Extended PL/0 Compiler 详细设计文档

胡京徽(11061191)

January 2, 2014

1 需求说明

1.1 文法说明

鉴于中文的文法在编程时的对应关系容易混乱，我就先将所给的文法翻译成了英文。具体文法如下：

program ::= *block* .

block ::= [*constdec*][*vardec*]{[*procdec*]|[*fundec*]} *compstmt*

constdec ::= **const** *constdef* {, *constdef* };

constdef ::= *ident* = *const*

const ::= [+|−] *unsign* | *character*

character ::= ′ *letter* ′|′ *digit* ′

string ::= "{*ASCII characters with decimal code number varys from* 32 *to* 126 *exclude* 34}"

unsign ::= *digit* { *digit* }

ident ::= *letter* { *letter* | *digit* }

vardec ::= **var** *vardef* ; { *vardef* ; }

vardef ::= *ident* {, *ident* } : *type*

type ::= *basictype* |**array**′[′ *unsign* ′]′**of** *basictype*

basictype ::= **integer**|**char**

procdec ::= *prochead block* {; *prochead block* };

fundec ::= *funhead block* {; *funhead block* };

prochead ::= **procedure** *ident* ′(′[*paralist*]′)′;

funhead ::= **function** *ident* ′(′[*paralist*]′)′ : *basictype* ;

paralist ::= [**var**] *ident* {, *ident* } : *basictype* {; *paralist* }

statement ::= *assignstmt* | *ifstmt* | *repeatstmt* | *pcallstmt*
| *compstmt* | *readstmt* | *writestmt* | *forstmt* | *nullstmt*

assignstmt ::= *ident* := *expression* | *funident* := *expression*
| *ident* ′[′ *expression* ′]′ := *expression*

funident ::= *ident*

expression ::= [+|−] *term* { *addop term* }

term ::= *factor* { *multop factor* }

factor ::= *ident* | *ident* ′[′ *expression* ′]′| *unsign* |′(′ *expression* ′)′| *fcallstmt*

fcallstmt ::= *ident* ′(′[*arglist*]′)′

arglist ::= *argument* {, *argument* }

argument ::= *expression*

addop ::= +|−

multop ::= $*|/$

condition ::= *expression* *relop* *expression*

relop ::= $<|<=|>|>=|=|<>$

ifstmt ::= **if** *condition* **then** *statement*
|**if** *condition* **then** *statement* **else** *statement*

repeatstmt ::= **repeat** *statement* **until** *condition*

forstmt ::= **for** *ident* := *expression* (**to**|**downto**) *expression* **do** *statement*

pcallstmt ::= *ident* $'('[$ *arglist* $]')'$

compstmt ::= **begin** *statement* $\{;$ *statement* $\}$**end**

readstmt ::= **read**$'('$ *ident* $\{,$ *ident* $\}')'$

writestmt ::= **write**$'('$ *string* , *expression* $')'|$**write**$'('$ *string* $')'|$**write**$'('$ *expression* $')'$

letter ::= $a|b|c|...|z|A|B|C|...|Z$

digit ::= $0|1|2|3|...|9$

1.2 目标代码说明

最终生成的目标代码是x86汇编遵寻Intel的x86传统。具体细节可参考Intel的手册。这里就不多说了。

1.3 文法解读

接下来对文法进行具体解释:

1.3.1 程序的解读

program ::= *block* .

这句定义了程序是由分程序加“.”组成，其中“.”可以判定程序的结束。这句的语法图见Figure 1。 具体实例:

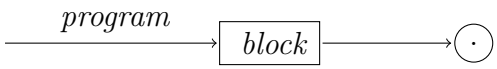

Figure 1: program 语法图

```
count a = 2; .
```

这就是一个完整的程序，“.”表示程序的结束。

1.3.2 分程序的解读

block ::= [*constdec*][*vardec*]{[*procdec*]|[*fundec*]} *compstmt*

constdec ::= **const** *constdef* {, *constdef* };

constdef ::= *ident* = *const*

vardec ::= **var** *vardef* ;{ *vardef* ;}

vardef ::= *ident* {, *ident* } : *type*

procdec ::= *prochead block* {; *prochead block* };

fundec ::= *funhead block* {; *funhead block* };

prochead ::= **procedure** *ident* '('[*paralist*]')';

funhead ::= **function** *ident* '('[*paralist*]')' : *basictype* ;

compstmt ::= **begin** *statement* {; *statement* }**end**

这几句定义了 *block* 文法的完整视图，语法图见Figure 2。在分程序 *block* 中先进行常量定义 *constdec* ，然后是变量定义 *vardef* ，接着是过程的声明 *procdec* 和函数声明 *fundec* 最终的 *block* 通过一个复合语句 *compstmt* 退出。这样的顺序是不能改变的。具体实例:

```
const numbera=0, numberb=1; // 常量定义
var i, j, sum: integer;     // 变量定义
procedure  test();          // 过程声明部分
  test 的程序（略）
function add():integer;     // 函数声明部分
begin
  sum := numbera + numberb; // add 的分程序部分
  add := sum
end;
                            // 复合语句部分
begin
  i := sum
end
```

说明:

1. 常量定义必须在变量前面，这种顺序不能改变。如: `var i:integer;const a=1;`就是错误的。
2. 相连的过程和函数的声明是可以打乱顺序的，过程和函数都可以右参数列表，用于传入参数。
3. 常量是可以连续定义的，之间使用逗号隔开，最后以分号结束常量的定义.
4. 变量的定义也可以连续定义，之间使用逗号隔开，另外使用冒号后跟变量类型来说明定义的变量的类型。变量的定义的结束是使用分号。
5. 常量定义，变量定义，过程声明，函数声明对一个分程序来说是可有可无的，只有复合语句是必须部分。

