the optimizing transformations

The choice of a set of optimizing transformations guided by the desire to cover as many as possible of the existing and well-established manual methods of program improvement. It should be noted that it was sometimes possible to cover some special techniques by more general transformations.

A collection of selected machine-independent optimizations will be listed below.

4.1.1. Transformations connected with implementation of procedure calls and parameters passing. These transformations are based on a differentiation between recursive and non-recursive procedure calls, on a selection of the most appropriate method of parameters substitution (in particular, performing some substitutions during compilation) and on the elimination of unnecessary copying of local variables of recursive procedures. These transformations require a knowledge of the interstatement and interprocedural information flow as well as of procedure call nesting.

4.1.2. Transfer of computations to the compilation phase. This group of optimizations includes:

computation of constants

removal of constant references

replacement of a destination with the source x and subsequent elimination of the assignment $y := x$, if x is the only source for y.

removal of if-statements with constant if-clause or with identical alternatives.

In general, for these transformations an information flow analysis is required.

4.1.3. Unloading of a repeated component of the control flow graph (for statement body, recursive procedure body). This transformation consists in a transfer of those computations, which are constant during each execution of the body, to the entry or exit points of the body. These transformations require an analysis of control as well as information flow of the repetitive component.

4.1.4. Elimination of redundant identical expressions consists in finding and eliminating all but one occurrence of identical (both statically and dinamically) expressions. The transformation also requires a total control and information flow analysis. This optimization covers a variety of special optimizations related to parameter substitution, subscript evaluation etc.

4.1.5. Reduction of operator strength. When possible, a repetitive computation of powers and multiplications for regularly advancing arguments is replaced by a computation "in differences", that is by multiplications and additions respectively. It includes as a special case an incrementing of linearized subscripts. This transformation requires an information flow analysis inside repeated components.

4.1.6. Program garbage collection. This transformation consists in finding and removal of value sources whose values are not used and statements which have no predecessors. By definition, this optimization requires a total information and control flow analysis.

4.1.7. Global memory economy. This consists in introducing a kind of EQUIVALENCE statements for those variables which, belonging to one same scope, can be allocated, however, into the same memory region.

4.2. Analysis

As the list of transformations shows, there exists some global information about program data and control structures which is equally necessary for various transformations. In order to speed up optimization transformations it is crucial to be able to obtain easily this information. This is achieved by the following methods.

First, INTEL structures have been chosen such that they facilitate optimization analysis and transformations. Textual and label control sequencing of source language statements is replaced in the INTEL language by explicit arcs of a control flow graph. Explicit information connections can be formed between results and arguments of INTEL statements belonging to a linear component of a control flow graph.

Secondly, useful information which is not explicitly presented in an INTEL program or is too overloaded by unnecessary details is collected and presented in a concise form in special tables or lists (so-called *shadows*) during the analysis phase preceding the optimization itself. As an example, a procedure call nesting graph can be mentioned. This graph is used for deciding on which procedures are recursive procedures and for correction of the control and information flow graphs. Another example is a table of arguments/results of INTEL program statements and larger constructions - hammocks and zones (see below).

Thirdly, a special fast algorithms have been developed which search for useful subgraphs of a control flow graph-*zones* which represent repeated components and *hammocks* which are connected with remaining part of the graph only through its two vertices: one for ingoing arcs and another for outgoing arcs. During the analysis phase the control flow graph of an INTEL program is decomposed into an hierarchy of nested hammocks and zones (Fig. 8).

4.3. Factorization

Realistic implementation of some global optimizations is hampered by the fact that the time and space complexity of the corresponding algorithms is unjustifyably high. The problem scale can be greatly reduced if a method called *factorization* of the control flow graph is applied. A factorized graph is considered as a hierarchical sequence of nested fragments such that at the lowest level of factorization optimization algorithms are applied only in within fragments which at the next level will be considered as elementary objects. In the BETA system such fragments are the already mentioned hammocks and zones.

Fig. 15 shows how the optimization algorithms are arranged in the sequence of compilation passes of the BETA compiler.

4.4. Preliminary code generation

4.4.1. This phase translated INTEL programs in the INGEL (INtermediate GEneration Language) language, collects the information needed for fast register allocation, allocates memory for some working values, and collects information on the use of constants and other objects of a compiled program.

The *INGEL language* is close to an object computer language. INGEL statements are:

- object computer instructions whose address part contains references to tables of information about program data;
- special *generation statements* subjected to an interpretation at the second generation phase. One of the purposes of these statements is a final choice between one of the several implementations of INTEL constructions prepared in advance during the preliminary generation.

4.4.2. Allocation of fast registers is an important factor greatly influencing the object program efficiency. The optimization phase selects a set of program objects potentially suitable for implementation through fast registers. At the preliminary generation phase a provisional register allocation is made and a corresponding object attribute gets the value true. Non-allocated objects get false as the attribute value. "Marginal" objects are implemented both ways postponing the final choise to the second phase (Fig. 16).

4.4.3. The information about program data use serves the following purposes:

- to locate data related to removed program garbage,
- to locate constants which can be allocated directly in address part of machine instructions;
- to locate arrays and structures which do not require descriptors.

4.5. Memory allocation

There are two strategies of memory allocation available in the BETA compiler. Each of them is selected by specifying the corresponding compilation mode.

The first strategy is a conventional one, when automatic local data are automatically allocated relative to the block base upon entering the block.

The second strategy is one where all data are considered to be as global as possible. If a block contains an array declaration, the actual memory allocation for the components is done inside the block but desciptors are allocated in an embracing block.

This "globalization" principle is extended also to procedures except for recursive procedures co-routines and "parallel" ones. In order to distinguish between those recursive, parallel and co-routine procedures are termed "procedures" and the others are termed "*subroutines*". It means

that subroutine parameters and static local objects are allocated at least in a block in which the subroutine is declared. Globalization, however, does not prevent the allocation of local data of statically parallel subroutines in shared parts of the memory.

This strategy requires more memory but much less memory control operations.

At the end of the preliminary generation all program constants are converted into an object code with subsequent global economization using a hashing technique.

4.6. The coding of subroutines and procedures

An INTEL procedure call fragment consists of the following sequence of statements

1) CALL START

2) optional statements calculating bound-pairs of actual parameters

3) optional statements calculating values of actual parameters and containing as many LOAD statements as the number of parameters

4) CALL

The property "to be a subroutine" is stored as an attribute value.

4.6.1. Subroutine call coding.

1) CALL START allocates the space in the stack for the subroutine result, if any, otherwise it causes an empty action.

2) calculating statements are translated into the INGEL language in the conventional way.

3) LOAD is converted into an assignment which assigns to the formal parameter variable

- a thunk (a procedure calculating an "address") if the parameter is called by name

- a reference if the parameter is called by reference
- a value (with memory allocation if necessary when the parameter is called by value.

4) CALL is implemented in the conventional way as a subroutine jump.

4.6.2. Procedure call coding.

1) CALL START, in addition to the space for a possible result, allocates in the stack the space for the procedure descriptor which will contain the information about actual parameters of the current call.

2) Calculating statements as above

3) The LOAD statement are converted into instructions which load different fields of the procedure descriptor with:

- information about the thunk and its environment if the parameter is called by name
- reference to an actual parameter
- address of the space allocated in the stack for the value of the formal parameter (if it is called by value) with additional work on an array descriptor (if the parameter is an array) and with subsequent assignment of the actual parameter value

4) CALL implementation as above.

4.7. Final code generation

The final generation is a straight forward action. References to object and constant tables are replaced by addresses relative to an appropriate base. Special generation statements are interpreted and unused alternatives are removed from the code. The object program is assembled and either relocated to a starting position for immediate running or stored as a load module (Fig. 9).

LECTURE 5

COMPILER WRITING SYSTEMS AS A FACTOR OF UNIFICATION AND COMPARISON OF PROGRAMMING LANGUAGES

5.1. Introduction

Let us first formulate a few thesis which reflect our view of the development of programming languages.

