

Chapter One : An Overview of a Compiler

- o Introduction to Compiler
- o Compiler Architecture / Structure
- o Phases of Compiler
- o Exercise

Chapter One : An Overview of a Compiler

oWhat is Compiler ?

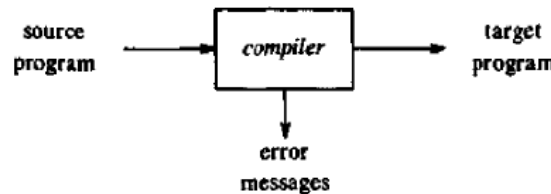
oWhat is the need of Compiler?

Chapter One : An Overview of a Compiler

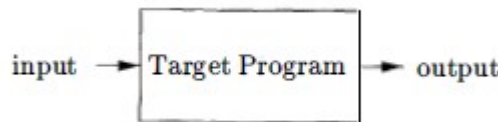
Introduction to Compiler

o **Compiler** : Simply stated, it is a language processor , a compiler is a **program** that can read a program in one language - **the source language** - and translate it into an equivalent program in another language - **the target language**; see Fig. below.

o An important role of the compiler is to report any errors in the source program that it detects during the translation process



o If the target program is an executable machine-language program, it is run by the user to process inputs and see Fig. Below .



Chapter One : An Overview of a Compiler

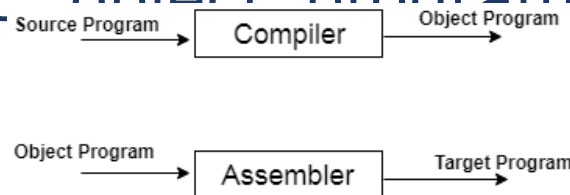
Introduction to Compiler

- o A **compiler** is a translator that converts the high-level language into the machine language.
- o **High-level language** is written by a developer and machine language can be understood by the processor.
- o Compiler is used to show errors to the programmer.
- o The main purpose of compiler is to change the code written in one language without changing the meaning of the program.

Chapter One : An Overview of a Compiler

Introduction to Compiler

- o When you execute a program which is written in HLL programming language then it executes into two parts.
- o In the first part, the source program **compiled and translated into the object program (low level language).**
- o In the second part **object program translated into the target program**



Chapter One : An Overview of a Compiler

This subject discusses the various techniques used to achieve this objective(Compiler Design).

In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.

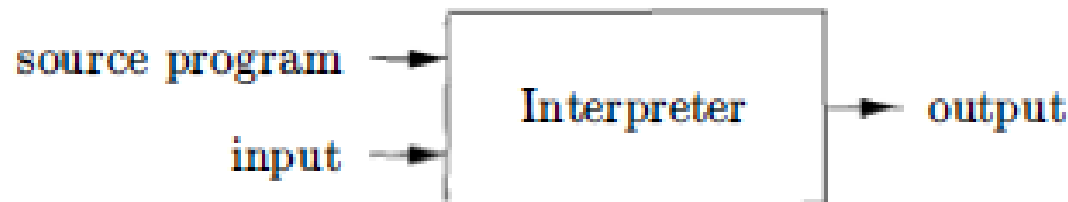
- o Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
- o Techniques used in a parser can be used in a query processing system such as SQL.
- o Many software having a complex front-end may need techniques used in compiler design.
- o A symbolic equation solver which takes an equation as input. That
 - o program should parse the given input equation.
 - o Most of the techniques used in compiler design can be used in Natural Language

Processing (NLP) systems.

Chapter One : An Overview of a Compiler

Introduction to Compiler

- o An **interpreter** is another common kind of language processor that instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on input supplied by the user.

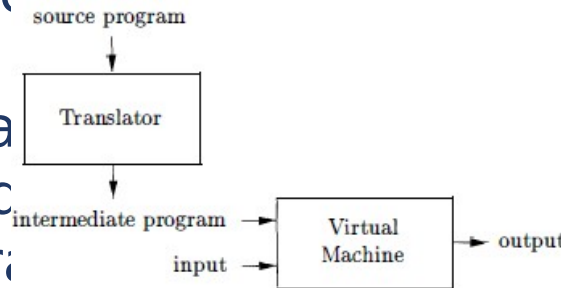


- o The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- o An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Chapter One : An Overview of a Compiler

