



西安交通大学

## 编译原理

# 实 验 报 告

学院：电信学院

班级：计算机73班

姓名：杨志远

学号：2176112484

## 目 录

实验一 编译原理专题实验讲解.....	1
实验二 词法分析.....	4
实验三 语法分析器生成工具 YACC.....	9
实验四 语法分析结束.....	13
实验五 语义分析.....	16
实验六与七.....	18

# 实验一 编译原理专题实验讲解

## 一. 实验目的

- 1、熟悉 Linux 系统环境。
- 2、熟悉 Cool 语言的源程序语法特点。
- 3、了解实验系统组成及安装，掌握 Cool 语言，了解 Cool 程序编译和执行的工作过程。

## 二. 实验内容

### 1. COOL 语言的认识和基本了解

COOL 语言是轻量级的语言，但具有对象、静态类型和内存自动管理的特性。Cool 程序由类组成，类似 java 语言，有主类和主方法。Cool 语言是一种表达式(expression)语言。

### 2. 编译实验的基本介绍

编译实验主要包括词法分析，语法分析，语义分析和代码生成。

### 3. 实验环境的搭建及操作流程

安装 flex 和 bison。

每次实验前须修改相应的 cool/assignments/PA<sub>x</sub>( $x=\{1,2,3,4,5\}$ )下的 Makefile 文件中的 CLASSDIR，修改为 cool 文件夹在你的系统中的路径。执行 make 命令，并阅读 README 文件。

### 4. 完成 hello\_world 实例的运行

在 cool/cool/bin 目录下执行 ./coolc ../examples/hello\_world.cl 生成 hello\_world.s

执行 ./spim -trap\_file ../lib/trap.handler -file ../examples/hello\_world.s

### 5. 利用 COOL 语言设计一个运算栈

其中输入 int 压入一个 int 数字，输入 s 则压入 s，输入 e 则完成相应的动作，即若栈顶为 s 则弹出 s 并交换栈顶两个数字，若栈顶为 + 则弹出 + 和其后两个数字并压入和，x 退出。

其中 '+' 和 s 的实现如下

```
if stack.head() = "+" then
{
    stack <- stack.tail();
    (let a : Int <- new Int, b : Int <- new Int in
        {
            a <- z.a2i(stack.head());
            stack <- stack.tail();
            b <- z.a2i(stack.head());
            stack <- stack.tail();
            a <- a + b;
            stack <- stack.cons(z.i2a(a));
        }
    );
}
else
if stack.head() = "s" then
{
    stack <- stack.tail();
    (let a : String <- new String, b : String <- new String in
        {
            a <- stack.head();
            stack <- stack.tail();
            b <- stack.head();
```

```

        stack <- stack.tail();
        stack <- stack.cons(a);
        stack <- stack.cons(b);
    }
    );
}
else
    out_string("")

```

对于 '+'，读入 a, b, 将和压栈

对于 s, 读入 a, b, 交换后压栈

### 三. 实验结果

#### 1. hello\_world 实例的运行

```

test@ubuntu:~/Downloads/cool/cool/bin$ ./reference-lexer ../examples/hello_world.cl
bash: ./reference-lexer: Permission denied
test@ubuntu:~/Downloads/cool/cool/bin$ chmod 777 ./reference-lexer
test@ubuntu:~/Downloads/cool/cool/bin$ ./reference-lexer ../examples/hello_world.cl
#name "../examples/hello_world.cl"
#1 CLASS
#1 TYPEID Main
#1 INHERITS
#1 TYPEID IO
#1 '{'
#2 OBJECTID main
#2 '('
#2 ')'
#2 ':'
#2 TYPEID SELF_TYPE
#2 '{'
#3 OBJECTID out_string
#3 '('
#3 STR_CONST "Hello, World.\n"
#3 ')'
#4 '}'
#4 ';'
#5 '}'
#5 ';'

```

```

test@ubuntu:~/Downloads/cool/cool/bin$ ./spim -trap_file ../lib/trap.handler -file ../examples/hello_world.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ../lib/trap.handler
Hello, World.
COOL program successfully executed
test@ubuntu:~/Downloads/cool/cool/bin$

```

```

test@ubuntu:~/Downloads/cool/cool/assignments/PA1$ make
/home/test/Downloads/cool/cool/etc/link-shared 1 Makefile
Makefile already exists. Skipping Makefile.
/home/test/Downloads/cool/cool/etc/copy-skel 1 stack.cl README.SKEL
stack.cl already exists. Skipping stack.cl.
README.SKEL already exists. Skipping README.SKEL.
create stack.s
/home/test/Downloads/cool/cool/bin/coolc stack.cl
test stack.s
/home/test/Downloads/cool/cool/bin/spim -trap_file /home/test/Downloads/cool/cool/lib/trap.handler -file stack.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /home/test/Downloads/cool/cool/lib/trap.handler
Hello, World.
COOL program successfully executed