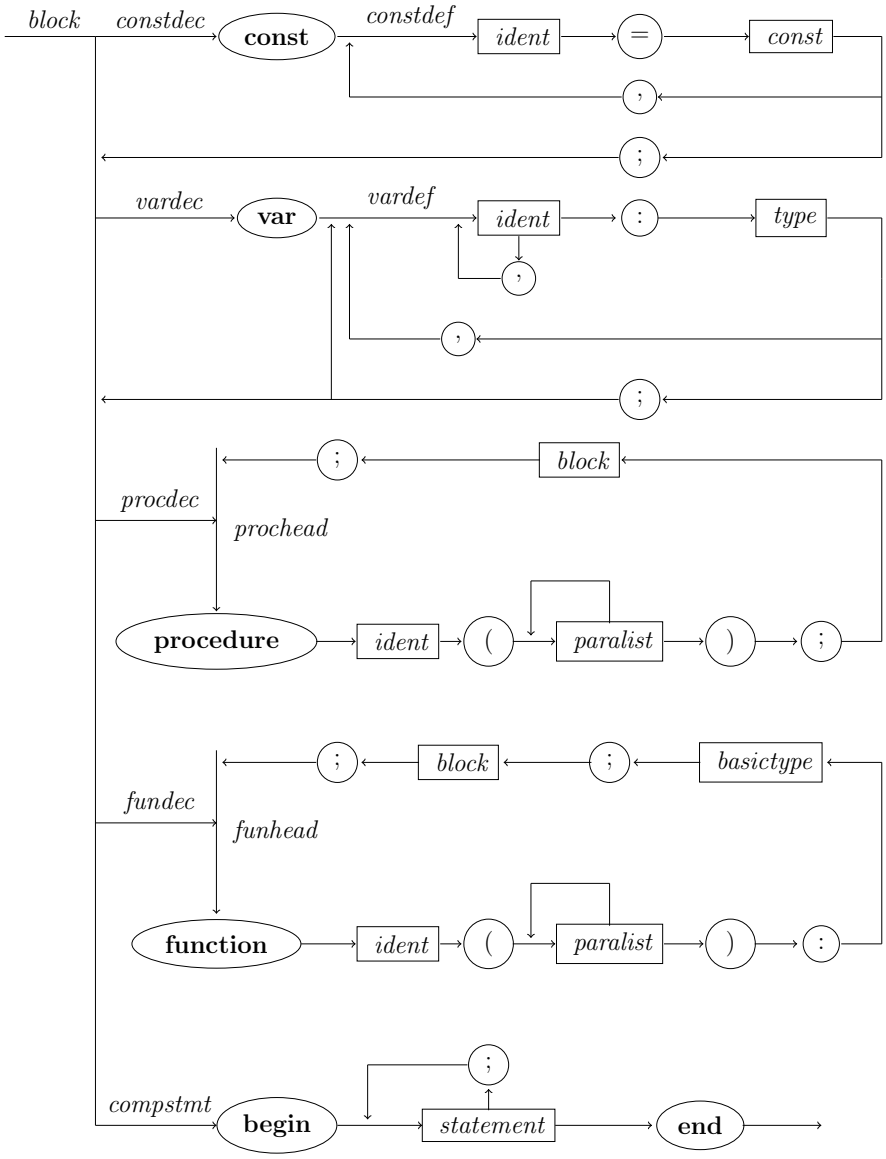

Figure 2: block 语法图

1.4 语句的解读

statement ::= *assignstmt* | *ifstmt* | *repeatstmt* | *pcallstmt*
| *compstmt* | *readstmt* | *writestmt* | *forstmt* | *nullstmt*

assignstmt ::= *ident* := *expression* | *funident* := *expression*
| *ident* '[' *expression* ']' := *expression*

ifstmt ::= **if** *condition* **then** *statement*
|**if** *condition* **then** *statement* **else** *statement*

repeatstmt ::= **repeat** *statement* **until** *condition*

forstmt ::= **for** *ident* := *expression* (**to**|**downto**) *expression* **do** *statement*

pcallstmt ::= *ident* ′(′[*arglist*]′)′

compstmt ::= **begin** *statement* {; *statement* }**end**

readstmt ::= **read**′(′ *ident* {, *ident* }′)′

writestmt ::= **write**′(′ *string* , *expression* ′)′|**write**′(′ *string* ′)′|**write**′(′ *expression* ′)′

这几句定义了语句的文法的完整视图,语句的文法图见Figure 3。 具体实例:

