Yueq C Compiler 技术报告(申优)

齐悦 33060109

yueq@cse.buaa.edu.cn

目录

YCC 概述

Yueq C Compiler(YCC)是我在阅读过课本和 Kenneth Louden 的《编译原理及实践》一书之后，根据作业所要求的**中偏高**文法改进而成的。主要参考了 Pascal-S 编译器和 Tiny 编译器。

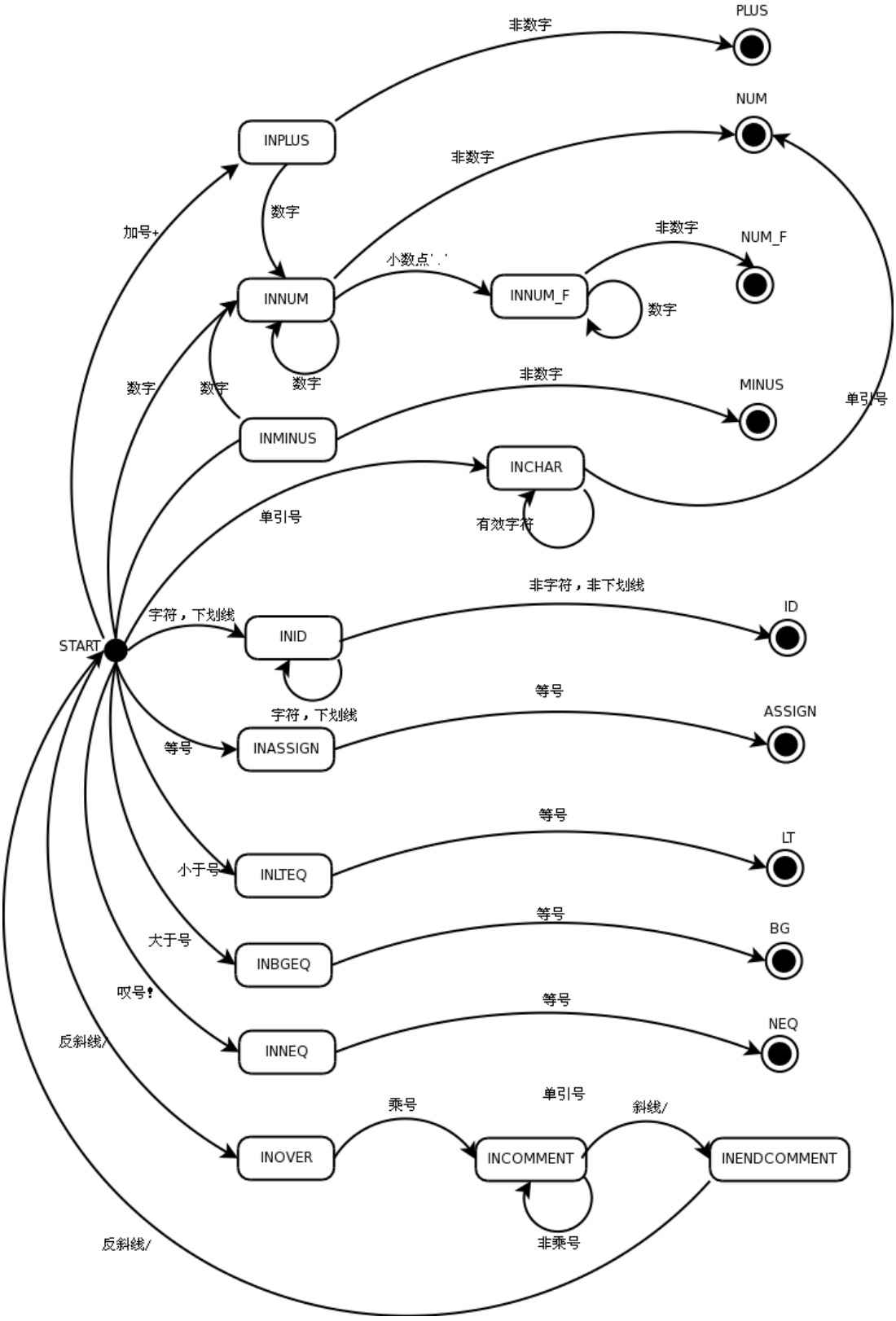
YCC 采用了多遍扫描的技术，将语法树构造、符号表构造、类型检查、代码生成分成四遍来进行，这与与 Wirth 给出的 Pascal-S 编译器的一遍扫描不同，其中的细节我会在下面具体给出。

关于本项目的更多信息可以参看 http://www.sourceforge.net/projects/ycc

词法分析

有穷自动机

YCC 中用一个 DFA 进行词法分析。给定文法的正则表达式被转换成如下所示的 DFA，为了记录 DFA 的状态，我使用了一个枚举类型来保存状态：



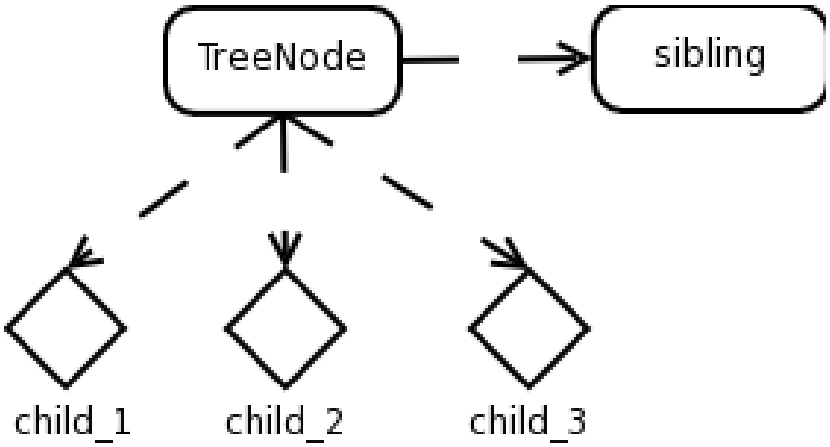
保留字

在 YCC 中有如下保留字:

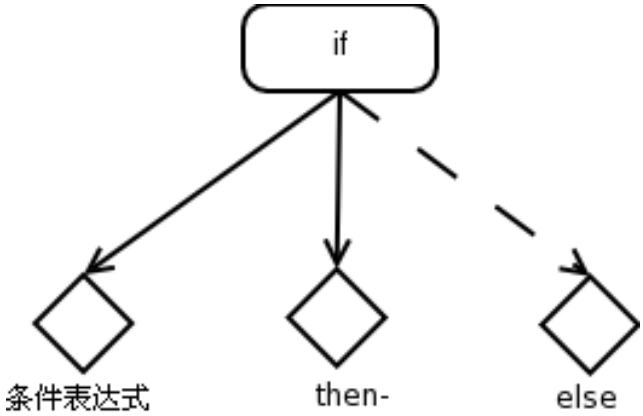
if	*else*	*do*	*while*	*switch*	*case*
int *return*	*float* *scanf*	*char* *printf*	*const*	*void*	*main*

语法分析

在语法分析阶段，YCC 进行第一次遍历，利用 parse()函数构造了一棵语法树。Parse()函数根据不同的语法成分构造了语法树的结点——孩子结点、兄弟结点，以及结点的结点类型、语句类型、表达式类型、变量名、变量值、变量类型、行号、等一系列内容。这些信息统一存放在 TreeNode 结构的结点内部。下面我用图形来展示一棵语法树的样子，其中实线代表必须出现的部分，虚线代表有可能出现的部分：

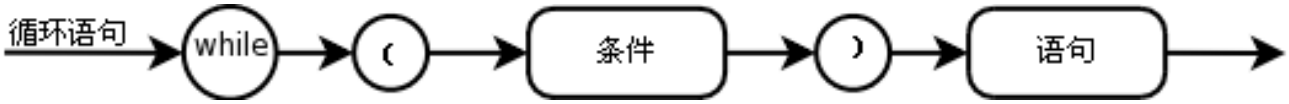


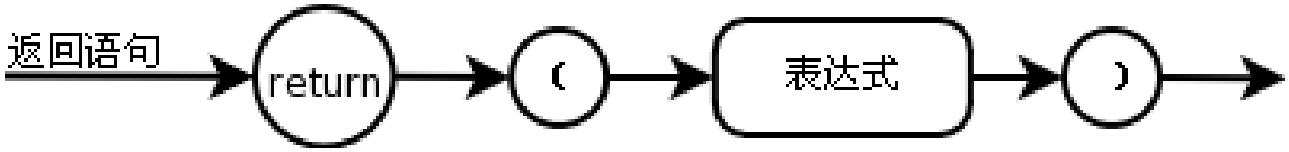
我用一个 IF 语句来说明语法树的样子：对于一个 IF 语句，其条件表达式和 then 部分是必须的（当然 then 部分也有可能为空），而 else 部分则是可选的。具体的语法树如下：

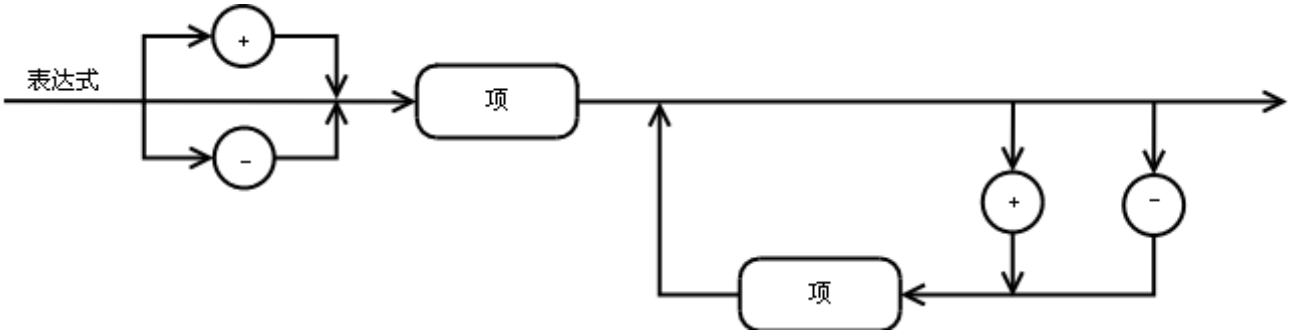


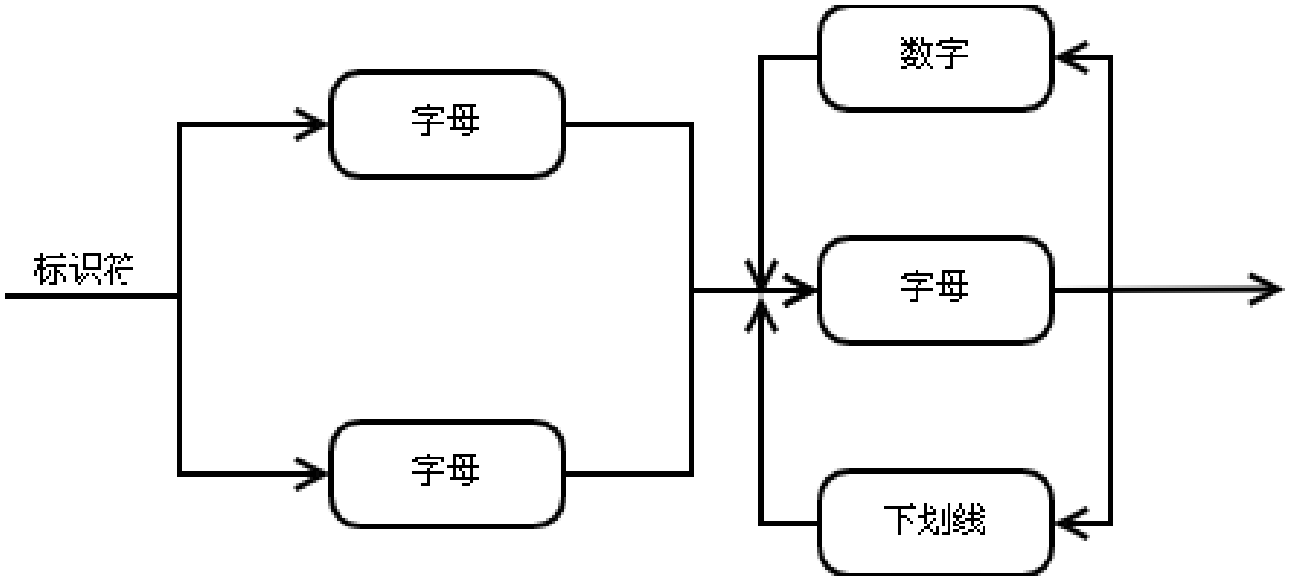
（我对文法进行了一定的扩充，使得这个编译器可以通过 do-while，数组等语法结构。扩充部分的产生式请详见附录。）

