Bab 6

Top-Down Parsing dan Recursive Descent

6.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Menjelaskan konsep top-down parsing dan perbedaannya dengan bottom-up parsing
2. Memahami LL parsing dan karakteristiknya
3. Mengimplementasikan recursive descent parser untuk grammar sederhana
4. Menangani precedence dan associativity dalam recursive descent parser
5. Mengimplementasikan error recovery pada recursive descent parser
6. Mengintegrasikan lexer dengan recursive descent parser
7. Mengevaluasi ekspresi aritmatika menggunakan recursive descent parser

6.2 Pendahuluan

Pada bab ini kita mengimplementasikan **parser hand-written** (recursive descent) untuk **grammar proyek subset C** (Bab 5, Bagian **??**). Parser ini mengenali program, statement, declaration, assignment, print-statement, dan ekspresi dengan precedence/associativity yang sama dengan parser proyek di Bab 8 (`simplec.y`), sehingga berfungsi sebagai implementasi manual untuk bahasa yang sama dengan proyek.

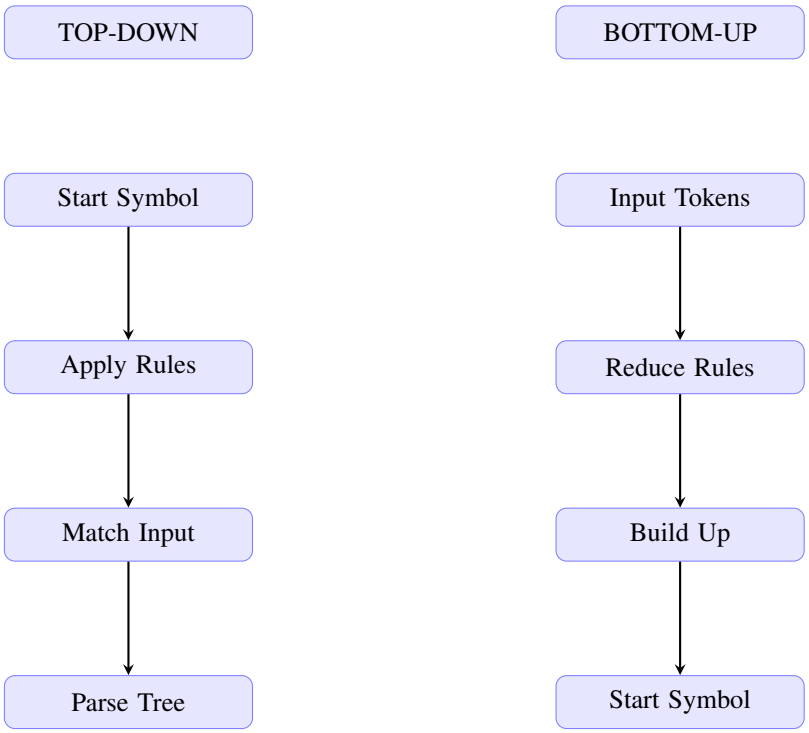
Setelah mempelajari lexical analysis dan context-free grammar pada bab-bab sebelumnya, kita sekarang akan mempelajari bagaimana mengimplementasikan parser yang menganalisis struktur sintaksis dari stream token yang dihasilkan oleh lexer. Top-down parsing adalah salah satu pendekatan yang paling intuitif dan mudah diimplementasikan secara manual.

Menurut sumber terbuka:

> "Top-down parsers (recursive descent) – easy to hand-write; better for LL(1) grammars, when unambiguous. Works by writing functions for grammar nonterminals (e.g. expression(), term(), factor()) that consume tokens one at a time."[1]

Pendekatan top-down parsing dimulai dari start symbol grammar dan mencoba menurunkan (derive) input dengan membangun parse tree dari root ke leaves. Ini berbeda dengan bottom-up parsing yang membangun parse tree dari leaves ke root.

Gambar 6.1 menunjukkan perbandingan top-down dan bottom-up parsing.