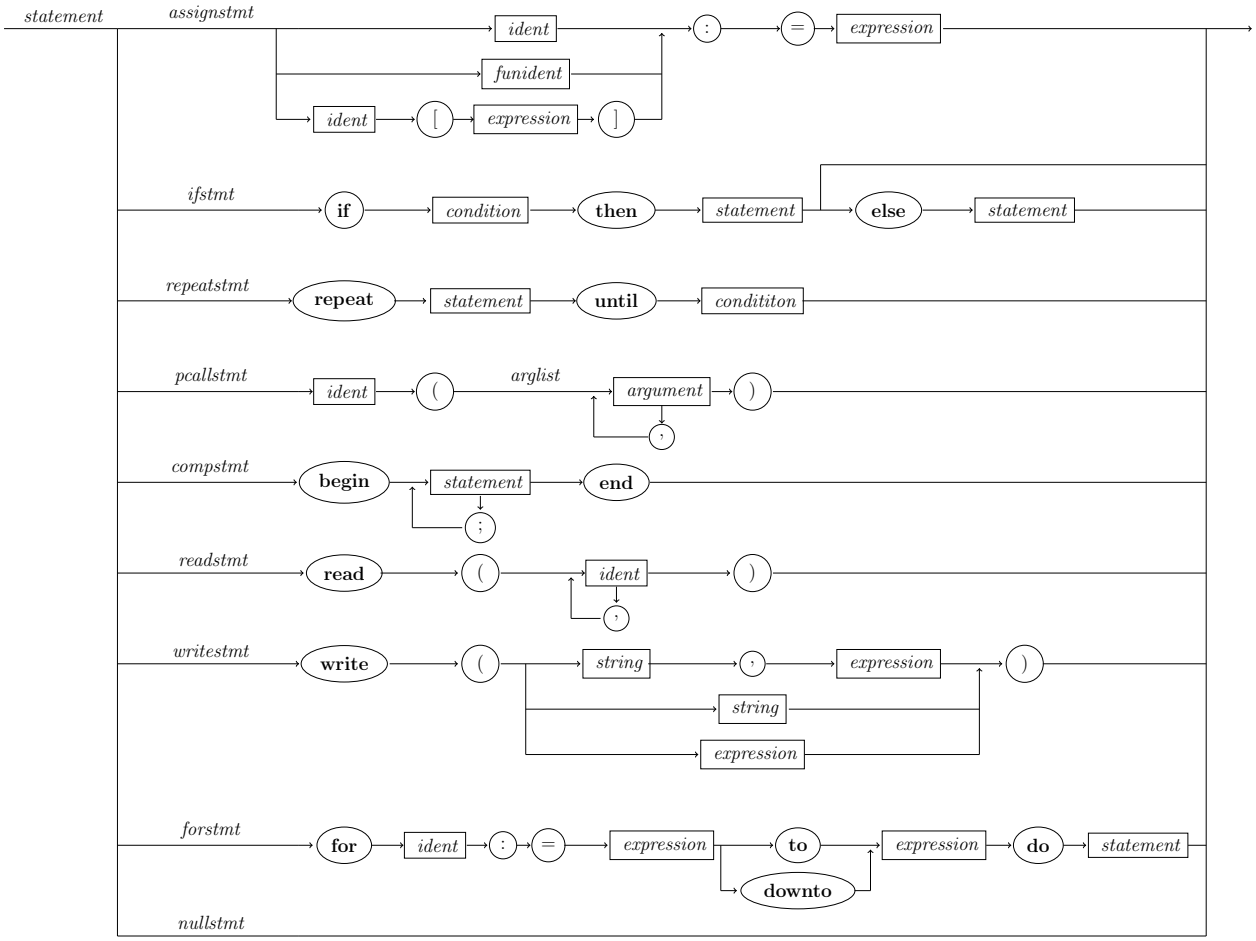

Figure 3: statement 语法图

```
const a=1, b=2, test=11, i=1;
var sum, k: integer; A:array[3] of of integer;
procedure testproc();
  testproc具体实现略;
begin
  // 赋值语句
  sum := a + b;
  A[i] := b;
  // 条件语句
  if a > b then
    sum := a;
  eles
    sum := b;
  // 重复语句
  repeat
    sum := sum + 1;
  until sum > 4
  // for语句 (步长为一)
  for i := 1 to 10 do
    sum := sum + i
  //过程调用语句
  testproc();
  //读语句
  read (k, sum);
  //写语句
  write("hello world!");
  write(sum)
end
```

说明:

1. 赋值语句可以是表达式给变量的赋值，可以是表达式给数组赋值，还可以给表达式给函数给返回值。
2. 条件语句的**else**悬挂的解决方法是总将**else** 和最近的**if**进行匹配。
3. for语句的步长设为一，**to**表示变量值加一，**downto** 表示变量值减一。
4. 读语句，写语句和过程调用语句比较简单。
5. 复合语句被**begin**和**end**围起来，之间是以分号隔开。
6. 语句可以为空，即什么都没有。

1.5 类型的解读

type ::= *basictype* |**array**′[′ *unsign* ′]′**of** *basictype*

basictype ::= **integer**|**char**

const ::= [+|−] *unsign* | *character*

character ::= ′ *letter* ′|′ *digit* ′

string ::= “{*ASCII characters with decimal code number varys from* 32 *to* 126 *exclude* 34}”

unsign ::= *digit* { *digit* }

letter ::= $a|b|c|...|z|A|B|C|...|Z$

digit ::= $0|1|2|3|...|9$

类型的定义比较简单，就不画语法图了。直接说明。

1. 具体的类型分为两类：基本类型和数组。
2. 基本类型包括**integer**和**char**。整型包含正负的整数；char型包含数字位和大小写字母，定义时以单引号隔开。
3. 字符串包含十进制ASCII码值从32到126的所有的值但是得剔除值为34的双引号。这样有利于词法分析的状态机的设计。另外字符串不是一种类型，只用于写语句的打印。
4. 常量的定义可以是正负整数，也可以是字符。若是字符，则会使用ASCII 码值进行运算。

1.6 表达式，项，因子的解读

1.6.1 表达式

expression ::= [+|−] *term* { *addop* *term* }

表达式的语法图见Figure 4。下面是一些表达式的样例:

```
- 3 + a
a + b
1 - 10
a + 1 - 5
```

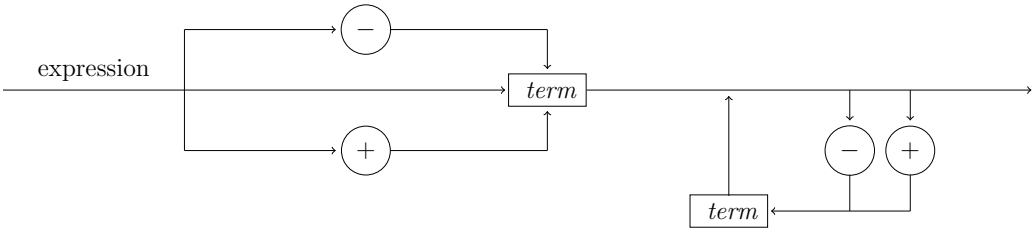

Figure 4: expression 语法图

1.6.2 项

term ::= *factor* { *multop* *factor* }

项的语法图见Figure 5。下面是一些项的样例:

```
a * b
1 / 10
a / 1 * 5
```

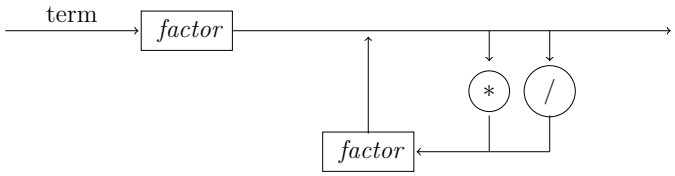

Figure 5: term 语法图

1.6.3 因子

factor ::= *ident* | *ident* '[' *expression* ']' | *unsign* '(' *expression* ')' | *fcallstmt*

fcallstmt ::= *ident* '(' [*arglist*] ')'

因子的语法图见Figure 6。下面是一些因子的样例:

```
const a = 1;
var A:array[3] of integer;
function somefun(): integer;
  somefun程序略;
// 下面的是因子
a
A[2]
(a + b)
somefun()
```