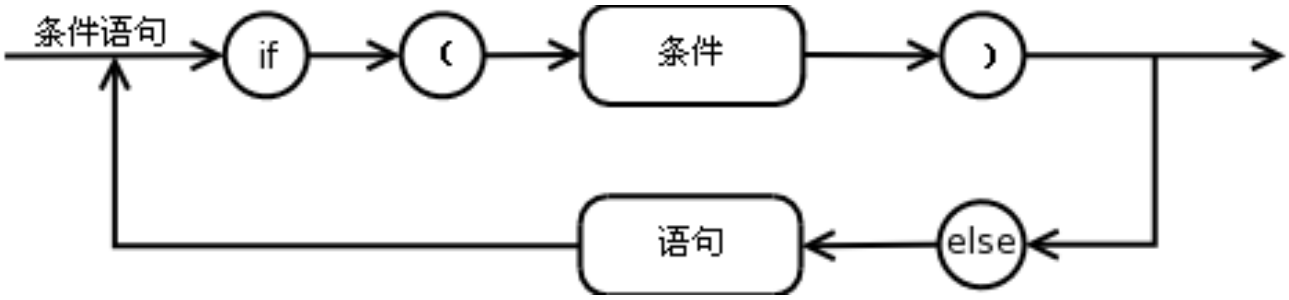
语法图

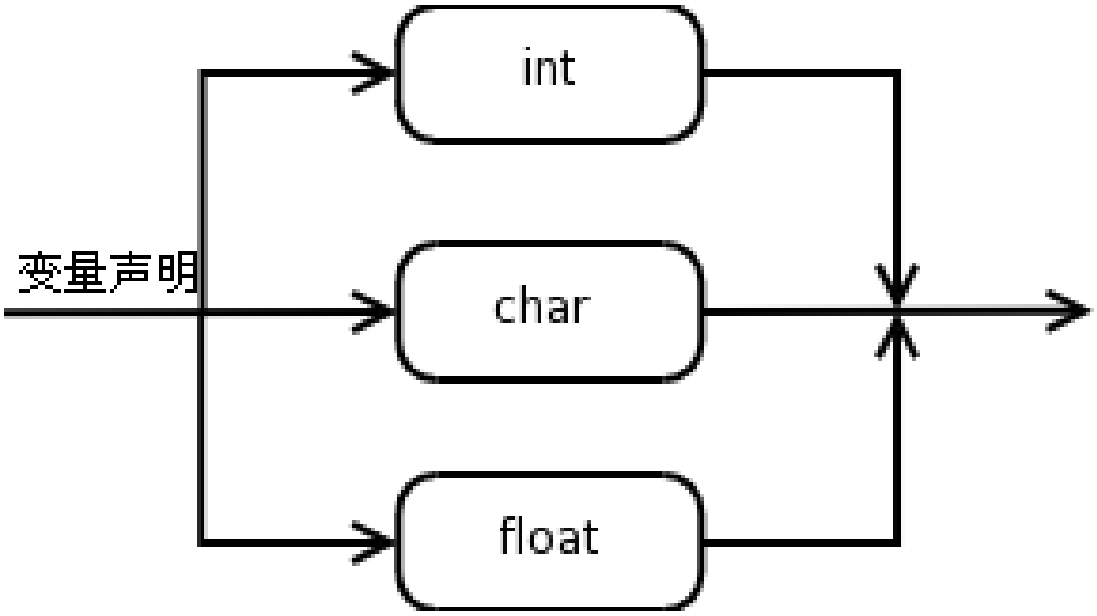

循环语句
while
(
条件
)
语句

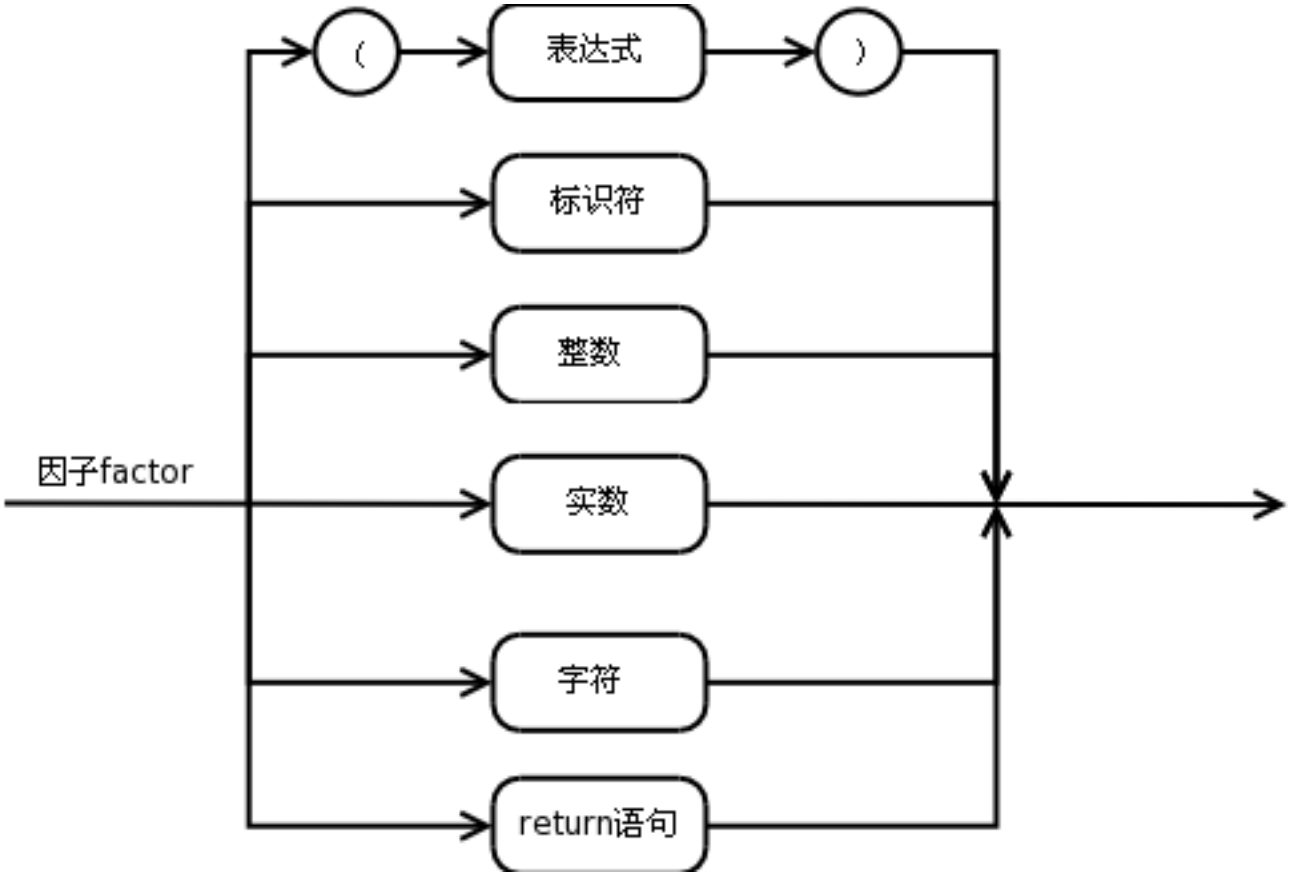

返回语句
return
(
表达式
)

