UNIVERSITY OF KERBALA
COLLEGE OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

OPERATOR PRECEDENCE PARSING

Submitted to:
Dr. Hiba Adnan

Prepared by:
Abdullah Mohammed
Muntadher Haider

December 24, 2025

Abstract

This report presents an in-depth analysis of Operator Precedence Parsing, a robust bottom-up parsing technique essential in the field of compiler design. While less general than LR parsers, Operator Precedence Parsers are highly efficient for a specific class of grammars—Operator Grammars—making them ideal for parsing mathematical and logical expressions. This document explores the mathematical foundations of operator grammars, the derivation of precedence relations ($\lessdot$, $\doteq$, $\gtrdot$), and the algorithmic construction of Leading and Trailing sets. Furthermore, it introduces the concept of Precedence Functions as a memory-optimization technique and discusses error recovery strategies. A comprehensive, step-by-step trace of a complex expression involving parentheses is provided to illustrate the practical mechanism of the shift-reduce process.

Contents

1 Introduction **3**
1.1 Overview of Parsing 3
1.2 Significance of Operator Precedence Parsing 3

2 Theoretical Foundations **3**
2.1 Context-Free Grammars 3
2.2 Operator Grammars 3

3 Operator Precedence Relations **4**
3.1 Formal Definitions 4

4 Construction Algorithms **4**
4.1 Computing LEADING Sets 4
4.2 Computing TRAILING Sets 5
4.3 Populating the Precedence Matrix 5

5 Parsing Algorithm and Implementation **5**
5.1 Pseudo-code 5

6 Optimization: Precedence Functions **5**

7 Error Handling and Recovery **6**

8 Comprehensive Example Trace **6**

9 Analysis and Discussion **7**
9.1 Complexity 7
9.2 Advantages 7
9.3 Limitations 7

10 Conclusion **8**

1 Introduction

1.1 Overview of Parsing

Parsing, or syntax analysis, constitutes the second and perhaps most critical phase of a compiler's front end. Following lexical analysis, which tokenizes the source code, the parser validates the stream of tokens against the grammatical rules of the programming language. Its output is typically a Parse Tree or an Abstract Syntax Tree (AST), which represents the hierarchical structure of the code.

Parsers are broadly categorized into two families:

1. **Top-Down Parsers**: Build the tree from the root (S) down to the leaves. Examples include Recursive Descent and LL(1) parsers.
2. **Bottom-Up Parsers**: Construct the tree from the leaves (terminals) up to the root. Examples include LR(k), LALR, and the subject of this report, the Operator Precedence Parser.

1.2 Significance of Operator Precedence Parsing

The Operator Precedence Parser (OPP) is a specialized shift-reduce parser. Historically, it was one of the earliest parsing methods developed. Its significance lies in its implementation simplicity and its intuitive handling of mathematical expressions, where operator precedence (e.g., $*$ over +) and associativity (e.g., left-to-right evaluation) are paramount. Although it does not support the full range of Context-Free Grammars (CFG), its speed and small footprint make it a technique of choice for simple expression evaluators and specific subsystems within larger compilers.

2 Theoretical Foundations

2.1 Context-Free Grammars

A Context-Free Grammar (CFG) is defined by a 4-tuple $G = (V, T, P, S)$, where:

- V is a finite set of non-terminals (variables).
- T is a finite set of terminals.
- P is a finite set of production rules of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$.
- S is the start symbol.

2.2 Operator Grammars

For Operator Precedence Parsing to be applicable, the grammar must be an **Operator Grammar**. An operator grammar imposes two strict constraints on the production rules:

1. **No ϵ-productions**: There are no rules of the form $A \rightarrow \epsilon$ (empty string).
2. **No Adjacent Non-terminals**: There are no rules where two non-terminals appear consecutively. A production like $A \rightarrow BC$ is forbidden.

These properties ensure that every relationship between non-terminals is mediated by a terminal (an operator), allowing the parser to determine the structure based solely on the precedence of these terminals.