In our opinion the current moment marks a breakpoint in a historically observable period of the development of ±25 years.

The past left us with a family of so-called "universal" programming languages starting with FORTRAN. These languages are devoted to professional (system) programmers as well as for non-professionals (users) and, in some sense, are inconvenient for both groups. Moreover, most of these languages have inherited many defects caused by our insufficient knowledge at the time they were created. In spite of these defects, human habits and conserving factors of software accumulation will result in these languages still being in heavy use in 1980's and some of them even in 1990's. The only positive factor is that the family of these language is well outlined, and is becoming more and more isolated in its stability from general progress in the programming though endowing it with some fundamental concepts.

These languages have a lot in common, although the commonality may be difficult to discover due to a descriptive variety and the above mentioned defects. Nevertheless, some unification of these languages is highly desirable for more economical design of instrumental tools, or even integrated compiler writing systems serving many languages. This unification is no less important for teaching purposes. We must

have it in order to make the education of future programmers (and users) really multilanguage instead of feeding them forever with mother milk of a single language. The only way to achieve this is to teach them concepts, not a language, and the ability to find these concepts in any language. Such an over-language position will make, in our opinion, a person much more responsive to a change of a language environment.

In future years the process of creation of languages, in our opinion, will definitely grow, but in patterns reflecting similar linguistic processes in natural languages when they are used for a professional activity. These processes are characterized by the appearance of numerous, rather strict, but relatively stable professional jargons, accomponied by a slow but steadily growth of the use of a small number of "ground" languages serving as a base for formation of jargons and a tool for general communication (as English, for example).

Similarly, the future of programming will be characterized by the design of many specialized languages for communication with a computer in a variety of stable applications. These languages, however, will be implemented by means of a small family of professional programming languages which, besides their universality and richness, will possess other properties supporting their professional use.

The design of languages of both kinds requires a deep and real knowledge. These languages have to be, in a definite sense, ideal and unmistakable. Unification of language concepts, and its projection on real languages, gives us some knowledge which helps us to design new languages with many more deliberations.

Our experience of the development of the BETA system shows that the design and implementation of compiles writing systems is, equally, a powerfull source and consumer of language unification. The necessity to design and implement a number of universal mechanisms, oriented to different languages, form an objective and constructive basis of uni-

fication. A simultaneous analysis of several different languages is no less stimulating process.

The second lecture on the INTEL language was, actually, an attempt to present a brief account of a basis for a semantic unification of algorithmic languages. In the first part of this lecture we shall supplement this treatment with a discussion of how this unification can be advanced "one step ahead" by an identification and isolation (at the parsing and semantic analysis phases) of above mentioned universal grammatical elements for which universal (i.e. serving several languages) synthesis procedures can be developed.

In the next part of the lecture some very brief comparative analysis of some new algorithmic languages will be presented. We observe that new languages, being designed in a relatively dense conceptual atmosphere, actively borrowing from each other the best variants of similar constructions, are converging more and more (at least, over some collection of basic constructions constituting a kind of "golden book" of programming). We will present also, as we understand them, main criteria for evaluating an algorithmic language.

5.2. Universal executive procedures for synthesis

A universal EP of synthesis of, say, for statements, is designed in its maximal generality when the definition is taken in a complete form (as in ALGOL 68):

for i from L by H until U while B do

Ajusting the general construction to its partial versions in different languages is performed by several methods.

5.2.1. Generalization. A partial construction is "streched" up to a general one introducing supposed elements by "de fault" during the semantical analysis. For example

if a language contains only headings in the form

for i := L until U do

then some constant constituents are inserted with a result as

for i := from L by 1 until U while true do

This method is straightforward but inefficient because it makes a simple construction more complicated. It is justified if the synthesis procedure is selective enough to recognize constant constituents.

5.2.2. Attributization. We do not introduce any dummy constituents, but form special attributes an analysis of which will provide the synthesis procedure with necessary selectivity. For example, a control variable i of a for statement can have attributes "i is local in the loop body" or "i can be assigned by a value in the loop body". A value of such attribute is a language constant and is introduced into the construction during parsing or semantic analysis.

5.2.3. Projection of a covering metaform. We have just noticed that partial constructions of a source language look like some of their constituents or attributes are language constants - the same for any use of the construction in the language. Then, if a synthesis procedure is well structurized, i.e. a part of the procedure respondible for processing the corresponding component (constituent or attribute) is easily separated from other parts, then an adaptation of the general procedure to its partial form is possible at a metaprocessor level, during an implementation of the given source language. This adaptation consists of checking the variability of the language component, and deleting program branches which correspond constant con-

structions. A similar technique is already applied in some macroassemblers and can be one of the most effective adaptation methods.

5.3. Criteria for evaluation of programming languages

Language design processes have led to the development of several language evaluation criteria. The relative importance of these criteria at different times has been estimated differently.

These criteria are

- universality, which dominated at early 60's and greatly influencing ALGOL 60 and PL/I. This criterium resulted in the design of some very general constructions (A for statement and procedures parameters in ALGOL 60);
- independence of a specific computer model or installation;
- teachability, or more specifically,
- modularity, which allows a programmer to work only with a necessary (or known) fragment of a language;
- orthogonality (in the sense of ALGOL 68 philosophy);
- security from errors, or at least a desire to discover most of errors statically; usually supposed some redundancy which is also helpful psychologically;
- object program efficiency, or, to be more correct, the possibility of writing efficient programs. Sometimes this requirement is expanded up to a necessity of a complete control over object machine features;
- convenience of structuring and modification of a program, especially important for collective efforts for a program design.

It seems to us that the last criteria are considered now more important.

Our analysis has been devoted mainly such languages as ALGOL 60, PL/I, SIMULA 67, ALGOL 68 and Pascal. The limits

of the lecture prevent us from exposing all the analysis. We restrict ourselves to a general conclusion, and - in order to give you a flavor of our approach - some discussion on the object part of the languages.

5.4. Data types

5.4.1. Arithmetic types. The most flexible machine-independent precision specification exists in PL/I; Pascal "subrange type" is also interesting, it can be understood as short or long integer specification and is easily generalized on fixed point numbers.

5.4.2. Identifier type as introduced in Pascal

```
side = (bottom, top, left, right)
```

is now an undisputable component of new languages. It does not add much to the expressive power of a language but stimulates a clear style of writing and simplifies future modifications. In ALGOL 68 it would be written as

```
int bottom = 1, top = 2, left = 3, right = 4
```

which is less secure and concise. Identifier types allow to introduce binary strings in a more natural way than in other languages (set types).

5.4.3. Arrays. Exclusion of dynamic arrays from the Pascal language is, of course, an extreme decision. Nevertheless a more explicit syntactic distinction between dynamic and static arrays is highly desirable. On the other hand further generalization of a dynamic array (up to ALGOL 68 flexible array) as a pointer which can refer to arrays created by local generators is quite possible.

5.4.4. Records (structures) and unions. ALGOL 68 unions seem not to be a wholly successful construction. They require the most complicated coersions, and cause some difficulties in separate compilation. This is an appropriate example of insufficient redundancy. Pascal "variant records" look more transparent and convenient. The most developed concept, however, is presented by SIMULA 67 classes which have more clear and logical structure, are ideally suitable for modularization of programs and more secure due to an obligatory initialization. PL/I trick of so-called left-to-right correspondence is a very powerful but very unsecure tool which approaches PL/I to typeless languages.

5.4.5. Pointers. Distinguishing qualified and nonqualified pointers influences both error protection and the compactness of notation; languages also differ from each other by sets of objects referrable by pointers. Most languages prohibit references to constants (except a convenient solution of ALGOL 68 which only prevents an assignment to such references). A general tendency is to restrict the use of pointers as an insecure low level tool (SIMULA 67 allows only one-level references to texts and classes; Pascal prohibits pointers to explicitly declared variables and their components). On the other hand, both languages have "with (connection) statements" which give in most important cases, a non-variable reference to components of compound variables.

