Documents
for
Course *Compilers*
(Chinese)

By Rui.H

Jan.8 2016

一．需求说明 Requirements

1．文法说明 Extended PL/0 grammar

扩充PL/0文法 934

<程序> ::= <分程序>.
<分程序> ::= [<常量说明部分>][<变量说明部分>]{[<过程说明部分>]|[<函数说明部分>]}<复合语句>
<常量说明部分> ::= const<常量定义>{,<常量定义>};
<常量定义> ::= <标识符>＝ <常量>
<常量> ::= [+| -] (<无符号整数>| <无符号实数>)|<字符>
<字符> ::= '<字母>' | '<数字>'
<无符号实数> ::= <无符号整数>.<无符号整数>
<字符串> ::= "{十进制编码为32,33,35-126的ASCII字符}"
<无符号整数> ::= <数字>{<数字>}
<标识符> ::= <字母>{<字母>|<数字>}
<变量说明部分> ::= var <变量说明> ; {<变量说明>;}
<变量说明> ::= <标识符>{, <标识符>} : <类型>
<类型> ::= <基本类型>|array'['<无符号整数>']' of <基本类型>
<基本类型> ::= integer | char | real
<过程说明部分> ::= <过程首部><分程序>{;<过程首部><分程序>};
<函数说明部分> ::= <函数首部><分程序>{;<函数首部><分程序>};
<过程首部> ::= procedure<标识符>[<形式参数表>];
<函数首部> ::= function <标识符>[<形式参数表>]: <基本类型>;
<形式参数表> ::= '('<形式参数段>{; <形式参数段>}')'
<形式参数段> ::= [var]<标识符>{, <标识符>}: <基本类型>
<语句> ::= <赋值语句>|<条件语句>|<当循环语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for循环语句>|<空>
<赋值语句> ::= <标识符> := <表达式>| <函数标识符> := <表达式> | <标识符>'['<表达式>']':= <表达式>
<函数标识符> ::= <标识符>
<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>|<标识符>'['<表达式>']'|<无符号整数>| <无符号实数>| '('<表达式>')' | <函数调用语句>
<函数调用语句> ::= <标识符>[<实在参数表>]
<实在参数表> ::= '(' <实在参数> {, <实在参数>}')'
<实在参数> ::= <表达式>
<加法运算符> ::= +|-
<乘法运算符> ::= *|/
<条件> ::= <表达式><关系运算符><表达式>
<关系运算符> ::= <|<=|>|>= |=|<>
<条件语句> ::= if<条件>then<语句> | if<条件>then<语句>else<语句>

<当循环语句>　::=　while<条件>do<语句>
<for循环语句>　::=　for <标识符>　:= <表达式>　（downto | to）　<表达式>　do <语句>　//步长为1
<过程调用语句>　::=　<标识符>[<实在参数表>]
<复合语句>　::=　begin<语句>{; <语句>}end
<读语句>　::=　read'('<标识符>{,<标识符>}')'
<写语句>　::=　write '(' <字符串>,<表达式> ')'|write'(' <字符串> ')'|write'('<表达式>')'
<字母>　::=　a|b|c|d…x|y|z|A|B|…Z
<数字>　::=　0|1|2|3…8|9

1. 附加说明：

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算
（2）标识符区分大小写字母
（3）赋值语句中<函数标识符> := <表达式> 作为函数的返回值，其类型应与返回类型一致，此语句后面的语句可继续执行
（4）写语句中的字符串原样输出，表达式只有单个字符类型的变量或常量按字符输出，其他表达式均按整型或实型输出
（5）数组的下标从0开始
（6）带var的形式参数为变量形参，实参与该类形参传递数据时是传地址

2. Note:

经验证该获取的文法符合使用递归下降子程序法分析的条件。

没有对文法进一步改写/扩充。

2．目标代码说明 Object Code Specification

生成的目标代码以 P-code 形式给出。共有 59 条 P-code 指令。指令结构为

```
struct order { int f; int x; double y;}
```

