# گزارش پروژه درس کامپایلر

رضا نعمت اللهی – 9932106

## شرح پروژه:

این پروژه شامل سه فاز intermediate code generation, syntax analysis, lexical analysis می باشد.

برای فاز lexical analysis از ابزار flex و برای قسمت syntax analysis از ابزار bison استفاده شده است.

فاز intermediate code generation با استفاده از زبان برنامه نویسی C انجام شده که به همراه قسمت syntax analysis در فایل bison پیاده سازی شده است.

هدف از این پروژه پیاده سازی کامایلری اس که بتواند چهار عمل جمع، تفریق، ضرب و تقسیم که در ادامه شیوه محاسبه آن ها آمده است را محاسبه کند و همچنین intermediate code به زبان C تولید کند.

عبارت شامل اعداد صحیح (حداکثر 10 رقمی)، عملگر های جمع، تفریق، ضرب و تقسیم و پرانتز و فاصله خالی است و هر عملگر بصورت زیر عمل می کند:

- b+a: ارقامی از عدد b که در عدد a نیستند به انتهای a الحاق می شوند.
- b-a: ارقامی از عدد b که در عدد a هستند از a حذف می شوند.
- b*a: رقم حاصل از جمع ارقام (یا جمع جمع ارقام) عدد b در صورت عدم وجود در a به انتهای a الحاق می شود .
- b/a: رقم حاصل از جمع ارقام (یا جمع جمع ارقام) عدد b در صورت وجود در a از a حذف می شود .
- اولویت عملگرها و شرکت پذیری آنها مطابق معمول است .
- فرض می شود که عبارت ورودی فاقد خطای کامپایلری است.

## 1. Lexical analysis:

در این فاز یک lexical analyzer به کمک ابزار flex نوشته شده که وظیفه ان پیدا کردن token های عبارت ورودی است. سپس آن token ها را به parser میدهد که در فاز دوم استفاده میشوند.

برای این کار ابتدا در قسمت rules در فایل lex، regular expression مربوط به هر token نوشته میشود. ابتدا و انتهای این قسمت با علامت %% مشخص میشود.

به عنوان مثال بر اساس regular expression نوشته شده، اعداد ورودی میتوانند بین 1 تا 10 رقم باشند.

در این عکس rule های نوشته شده آورده شده اند:

```
8   /* rules */
9   %%
10  [0-9]{1,10} {strcpy(yylval.num[0], yytext); yylval.num[1][0] = 0; return digit;}
11  [*] {return MUL;}
12  [/] {return DIV;}
13  [+] {return ADD;}
14  [-] {return SUB;}
15  [=] {return EQ;}
16  [(] {return PAR_1;}
17  [)] {return PAR_2;}
18  [\n] {return 0;}
19  [ ] {}
20  . {printf("%s is not a valid character.\n", yytext);}
21  %%
```

در ابتدای فایل lex هدر فایل هایی که از آن ها استفاده شده است include میشوند.

فایل bison_file.tab.h پس از کامپایل فایل bison در فاز syntax analysis ساخته میشود.

```
1   /* C declarations */
2   %{
3       #include <stdio.h>
4       #include <string.h>
5       #include "bison_file.tab.h"
6   %}
```

## 2. Syntax analysis:

در این فاز گرامر زبان مشخص میشود. همچنین برای هر production در گرامر semantic action مربوط به آن نوشته میشود.

گرامر استفاده شده در این پروژه به این صورت است:

S → Expr =

Expr → Expr + Term | Expr - Term | Term

Term →Term * Factor | Term / Factor | Factor

Factor → Digit | (Expr)

در ادامه بخش های مختلف فایل bison توضیح داده میشود.

- در ابتدا declaration توابع مشخص میشوند. همچنین هدر فایل های C که اسفاده شده اند include میشوند:

```
// C declarations
%{
    void yyerror (char *s);
    int yylex();
    #include <stdio.h>
    #include <stdlib.h>
    #include <ctype.h>
    #include <string.h>

    const int max_arr_size = 30;

    void add(char * a, char * b, char * c);
    void sub(char * a, char * b, char * c);
    void multiply(char * a, char * b, char * c);
    void division(char * a, char * b, char * c);
    void print_OP(char a[][max_arr_size], char b[][max_arr_size], char c[][max_arr_size], char operation);

    int node_counter = 1;
%}
```

- در ادامه یک Union در bison تعریف میشود که با استفاده از ان ابزار bison موجه میشود که attribute هر token یا Non-Terminal از چه type است. همچنین type متغیر yylval در فایل lex نیز توسط این Union مشخص میشود:

```
%union{
    char num[2][30 /*max size of array*/]; // first row is for value. second row is for node number (Tx)
}
```

- Data type مشخص شده در Union بالا یک آرایه دو بعدی است که سطر اول آن مربوط به مقدار عددی و سطر دوم آن مربوط به شمارنده node مربوطه در درخت فرضی ساخته شده عبارت ورودی میباشد.