5.4.6. Most languages with typing require a specification of formal procedures. The most convenient method of specifying formal and variable procedures in that of PL/I; it is possible to say that it just reflects a minimal programmer's selfdiscipline with respect to the degree of detail of the specifications.

The multi-entry procedure feature of PL/I seems, however, much more disputable. Without special analysis, it reduces efficiency and, in some cases, security.

Recently, the concept of coroutines has become more and more appreciated. It seems that a language which treats procedure calls and jumps into coroutines "equally" would be rather interesting.

5.5. Name declarations

By "names" here we mean identifiers explicitly declared in a program as existing and possessing some "internal objects" in some scope.

5.5.1. Constants. Most recent languages have provided means to name constants. This linguistic separation of named constants from variables enhances security as well as efficiency. Pascal is a language with an extensive mechanism for constant definitions.

5.5.2. Scopes. Traditionally, declarations play a dual role: they, first, introduce a new name (notation) in some "scope" and, secondly, create a new object (embodied in a memory segment) which is characterized with its "life-time". All languages with pointers permit one to distinguish these two kinds of scopes. In particular, they permit one to create "anonymous" objects.

The richest variety of scopes is presented by PL/I with its traditional local variables, global variables (very useful for separate compilation und actually superseding more powerful but less elegant *own* concept of ALGOL 60), and availability of field names as variables. SIMULA 67 has a most elaborated concept of restoring (if necessary) a definition scope for "shadowed" identifier-by means of qualified remote identifiers. The concept unfortunately lacks a restoration of the uppermost "standard" level of naming. This, however, could be achieved by an inclusion of standard declarations in some standard class (as it is possible in PL/I).

5.5.3. Elaboration of declarations. So far there is no uniform approach to the choise of a point for the elaboration of declarations. However, the general tendency is to allocate, in the "begin" part of a block, static local objects and descriptors of dynamic objects. Allocation of memory for dynamic objects proper is postponed until a point of getting a value of the object (by "generators" or "allocation" statements).

5.5.4. Initialization. Most languages do not require obligatory initialization. The authors of SIMULA 67 assure an obligatory initialization of all variables. This raises security considerably. In our opinion, an optional initialization as presented, say, in PL/I is quite convenient. As to the obligatory initialization, it could be justified for "control" objects only (pointers, variable procedures etc.). In such cases it seems desirable that an "empty" reference should refer to some protected memory, and should cause an interrupt in the case of an illegal fetching.

5.5.5. Based objects. Usually special statements are used for an allocation of based objects (generators in ALGOL 68, ALLOCATE statements in PL/I). A main problem, still not satisfactory solved, is that of releasing previously allocated memory. ALGOL 68 and SIMULA 67 rely on a garbage collector; PL/I has facilities allowing a programmer to control memory releasing (mechanism of areas and explicit FREE statement). Both approaches have deficiencies: the PL/I solution makes control on pointer use impossible; the garbage collector approach is straightforward but often inefficient. In our opinion some compromise and combined solution would be more appropriate (areas, with "offsets" attached to them, local generators and restrictions on the use of pointers).

5.6. Resume of the comparison

Below we shall make a resume of our comparative analysis. The languages will be ordered according their "distance" from an ideal; languages grouped between semicolons are, roughly, at an equal distance.

Universality: PL/I, ALGOL 68, SIMULA 67; Pascal

Teachability: Pascal (compact and modest);
SIMULA 67 (logical and not very big);
ALGOL 68 (very regular but big);
PL/I (big and not sufficiently regular).

Modularity in teaching: PL/I, SIMULA 67, PASCAL; ALGOL 68

Orthogonality: ALGOL 68; PL/I; Pascal, SIMULA 67

Security: ALGOL 68, SIMULA 67, Pascal; PL/1

Possibility to write efficient programs: Pascal; ALGOL 68;
PL/1, SIMULA 67 (the last two require a large run-time library of control routines)

Modularity of programs: SIMULA 67; PL/1; ALGOL 68; Pascal

Ease of modifications: Pascal; PL/1; ALGOL 68; SIMULA 67

5.7. Conclusion

Don Knuth in his recent and previously referred paper [7] has written: "At the present time I think we are on the verge of discovering at last what programming languages should really be like. I look forward to seeing many responsible experiments with language design during the next few years; and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages). Furthermore, I'm guessing that people will become so disenchanted with the languages they are now using - even COBOL and FORTRAN - that this new language, UTOPIA 84, will have a chance to take over. At present we are far from that goal, yet there are indications that such a language is very slowly taking shape."

We shall not even try to search for a better expression of the reasons which motivated us to start our research into many-language systems (which in its turn have led us to a comparative study of languages). Noting a mixture of an irony and enthusiasm in Knuth's proposed name of this ideal language, we would like to add that this attractive utopia can become a reality of the future if it will be created with a full use of the reality of the present. This use consists of, on one hand, an identification of undisputable language patterns and determination of their real role in a language and, on the other hand, deep critical analysis of all language defects regardless of their origin.

A more specific goal of our presentation is a desire to stress that potential and actual role which compiler writing systems can play in finding and purification of language essentials. These systems, by virtue, oriented to universal and applicable to many languages procedures, have to cope with any detail of a language and, thus, become, at one time, a very strong and very sensitive judge of language properties.

Moreover, a materialization of such universal procedures requires a development of several intermediate program forms. So far, determination of such intermediate forms took place mainly on a metalanguage level related to the syntactic properties of a language. The many-language character of the BETA system led us to a development of a variant of an intermediate form which has to serve as a semantic basis of a family of source languages.

At the end of this lecture and all the course the author wishes to mention one more citation. At the IFIP Congress 75 in Stockholm professor G. Seegmüller in his very instructive talk has referred to a problem of development of production compiler writing systems as one of most important problems in modern system programming. He said [8]:

"... 1. There does not exist a design (let alone an implementation) of a compiler compiler which, being fed with syntax, semantics, a target system description, and constraints for the compiler and its object programs, will deliver a pro-

duct which is competitive in the usual commercial sense. Currently, compiler compiler outputs will very likely be better as far as reliability is concerned, but they will be inferior with respect to most of the other performance specifications."

This problem statement is, really, a challenge for system programmers and computer science researchers. Our experience has attracted, on one hand, our attention to many problems about which we had no idea at the start of the BETA project. On the other hand, this experience made us optimists with respect to a solution of the problem stated by Seegmüller as well as with respect to a progress in the development of programming languages described by Knuth. If this feeling of optimism can be shared by the audience the author will be able to consider his task fulfilled.

Acknowledgements

The BETA project is being accomplished as a part of R & D program in interest of ES EVM ("Ryad") series of computers. The development team consists of (in order of the Russian alphabet) V.G.Bekasov, A.A. Baehrs, V.V.Grushetsky, A.P.Ershov, L.L.Zmiyevskaya, V.N.Kasyanov, L.A.Korneva, S.G.Pokrovsky, I.V.Pottosin, A.F.Raz, V.K.Sabelfeld, F.G. Svetlakova, G.G.Stepanov, M.B.Trakhtenbrot, T.S.Yanchuk.

The material of the lectures is, mainly, compiled from publications [2,3,4,5,6] and some working materials of the project. The demonstrational example has been prepared by V.V.Grushetsky, S.B.Pokrovsky, V.N.Kasyanov and G.G.Stepanov.

The author, still preserving his full responsibility for possible deficiences of the text, has a unique opportunity to thank his collegues C.A.R.Hoare, M.Griffiths, D.Gries, K.H.A.Koster and W.Wulf who kindly agreed to edit the English draft of the materials.