其中 f 为操作码，x、y 分别为相关的域，可以代表变量的分程序层次，相对地址，运算操作数等信息。

目标代码指令列表及含义如下：

表 15.14　Pascal-S 指令代码

助记符	f	x	y	功　能
LDA	0	x	y	把变量地址装入栈顶(x 为层次,y 为变量相对地址)
LOD	1	x	y	装入值
LDI	2	x	y	间接装入
DIS	3	x	y	更新 display
FCT	8		y	标准函数(0..18,由 y 给定)
INT	9		y	将栈顶元素加上 y
JMP	10		y	无条件转移到 y
JPC	11		y	如栈顶内容为假,转移到 y
SWT	12		y	转移到 y,查找情况表
CAS	13		y	情况表的登记项——伪指令,不能执行
F1U	14		y	增量步长型 for 循环体的入口测试
F2U	15		y	增量步长型 for 循环体的再入口测试
F1D	16		y	减量步长型 for 循环体的入口测试
F2D	17		y	减量步长型 for 循环体的再入口测试
MKS	18		y	标记栈
CAL	19		y	调用用户过程或函数
IDX	20		y	取下标变量地址(元素长度=1)
IXX	21		y	取下标变量地址
LDB	22		y	装入块
CPB	23		y	复制块
LDC	24		y	装入字面常量
LDR	25		y	装入实数
FLT	26		y	转换浮点数
RED	27		y	读(y 表示类型,1：整型,2：实型,3：字符型)
WRS	28		y	写字符
WRW	29		y	写——隐含域宽
WRU	30		y	写——给定域宽
HLT	31			停止
EXP	32			退出过程
EXF	33			退出函数
LDT	34			取栈顶单元内容为地址的单元内容
NOT	35			逻辑非
MUS	36			求负
WRR	37			写实数——给定域宽
STO	38			将栈顶内容存入以栈顶次高元为地址的单元
EQR	39			实型等于比较
NER	40			实型不等比较
LSR	41			实型小于比较
LER	42			实型小于等于比较
GTR	43			实型大于比较
GER	44			实型大于等于比较
EQL	45			整型相等比较
NEQ	46			整型不等比较

LSS	47			整型小于比较
LER	48			整型小于等于比较
GRT	48			整型大于比较
GEQ	50			整型大于等于比较
ORR	51			逻辑或
ADD	52			整型加
SUB	53			整型减
ADR	54			实型加
SUR	55			实型减
助记符	f	x	y	功　　能
AND	56			逻辑与
MUL	57			整型乘
DIV	58			整型除
MOD	59			取模(求余)
MUR	60			实型乘
DIR	61			实型除
RDL	62			readln(读完一行)
WRL	63			writeln(换行写)

二．详细设计 Detailed Design

1．程序结构 Program Structure

整个程序总体上分为词法的分析，语法的分析，其中语法的分析包括了处理各种语句，并且插入语义分析片段。

```
Lexer lexer(in);
Parser parser(lexer);
parser.program();
printTablesForDebug();
Interpret inter(lc);
inter.run();
return 0;
```

语法和词法的分析都需要对符号表进行管理。

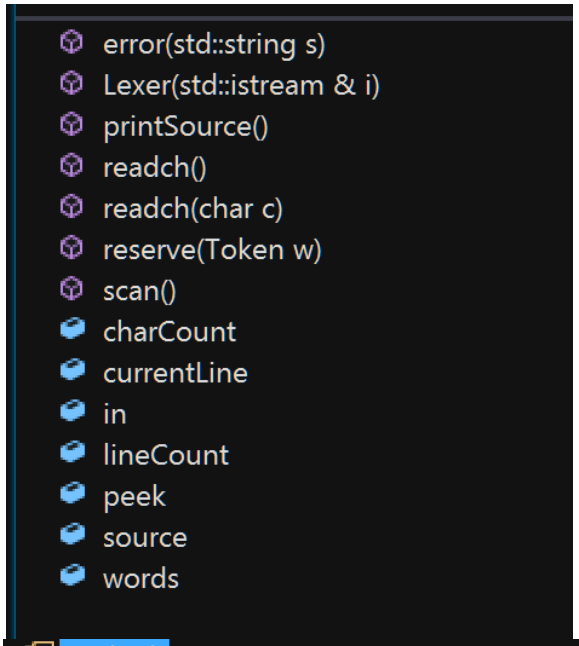
最后由解释执行程序对中间代码进行执行。

整体框架如上图所示，词法分析器接收源程序输入流in，语法分析程序从词法分析器Lexer中不断获取下一单词，在识别到相应语法成分时进行中间代码的生成，整个翻译过程是语法制导翻译技