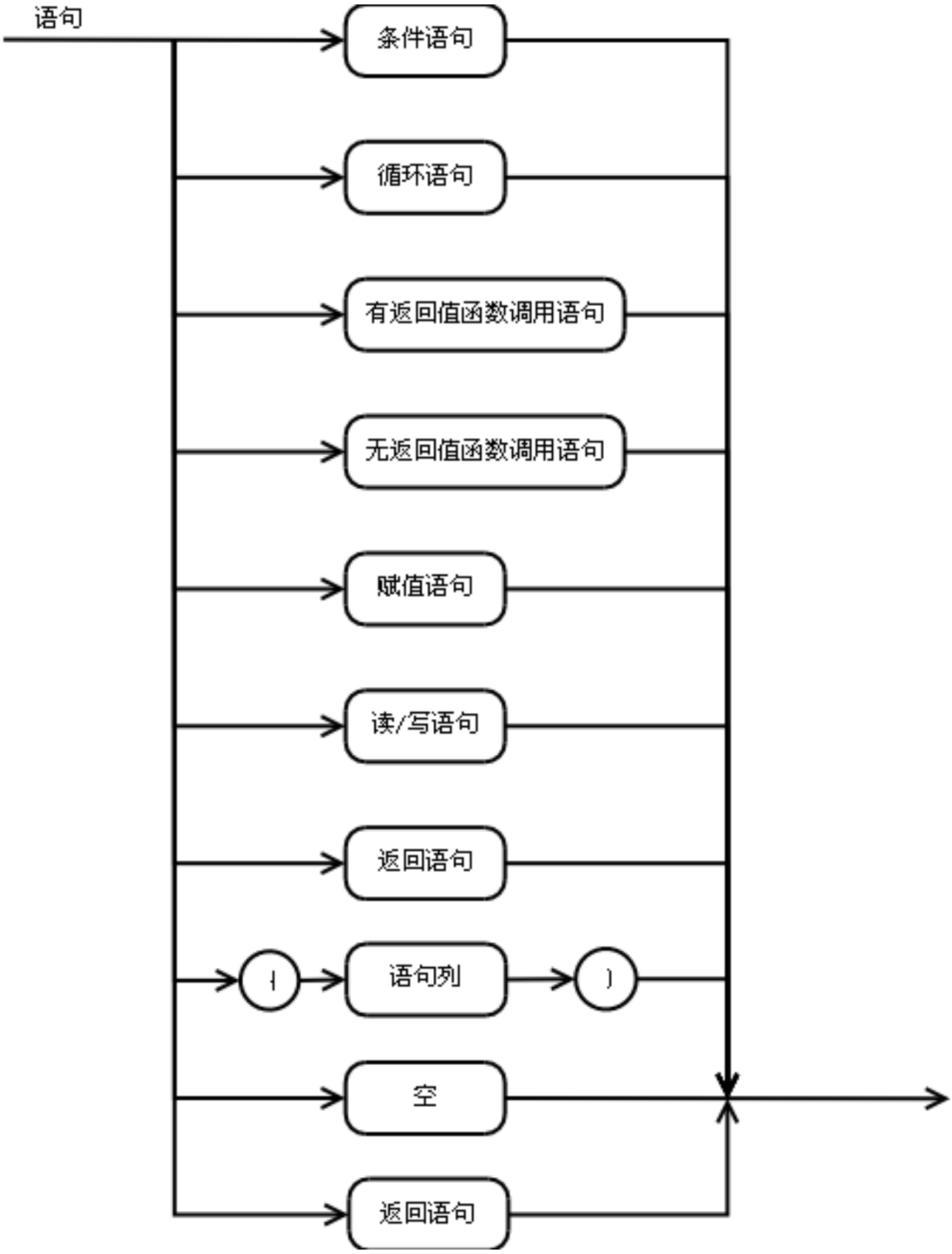

表达式
+
-
项
+
-
项


标识符
字母
字母
数字
字母
下划线


条件语句
if
(
条件
)
语句
else


变量声明
int
char
float


因子factor
(
表达式
)
标识符
整数
实数
字符
return语句

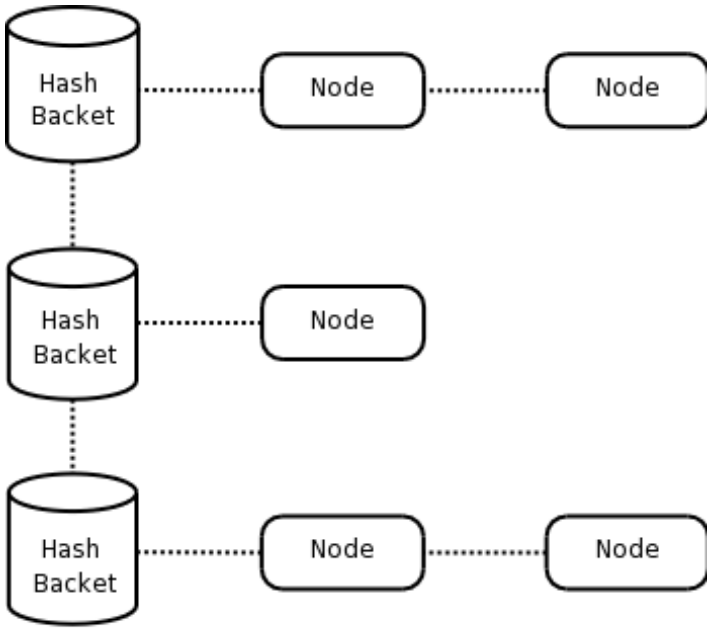

语句
条件语句
循环语句
有返回值函数调用语句
无返回值函数调用语句
赋值语句
读/写语句
返回语句
{
语句列
}
空
返回语句

语义分析

在语义分析阶段，YCC 通过 buildSymtab()函数进行了符号表构造，符号表构造是 YCC 四遍扫描中的一遍。

符号表结构

YCC 中的符号表是由一个二维链表构成的，这样作主要是为了提高存取速度。对每一个变量求得 Hash 值后，将其放入对应的 Hash 桶里。Hash 桶被保存在一个线性表中，每一个 Hash 桶都是这个链表中的一个结点。在查找一个结点时，只要算出 Hash 值后从相应的 Hash 桶找出就可以了。符号表可以用下图来进行说明：



类型检查

ANSI C 允许任意方向的隐式类型转换。但我在设计的时候，只是允许了隐式的向上类型转换。这样做主要是为了防止因为截断而造成的数据丢失。（在现代编译器中，向下类型转换应该给出一个 Warning 提示。）

YCC 中允许如下的隐式类型转换：

- char ==> int
- int ==> float

由于我们在语法分析时已经构造了一棵语法树，所以用树的后续遍历很容易完成这种隐式的向上类型转换检查。因为对于一个赋值语句，表达式左端是第一个孩子结点，右端是第二个孩子结点。当进行后续遍历的时候，我们会先读取到右表达式的类型再得到左表达式的类型。这样就可以检查两个类型是否满足上面的隐式类型转换的要求。

代码生成

虚拟机设计

YVM 虚拟机（Yueq Virtual Machine）是根据《编译原理及实践》中的 TM 虚拟机编写的，它没有采用 Pascal-S 中栈式虚拟机的设计思路，而是采用了一种类汇编的目标语言。通过寄存器操作，对数据进行读写。另外，由于这里的中间代码没有设计标号，所以跳转语句必须计算出跳转的绝对地址。

