

PMAT 402 - Systems Programming

Assignment -3

Simplified Pascal Compiler in C++

Gandholi Sarat - 23008

April 10, 2025

Contents

1	Objective	2
2	Pascal Grammar Given	2
3	Code Explanation	2
3.1	Lexical Mapping	2
3.2	Tokenization	2
3.3	Token Management	3
3.4	Parsing Functions	3
3.5	Parsing Entry Point	3
3.6	Main Function	4
4	Full Code	4
5	Code Output	9

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.

```

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

```

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.

```
1 int get_NextToken () {
2     if (index1 < tokens.size()) {
3         return tokens[index1++];
4     }
5     return -1;
6 }
```

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`:

```
1 bool assign () {
2     if (TOKEN == 22) {
3         TOKEN = get_NextToken ();
4         if (TOKEN == 15) {
5             TOKEN = get_NextToken ();
6             if (exp()) {
7                 return true;
8             }
9         }
10    }
11    return false;
12 }
```

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.

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

```

6     cout << "Valid syntax\n";
7 } else {
8     cout << "Invalid syntax\n";
9 }
10 }

```

3.6 Main Function

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

```

1 int main() {
2     string input;
3     cout << "Enter input string: ";
4     getline(cin, input);
5     parse(input);
6     return 0;
7 }

```

4 Full Code

```

1 #include <iostream>
2 #include <string>
3 #include <map>
4 #include <sstream>
5 #include <vector>
6
7 using namespace std;
8
9 map<string, int> lex = {
10     {"PROGRAM", 1}, {"VAR", 2}, {"BEGIN", 3}, {"END", 4}, {"END.", 5},
11     {"INTEGER", 6}, {"FOR", 7}, {"READ", 8}, {"WRITE", 9}, {"TO", 10},
12     {"DO", 11}, {";", 12}, {":", 13}, {"", 14}, {":=", 15},
13     {"+", 16}, {"-", 17}, {"*", 18}, {"DIV", 19}, {"(", 20}, {")", 21},
14     {"i", 22}, {"int", 23}
15 };
16
17 vector<int> tokens;
18 int index1 = 0;
19 int TOKEN = -1;
20
21 vector<int> tokenize(string input) {
22     vector<int> tokenList;
23     stringstream ss(input);
24     string word;
25     while (ss >> word) {
26         if (lex.find(word) != lex.end()) {
27             tokenList.push_back(lex[word]);
28         } else {
29             cout << "Error: Unknown token '" << word << "'\n";
30             exit(1);
31         }
32     }
33 }

```

```

32     }
33     return tokenList;
34 }
35
36 int getNextToken() {
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 = getNextToken();
63         if (prog_name()) {
64             if (TOKEN == 2) { // VAR
65                 TOKEN = getNextToken();
66                 if (dec_list()) {
67                     if (TOKEN == 3) { // BEGIN
68                         TOKEN = getNextToken();
69                         if (stmt_list()) {
70                             if (TOKEN == 4) { // END
71                                 TOKEN = getNextToken();
72                                 if (TOKEN == 5) { // END.
73                                     TOKEN = getNextToken();
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 }
```

```
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
252     stmt_list END
253 }
254
255 void parse(string input) {
256     tokens = tokenize(input);
257     index1 = 0;
258     TOKEN = get_NextToken();
259     if (stmt() && index1 == tokens.size()) { // Changed from prog() to
260         stmt()
261         cout << "Valid syntax\n";
262     } else {
263         cout << "Invalid syntax\n";
264     }
265 }
266
267 int main() {
268     string input;
269     cout << "Enter input string: ";
270     getline(cin, input);
271     parse(input);
272     return 0;
273 }
```

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