术。然后由解释程序对生成的中间代码lc序列进行执行。

2．类/方法/函数功能 Classes and Methods

a) Lexer 类是词法分析器的抽象，其方法和成员变量图如下：



其中 peek 是当前所取的字符。

Readch()用于读取下一个字符到 peek 中。

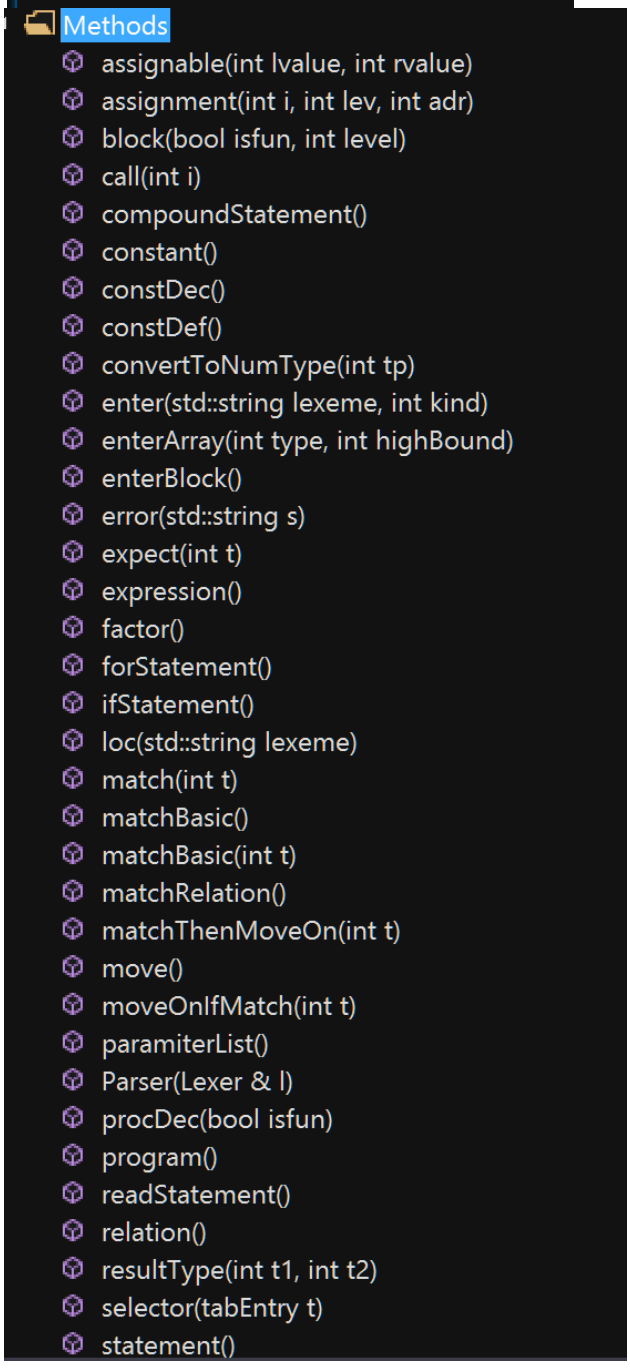
Reserve（）用于添加保留关键字 Token。

Scan（）调用 readch()，跳过空白符，并且组合出下一个 Token 的类别和属性值并返回。

b) Token类是所有的单词的抽象，包括以下三个属性：

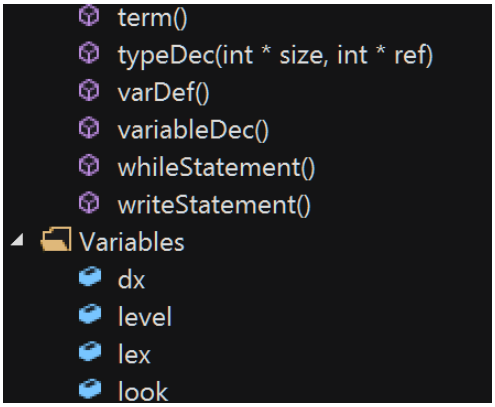
```
int tag;  用于标识单词种类
float value; 用于表示常量值等
std::string lexeme;  用于标识符的字符串表示
```

c) Parser类是对语法分析器的抽象，其方法如下：



属性值含义：

```
Lexer& lex;词法分析器引用
Token look;当前取得的单词
int  dx,  level;  //state  for  current block  !!!当前分程序的变量地址分配指针和层次信息。
```



其中较为关键的有以下函数：

A. Block()用于处理分程序，布尔参数isfun代表当前分程序是函数还是过程，level代表当前层次。

值得注意的是由于文法中分程序嵌套递归定义，在每次进入block分析时要注意对当前调用层的分程序的变量地址分配指针和层次信息进行保存，在分析完当前分程序即将退出时，需要再次对调用层的上述信息进行恢复。

关键代码如下：

保存部分：

```
int dx_bak = this->dx, level_bak = this->level;//remember to backup
before you override and to restore before you go.
this->dx = 5; // data allocation index
this->level = level;
```