```

a)./coolc ../examples/hello\_world.cl 命令生成 hello\_world.s

b)./spim -trap\_file ../lib/trap.handler -file ../examples/hello\_world.s 输出结果

c)make

## 2. 运算栈的运行

```
Loaded: /home/test/Downloads/cool/cool/lib/trap.handler
>2
>+
>3
>d
3
+
2
>s
>d
s
3
+
2
>e
>d
+
3
2
>e
>d
5
>x
Abort called from class Main
test@ubuntu:~/Downloads/cool/cool/assignments/PA1$
```

a)2, +, 3, d, 分别压栈, 栈: 2 + 3d

b)s 执行 d 交换, 栈: 2 3 +

c)s 执行 ' + ' 求和, 栈: 5

## 四. 实验总结

这是我第一次使用虚拟机, 一开始对环境不很熟悉, 操作起来有些茫然。所幸老师提供了例程。通过学习例程, 我们小组通过合作, 在老师给的框架的基础上实现了运算栈。第一次使用 cool 语言编程取得的小小成功也带来了不小的成就感。在这次实验尝试中, 我对 linux 系统与 cool 语言有了初步的认识。运算栈的构造也加深了我对栈的理解。

## 实验二 词法分析

### 一. 实验目的

掌握词法单元的正规式描述，词法分析器的原理及工作过程。

### 二. 实验要求

构造并调试成功词法分析器。写出 Cool 词法单位的正规式表示、建立符号表。

### 三. 实验原理

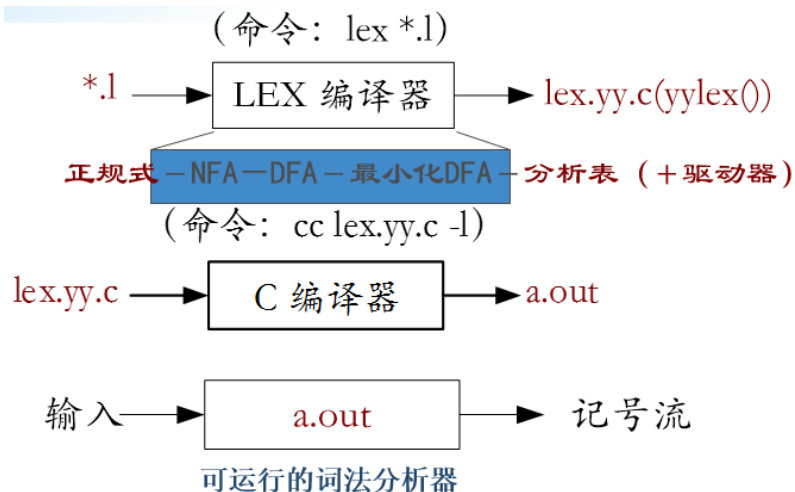
词法分析是将 Cool 源程序中的词素切分成一个个 token，并做一些规范性检查。主要作用是根据语言的词法规则对输入的源程序字符流进行分析，识别出一个个的单词，再将各单词对应的词法记号依次提供给语法分析器，这些记号将作为语言文法的终结符。在词法分析的过程中，还进行词法检查，能跳过源程序的注释、换行符、空白符及一些无用符号，并进行行列计数，用于定位源程序出错的行列号及出错部分。

词法分析主要包括以下几部分内容：

- (1) 关键字：由程序语言定义的具有固定意义的标识符号，如 if,else 等。
- (2) 标识符：用来表示各种名字，如变量名、数组名、类型名称等。
- (3) 常数：常数的类型一般由整型、浮点型等；
- (4) 运算符：如 +, -, \*, / 等；
- (5) 分界符：如逗号、分号、括号等。

### 四. 实验步骤

#### 1. Flex 词法分析器分析流程



#### 1.1 实现行数的统计

`\n {++nline;}` 即匹配到一个换行符，nline 加 1

#### 3. 分析行数，单词数，字符数

`\n {++nline;}` 匹配到一个换行符，nline 加 1

`[\t] {}` 忽略占位符

`(\{digit\}|\{char\})(\{char\}|\{digit\})* {++nword; nchar+=yylen;}` 匹配到单词，nword+1，同时利用所提供的 `yylen` 函数统计字符数

#### 1.2 对给定程序文件的词法分析

```
#include <stdio.h>
#include <string.h>
int nchar, nword, nline; /* 分别记录字符个数、字数和行数 */
%}
%option yylineno
%%

[\t]          /* 匹配到一个空格或 Tab 键就“吃”掉它们 */
\n           {nline=nline+1;} /* 匹配到一个换行符，行数加 1 */
[^\t\n]+ {
```

```

        /* 匹配到一个不包括空格、Tab 键和换行符的字，
           字数加 1，字符数加 yyleng(字符长度) */
        nchar=nchar+yyleng;
        nword=nword+1;
    }

```

```

%%
int main()
{
    FILE *fp;
    char str[50];
    printf("Press CTRL+d to quit.\nInput file's location:\n");
    scanf("%s",str);
    yyin=fopen(str,"r");
    yylex();          /* 调用词法分析器，直到输入结束 */
    printf("nchar=%d, nword=%d, nline=%d\n", nchar, nword, nline);
    return 0;
}