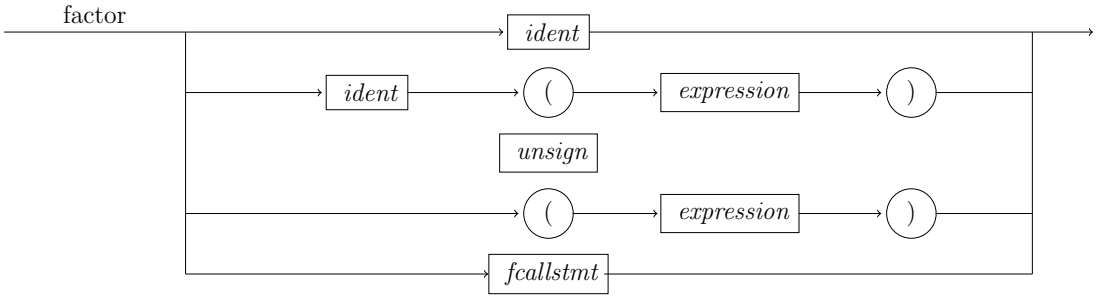

Figure 6: factor 语法图

1.7 其他一些杂项

ident ::= *letter* { *letter* | *digit* }

paralist ::= [**var**] *ident* {, *ident* } : *basictype* {; *paralist* }

funident ::= *ident*

arglist ::= *argument* {, *argument* }

argument ::= *expression*

addop ::= +|−

multop ::= ∗|/

condition ::= *expression relop expression*

relop ::= < | <= | > | >= | = | <>

杂项的说明:

1. 关于标识符的解析可以总结成一个状态机，见Figure 7
2. 形参列表需要给出参数类型。
3. 参数列表是由实参组成，之间使用逗号隔开。
4. 实参被定义成一个表达式。
5. 条件是基于一些比较，零表示假，非零表示真。

1.8 优化方案

DAG图消除公共子表达式。

2 详细设计

2.1 程序结构

程序的文件列表如下:

```
analyse.c analyse.h code.c code.h error.c error.h
global.h main.c nspace.c nspace.h parse.c parse.h
quad.c quad.h scan.c scan.h symtab.c symtab.h
util.c util.h elf.h elf.c
```

`main` 是主函数入口；`scan` 做词法分析；`parse` 做语法分析生成语法树；`analyse` 对语法树打印，并进行一些分析；`code` 语义分析，解析语法树，生成四元式；`symtab` 管理符号表；`nspace` 管理命名空间；`quad` 发射四元式；`elf` 将四元式生成x86汇编；`util` 是一些工具函数。

2.2 函数功能

捡几个比较重要的说。
getToken():词法分析，获取单词
parse():语法分析，生成语法树
analyse():分析语法树
code():生成四元式
elf():生成elf文本格式的x86汇编

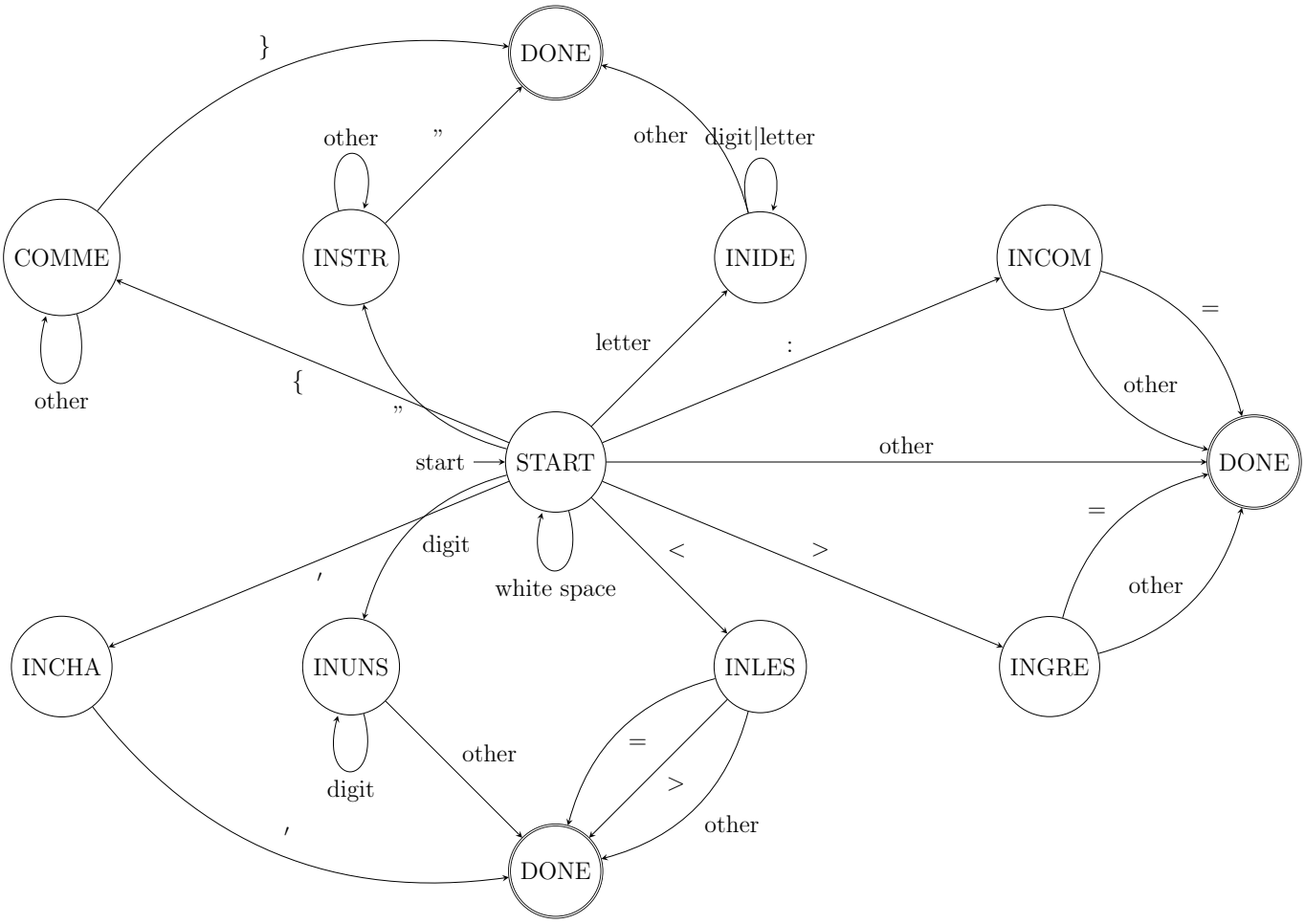

Figure 7: getToken 状态机

2.3 调用关系

main − > parse − > analyse − > code
parse 使用递归下降分析法建立语法树。里面有很多建立节点的函数，这些节点定义根据文法而来，具体如下：

```
static PgmSP PgmB(void);
static BlockSP BlockB(void);
static ConstDecSP ConstDecB(void);
static ConstDefSP ConstDefB(void);
static VarDecSP VarDecB(void);
static VarDefSP VarDefB(void);
static PFDecListSP PFDecListB(void);
static ProcDecSP ProcDecB(void);
...
```

```
static TermSP TermB(void);
static FactorSP FactorB(void);
static CondSP CondB(void);
static IdentSP IdentB(IDREADMODE mode);
static ParaListSP ParaListB(void);
static ParaDefSP ParaDefB(void);
static ArgListSP ArgListB(void);
```

