语义分析程序的设计与实现

班号：2009211315 姓名：李根 学号：09211541

# 一：实验内容：

编写语法分析程序，实现对算术表达式的语法分析，要求所分析的算术表达式由如下的文法产生。

1、E->E+T|E-T|T

2、T->T\*F|T/F|F

3、F->id|(E)|num

# 二：实验要求：

在对表达式进行分析的同时，输出所采用的产生式。

可以采用多种方法

◆编写递归调用程序，实现自顶向下的分析。

◆编写LL(1)语法分析程序，要求：

◇编程实现算法4.2，为给定的文法自动构造预测分析表

◇编程实现算法4.1，构造LL(1)预测分析程序，

◆编写语法分析程序，实现自底向上的分析，要求：

◇构造识别所有活前缀的DFA

◇构造LR分析表

◇编程实现算法4.3，构造LR分析程序

◆利用yacc自动生成语法分析程序，调用LEX自动生成的词法分析器程序

# 三：实验方法：

## ◆由LEX建立YACC的词法分析程序

由LEX产生的词法分析程序可用于YACC，LEX编译程序根据LEX源程序产生词法分析程序yylex(),这个名字就是YACC所需要的词法分析程序的名字。如果YACC要调用LEX产生的词法分析程序，则在YACC源程序的第三部分用语句#include“lex.yy.c”代替函数yylex()的定义，这一yylex()就可以访问YACC中记号的名字，因为LEX的输出时候YACC输出文件的一部分，所有，每个LEX的动作都返回YACC知道的终结符。

在UNIX的环境下，如果LEX源程序在first.l中，YACC的源程序在second.y中，可以使用以下命令得到所需要的分析程序。

Lex first.l

Yacc second.y

cc-o yaccdemo y.tab.c lex.yy.c

## ◆词法分析

用户必须提供一个词法分析器来读取输入流并把记号(带有值，如果需要的话)传达到解析器。词法分析器使叫做 yylex 的整数值的函数。这个函数返回一个整数的记号编号，它表示读取的记号的种类。如果这个记号关联着一个值，应当把它赋予外部变量 yylval。

为使通信得以发生，解析器和词法分析器必须在记号编号上达成一致。编号可以由 Yacc 或用户来选择。在这两种情况下，使用 C 语言的“# define”机制允许词法分析器使用符号来返回这些编号。例如，假定在 Yacc 规定文件的声明段中已经定义记号名字 DIGIT。

它的意图是返回一个 DIGIT 记号编号，和等于这个数字的数值的一个值。倘若词法分析器代码位于规定文件的程序段，标识符 DIGIT 将被定义为与记号 DIGIT 关联的记号编号。

这种机制导致清晰的、易于修改的词法分析器；唯一的缺点是在文法中需要避免使用任何在 C 语言或解析器中保留的或有意义的记号名字；例如，使用记号名字 if 或 while 就一定会导致编译词法分析器时出现严峻的困难。记号名字 error 保留给错误处理，不应该随便使用。

同上所述，记号编号可以由 Yacc 或用户来选择。在缺省的条件下，编号由 Yacc 选择。文字字符的缺省记号编号是它在本地字符集中的字符数值。其他名字赋予从 257 开始的记号编号。

要把一个记号编号赋予一个记号(包括文字)，可以在声明段中记号或文字的第一次出现时直接跟随着一个非负整数。这个整数被接受为这个名字或文字的记号编号。不通过这种机制定义的名字和文字保持它们的缺省定义。所有记号编号都是不同的是很重要的。

构造词法分析器的一个有用的工具是 Mike Lesk[8]开发的 Lex 程序。这些词法分析器设计用来与 Yacc 解析器紧密协调工作。这些词法分析器的规定使用正则表达式而不是文法规则。可以轻易的用 Lex 生成非常复杂的词法分析器，但是仍有一些语言(比如 FORTRAN)不适应任何理论框架，它的词法分析器必须手工制作。