Gambar 6.1: Perbandingan top-down dan bottom-up parsing

6.3 Konsep Top-Down Parsing

6.3.1 Definisi Top-Down Parsing

Top-down parsing adalah teknik parsing yang dimulai dari start symbol grammar dan mencoba menurunkan input dengan menerapkan production rules dari atas ke bawah. Parser mencoba mencocokkan input dengan memprediksi production mana yang harus digunakan berdasarkan lookahead token.

Karakteristik utama top-down parsing:

- Membangun parse tree dari root (start symbol) ke leaves (terminals)

- Menggunakan leftmost derivation
- Memerlukan lookahead untuk memprediksi production yang tepat
- Dapat diimplementasikan secara recursive atau iterative dengan stack

6.3.2 LL Parsing

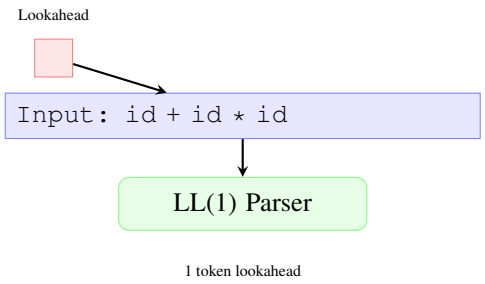
LL parsing adalah kelas top-down parsing yang membaca input dari **L**eft ke right dan menghasilkan **L**eftmost derivation. Notasi LL(k) menunjukkan bahwa parser menggunakan k token lookahead untuk membuat keputusan parsing.

Menurut sumber dari USNA:

> "Top-down parsing starts from the start symbol and tries to rewrite it to match the input, building a parse tree from root to leaves. LL parsing means scanning input Left-to-right, producing a Leftmost derivation, using k-token lookahead (usually LL(1))."[1]

LL(1) adalah yang paling umum digunakan karena hanya memerlukan satu token lookahead, membuat implementasinya lebih sederhana dan efisien.

Gambar 6.2 menunjukkan konsep LL parsing.



Gambar 6.2: Konsep LL parsing dengan lookahead

6.3.3 Keuntungan dan Keterbatasan Top-Down Parsing

Keuntungan:

- Mudah diimplementasikan secara manual (recursive descent)
- Error messages yang lebih intuitif (dapat menunjukkan posisi error dengan tepat)
- Tidak memerlukan preprocessing grammar yang kompleks (untuk grammar LL(1))

[1] https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/lec/l09/lec.html

- Cocok untuk grammar yang sudah dalam bentuk yang sesuai

Keterbatasan:

- Tidak dapat menangani left recursion secara langsung
- Memerlukan grammar yang sudah di-factoring untuk menghindari ambiguity
- Tidak sekuat LR parsing dalam hal kemampuan parsing
- Beberapa grammar memerlukan transformasi sebelum dapat di-parse dengan top-down

6.4 Recursive Descent Parsing

6.4.1 Konsep Recursive Descent

Recursive descent parsing adalah teknik implementasi top-down parsing di mana setiap non-terminal dalam grammar direpresentasikan sebagai sebuah fungsi. Fungsi-fungsi ini saling memanggil secara recursive sesuai dengan struktur grammar.

Menurut sumber dari Ernest Chu:

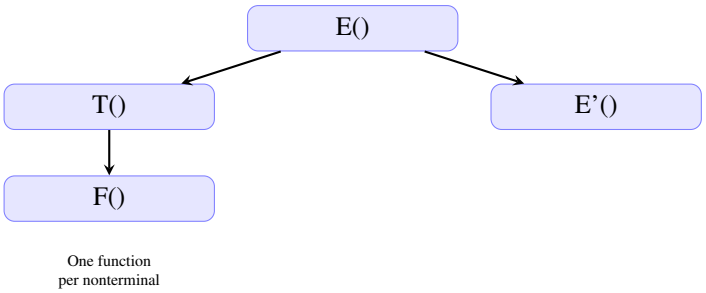
> "Recursive-descent parsing is a hand-written parser (one function per non-terminal), possibly with backtracking. When you eliminate left recursion and factor grammar properly, you can build deterministic predictive parsers (LL(1))—recursive descent without backtracking."[2]

Struktur dasar recursive descent parser:

1. Setiap non-terminal memiliki fungsi sendiri
2. Fungsi membaca token dari input stream
3. Fungsi memanggil fungsi lain sesuai dengan production rules
4. Terminal dicocokkan langsung dengan token saat ini

Gambar 6.3 menunjukkan struktur recursive descent parser.