每个节点的数据结构，如下：

```
typedef struct _PgmS *PgmSP;
typedef struct _BlockS *BlockSP;
typedef struct _ConstDecS *ConstDecSP;
typedef struct _ConstDefS *ConstDefSP;
typedef struct _VarDecS *VarDecSP;
typedef struct _VarDefS *VarDefSP;
typedef struct _PFDecListS *PFDecListSP;
typedef struct _ProcDecS *ProcDecSP;
...
typedef struct _FactorS *FactorSP;
typedef struct _CondS *CondSP;
typedef struct _IdentS *IdentSP;
typedef struct _ParaListS *ParaListSP;
typedef struct _ParaDefS *ParaDefSP;
typedef struct _ArgListS *ArgListSP;
/* declaretion of a bundle of node type */
typedef enum {
  Fun_PFDec_t , Proc_PFDec_t
} PFDec_t;
...
typedef enum {
  /* normal identifier type */
  Init_Ident_t, Proc_Ident_t, Int_Fun_Ident_t,
  Char_Fun_Ident_t,
  /* const identifier type */
  Int_Const_Ident_t, Char_Const_Ident_t,
  /* variable identifier type */
  Int_Var_Ident_t, Char_Var_Ident_t,
  IntArr_Var_Ident_t, CharArr_Var_Ident_t,
  /* parameter identifier type */
  /* call by value */
  Int_Para_Val_Ident_t, Char_Para_Val_Ident_t,
  /* call by address */
  Int_Para_Ref_Ident_t, Char_Para_Ref_Ident_t
} Ident_t;
...
typedef enum {
  StrId_Write_t, Str_Write_t, Id_Write_t
} Write_t;
/* declaretion of a bundle of struct */
```

```
/* Program */
typedef struct _PgmS {
  BlockSP bp;
} PgmS;
/* block */
typedef struct _BlockS {
  ConstDecSP cdp;
  VarDecSP vdp;
  PFDecListSP pfdlp;
  CompStmtSP csp;
} BlockS;
typedef struct _ConstDecS {
  ConstDefSP cdp;
  ConstDecSP next;
} ConstDecS;
typedef struct _ConstDefS {
  IdentSP idp;
} ConstDefS;
...
typedef struct _ArgListS {
  ExprSP ep;
  ArgListSP next;
} ArgListS;
```

analyse 做语法分析，code 遍历语法树各个节点，生成四元式。数据结构有点多，就不粘贴了。

2.4 符号表管理

符号表采用栈式符号表，每个函数或过程进入是申请符号表，退出时弹出。每个符号表项使用符号名hash查找，从栈顶往栈底查。数据结构如下：

```
/* hash size */
extern int HASHSIZE;
/* hash shift */
extern int SHIFT;

typedef struct _SymTabS *SymTabSP;
typedef struct _SymLineS *SymLineSP;
typedef struct _SymBucketS *SymBucketSP;
typedef struct _SymTabES *SymTabESP;

typedef enum { 表项类型
  Nop_Obj_t, Const_Obj_t, Var_Obj_t,
  Proc_Obj_t, Fun_Obj_t, Array_Obj_t,
  Para_Obj_t, Tmp_Obj_t
} Obj_t;

typedef enum { 符号类型
  Int_Type_t, Char_Type_t, Nop_Type_t
} Type_t;
```

```
/**
 * symbol table stack
 * is a stack that
 * manage symbol table in
 * each function
 */
typedef struct _SymTabS { 符号表链，使用栈式符号表
  SymBucketSP *sbp;       hash 符号索引表头
  char *ns;               // namespace for a block
  SymTabSP prev;
  SymTabSP next;
} SymTabS;

/**
 * store which line a varible
 * be referenced
 */
typedef struct _SymLineS { 符号的行号记录
  int lineno;
  SymLineSP next;
} SymLineS;

/**
 * symbol table entry
 * bucket list
 */
typedef struct _SymBucketS {
  SymTabESP ep;           // element infomation
  SymBucketSP next;
} SymBucketS;

typedef struct _SymTabES { 符号表项
  char *name;             // identifier name
  char *label;            // namespace label
  int val;                // array length
        // or const value
  SymLineSP lines;        // referenced lines
  Obj_t obj;              // object type
  Type_t type;            // type
  SymTabSP stp;           // point to symbol table
} SymTabES;
```

对应的符号表操作

```
SymTabSP pop(void);
SymTabSP newstab(void);
void push(SymTabSP);
char *mkUsi(int);
SymTabESP sym_insert_const(IdentSP);
SymTabESP sym_insert_var(IdentSP);
```

```
SymTabESP sym_insert_para(IdentSP);
SymTabESP sym_insert_fun(IdentSP, ParaListSP);
SymTabESP sym_insert_proc(IdentSP, ParaListSP);
SymTabESP sym_insert_tmp();
char *genLabel(void);
SymTabESP sym_lookup(char *);
void printTab(SymTabSP);
```

2.5 存储分配

常量编译时候直接嵌入汇编里，无需分配空间；变量动态分配存储空间见下面：运行栈示意图的variables 部分，数组也能分配在这个区域，不同的是数组是分配一段连续的内存空间。临时变量动态分配在变量空间下面。参数压每个函数栈。那些saved ebp 是display 区。

```
/**
 *             RUNTIME STACK OVERVIEW
 *
 *                  high addr
 *               |    ... ...   |
 *               |    ... ...   |
 *               | parameter(1) |
 *               | parameter(2) |       ||
 *               | parameter(3) |       ||
 *               | saved ebp(3) |       ||
 *               | saved ebp(2) |       ||
 *      ebp+8 -> | saved ebp(1) |       ||
 *      ebp+4 -> | return addr  |       || stack
 *        ebp -> | prev ebp     |       || pointer
 *               | return value |       || increase
 *               | variables(1) |       ||
 *               | variables(2) |      \||/
 *               | variables(3) |       \/
 *               | temporary(1) |
 *               | temporary(2) |
 *               | temporary(3) |
 *               | saved ebx    |
 *               | saved esi    |
 *               | saved edi    |
 *               |    ... ...   |
 *        esp -> |    ... ...   |
 *
 **                 low addr
 */
```