```

### 1.3.统计给定程序中 if 个数

```

%{
/*
 * 统计单词数、行数、字符数
 * 用 gcc 生成可执行程序:
 *   flex -o ex1.c lexsamp1.l
 *   gcc -o ex1 ex1.c
 * 之后运行 ./ex1 即可运行
 */
#include <stdio.h>
#include <string.h>
int nchar, nword, nline, nif; /* 分别记录字符个数、字数和行数 */
%}
%option yylineno
%%
"if" {nif=nif+1;
      nchar=nchar+2;
      nword=nword+1;}
[ \t]          /* 匹配到一个空格或 Tab 键就“吃”掉它们 */
\n             {nline=nline+1;} /* 匹配到一个换行符，行数加 1 */
^[ \t\n]+ {
    /* 匹配到一个不包括空格、Tab 键和换行符的字，
       字数加 1，字符数加 yyleng(字符长度) */
    nchar=nchar+yyleng;
    nword=nword+1;
}

```

```
}
```

```
%o%
```

```
int main()
```

```
{
```

```
    FILE *fp;
```

```
    char str[50];
```

```
    printf("Press CTRL+d to quit.\nInput file's location:\n");
```

```
    scanf("%s",str);
```

```
    yyin=fopen(str,"r");
```

```
    yylex();          /* 调用词法分析器，直到输入结束 */
```

```
    printf("nchar=%d, nword=%d, nline=%d\n, nif=%d\n", nchar, nword, nline, nif);
```

```
    return 0;
```

```
}
```

## 2.1 多重入口的实现

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

```
%o}
```

```
%start AA BB CC
```

```
%o%
```

```
^a      { ECHO; BEGIN AA; }
```

```
^b      { ECHO; BEGIN BB; }
```

```
^c      { ECHO; BEGIN CC; }
```

```
\n|(\t)+|" "+ { ECHO; BEGIN 0; }
```

```
magic   { printf ("zero");}
```

```
*/
```

```
<AA>magic { printf ("first"); break;}
```

```
<BB>magic { printf ("second");break;}
```

```
<CC>magic { printf ("third");break;}
```

```
magic   { printf ("zero");}
```

```
int main()
```

```
{
```

```
    int a;
```

```
    a = yylex();
```

```
    return 0;
```

```
}
```

## 2.2 互斥入口的实现

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

```
%o}
```

```
%x AA BB CC
```

```
%o%
```

```
^a      { ECHO; BEGIN AA; }
```

```
^b      { ECHO; BEGIN BB; }
```

```
^c      { ECHO; BEGIN CC; }
```

```
/*注释不能顶头 */
```

```
\n|(\t)+|" "+ { ECHO; BEGIN 0; }
```



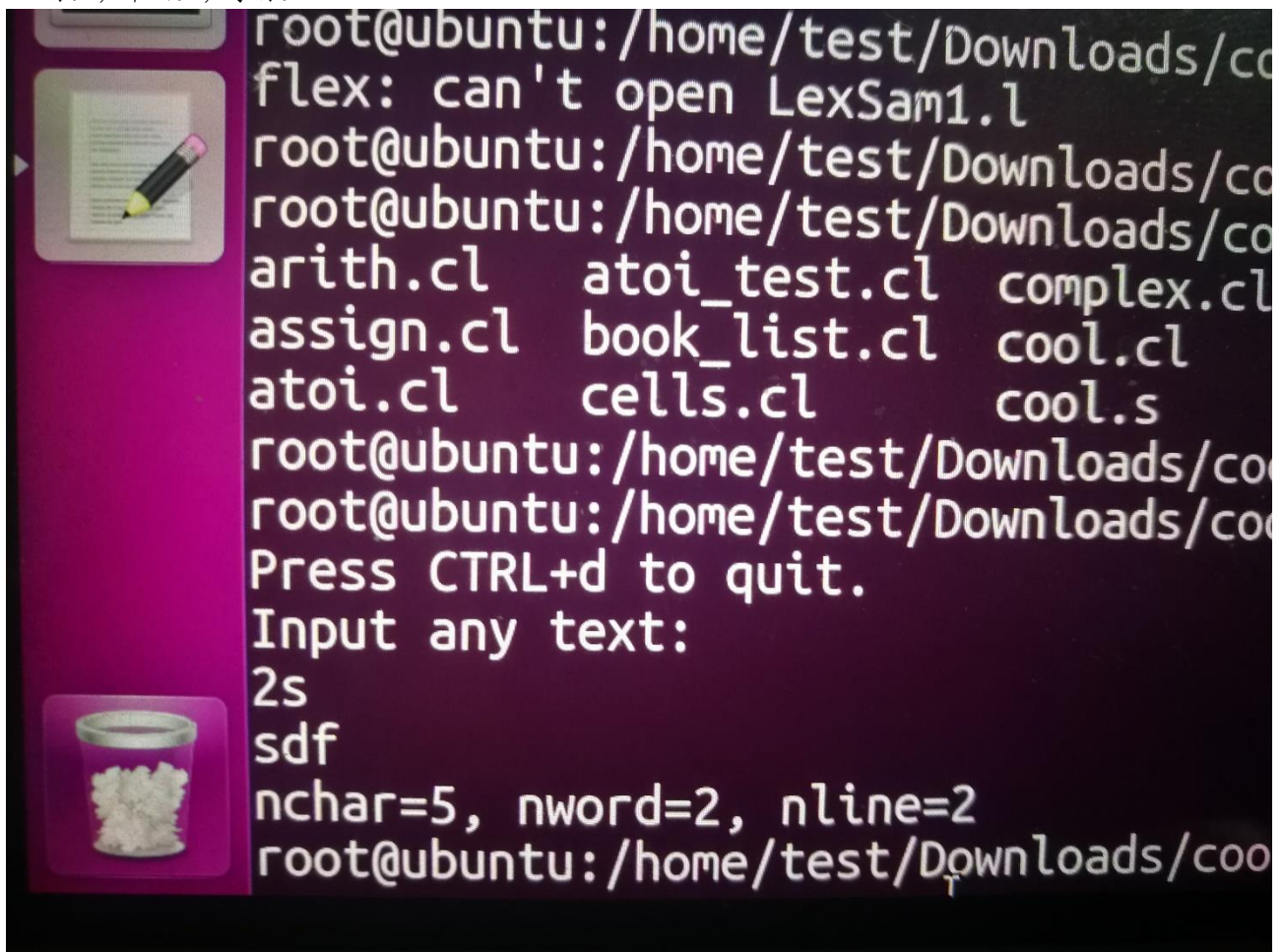
```

/**/
/* 如果这个规则位于所有<?>magic 之前, 则
   * <?>magic 规则永远无法起作用
   *
magic      { printf ("zero");}
*/
<AA>magic  { printf ("first");}
<AA>\n     {BEGIN 0;}
<BB>magic  { printf ("second");break;}
<BB>\n     {BEGIN 0;}
<CC>magic  { printf ("third");break;}
<CC>\n     {BEGIN 0;}
%%
int main()
{
    int a;
    a = yylex();
    return 0;
}