Introduction to Compiler

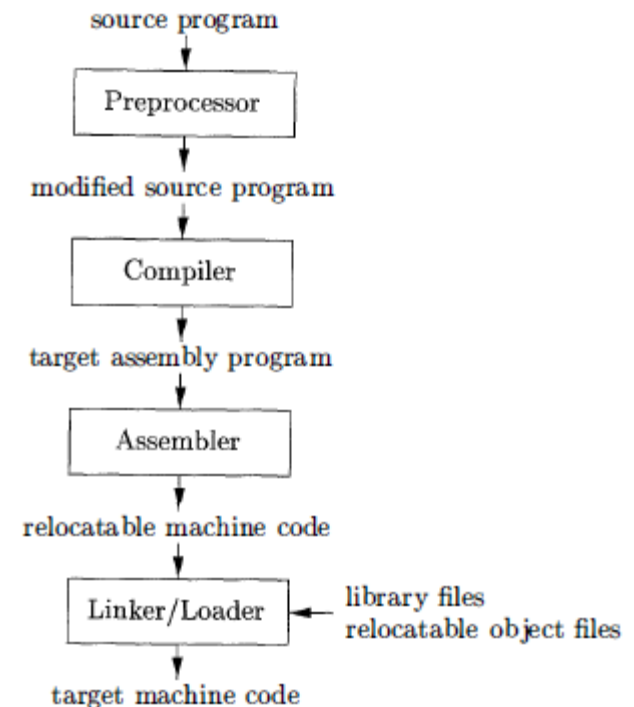
- o Example 1.1 : Java language processors combine compilation and interpretation, as shown in Fig. (hybrid compiler). A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some compilers, translate the source language immediately into machine language. They run the intermediate program.



Chapter One : An Overview of a Compiler

Context of Compiler

- o In addition to a compiler, several other programs are required to create an executable target program, as shown in Fig.
- o A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a **preprocessor**.
- o The preprocessor may also expand shorthands, called **macros**, into source language statements.
- o The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file
- o The **loader** then puts together all executable object files into memory for execution



Chapter One : An Overview of a Compiler

Context of Compiler

- o The **modified source program** is then fed to a compiler. The compiler may produce an assembly-language program as its output, **because assembly language is easier to produce as output and is easier to debug.**
- o The **assembly language** is then processed by a program called an assembler that produces relocatable machine code as its output.
- o Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.

Structure of a Compiler

There are two major parts of a compiler : **Analysis and Synthesis** . The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Analysis Part

- o The analysis part breaks up the source program into constituent **pieces** and **imposes a grammatical structure** on them.
- o It then uses this structure to create **an intermediate representation** of the source program.
- o If the analysis part detects that the source program is either **syntactically ill formed or semantically unsound**, then it must provide informative messages, so the user can take corrective action.
- o The analysis part also collects information about the source program and stores it in a data structure called a **symbol table**, which is passed along with the intermediate representation to the synthesis part.
- o **Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the phases in this part.**