2.6 四元式设计

四元式设计如下：

算术指令：

ADD r, s, d	d = r + s
SUB r, s, d	d = r - s
MUL r, s, d	d = r * s
DIV r, s, d	d = r / s
INC -, -, d	d++
DEC -, -, d	d–
NEG r, -, d	d = -r

存取赋值指令:

LOAD r, -, d	load the value of r into d
ASS r, -, d	assign r into d; d:=r
ASSA r, s, d	d := r[s]

条件指令:

EQU r, s, d	brance to label d if (r == s)
NEQ r, s, d	brance to label d if (r != s)
GTT r, s, d	brance to label d if (r > s)
GEQ r, s, d	brance to label d if (r >= s)
LST r, s, d	brance to label d if (r < s)
LEQ r, s, d	brance to label d if (r <= s)

无条件跳转指令:

JMP -, -, d(label)	jump to label

栈操作指令:

PUSH -, -, d	push d into stack
PUSHA -, -, d	push the address of d into stack
POP -, -, d	pop d out of stack

函数调用:

LABEL -, -, d(label)	label for brance
CALL r(funlabel), , d	d = r()
SRET -, -, d	set function return value
ENTER -, -, d(funlabel)	enterance of a function
FIN -, -, -	finish a function

IO指令:

READ -, -, d	read a integer
READC -, -, d	read a char
WRI -, -, d	write a integer
WRC -, -, d	write a char
WRS -, -, d	write a string

2.7 优化方案

2.7.1 DAG图

基本块结构如下:

```
typedef struct _BBS *BBSP;
typedef struct _BBListS *BBListSP;

typedef struct _BBListS {
```

```
  BBSP bbp;
  BBListSP next;
} BBListS;

typedef struct _BBS {
  int id;
  // point to a function or procedure quadruples
  QuadSP qhead;
  QuadSP qtail;
  QuadSP scope;
  QuadSP first; // first position of basic block
  QuadSP last;  // last position of basic block
} BBS;
```

DAG图的优化见`dag.c` 文件。主要是先建立dag图在使用递归式重DAG图推到代码序列。

2.8 错误处理

1. 该标识符未定义
2. 识符重复定义
3. 应是标识符
4. 应是‘(’
5. 应是‘: ’，在说明类型时必须有此冒号
6. 非法符号，编译将跳读该符号和后面的某些符号
7. 形式参数表中，形参说明应该以标识符或var开头
8. 应是of
9. 应是‘(’
10. 类型定义必须以array或基本类型开头
11. 应是‘[’
12. 应是‘] ’
13. 应是‘; ’
14. 函数结果必须是integer或char类型
15. 应是‘= ’，‘:= ’只能在赋值语句里使用，而不能在说明中使用
16. 在if后面必须是条件
17. 在for后面的循环变量只能是integer或char型
18. for语句中初值或终值表达式必须与循环变量类型相同
19. 数太大

20. 程序结尾是‘.’，请检查相应的begin和end
21. 非法字符
22. 在常量定义中，等号后面必须是常数
23. 下标表达式类型必须是integer型
24. 数组说明中，下界小于0，类型不是integer型。
25. 没有这样的数组
26. 该算术表达式的类型不合法。注意，数组整体不能作为算术运算的操作数
27. 实参和对应形参类型应相同
28. 应是变量
29. 实参个数与形参个数不等
30. 表达式中不能出现类型或过程标识符
31. 应是变量或过程/函数标识符
32. 在赋值语句中被赋值变量应与表达式类型相同
33. 数组越界
34. 应是常量
35. 应是‘:=’
36. 应是then
37. 应是do
38. 应是to
39. 应是begin
40. 应是end
41. 因子必须以标识符、常量或‘(’开始

3 操作说明

3.1 运行平台

操作系统：ubuntu 12.04
汇编器：nasm 2.09.10
连接器：gcc 4.6.3
gcc ubuntu 下已经集成，无需安装。nasm 汇编器安装步骤很简单，只需向shell里键入如下指令：

```
$ sudo apt-get install nasm
```

3.2 操作步骤

打开shell后：键入如下指令即可编译运行。

```
$ make                      # make编译生成一个compiler的可执行文件
$ ./compiler filename.xxx   # 编译filename.xxx 生成 filename.asm 的 elf文件
$ nasm -f elf filename.asm # 使用 nasm 汇编 filename.asm 生成 filename.o 文件
$ gcc filename.o            # gcc 连接 filename.o 文件 生成 a.out 可执行文件
$ ./a.out                   # 运行 a.out
```

4 测试报告

4.1 测试程序及测试结果

4.1.1 test1.txt（正确）

冒泡排序

```
 1:  {program bubble sort;}
 2:  var
 3:    a:array[5] of integer {= (4, 5, 2, 7, 0)};
 4:    i, j: integer;
 5:  procedure swap(var x,y:integer);
 6:    var
 7:      t: integer;
 8:    begin
 9:      t := x;
10:      x := y;
11:      y := t
12:    end;
13:  begin { main }
14:    a[0] := 4;
15:    a[1] := 5;
16:    a[2] := 2;
17:    a[3] := 7;
18:    a[4] := 0;
19:    for i := 0 to 4 do
20:    begin
21:      for j := i to 4 do
22:      begin
23:        if a[i] > a[j] then
24:        begin
25:          swap(a[i], a[j])
26:        end;
27:      end;
28:    end;
29:    for i := 0 to 4 do
30:    begin
31:      write(a[i])
32:    end
33:  end.
```

运行结果:

```
0
2
4
5
7
```

4.1.2 test2.txt（正确）

求阶层，同时测试了多层次的display区。

```
 1:  { test for multi-level and display region disign}
 2:  var vn, out: integer;
 3:  function fac(n:integer): integer; {find factorial of n}
 4:  begin
 5:    if n=0 then
 6:    begin
 7:      fac := 1
 8:    end
 9:    else
10:    begin
11:      fac := fac(n-1) * n
12:    end;
13:  end;
14:
15:  function level1(n:integer):integer; {test for mutilevel}
16:    function level2(n:integer):integer;
17:      function level3(n:integer):integer;
18:        begin
19:          level3 := fac(n)
20:        end;
21:      begin
22:        level2 := level3(n)
23:      end;
24:    begin
25:      level1 := level2(n)
26:    end;
27:
28:  begin
29:    for vn := 0 to 4 do
30:      write(level1(vn))
31:  end.
```