```

## 五. 实验结果

### 1.1 行数, 单词数, 字符数



每出现一个换行符, numline++

每出现一个单词, numword++, 同时用 numchar 统计字符数。

### 1.2 对给定程序文件的词法分析

所分析的源程序如下:

```

class Main inherits IO {
  main(): SELF_TYPE {
    out_string("Hello, World.\n")
  };
};

```

结果如下图：

```

test@ubuntu:~/Downloads/cool/cool/examples$ ./exp2 exp2
Press CTRL+d to quit.
Input file's location:
hello_world.cl
nchar=69, nword=12, nline=5

```

1.3 实现统计一个给定程序的个数，如统计一个程序中 if 的个数（选做）

所分析的源程序如下：

```

class Main inherits IO {
  main(): SELF_TYPE {
    if then
  };
};

```

```

test@ubuntu:~/Downloads/cool/cool/examples$ cc -o exp5 lex.yy.c -ll
test@ubuntu:~/Downloads/cool/cool/examples$ ./exp5 exp5
Press CTRL+d to quit.
Input file's location:
hello_world.cl
nchar=47, nword=12, nline=5
, nif=1
test@ubuntu:~/Downloads/cool/cool/examples$

```

2.1 实现多重入口

```

test@ubuntu:~/Downloads/cool/cool/examples$ cc -o exp5 lex.yy.c -ll
test@ubuntu:~/Downloads/cool/cool/examples$ ./exp5 exp5
Press CTRL+d to quit.
Input file's location:
hello_world.cl
nchar=47, nword=12, nline=5
, nif=1
test@ubuntu:~/Downloads/cool/cool/examples$

```

2.2 采用多种方式实现多重入口（选做）

互斥入口

```

test@ubuntu:~/Downloads/cool/cool/examples$ flex t4i.l
test@ubuntu:~/Downloads/cool/cool/examples$ cc -o exp4i lex.yy.c -ll
test@ubuntu:~/Downloads/cool/cool/examples$ ./exp4i exp4i
amagic
afirst

bmagic
bsecond

```

## 六. 实验总结

这次实验是一次比较成功的实验。在老师的帮助下，我们小组三人通过讨论与查阅相关资料，各自分别实现了这次实验的全部实验内容。由于实验后只有一份程序保存留下来，所以我们组统一使用了这份程序。

在这次实验中，我们了解到了词法分析器的构造过程，也分别实现了多重入口与互斥入口。在实验中，我们也发现了在构造过程中，所写的词法分析语句的顺序不同，可能会对最后的分析产生巨大的影响。所以这也要求我们在构造词法分析器的时候，做到耐心细致，充分全面的考虑各种可能的情况。此外，我们还学习到了在 linux 系统中，如何在文件中通过 fopen 函数调用其他的给定文件。

## 实验三 语法分析器生成工具 YACC

### 一. 实验要求

#### 作业一 必做

- 编写一个简单的.y 文件，能实现一个简单的一位数十进制计算器，包括+，-，\*，/运算。考虑优先级和结合性，比如：括号，输入4\*(1+4)，输出20。(必做，只使用 yacc)
- 在任务1的基础上实现多位数字的处理，能识别负数，增加报错功能。(必做)
- 在任务2的基础上使用.l 文件完成词法分析识别数字，实现浮点数处理功能。(必做 flex 和 yacc 联合使用)