[2] https://ernestchu.github.io/course-notes/courses/cse360-design-and-implementation-of-compiler/syntax-analysis/top-down-parsing.html



Gambar 6.3: Struktur recursive descent parser

6.4.2 Implementasi Dasar

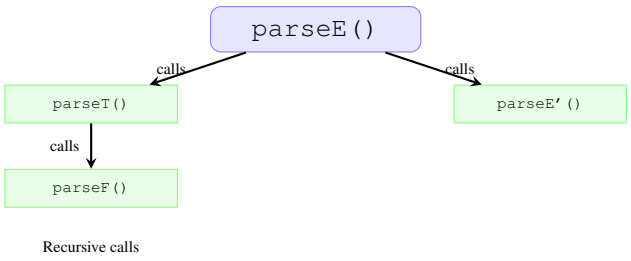
Mari kita lihat contoh implementasi recursive descent parser untuk grammar ekspresi aritmatika sederhana. Grammar yang akan kita gunakan:

```
E  -> T E'
E' -> + T E' | epsilon
T  -> F T'
T' -> * F T' | epsilon
F  -> ( E ) | id | num
```

Grammar ini sudah dalam bentuk yang sesuai untuk LL(1) parsing karena:

- Tidak ada left recursion
- Sudah di-factoring (E' dan T' menangani associativity)
- Setiap production dapat diputuskan dengan satu token lookahead

Gambar 6.4 menunjukkan contoh pemanggilan fungsi dalam recursive descent parser.



Gambar 6.4: Contoh pemanggilan fungsi recursive descent

Implementasi dalam C++:

Listing 6.1: Struktur dasar recursive descent parser

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
```

```
4
5  enum TokenType {
6      TOK_ID, TOK_NUM, TOK_PLUS, TOK_MUL,
7      TOK_LPAREN, TOK_RPAREN, TOK_END, TOK_ERROR
8  };
9
10 struct Token {
11     TokenType type;
12     std::string lexeme;
13     int line, col;
14 };
15
16 class RecursiveDescentParser {
17 private:
18     std::vector<Token> tokens;
19     size_t current;
20     Token lookahead;
21
22     void nextToken() {
23         if (current < tokens.size()) {
24             lookahead = tokens[current++];
25         } else {
26             lookahead = {TOK_END, "", 0, 0};
27         }
28     }
29
30     void match(TokenType expected) {
31         if (lookahead.type == expected) {
32             nextToken();
33         } else {
34             error("Expected " + tokenToString(expected) +
35                   " but got " + lookahead.lexeme);
36         }
37     }
38
39     void error(const std::string& msg) {
40         std::cerr << "Syntax error at line " << lookahead.line
41                   << ", col " << lookahead.col << ": " << msg << std::
   ↪ endl;
42         exit(1);
43     }
44
45 public:
46     RecursiveDescentParser(const std::vector<Token>& t)
47         : tokens(t), current(0) {
48         nextToken();
49     }
50
51     // Grammar: E -> T E'
52     void parseE() {
53         parseT();
54         parseEPrime();
55     }
56
```

```
    // Grammar: E' -> + T E' | epsilon
    void parseEPrime() {
        if (lookahead.type == TOK_PLUS) {
            match(TOK_PLUS);
            parseT();
            parseEPrime();
        }
        // else: epsilon production, do nothing
    }

    // Grammar: T -> F T'
    void parseT() {
        parseF();
        parseTPrime();
    }

    // Grammar: T' -> * F T' | epsilon
    void parseTPrime() {
        if (lookahead.type == TOK_MUL) {
            match(TOK_MUL);
            parseF();
            parseTPrime();
        }
        // else: epsilon production, do nothing
    }

