Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

1 February 2021

UCS1602 - Compiler Design

Exercise 1: Lexical Analyser using C

Objective:

Develop a scanner that will recognize all the above specified tokens. Test your program for all specified tokens. Example input and output specification is given below.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>

int check_keyword(char *token){
    int res = 1;
```

```
    FILE *fp;
    fp = fopen("Keywords.txt", "r");
    if(fp == NULL){
        printf("\nRead Error");
        return 0;
    }
    else{
        char *key = (char*)calloc(100, sizeof(char));
        char ch = fgetc(fp);
        while(ch != EOF){
            if(ch == '\n'){
                res = strcmp(token, key);
                strcpy(key, "");
                if(res == 0)
                    break;
            }
            else{
                strncat(key, &ch, 1);
            }
            ch = fgetc(fp);
        }
    }
    fclose(fp);
    return !res;
}

int check_operator(char *token){
    int res = 0;
    FILE *fp;
    fp = fopen("ArithmeticOp.txt", "r");
    if(fp == NULL){
        printf("\nRead Error");
        return 0;
    }
    else{
        char *key = (char*)calloc(100, sizeof(char));
        char ch = fgetc(fp);
        while(ch != EOF){
            if(ch == '\n'){
                res = strcmp(token, key);
                strcpy(key, "");
                if(res == 0){
                    res++;
                    fclose(fp);
                    return res;
```

```
                }
            }
            else{
                strncat(key, &ch, 1);
            }
            ch = fgetc(fp);
        }
    }
    fclose(fp);
    fp = fopen("RelationalOp.txt", "r");
    if(fp == NULL){
        printf("\nRead Error");
        return 0;
    }
    else{
        char *key = (char*)calloc(100, sizeof(char));
        char ch = fgetc(fp);
        while(ch != EOF){
            if(ch == '\n'){
                res = strcmp(token, key);
                strcpy(key, "");
                if(res == 0){
                    res+=2;
                    fclose(fp);
                    return res;
                }
            }
            else{
                strncat(key, &ch, 1);
            }
            ch = fgetc(fp);
        }
    }
    fclose(fp);
    fp = fopen("LogicalOp.txt", "r");
    if(fp == NULL){
        printf("\nRead Error");
        return 0;
    }
    else{
        char *key = (char*)calloc(100, sizeof(char));
        char ch = fgetc(fp);
        while(ch != EOF){
            if(ch == '\n'){
                res = strcmp(token, key);
```

```
98                  strcpy(key, "");
99                  if(res == 0){
100                     res+=3;
101                     fclose(fp);
102                     return res;
103                 }
104             }
105             else{
106                 strncat(key, &ch, 1);
107             }
108             ch = fgetc(fp);
109         }
110     }
111     fclose(fp);
112 }
113 
114 int check_separator(char token){
115     int res = 0;
116     FILE *fp;
117     fp = fopen("Separators.txt", "r");
118     if(fp == NULL){
119         printf("\nRead Error");
120         return 0;
121     }
122     else{
123         char ch = fgetc(fp);
124         while(ch != EOF){
125             if(ch == token){
126                 res = 1;
127                 break;
128             }
129             ch = fgetc(fp);
130         }
131     }
132     fclose(fp);
133     return res;
134 }
135 
136 char* lexer(char *content){
137     char *lex = (char*)calloc(100, sizeof(char));
138     char *tok = strtok(content, " ");
139 
140     int ctr = 0;
141     char *token_list[100];
142 
```

```
143     for(int i = 0;i<100;i++){
144         token_list[i] = (char*)calloc(100, sizeof(char));
145     }
146 
147     while(tok){
148         strcpy(token_list[ctr], tok);
149         ctr++;
150         tok = strtok(NULL, " ");
151     }
152 
153     for(int j = 0; j < ctr;j++){
154 
155         char *t = (char*)calloc(100, sizeof(char));
156         strcpy(t, token_list[j]);
157 
158         if(t[strlen(t) - 1] == '/' && t[strlen(t) - 2] == '*'
    ){
159             strcat(lex, "MC ");
160             break;
161         }
162 
163         if (t[0] == '/'){
164             if (t[1] == '/')
165             {
166                 strcat(lex, "SC ");
167                 break;
168             }
169             else if (t[1] == '*')
170             {
171                 strcat(lex, "MC ");
172                 break;
173             }
174         }
175         int kw = check_keyword(t); //Check if keyword
176         int op = check_operator(t);//Check if operator
177 
178         if(op == 1){
179             strcat(lex, "ARITHOP ");
180         }
181         else if(op == 2){
182             strcat(lex, "RELOP ");
183         }
184         else if(op == 3){
185             strcat(lex, "LOGICALOP ");
186         }
```

```
187         else if(kw == 1){
188             strcat(lex, "KW ");
189         }
190         else if(strcmp(t, "=") == 0){
191             strcat(lex, "ASSIGN ");
192         }
193         else{
194             char *cp = (char *)calloc(100, sizeof(char));
195             strcpy(cp, t);
196             char *token = strtok(t, "(");
197             int func = check_keyword(token);
198             if(func == 1){
199                 if((strcmp(token, "if") == 0) || (strcmp(
    token, "for") == 0) || (strcmp(token, "while") == 0)){
200                     strcat(lex, "KW SP ");
201                     token = strtok(NULL, "(");
202                     for(int k = 0;token[k];k++){
203                         if(isalpha(token[k])){
204                             strcat(lex, "ID ");
205                             while(isalpha(token[k]) ||
    isdigit(token[k]) || token[k] == '_')
206                                 k++;
207                             k--;
208                         }
209                         else if(token[k] == '=')
210                             strcat(lex, "ASSIGN ");
211                         else if(check_separator(token[k]))
212                             strcat(lex, "SP ");
213                         else if(isdigit(token[k])){
214                             strcat(lex, "NUMCONSTANT ");
215                             while(isdigit(token[k]))
216                                 k++;
217                             k--;
218                         }
219                         else if(token[k] == '\''){
220                             strcat(lex, "CHARCONSTANT ");
221                             k++;
222                             while(token[k] != '\'')
223                                 k++;
224                         }
225                         else if(token[k] == '\"'){
226                             strcat(lex, "STRINGCONSTANT ");
227                             k++;
228                             while(token[k] != '\"')
229                                 k++;
```

```
230                         }
231                         else{
232                             char *c = (char*)calloc(10,
    sizeof(char));
233                             strncpy(c, &token[k], 1);
234                             char next = token[k+1];
235                             if(next == '=' || next == '|' ||
    next == '&'){
236                                 strncat(c, &token[++k], 1);
237                             }
238                             else if(check_operator(&next)>0){
239                                 k++;
240                             }
241                             else;
242 
243                             int check = check_operator(c);
244                             if(check == 1)
245                                 strcat(lex, "ARITHOP ");
246                             else if(check == 2)
247                                 strcat(lex, "RELOP ");
248                             else if(check == 3)
249                                 strcat(lex, "LOGICALOP ");
250                             else
251                                 strcat(lex, "INV ");
252                         }
253                     }
254                 }
255                 else{
256                     strcat(lex, "FC ");
257                     if(strcmp(token, "main")){
258                         int flag = 0;
259                         while(!flag && token_list[j]){
260                             t = token_list[j];
261                             for(int k = 0; token[k];k++){
262                                 if(token[k] == ';')
263                                     flag = 1;
264                             }
265                             j++;
266                         }
267                         j--;
268                     }
269                 }
270             }
271             else{
272                 if (strcmp(token, cp) != 0)
```