恢复部分：

```
this->dx = dx_bak;
this->level = level_bak;//remember to backup before you override and to
restore before you go.
```

B. Assignment()函数主要用于对赋值语句的语法分析和语义分析和中间代码生成。

赋值时重要的一步就是对赋值号两边的表达式类型进行检查。

分为以下几种情况处理，首先判断是否两侧类型相等，若不相等再判断是否进行符合向上转换的规则，将右边的表达式类型向上（无损）转换为左边的。如果不能做到向上转换，则提示相应错误。

算法实现如下：

```
if (typ_fir == typ_sec) {
    if (matchBasic(typ_fir))//identical basic assignment
        emit(38);
    else
        error("non basic type assignment encountered!");
}
else if (typ_fir == Symbol::unreal && matchBasic(typ_sec)) { //cast
rvalue(any basic) to real lvalue.
    emit(38);
}else if(typ_fir == Symbol::unint && typ_sec ==
Symbol::literal){//TODO: maybe constant is allowed??
    emit(38);
}else {
    // NO We don't do other type cast.
    error("assignment type conflict!");
```

```
        emit(38);//suppose we do
            }
```

C. ifStatement(),WhileStatement(),forStatement()对相应语法结构的处理有类似之处。它们都要求对中间代码生成时先产生跳转的代码位置标号，等到对循环体/执行体进行语法解析和中间代码生成之后，才能确定跳出相应结构的出口处代码地址，这是才能返填回相应的位置。

以If语句的处理为例，当带有else块时，lc1就是当if判断的条件为假时应该跳转到代码位置，lc2是else部分语句执行完后的代码位置。

我们需要先记录下lc1和lc2指向的代码位置，等到处理完statement（）语句体时，再将当前代码位置返填到lc1和lc2处跳转代码的跳转位置y域中。

实现片段如下：

```
int lc1 = lc;
        emit(11); //jumpc, we'll come back for the y later
        moveOnIfMatch(Symbol::then);
        statement();
        if (matchThenMoveOn(Symbol::elsesym)) {
            int lc2 = lc;
            emit(10); //immediate jump to y
            code[lc1].y = lc;
            statement();
            code[lc2].y = lc;
        }else
            code[lc1].y = lc;
```

D. expression(),term()用于识别和处理表达式和项，本身文法产生式具有左递归的性质，经过处理后的扩充 BNF 范式表达包含了对某语法成分的零到多次的循环性。这种特性和常量说明部分（conDec()）和变量声明部分（varDec()）有相似特征。

下面以表达式的处理为例：

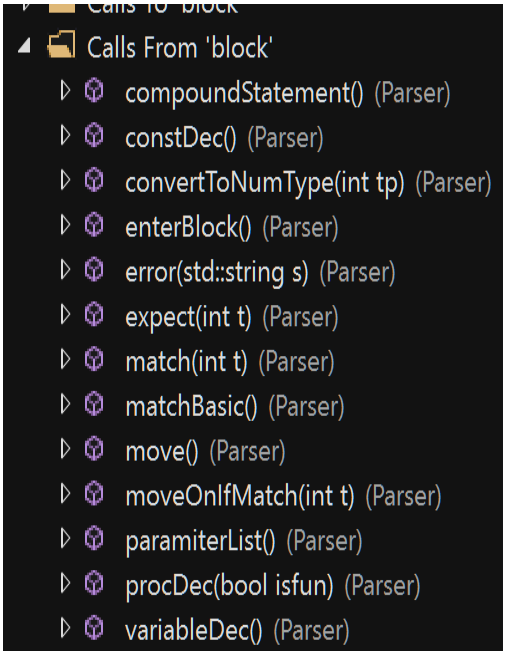
当当前读到的是加号或者减号，就考虑调用term（）识别一次语法成分<项>。之后再次判断是否还有更多的项需要识别，知道下一个符号不再是加号/减号。

```
while (match('+') || match('-')) {
    op = look.tag;
    move();
    typ_sec = term();
    typ_fir = resultType(typ_fir, typ_sec);
    switch (typ_fir){
    case Symbol::literal:
    case Symbol::unint:
        if (op == '+') emit(52);
        else emit(53);
        break;
    case Symbol::unreal:
        if (op == '+') emit(54);
```

```
            else emit(55);
        default:
            break;
        }
}
```

