17/10/22

YACC

Introduction to YACC

A parser generator is a program that takes as input a specification of a syntax and produces as output a procedure for recognizing that language. Historically they are called compiler-compilers.

YACC is an LALR(1) (LookAhead, Left to Right, Rightmost derivation producer with 1 lookahead token) parser generator.

Input file:

YACC input file is divided into 3 parts:

```
/*definitions */
...
%%
/*rules */
...
%%
/* auxiliary routines */
...
```

Input file : Definition Part :

- The definition part includes information about the tokens used in the syntax definition:

```
%token NUMBER
%token ID
```

- The definition part can include C code external to the definition of the parser and variable declarations within %{ and %} in the first column

- It could also include the specification of the starting symbol in the grammar :

```
%start nonterminal
```

Input file : Rule Part

- The rules part contains grammar definition in a modified BNF form.

- Actions is C code in { } and can be embeded inside (Translation schemes).

Input file : Auxiliary Routines part

- The auxiliar routines part is only C code
- It includes function definitions for every functions needed in rules part.

For compiling YACC program:

1. Write Lex program in a file file.l and yacc in a file file.y
2. Open terminal and Navigate to the directory where you have saved the files.
3. Type lex file.l
4. Type yacc file.y
5. Type cc lex.yy.c y.tab.h -ll
6. Type ./a.out

- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() must call the function yyparse().

Input file:

- If yylex() is not defined in the auxiliary routines section then it should be included:
  #include "lex.yy.c"
- YACC input file generally finishes with: .y

Output file:

- The o/p of YACC is a file named y.tab.c
- If it contains main() definition it must be compiled to to be executed.
- Otherwise the code can be an external function definition for the function int yyparse()
- If called with the -d option in the command line Yacc produces as output a header file y.tab.h with all its specific definition
- If called with -v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Algorithm

Step 1: Start the program

Step 2: Reading an expression

Step 3: Checking the validating of the given expression according to the rule using yacc.

Step 4: Using expression rule print the result of the given values

Step 5: Stop the program.

Exp: 3

17/10/22

20

YACC

Aim: Write program to design parser for arithmetic expressions using YACC

(i) Program to recognize a valid arithmetic expression that uses operators +, -, * and /

Program:

```
// lexical Analyzer Source code
%{
    #include "y.tab.h"
    extern yylval;
%}
%%
[0-9]+  {
            yylval = atoi(yytext);
            return NUMBER;
        }
[a-zA-Z]+ {return ID;}
[\t]+     ; /* for skipping white spaces */

\n        {return 0;}
.         {return yytext[0];}
%%
```

Output

```
lex evaluati_exp.l
yacc -d evaluati_exp.y
cc lex.yy.c y.tab.c -ll
./a.out

Enter the expression
7*(5-3)/2
Result = 7
```



```
//Parser source code
%{
    #include<stdio.h>
%}
%token NUMBER ID
  //setting the precedence and associativity of operators.
%left '+' '-'
%left '*' '/'

%%
E:T    {
        printf("Result = %d \n", $$);
        return 0;
       }
T:
    T '+' T { $$ = $1 + $3; }
  | T '-' T { $$ = $1 - $3; }
  | T '*' T { $$ = $1 * $3; }
  | T '/' T { $$ = $1 / $3; }
  | '-' NUMBER { $$ = $2; }
  | '-' ID { $$ = $2; }
  | '(' T ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  | ID { $$ = $1; };
%%
int main()
{ printf(" Enter the expression \n");
  yyparse();
}
int yyerror(char *s) {
    printf(" Expression is invalid \n");
}
```

Algorithm

Step 1: Start the program

Step 2: Reading an expression

Step 3: checking the validating of the given expression according to the rule using yacc.

Step 4: Using expression rule print the result of the given values.

Step 5: Stop the program

(ii) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

Program:

```
// Program to recognize a valid variable

lex Part
%{
   #include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return letter;
[0-9]    return digit
.        return yytext[0];
\n       return 0
%%
  int yywrap()
  {
     return 1;
  }
YACC Part
%{
      #include<stdio.h>
      int valid=1;
%}
% token digit letter
```

Output

```
yacc -d valid.y
lex validy
lex validy
gcc lex.yy.c y.tab.c -w
./a.out
Enter a name to tested for identifier  abc

Its an identifier

./a.out
Enter a name to tested for identifier  848_f

Its not an identifier
```



```
%%
start : letter s
s: letter s
   | digit s
   |
   ;
%%
int yyerror()
{
   printf("\n Its not a identifier\n");
   valid=0;
   return 0;
}
int main()
{
   printf("\n Enter a name to tested for identifier");
   yyparse();
   if(valid)
   {
      printf("\n Its an identifier\n");
   }
}
```

Algorithm

Input: Read an arithmetic expression

Output: Print whether the expression is valid or not

Step 1: Start

Step 2: ~~Print whether the expression is valid or not.~~

Step 3: Define the definition section of lex and yacc if any.

Step 3: Define the rule section of lex with numbers.

```
%%
[0-9]+ {
        yylval = atoi(yytext);
        return NUMBER;
}
%%
```

Step 4: Define the rule section of yacc with arithmetic operators and print the result by performing the operation.

```
%%
A:E
  {  printf("\n Result = %d \n", $$);}
;
E:E'+'E {$$ = $1 + $3;}
 |E'-'E {$$ = $1 - $3;}
 |E'*'E {$$ = $1 * $3;}
 |E'/'E {$$ = $1 / $3;}
 |E'%'E {$$ = $1 % $3;}
 |'('E')' {$$ = $2;}
 |NUMBER {$$ = $1;}
 ;
%%
```

Step 5: Define the main function in yacc and call yyparse()

Step 6: Call yyerror() in yacc if the expression is invalid

Step 7: Invoke yywrap() in lex to check the end.

Step 8: Stop

(iii) Write a program to implement arithmetic calculator.

Program:

// lex analyzer source code

```
%{
    #include <stdio.h>
    #include "y.tab.h"
    extern int yylval;
%}
%%
[0-9]+ {
        yylval = atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
  int yywrap()
  {
      return 1;
  }
```

// Parser source code

```
%{
    #include <stdio.h>
    int flag = 0;
%}
```

Output

```
lex calc.l
yacc -d calc.y
gcc lex.yy.c y.tab.c -w
./a.out
```

Enter any arithmetic expression which can have operations Addition, Subtraction, multiplication, division, modulus and Round brackets

4+5

Result = 9

Entered arithmetic expression is valid


```
% token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%
Arithmetic Expression : E {
        Printf("\n Result = %d \n", $$);
        return 0;
    };

E:E'+'E { $$= $1 + $3; }
|E'-'E  { $$= $1 - $3; }
|E'*'E  { $$= $1 * $3; }
|E'/'E  { $$= $1 / $3; }
|E'%'E  { $$= $1 % $3; }
|'('E')' { $$= $2; }
|NUMBER { $$= $1 ; }
;
%%
//driver code
void main()
{
  printf("\n Enter any Arithmetic Expression which can
        have operations Addition, Subtraction, Multiplication,
        Division, modulus and Round brackets :\n");
  yyparse();
```

```
if (flag==0)
    printf("\n Entered Arithmetic expression is valid\n");
}

void yyerror()
{
    printf("\nEntered Arithmetic expression is invalid\n");
    flag=1;
}
```

Result:

Program has been successfully executed and hence obtained the output successfully.