    // Grammar: F -> ( E ) | id | num
    void parseF() {
        if (lookahead.type == TOK_LPAREN) {
            match(TOK_LPAREN);
            parseE();
            match(TOK_RPAREN);
        } else if (lookahead.type == TOK_ID) {
            match(TOK_ID);
        } else if (lookahead.type == TOK_NUM) {
            match(TOK_NUM);
        } else {
            error("Expected identifier, number, or '('");
        }
    }

    void parse() {
        parseE();
        if (lookahead.type != TOK_END) {
            error("Extra input after expression");
        }
        std::cout << "Parse successful!" << std::endl;
    }
};
```

6.5 Handling Precedence dan Associativity

6.5.1 Konsep Precedence

Precedence menentukan urutan evaluasi operator ketika beberapa operator muncul dalam ekspresi yang sama. Misalnya, dalam ekspresi `a + b * c`, operator `*` memiliki precedence lebih tinggi daripada +, sehingga dievaluasi terlebih dahulu.

Dalam recursive descent parser, precedence di-handle melalui struktur grammar. Operator dengan precedence lebih tinggi berada di level yang lebih dalam dalam parse tree.

6.5.2 Handling Associativity

Associativity menentukan bagaimana operator dengan precedence yang sama dievaluasi. Ada dua jenis:

- **Left-associative**: Dievaluasi dari kiri ke kanan, misalnya `a - b - c` = `(a - b) - c`
- **Right-associative**: Dievaluasi dari kanan ke kiri, misalnya `a = b = c` = `a = (b = c)`

Dalam grammar yang kita gunakan, E' dan T' menggunakan left recursion yang diubah menjadi right recursion untuk menangani left associativity dengan benar.

Contoh grammar untuk menangani precedence dan associativity:

```
E  -> T E'         (expression level, lowest precedence)
E' -> + T E' | epsilon  (addition, left-associative)
T  -> F T'         (term level, higher precedence)
T' -> * F T' | epsilon  (multiplication, left-associative)
F  -> ( E ) | id  (factor level, highest precedence)
```

Struktur ini memastikan bahwa:

- Operator `*` memiliki precedence lebih tinggi daripada + (T berada di bawah E)
- Kedua operator left-associative (menggunakan right recursion dengan tail call)
- Parentheses memiliki precedence tertinggi (F)

6.5.3 Implementasi dengan Evaluasi

Berikut adalah implementasi recursive descent parser yang tidak hanya mem-parse tetapi juga mengevaluasi ekspresi:

Listing 6.2: Recursive descent parser dengan evaluasi

```
class ExpressionEvaluator {
private:
    std::vector<Token> tokens;
    size_t current;
    Token lookahead;

    void nextToken() {
        if (current < tokens.size()) {
            lookahead = tokens[current++];
        } else {
            lookahead = {TOK_END, "", 0, 0};
        }
    }

    void match(TokenType expected) {
        if (lookahead.type == expected) {
            nextToken();
        } else {
            throw std::runtime_error("Syntax error");
        }
    }

public:
    ExpressionEvaluator(const std::vector<Token>& t)
        : tokens(t), current(0) {
        nextToken();
    }

    // E -> T E'
    // Returns value of expression
    int parseE() {
        int value = parseT();
        return parseEPrime(value);
    }

    // E' -> + T E' | epsilon
    // Accumulates addition operations
    int parseEPrime(int left) {
        if (lookahead.type == TOK_PLUS) {
            match(TOK_PLUS);
            int right = parseT();
            return parseEPrime(left + right);
        }
        return left;  // epsilon production
    }

    // T -> F T'
    int parseT() {
        int value = parseF();
        return parseTPrime(value);
    }

    // T' -> * F T' | epsilon
```

```
    // Accumulates multiplication operations
    int parseTPrime(int left) {
        if (lookahead.type == TOK_MUL) {
            match(TOK_MUL);
            int right = parseF();
            return parseTPrime(left * right);
        }
        return left;  // epsilon production
    }