Chapter One : An Overview of a Compiler

Structure of a Compiler

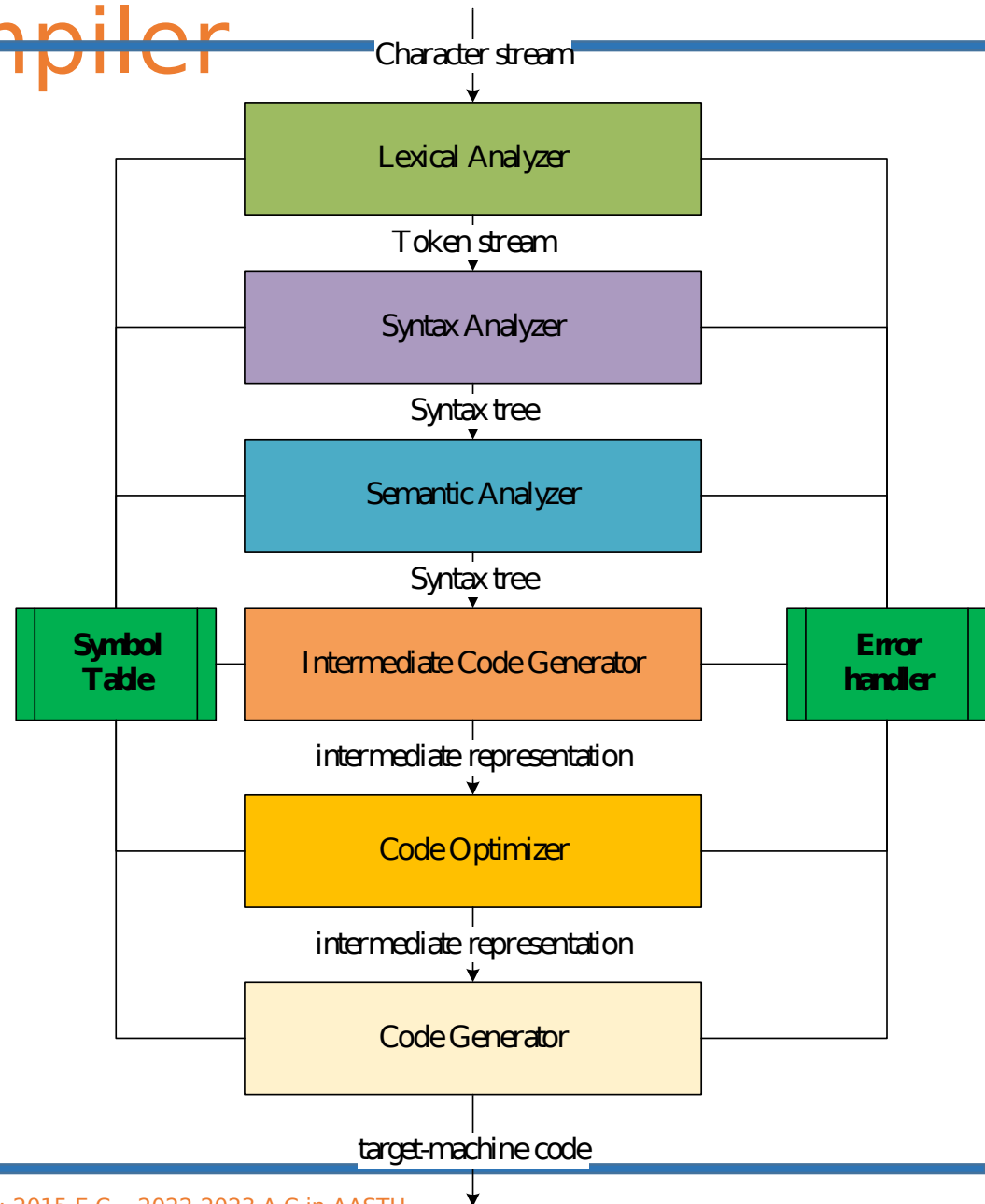
Synthesis Part

- o The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- o The **symbol table**, which stores information about the entire source program, is used by all phases of the compiler.
- o If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.
- o **Intermediate Code Generator, Code Generator, and Code Optimizer** are the phases in this part.

Chapter One : An Overview of a Compiler

The Phases of a Compiler

- o Conceptually, a compiler operates in phases, each of which transforms the Source program from one representation to another.
- o The first three phases, forming the bulk of **analysis operation of a compiler**. And the last three phases forming the bulk of **Synthesis operation of a compiler**.
- o The other activities, **symbol-table management and error handling**, are shown interacting with the six phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.



Chapter One : An Overview of a Compiler

The Phases of a Compiler

Lexical Analysis

- o The first phase of a compiler is called **lexical analysis or scanning**. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**.
- o A token describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)
- o For each lexeme, the lexical analyzer produces as output a token of the form that it passes on to the subsequent phase, syntax analysis .

<token- name, attribute-value>

- In the token, the first component **token- name** is an **abstract symbol** that is used during syntax analysis , and the second component **attribute-value** points to an entry in the symbol table for this token. Information from the symbol-table entry 'is needed for semantic analysis and code generation.

