9/19 YACC编程讲解

YACC程序结构

框架：

```
%{
//include
%}
//定义(definations)
%%
//规则(rules)
%%
//代码(user code)
int main(int argc, char **argv)
{
  yylex()
  return 0;
}
int yywrap()
{
    return 1;
}
```

1. **定义部分**（Definitions Section）：
   - 在这一部分，用户可以定义宏（与C预处理器宏类似）和导入所需的头文件。
   - 这里也可以定义联合体（union）来指定yacc语义值的类型。后续实验可能会用到
   - `%token`指令用于声明词法符号。
   - `%start`可以用来声明开始符号。
   - 例如:

     ```
     %{
       #include <stdio.h>
     %}
     %token NUMBER
     %token PLUS MINUS TIMES DIVIDE
     ```

2. **规则部分**（Rules Section）：
   - 在这里，用户定义文法规则，说明如何从一个或多个已知的符号组合生成新的符号。
   - 一个文法规则的左边是一个非终结符，右边是由终结符和/或非终结符组成的序列。右边和左边之间由冒号分隔，规则以分号结束。
   - 文法右侧可以设定**语法制导翻译的规则**，当识别到该规则时自动执行
   - 例如:

```
%%
expression:
    NUMBER
  | expression PLUS expression
  | expression MINUS expression
  ;
```

3. **用户子程序部分**（User Subroutines Section）：
    - 通常这里会包括`yacc`调用的词法分析器（通常由`lex`生成，本次实验我们自行定义）。
    - 还可以包含其他需要的C函数和主函数`main()`.
    - 例如:

```
%%
int yylex() {
  return getchar();
}
int main() {
  yyparse();
  return 0;
}

int yyerror(char *s) {
  fprintf(stderr, "Error: %s\n", s);
  return 0;
}
```

当你运行`yacc`工具时，它会生成一个C源代码文件（通常命名为`y.tab.c`），该文件包含一个语法分析器。这个生成的文件还需要一个词法分析器，通常由`lex`或`flex`工具生成，然后一起编译。**但**在本次实验中，我们采用自行定义的方式提供`yylex`函数

> `yylex`函数是一个词法分析器（也称为扫描器或lexer）的主要组成部分，它在语法分析过程中被`yacc`生成的语法分析器（或称为parser）调用。其主要任务是读取输入流，识别并返回词法单元或token。每当语法分析器需要读取下一个token时，它就调用`yylex`函数。
>
> 具体来说，当你使用工具如`lex`或`flex`编写词法分析规则并生成lexer时，这些工具会为你生成一个名为`yylex`的函数。这个函数会根据你提供的规则对输入进行扫描，并返回相应的token。后续实验我们才采用这种方式生成词法分析器

实验部分 -- 基础代码

一个计算表达式值的基础yacc代码如下：

```
%{
/*********************************************
YACC file
基础程序
Date:2023/9/19
forked SherryXiye
```

```
*********************************************/
#include<stdio.h>
#include<stdlib.h>
#ifndef YYSTYPE
#define YYSTYPE double
#endif
int yylex();
extern int yyparse();
FILE* yyin;
void yyerror(const char* s);
%}
//注意先后定义的优先级区别
%left '+' '-'
%left '*' '/'
%right UMINUS

%%


lines   :       lines expr '\n' { printf("%f\n", $2); }
        |       lines '\n'
        |
        ;

expr    :       expr '+' expr   { $$=$1+$3; }
        |       expr '-' expr   { $$=$1-$3; }
        |       expr '*' expr   { $$=$1*$3; }
        |       expr '/' expr   { $$=$1/$3; }
        |       '('expr')'      { $$=$2;}
        |       '-' expr %prec UMINUS   {$$=-$2;}
        |       NUMBER
        ;

NUMBER  :       '0'         {$$=0.0;}
        |       '1'         {$$=1.0;}
        |       '2'         {$$=2.0;}
        |       '3'         {$$=3.0;}
        |       '4'         {$$=4.0;}
        |       '5'         {$$=5.0;}
        |       '6'         {$$=6.0;}
        |       '7'         {$$=7.0;}
        |       '8'         {$$=8.0;}
        |       '9'         {$$=9.0;}
        ;

%%

// programs section

int yylex()
{
    return getchar();
```

```
}

int main(void)
{
    yyin=stdin;
    do{
        yyparse();
    }while(!feof(yyin));
    return 0;
}
void yyerror(const char* s){
    fprintf(stderr,"Parse error: %s\n",s);
    exit(1);
}
```

基础程序讲解

1. **头部**:

   ```
   %{
   //...
   #ifndef YYSTYPE
   #define YYSTYPE double
   #endif
   int yylex();
   extern int yyparse();
   FILE* yyin;
   void yyerror(const char* s);
   %}
   ```