```
273                 {
274                     strcat(lex, "FC ");
275                 }
276                 else{
277                     for(int i = 0;token[i];i++){
278                         if(isalpha(token[i])){
279                             strcat(lex, "ID ");
280                             while(isalpha(token[i]) ||
    isdigit(token[i]) || token[i] == '_')
281                                 i++;
282                             i--;
283                         }
284                         else if(token[i] == '=')
285                             strcat(lex, "ASSIGN ");
286                         else if(check_separator(token[i]))
287                             strcat(lex, "SP ");
288                         else if(isdigit(token[i])){
289                             strcat(lex, "NUMCONSTANT ");
290                             while(isdigit(token[i]))
291                                 i++;
292                             i--;
293                         }
294                         else{
295                             char *c = (char*)calloc(10,
    sizeof(char));
296                             strncpy(c, &token[i], 1);
297                             char next = token[i+1];
298                             if(next == '=' || next == '|' ||
    next == '&'){
299                                 strncat(c, &token[++i], 1);
300                             }
301                             else if(check_operator(&next)>0)
302                                 i++;
303                             else;
304
305                             int check = check_operator(c);
306                             if(check == 1)
307                                 strcat(lex, "ARITHOP ");
308                             else if(check == 2)
309                                 strcat(lex, "RELOP ");
310                             else if(check == 3)
311                                 strcat(lex, "LOGICALOP ");
312                             else
313                                 strcat(lex, "INV ");
314
```

```
                        }
                    }
                }
            }
        }
    }
    return lex;

}

int line_count(char *file){
    FILE *fp;
    int count = 0;
    fp = fopen(file, "r");

    if (fp == NULL){
        return 0;
    }
    for(char c = getc(fp); c != EOF; c = getc(fp))
        if (c == '\n')
            count = count + 1;
    fclose(fp);
    return count;
}

int main(){
    FILE *fp;
    char ch;
    char *filename = (char*)calloc(100, sizeof(char));
    char *content = (char*)calloc(100, sizeof(char));
    char *copy = (char*)calloc(100, sizeof(char));
    char *lex = (char*)calloc(200, sizeof(char));

    /*Single line
    scanf(" %[^\n]", content);
    strcpy(copy, content);
    strcpy(lex, lexer(copy));

    printf("Ip: %s\n", content);
    printf("Op: %s\n", lex);
    */
    //File
    printf("\nEnter file name:");scanf(" %[^\n]", filename);

```

```
359     printf("
        ------------------------------------------------------------------------
        \n");
360     printf("FC: Function call\n");
361     printf("KW: Keyword\n");
362     printf("ID:identifier");
363     printf("RELOP: Relational operator\n");
364     printf("LOGICALOP: Logical Operator\n");
365     printf("ARITHOP: Arithmetic Operator\n");
366     printf("SP:Separator\n");
367     printf("
        ------------------------------------------------------------------------
        \n");
368
369     fp = fopen(filename, "r");
370     fscanf(fp, " %[^\n]", content);
371     int c = 0;
372     while(c < line_count(filename)){
373         strcpy(copy, content);
374         strcpy(lex, lexer(copy));
375         printf("%s\n", lex);
376         fscanf(fp, " %[^\n]", content);
377         c++;
378     }
379     fclose(fp);
380
381
382 }
```

Input file:

```
/*Multiline
comment*/
main()
{
  int a=10,b=20;
  if(a!=b)
     printf(  a is greater );
   else
     printf(  b is greater );
}
add()
{
  int a = 10;
}
//Single line comment
```

Output:

```
Enter file name:file.c
________________________________________________________________________
FC: Function call
KW: Keyword
ID:identifierRELOP: Relational operator
LOGICALOP: Logical Operator
ARITHOP: Arithmetic Operator
SP:Separator
________________________________________________________________________
MC
MC
FC
SP
KW ID ASSIGN NUMCONSTANT SP ID ASSIGN NUMCONSTANT SP
KW SP ID RELOP ID SP
FC
KW
FC
SP
FC
SP
KW ID ASSIGN NUMCONSTANT SP
SP
```

Learning Outcomes:

- Understood the basic working of a Lexical Analyser.
- Learnt to parse a program for detection and identification of tokens.
- Learnt to match regular expressions.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

1 February 2021

UCS1602 - Compiler Design

Exercise 2: Implementation of Lexical Analyzer

Objective:

Develop a Lexical analyzer to recognize the patterns namely, identifiers,constants, comments and operators using the following regular expressions.Construct symbol table for the identifiers with the following information

Code:

```
/*Inclusion*/
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>

```

```
7       int symbol_count  = 0, flag=0, fg[20],base = 1000;
8       char *symbol_table[100];
9       char *values[100];
10
11  void set_const(char *val){
12      strcpy(val, yytext);
13  }
14  void set_flag(int *flag){
15      if(strcmp(yytext, "int") == 0)
16          *flag = 1;
17      else if(strcmp(yytext, "float") == 0)
18          *flag = 2;
19      else if(strcmp(yytext, "double") == 0)
20          *flag = 3;
21      else if(strcmp(yytext, "char") == 0)
22          *flag = 4;
23  }
24
25  void construct_table(char *symbol_table[], int *symbol_count)
        {
26      int size = 0;
27      int addr = 1000;
28      symbol_table[*symbol_count] = (char*)calloc(100, sizeof(
        char));
29      strcat(symbol_table[*symbol_count], yytext);strcat(
        symbol_table[*symbol_count], " ");
30      if(flag == 1){
31          strcat(symbol_table[*symbol_count], "int");strcat(
        symbol_table[*symbol_count], " ");
32          size = 2;
33      }
34      else if(flag == 2){
35          strcat(symbol_table[*symbol_count], "float");strcat(
        symbol_table[*symbol_count], " ");
36          size = 4;
37      }
38      else if(flag == 3){
39          strcat(symbol_table[*symbol_count], "double");strcat(
        symbol_table[*symbol_count], " ");
40          size = 8;
41      }
42      else if(flag == 4){
43          strcat(symbol_table[*symbol_count], "char");strcat(
        symbol_table[*symbol_count], " ");
44          size = 1;
```

```
45      }
46      char *dummy=(char*)calloc(100, sizeof (char));
47      sprintf(dummy, "%d", size);
48      strcat(symbol_table[*symbol_count], dummy);strcat(
        symbol_table[*symbol_count], " ");
49      sprintf(dummy, "%d", base_addr);base_addr += size;
50      strcat(symbol_table[*symbol_count], dummy);strcat(
        symbol_table[*symbol_count], " ");
51      strcat(symbol_table[*symbol_count], val);strcat(
        symbol_table[*symbol_count], " ");
52  }
53  %}
54  /*Rules*/
55
56  /*Preprocessor directives*/
57  inc #(.)*
58
59
60  /*Keywords*/
61  kw int|char|float|double|if|else|for|while|do
62
63  /*Function*/
64  funcCall [a-zA-Z]([a-zA-Z]|[0-9])*\(
65
66  /*ID*/
67  id [a-zA-Z]([a-zA-Z]|[0-9])*
68
69  /*Constant*/
70
71  numConst [0-9]+
72  charConst \'[a-zA-Z]\'
73  strConst \"[a-z A-Z]*\"
74
75  /*Comments*/
76  single \/\/(.)*
77  multi \/\*(.*\n?)*\*\/
78
79  /*Operators*/
80  relOp <|<=|>|>=|==|!=
81  arithOp "+"|"-"|"*"|"/"|"%"
82  logicOp &&|\|\||!
83
84  /*Separators*/
85  sep [!@#$^&(){};:,]
86
```