## ◆解析器如何工作

Yacc 把规定文件转换成 C 程序，它依据给出的规定解析输入。做从规定到解析器转换的算法是复杂的，就不在这里讨论了(更多信息参见引用)。但是，解析器自身就相对简单了，理解它是如何工作的，尽管不是严格必须的，但会使错误修复和歧义处置更加易于理解。

Yacc 提供的解析器是由带有一个栈的有穷状态自动机组成。解析器自身还有能力读取和记住(叫做超前(lookahead)记号)下一个输入记号。当前状态总是在栈顶。有穷状态自动机的状态是一个给定的小整数标签(label)；最初时，机器是在状态 0 下，栈只包含状态 0，没有读取超前记号。

机器对它只能获得四个动作，叫做移进(shift)、归约(reduce)、接受和错误。

# 第四：YACC内部名称：

|  |  |
| --- | --- |
| YACC内部名称 | 说明 |
| y.tab.c  y.tab.h  yyparse  yylval  yyerror  error  yyerrok、yychar  YYSTYPE  yydebug | YACC输出文件名  YACC生成的头文件，包含有记号定义  YACC分析程序  栈中的当前记号的值  由YACC使用的用户定义的错误信息打印程序  YACC错误违记号  在错误处理之后，回到正常操作方式  变了，记录导致错误的先行记号  定义分析栈值类型的预处理器符号  变量，用户置1生成有关分析动作的运行信息 |

# 第五：运行结果（源代码见附录）

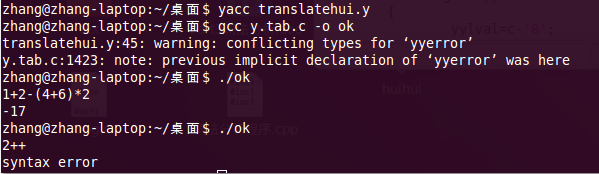
在linix下输入的命令行依次是：

**Cd 桌面**

**Yacc translate.y**

**Gcc y.tab.c –o rong**

**./rong**

****

注：Cd 桌面 到桌面目录下

Yacc translate.y 用yacc编译器编译translate.y程序到C文件

Gcc y.tab.c –o rong 用C编译器生成可执行文件

./rong 运行可执行文件

输入表达式，打印正确计算结果，否则，输出syntax error

# 第六：实验总结

1.用C语言实现词法分析的效率较高，但用LEX可以机械的执行程序，不用思考，虽然效率不是很高，但比较方便。Lex可以智能的读单词，并且按照最长匹配原则和优先匹配原则识别单词。Yacc则很简单的文法式进行分析，不需要定义递归的详细飞计算法则，直接修改生成式和标记符就可以应用于语法分析，简单方便，可移植性高。

2.yacc程序，是在linix环境下，用命令行执行的，.y文件不能直接在windows环境下打开，但是，下载了Parser Generator后，就可以直接读取.y文件，但此时不能运行yacc，还需要进行环境变量（例如PASH）的配置，才能在windows环境下使用yacc编译器。这个对于未安装虚拟机，初识linix的人使用yacc提供了很大的方便。

3.通过lex yacc的编程练习，明白了词法分析和语法分析的基本操作，弄清了原理，为下一步进行语义分析打下了良好的基础。

# 第七：附录

## 附录一：yacc程序，加注释

%{

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

typedef struct x{int i;int c;} type;

%}

%union{

int num;

char \*id;

type c;

}

%token <num> NUM

%token <id> ID

%type <c> expr,term,factor

%%

line: expr'\n'

{ if($1.i>=0)

printf("%d num\n",$1.i);

else

printf("id\n");

}

;

expr: expr '+' term

{ if($1.c==0&&($3.c==0))

{

$$.i=$1.i+$3.i;

$$.c=0;

printf("E->E+T E.type:num\n");

}

else

{

$$.c=1;

$$.i=-1;

printf("E->E+T E.type:id\n");