#### 作业二 选做2

完成布尔表达式的求值

- 输入以 true/false 组成的布尔表达式
- 分析表达式语法关系 (||, &&, !)，求值

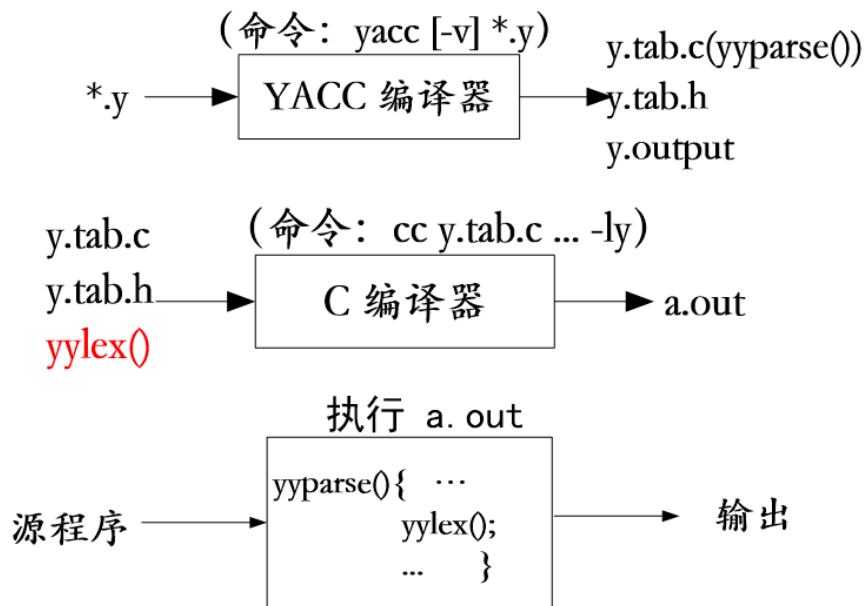
### 二. 实验原理 (YACC 概述)

YACC: Yet Another Compiler Compiler. 是一个语法分析器的生成器，接受用 BNF 形式的 LALR(1) 文法 (LL(1)、SLR(1)的真超集)。

#### 1. 工作方式

分析器的构造步骤：

**产生式 → 识别活前缀的DFA → 分析表 (+ 驱动器)**



#### 2. YACC 源程序

声明部分：

##### 1) 文法的开始符号：

%start n\_name (默认第一个产生式的左部是开始符号)

##### 2) 终结符：

直接表示：‘-’ 和 ‘\*’ 等； (无需说明，即可在产生式中使用)

名字： %token t\_name (内部编码，LEX 中使用 t\_name)

##### 3) 终结符的优先级与结合性：

结合性：

左结合 — %left

右结合 — %right

无结合性 — %nonassoc

优先级：从上到下依次递增

%left '+' '-' /\* 优先级 低 \*/

%left '\*' '/' /\* 优先级 高 \*/

4) 重新定义语义栈类型。

用户自定义子程序：

这部分与 LEX 的第三部分作用相同，所有语义动作中的 C 源程序均可定义在此

3. 翻译规则

翻译规则  $\rightarrow$  非终结符 ':' 候选项集 ':'

候选项集  $\rightarrow \epsilon$

| 候选项

| 候选项集 '|' 候选项

候选项  $\rightarrow$  文法符号序列 语义动作

语义动作  $\rightarrow \epsilon$  | 嵌入语义动作

嵌入语义动作  $\rightarrow$  '{' C 语言语句序列 '}'

文法符号序列  $\rightarrow \epsilon$

| 文法符号序列 文法符号

文法符号  $\rightarrow$  终结符 | 非终结符 | 嵌入语义动作

### 三. 实验结果

1. YACC 实现简单计算器

```
%{
#include "parser.tab.h"
%}
%%

[0-9]+ {yylval=atoi(yytext);return T_NUM;}
- {return T_MINUS;}
\* {return T_MUL;}
\/ {return T_DIV;}
\( {return T_LEFT;}
\) {return T_RIGHT;}
\+ {return T_ADD;}
\n {return T_NEWLINE;}
. {return 0;}
%%

int yywrap(void){
    return 1;
}

%{
#include <stdio.h>
void yyerror(const char *str){}
%}

%token T_MINUS T_ADD T_NUM T_NEWLINE T_MUL T_DIV T_LEFT T_RIGHT

%%

S :E T_NEWLINE {printf("ans=%d\n",$1);}
```

```

| /* empty */ { /* empty */}
;

E :E T_MUL E {$$=$1*$3;}
  |T_LEFT E T_RIGHT {$$=$2;}
  |E T_ADD E {$$=$1+$3;}
  |E T_MINUS E {$$=$1-$3;}
  |T_NUM {$$=$1;}
;

%%

int main() {
    return yyparse();
}

2. 布尔表达式的求值

%{
#include "parser.tab.h"
%}

%%

TRUE {return T_T;}
FALSE {return T_F;}
\|\| {return T_OR;}
&& {return T_AND;}
\! {return T_NOT;}
\( {return T_LEFT;}
\) {return T_RIGHT;}
\n {return T_NEWLINE;}
. {return 0;}
%%

int yywrap(void) {
    return 1;
}

%{
#include <stdio.h>
void yyerror(const char *str) {}
%}

%token T_T T_F T_AND T_OR T_NOT T_LEFT T_RIGHT T_NEWLINE

%%

S :E T_NEWLINE {if($1)
                printf("true\n");
                else
                printf("false\n");}

