

Department of CSE

SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

14 April 2021

UCS 1602 - Compiler Design

Exercise 7: Generation of Intermediate Code Using Lex and Yacc

Aim:

The new Language Pascal-2021 is introduced with the following programming constructs:

Datatypes

- Integer
- Real
- Char

Declaration Statement

- var: type;
- var: type = constant;

Conditional Statement

- if condition then — else — endif

Generate Intermediate code in the form of Three Address Code sequence for the sample input program written using declaration, conditional and assignment statements in new language Pascal-2021.

Code - Yacc Parser File:

```
1 %{
2     #include <stdio.h>
3     #include <stdlib.h>
4     #include <string.h>
5     #include <math.h>
6
7     int yylex(void);
8     int yyerror(char *);
9     int yywrap();
10
11     int vars = 0, labels = 0;
12
13     struct info{
14         char *var;
15         char *code;
16         int intval;
17         float floatval;
18         char charval;
19     };
20
21     typedef struct info node;
22
23     node *makeNode(){
24         //creating a new node to store intermediate code
25
26         node *n = (node *)malloc(sizeof(node));
27         n->intval = 0;
28         n->floatval = 0;
29         n->charval = 0;
30         n->var = (char *)malloc(50 * sizeof(char));
31         n->code = (char *)malloc(5000 * sizeof(char));
32
33         return n;
34     }
35 %}
36
37 /*Declaration of tokens and precedence*/
38 %token BGN END IF THEN ELSE INT CHAR REALVAR
39 %token REAL CHCONST VAR NUM RELOP ADDOP MULOP
40
41 /*Increasing precedence*/
42 %right MULOP
43 %left ADDOP
44
45 /*Declaration of the types that YYSTYPE can take with the union*/
46 %union{
47     int intval;
```

```

48     float floatval;
49     char ch;
50     char *str;
51     struct info *Node;
52 }
53
54 /*Declaring types for the tokens*/
55 %type<str>      VAR RELOP ADDOP MULOP
56 %type<intval>   NUM
57 %type<floatval> REAL
58 %type<ch>       CHCONST
59 %type<Node>     Program Structure Declarations Statements
60 %type<Node>     Declaration Type Value Statement
61 %type<Node>     Assignment Conditional Condition Expr
62 %type<Node>     E T F
63
64 %%
65
66 Program          :   Structure{
67                     printf("\nL%-5d - |\n%s", 0, $$->code);
68                     }
69                 ;
70
71 Structure         :   Declarations BGN Statements END{
72                     sprintf($$->code, "%s%10s\n%s", $1->code, "|", $3
->code);
73                     }
74                 ;
75
76 Declarations      :   Declaration Declarations{
77                     $$ = makeNode();
78                     sprintf($$->code, "%s%s", $1->code, $2->code);
79                     }
80
81                 |   Declaration{
82                     $$ = $1;
83                     }
84                 ;
85
86 Declaration       :   VAR ':' Type ';' {
87                     $$ = makeNode();
88                     sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $3
->var);
89                     }
90
91                 |   VAR ':' Type '=' Value ';' {
92                     $$ = makeNode();
93                     sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $5
->var);
94                     }
95                 ;

```

```

96
97 Type      :      INT{
98             $$ = makeNode();
99             $$->intval = 0;
100             sprintf($$->var, "%d", 0);
101             sprintf($$->code, "");
102         }
103
104     |      REALVAR{
105             $$ = makeNode();
106             $$->floatval = 0.0;
107             sprintf($$->var, "%.2f", 0.0);
108             sprintf($$->code, "");
109         }
110
111     |      CHAR{
112             $$ = makeNode();
113             $$->charval = 0;
114             sprintf($$->var, "%s", "NULL");
115             sprintf($$->code, "");
116         }
117     ;
118
119 Value      :      NUM{
120             $$ = makeNode();
121             $$->intval = $1;
122             sprintf($$->var, "%d", $1);
123             sprintf($$->code, "");
124         }
125
126     |      REAL{
127             $$ = makeNode();
128             $$->floatval = $1;
129             sprintf($$->var, "%.2f", $1);
130             sprintf($$->code, "");
131         }
132
133     |      CHCONST{
134             $$ = makeNode();
135             $$->intval = $1;
136             sprintf($$->var, "%c", $1);
137             sprintf($$->code, "");
138         }
139     ;
140
141 Statements :      Statement Statements{
142             $$ = makeNode();
143             sprintf($$->code, "%s%s", $1->code, $2->code);
144         }
145
146     |      Statement{