References

1. A.P.Ershov. A multilanguage programming system oriented to language description and universal optimization algorithms. Proc. of the IFIP Working Conference on ALGOL 68 Implementation, Munich, July 20-24, 1970

2. A.P.Ershow, S.B.Pokrovsky, V.K.Sabelfeld. Internal language in a many-language programming system as a mean of a formalization of the semantics of source languages. A talk to a conference on formalization of the semantics of programming languages. Frankfurt/Oder, September 23-27, 1974. Elektronische Informationsverarbeitung und Kybernetik, 4-6/1975

3. V.V.Grushetsky, A.P.Ershov, S.B.Pokrovsky, I.V.Pottosin. Decomposition, synthesis, and optimization methods in a many-language programming system. Ibid

4. G.G.Stepanov. Generation methods in a many-language programming system. - Ibid

5. V.V.Grushetsky, A.P.Ershov, S.B.Pokrovsky. "Prochrustian bed" for source languages in a many-language programming system. Acta Polytechnica, ČVUT, Phaha, 1975 (in print)

6. A.P.Ershov, S.B.Pokrovsky. On programming languages unification. Cybernetics problems, Moscow, 1976 (in print)

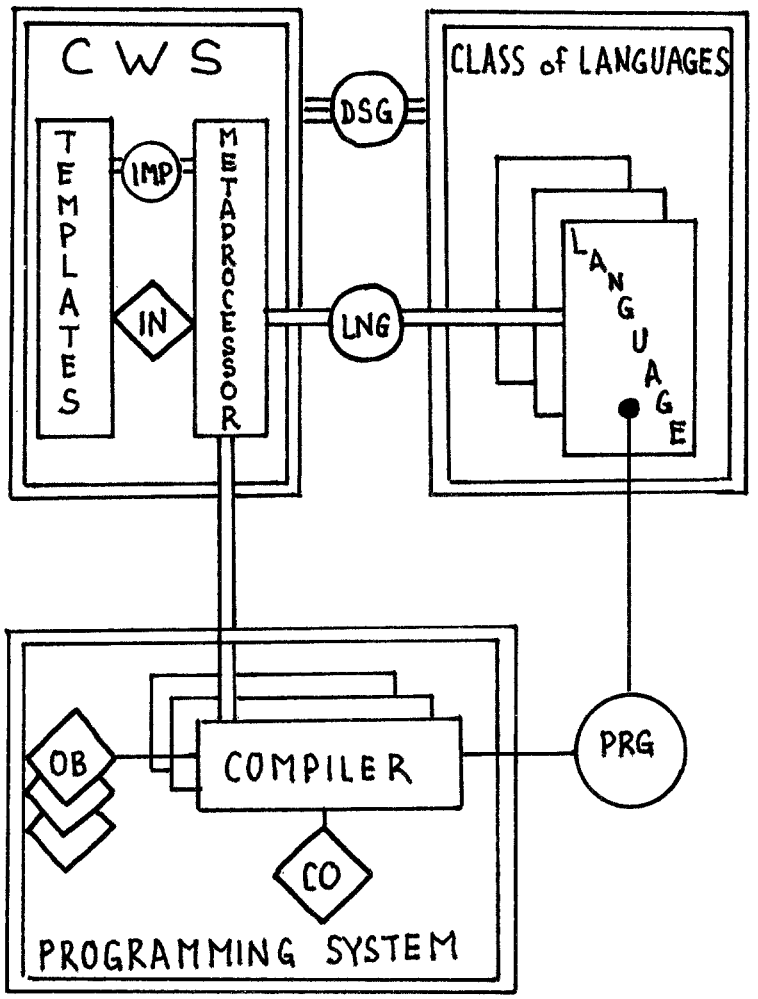
7. D.Knuth. Structured programming with go to. Computing Surways. No.6, 1974

8. G. Seegmüller. System programming as an emerging discipline. Invited talk, IFIP Congress 74, Stockholm, August 1974

9. A.P.Ershov (Ed.) ALPHA - an automatic programming system. Academic Press, London, 1971

Index of terms

advancing 3.1.2.
attached function 3.2.1.
attribute 3.1.1.
BETA compiler 1.2.
BETA project 1.1.
BETA system 1.1.
brotherhood 3.1.1.
compiler writing system 1.1.
context 3.2.1.
control flow graph 2.2.2.
designer 1.1.
dynamic enclosure graph 2.3.1.
executive procedure 3.2.2.
factorization 4.3.
fragment 2.2.2.
generation (memory elements) 2.3.1.
generation statement 4.4.1.
grammatical element 3.1.1.
grammatical structure 3.3.1.
hammock 4.2.
hidden alphabet 3.3.1.
identification 3.1.1.
implementor 1.1.
information connection 2.2.2.
INGEL language 4.4.1.
instrumental computer 1.1.
INTEL language 1.2.
internal state 3.3.1.
leading definition 3.6.
lexeme 3.1.1.
lexical automaton 3.2.1.
linguist 1.1.
metaprocessor 1.1.
mode graph 2.2.1.
multi-operation 2.4.5.
notion 3.1.1.
parsing string 3.1.1.
position 3.1.1.
position expansion 3.6.
pseudonotion 3.6.
proper function 3.2.1.
semantic induction 3.1.1.
semantic lexeme 3.1.1.
shadow 4.2.
stack 2.3.1.
static nesting graph 2.2.3.
subroutine 4.5.
substituent 3.1.1.
synthesis 3.1.2.
thunk 2.2.2.
universal multilanguage compiler 1.1.
zone 4.2.



