PMAT 402 - Systems Programming
Assignment -3
Simplified Pascal Compiler in C++

Gandholi Sarat - 23008

April 10, 2025

Contents

1 Objective

To design and implement (as an entire class of students) a recursive descent parser in C++ for a simple programming language resembling Pascal, incorporating lexical analysis, syntax rules, and parsing functions to validate program input.

2 Pascal Grammar Given

```
1  <prog>         ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2  <prog-name>    ::= id
3  <dec-list>     ::= <dec> | <dec-list> ; <dec>
4  <dec>          ::= <id-list> : <type>
5  <type>         ::= INTEGER
6  <id-list>      ::= id | <id-list> , id
7  <stmt-list>    ::= <stmt> | <stmt-list> ; <stmt>
8  <stmt>         ::= <assign> | <read> | <write> | <for>
9  <assign>       ::= id := <exp>
10 <exp>          ::= <term> | <exp> + <term> | <exp> - <term>
11 <term>         ::= <factor> | <term> * <factor> | <term> DIV <factor>
12 <factor>       ::= id | int | ( <exp> )
13 <read>         ::= READ ( <id-list> )
14 <write>        ::= WRITE ( <id-list> )
15 <for>          ::= FOR <index-exp> DO <body>
16 <index-exp>    ::= id := <exp> TO <exp>
17 <body>         ::= <stmt> | BEGIN <stmt-list> END
```

Figure 1: Pascal Grammar

3 Code Explanation

3.1 Lexical Mapping

The lex map defines token-to-integer associations for keywords, symbols, and identifiers. This simulates the output of a lexical analyzer.

```
map <string , int > lex = {
  {"PROGRAM", 1}, {"VAR", 2}, {"BEGIN", 3}, ..., {"int", 23}
};
```

3.2 Tokenization

The tokenize function splits the input string into words and converts them into token integers using the lex map. If an unknown token is encountered, an error is displayed and the program exits.

3.3 Token Management

The current token is managed using a global variable TOKEN, and the function `get_NextToken()` retrieves the next token from the token stream.

```
int get_NextToken () {
  if (index1 < tokens.size()) {
    return tokens[index1++];
  }
  return -1;
}
```

3.4 Parsing Functions

Each function corresponds to a non-terminal symbol in the grammar. These functions recursively validate the input token sequence.

- `prog()` parses the full program structure.
- `stmt()` identifies whether a statement is an assignment, read, write, or for-loop.
- `exp()`, `term()`, and `factor()` handle expressions based on operator precedence.

Example: `assign()` checks if a statement is in the form `i := expression`:

```
bool assign () {
  if (TOKEN == 22) {
    TOKEN = get_NextToken ();
    if (TOKEN == 15) {
      TOKEN = get_NextToken ();
      if (exp()) {
        return true;
      }
    }
  }
  return false;
}
```

3.5 Parsing Entry Point

The `parse()` function serves as the entry point for the parser. It first tokenizes the input string and initializes the token stream. The syntax validation process begins by invoking `stmt()`, which acts as the top-level non-terminal for parsing single or compound statements. If the entire token stream is successfully consumed and validated, the input is deemed syntactically correct; otherwise, an error message is displayed.

```
void parse(string input) {
  tokens = tokenize(input);
  index1 = 0;
  TOKEN = get_NextToken ();
  if (stmt() && index1 == tokens.size()) {
```

```
    cout << "Valid syntax\n";
  } else {
    cout << "Invalid syntax\n";
  }
}
```

3.6 Main Function

The `main()` function reads user input and passes it to `parse()` for validation.

```
int main() {
  string input;
  cout << "Enter input string: ";
  getline(cin, input);
  parse(input);
  return 0;
}
```

4 Full Code

```
#include <iostream>
#include <string>
#include <map>
#include <sstream>
#include <vector>

using namespace std;

map<string, int> lex = {
    {"PROGRAM", 1}, {"VAR", 2}, {"BEGIN", 3}, {"END", 4}, {"END.", 5},
    {"INTEGER", 6}, {"FOR", 7}, {"READ", 8}, {"WRITE", 9}, {"TO", 10},
    {"DO", 11}, {";", 12}, {":", 13}, {",", 14}, {":=", 15},
    {"+", 16}, {"-", 17}, {"*", 18}, {"DIV", 19}, {"(", 20}, {")", 21},
    {"i", 22}, {"int", 23}
};

vector<int> tokens;
int index1 = 0;
int TOKEN = -1;

vector<int> tokenize(string input) {
    vector<int> tokenList;
    stringstream ss(input);
    string word;
    while (ss >> word) {
        if (lex.find(word) != lex.end()) {
            tokenList.push_back(lex[word]);
        } else {
            cout << "Error: Unknown token '" << word << "'\n";
            exit(1);
        }
```

```
    }
    return tokenList;
}

int get_NextToken() {
    if (index1 < tokens.size()) {
        return tokens[index1++];
    }
    return -1;
}

bool prog();
bool prog_name();
bool dec_list();
bool type();
bool id_list();
bool stmt_list();
bool stmt();
bool assign();
bool exp();
bool term();
bool factor();
bool read();
bool write();
bool for_stmt();
bool index_exp();
bool body();

bool prog() {
    if (TOKEN == 1) { // PROGRAM
        TOKEN = get_NextToken();
        if (prog_name()) {
            if (TOKEN == 2) { // VAR
                TOKEN = get_NextToken();
                if (dec_list()) {
                    if (TOKEN == 3) { // BEGIN
                        TOKEN = get_NextToken();
                        if (stmt_list()) {
                            if (TOKEN == 4) { // END
                                TOKEN = get_NextToken();
                                if (TOKEN == 5) { // END.
                                    TOKEN = get_NextToken();
                                    return index1 == tokens.size();
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return false;
}