```

```

| /* empty */ { /* empty */ }
;

E :E T_AND E {$$=$1&&$3;}
  |T_LEFT E T_RIGHT {$$=$2;}
  |E T_OR E {$$=$1 || $3;}
  |T_NOT E {$$=!$1;}
  |T_T {$$=1;}
  |T_F {$$=0;}
;

%%

int main() {
    return yyparse();
}

```

## 四. 运行结果

### 1. 完成计算器

```

test@ubuntu:~/Downloads/cool/cool/include/PA3$ ./calc
4*(1+4)
ans=20

```

运行结果符合四则运算的规则，且支持对浮点数的计算。

### 2. 完成布尔表达式的求值

```

test@ubuntu:~/cool/include/PA3/4.2.1$ ./calc
TRUE
true

test@ubuntu:~/cool/include/PA3/4.2.1$ ./calc
TRUE&&FALSE
false

test@ubuntu:~/cool/include/PA3/4.2.1$ ./calc
TRUE || FALSE
true

```

## 五. 实验总结

在这次实验中，我们成功实现了简单计算器和布尔求值。我们小组一开始花了大量的时间在讨论计数器带括号的实现。因为开始时我们认为以例程中给出的方法增加对“（”和“）”的声明，可能不能按照运算规则实现类似  $a*(b+c)$  等运算的分析。最后我们决定进行一下实际操作，结果惊喜的发现通过例程给出的方法编写的语义分析语句就能够很好的完成各种运算语句的识别。之后我们分析原因，认为是  $a*(b+c)$  运算顺序的实现，需要基于语义的分析，这部分可能已经封装完成。

在布尔表达式语法分析器的构造中，我们主要碰到的一个问题就是“||”符号的声明。刚开始时，我们的程序始终不能编译通过。通过请教老师，我们发现了“||”与“|”可能存在二义性，所以对“||”转译时，需要对两“|”分别转译（即 `\|\| {return T_OR;}`）。通过这样的方式，我们成功解决了这个问题，也最终比较顺利的实现了布尔运算的语法分析器。

## 实验四 语法分析结束

### 一. 实验原理

本次实验完成语法分析后续部分, 根据 cool 语言的标准文法 (产生式) 完成 cool 语言语法分析, 输出语法, 分析树结果。

### 二. 实验步骤

补充 cool.y 的内容:

1) 为引入的新的非终结符添加额外的类型声明;

2) 完成终结符的部分;

3) 操作符添加优先级声明

1. hello world 语法分析

```
program : class_list { ast_root = program($1); }
        ;

class_list : class
            /* single class */      { $$ = single_Classes($1); parse_results = $$; }
        ;

class: CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
      { $$=class_($2,$4,$6,stringtable.add_string(curr_filename));};

feature_list: /* empty */ { $$=nil_Features();}
            | feature_list feature
            { $$=append_Features($1,single_Features($2));};

feature: OBJECTID '(' ')' ':' TYPEID '{' expression '}' ';'
        { $$=method($1,nil_Formals(),$5,$7);};

expression: OBJECTID '(' para_list ')'
           { $$=dispatch(object(idtable.add_string("self")), $1,$3);};
para_list: expression { $$=single_Expressions($1);};
expression: OBJECTID '(' para_list ')'
           { $$=dispatch(object(idtable.add_string("self")), $1,$3);}
           | STR_CONST { $$=string_const($1);}
```

2. 加法类语法分析

```
program : class_list { ast_root = program($1); }
        ;

class_list : class
            /* single class */      { $$ = single_Classes($1); parse_results = $$; }
        ;

class: /*CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
      { $$=class_($2,$4,$6,stringtable.add_string(curr_filename));}*/
      CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
      { $$=class_($2,$4,$6,stringtable.add_string(curr_filename));};

feature_list: /* empty */ { $$=nil_Features();}
            | feature_list feature
            { $$=append_Features($1,single_Features($2));};

feature: /*OBJECTID '(' ')' ':' TYPEID '{' expression '}' ';'
        { $$=method($1,$5,$7);}*/
        OBJECTID '(' OBJECTID ':' TYPEID ',' OBJECTID ':' TYPEID ')' ':' TYPEID '{' expression '}' ';
```

```
{$$=method($1,append_Formals(single_Formals(formal($3,$5)), single_Formals(formal($7,$9))),$12,$14);};
```

expression: OBJECTID '+' para\_list

```
{$$=dispatch(object(idtable.add_string("self")), $1,$3);};
```

para\_list: OBJECTID {\$\$=single\_Expressions(object(\$1));};