    // F -> ( E ) | num
    int parseF() {
        if (lookahead.type == TOK_LPAREN) {
            match(TOK_LPAREN);
            int value = parseE();
            match(TOK_RPAREN);
            return value;
        } else if (lookahead.type == TOK_NUM) {
            int value = std::stoi(lookahead.lexeme);
            match(TOK_NUM);
            return value;
        } else {
            throw std::runtime_error("Expected number or '('");
        }
    }

    int evaluate() {
        int result = parseE();
        if (lookahead.type != TOK_END) {
            throw std::runtime_error("Extra input");
        }
        return result;
    }
};
```

6.6 Error Recovery pada Recursive Descent

6.6.1 Pentingnya Error Recovery

Error recovery adalah kemampuan parser untuk melanjutkan parsing setelah menemukan error, sehingga dapat melaporkan multiple errors dalam satu pass. Tanpa error recovery, parser akan berhenti pada error pertama.

6.6.2 Strategi Error Recovery

Beberapa strategi error recovery yang umum digunakan:

Synchronization Points

Menentukan synchronization points (token-token yang dapat digunakan untuk recovery), seperti:

- Statement terminators (`;`, `}`)
- Keywords yang menandai awal konstruksi baru (`if`, `while`, `return`)
- Operator yang jelas (+, −, ⋆)

Panic Mode Recovery

Ketika error ditemukan, parser membuang token sampai menemukan synchronization point:

Listing 6.3: Panic mode error recovery

```
void parseE() {
    parseT();
    parseEPrime();
}

void parseEPrime() {
    if (lookahead.type == TOK_PLUS) {
        match(TOK_PLUS);
        parseT();
        parseEPrime();
    } else if (!isValidFollow(lookahead.type)) {
        // Error recovery: skip until synchronization point
        error("Expected '+' or end of expression");
        while (!isSynchronizationPoint(lookahead.type) &&
               lookahead.type != TOK_END) {
            nextToken();
        }
    }
    // else: valid follow token, epsilon production
}

bool isSynchronizationPoint(TokenType t) {
    return t == TOK_RPAREN || t == TOK_END ||
           t == TOK_SEMICOLON;
}

bool isValidFollow(TokenType t) {
    return t == TOK_RPAREN || t == TOK_END ||
           t == TOK_SEMICOLON || t == TOK_PLUS;
}
```

Error Production

Menambahkan production khusus untuk menangani error:

```
E' -> + T E' | epsilon | error E'
```

Ketika error ditemukan, parser dapat menggunakan error production untuk recovery.

6.6.3 Implementasi Error Recovery yang Lebih Baik

Berikut adalah implementasi yang lebih robust dengan error recovery:

Listing 6.4: Error recovery yang lebih baik

```
class ParserWithRecovery {
private:
    int errorCount;
    std::vector<Token> tokens;
    size_t current;
    Token lookahead;

    void nextToken() {
        if (current < tokens.size()) {
            lookahead = tokens[current++];
        } else {
            lookahead = {TOK_END, "", 0, 0};
        }
    }

    void error(const std::string& msg) {
        errorCount++;
        std::cerr << "Error at line " << lookahead.line
                  << ", col " << lookahead.col << ": "
                  << msg << std::endl;
    }

    void synchronize() {
        // Skip tokens until synchronization point
        while (lookahead.type != TOK_END) {
            if (isSynchronizationPoint(lookahead.type)) {
                return;
            }
            nextToken();
        }
    }

    bool isSynchronizationPoint(TokenType t) {
        return t == TOK_RPAREN || t == TOK_SEMICOLON ||
               t == TOK_END || t == TOK_PLUS || t == TOK_MUL;
    }

public:
    ParserWithRecovery(const std::vector<Token>& t)
        : tokens(t), current(0), errorCount(0) {
        nextToken();
    }

    void parseE() {
        try {
            parseT();
```

```
            parseEPrime();
        } catch (...) {
            error("Error in expression");
            synchronize();
        }
    }

    void parseEPrime() {
        if (lookahead.type == TOK_PLUS) {
            match(TOK_PLUS);
            parseT();
            parseEPrime();
        } else if (!isValidFollow(lookahead.type)) {
            error("Expected '+' or end of expression");
            synchronize();
        }
    }

    int getErrorCount() const { return errorCount; }
};
```

6.7 Integrasi Lexer dengan Parser

6.7.1 Arsitektur Integrasi

Dalam implementasi praktis, lexer dan parser bekerja bersama dalam pipeline:

```
Source Code -> Lexer -> Token Stream -> Parser -> Parse Tree/AST
```

Lexer membaca source code karakter demi karakter dan menghasilkan stream token. Parser kemudian membaca token dari stream ini.

6.7.2 Implementasi Terintegrasi

Berikut adalah contoh implementasi lexer dan parser yang terintegrasi:

Listing 6.5: Lexer dan parser terintegrasi

```
#include <iostream>
#include <string>
#include <cctype>

class IntegratedLexerParser {
private:
    std::string input;
    size_t pos;
    int line, col;
    Token lookahead;

    // Lexer functions
    void skipWhitespace() {
        while (pos < input.size() && isspace(input[pos])) {
```

```
            if (input[pos] == '\n') {
                line++;
                col = 1;
            } else {
                col++;
            }
            pos++;
        }
    }

    Token nextToken() {
        skipWhitespace();

        if (pos >= input.size()) {
            return {TOK_END, "", line, col};
        }

        char c = input[pos];
        int startCol = col;

        // Identifier: [a-zA-Z][a-zA-Z0-9]*
        if (isalpha(c)) {
            std::string lexeme;
            while (pos < input.size() && isalnum(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return {TOK_ID, lexeme, line, startCol};
        }

        // Number: [0-9]+
        if (isdigit(c)) {
            std::string lexeme;
            while (pos < input.size() && isdigit(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return {TOK_NUM, lexeme, line, startCol};
        }

        // Operators and punctuation
        pos++;
        col++;
        switch (c) {
            case '+': return {TOK_PLUS, "+", line, startCol};
            case '*': return {TOK_MUL, "*", line, startCol};
            case '(': return {TOK_LPAREN, "(", line, startCol};
            case ')': return {TOK_RPAREN, ")", line, startCol};
            default: return {TOK_ERROR, std::string(1, c), line, startCol
↪ };
        }
    }

    void match(TokenType expected) {
```

```
68          if (lookahead.type == expected) {
69              lookahead = nextToken();
70          } else {
71              std::cerr << "Error: Expected " << expected
72                        << " but got " << lookahead.lexeme
73                        << " at line " << lookahead.line
74                        << ", col " << lookahead.col << std::endl;
75              exit(1);
76          }
77      }
78
79  public:
80      IntegratedLexerParser(const std::string& s)
81          : input(s), pos(0), line(1), col(1) {
82          lookahead = nextToken();
83      }
84
85      // Parser functions (same as before)
86      void parseE() {
87          parseT();
88          parseEPrime();
89      }
90
91      void parseEPrime() {
92          if (lookahead.type == TOK_PLUS) {
93              match(TOK_PLUS);
94              parseT();
95              parseEPrime();
96          }
97      }
98
99      void parseT() {
100         parseF();
101         parseTPrime();
102     }
103
104     void parseTPrime() {
105         if (lookahead.type == TOK_MUL) {
106             match(TOK_MUL);
107             parseF();
108             parseTPrime();
109         }
110     }
111
112     void parseF() {
113         if (lookahead.type == TOK_LPAREN) {
114             match(TOK_LPAREN);
115             parseE();
116             match(TOK_RPAREN);
117         } else if (lookahead.type == TOK_ID) {
118             match(TOK_ID);
119         } else if (lookahead.type == TOK_NUM) {
120             match(TOK_NUM);
121         } else {
```