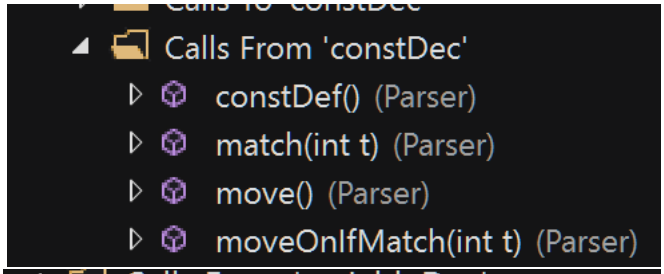
3. 调用依赖关系 Call Dependencies

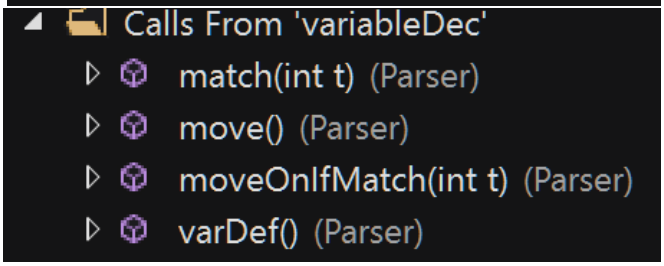
a) Parser 类中处理<分程序>语法成分的 block 函数调用的函数如右图：



按照<分程序> ::= [<常量说明部分>][<变量说明部分>]{[<过程说明部分>]| [<函数说明部分>]}<复合语句>的产生式，在block中先调用paramiterList处理参数表部分，然后调用constDec处理常量说明部分，再调用variableDec处理变量说明，然后再循环调用procDec处理嵌套的过程/函数说明，最后调用compoundStatement处理复合语句即可完成当前分程序的语法识别处理。

b) 常量/变量声明部分如右图所示，它将循环调用constDef/varDef来处理单独的一个常量/变量声明。





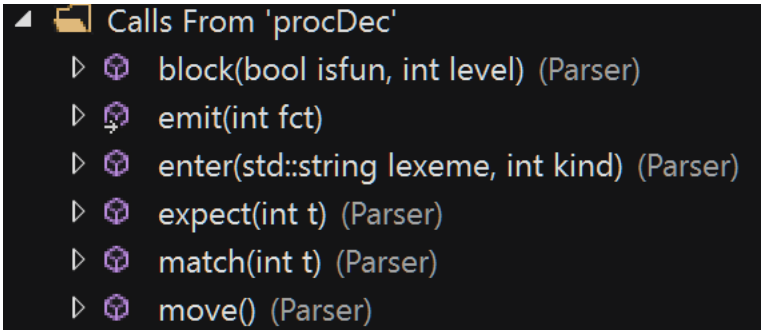
关键部分是变量声明时需要考虑多个同类型变量的声明处理中，类型是最后才能获得的，识别到之后，需要返填回之前填入的那些符号表中的元素中的类型域。

实现部分如下，typ是从当前读到的类型符号，t0-t1范围的符号表项是当前同类型的变量在符号表中的位置，需要逐一填入类型和分配的地址空间。

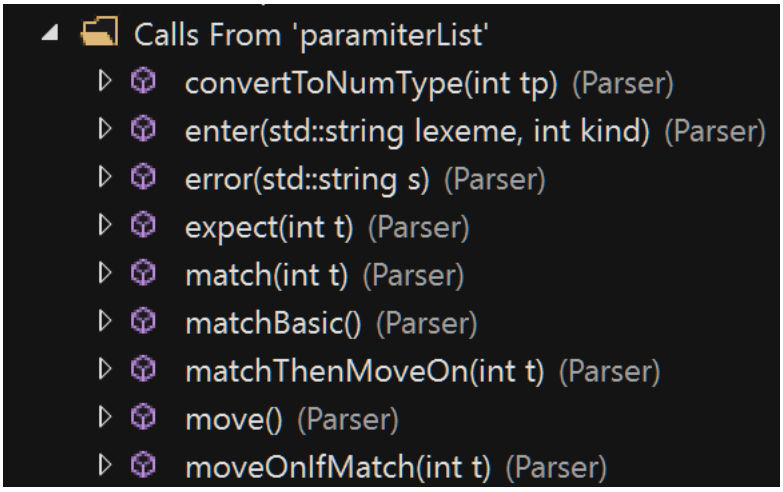
```
typ = convertToNumType(look.tag);
        while (t0 < t1) {
            t0 += 1;
```

```
        tab[t0].typ = typ;
        tab[t0].ref = 0;
        tab[t0].adr = dx;
        tab[t0].lev = level;
        tab[t0].varParam = varParam; //KEY flag for var params.
        dx += 1;//afterall, basic type all have size one.
            }
```