```
/* Pattern Action pairs*/
%%
{inc} {printf("PREDIR ");}
{relOp} {printf("RELOP ");}
{arithOp} {printf("ARITHOP ");}
{logicOp} {printf("LOGOP ");}
{numConst} {printf("NUMCONST "); set_const(val);}
{charConst} {printf("CHARCONST ");set_const(val);}
{strConst} {printf("STRCONST "); set_const(val);}
{single} {printf("SC ");}
{multi} {printf("MC ");}
{kw} {printf("KW "); set_flag(&flag);}
{funcCall} {printf("FC ");}
{id} {printf("ID "); construct_table(symbol_table, &
    symbol_count);}
{sep} {printf("SP ");}
"=" {printf("ASSIGN ");}
"\n" {printf("\n");}
%%

int yywrap(void){}

void printTable(char *symbol_table[100], int symbol_count){
    for(int i = 0; i<symbol_count;i++){
        char *token = strtok(symbol_table[i], " ");
        while(token){
            printf("%s ", token);
            token = strtok(NULL, " ");
        }
        printf("\n");
    }
}
int main(){
    char *name = (char*)calloc(100, sizeof(char));
    printf("Enter filename: ");scanf(" %[^\n]", name);

    yyin = fopen(name, "r+");
    yylex();

    printTable(symbol_table, symbol_count);
    return 0;
}
```

Input file:

```
#include<stdio.h>
/*Multiline
comment*/
main()
{
  float c = 20;
  int a=10,b=20;

  if (a != b)
      printf(  a is greater );
  else
      printf(  b is greater );
}
add()
{
  int a = 10;
}
//Single line comment
```

Output:

```
Enter filename: file.c
PREDIR
MC
FC
SP
  KW  ID  ASSIGN  NUMCONST SP
  KW  ID ASSIGN NUMCONST SP ID ASSIGN NUMCONST SP

  KW  SP ID  RELOP  ID SP
      FC SP
  KW
      FC SP
SP
FC
SP
  KW  ID  ASSIGN  NUMCONST SP
SP
SC
 Name  Type  Size  Addr Value
    c float     4  1000 20
    a   int     2  1004 10
    b   int     2  1006 20
```

Learning Outcomes:

- Understood the basic working of Lex tool for tokenising.
- Learnt how to construct symbol table from code using lex tool.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

1 February 2021

UCS1602 - Compiler Design

Exercise 3: Elimination of Immediate LeftRecursion using C

Objective:

Write a program in C to find whether the given grammar is LeftRecursive or not. If it is found to be left recursive, convert the grammar in such a way that the left recursion is removed.

Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int elim_lr(char* production){

```

```
    char* prod = (char*)calloc(100, sizeof(char));
    strcpy(prod, production);

    char* token = strtok(prod, "->");
    char sym = token[0];

    token = strtok(NULL, "->");

    char* tok = strtok(token, "|");
    char *alpha[10];
    int al = 0;

    char *beta[10];
    int be = 0;
    while(tok){
        if(sym == tok[0]){
            alpha[al] = (char *)calloc(100, sizeof(char));
            for(int i = 1; tok[i]; i++){
                alpha[al][i-1] = tok[i];
            }
            al++;
        }
        else{
            beta[be++] = (char*)calloc(100, sizeof(char));
            strcpy(beta[be-1], tok);
        }
        tok = strtok(NULL, "|");
    }

    if(be == 0){
        printf("%s is a Left Recursive production, but cannot
    be reduced", production);
        return 0;
    }

    printf("%c -> ", sym);
    for(int i = 0;i<be;i++){
        printf("%s%c'", beta[i], sym);
        if(i+1 != be)
            printf(" | ");
    }
    printf("\n");
    printf("%c' -> epsilon| ", sym);
    for (int i = 0; i < al;i++){
        printf("%s%c'", alpha[i], sym);
```

```
51          if(i+1 != al)
52              printf(" | ");
53      }
54      printf("\n");
55  }
56
57  int check_lr(char* production){
58      char* prod = (char*)calloc(100, sizeof(char));
59      strcpy(prod, production);
60      char *token = strtok(prod, "->");
61      char sym = token[0];
62      token = strtok(NULL, "->");
63      if(sym == token[0])
64          elim_lr(production);
65      else
66          printf("%s\n", production);
67  }
68
69  int line_count(char *file){
70      FILE *fp;
71      int count = 0;
72      fp = fopen(file, "r");
73
74      if (fp == NULL){
75          return 0;
76      }
77      for(char c = getc(fp); c != EOF; c = getc(fp))
78          if (c == '\n')
79              count = count + 1;
80      fclose(fp);
81      return count;
82  }
83
84  int main(){
85      char *file_name = (char*)calloc(100, sizeof(char));
86      char *production = (char *)calloc(100, sizeof(char));
87      printf("\nEnter file name: ");
88      scanf(" %[^\n]", file_name);
89
90      FILE *fp;
91      fp = fopen(file_name, "r+");
92      int ctr = 0;
```

Input file:

```
A->AB1|AB0|1
B->B1|BA0|0
E->E*T
```

Output:

```
Enter file name: file2
A->B0A' | 1A'
A'->epsilon| B1A'
B->A0B' | 0B'
B'->epsilon| 1B'
```

Learning Outcomes:

- Understood the basic concept of left recursion and need for its elimination.
- Learnt how to remove left-recursion from specified grammar using C.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

26 February 2021

UCS1602 - Compiler Design

Exercise 4: Recursive Descent Parser using C

Objective:

Write a program in C to construct Recursive Descent Parser for the following grammar which is for arithmetic expression involving + and *. Check the Grammar for left recursion and convert into suitable for this parser. Write recursive functions for every non-terminal. Call the function for start symbol of the Grammar in main(). Extend this parser to include division, subtraction and parenthesis operators

Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

```

```
void tab(int val){
    while(val--)
        printf("\t");
}
int F(char *, int *, int);
int Tprime(char *, int *, int);
int T(char *, int *, int);
int Eprime(char *, int *, int);
int E(char *, int *, int);

int main(){

    char *str = (char*)calloc(100, sizeof(char));
    printf("\nEnter string to parse: ");
    scanf(" %s", str);
    strcat(str, "$");
    int look_ahead = 0;

    printf("----------------------------\n");
    printf("Enter E\n");
    E(str, &look_ahead, 1);
    printf("Exit E\n");
    printf("----------------------------\n");

    if (str[look_ahead] == '$')
        printf("\nSuccess");
    else
        printf("\nFailure: %c at position %d not expected. \n
   ", str[look_ahead], look_ahead);
}

int F(char *str, int *look_ahead, int level)
{
    if (str[*look_ahead] == 'i')
    {
        tab(level);
        printf("F: i matched\n");
        (*look_ahead)++;
    }
    else if (str[*look_ahead] == '(')
    {
        tab(level);
        printf("F: ( matched\n");
        (*look_ahead)++;
        E(str, look_ahead, level + 1);
```