```

```
bool prog_name() {
    return TOKEN == 22; // i
}

bool dec_list() {
    if (id_list()) {
        if (TOKEN == 13) { // :
            TOKEN = get_NextToken();
            if (type()) {
                return true;
            }
        }
    }
    return false;
}

bool type() {
    return TOKEN == 6; // INTEGER
}

bool id_list() {
    if (TOKEN == 22) { // i
        TOKEN = get_NextToken();
        while (TOKEN == 14) { // ,
            TOKEN = get_NextToken();
            if (TOKEN != 22) return false;
            TOKEN = get_NextToken();
        }
        return true;
    }
    return false;
}

bool stmt_list() {
    if (stmt()) {
        while (TOKEN == 12) { // ;
            TOKEN = get_NextToken();
            if (!stmt()) return false;
        }
        return true;
    }
    return false;
}

bool stmt() {
    return assign() || read() || write() || for_stmt();
}

bool assign() {
    if (TOKEN == 22) { // i
        TOKEN = get_NextToken();
        if (TOKEN == 15) { // :=
            TOKEN = get_NextToken();
            if (exp()) {
```

```
                    return true;
                }
            }
        }
        return false;
    }

    bool exp() {
        if (term()) {
            while (TOKEN == 16 || TOKEN == 17) { // + or -
                TOKEN = get_NextToken();
                if (!term()) return false;
            }
            return true;
        }
        return false;
    }

    bool term() {
        if (factor()) {
            while (TOKEN == 18 || TOKEN == 19) { // * or DIV
                TOKEN = get_NextToken();
                if (!factor()) return false;
            }
            return true;
        }
        return false;
    }

    bool factor() {
        if (TOKEN == 22 || TOKEN == 23) { // i or int
            TOKEN = get_NextToken();
            return true;
        } else if (TOKEN == 20) { // (
            TOKEN = get_NextToken();
            if (exp()) {
                if (TOKEN == 21) { // )
                    TOKEN = get_NextToken();
                    return true;
                }
            }
        }
        return false;
    }

    bool read() {
        if (TOKEN == 8) { // READ
            TOKEN = get_NextToken();
            if (TOKEN == 20) { // (
                TOKEN = get_NextToken();
                if (id_list()) {
                    if (TOKEN == 21) { // )
                        TOKEN = get_NextToken();
                        return true;
```

```
                    }
                }
            }
        }
        return false;
    }

    bool write() {
        if (TOKEN == 9) { // WRITE
            TOKEN = get_NextToken();
            if (TOKEN == 20) { // (
                TOKEN = get_NextToken();
                if (id_list()) {
                    if (TOKEN == 21) { // )
                        TOKEN = get_NextToken();
                        return true;
                    }
                }
            }
        }
        return false;
    }

    bool for_stmt() {
        if (TOKEN == 7) { // FOR
            TOKEN = get_NextToken();
            if (index_exp()) {
                if (TOKEN == 10) { // TO
                    TOKEN = get_NextToken();
                    if (exp()) {
                        if (TOKEN == 11) { // DO
                            TOKEN = get_NextToken();
                            if (body()) {
                                return true;
                            }
                        }
                    }
                }
            }
        }
        return false;
    }

    bool index_exp() {
        if (TOKEN == 22) { // i
            TOKEN = get_NextToken();
            if (TOKEN == 15) { // :=
                TOKEN = get_NextToken();
                if (exp()) {
                    return true;
                }
            }
        }
        return false;
```

```
248 }
249
250 bool body() {
251     return stmt() || (TOKEN == 3 && stmt_list() && TOKEN == 4); // BEGIN
      stmt-list END
252 }
253
254 void parse(string input) {
255     tokens = tokenize(input);
256     index1 = 0;
257     TOKEN = get_NextToken();
258     if (stmt() && index1 == tokens.size()) { // Changed from prog() to
     stmt()
259         cout << "Valid syntax\n";
260     } else {
261         cout << "Invalid syntax\n";
262     }
263 }
264
265 int main() {
266     string input;
267     cout << "Enter input string: ";
268     getline(cin, input);
269     parse(input);
270     return 0;
271 }
```

5 Code Output

```
Enter input string: i := i * i
Valid syntax

Enter input string: i := i DIV i
Valid syntax

Enter input string: READ ( i , i , i )
Valid syntax

Enter input string: WRITE ( i , i )
Valid syntax

Enter input string: i := i DIV
Invalid syntax

Enter input string: WRITE ( i ,
Invalid syntax
```