MACHINES

IN - instrumental
CO - compiling
OB - object

PEOPLE

DSG - designer
LNG - linguist
IMP - implementor
PRG - programmer

Fig. 1. Compiler writing system

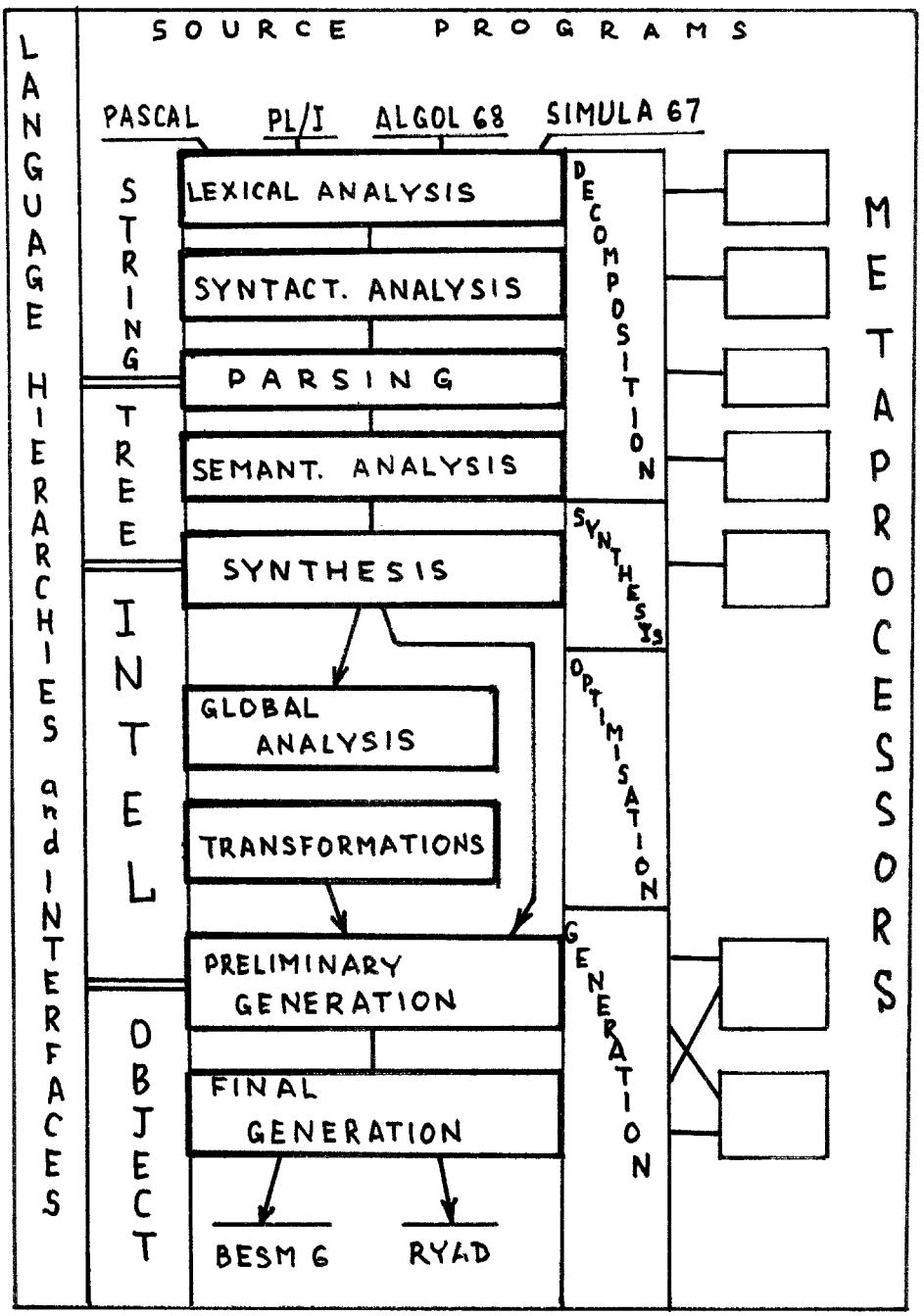


Fig.2. Compilation scheme in the BETA compiler

SOURCE PROGRAM

```
_VAR I,V : INT; _PROC FACT(N : INT);
   _BEGIN _IF N = Ø _THEN U := 1 _ELSE
      _BEGIN FACT(N-1); V := V*N _END
   _END;
   _BEGIN _FOR I := 1 _TO 6 _DO
      _BEGIN FACT(I*2); PRINT(V) _END
   _END
```

LEXICAL STRING

PROGRAM:

```
var id↑i, id↑u = id↑int;
proc id↑fact (id↑n : id↑int);
begin if id↑n = num↑0 then
id↑u := num↑1 else begin
id↑fact (id↑n - num↑1);
id↑u := id↑u * id↑n end end;
begin for id↑i := num↑1 to num↑6 do
begin id↑fact (id↑i * num↑2);
id↑print (id↑u) end end .
```

comment: ↑ denotes a reference to a table entry

TABLES:

symbols

i
u
int
fact
n
print

numerals

0
1
6
2

Fig. 3. Source program and result of the lexical phase

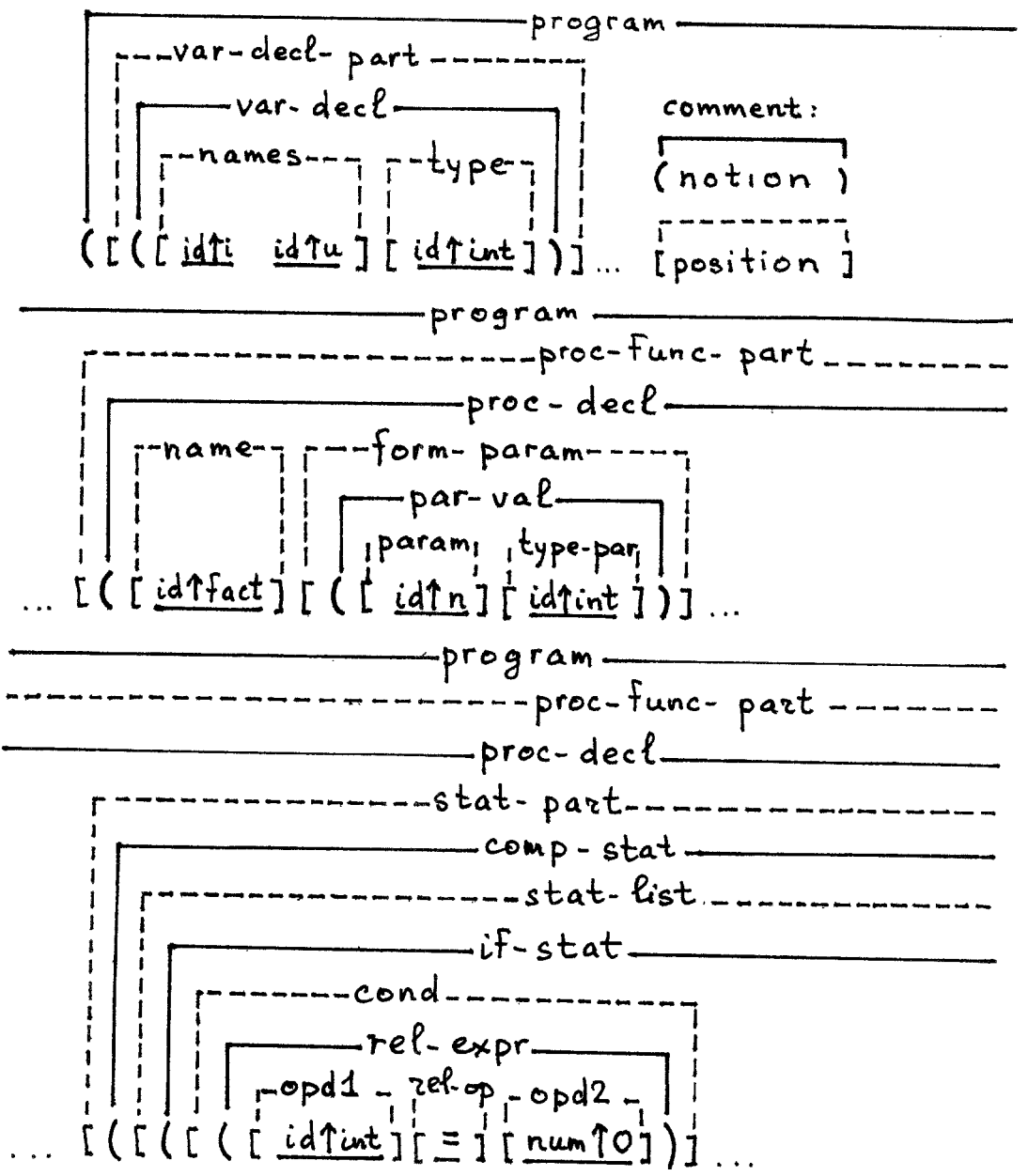


Fig 4. Parsing string (fragment)