Chapter One : An Overview of a Compiler

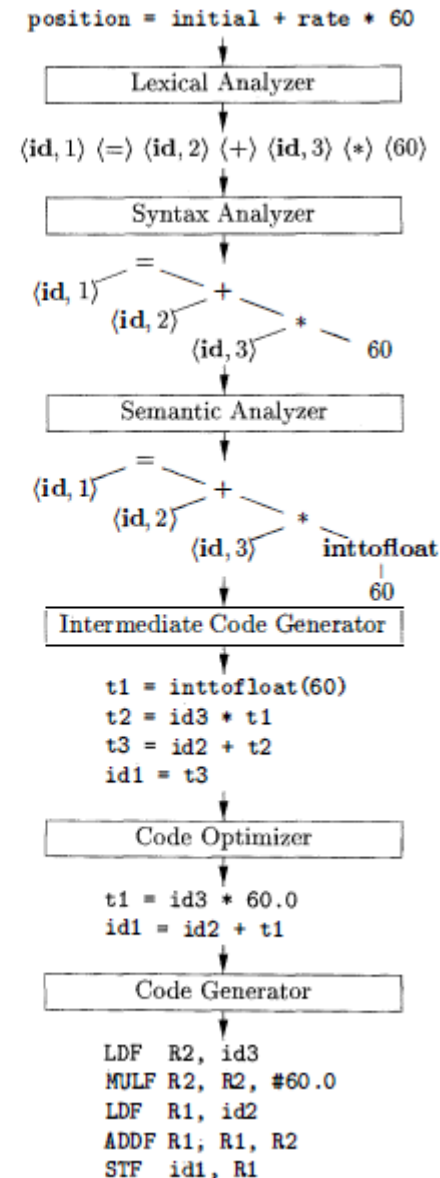
The Phases of a Compiler

Lexical Analysis

- o Puts information about identifiers(token) into the symbol table.
- o Regular expressions are used to describe tokens (lexical constructs).
- o A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.
- o position, =, initial, +, rate, * and 60 are token

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE



Chapter One : An Overview of a Compiler

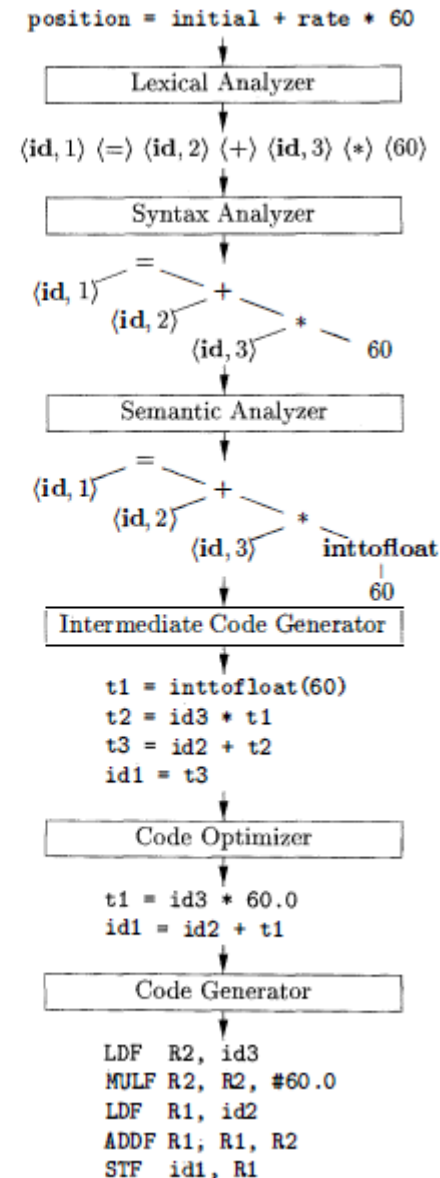
The Phases of a Compiler

Syntax Analysis

- o A Syntax Analyzer creates the **syntactic structure (generally a parse tree)** of the given program.
- o A syntax analyzer is **also called a parser**.
- o A parse tree describes a **syntactic structure**
- o The syntax of a language is specified by a **context free grammar (CFG)**.
- o The rules in a CFG are mostly recursive.
- o A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not. If it satisfies, the syntax analyzer creates a parse tree for the given program.