```

```

147         $$ = $1;
148     }
149 ;
150
151 Statement      :   Assignment {
152                 $$ = $1;
153             }
154
155             |   Conditional{
156                 $$ = $1;
157             }
158 ;
159
160 Assignment     :   VAR '=' Expr ';' {
161                 $$ = makeNode();
162                 char tac[100];
163                 sprintf($$->var, "%s", $1);
164                 sprintf(tac, "%10s %-5s := %s\n", "|", $$->var, $3
->var);
165
166                 sprintf($$->code, "%s%s", $3->code, tac);
167             }
168 ;
169 Expr           :   E{
170                 $$ = $1;
171             }
172 ;
173
174 E             :   T MULOP E{
175                 $$ = makeNode();
176                 char tac[100];
177                 sprintf($$->var, "x%d", ++vars);
178                 sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->
var, $1->var, $2, $3->var);
179                 sprintf($$->code, "%s%s%s", $1->code, $3->code,
tac);
180             }
181
182             |   T{
183                 $$ = $1;
184             }
185
186             |   F{
187                 $$ = $1;
188             }
189 ;
190
191 T             :   T ADDOP F{
192                 $$ = makeNode();
193                 char tac[100];
194                 sprintf($$->var, "x%d", ++vars);

```

```

195         sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->
var, $1->var, $2, $3->var);
196         sprintf($$->code, "%s%s%s", $1->code, $3->code,
tac);
197     }
198
199     |   F{
200         $$ = $1;
201     }
202 ;
203
204 F   :   VAR{
205         $$ = makeNode();
206         sprintf($$->var, "%s", $1);
207         sprintf($$->code, "");
208     }
209
210     |   NUM{
211         $$ = makeNode();
212         $$->intval = $1;
213         sprintf($$->var, "%d", $1);
214         sprintf($$->code, "");
215     }
216
217     |   REAL{
218         $$ = makeNode();
219         $$->floatval = $1;
220         sprintf($$->var, "%.2f", $1);
221         sprintf($$->code, "");
222     }
223
224     |   CHCONST{
225         $$ = makeNode();
226         $$->charval = $1;
227         sprintf($$->var, "'%c'", $1);
228         sprintf($$->code, "");
229     }
230 ;
231
232 Conditional   :   IF '(' Condition ')' THEN Statements ELSE Statements
END IF{
233         $$ = makeNode();
234         int condnBlock = ++labels;
235         int endBlock = ++labels;
236         sprintf($$->code, "%s%10s if %s then goto L%d\n%s
%10s goto L%d\n%10s\nL%-5d - |\n%s%10s\nL%-5d - |\n", $3->code, "|", $3
->var, condnBlock, $8->code, "|", endBlock, "|", condnBlock, $6->code,
"|", endBlock);
237     }
238 ;
239

```

```

240 Condition      :      Expr RELOP Expr{
241                  $$ = makeNode();
242                  char tac[100];
243                  sprintf($$->var, "%s%s%s", $1->var, $2, $3->var);
244                  sprintf($$->code, "%s%s", $1->code, $3->code);
245                  }
246                  ;
247 %%
248
249 int yyerror(char* str){
250     printf("\n%s", str);
251     return 0;
252 }
253
254 int yywrap(){
255     return 1;
256 }
257
258 int main(){
259     printf("\n\t\tIntermediate Code Generation\n");
260     printf("\nYour Code:\n\n");
261     system("cat Code.txt");
262     printf("\n\nThree Address Code:\n");
263
264     yyparse();
265     return 0;
266 }
267
268 /*
269 Usage:
270
271 yacc -d -Wnone TAC.y
272 lex TAC.l
273 gcc y.tab.c lex.yy.c -w
274 ./a.out < Code.txt
275
276 */

```

Code - Lex Grammar File:

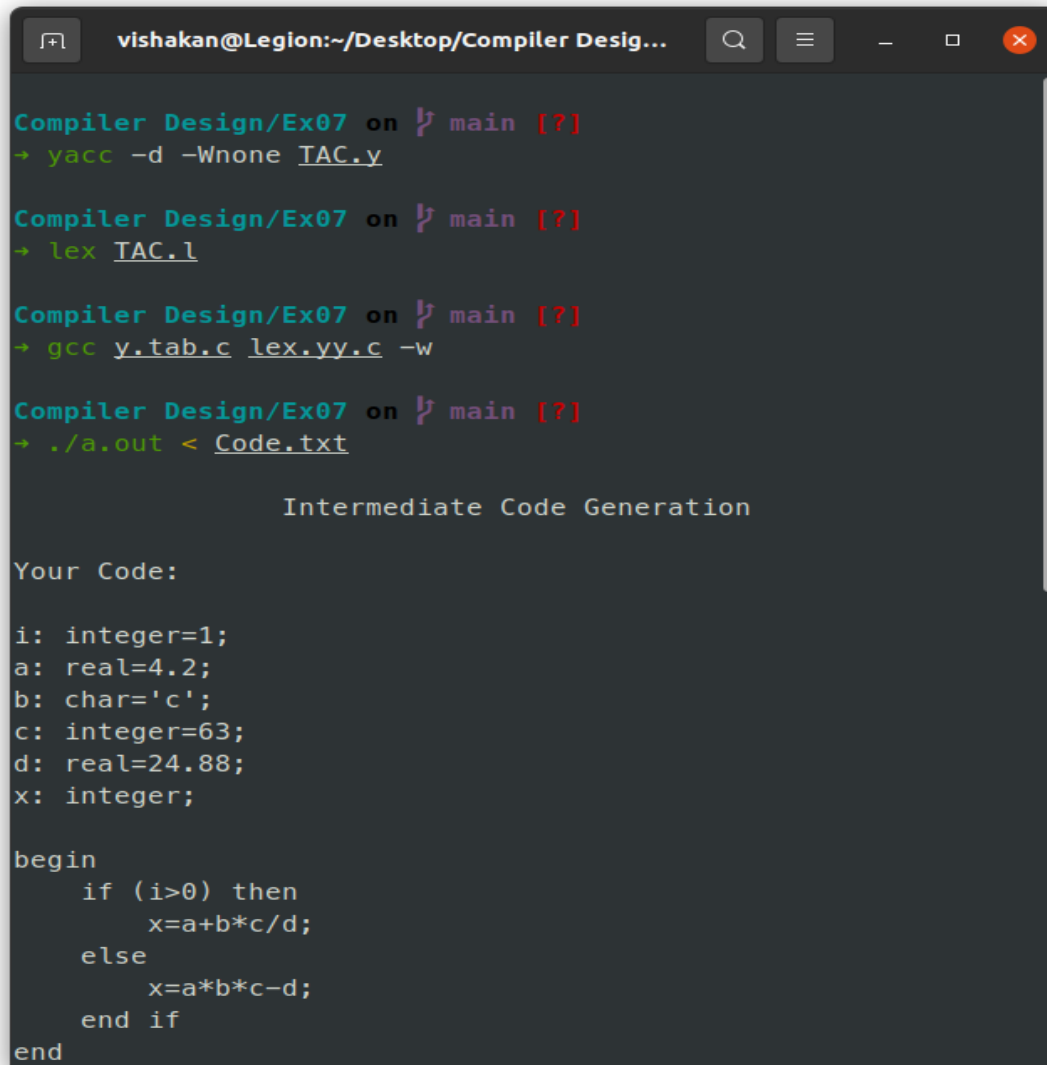
```
1 %{
2     #include <stdio.h>
3     #include <stdlib.h>
4     #include <string.h>
5     #include "y.tab.h"
6 %}
7
8 term      ([a-zA-Z\_][a-zA-Z\_0-9]*)
9 num       ([0-9]+)
10 real      {num}\.{num}
11 relop     ("<" | "<=" | ">" | ">=" | "==" | "!=")
12 addop     ("+" | "-")
13 mulop     ("*" | "/" | "%")
14 spl       (";" | "," | "{" | "}" | "(" | ")" | "=" | "&" | "|" | "!" | ":")
15
16 %%
17 "begin"    {return BGN;}
18 "end"      {return END;}
19 "if"       {return IF;}
20 "then"     {return THEN;}
21 "else"     {return ELSE;}
22 "integer"  {return INT;}
23 "char"     {return CHAR;}
24 "real"     {return REALVAR;}
25 '['. '[']  {yylval.ch = yytext[1]; return CHCONST;}
26 {term}     {yylval.str = strdup(yytext); return VAR;}
27 {real}     {yylval.floatval = atof(yytext); return REAL;}
28 {num}      {yylval.intval = atoi(yytext); return NUM;}
29 {relop}    {yylval.str = strdup(yytext); return RELOP;}
30 {mulop}    {yylval.str = strdup(yytext); return MULOP;}
31 {addop}    {yylval.str = strdup(yytext); return ADDOP;}
32 {spl}      {return *yytext;}
33 [ \t\n]+   {;}
34
35 .          {char errmsg[100];
36             strcpy(errmsg, "Invalid Character: ");
37             strcat(errmsg, yytext);
38             strcat(errmsg, "\n");
39             yyerror(errmsg);}
40
41 %%
```