运行结果:

```
1
1
2
6
24
```

4.1.3 test3.txt（正确）

快排测试，输入的第一个数是要排序的总数（小于10），接着输入所有要求排序的数字。这里测试的读写。

```
 1:  {qsort test}
 2:  var a: array [10] of integer;
 3:      i,num,temp:integer;
 4:  procedure qsort(l,h:integer);
 5:  var i,j,t,m:integer;
 6:    procedure swap(var i, j:integer);
 7:    var temp:integer;
 8:    begin temp:=i;i:=j;j:=temp end;
 9:
10:  begin
11:    i:=l;
12:    j:=h;
13:    m:=a[(i+j) / 2];
14:    repeat
15:    begin
16:      if a[i]<m then repeat i:=i+1 until a[i]>=m;
17:      if m<a[j] then repeat j:=j-1 until m>=a[j];
18:
19:      if i<=j then
20:      begin
21:      swap(a[i],a[j]);
22:      i:=i+1;
23:      j:=j-1;
24:      end;
25:    end
26:    until i>j;
27:
28:    if j>l then qsort(l,j);
29:    if i<h then qsort(i,h);
30:  end;
31:
32:  begin
33:  write("please input num <10 ");
34:  read(num);
35:  for i:=1 to num do begin
36:  write("please input number> ");read(temp); a[i-1]:=temp end;
37:
38:  qsort(0, num-1);
39:  write("number after sort");
40:  for i:=0 to num-1 do write(a[i]);
41:  end.
```

运行结果:
视具体情况而定，输出排序后的序列（升序）。下面是一种可能的情况。

```
please input num <10
4
```

```
please input number>
-1
please input number>
4
please input number>
0
please input number>
2431
number after sort
-1
0
4
2431
```

4.1.4 text4.txt（正确）

用来测试取余，和条件分支。

```
 1:  var x,y,g,m:integer;
 2:      i:integer;
 3:      a,b:integer;
 4:  procedure swap();
 5:    var temp:integer;
 6:    begin
 7:      temp:=x;
 8:      x:=y;
 9:      y:=temp
10:    end;
11:  function mod(var fArg1,fArg2:integer):integer;
12:    begin
13:      fArg1:=fArg1-fArg1/fArg2*fArg2;
14:      mod:=fArg1
15:    end;
16:  begin
17:    for i:=3 downto 1 do
18:    begin
19:      write("input x: ");
20:      read(x);
21:      write("input y: ");
22:      read(y);
23:
24:      x:=mod(x,y);
25:      write("x mod y = ",x);
26:      write("choice 1 2 3: ");
27:      read(g);
28:      if  g = 1 then
29:        write("good  ")
30:      else if g = 2 then
31:        write("better ")
32:      else if g = 3 then
```

```
33:         write("best ")
34:     end
35:  end.
```

运行结果：下面是某种输入下对应的输出。

```
input x:
4
input y:
4
x mod y =
0
choice 1 2 3:
1
good
input x:
7
input y:
5
x mod y =
2
choice 1 2 3:
2
better
input x:
-3
input y:
2
x mod y =
1
choice 1 2 3:
3
best
```

4.1.5 text5.txt（正确）

GCD 测试

```
 1:  {greatest common divisor , recursive}
 2:  var i,m,n:integer;
 3:  function gcd(i,j:integer):integer;
 4:  begin
 5:  if i=j then gcd:=i;
 6:  if i>j then gcd:=gcd(i-j,j);
 7:  if i<j then gcd:=gcd(i,j-i);
 8:  end;
 9:  begin
10:    for i := 1 to 3 do
11:    begin
12:      read(m, n);
```

```
13:      write(gcd(m,n))
14:    end
15:  end.
```

运行结果:

```
2 34
2
45 3
3
2343 23
1
```

4.1.6 text6.txt（错误）

```
 1:  {program bubble sort;}
 2:  var
 3:    a:array[5] of integer {= (4, 5, 2, 7, 0)};
 4:    i, j: integer;
 5:  procedure swap(var x,y:integer);
 6:    var
 7:      t: integer;
 8:    begin
 9:      t := x;
10:      x := y;
11:      y := t
12:    end;
13:  begin { main }
14:    a[0] = 4;
15:    a[1] = 5;
16:    a[2] = 2;
17:    a[3] = 7;
18:    a[4] = 0;
19:    for i := 0 to 4 do
20:    begin
21:      for j := i to 4 do
22:      begin
23:        if a[i] > a[j] then
24:        begin
25:          swap(a[i], a[j])
26:        end;
27:      end;
28:    end;
29:    for i := 0 to 4 do
30:    begin
31:      write(a[i])
32:    end
33:  end.
```

报错，第14，15，16，17，18行]后缺少:=

```
compiler version 0.9.7 starting ...
syntax error:14: Missing a ':=' after -> ]
syntax error:15: Missing a ':=' after -> ]
syntax error:16: Missing a ':=' after -> ]
syntax error:17: Missing a ':=' after -> ]
syntax error:18: Missing a ':=' after -> ]
```

4.1.7 text7.txt（错误）

```
 1:  { test for multi-level and display region disign}
 2:  var vn, out: integer;
 3:  function fac(n:integer): integer; {find factorial of n}
 4:  begin
 5:    if n=0 then
 6:    begin
 7:      fac := 1
 8:    end
 9:    else
10:    begin
11:      fac := fac(n-1) * n
12:    end;
13:  end;
14:
15:  function level1(n:integer); {test for mutilevel}
16:    function level2(n:integer):integer;
17:      function level3(n:integer):integer;
18:        begin
19:          level3 := fac(n)
20:        end;
21:      begin
22:        level2 := level3(n)
23:      end;
24:    begin
25:      level1 := level2(n)
26:    end;
27:
28:  begin
29:    for vn := 0 to 4 do
30:      write(level1(vn))
31:  end.
```

语法错误: 15行)后面缺少:。函数没有返回值类型。

```
compiler version 0.9.7 starting ...
syntax error:15: Missing a ':' after -> )
syntax error:16: Fatal, Unexpect symbol token -> function
```