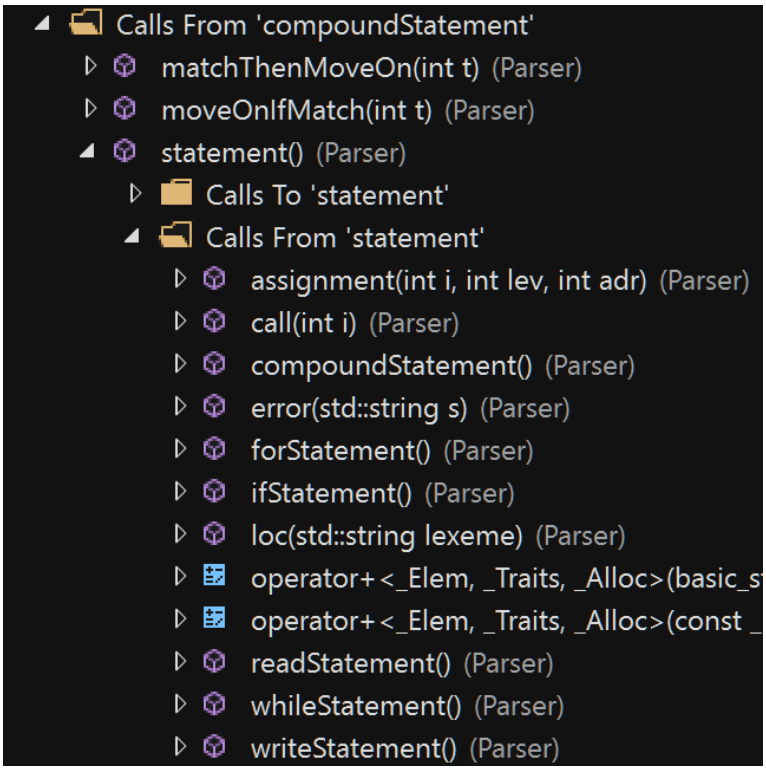
c) 分程序声明部分由procDec函数负责，它将读取过程/函数的标识符名字，然后使用enter填入符号表。之后调用block来处理后续的可能的形式参数表和返回类型。注意此处调用block时的两个参数分别指示即将识别的是过程/函数，以及其层次，应该是当前层加一。



d) 形式参数表的调用函数如下所示。该部分和变量声明的处理有类似之处，需要返填符号表，需要注意识别变量是否是传地址方式引用。



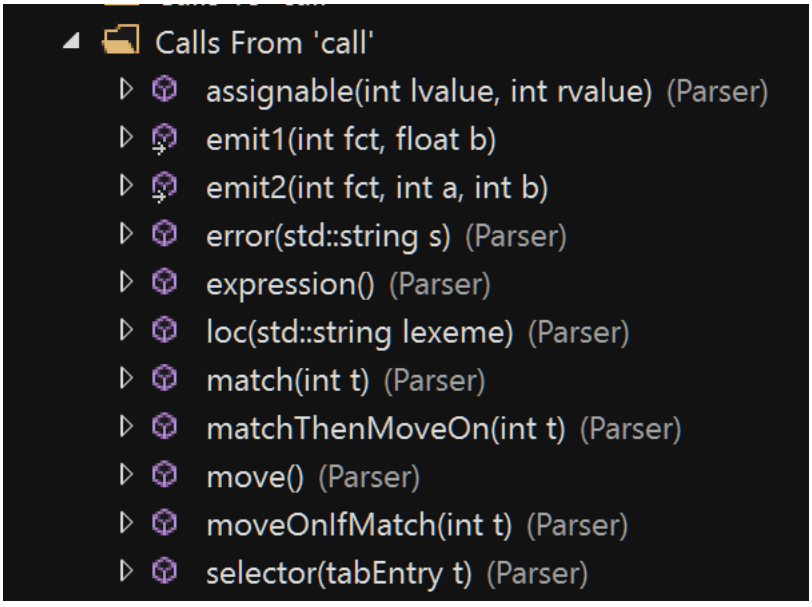
e) Statement 用于各种语句的开始符号识别和相应子程序的调用，它调用的函数如下所示：