```
49          if (str[*look_ahead] == ')')
50          {
51              tab(level);
52              printf("F: ) matched\n");
53              (*look_ahead)++;
54          }
55      }
56  }
57
58  int Tprime(char *str, int *look_ahead, int level){
59      if(str[*look_ahead] == '*'){
60          tab(level);
61          printf("T': * matched\n");
62          (*look_ahead)++;
63
64          tab(level);
65          printf("----------------------------\n");
66          tab(level);
67          printf("Enter F\n");
68          F(str, look_ahead, level + 1);
69          tab(level);
70          printf("Exit F\n");
71          tab(level);
72          printf("----------------------------\n");
73
74          tab(level);
75          printf("----------------------------\n");
76          tab(level);
77          printf("Enter T'\n");
78          Tprime(str, look_ahead, level + 1);
79          tab(level);
80          printf("Exit T'\n");
81          tab(level);
82          printf("----------------------------\n");
83
84      }
85      else if(str[*look_ahead] == '/'){
86          tab(level);
87          printf("T': / matched\n");
88          (*look_ahead)++;
89
90          tab(level);
91          printf("----------------------------\n");
92          tab(level);
93          printf("Enter F\n");
```

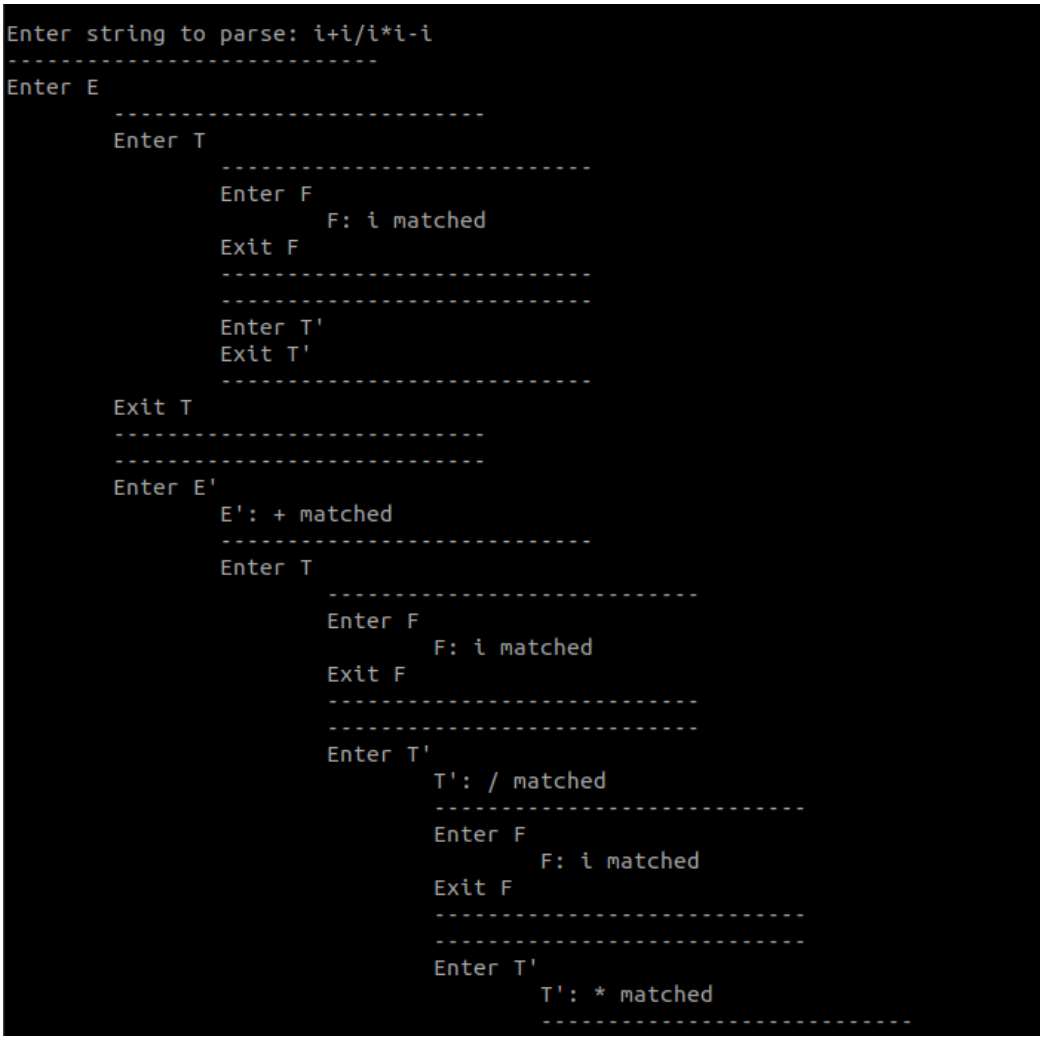
```
94          F(str, look_ahead, level + 1);
95          tab(level);
96          printf("Exit F\n");
97          tab(level);
98          printf("----------------------------\n");
99
100         tab(level);
101         printf("----------------------------\n");
102         tab(level);
103         printf("Enter T'\n");
104         Tprime(str, look_ahead, level + 1);
105         tab(level);
106         printf("Exit T'\n");
107         tab(level);
108         printf("----------------------------\n");
109
110     }
111 }
112
113 int T(char *str, int *look_ahead, int level){
114     tab(level);
115     printf("----------------------------\n");
116     tab(level);
117     printf("Enter F\n");
118     F(str, look_ahead, level + 1);
119     tab(level);
120     printf("Exit F\n");
121     tab(level);
122     printf("----------------------------\n");
123
124     tab(level);
125     printf("----------------------------\n");
126     tab(level);
127     printf("Enter T'\n");
128     Tprime(str, look_ahead, level + 1);
129     tab(level);
130     printf("Exit T'\n");
131     tab(level);
132     printf("----------------------------\n");
133 }
134 int Eprime(char *str, int *look_ahead, int level){
135
136     if(str[*look_ahead] == '+'){
137         tab(level);
138         printf("E': + matched\n");
```

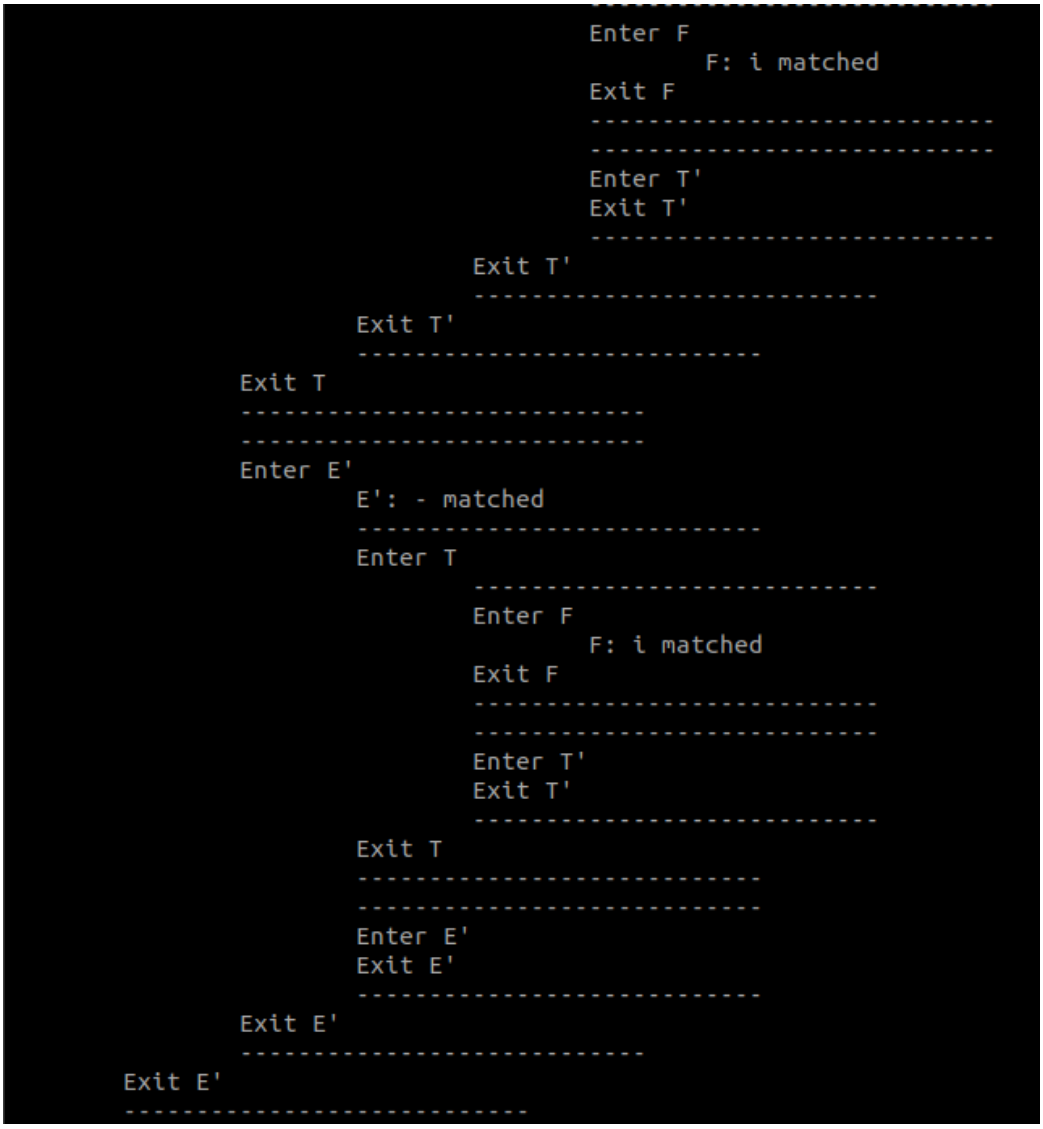
```
139         (*look_ahead)++;
140 
141         tab(level);
142         printf("----------------------------\n");
143         tab(level);
144         printf("Enter T\n");
145         T(str, look_ahead, level + 1);
146         tab(level);
147         printf("Exit T\n");
148         tab(level);
149         printf("----------------------------\n");
150 
151         tab(level);
152         printf("----------------------------\n");
153         tab(level);
154         printf("Enter E'\n");
155         Eprime(str, look_ahead, level + 1);
156         tab(level);
157         printf("Exit E'\n");
158         tab(level);
159         printf("----------------------------\n");
160     }
161     else if(str[*look_ahead] == '-'){
162         tab(level);
163         printf("E': - matched\n");
164         (*look_ahead)++;
165 
166         tab(level);
167         printf("----------------------------\n");
168         tab(level);
169         printf("Enter T\n");
170         T(str, look_ahead, level + 1);
171         tab(level);
172         printf("Exit T\n");
173         tab(level);
174         printf("----------------------------\n");
175 
176         tab(level);
177         printf("----------------------------\n");
178         tab(level);
179         printf("Enter E'\n");
180         Eprime(str, look_ahead, level + 1);
181         tab(level);
182         printf("Exit E'\n");
183         tab(level);
```