### 三. 实验结果

1.hello world 代码

```
class X inherits IO {  
  plus (a:Int, b:Int): Int {  
    a+b  
  };  
};
```

```
test@ubuntu:~/Downloads/cool/assignments/PA3$ ./lexer hello_world.cl | ./parser  
#5  
_program  
#5  
_class  
_Main  
_IO  
_hello_world.cl  
(  
#4  
_method  
_main  
SELF_TYPE  
#3  
_dispatch  
#3  
_object  
_self  
:_no_type  
out_string  
(  
#3  
_string  
_Hello,world.\n  
:_no_type  
)  
:_no_type  
)  
test@ubuntu:~/Downloads/cool/assignments/PA3$
```

2.加法类程序代码

```
class Main inherits IO{  
  main(): SELF_TYPE {  
    out_string("Hello,world.\n")  
  };  
};
```



```

test@ubuntu:~/Downloads/cool/assignments/PA3.1$ ./lexer minimal.cl | ./parser
#5
program
#5
class
X
IO
"minimal.cl"
(
#4
method
plus
#4
formal
a
Int
#4
formal
b
Int
Int
#3
dispatch
#3
object
self
: _no_type
a
(
#3
object
b
: _no_type
)
: _no_type
)
test@ubuntu:~/Downloads/cool/assignments/PA3.1$

```

#### 四. 实验总结

这次实验，我们实现了对“hello world”文件的语法分析与加法类程序的语法分析。这次实验，由于对这类语法分析器的了解还不够，所以开始的时候遇到了不少阻力。在老师的帮助下，我们加深了理解，最终完成了分析器的实现。

## 实验五 语义分析

### 一. 实验原理

本次实验完成语义分析中继承关系的检查，对所存在的循环继承报错输出。

### 二. 实验步骤

#### 1. 继承关系的检查

对形如如下的继承关系，根据有向图是否成环可以容易的判断出是否出现循环继承

```
class A inherits B {};
```

```
class B inherits C {};
```

```
class C inherits A {};
```

#### 2. 使用容器设计有向图的数据结构

算法思想：

program\_class 中可以访问程序的所有类，通过遍历这些类，建立类集成的有向图信息。从一个类出发，通过其继承链不断跳转到其父类，跳转过程中记录跳转历史，如果某个类型在跳转历史中出现过，则出现循环

```
int program_class::check_inheritance_graph()
{
// 二维 vector 存储继承信息
    std::vector<std::vector<Symbol> > son_parent;
    Symbol sonname,parentname;
    int cnt=0;
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        std::vector<Symbol> temp;
        cur_class = classes->nth(i);
        sonname=cur_class->get_name();
        parentname=cur_class->get_parent();
        temp.push_back(sonname);
        temp.push_back(parentname);
        son_parent.push_back(temp);
        cnt++;
    }
    // 生成跳转路径 path
    for (int i=0; i<cnt; i++){
        std::vector<Symbol> path;
        path.push_back(son_parent[i][0]);
        //遇到与当前节点的父节点相同的子字节点压入
        int j=i;
        for(int k=j+1;k<cnt;k++){
            if(son_parent[k][0]==son_parent[j][1]){
                path.push_back(son_parent[k][0]);
                j=k;
            }
        }
        path.push_back(son_parent[j][1]);
        //扔进 set 比较是否有相同元素
        std::set<Symbol> finalpath(path.begin(),path.end());
        if(path.size()!=finalpath.size()){
            //test
            std::cout<<"path: ";
            for (int i=0;i<path.size();i++)
                std::cout<<path[i]<<' ';
```

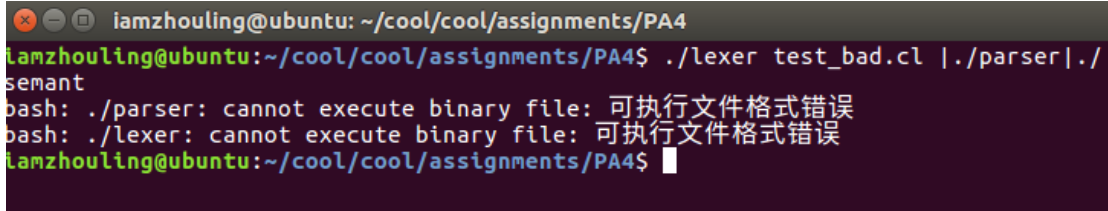
```

        std::cout<<endl;
        return 1;
    }
}
return 0;
}

```

使用二维 vector son\_parent 存储继承信息，然后从一个节点出发，根据继承信息生成跳转路径 path，将生成的 path 扔进 set 检查长度判断是否有相同元素。对每个节点重复上述工作，若均无重复则返回 0。

### 三. 实验结果



```

iamzhouling@ubuntu: ~/cool/cool/assignments/PA4
iamzhouling@ubuntu:~/cool/cool/assignments/PA4$ ./lexer test_bad.cl |./parser|./
semant
bash: ./parser: cannot execute binary file: 可执行文件格式错误
bash: ./lexer: cannot execute binary file: 可执行文件格式错误
iamzhouling@ubuntu:~/cool/cool/assignments/PA4$ 