这是一个汇聚型的函数，它负责判断当前语句的开始符号，如果是一个关键字，那么转入相应的 if/read/write/while/for 结构的处理函数中，如果是一个标识符那么进一步判断是赋值语句还是函数/过程调用语句。

f) Call 用于对函数/过程调用语句的语法识别和语义检查。

它调用的函数如下所示：



它将会从符号表中找到相应分程序的项，然后按照符号表中声明的形式参数一一对应地匹配后续读到的实际参数。

实际参数可能是一个表达式，那么我们要使用 expression（）对其求值，然后根据表达式类型通过 assignable 检测是否符合形式参数类型，或者是否符合向上转换的规则。

最后在匹配完所有实际参数之后，需要检查形式参数表中的参数是否全都匹配完了，否则报错。

- 4．符号表管理方案 Symbol Table

符号表tab中每个登记项由8个域组成：

```
struct tabEntry {
    std::string name; 标识符名字
    int link; //previous id location within the same subblock.
指向同一分程序中上一个标识符在tab表中的位置。
  int obj; //kind标识符的种类。
  int typ; //type标识符的类型。
  int ref; //ptr to atab or btab标识符为数组时，指向数组信息表中
的位置。标识符为过程名或者函数名时指向分程序表中的位置。
  int lev;//static level for nested subprocedures. 标识符所在的
分程序的静态层次。
  bool varParam;指示是否为传地址的参数。
  float adr; //location in stack for vars, code ptr for
proc/func, value for all kinds of constant
  对于常量名，填入相应的值。对于变量，这里填入运行栈中的存储单元
相对地址。对于过程名函数名，填入目标代码的入口地址。
   };
```

符号表分为四部分，主表 tab，数组表 atab，分程序表 btab 和编译时使用的记录当前编译分程序的外层们的静态链表 dtab。

符号表们通过三个 enter 函数登录信息，通过 loc 函数查找标识符在符号表中的位置。

void enter(std::string lexeme, int kind)用于对普通标识符的登记，参数为标识符名字和种类。

实现部分如下，关键部分在于首先要根据当前的分程序作用域的last指针从后往前找，看是否有redefine的问题。如果没有，填完相应表项之后要记得更新当前分程序层次的最后一个符号表项指针last。

```
    if (t == TMAX)
            error("table full!!");
        int j, l;
        tab[0].name = lexeme;
        l = j = btab[dtab[level]].last;
        while (tab[j].name != lexeme) {
            j = tab[j].link;
        }
        if (j != 0) error("symbol" + lexeme + "redefined!");
        else {
            t += 1;
            tab[t].name = lexeme;
            tab[t].link = l;
            tab[t].obj = kind;
            tab[t].typ = Symbol::nul;
            tab[t].ref = 0;
            tab[t].lev = level;
            tab[t].adr = 0;
            tab[t].varParam = false; //inital value, which may be changed after
by parameterList();
            btab[dtab[level]].last = t;
            }
```

`int enterArray(int type, int highBound)`和 `void enterBlock()`分别用于登录数组和分程序标识符到符号表中，因为这两类符号需要参照 `atab`，`dtab`。

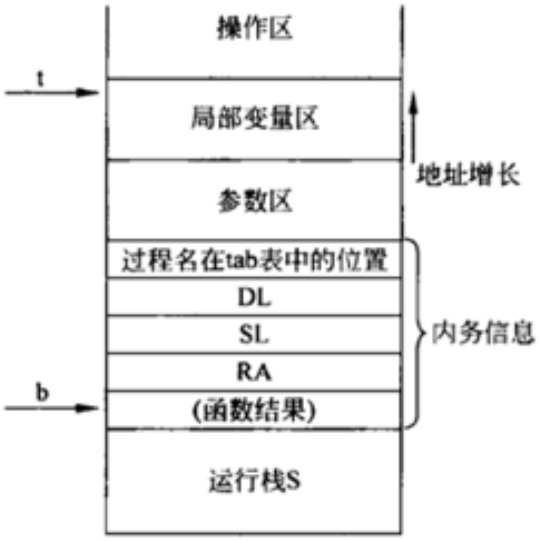
其中标识符查找函数 `loc` 接收查找标识符名字，然后从当前层的最后一个标识符开始向上查找，如果没找到，那么根据 `dtab` 向上一层查找，直到最外层如还没找到就报错。

实现部分如下，`i` 为当前层次，`j` 为相对地址：

```
int loc(std::string lexeme)
    {
        int i = level, j;
        tab[0].name = lexeme;
        do {
            j = btab[dtab[i]].last;
            while (tab[j].name != lexeme){
                j = tab[j].link;
```

```
        }
        i -= 1;
    } while (i >= 0 && j == 0);
    if (j == 0)
        error("id:" + lexeme + "not found!");
    return j;
    }
```