YVM 由数据区、指令存储区和 8 个通用寄存器构成。在通用寄存器中含有一个程序计数器 PC。

指令设计

操作码有 16 个，共分两类：一种是寄存器-存储器类型（R-M），一种是寄存器-寄存器类型（R-R）。

寄存器-寄存器指令

格式：操作码 r,s,t

ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t]
IN	reg[r] = scanf()
OUT	printf(reg[r])
HALT	程序终止

寄存器-存储器指令

格式：操作码 r,d(s)

这里操作数 r、s 为寄存器，而 d 代表偏移量。这种指令为两地址指令，第一个地址是一个寄存器，而第二个地址是存储器地址 a，用 a=d+reg[s]给出，这里 a 必须为有效地址。如果 a 超出正规的范围，在执行中就会产生 DMEM_ERR。

（以下 dMem[a]代表地址 a 中的值，类似于 C 语言中指针的星号操作）(a = d + reg[s])

LD	reg[r] = dMem[a]装入内存
LDA	reg[r] = a 装入地址
LDC	reg[r] = d 装入常数
ST	dMem[a] = reg[r] 将寄存器 r 的值装入地址 a 所指的空间内
JLT	dMem[a] = reg[r] 当寄存器 r 的值小于零时跳转到 a
JLE	寄存器 r 的值小于或等于零时跳转到 a
JGE	寄存器 r 的值大于零时跳转到 a

LD	reg[r] = dMem[a]装入内存
JEQ	寄存器 r 的值等于零时跳转到 a
JNE	寄存器 r 的值不等于零时跳转到 a

模拟器

YVM 模拟器可以读入虚拟机代码，对于虚拟机文件作出如下约定：

- 模拟器可以接受任意多行的空行
- 从分号开始到当前行结束为注释（这与 MASM 相同）

为了模拟 Microsoft 公司提供的调试工具 debug.exe，YVM 模拟器也提供了单步执行、查看寄存器状态等简单功能。具体功能如下表所示：

命令	*功能*
Go	顺序向下执行到 HALT
Step n	向下走 n 步
Reg	查看当前寄存器状态
Trace	回显走过的行
Print	打印走过的行数
Clear	清除寄存器
Help	显示帮助
Quit	退出模拟器