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE



Chapter One : An Overview of a Compiler

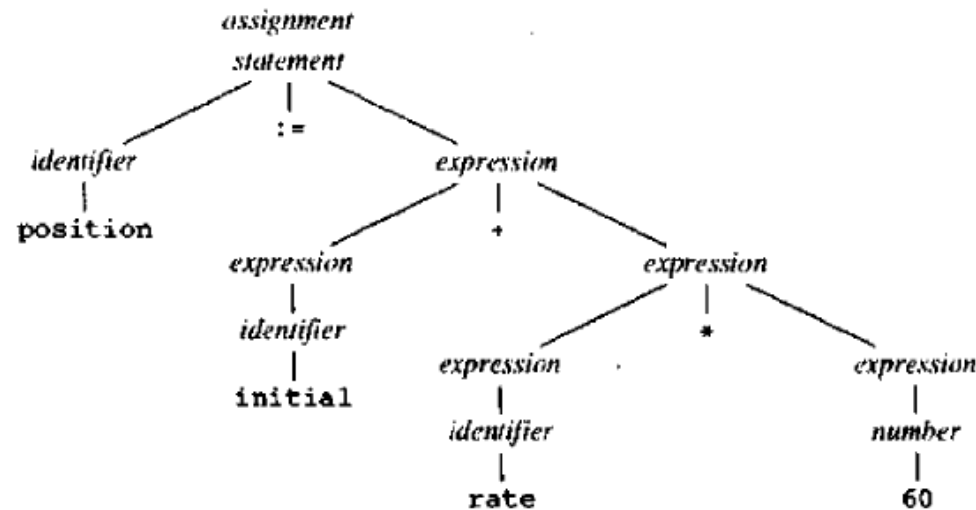
The Phases of a Compiler

Syntax Analysis

- o Generate CFG for the given parse tree:
- o Depending on how the parse tree is created, there are different parsing techniques. These parsing techniques are categorized into two groups:
 - o Top-Down Parsing,
 - o Bottom-Up Parsing

Top-Down Parsing:

- o Construction of the parse tree starts at the root, and proceeds towards the leaves.
- o Efficient top-down parsers can be easily constructed by hand.
- o Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).



Bottom-Up Parsing:

- o Construction of the parse tree starts at the leaves, and proceeds towards the root.
- o Normally efficient bottom-up parsers are created with the help of some software tools.
- o Bottom-up parsing is also known as **shift-reduce parsing**.

Chapter One : An Overview of a Compiler

The Phases of a Compiler

Semantic Analysis

- o A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- o Semantic Analysis makes sure that declarations and statements of program are semantically correct.
- o It is a collection of procedures which is called by parser as and when required by grammar.
- o Type-checking is an important part of semantic analyzer.
- o Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- o Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules) . The result is a syntax-directed translation and Attribute grammars

Chapter One : An Overview of a Compiler

The Phases of a Compiler

Semantic Analysis

- o The semantic analyzer uses the **syntax tree** and the information in the **symbol table** to check the source program for semantic consistency with the language definition.
- o It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Chapter One : An Overview of a Compiler

The Phases of a Compiler

Intermediate Code Generation

- o A compiler may produce an explicit intermediate codes representing the source program.
- o These intermediate codes are generally machine architecture independent. But the level of intermediate codes is close to the level of machine codes.

*newval := oldval * fact + 1*



*id1 := id2 * id3 + 1*



MULT id2, id3, temp1
ADD temp1, #1, temp2
MOV temp2, id1

Chapter One : An Overview of a Compiler

The Phases of a Compiler

Code Optimization

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

```
MULT id2, id3, temp1  
ADD temp1, #1, id1
```

Chapter One : An Overview of a Compiler

The Phases of a Compiler

Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers Or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.
- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

Chapter One : An Overview of a Compiler

Symbol-Table Management

- An essential function of a compiler is to record the variables names used in the source program and collect information about various attributes of each name
- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its program can be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g., by value or by reference), and the type returned
- The symbol table is a data structure containing a record for each variable name, with fields for attributes of the name
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

Chapter One : An Overview of a Compiler

Compiler Construction Tools

- There are tools created to help implement the various phases of a compiler
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithm
- Some commonly used compiler construction tools include:
 - **Parser generators** that automatically produce syntax analyzers from a grammatical description of a programming language
 - **Scanner generators** that produce lexical analyzers from a regular-expression description of the tokens of the language
 - **Syntax-directed** translation engines that produce collections of routines for walking a parse tree and generating intermediate code
 - **Code generator** generators that produce a code a generator form a collection of rules for translating each operation of the intermediate language into the machine language for a target machine
 - **Data-flow analysis engines** that facilitate the gathering of information about how values are transmitted from one part of the program to each other part. Data flow analysis is key part of code optimization
 - **Compiler-construction** toolkits that provide an integrated set of routines for constructing various phases of a compiler

Chapter One : An Overview of a Compiler

Exercises

- What is the difference between a compiler and an interpreter?
- What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?
- What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?
- A compiler that translates a high-level language into another high-level language is called a source-to-source translator. What advantages are there to using C as a target language for a compiler?
- Describe some of the tasks that an assembler needs to perform.