   这部分代码定义了预处理部分，主要为生成的C程序提供一些头文件和定义。这里定义了`YYSTYPE`为`double`，意味着`yacc`产生的值应该是双精度浮点数。

2. **优先级定义**:

   ```
   %left '+' '-'
   %left '*' '/'
   %right UMINUS
   ```

   这部分定义了算术运算符的优先级，越靠下优先级越高。`UMINUS`用于识别负数。

3. **语法规则**: 以下部分定义了如何解析算术表达式:
   - `lines`用于处理多行输入，每行都是一个表达式。
   - `expr`定义了表达式的结构，包括加、减、乘、除等操作。
   - `NUMBER`定义了如何解析数字字符，并将其转换为`double`类型的值。

4. **词法分析器**: 目前`yylex`函数非常简单，它只是从输入中读取并返回下一个字符。这就是为什么在`NUMBER`规则中直接使用字符（如`'0'`、`'1'`等）来识别数字的原因。

5. **主函数**: `main`函数将`yyin`设置为标准输入`stdin`，然后不断调用`yyparse`直到达到文件尾。这意味着此程序从标准输入读取数据，并为每一行计算结果。

6. **错误处理**: `yyerror` 函数用于处理语法错误，当 `yacc` 生成的 `yyparse` 函数遇到错误时，它会调用这个函数。这里的实现简单地打印错误消息并退出。

这个程序将从标准输入读取算术表达式，每当它读到一个换行符时，它就计算表达式的值并打印结果。但目前这个程序只能处理单个数字字符（0-9）而不是多位数字。

程序编译流程

一个yacc程序变为可执行语法分析程序的步骤如下：

1. 使用yacc编译**.y**文件生成语法分析器的**.c**文件：

```
yacc expr.y #默认生成文件为y.tab.c
```

2. 使用gcc等编译器编译生成可执行文件

```
gcc y.tab.c -o compute_expr
```

实验部分 -- 进阶代码

更复杂表达式计算 参考框架

```
%{
/*********************************************
将所有的词法分析功能均放在 yylex 函数内实现，为 +、-、*、\、(、 ) 每个运算符及整数分别定义一个单词
类别，在 yylex 内实现代码，能
识别这些单词，并将单词类别返回给词法分析程序。
实现功能更强的词法分析程序，可识别并忽略空格、制表符、回车等
空白符，能识别多位十进制整数。
YACC file
*********************************************/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#ifndef YYSTYPE
#define YYSTYPE double
#endif
int yylex();
extern int yyparse();
FILE* yyin;
void yyerror(const char* s);
%}

//TODO:给每个符号定义一个单词类别
%token ADD MINUS
%token NUMBER
%left ADD MINUS
%right UMINUS

%%
```

```
lines   :       lines expr ';' { printf("%f\n", $2); }
        |       lines ';'
        |
        ;
//TODO:完善表达式的规则
expr    :       expr ADD expr   { $$=$1+$3; }
        |       expr MINUS expr   { $$=$1-$3; }
        |       MINUS expr %prec UMINUS   {$$=-$2;}
        |       NUMBER  {$$=$1;}
        ;

%%

// programs section

int yylex()
{
    int t;
    while(1){
        t=getchar();
        if(t==' '||t=='\t'||t=='\n'){
            //do noting
        }else if(isdigit(t)){
            //TODO:解析多位数字返回数字类型
        }else if(t=='+'){
            return ADD;
        }else if(t=='-'){
            return MINUS;
        }//TODO:识别其他符号
        else{
            return t;
        }
    }
}

int main(void)
{
    yyin=stdin;
    do{
        yyparse();
    }while(!feof(yyin));
    return 0;
}
void yyerror(const char* s){
    fprintf(stderr,"Parse error: %s\n",s);
    exit(1);
}
```

中缀转后缀表达式的一些提示

Easy Task。关键在于：

1. `YYTYPE` 应该改成什么？
2. 语法制导翻译的规则如何编写？

符号表的实现 -- 计算表达式值程序的改进

为了实现表达式的赋值功能，我们需要实现符号表的功能，直接原因是我们在给变量赋值后需要存储符号变量的值，后续再次出现该变量时才可以得知该变量的值。

可以由此总结，为了实现简单的符号表，我们的程序实现至少需要包含以下部分更新：

1. **符号表的结构，**并且需要包含插入值和查询值的功能
2. **语法规则，**需要单独设定标识符Token；定义赋值语句的语法；识别到标识符时，可能进行查询值或插入值的操作
3. **词法分析器，** 需要能够识别标识符
4. **错误处理，** 设定错误处理的好处在于方便你debug

由于我们并未尝试编译一整个c++程序，目前我们的符号表还不用考虑**作用域**的问题，是一个单层次的符号表，大家可以进一步思考：如果是需要对C++程序进行语法分析，我们的符号表还需要考虑哪些问题？