```
            std::cerr << "Error: Expected id, num, or '('" << std::endl;
            exit(1);
        }
    }

    void parse() {
        parseE();
        if (lookahead.type != TOK_END) {
            std::cerr << "Error: Extra input" << std::endl;
            exit(1);
        }
        std::cout << "Parse successful!" << std::endl;
    }
};
```

6.8 Contoh Lengkap: Parser untuk Ekspresi Aritmatika

Berikut adalah contoh lengkap implementasi recursive descent parser untuk ekspresi aritmatika yang dapat menangani:

- Operasi penjumlahan dan perkalian
- Precedence (perkalian lebih tinggi dari penjumlahan)
- Left associativity
- Parentheses
- Identifier dan literal angka
- Error reporting yang informatif

Listing 6.6: Parser lengkap untuk ekspresi aritmatika

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <stdexcept>

enum TokenType {
    TOK_ID, TOK_NUM, TOK_PLUS, TOK_MINUS, TOK_MUL, TOK_DIV,
    TOK_LPAREN, TOK_RPAREN, TOK_END, TOK_ERROR
};

struct Token {
    TokenType type;
    std::string lexeme;
    int line, col;

    Token(TokenType t, const std::string& l, int ln, int c)
```

```
        : type(t), lexeme(l), line(ln), col(c) {}
};

class ArithmeticParser {
private:
    std::string input;
    size_t pos;
    int line, col;
    Token lookahead;
    std::vector<std::string> errors;

    void skipWhitespace() {
        while (pos < input.size() && isspace(input[pos])) {
            if (input[pos] == '\n') {
                line++;
                col = 1;
            } else {
                col++;
            }
            pos++;
        }
    }

    Token nextToken() {
        skipWhitespace();

        if (pos >= input.size()) {
            return Token(TOK_END, "", line, col);
        }

        char c = input[pos];
        int startLine = line, startCol = col;

        // Identifier
        if (isalpha(c) || c == '_') {
            std::string lexeme;
            while (pos < input.size() &&
                   (isalnum(input[pos]) || input[pos] == '_')) {
                lexeme += input[pos++];
                col++;
            }
            return Token(TOK_ID, lexeme, startLine, startCol);
        }

        // Number
        if (isdigit(c)) {
            std::string lexeme;
            while (pos < input.size() && isdigit(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return Token(TOK_NUM, lexeme, startLine, startCol);
        }

