PMAT 402 - Systems Programming
Assignment -2
Simplified Pascal Compiler in C++

Gandholi Sarat - 23008

April 10, 2025

Contents

I 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.

II 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

III Code Explanation

The parser code is written in C++ and consists of several main components:

III.I 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}
};
```

III.II 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.

III.III 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;
}
```

III.IV 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.

For 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;
}
```

III.V Parsing Entry Point

The `parse()` function is the entry point. It calls `tokenize()` and begins the syntax validation process. The current configuration uses `stmt()` as the top-level non-terminal to validate single or compound statements.

```
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";
```

```
    }
}
```

III.VI 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;
}
```

IV 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;
}
```

```
35
36 int get_NextToken() {
37     if (index1 < tokens.size()) {
38         return tokens[index1++];
39     }
40     return -1;
41 }
42
43 bool prog();
44 bool prog_name();
45 bool dec_list();
46 bool type();
47 bool id_list();
48 bool stmt_list();
49 bool stmt();
50 bool assign();
51 bool exp();
52 bool term();
53 bool factor();
54 bool read();
55 bool write();
56 bool for_stmt();
57 bool index_exp();
58 bool body();
59
60 bool prog() {
61     if (TOKEN == 1) { // PROGRAM
62         TOKEN = get_NextToken();
63         if (prog_name()) {
64             if (TOKEN == 2) { // VAR
65                 TOKEN = get_NextToken();
66                 if (dec_list()) {
67                     if (TOKEN == 3) { // BEGIN
68                         TOKEN = get_NextToken();
69                         if (stmt_list()) {
70                             if (TOKEN == 4) { // END
71                                 TOKEN = get_NextToken();
72                                 if (TOKEN == 5) { // END.
73                                     TOKEN = get_NextToken();
74                                     return index1 == tokens.size();
75                                 }
76                             }
77                         }
78                     }
79                 }
80             }
81         }
82     }
83     return false;
84 }
85
86 bool prog_name() {
87     return TOKEN == 22; // i
88 }
```

```
89
90  bool dec_list() {
91      if (id_list()) {
92          if (TOKEN == 13) { // :
93              TOKEN = get_NextToken();
94              if (type()) {
95                  return true;
96              }
97          }
98      }
99      return false;
100 }
101
102 bool type() {
103     return TOKEN == 6; // INTEGER
104 }
105
106 bool id_list() {
107     if (TOKEN == 22) { // i
108         TOKEN = get_NextToken();
109         while (TOKEN == 14) { // ,
110             TOKEN = get_NextToken();
111             if (TOKEN != 22) return false;
112             TOKEN = get_NextToken();
113         }
114         return true;
115     }
116     return false;
117 }
118
119 bool stmt_list() {
120     if (stmt()) {
121         while (TOKEN == 12) { // ;
122             TOKEN = get_NextToken();
123             if (!stmt()) return false;
124         }
125         return true;
126     }
127     return false;
128 }
129
130 bool stmt() {
131     return assign() || read() || write() || for_stmt();
132 }
133
134 bool assign() {
135     if (TOKEN == 22) { // i
136         TOKEN = get_NextToken();
137         if (TOKEN == 15) { // :=
138             TOKEN = get_NextToken();
139             if (exp()) {
140                 return true;
141             }
142         }
```

```
143     }
144     return false;
145 }
146 
147 bool exp() {
148     if (term()) {
149         while (TOKEN == 16 || TOKEN == 17) { // + or -
150             TOKEN = get_NextToken();
151             if (!term()) return false;
152         }
153         return true;
154     }
155     return false;
156 }
157 
158 bool term() {
159     if (factor()) {
160         while (TOKEN == 18 || TOKEN == 19) { // * or DIV
161             TOKEN = get_NextToken();
162             if (!factor()) return false;
163         }
164         return true;
165     }
166     return false;
167 }
168 
169 bool factor() {
170     if (TOKEN == 22 || TOKEN == 23) { // i or int
171         TOKEN = get_NextToken();
172         return true;
173     } else if (TOKEN == 20) { // (
174         TOKEN = get_NextToken();
175         if (exp()) {
176             if (TOKEN == 21) { // )
177                 TOKEN = get_NextToken();
178                 return true;
179             }
180         }
181     }
182     return false;
183 }
184 
185 bool read() {
186     if (TOKEN == 8) { // READ
187         TOKEN = get_NextToken();
188         if (TOKEN == 20) { // (
189             TOKEN = get_NextToken();
190             if (id_list()) {
191                 if (TOKEN == 21) { // )
192                     TOKEN = get_NextToken();
193                     return true;
194                 }
195             }
196         }
```

```
197     }
198     return false;
199 }
200 
201 bool write() {
202     if (TOKEN == 9) { // WRITE
203         TOKEN = get_NextToken();
204         if (TOKEN == 20) { // (
205             TOKEN = get_NextToken();
206             if (id_list()) {
207                 if (TOKEN == 21) { // )
208                     TOKEN = get_NextToken();
209                     return true;
210                 }
211             }
212         }
213     }
214     return false;
215 }
216 
217 bool for_stmt() {
218     if (TOKEN == 7) { // FOR
219         TOKEN = get_NextToken();
220         if (index_exp()) {
221             if (TOKEN == 10) { // TO
222                 TOKEN = get_NextToken();
223                 if (exp()) {
224                     if (TOKEN == 11) { // DO
225                         TOKEN = get_NextToken();
226                         if (body()) {
227                             return true;
228                         }
229                     }
230                 }
231             }
232         }
233     }
234     return false;
235 }
236 
237 bool index_exp() {
238     if (TOKEN == 22) { // i
239         TOKEN = get_NextToken();
240         if (TOKEN == 15) { // :=
241             TOKEN = get_NextToken();
242             if (exp()) {
243                 return true;
244             }
245         }
246     }
247     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 }
```

Listing 1: SIC Instruction Parser in C++

V 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
```