```

### 四. 实验总结

这次实验实现了语义分析器。可以看出，该分析器对循环继承进行了报错，并打印了错误路径。这次实验也加深了我们对语义分析的理解。

## 实验六与七

### 一. 实验原理

本次实验接着完成语义分析的工作，使用符号表进行作用域的检查

### 二. 实验步骤

#### 1. 符号表对作用域的检查

- 1) 当前符号是否不存在当前作用域中（未定义就使用）
- 2) 当前符号是否重定义（同名同类型、同名不同类型）

#### 2. 思路分析

步骤 1：遍历整个 AST，即合理的递归调用各个节点的 analyze 函数，例如之前例子中的 method\_class 节点，需要依次调用其形参列表，返回值和函数体的 analyze 函数。正确的调用 analyze 函数是作用域检查的前提，可以通过在 analyze 函数中打印信息来验证 AST 遍历是否正确

步骤 2：分析得出当调用哪些类型节点的 analyze 时，需要（1）变换（进入或退出）作用域，或（2）在当前作用域增加新节点，或（3）查询验证当前符号是否在合理的作用域内

#### 3. 使用 AST 进行递归的语义检查

检查 plus 方法

检查第一个形参

检查第二个形参

检查 PlusExpr 表达式

检查左操作数

检查右操作数

检查返回类型是否一致

#### 4. cool 语言提供的支持

符号表：SymbolTable (symtab.h)

enterscope() 创建一个新的作用域，作为当前作用域的子域

exitscope() 销毁当前作用域，回到父级作用域

addid(symbol) 在当前作用域中添加一个新的符号

probe(symbol) 查看 symbol 是否在当前作用域中

lookup(symbol) 在当前域及所有父域中寻找 symbol 并返回

#### 5. 部分分析函数如下

```
void program_class::analyze()
{
    std::cout << "analyzing program " << std::endl;
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        cur_class = classes->nth(i);
        symboltable->enterscope();
        cur_class->analyze();
        symboltable->exitscope();
    }
}

void class__class::analyze()
{
    std::cout<<"analyzing class"<<std::endl;
    for (int i = features->first(); features->more(i); i = features->next(i)){
        Feature cur_fe = features->nth(i);
        cur_fe->analyze();
    }
}
```

首先，对每一个 class 建立一个新的作用域调用 analyze() 进行分析，并在分析完成后销毁当前作用域。然后重复地对于每一个具体的 feature 进行语义检查即可。

### 三. 实验结果

要分析的 cool 程序如下

```
class A inherits B {  
};  
class B {  
};  
class D {  
  foo1 (a : A) : SELF_TYPE {  
    self  
  };  
  foo2 (b : B) : SELF_TYPE {  
    self  
  };  
};  
class X {  
  plus(a:Int, b:Int) : Int {  
    a + b  
  };  
};  
Class Main inherits IO {  
  a : A;  
  b : B;  
  d : D;  
  main(): SELF_TYPE {  
    self  
  };  
};
```

分析结果如下:

analyzing program

analyzing class

analyzing class

analyzing class

analyzing method

method good

analyzing formal

formal good

a

A

analyzing object

not have been define, error

analyzing method

method good

analyzing formal

formal good

b

B

analyzing object

not have been define, error

analyzing class

analyzing method  
method good  
analyzing formal  
formal good  
a  
Int  
analyzing formal  
formal good  
b  
Int  
analyzing plus  
analyzing object  
already have been defined  
analyzing object  
already have been defined  
analyzing class  
analyzing attr  
analyzing no expr  
attr good  
a  
A  
analyzing attr  
analyzing no expr  
attr good  
b  
B  
analyzing attr  
analyzing no expr  
attr good  
d  
D  
analyzing method  
method good  
analyzing object  
not have been define, error  
(略去了生成的 AST 树)

#### 四. 实验总结

该语义分析器对于每一个进入的 class 进行检查，分析了其中各个变量的属性值并输出检查结果。

## 结语：实验心得与建议

编译上机实验让我第一次接触了虚拟机与 linux 系统。在实验中，老师对编译知识的讲解与实际的动手操作相辅相成，从词法、语法、语义各方面全面加深了我对编译原理的理解。在实验中，老师的帮助与小组的讨论拓宽了我对知识的认识，也帮我扫除了不少知识上的盲区。感谢老师同学们细致耐心的帮助。总而言之，这次的编译上机实验，对我可谓是受益匪浅。

当然，我觉得编译实验还有可以改进的地方：

其一，一部分实验课程相对与我们课内的学习课程，在知识点上存在超前现象，这也使得我们对实验内容的理解存在不到位的问题，完成实验内容就会比较困难。如果能够同步课内与实验的进步，效果会更加良好。

其二，cool 语言较为冷门，而了解一门新的语言需要一个比较长的过程和比较大的努力，所以关于实验语言，可否采用一些更大众的语言，以利于入门。

其三，部分内容确实难度较大，希望老师能够再讲解透彻些，否则，我们只能模仿例程，完成填空，但是对具体知识的掌握与运用帮助不大。

编译器实验是一门很重要的实验，希望编译器实验能够越办越好！