```

```
72          // Operators
73          pos++;
74          col++;
75          switch (c) {
76              case '+': return Token(TOK_PLUS, "+", startLine, startCol);
77              case '-': return Token(TOK_MINUS, "-", startLine, startCol);
78              case '*': return Token(TOK_MUL, "*", startLine, startCol);
79              case '/': return Token(TOK_DIV, "/", startLine, startCol);
80              case '(': return Token(TOK_LPAREN, "(", startLine, startCol);
81              case ')': return Token(TOK_RPAREN, ")", startLine, startCol);
82              default:
83                  return Token(TOK_ERROR, std::string(1, c), startLine,
↪ startCol);
84          }
85      }
86
87      void match(TokenType expected) {
88          if (lookahead.type == expected) {
89              lookahead = nextToken();
90          } else {
91              std::string msg = "Expected " + tokenToString(expected) +
92                                " but got " + lookahead.lexeme +
93                                " at line " + std::to_string(lookahead.line)
↪ +
94                                ", col " + std::to_string(lookahead.col);
95              errors.push_back(msg);
96              throw std::runtime_error(msg);
97          }
98      }
99
100     std::string tokenToString(TokenType t) {
101         switch (t) {
102             case TOK_ID: return "identifier";
103             case TOK_NUM: return "number";
104             case TOK_PLUS: return "'+'";
105             case TOK_MINUS: return "'-'";
106             case TOK_MUL: return "'*'";
107             case TOK_DIV: return "'/'";
108             case TOK_LPAREN: return "'('";
109             case TOK_RPAREN: return "')'";
110             case TOK_END: return "end of input";
111             default: return "unknown";
112         }
113     }
114
115 public:
116     ArithmeticParser(const std::string& s)
117         : input(s), pos(0), line(1), col(1) {
118         lookahead = nextToken();
119     }
120
121     // E -> T E'
122     void parseE() {
123         parseT();
```

```
124         parseEPrime();
125     }
126
127     // E' -> (+ | -) T E' | epsilon
128     void parseEPrime() {
129         if (lookahead.type == TOK_PLUS || lookahead.type == TOK_MINUS) {
130             TokenType op = lookahead.type;
131             match(op);
132             parseT();
133             parseEPrime();
134         }
135     }
136
137     // T -> F T'
138     void parseT() {
139         parseF();
140         parseTPrime();
141     }
142
143     // T' -> (* | /) F T' | epsilon
144     void parseTPrime() {
145         if (lookahead.type == TOK_MUL || lookahead.type == TOK_DIV) {
146             TokenType op = lookahead.type;
147             match(op);
148             parseF();
149             parseTPrime();
150         }
151     }
152
153     // F -> ( E ) | id | num
154     void parseF() {
155         if (lookahead.type == TOK_LPAREN) {
156             match(TOK_LPAREN);
157             parseE();
158             match(TOK_RPAREN);
159         } else if (lookahead.type == TOK_ID) {
160             match(TOK_ID);
161         } else if (lookahead.type == TOK_NUM) {
162             match(TOK_NUM);
163         } else {
164             std::string msg = "Expected identifier, number, or '(' at
↪ line " +
165                               std::to_string(lookahead.line) +
166                               ", col " + std::to_string(lookahead.col);
167             errors.push_back(msg);
168             throw std::runtime_error(msg);
169         }
170     }
171
172     bool parse() {
173         try {
174             parseE();
175             if (lookahead.type != TOK_END) {
176                 errors.push_back("Extra input after expression");
```

```
                return false;
            }
            return errors.empty();
        } catch (...) {
            return false;
        }
    }

    void printErrors() {
        for (const auto& err : errors) {
            std::cerr << err << std::endl;
        }
    }
};

int main() {
    std::string input;
    std::cout << "Enter expression: ";
    std::getline(std::cin, input);

    ArithmeticParser parser(input);
    if (parser.parse()) {
        std::cout << "Parse successful!" << std::endl;
    } else {
        std::cout << "Parse failed!" << std::endl;
        parser.printErrors();
    }

    return 0;
}
```

6.9 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Top-down parsing adalah teknik parsing yang membangun parse tree dari root ke leaves
2. LL parsing adalah kelas top-down parsing yang membaca input left-to-right dan menghasilkan leftmost derivation
3. Recursive descent parsing adalah implementasi top-down parsing di mana setiap non-terminal direpresentasikan sebagai fungsi
4. Precedence dan associativity di-handle melalui struktur grammar yang tepat
5. Error recovery memungkinkan parser untuk melanjutkan setelah menemukan error
6. Lexer dan parser dapat diintegrasikan dalam satu implementasi yang efisien

Recursive descent parsing adalah teknik yang sangat cocok untuk implementasi manual parser karena mudah dipahami dan diimplementasikan. Namun, perlu diingat bahwa grammar harus dalam bentuk yang sesuai (tanpa left recursion, sudah di-factoring) untuk dapat di-parse dengan pendekatan ini. Parser yang dibahas di bab ini mengimplementasikan grammar proyek subset C (Bab 5); parser proyek dengan Bison dibangun di Bab 8 (`simplec.y`).

6.10 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang top-down parsing dan recursive descent, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 4: Syntax Analysis, Section 4.4: Top-Down Parsing
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 3: Scanners, Section 3.4: Top-Down Parsing
- **OpenGenus Tutorial**: Build Lexer [1] - Bagian tentang recursive descent parsing
- **USNA Course Notes**: Top-Down Parsing [3]
- **Ernest Chu Course Notes**: Syntax Analysis - Top-Down Parsing [4]
- **TutorialsPoint**: Compiler Design - Top Down Parser [5]

[3] https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/lec/l09/lec.html

[4] https://ernestchu.github.io/course-notes/courses/cse360-design-and-implementation-of-compiler/syntax-analysis/top-down-parsing.html

[5] https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm

Daftar Pustaka

[1] OpenGenus. *Build Lexer*. Tutorial on hand-written lexers. 2024. URL: https://iq.opengenus.org/build-lexer/.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.