3 Operator Precedence Relations

The parser makes decisions (shift or reduce) based on the relationship between the terminal at the top of the stack and the current lookahead terminal from the input. Three disjoint relations are defined:

3.1 Formal Definitions

Let $a, b \in T$ (terminals) and $A, B \in V$ (non-terminals). The relations are defined as follows:

1. **Yields Precedence ($a \lessdot b$)**:
   - Meaning: Terminal a has lower precedence than b. We must shift b to process it before reducing a.
   - Condition: $a \lessdot b$ if there is a production $A \to \alpha a B \beta$ where $b \in \mathit{LEADING}(B)$.
2. **Equal Precedence ($a \doteq b$)**:
   - Meaning: Terminals a and b have the same precedence and are part of the same handle.
   - Condition: $a \doteq b$ if there is a production $A \to \alpha a \beta b \gamma$ where β is either a single non-terminal or ϵ.
3. **Takes Precedence ($a \gtrdot b$)**:
   - Meaning: Terminal a has higher precedence than b. We must reduce the handle containing a before processing b.
   - Condition: $a \gtrdot b$ if there is a production $A \to \alpha B b \beta$ where $a \in \mathit{TRAILING}(B)$.

4 Construction Algorithms

To build the parsing table, we first need to compute two auxiliary sets for each non-terminal: **LEADING** and **TRAILING**.

4.1 Computing LEADING Sets

The set *LEADING*(*A*) contains all terminals that can appear as the first terminal in a string derived from non-terminal *A*.

$$LEADING(A) = \{a \mid A \Rightarrow^{+} \gamma a \delta, \text{ where } \gamma \in V^{*} \text{ or } \epsilon\}$$

Algorithm for LEADING(A):

1. For each production $A \rightarrow \gamma a \delta$ where γ is ϵ or a single non-terminal, add a to *LEADING*(*A*).
2. Repeat until no new terminals can be added:
3. If $A \rightarrow B\alpha$ is a production, add all terminals in *LEADING*(*B*) to *LEADING*(*A*)

.

4.2 Computing TRAILING Sets

The set $TRAILING(A)$ contains all terminals that can appear as the last terminal in a string derived from A.

$$TRAILING(A) = \{a \mid A \Rightarrow^{+} \gamma a \delta, \text{ where } \delta \in V^{*} \text{ or } \epsilon\}$$

Algorithm for TRAILING(A):

1. For each production $A \rightarrow \gamma a \delta$ where δ is ϵ or a single non-terminal, add a to $TRAILING(A)$.
2. Repeat until no new terminals can be added:
3. If $A \rightarrow \alpha B$ is a production, add all terminals in $TRAILING(B)$ to $TRAILING(A)$.

4.3 Populating the Precedence Matrix

Once the sets are ready, the matrix $M[a, b]$ is filled:

1. For production $A \rightarrow \ldots aB \ldots$, set $a \lessdot x$ for all $x \in LEADING(B)$.
2. For production $A \rightarrow \ldots Bb \ldots$, set $x \gtrdot b$ for all $x \in TRAILING(B)$.
3. For production $A \rightarrow \ldots ab \ldots$ or $A \rightarrow \ldots aCb \ldots$, set $a \doteq b$.
4. **End Markers**: \$ yields precedence to everything in LEADING(S), and everything in TRAILING(S) takes precedence over \$.

5 Parsing Algorithm and Implementation

The Operator Precedence Parser uses an input buffer and a stack. The critical difference from other bottom-up parsers is that non-terminals on the stack are essentially transparent; the parser looks for the topmost *terminal* to compare with the input.

5.1 Pseudo-code

6 Optimization: Precedence Functions

The precedence table for a language with n terminals requires $O(n^2)$ space. For languages with many operators, this can be large. **Precedence Functions** map each terminal to two integers, f (stack precedence) and g (input precedence), reducing space to $O(2n)$.

We seek functions f and g such that:

- $f(a) < g(b)$ whenever $a \lessdot b$
- $f(a) = g(b)$ whenever $a \doteq b$
- $f(a) > g(b)$ whenever $a \gtrdot b$

This is constructed by building a directed graph where nodes are terminals. While not all tables can be linearized into functions (if there are cycles in the graph), most practical arithmetic grammars can be.