安装方法

YCC 在 GNU/Linux 2.6.15-1.1833_FC4smp、GCC 4.0.2 环境下编译通过。
在解压 ycc.tar.gz 文件之后，进入所在目录后，输入“make all”命令（或是输入 make 和 make yvm）。编译完成后，输入“./ycc 文件名”，便可以编译了。

测试样例

我写了一些样例以供测试使用。测试文件的命名是按以下规则的 test0.c0, test1.c0, test2.c0...我们假定用 test.c0 来做测试。
第一步，进入编译好的目录，输入“./ycc test0.c0”。若编译成功，则屏幕会回显“Compile Completed”，并生成相应的.vm 文件（Virtual Machine 虚拟机文件）。如果有错误发生则会回显错误处理的结果。
第二步，在当前目录下输入“./yvm test0.vm”便可以进入虚拟机解释执行目标代码了。

测试数据都在/test 子目录下，他们的所测试的功能如下表所示：

名称	*功能*
Test0.c0	基本语句测试

名称	*功能*
Test1.c0	递归、返回值测试
Test2.c0	数组测试
Test3.c0	综合测试 1
Test4.c0	综合测试 2

以及我扩充文法的一些测试程序：

名称	*功能*
Test_E0.c0	扩展测试 0
Test_E1.c0	扩展测试 1
Test_E2.c0	扩展测试 2
Test_E3.c0	扩展测试 3
Test_E4.c0	扩展测试 4

运行参数

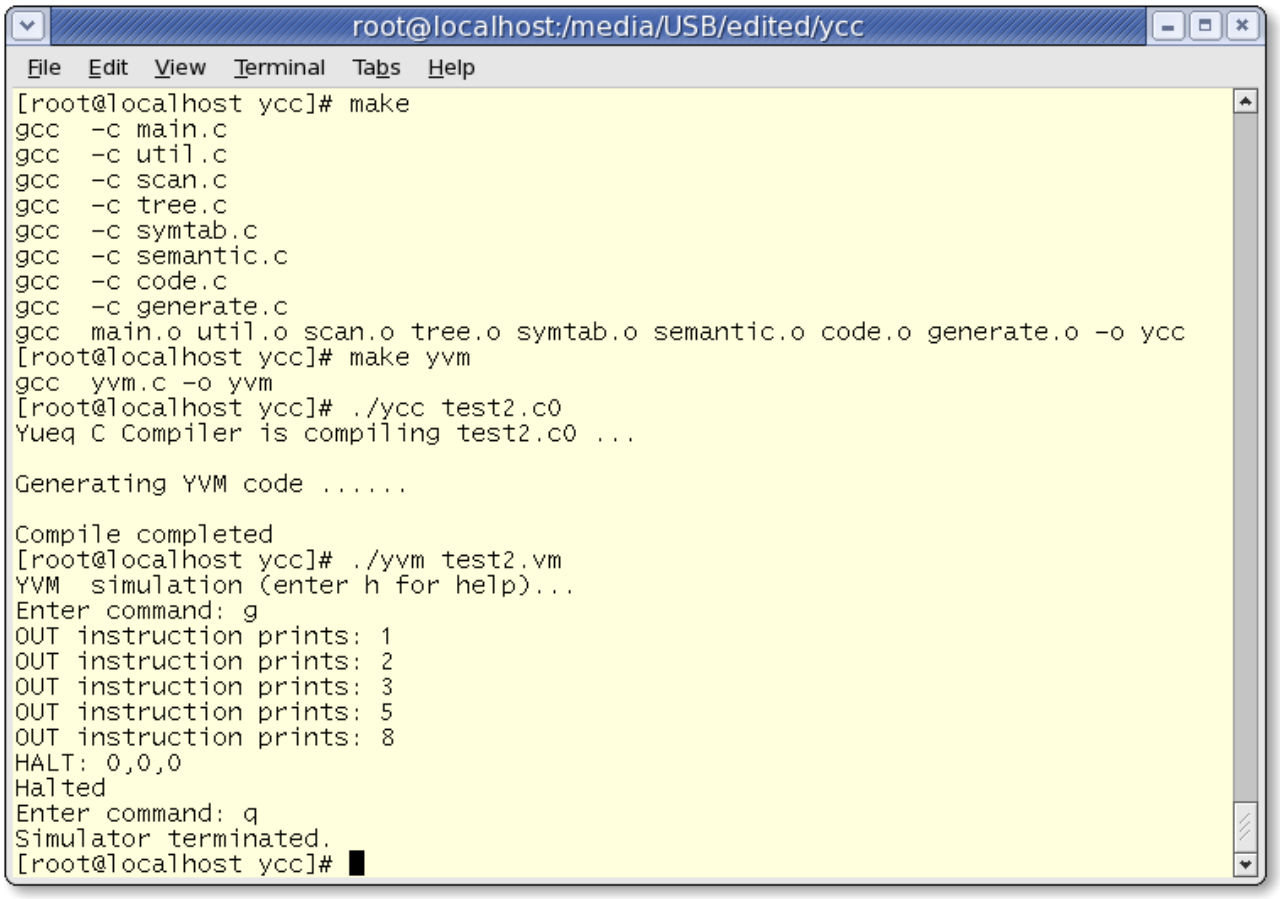
在 ycc 后加上如下参数可以得到相应功能，

如：ycc -p -e SOURCE.c0 就可以在编译 SOURCE.c0 之后看到源代码和语法树的相应结构。

参数	*功能*
-e	显示加行号的源代码
-s	显示词法分析过程
-p	显示语法分析过程
-t	显示语义分析过程
-h --help	显示帮助
-v --version	显示版本号

截图

下面是我在 Gnome Terminal 下面编译并解释运行 test2 测试数据的过程截图，test2 实现的是在程序内部对一个 5 元数组赋值为 Fibonacci 数列并将他们依次输出：



错误处理

在 PL/0 和 Pascal-s 中，每一个错误分配一个错误编号且都被保存在一个集合中，并当生成代码之前统一输出。YCC 采用了类似的机制，但我没有在这里设置统一的错误编号，而仅是指出错误的行号，并给出简单的错误信息。之所以选择这样做，是因为我认为对于一个程序员而言，只需要错误行号就可以解决 BUG。比如，我只给出“Expected token: （”，而不是将“缺少左括号”这个信息赋予一个编号，并在最后统一打印。