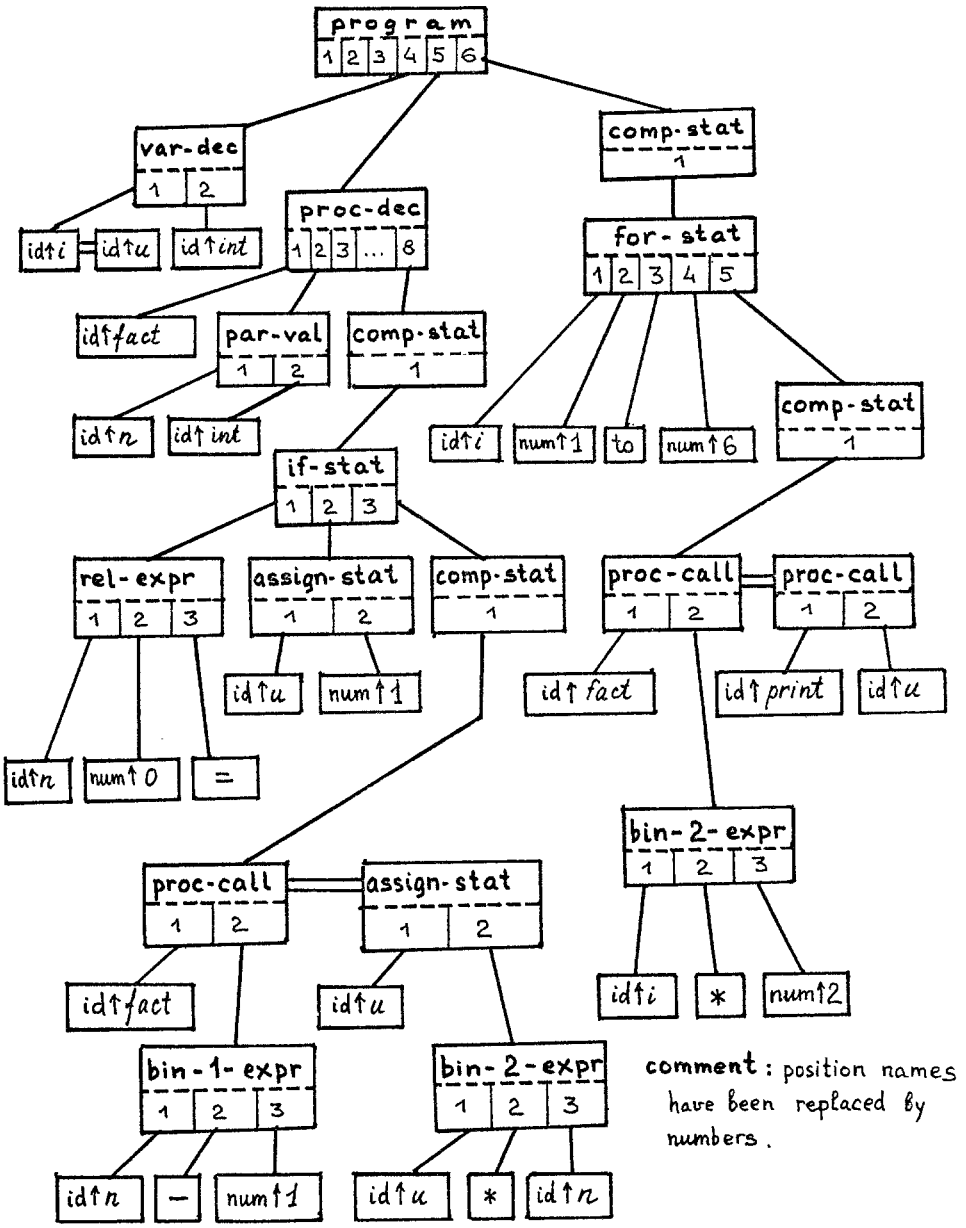
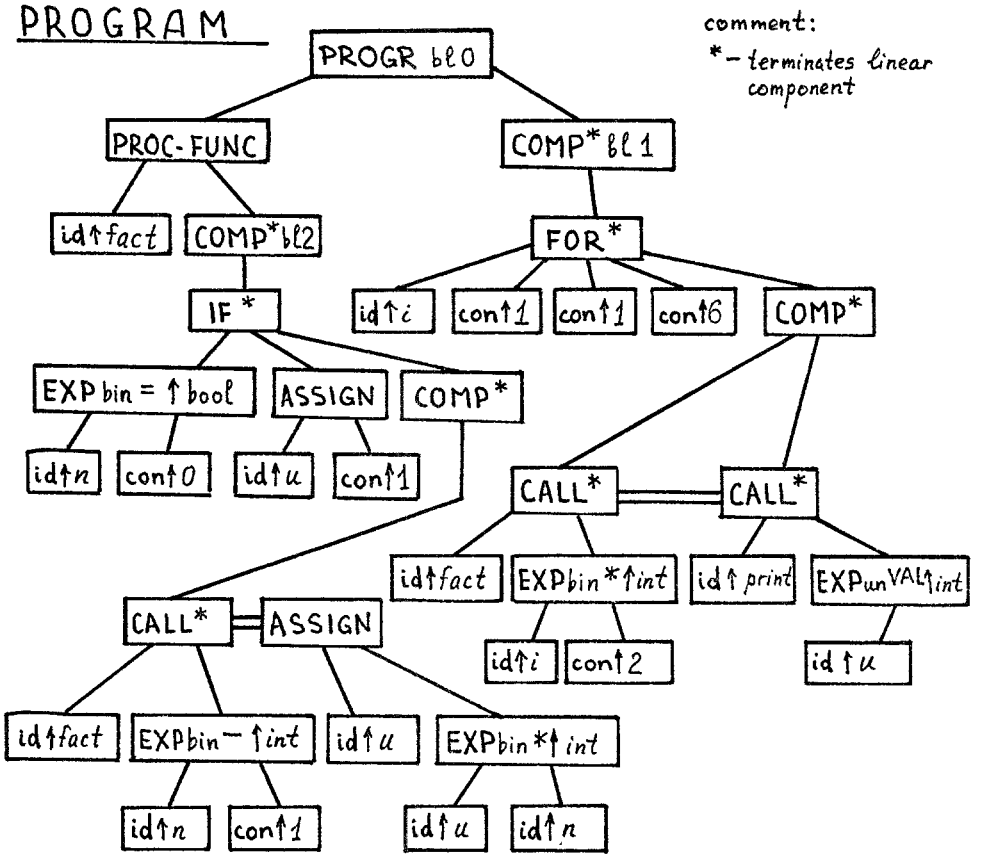


Fig. 5. Parsing tree



→ MODES

int
bool
proc (1, nil, par1(val, ↑int, ↑n))
proc'(1, nil, par1(val, ↑int, ↑ext))

→ IDENTIFIERS

i : var, bl 1, ↑int
u : var, bl 1, ↑int
fact : proc, bl 1, ↑proc
n : par, bl 2, val, ↑int
print : proc, bl 0, ↑proc'