Sample - Parsed Pascal-2021 Code:

```
1 i: integer=1;
2 a: real=4.2;
3 b: char='c';
4 c: integer=63;
5 d: real=24.88;
6 x: integer;
7 y: real;
8
9 begin
10     if (i>0) then
11         x=a+b*c/d;
12     else
13         x=a*b*c-d;
14     end if
15 end
```

Output 1 - Compilation & Code:

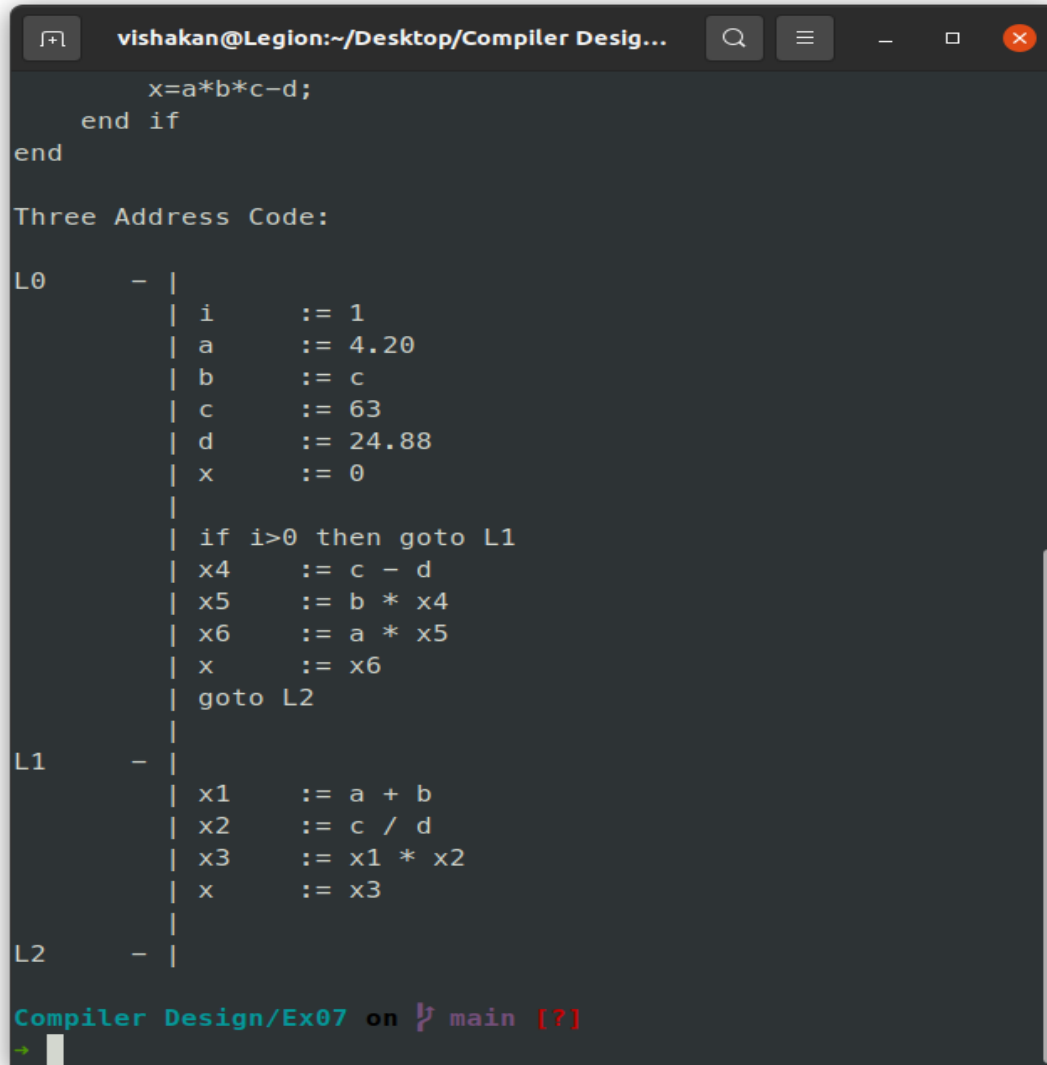
Figure 1: Console Output - Compilation & Code.



```
vishakan@Legion:~/Desktop/Compiler Desig...  
Compiler Design/Ex07 on main [?]  
→ yacc -d -Wnone TAC.y  
  
Compiler Design/Ex07 on main [?]  
→ lex TAC.l  
  
Compiler Design/Ex07 on main [?]  
→ gcc y.tab.c lex.yy.c -w  
  
Compiler Design/Ex07 on main [?]  
→ ./a.out < Code.txt  
  
Intermediate Code Generation  
  
Your Code:  
  
i: integer=1;  
a: real=4.2;  
b: char='c';  
c: integer=63;  
d: real=24.88;  
x: integer;  
  
begin  
  if (i>0) then  
    x=a+b*c/d;  
  else  
    x=a*b*c-d;  
  end if  
end
```