}

}

|expr '-' term {if($1.c==0&&($3.c==0))

{

$$.i=$1.i-$3.i;

$$.c=0;

printf("E->E-T E.type:num\n");

}

else

{

$$.c=1;

$$.i=-1;

printf("E->E-T E.type:id\n");

}

}

|term {

$$.i=$1.i;

$$.c=$1.c;

if($$.c==0 )

printf("E->T E.type:num\n");

else

printf("E->T E.type:id\n");

}

;

term: term '\*' factor {if($1.c==0&&($3.c==0))

{

$$.i=$1.i\*$3.i;

$$.c=0;

printf("T->T\*F T.type:num\n");

}

else

{

$$.c=1;

$$.i=-1;

printf("T->T\*F T.type:id\n");

}

}

|term '/' factor {if($1.c==0&&($3.c==0))

{

$$.i=$1.i/$3.i;

$$.c=0;

printf("T->T/F T.type:num\n");

}

else

{

$$.c=1;

$$.i=-1;

printf("T->T/F T.type:id\n");

}

}

|factor {

$$.c=$1.c;

$$.i=$1.i;

if($$.c==0)

printf("T->F T.type:num\n");

else

printf("T->F T.type:id\n");

}

;

factor: ID {$$.c=1;$$.i=-1;}

|'(' expr ')' {

$$.c=$2.c;

$$.i=$2.i;

if($$.c==0)

printf("F->(E) F.type:num\n");

else

printf("F->(E) F.type:id\n");}

|NUM {$$.i=$1;$$.c=0;}

;

%%

main()

{return yyparse();}

int yylex(void)

{

static int done =0;

int c;

if(done) return 0;

while((c=getchar())==' ');

if(isdigit(c))

{

ungetc(c,stdin);

yylval.num=c-'0';

scanf("%d",&yylval);

printf("\nnum\n");

printf("F->num F.type:num\n");

return(NUM);

}

else if(c<='z'&&c>='a'||c<='Z'&&c>='A')

{

yylval.id=ID;

printf("\nid\n");

printf("F->id F.type:id\n");

return(ID);

}

else if(c=='\n') done = 1;

return c;

}

void yyerror(char \*s)

{

printf("%s\n",s);

}

%{ /\*第一节，普通的C语言声明\*/

#include<stdio.h>

#include<ctype.h>

%}

%token ID/\*说明记号，在这里说明的记号可以用于随后的翻译规则部分和辅助过程中\*/

%token NUM

%% /\*文法记号声明，默认第一个产生式的左部非终结符号时候文法的开始符号\*/

/\*文法产生式和相关语义动作\*/

line: expr'\n' {printf("%d\n",$1);}

;

/\*表示输入是表达式后面跟一个换行符\*/

expr: expr '+' term {$$=$1+$3;}/\*带单引号的是终结符\*/

|expr '-' term {$$=$1-$3;}/\*$$表示和规则左部非终结属性，$i表示右部第i个文法的相关属性\*/

|term {$$=$1;}

;

term: term '\*' factor {$$=$1\*$3;}/\*非终结符的名字要对应\*/

|term '/' factor {$$=$1/$3;}

|factor {$$=$1;}

;

factor: ID {$$=$1;}

|'(' expr ')' {$$=$2;}

|NUM {$$=$1;}

;

main()

{return yyparse();}/\*分析成功返回0，失败返回1\*/

int yylex(void)

{

static int done =0;

int c;

if(done) return 0;

while((c=getchar())=='$');/\*读字符作为返回值\*/

if(isdigit(c))/\*判断是否是数字\*/

{

yylval=c-'0';

return(NUM);

}

else if(c<='z'&&c>='a'||c<='Z'&&c>='A')

{

yylval=c;

return(ID);

}

else if(c=='\n')done = 1;

return c;

}

void yyerror(char \*s)/\*在分析错误时，调用，打印错误信息\*/

{

printf("%s\n",s);

}