文件组成

YCC 目录下文件及其作用如下

/

scan.[ch]	词法分析
tree.[ch]	语法树构建
semantic.[ch]	语义分析
symtab.[ch]	符号表建立、类型检查
code.[ch]	代码生成接口
generate.[ch]	目标代码生成器

	util.[ch]	在 Louden[1]中提供的使用工具
	main.c	主函数
	yvm.c	虚拟机模拟器
	Makefile	makefile 文件
	Copying	一份 GPL 协议
/bin		存放已经在 FC4/GCC4.0 下编译通过的二进制文件
	ycc	c0 编译器
	yvm	虚拟机模拟器
/doc		
	ycc.odt	本文档的 Open Document Templete 格式文件，用 OpenOffice 打开
	ycc.doc	本文档的 Microsoft Word 格式文件，用 Word 打开
	ycc.pdf	本文档的 Portable Document Format 格式文件，用 Acrobat Reader 打开
/test		测试数据
	test0.c0	
	test1.c0	
	test2.c0	
	test3.c0	
	test4.c0	
	test_e0.c0	
	test_e1.c0	
	test_e2.c0	
	test_e3.c0	
	test_e4.c0	
/pics		存放本文档图的 dia 源文件（可用 Dia Digrams 打开）和 PNG 文件

开发总结

我的整个开发过程可以说是并不顺利：我首先阅读了课本中有关 PL0 和 Pascal-S 的章节，决定根据用所得到的文法对 Pascal-S 进行修改，并用 C 语言重写。在调试工作中，我发现 Pascal-S 中的一遍扫描并不是一个好的设计，甚至可以说有些糟糕——因为每当希望对某处进行修改的时候，我必须同时对语法分析、语义分析、错误处理甚至还有目标代码的生成同时进行改动，如果我不这样做的话，很可能我生成的目标代码和语义分析就是两回事了。当然，这和我没有做好跟踪也有关系，不过我觉得这么几千行的东西也没有必要动用庞大的 CASE 工具来辅助开发。之后我选择了《编译原理及实践》提供的 TINY 样例作为参考，因为其中给出的基于多遍扫描的 TINY 满足了我当时的需求——我需要一个模块化更强的东西来加快我的开发。所以我选择了放弃一遍扫描的 Pascal-S 但这个选择也让我写了 3000 多行的代码付诸东流。

另外一个让我感触很深的问题是版本控制。可以说，对于我来说，这样大小的程序没有版本控制基本属于 Mission Impossible 的程度。在以往自己的程序都是为了测试某些算法和熟练语言，所以通常都是只有一个文件。对于这种单人完成一个含有大量子程序调用和递归的程序，版本控制就显得异常重要。我曾经考虑过是否在本机跑一个 CVS 来完成版本控制，但做了两天之后发现我一般都想不起来 commit 更改。于是我每天都做一次所有文件的整体备份，来完成很初级的版本控制工作。

我觉得 C0 文法并不很适合用递归下降分析法来完成。比如 int a 和 int add(int x)就是这样。只有当我读到左括号的时候，才能判断是一个函数还是一个整形变量。但在 PL0 中则不然，关键字 FUNCTION 和 VAR 明确的表明了这是一个变量还是函数声明，所以可以轻松地转入相应的子程序。

值得一提的一点是，YCC 的整个过程都是使用开源软件开发的：代码编辑 vim、编译器使用 GCC、文档编辑用 OpenOffice Writer、画图用 Dia Diagrams。从最后的结果来看，这个文档的效果还是“相当的”不错：）。不过我仍然另存了一份 doc 文件以方便没有 OpenOffice 和 PDF 阅读器的人。

参考文献

[1].（美）Kennech C. Louden 著，冯博琴，冯岚译。《编译原理及实践》，北京：机械工业出版社，2000

[2].高仲仪，金茅忠编著。《编译原理及编译程序构造》，北京：北京航空航天大学出版社，1990

[3].霍林主编。《编译技术课程设计与上机指导》，重庆：重庆大学出版社，2001

附录

文法

基本文法 7A238

＜加法运算符＞ ::= ＋｜－
＜乘法运算符＞ ::= *｜/
＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==
＜字母＞ ::= _｜a｜. . .｜z｜A｜. . .｜Z
＜数字＞ ::= 0｜＜非零数字＞
＜非零数字＞ ::= 1｜. . .｜9
＜字符＞ ::= ‘＜加法运算符＞｜＜乘法运算符＞｜＜字母＞｜＜数字＞’
＜字符串＞ ::= “｛＜合法字符＞｝”　　//字符串中可以出现所有合法的可打印字符集中的字符
＜程序＞ ::= ［＜常量说明部分＞］［＜变量说明部分＞］{＜有返回值函数定义部分＞}{＜无返回值函数定义部分＞}＜主函数＞
＜常量说明部分＞ ::= const＜常量定义＞;{ const＜常量定义＞;}
＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}｜ float＜标识符＞＝＜实数＞{,＜标识符＞＝＜实数＞}｜ char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}
＜整数＞ ::= ［＋｜－］＜非零数字＞｛＜数字＞｝｜0
＜实数＞ ::= ［＋｜－］＜整数＞[.＜整数＞]
＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝
＜声明头部＞ ::= int＜标识符＞ |float ＜标识符＞|char＜标识符＞
＜变量说明部分＞ ::= ＜变量定义＞;{＜变量定义＞;}
＜变量定义＞ ::= ＜类型标识符＞＜标识符＞{,＜标识符＞}
＜常量＞ ::= ＜整数＞｜＜实数＞|＜字符＞
＜类型标识符＞ ::= int ｜ float ｜ char
＜有返回值函数定义部分＞ ::= ＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’
＜无返回值函数定义部分＞ ::= void＜标识符＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’
＜复合语句＞ ::= ［＜常量说明部分＞］［＜变量说明部分＞］＜语句列＞
＜参数＞ ::= ＜参数表＞
＜参数表＞ ::= ＜类型标识符＞＜标识符＞{，＜类型标识符＞＜标识符＞}｜ 空
＜主函数＞ ::= void main‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’
＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}
＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}
＜因子＞ ::= ＜标识符＞｜‘(’＜表达式＞‘)’｜＜整数＞｜＜有返回值函数调用语句＞|＜实数＞|＜字符＞ //返回值为 char 型的函数和＜字符＞，用字符的 ASCII 码参加计算
＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜ ‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞ |＜情况语句＞｜＜返回语句＞;
＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞
＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］
＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为 0 条件为假，否则为真
＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞
＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’
＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}
＜情况子语句＞ ::= case＜常量＞：＜语句＞
＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’
＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’
＜值参数表＞ ::= ＜表达式＞{，＜表达式＞}｜＜空＞
＜语句列＞ ::= ＜语句＞｛＜语句＞｝
＜读语句＞ ::= scanf ‘(’＜标识符＞‘)’
＜写语句＞ ::= printf ‘(’[＜字符串＞,][＜表达式 ＞] ‘)’
＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