Output 2 - Intermediate Code:

Figure 2: Console Output - Intermediate Code.

A terminal window with a dark background and light text. The window title is "vishakan@Legion:~/Desktop/Compiler Desig...". The output shows a code snippet, followed by the heading "Three Address Code:", and then three labeled blocks of code (L0, L1, L2) separated by vertical lines. At the bottom, it says "Compiler Design/Ex07 on main [?]" with a cursor.

```
x=a*b*c-d;  
end if  
end  
  
Three Address Code:  
  
L0 - |  
    | i      := 1  
    | a      := 4.20  
    | b      := c  
    | c      := 63  
    | d      := 24.88  
    | x      := 0  
    |  
    | if i>0 then goto L1  
    | x4     := c - d  
    | x5     := b * x4  
    | x6     := a * x5  
    | x      := x6  
    | goto L2  
    |  
L1 - |  
    | x1     := a + b  
    | x2     := c / d  
    | x3     := x1 * x2  
    | x      := x3  
    |  
L2 - |  
  
Compiler Design/Ex07 on main [?]  
→
```

Learning Outcome:

- I learnt more theory behind **Yacc Parser Generator**.
- I understood how to construct a grammar for a basic syntax checker.
- I learnt that grammar can be built upon layer by layer, each one adding more detail and complexity.
- I was able to implement the required token recognition with Lex tool.
- I was able to implement the required intermediate code generator with the Yacc tool and Lex tool.
- I understood the use of the %union declaration for **yyval**'s types for passing different values from Lex to Yacc.
- I declared a custom structure to store intermediate code and variables/values and assigned them values while parsing the respective grammar using the \$\$ operator of Yacc.
- I made use of the **sprintf()** function to create intermediate code conveniently.
- I understood that precedences can only be given to tokens in Yacc, and not for grammar symbols.
- I was able to construct intermediate code for conditional block with appropriate grammar definition.
- I came to know that there was no need to return the structures I created inside the parsing of a lower grammar to pass it up to the higher grammar, as it gets implicitly passed up and can be called with the \$ operator.
- I understood that subtle grammar differences need to be made in the Yacc grammar definition to work for right and left associativity & precedences.
- I learnt to call yyerror() with a custom error message within the Lex code.