→ CONSTANTS

0 ↑int
1 ↑int
2 ↑int
6 ↑int

Fig. 6. Attributed parsing tree

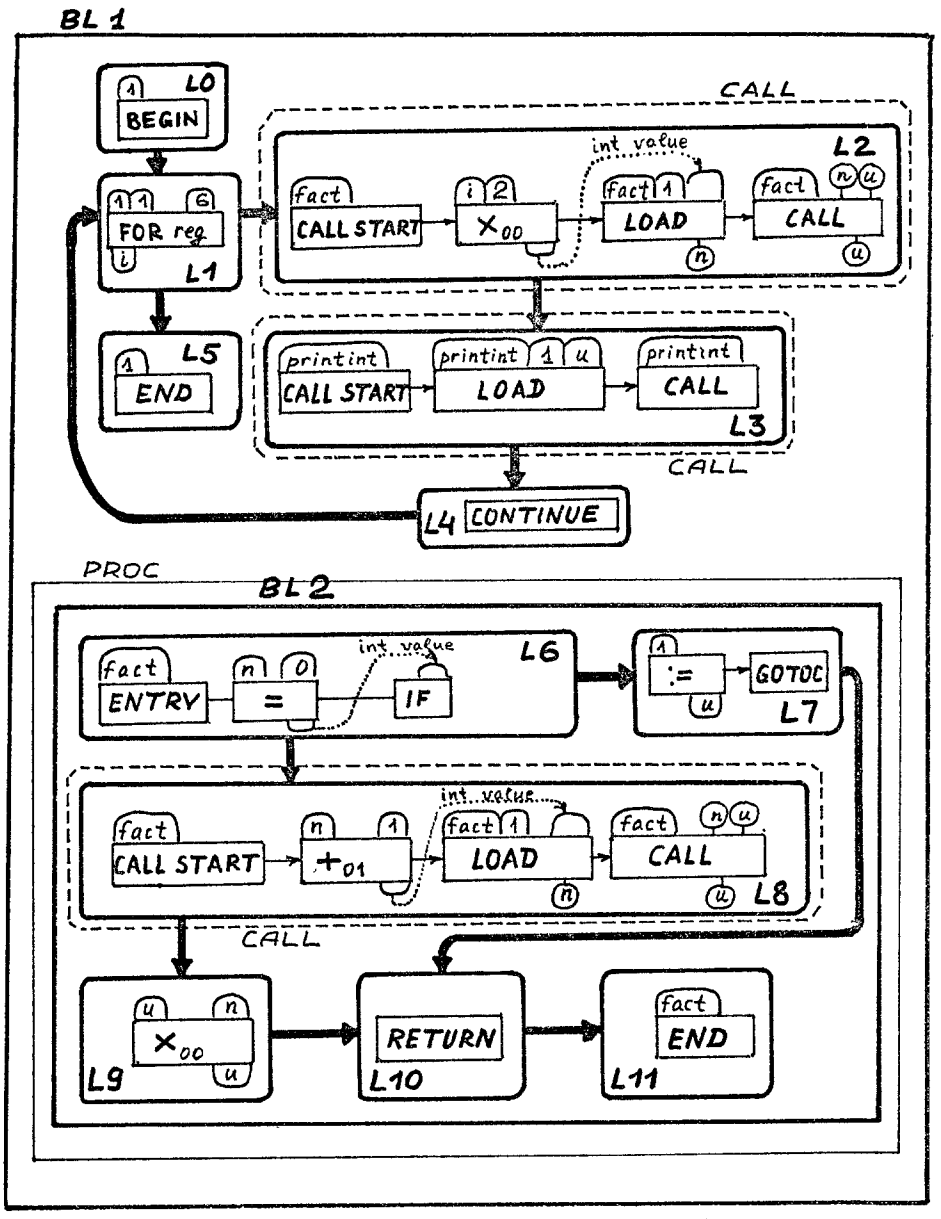


Fig. 7a. INTEL program (control flow graph)

```
MODE GRAPH    comment: parnum-number of parameters
                       resmode-mode of the result
                       call-called by

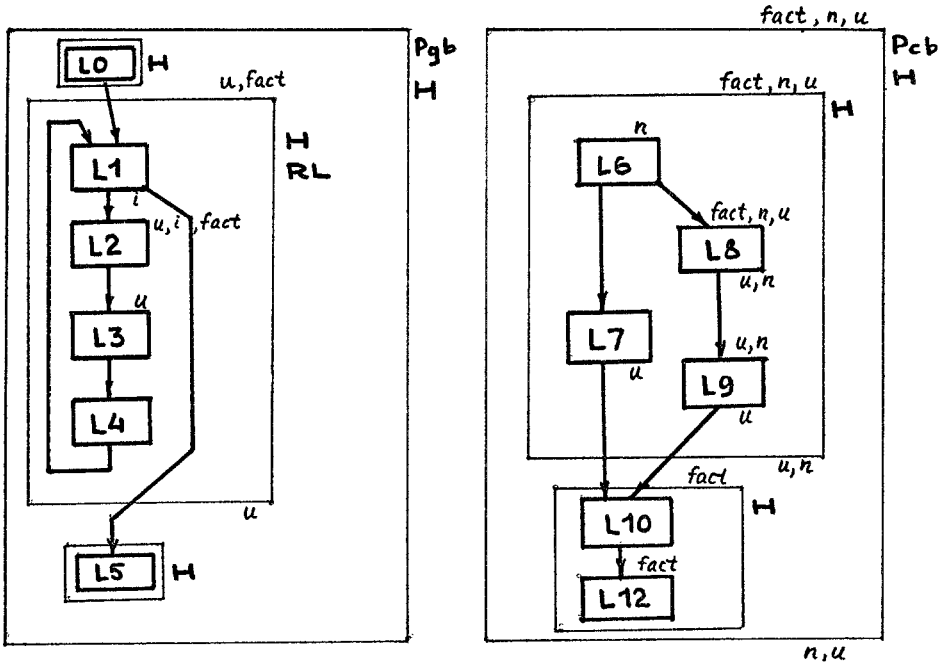
integer
boolean
general procedure (parnum: 1, resmode: ↑nil,
        par 1: (call: val, mode: ↑int))
```

```
STATIC NESTING GRAPH

block 0 (nest: 0, dynam: no, proc: no, start: ↑L0,
        locob: ↑nil, sons: ↑bl1, father: ↑nil)
block 1 (nest: 1, dynam: yes, proc: no, start: ↑L0,
        locob: ↑i, sons: ↑bl2, father: ↑bl0)
block 2 (nest: 2, dynam: yes, proc: ↑fact, start: ↑L6,
        locob: n, sons: ↑nil, father: ↑bl1)
```

```
OBJECTS       comment: reg-implemented by a register
                       rec-recursive, par-parallel calls
                       cor-coroutine
variables:
locbl1 → i (mode: ↑int, memcl: stat, scope: bl1,
             reg: yes, next: ↑u)
         u (mode: ↑int, memcl: stat, scope: bl1,
             reg: yes, next: ↑nil)
locbl2 → n (mode: ↑int, memcl: forpar: (call: val),
             scope: bl2, reg: yes, next: ↑nil)
procedures:
fact (body: bl2, parnum: 1, mode: ↑gen.proc., rec: yes,
   par: no, cor: no, prm: (par1: (call: val, var: n)))
printint (body: blo, parnum: 1, mod: ↑gen.proc., rec: no,
   par: no, cor: no, prm: (par1: (call: val, var: externa
   (mode: ↑int))))
```

Fig. 7b. INTEL program (other structures)



blocks: 0 1 2

ATTRIBUTES

dynamics	no	yes	yes
parallel calls	no	no	no
interrupts	no	no	no
base is necessary	yes	no	yes

proc's :	fact	printint
constant	yes	yes
dynamics	yes	no
recursive	yes	no
parallel	no	no
coroutine	no	no
side-effect	yes	yes
synchronization	no	no
calls others	yes	no

comments:
- H - hammock
- RL - regular loop
- Pgb - program body
- Pcb - procedure body

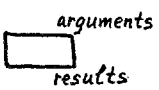


Fig.8. Shadows and optimisation attributes

```
GENERAL REGISTERS

0 - computational
1 - working base
2 - father block base
3 - current block base
4 - program base
5 - stack marker
6 - proc calls
7 - variable i

L0    BALR  4,0          }
      L     2,240(0,4)   }
      LR    3,2          } BEGIN
      LA    5,40         }
      AR    5,3          }

L1    LA    7,1                FOR

L2    LA    6,2          }
      BAL   1,160(0,4)   } CALL
      A     5,32(0,2)    } START
      LA    1,2          }
      MR    0,7          } *oo
      ST    1,16(0,6)          LOAD
      L     1,20(0,2)    }
      BAL   0,184(0,4)   } CALL

L3    L     0,36(0,2)    } LOAD
      ST    0,40(0,2)    }
      BAL   1,PRINT(0,4)       CALL

L4    A     7,4(0,2)     }
      C     7,28(0,2)    } CONT
      BC    12,16(0,4)   }

L5    SVS   (something)        END

L6    LA    0,0          }
fact  C     0,16(0,3)    } n = 0?
      BC    7,112(0,4)         IF

L7    LA    0,1          }
      ST    0,36(0,2)    } u := 1
      BC    15,156(0,4)  }

L8    LA    6,2          }
      BAL   1,160(0,4)   } CALL
      A     5,32(0,2)    } START
      L     0,16(0,3)    }
      S     0,4(0,2)     } n - 1
      ST    1,16(0,6)    }
      L     1,20(0,2)    } CALL
      BAL   0,184(0,4)   }

L9    L     1,36(0,2)    }
      M     0,16(0,3)    } u:=u*n
      ST    1,36(0,2)    }

L10   BC    15,204(0,4)      RETURN

MARK  ST    6,4(0,5)     }
      SLL   6,2          }
      L     6,4(0,2)     }  MARK
      ST    6,0(0,5)     }  STACK
      ST    3,8(0,5)     }  sbrt
      LR    6,5          }
      BCR   15,1         }

CALL  ST    0,12(0,6)    }
      LR    3,6          }
      L     6,4(0,3)     }  PROC
      SLL   6,2          }  CALL
      ST    3,8(6,2)     }  sbrt
      BCR   15,1         }

RET   LR    5,3          }
      L     3,8(0,5)     }
      LR    1,3          }
      L     6,4(0,1)     }
      SLL   6,2          }  RETURN
      ST    1,8(6,2)     }  sbrt
      L     1,0(0,1)     }
      C     1,0(0,2)     }
      BC    7,212(0,4)   }
      BC    15,12(0,5)   }

      DC    x'0'         }
      DC    x'1'         } stack start
      DC    x'0'         } block 1 mark
      DC    A(*-12)      }
      DC    x'0'         }  display
      DC    A(*-172)     'fact' addr.
      DC    printint       address
      DC    x'6'             6
      DC    x'14'           20
      DC    x'0'             u
      DC    x'0'         form.par.
                         PRINTINT
and somewhere 'PRINTINT' body
```