扩充文法

＜声明头部＞ --> int＜标识符＞ |float ＜标识符＞|char＜标识符＞|＜数组声明＞
＜数组声明＞-->int＜标识符＞[＜整数＞]　　注意：这里数组下标只支持正整数。

```
<条件> ::= <表达式><关系运算符><表达式>|<表达式>|<赋值语句>
<语句> ::= <条件语句>|<循环语句>| '{'<语句列>'}' |<有返回值函数调用语句>;|<无返回值函数调用语句>; |<赋值语句>;|<读语句>;|<写语句>;|<空> |<情况语句>|<返回语句>|<do 语句>;
<do 语句>::= do '{'<语句列>'}' while(<表达式>);
<返回语句>::= return[ '(' <表达式> ')' ]|return[<表达式>]
<参数表>::= <类型标识符><标识符>{, <类型标识符><标识符>}| 空
空::= void | 空                                          //"空"与"<空>"不同, "空"为 epsilon
<注释> ::= /* {<合法字符>} */                             //注释不能嵌套、注释可以出现在任何位置。
<读语句>::=scanf '(' <标识符> ')'                         //读语句的返回值为其读入的值
```

简单来说，扩充文法增加了：

1. 一维整型数组　　如：int a[5];
2. 赋值语句作为返回值　　如：if (a = 3) {语句}　注意：赋值成功则为 1。
3. do-while 语句
4. K&R 第一版、第二版 return 形式均可接受，第二版形式为扩充文法
   　　如：return (a); 或 return a;
5. 空参数可写为 void　　如：int a(void){语句}
6. 注释　　如：/* this is in comment */

测试点

下列测试点包括了基本文法和扩充文法。Test0~test4 测试了基本文法和部分扩充文法。test_e0~test_e4 测试了所有扩充的文法。具体文件请见 test 子目录下的相关文件。

测试点 test0.c0

```
void main()
{
int temp, x, y;
x = 5;
y = 3;

printf(x);
printf(y);

if (x < y)
{
//swap
temp = x;
x = y;
y = temp;
}
```

```
printf(x);
printf(y);
}
```
运行结果：3、5

测试点 test1.c0

```
int gcd(int u,int v)
{
     if(v == 0)
          return (u);
     else
          return (gcd(v,u-u/v*v));
}

void main()
{
     int x;
     int y;
     x=scanf(x);
     y=scanf(y);
     printf(gcd(x,y));
}
```

测试点 test2.c0

```
void main()
{
int i;
int a[5]
a[0] = 1;
a[1] = 2;
a[2] = 3;
a[3] = 5;
a[4] = 8;

while (i < 5)
{
```

```
    printf(a[i]);
    i = i + 1;
}
}
```

运行结果：1、2、3、5、8

测试点 test3.c0

```
/* Quick Sort for 10 integers */
int x[10];
int minloc(int a[],int low,int high)
{
int i,x,k;
k=low;
x=a[low];
i=low + 1;
while(i<high)
{
if(a[i]<x)
{
x=a[i];
k=i;
}
i=i + 1;
}
return (k);
}
void sort(int a[],int low,int high)
{
int i;
int k;
int t;
i=low;
while(i<high - 1)
{
k=minloc(a[],i,high);
t=a[k];
a[k]=a[i];
```

```
a[i]=t;
i=i + 1;
}
}

void main(void)
{
int i;
i=0;
while(i<10)
{
x[i]=scanf(x[i]);
i=i + 1;
}
sort(x[],0,10);
i=0;
while(i<10)
{
printf(x[i]);
i=i + 1;
}
}
```

测试点 test4.c0

```
/* Confusing : ) */
int _;void __(int _){int _o_;if(_<'*')_=_ + 1;else{_o_ =1;}_o_=_o_ * _;return(_o_);}void main(void){_=1;printf(__(_));}
```

测试点 test_e0.c0

```
void main()
{
     int a;
     a = 10;
     do {
          a = a+1;
     } while(a < 20);

     printf(a);
```

```
}
```

测试点 test_e01.c0

```
void main()
{
     int a;
     /* this is a test for comment */
     printf(a);
}
```

测试点 test_e02.c0

```
/* 测试 K&R 2nd return */
int gcd(int u,int v)
{
     if(v == 0)
          return u;
     else
          return gcd(v,u-u/v*v);
}
void main()
{
     int x;
     int y;
     x=scanf(x);
     y=scanf(y);
     printf(gcd(x,y));
}
```

测试点 test_e03.c0

```
/* a test for return */
int a(int x)
{
     return x;
}
void main()
{
     int y;
```

```
    int b;
    y = 10;
    b = a(y);
    printf(b);
}
```

运行结果：10

测试点 test_e04.c0

```
void main()
{
    int a;
    a =2;
    if (a = 3)    /* test for expression as a para */
    {
        printf(a);
    }
}
```

运行结果：3