- در ادامه token ها و Non-Terminal های استفاده شده در گرامر تعریف میشوند:

```
26  // tokens
27  %token<num> digit
28  %token ADD
29  %token SUB
30  %token MUL
31  %token DIV
32  %token EQ
33  %token PAR_1
34  %token PAR_2
35
36  // start symbol
37  %start s
38
39  // none-terminals type
40  %type <num> expr
41  %type <num> term
42  %type <num> factor
```

- در قسمت بعد production ها و semantic action های هرکدام تعریف شده اند. ابتدا و انتهای این قسمت با علامت %% مشخص میشود:

```
44  // productions and semantic actions
45  %%
46  s: expr EQ
47       ;
48
49  expr: expr ADD term {add($1[0], $3[0], $$[0]); print_OP($1, $3, $$, '+');}| // add operation
50        expr SUB term {sub($1[0], $3[0], $$[0]); print_OP($1, $3, $$, '-');}| // subtract operation
51        term          {strcpy($$[0], $1[0]); strcpy($$[1], $1[1]);} // if you dont specify a semantic action, the default is $$ = $1
52        ;
53
54  term: term MUL factor {multiply($1[0], $3[0], $$[0]); print_OP($1, $3, $$, '*');}| //multiply operation
55        term DIV factor {division($1[0], $3[0], $$[0]); print_OP($1, $3, $$, '/');}| //division operation
56        factor          {strcpy($$[0], $1[0]); strcpy($$[1], $1[1]);}
57        ;
58
59  factor: digit           {strcpy($$[0], $1[0]); strcpy($$[1], $1[1]);}| // digit
60          PAR_1 expr PAR_2 {strcpy($$[0], $2[0]); strcpy($$[1], $2[1]);}  // parenthesis
61          ;
62  %%
```

- در ادامه و در همان فایل bison توابع مربوط به semantic action ها به زبان C پیاده سازی شده اند.

## 3. Intermediate code generation:

- این فاز با اسفاده از زبان C پیاده سازی شده است. برای این قسمت تابع print_OP() پیاده سازی شده که وظیفه ان چاپ عبارت three address code به زبان C است. شیوه پیاده سازی آن به این شکل است:

```
161  // ==================== print three-address code ====================
162  void print_OP(char a[][max_arr_size], char b[][max_arr_size], char c[][max_arr_size], char operation){
163      if (a[1][0] == 0){
164          if (b[1][0] == 0){
165              printf("T%d=%s%c%s;\n", node_counter, a[0], operation, b[0]);
166          }
167          else{
168              printf("T%d=%s%cT%s;\n", node_counter, a[0], operation, b[1]);
169          }
170      }
171      else{
172          if (b[1][0] == 0){
173              printf("T%d=T%s%c%s;\n", node_counter, a[1], operation, b[0]);
174          }
175          else{
176              printf("T%d=T%s%cT%s;\n", node_counter, a[1], operation, b[1]);
177          }
178      }
179      printf("T%d=%s;\n", node_counter, c[0]);
180      sprintf(c[1], "%d", node_counter);
181      node_counter++;
182  }
```

تابع yyerror() برای چاپ کردن خطا به هنگام compile کردن عبارت ورودی تعریف میشود.

```
184  // print the compiler error
185  void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
```

و در اخر تابع main() برای اجرای برنامه نوشته میشود که در ان تابع yyparse() صدا زده میشود.

```
188  int main(){
189
190      printf("Compiler is running. Enter your input: \n");
191
192      yyparse();
193
194      return 0;
195  }
```

برای compile و اجرای برنامه می‌بایست دستورات زیر را در ماشین خود وارد کنید:

```
flex lex_file.l
bison -d -t bison_file.y
gcc lex.yy.c bison_file.tab.c
.\a
```