```
184         printf("----------------------------\n");
185     }
186 }
187
188 int E(char *str, int *look_ahead, int level){
189     tab(level);
190     printf("----------------------------\n");
191     tab(level);
192     printf("Enter T\n");
193     T(str, look_ahead, level+1);
194     tab(level);
195     printf("Exit T\n");
196     tab(level);
197     printf("----------------------------\n");
198
199     tab(level);
200     printf("----------------------------\n");
201     tab(level);
202     printf("Enter E'\n");
203     Eprime(str, look_ahead, level+1);
204     tab(level);
205     printf("Exit E'\n");
206     tab(level);
207     printf("----------------------------\n");
208 }
```

Output:

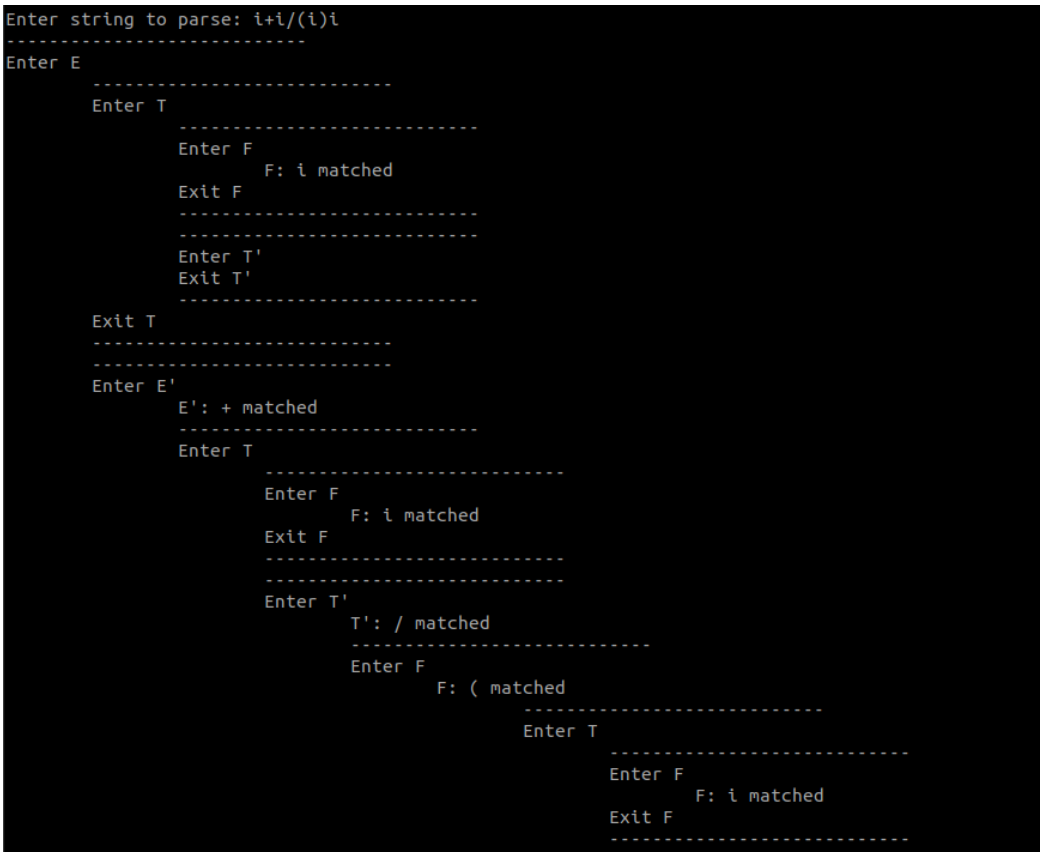
Success scenario:

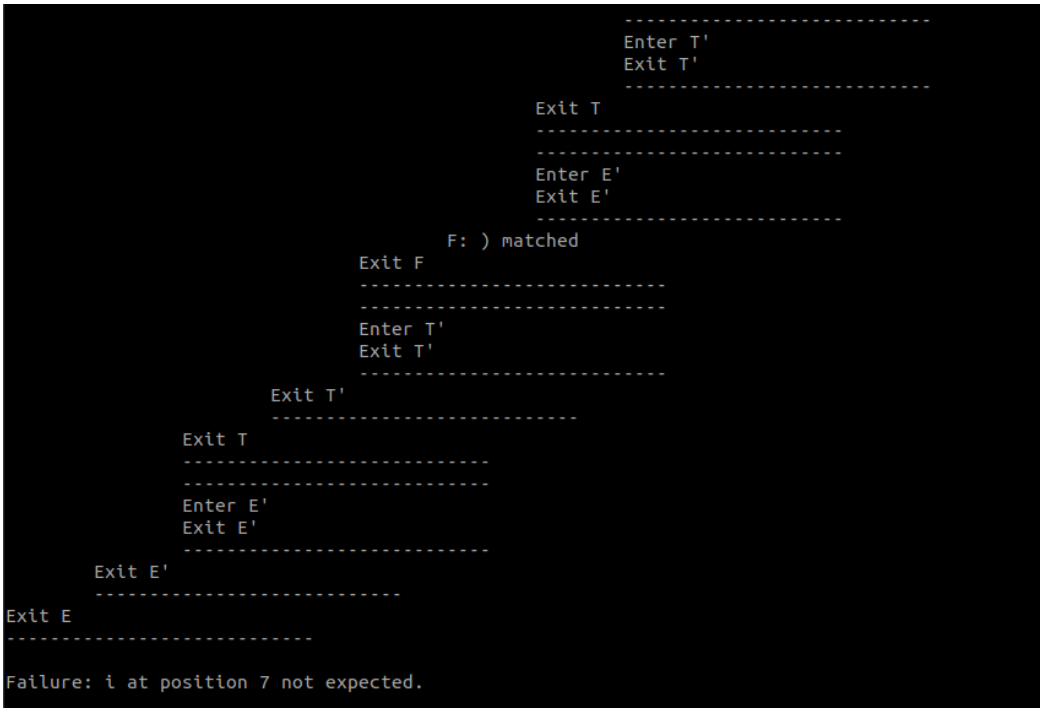



Enter F
F: i matched
Exit F
Enter T'
Exit T'
Exit T'
Exit T'
Exit T
Enter E'
E': - matched
Enter T
Enter F
F: i matched
Exit F
Enter T'
Exit T'
Exit T
Enter E'
Exit E'
Exit E'
Exit E'


Exit E
Success%

Failure scenario:





Learning Outcomes:

- Understood the basic working of recursive descent parser.
- Learnt how to use left-recursion-eliminated grammar to write code for recursive descent parser.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

5 March 2021

UCS1602 - Compiler Design

Exercise 5: Implementation of Desk Calculator using YaccTool

Objective:

Write Lex program to recognize relevant tokens required for the Yacc parser to implement desk calculator. Write the Grammar for the expression involving the operators. Precedence and associativity has to be preserved. Yacc is available as a command in linux. The grammar should have non-terminals E, Op and a terminal id.

Code:

Lex:

```
%{
    #include<stdio.h>
    #include "y.tab.c"
    extern YYSTYPE yylval;
%}

%%
[0-9]+ {yylval = atoi(yytext); return NUM;}
[\t] ;

[\n] return 0;

. return yytext[0];
%%
int yywrap(){
    return 1;
}
```

Yacc:

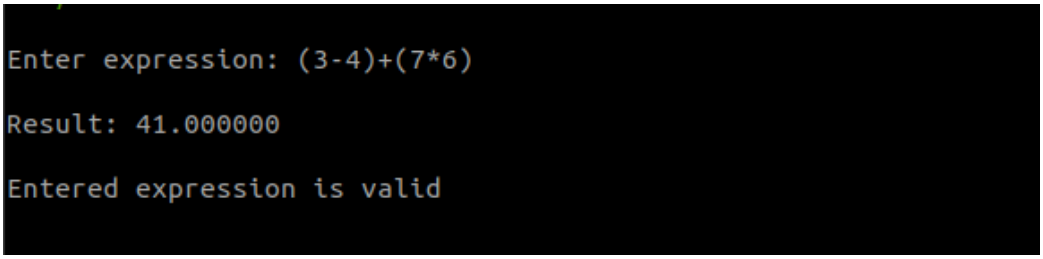
```
%{
    #include<stdio.h>
    #define YYSTYPE double
    int flag = 0;

    int yylex(void);

    double pow(double x, double y){
        double pdt = 1.0;
        while(y--){
            pdt *= x;
        }
        return pdt;
    }
    int op = 0;

%}

%token  NUM

%left '|'
%left '&'
```

```
%right '!'

%left '>' '<' '='

%left '+' '-'
%left '/' '*' '%'
%right '^'
%left '(' ')'

%%
P : E {printf("\nResult: %lf\n", $$);}
E : E '+' E {$$ = $1 + $3;}
  | E '-' E {$$ = $1 - $3;}
  | E '*' E {$$ = $1 * $3;}
  | E '/' E {$$ = $1 / $3;}
  | E '^' E {$$ = pow($1, $3);}
  |'('E')' {$$=$2;}
  | NUM {$$ = $1;}

E : E GR E {if($1 > $3){$$=1;} else{$$=0;}}
  | E GRE E {if($1 >= $3){$$=1;} else{$$=0;}}
  | E LE E {if($1 < $3){$$=1;} else{$$=0;}}
  | E LEE E {if($1 <= $3){$$=1;} else{$$=0;}}
  | E EQ E {if($1 == $3){$$=1;} else{$$=0;}}
  | E NEQ E {if($1 != $3){$$=1;} else{$$=0;}}

GR : '>'
GRE : '>''='
LE : '<'
LEE : '<''='
EQ : '=''='
NEQ : '!''='

E : E AND E {$$ = $1 * $3;}
  | E OR E {if($1==1||$3 ==1){$$=1;}else{$$=0;}}
  | NOT E { if($2==1){ $$=0; }else{ $$=1;}}

AND : '&''&'
OR : '|''|'
NOT : '!'

E : E LSHIFT E {$$ = (int)$1 << (int)$3;}
  | E RSHIFT E {$$ = (int)$1 >> (int)$3;}
  | E BAND E {$$ = (int)$1 & (int)$3;}
  | E BOR E {$$ = (int)$1 | (int)$3;}
```

```
68   | BNOT E {$$ = ~(int)$1;}
69
70 LSHIFT : '<''<'
71 RSHIFT : '>''>'
72 BAND : '&'
73 BOR : '|'
74 BNOT : '~'
75
76
77 ;
78 %%
79 int yyerror ()
80 {
81     printf("\nEntered expression is invalid\n\n");
82    flag=1;
83
84 }
85
86 int main (void){
87     printf("\nEnter expression: ");
88     yyparse();
89     if(flag==0)
90         printf("\nEntered expression is valid\n\n");
91     return 0;
92 }
```

Output:

Arithmetic Expression:



Boolean Expression:

```
Enter expression: 25==25

Result: 1.000000

Entered expression is valid
```

```
Enter expression: 25<=27

Result: 1.000000

Entered expression is valid
```

Bitwise operation:

```
Enter expression: 9<<1

Result: 18.000000

Entered expression is valid
```

```
Enter expression: 5&9

Result: 1.000000

Entered expression is valid
```

Learning Outcomes:

- Understood the basic working of Yacc tool.
- Learnt how to specify grammar in yacc.
- Learnt to use yacc efficiently to to perform actions for each grammatical structure.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

29 March 2021

UCS1602 - Compiler Design

Exercise 6:Implementation of Syntax Checker using YaccTool

Objective:

Develop a Syntax checker to recognize the tokens necessary for the following statements by writing suitable grammars Assignment statement Conditional statement Looping statement

Code:

Lex:

```
%{
    #include<stdio.h>
    #include "y.tab.c"
    extern YYSTYPE yylval;
%}

kw int|char|float|double|while|do
if if
else else
for for
num [0-9]+
id [a-z][a-z]*

%%
{num} {return NUM;}
{kw} {return KW;}
{if} {return IF;}
{for} {return FOR;}
{else} {return ELSE;}
"(" {return POPEN;}
")" {return PCLOSE;}
"{" {return BOPEN;}
"}" {return BCLOSE;}
{id} {return ID;}
("+="|"-="|"*="|"/="|"=") {return AOP;}
("++"|"--") {return CHANGE_OP;}
("=="|"!="|">"|"<"|">="|"<=") {return ROP;}
";" {return SEP;}
[+\-^*/,().] {return *yytext;}
[\t]
[ ]
[\n]

. return yytext[0];
%%
int yywrap(){
    return 1;
}
```

Yacc:

```
%{
```

```
2     #include<stdio.h>
3     #define YYSTYPE double
4
5     int flag = 0;
6     int yylex(void);
7 %}
8
9 %token NUM ID KW AOP
10 %token IF ELSE ROP
11 %token POPEN PCLOSE BOPEN BCLOSE
12 %token FOR WHILE
13 %token SEP
14 %token CHANGE_OP
15
16
17 %%
18 stmt : assn_stmt
19   | cond_stmt
20   | loop_stmt
21 ;
22 assn_stmt : ID AOP expr {printf("\nAssignment statement found
       \n");}
23 ;
24 expr : expr '+' expr
25   | expr '-' expr
26   | expr '*' expr
27   | expr '/' expr
28   | NUM
29   | ID
30 ;
31
32 cond_stmt : IF cond stmt continue {printf("\nConditional
       statement found\n");}
33 ;
34 cond : POPEN rel_expr PCLOSE
35 ;
36 continue : ELSE stmt
37          |
38 ;
39 rel_expr : expr ROP expr
40 ;
41
42 loop_stmt : for_stmt
43           | while_stmt
44 ;
```

```
45 for_stmt : FOR POPEN assn_stmt SEP rel_expr SEP inc_expr
       PCLOSE BOPEN stmt BCLOSE {printf("\nLooping statement
       found\n");}
46 ;
47
48 inc_expr :  assn_stmt
49             | expr CHANGE_OP
50 ;
51 while_stmt : WHILE cond BOPEN stmt BCLOSE
52 ;
53
54
55 %%
56
57 int yyerror (char const* s)
58 {
59     printf("\nSyntactically Incorrect: %s\n", s);
60     flag=1;
61 }
62
63 int main(int argc, char **argv){
64     if(argc != 2){
65         fprintf(stderr, "Enter file name as argument!\n");
66         return 1;
67     }
68     yyin = fopen(argv[1], "rt");
69     if (!yyin){
70         fprintf(stderr, "File not found!\n");
71         return 2;
72     }
73     yyparse();
74     if(flag==0)
75         printf("\nSyntactically correct\n");
76     return 0;
77 }
```

Output:

Correct syntax:

```
for(i = 0;i < 10; i++){
    if(x < 10)
        x += 8
    else
        y -= 9
}
```

```
Assignment statement found
Assignment statement found
Assignment statement found
Conditional statement found
Looping statement found
Syntactically correct
```

Incorrect syntax:

```
for(i = 0;i < 10; i++){
    if(x < 10)
        x += 8
    else
        y -= 9;
}
```

```
Assignment statement found

Assignment statement found

Assignment statement found

Conditional statement found

Syntactically Incorrect: syntax error
```

Learning Outcomes:

- Understood the basic concept of Syntax Checker.
- Learnt how to identify control structures using yacc and lex.
- Learnt to use yacc efficiently for specifying grammar.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

16 April 2021

UCS1602 - Compiler Design

Exercise 7:Generation of Intermediate Code using Lex and Yacc

Objective:

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:

Lex:

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "y.tab.h"
%}
%option yylineno

num [0-9]+
real {num}\.{num}

if if
else else
then then
begin begin
end end

rel_op ("<"|"<="|">"|">="|"=="|"!=")
add_op ("+"|"-")
mul_op ("*"|"/"|"%")
assn_op ("+="|"-="|"*="|"/="|"=")

id [a-z][a-z]*
spl (";"|","|"{"|"}"|"("|")"|"="|"&"|"|"|"!"|":")

%%
{num} {yylval.int_val = atoi(yytext);return INT_CONST;}
{real} {yylval.float_val = atof(yytext);return REAL_CONST;}
['].['] {yylval.char_val = yytext[1];return CHAR_CONST;}

"integer" {return INT;}
"real" {return REAL;}
"char" {return CHAR;}


"(" {return POPEN;}
")" {return PCLOSE;}

{if} {return IF;}
{else} {return ELSE;}
{then} {return THEN;}
{begin} {return BGN;}
{end} {return END;}

{rel_op} {yylval.str = strdup(yytext); return REL_OP;}
```

```
{mul_op} {yylval.str = strdup(yytext); return MUL_OP;}
{add_op} {yylval.str = strdup(yytext); return ADD_OP;}

{id} {yylval.str = strdup(yytext);return ID;}
{spl} {return *yytext;}
[\t\n]+  {;}
" " {;}
. {
    char errmsg[100];
    sprintf(errmsg, "Invalid Character: %s at line %d",
   yytext, yylineno);
    strcat(errmsg, "\n");
    yyerror(errmsg);
  }
%%
```

Yacc:

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <math.h>

    int yylex(void);
    int yyerror(char *);
    int yywrap();

    int tmp = 0;
    int jump = 0;

    struct info{
        char *var;
        char *code;
        int int_val;
        float float_val;
        char char_val;
    };

    typedef struct info node;

    node *makeNode(){
```

```
25          node *n = (node*)calloc(1, sizeof(node));
26          n->int_val = 0;
27          n->float_val = 0;
28          n->char_val = 0;
29          n->var = (char*)calloc(50, sizeof(char));
30          n->code = (char*)calloc(5000, sizeof(char));
31          return n;
32      }
33  %}
34
35  %token BGN END
36  %token INT REAL CHAR
37  %token INT_CONST REAL_CONST CHAR_CONST
38  %token ID
39  %token IF ELSE THEN REL_OP
40  %token POPEN PCLOSE
41  %token MUL_OP ADD_OP
42
43  %right MUL_OP
44  %left ADD_OP
45
46  %union{
47      int int_val;
48      float float_val;
49      char char_val;
50      char *str;
51      struct info *Node;
52  }
53
54  /*Declaring types for the tokens*/
55  %type<str> ID REL_OP ADD_OP MUL_OP
56  %type<int_val> INT_CONST
57  %type<float_val> REAL_CONST
58  %type<char_val> CHAR_CONST
59  %type<Node> program structure decl_stmts stmts
60  %type<Node> decl_stmt type value stmt
61  %type<Node> assn_stmt cond_stmt 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 : decl_stmts BGN stmts END{
72                 sprintf($$->code, "%s%10s\n%s", $1->code,
    "|", $3->code);
73             }
74 ;
75
76 decl_stmts : decl_stmt decl_stmts{
77                 $$ = makeNode();
78                 sprintf($$->code, "%s%s", $1->code, $2->code)
    ;
79              }
80
81            | decl_stmt{
82                 $$ = $1;
83              }
84 ;
85
86 decl_stmt : ID ':' type ';' {
87                 $$ = makeNode();
88                 sprintf($$->code, "%10s %-5s := %s\n", "|",
    $1, $3->var);
89             }
90
91           | ID ':' type '=' value ';'{
92                 $$ = makeNode();
93                 sprintf($$->code, "%10s %-5s := %s\n", "|",
    $1, $5->var);
94             }
95 ;
96
97 type : INT{
98             $$ = makeNode();
99             $$->int_val = 0;
100            sprintf($$->var, "%d", 0);
101            sprintf($$->code, "");
102       }
103
104     | REAL{
105            $$ = makeNode();
106            $$->float_val = 0.0;
107            sprintf($$->var, "%.2f", 0.0);
108            sprintf($$->code, "");
109       }
110
```