Algorithm 1 Operator Precedence Parsing

```
Input string w appended with $
Stack S ← [$]
a ← top terminal on S
b ← current input symbol
while a ≠ $ OR b ≠ $ do
    if a ⋘ b OR a ≐ b then
        Push b onto S
        Advance input pointer
    else if a ⋙ b then
        repeat
            Pop terminal x
            y ← top terminal on S
        until y ⋘ x
                                        ▷ Handle identified and reduced
    else
        Error: Missing entry in table
    end if
    a ← top terminal on S
    b ← current input symbol
end while
return Success
```

7 Error Handling and Recovery

One of the challenges in bottom-up parsing is reporting meaningful errors. In Operator Precedence Parsing, an error is detected when the corresponding entry $M[a, b]$ in the table is empty.

Recovery Strategies:

1. **Panic Mode**: Scan down the stack until a set of safe states is found, and skip input tokens until a synchronized token (like a semicolon or closing parenthesis) is found.
2. **Insertion/Deletion**: The parser can attempt to insert a missing operator or operand, or delete an unexpected one, to invoke a valid relation. For example, if *id id* is found (missing operator), the parser might insert a + or $*$.

8 Comprehensive Example Trace

Let us consider a grammar that supports parentheses, which creates nested precedence contexts.

Grammar:

1. $E \rightarrow E + T \mid T$
2. $T \rightarrow T * F \mid F$
3. $F \rightarrow (E) \mid id$

Key Precedence Relations:

- $*$ ⋙ + (Multiplication higher than addition)

- + m + (Left associativity)
- (≪+, (≪∗, (≪(
- +m), ∗m)
- (≐)

Input String: (*id* + *id*) ∗ *id*$

Step	Stack	Rel	Input	Action
1	$	≪	(id + id) * id $	Shift (
2	$ (	≪	id + id) * id $	Shift id
3	$ (id	m	+ id) * id $	Reduce $F \rightarrow id$
4	$ (	≪	+ id) * id $	Shift +
5	$ (+	≪	id) * id $	Shift id
6	$ (+ id	m	) * id $	Reduce $F \rightarrow id$
7	$ (+	m	) * id $	Reduce $E \rightarrow E + T$
8	$ (	≐	) * id $	Shift)
9	$ ( )	m	* id $	Reduce $F \rightarrow (E)$
10	$	≪	* id $	Shift *
11	$ *	≪	id $	Shift id
12	$ * id	m	$	Reduce $F \rightarrow id$
13	$ *	m	$	Reduce $T \rightarrow T * F$
14	$		$	**Accepted**

9 Analysis and Discussion

9.1 Complexity

The time complexity of operator precedence parsing is generally linear, $O(n)$, with respect to the length of the input string. This is because each terminal is shifted onto the stack once and reduced once. The calculation of Leading and Trailing sets is performed once during compiler generation (compile-time of the compiler), not during the parsing of source code.

9.2 Advantages

- **Implementation Speed**: Very fast due to simple lookup tables.
- **Code Size**: Requires minimal code to implement the loop.
- **No Explicit State**: Unlike LR parsers which maintain complex state machines, OPP relies purely on terminal relations.

9.3 Limitations

- **Unary Operators**: Handling unary minus (e.g., -5) vs binary minus ($5 - 3$) is difficult because the token is the same but precedence differs.
- **Grammar Restrictions**: Only a small subset of grammars are Operator Grammars.
- **Reduction ambiguity**: The parser knows *when* to reduce, but not always *what* production to reduce to (since it ignores non-terminals).

10 Conclusion

The Operator Precedence Parser serves as an excellent educational tool and a practical solution for specific parsing needs. While it has been largely superseded by LALR(1) parsers (like YACC and Bison) for full language parsing, its concepts of precedence relations and associativity handling remain fundamental to understanding how compilers process mathematical logic. This report has demonstrated the theoretical derivation of these relations and their practical application in a robust parsing algorithm.

References

[1] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.

[2] Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann.

[3] Grune, D., & Jacobs, C. J. (2008). *Parsing Techniques: A Practical Guide*. Springer Science & Business Media.

[4] Appel, A. W. (2002). *Modern Compiler Implementation in C*. Cambridge University Press.