Fig. 9. Object program (IBM/360)

LEXICAL ALPHABET	SOURCE ALPHABET
a	A
A	¯A
1	1
O	Ø
[	[
:=	:=
{	/*
if	_IF␣ or _ЕСЛИ␣
end	_END or _КОНЕЦ␣
id(endifier)	letter[<§|letter|digit>]
num(eral)	[|±|] digit [<§digit>]

ATTRIBUTES of SEMANTIC LEXEMES

id: kind | var | con | proc | lab | frp | field | scal | type | , scope: blocknumb, mode:↑MODE TABLE, representation; ↑SYMBOL TABLE

con: name: ↑IDENTABLE, value: ↑VALTABLE,
mode: ↑MODE TABLE, represent:↑|SYMBOL TABLE|NUMERAL TABLE|

Fig. 10. Lexical information (samples)

```
     ┌ leading definition
proc-decl ::= proc name form-param; label-decl-part
              con-decl-part type-decl-part
              var-decl-part proc-func-part stat-part

     name: id

                                 |par-val          |
                                 |var par-name     |
     form-param:[(<;§            |con par-con      |>)]
     └position                   |func par-func    |
      expansion                  |proc par-proc    |
```

comments:

$\left|\begin{matrix}A\\B\end{matrix}\right|$ - alternative

[ ] - optional

A - terminal symbol A

<A § B> - list of B separated by A

```
comp-stat ::= begin stat-list end
     stat-list: <;§ ALL-STAT>
     ALL-STAT =     |label-stat  |
     │              |assign-stat |
     │              |goto-stat   |
     └pseudonotion  |empty-stat  |
                    |proc-stat   |
                    |comp-stat   |
                    |if-stat     |
                    |case-stat   |
                    |while-stat  |
                    |repeat-stat |
                    |for-stat    |
                    |with-stat   |
                    |par-stat    |

bin-N-expr ::= left-opd N-op right-opd
```

Fig. 11. Regularized C-F grammar

Grammar

```
for-stat ::= for for-var := lowb step uppb do stat
      for-var : id
      lowb : expr
      step : | to |
             | downto |
      uppb : expr
      stat : all-stat (pseudonotion)
with-stat ::= with prefix do stat

      prefix : < ₂§ | id                | >
                    | buf-point ↑       |
                    | field-select      |
                    | sub-var           |

      stat : all-stat

                     | label-stat  |
                     | assign-stat |
                     | goto-stat   |
                     | empty-stat  |
                     | proc-stat   |
      all-stat =     | comp-stat   |
                     | if-stat     |
                     | case-stat   |
                     | while-stat  |
                     | for-stat    |
                     | with-stat   |
                     | par-stat    |
```

Data modes

```
mode for = struct (ref id for-var, ref expr lowb,
                   union (to,downto) step, ref expr uppl,
                   ref all-stat stat)

mode with = struct (ref prefix prefix, ref all-stat stat)

mode prefix = struct (ref union (id, buf-point, field-select,
                     sub-var) prefix, ref prefix next)

mode all-stat = union (label, assign, goto, empty, proc,
                       comp, if, case, while, for, with, par)
```

Fig. 12. Relation between a grammar and a parsing tree data base

```
Reminder → var-decl ::= names : type
                names: <, § id>
                type: ALL-TYPES (pseudonotion)


VAR-DECL
  begin ref type  TYPE := type of var-decl;
      TYPE := MODES(TYPE);
  M: ref id NAME := first names of var-decl;
     if NAME = nil goto EXIT;
     IDENT[HASH(NAME,block)] := (var, block, TYPE, NAME);
     NAME := next names of var-decl; goto M;
EXIT: end


MODES - EP for Mode table  |  give a reference to the
HASH  - EP for Ident table |  corresponding entry

first } standard procedures
next  }
block - block number
```

Fig. 13. Example of EP for semantic analysis
(for Pascal variable-declaration)

```
MODES ¢ ALGOL 68 style ¢

  mode exp ¢ simplified expression: binary operations and conditional
    expressions ¢ = union (binop, ifop);
  mode binop = struct (int opr, opd opd1, opd2);
  mode opd = union (ref exp, ref var, ref con);
  mode ifop = struct (ref exp if, then, else)

    operator coding: -2  -1  +1  +2
                      /   -   +   ×

GLOBALS CONTEXT: accumulated INTEL statement or multioperation
        append:  appends an argument to CONTEXT, second argument re-
                 version of the operation if necessary
        forward: forwards an INTEL statement into INTEL program
          MULTI  |
          ASSIGN |: arrange CONTEXT to be ready for accumulation of
          IF     |    arguments of the corresponding INTEL statement
        EXP:     current vertice of the parsing tree supposedly of
                    exp mode

EP body
    if EXP :=: binop

  then int OPR := opr of EXP
       if abs opr of CONTEXT = abs OPR then
          begin append (opd1 of EXP, true);
                append (opd2 of EXP, opr of CONTEXT = OPR) end
       else begin forward (CONTEXT); CONTEXT := MULTI(OPR);
                  append (opd1 of EXP, true);
                  append (opd2 of EXP, OPR > 0) end fi

  else if EXP :=: itop

       then begin lab then1 := new label;
                  lab else1 := new label;
                  lab fi1 := new label;
                      CONTEXT := IF (then1, else1);
                      append (if of EXP);
                      forward (CONTEXT);
                      var x := new variable;
                  CONTEXT := ASSIGN(x);
                  append (then of EXP);
                  forward (CONTEXT, fi1);
                      CONTEXT := ASSIGN (x);
                      append (else of EXP);
                      forward (CONTEXT, fi1) end

       else ERROR fi

fi
```

Fig. 14. Example of EP for synthesis (of expressions)

ANALYSIS	TRANSFORMATIONS
+ collecting information on references	o open substitution of identical actual parameters; replacement of calls "by value" by calls "by reference"; simplifications of calls and bodies of non-recursive procedures
+ constructing reference graph	
+ finding initial approximation to control and information flow graphs	o removal of local variables from recursive procedures, open substitution of short subroutines, economization of calls to thunks
+ constructing recursive call graph	
+ correcting control and information flow graphs and recursive call graph	o finding unique and linearly recursive variables
	o performing operations on constants
+ finding fragments of control flow graph	o cleaning up loops recursive procedures and hammocks; elimination of identical expressions; operation strength reduction
+ constructing final information flow graph	o removal of constant if-clauses
	o removal of unused statements; finding EQUIVALENCE relations
	o implementing EQUIVALENCE relations

Fig. 15. Sequence of optimisation passes

statement: i := i + 1

Attr (i) : int, reg

IBM/360:	BESM 6 :
a) LA 0,1(0,0)	a) LOAD ir (b_i)
A 0,1(0,b_i)	ADD ir'(b_1)
ST 0,1(0,b_i)	STORE ir (b_i)
b) A 'i', 'one'(0,b_1)	b) ADIADR "i"(1)

Fig. 16. Alternative code generatio