4.1.8 text8.txt（错误）

```
 1:  {qsort test}
```

```
 2:  var a: array [10] of integer;
 3:      i, a, num:integer;
 4:  procedure qsort(l,h:integer);
 5:  var i,j,t,m:integer;
 6:    procedure swap(var i, j:integer);
 7:    var temp:integer;
 8:    begin temp:=i;i:=j;j:=temp end;
 9:
10:  begin
11:    i:=l;
12:    j:=h;
13:    m:=a[(i+j) / 2];
14:    repeat
15:    begin
16:      if a[i]<m then repeat i:=i+1 until a[i]>=m;
17:      if m<a[j] then repeat j:=j-1 until m>=a[j];
18:
19:      if i<=j then
20:      begin
21:      swap(a[i],a[j]);
22:      i:=i+1;
23:      j:=j-1;
24:      end;
25:    end
26:    until i>j;
27:
28:    if j>l then qsort(l,j);
29:    if i<h then qsort(i,h);
30:  end;
31:
32:  begin
33:  write("please input num <10 ");
34:  read(num);
35:  for i:=1 to num do begin
36:  write("please input number> ");read(temp); a[i-1]:=temp end;
37:
38:  qsort(0, num-1);
39:  write("number after sort");
40:  for i:=0 to num-1 do write(a[i]);
41:  end.
```

语义错误：第3行a重复定义，36行的tmep未定义就使用。

```
compiler version 0.9.7 starting ...
semantic error:3: Duplicate defined symbol -> a
semantic error:36: First used an undefined symbol -> temp
semantic error:36: First used an undefined symbol -> temp
```

4.1.9 text9.txt（错误）

```
 1:  var x,y,g,m:integer;
```

```
 2:      i:integer;
 3:      a,b:integer;
 4:  procedure swap;
 5:    var temp:integer;
 6:    begin
 7:      temp:=x;
 8:      x:=y;
 9:      y:=temp
10:    end;
11:  function mod(var fArg1,fArg2:integer):integer;
12:    begin
13:      fArg1:=fArg1-fArg1/fArg2*fArg2;
14:      mod:=fArg1
15:    end;
16:  begin
17:    for i:=3 downto 1 do
18:    begin
19:      write("input x: ");
20:      read(x);
21:      write("input y: ");
22:      read(y);
23:
24:      x:=mod(x,y);
25:      write("x mod y = ",x);
26:      write("choice 1 2 3: ");
27:      read(g);
28:      if  g = 1 then
29:        write("good  ")
30:      else if g = 2 then
31:        write("better ")
32:      else if g = 3 then
33:        write("best ")
34:    end
35:  end.
```

swap函数头出现错误。

```
compiler version 0.9.7 starting ...
syntax error:4: Missing a '(' after -> swap
syntax error:6: Missing a identifier after -> ;
syntax error:7: Missing a ':' after -> begin
syntax error:7: Fatal, Unexpect symbol token -> :=
```

4.1.10 text10.txt（错误）

```
 1:  {greatest common divisor , recursive}
 2:  var i,m,n:integer;
 3:  function gcd(i,j:integer):integer;
 4:  begin
 5:  if i=j then gcd:=i;
```

```
 6:  if i>j then gcd:=gcd(i-j,j);
 7:  if i<j then gcd:=gcd(i,j-i);
 8:  end;
 9:  begin
10:    for i := 1 to 3 do
11:    begin
12:      read(m, n);
13:      write(gcd(m,n), m)
14:    end
15:  end.
```

write语句同时写两个表达式，这是错误的。

```
compiler version 0.9.7 starting ...
syntax error:13: Missing a ')' after -> )
syntax error:13: Missing a keyword 'end' after -> ,
syntax error:13: Missing a keyword 'end' after -> m
syntax error:14: Missing a '.' at the end of a program
```

4.2 测试结果分析

4.2.1 test1.txt（正确）

这个程序是冒泡排序。主要测试了函数调用，控制流，表达式以及一些基本语句。对数组的引用。

4.2.2 test2.txt（正确）

这个程序就是一个简单的求阶乘，但是使用了多层嵌套的调用模式，主要是测试display区是否调试正确。

4.2.3 test3.txt（正确）

这是一个快排的程序。除了对函数调用，以及传值和传引用的区别，还测试了数组访问，以及递归运行栈调用的正确性。

4.2.4 test4.txt（正确）

这是一个取余的测试程序，测试了整数除法的运算，以及一些基本表达式的运算。对函数调用以及全局变量的访问，外部变量寻址进行了测试。

4.2.5 test5.txt（正确）

这个程序是求最大公约数的例程，主要是用来测试多重递归问题，同时也测试了控制流的正确性。对读取语句也进行了相关的测试。

4.2.6 text6.txt（错误）

这个程序着重强调了报多个字符缺少的错误。对于文法中的赋值符号:=被勿用成=，错误处理可以找到多处这样的错误。

4.2.7 text7.txt（错误）

这个例程的错误在于函数忘记写返回值类型，这种错误相当与函数头处缺一个冒号，然后后面缺少类型。

4.2.8 text8.txt（错误）

这个程序错误出现与名字的未定义以及名字的重复定义，a在程序中重复定义了，这里报错了。temp变量在程序中未声明就使用了，这属于未定义就使用的错误。

4.2.9 text9.txt（错误）

这里的错误属于违法语法的错误，swap过程定义后没有添加括号。这里就直接指出，但是这样会引起其他的错误。

4.2.10 text10.txt（错误）

这里的错误就是写语句只有三种格式，（字符串，表达式）；（字符串）；（表达式）。而不能像源程序中的错误，写成（表达式，表达式）。

5 总结感想

进行了几周的努力，终于完成了拓展的PL/0编译器的建设。期间遇到的难关都在不断摸索中慢慢变得清晰。下面总结一下自己这几周的工作。

1. 进行了PL/0文法的解读很分析
2. 自己设计和不断修改四元式
3. 8000多行的c代码
4. 自己学习了x86汇编（运行栈太难调了）
5. 熟悉了Linux编程，和使用gcc调试汇编

回想这些日子以来自己不知不觉地已经做了这么多的工作，编译器在自己的工作下一天天的强大，感觉很好。但是自己还是没有时间做太多的优化，首先是自己当时没有组织好数据结构。白白浪费了很多时间重整数据结构，这是比较繁琐的。四元式的设计也是不断迭代才得到的最终版。

学习是循序渐进的过程，我没有奢求一次就完成整个编译器的构建的野心。只有在不断调试之后我才获得更好的实现方式，同时自己的代码能力也是在不断提高。

最后的话，写这个编译器是很值的。