```
111       | CHAR{
112                 $$ = makeNode();
113                 $$->char_val = 0;
114                 sprintf($$->var, "%s", "NULL");
115                 sprintf($$->code, "");
116           }
117                     ;
118
119 value : INT_CONST{
120                 $$ = makeNode();
121                 $$->int_val = $1;
122                 sprintf($$->var, "%d", $1);
123                 sprintf($$->code, "");
124           }
125         | REAL_CONST{
126                 $$ = makeNode();
127                 $$->float_val = $1;
128                 sprintf($$->var, "%.2f", $1);
129                 sprintf($$->code, "");
130           }
131         | CHAR_CONST{
132                 $$ = makeNode();
133                 $$->int_val = $1;
134                 sprintf($$->var, "%c", $1);
135                 sprintf($$->code, "");
136           }
137 ;
138
139 stmts : stmt stmts{
140                 $$ = makeNode();
141                 sprintf($$->code, "%s%s", $1->code, $2->code);
142           }
143         | stmt{
144                 $$ = $1;
145           }
146 ;
147
148 stmt : assn_stmt {
149           $$ = $1;
150         }
151       | cond_stmt{
152                 $$ = $1;
153         }
154 ;
155
```

```
156 assn_stmt : ID '=' expr ';'{
157                 $$ = makeNode();
158                 char tac[100];
159                 sprintf($$->var, "%s", $1);
160                 sprintf(tac, "%10s %-5s := %s\n", "|", $$->
    var, $3->var);
161                 sprintf($$->code, "%s%s", $3->code, tac);
162             }
163 ;
164
165 expr : E{
166         $$ = $1;
167        }
168 ;
169
170 E : T MUL_OP E{
171         $$ = makeNode();
172         char tac[100];
173         sprintf($$->var, "x%d", ++tmp);
174         sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var,
    $1->var, $2, $3->var);
175         sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);
176     }
177   | T{
178         $$ = $1;
179     }
180   | F{
181         $$ = $1;
182     }
183 ;
184
185 T : T ADD_OP F{
186         $$ = makeNode();
187         char tac[100];
188         sprintf($$->var, "x%d", ++tmp);
189         sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var,
    $1->var, $2, $3->var);
190         sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);
191     }
192   | F{
193         $$ = $1;
194     }
195 ;
196
197 F : ID{
```

```
198         $$ = makeNode();
199         sprintf($$->var, "%s", $1);
200         sprintf($$->code, "");
201     }
202     | INT_CONST{
203             $$ = makeNode();
204             $$->int_val = $1;
205             sprintf($$->var, "%d", $1);
206             sprintf($$->code, "");
207       }
208     | REAL_CONST{
209             $$ = makeNode();
210             $$->float_val = $1;
211             sprintf($$->var, "%.2f", $1);
212             sprintf($$->code, "");
213       }
214     | CHAR_CONST{
215             $$ = makeNode();
216             $$->char_val = $1;
217             sprintf($$->var, "'%c'", $1);
218             sprintf($$->code, "");
219       }
220 ;
221
222 cond_stmt : IF POPEN condition PCLOSE THEN stmts ELSE stmts
      END IF{
223                 $$ = makeNode();
224                 int condBlock = ++jump;
225                 int endBlock = ++jump;
226                 sprintf($$->code, "%s%10s if %s then goto L%d
      \n%s%10s goto L%d\n%10s\nL%-5d - |\n%s%10s\nL%-5d - |\n",
      $3->code, "|", $3->var, condBlock, $8->code, "|", endBlock
      , "|", condBlock, $6->code, "|", endBlock);
227             }
228 ;
229
230 condition : expr REL_OP expr{
231             $$ = makeNode();
232             char tac[100];
233             sprintf($$->var, "%s%s%s", $1->var, $2, $3->var);
234             sprintf($$->code, "%s%s", $1->code, $3->code);
235        }
236 ;
237 %%
238
```

```
int yyerror(char* str){
    printf("\n%s", str);
    return 0;
}

int yywrap(){
    return 1;
}

int main(){
    printf("\nGiven code\n");
    system("cat file.txt");
    printf("\n
    ------------------------------------------------------------------------------
    n");
    printf("\nThree Address Code\n");

    yyparse();
    return 0;
}
```

Input:

```
Given code
i:integer=1;
a:integer=4;
b:integer=3;
c:integer=6;
d:integer=2;
x:integer;
begin
if (i>0) then
   x=a+b*c/d;
else
   x=a*b*c-d;
end if
end
```

Output:

```

Three Address Code

L0     - |
         | i     := 1
         | a     := 4
         | b     := 3
         | c     := 6
         | d     := 2
         | 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     - |
```

Learning Outcomes:

- Understood the basic idea of Three Address Code.
- Learnt how to identify control structures and write TAC for them.
- Learnt to use yacc efficiently for string concatenation, and hence generate code.

Department of Computer Science and Engineering

S.G.Shivanirudh , 185001146, Semester VI

23 April 2021

UCS1602 - Compiler Design

Exercise 8: Code optimisation using C

Objective:

Develop a C program to optimise the code generated as intermediate code.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void optimize(char *s) {

    // addition
    if(s[3]=='+'){
        if(s[2]=='0'||s[4]=='0'){
            if(s[0]==s[2]||s[0]==s[4]){
```

```
11                  printf("\n");
12              }
13              else{
14                  printf("%c%c%c\n",s[0],s[1],s[2]=='0'?s[4]:s
    [2]);
15              }
16          }
17          else{
18              printf("%s",s);
19          }
20      }
21      else if(s[3]=='*'){
22          if(s[2]=='1'||s[4]=='1'){
23              if(s[0]==s[2]||s[0]==s[4]){
24                  printf("\n");
25              }else{
26                  printf("%c%c%c\n",s[0],s[1],s[2]=='1'?s[4]:s
    [2]);
27              }
28          }
29          if(s[2]==s[4]){
30              printf("%c%c%c+%c\n",s[0],s[1],s[2],s[4]);
31          }
32      }
33      else if(s[3]=='-'){
34          if(s[2]=='0'||s[4]=='0'){
35              if(s[0]==s[2]||s[0]==s[4]){
36                  printf("\n");
37              }else{
38                  printf("%c%c%c%c\n",s[0],s[1],s[2]=='0'?'-':'
    ',s[2]=='0'?s[4]:s[2]);
39              }
40          }
41          if(s[2]==s[4]){
42              printf("%c%c%c+%c\n",s[0],s[1],s[2],s[4]);
43          }
44      }
45      else if(s[3]=='/'){
46          if(s[4]=='1'){
47              if(s[0]==s[2]){
48                  printf("\n");
49              }else{
50                  printf("%c%c%c\n",s[0],s[1],s[2]);
51              }
52          }
```

```
53          if(s[2]=='0'){
54              printf("%c%c%c\n",s[0],s[1],'0');
55          }
56      }
57      else if(s[2]=='p'){
58          if(s[8]=='2'){
59              printf("%c%c%c*%c\n",s[0],s[1],s[6],s[6]);
60          }else{
61              printf("%s",s);
62          }
63      }
64  }
65
66  int main(int argc, char *argv[]){
67      FILE *fp;
68      fp = fopen(argv[1], "r");
69      int i = 0;
70      int tot = 0;
71      char lines[100][100];
72      while(fgets(lines[i], 100, fp)) {
73          lines[i][strlen(lines[i])] = '\0';
74          i++;
75      }
76      tot = i;
77
78      for(i = 0; i < tot; ++i) {
79          optimize(lines[i]);
80      }
81  }
```

Input file:

```
x=x+0
y=y*1
x=0+x
x=y+1
y=1*y
x=z+0
x=w*w
x=pow(i,2)
x=pow(i,3)
x=0-y
x=y-0
x=x/1
x=y/1
x=0/x
```

Output:

```
➜ ./a ip.txt



x=y+1

x=z
x=w+w
x=i*i
x=pow(i,3)
x=-y
x= y

x=y
x=0
```

Learning Outcomes:

- Understood the basic idea of Code optimisation.
- Learnt what sort of expressions needed to be simplified.