5．存储分配方案 Stack Allocation



运行时数据栈即运行栈，这个栈用来存储每个分程序的数据区。

每当目标代码调用一个过程或函数时（也就是著名的第十八号**mark stack**指令），运行栈会把栈顶指针加5，留出内务信息区，然后把本分程序在栈中将要占据的数据区在上面留出空闲空间。

同时还有一个display表，按静态层次存放了外层各分程序在运行栈中分配的数据区的基地址。便于在内层分程序执行时快速查找到外层分程序中说明的变量，避免了每次查询外层变量在栈上根据静态链反复跳转查找。

6. 解释执行程序 Interpreter

1. 数据结构：

解释执行程序抽象为 Interpret 类，包含以下数据结构和初始化过程：

```
class Interpret {
```

```
public:
    order ib;//instruction buffer
    int stop,
        pc,
        tp, //top stack index
        bs;//base index
    Interpret(int ter):stop(ter)
    {
        st[1] = 0;//return addr
        st[2] = 0;//static link
        st[3] = -1; //dynamic link
        st[4] = btab[1].last;主程序在符号表中的索引号
        display[0] = display[1] = 0;
        tp = btab[2].vsize - 1;主程序的临时空间
        bs = 0;
        pc = tab[static_cast<int>(st[4])].adr; 从符号表中取出主程序开始地址
    }
```

2. 执行过程：

经过上述的初始化之后，开始循环调用execNextInstruc（）执行下一条指令，如果出错则调用void errorInter(const char * str)打印当前错误代码及其位置和出错信息，不出错则执行直到停止位置。

3. 算法实现：

其中较为关键的指令实现算法有以下几个：

A. DIS 指令用于更新 display 表。该表按静态层次存放了各分程序在运行栈中分配的数据区的基地址。当被调用过程是外层过程，当它返回时，需要这条指令根据运行栈中各数据区的静态链重新填原来 display 表中的内容。具体算法如下，就是不断根据 SL 链（在 st[b+2]处）填写 display 表顶端，直到内外层次差为 0 时停止。

```
case 3:
```

```
        {int high = ib.y,
            low = ib.x,
            tem = bs;
        do {
            display[high--] = tem;
            tem = st[tem + 2];
        } while (high > low);
        break;
            }
```

B. 标记栈指令是用于准备调用过程/函数的。当前指令的 y 域为调用过程在符号表中的位置。这条指令将在运行栈上分配 5 个空间，栈顶为新过程在符号表中的位置，次栈顶为该分程序需要的数据区空间减一的值。

```
case 18: //Mark stack
        {
            int size = btab[tab[(int)ib.y].ref].vsize;
            if (tp + size > SMAX) {
                errorInter("RUNTIME:stack full");
                return 1;
            }
            tp += 5;
            st[tp - 1] = size - 1;
            st[tp] = ib.y; //new tab index
            break;
        }
```

C. 当实参已经全部搁置在栈顶之后，我们执行调用 call 指令。

该指令的 y 域是(float) btab[tab[i].ref].psize - 1)，也就是参数和内务区所占据的空间大小减一。

该指令的实现过程是先找到将调用过程的基地址，然后取得标记栈指令放置的符号表位置，然后取得当前层次，并根据层次将其基地址填入 display 表中。然后把内务信息（返回地址，静态动态链）全部填写上。最后初始化数据区中的临时变量区域，然后设置基指针 bs，栈顶 tp，和 pc，即可跳转。

```
case 19: //call
{
    int newBase = tp - ib.y;
    int newTabIndex = st[newBase + 4];
    int currentLev = tab[newTabIndex].lev;
    display[currentLev + 1] = newBase;
    int newTop = st[newBase + 3] + newBase; // vsize - 1 + newBase
    st[newBase + 1] = pc;//return addr
    st[newBase + 2] = display[currentLev]; //SL
    st[newBase + 3] = bs; //DL
    for (int j = tp + 1; j <= newTop; j++)
        st[j] = 0;
    bs = newBase;
    tp = newTop;
    pc = tab[newTabIndex].adr;
    break;
    }
```

D. index 指令用于计算数组元素地址。

该指令 y 域为 atab 的指针。栈顶为数组下标值，次栈顶为数组元素首地址。

首先我们检查栈顶的下标值是否小于零，然后根据 atab 指针找到数组上界，如果越界，提示错误。如果没有问题，那么我们丢弃栈顶下标值，然后从次栈顶（初始地址）加上下标值（相对地址），得到数组元素的地址。

实现过程如下：

```
case 20://load array elem adr
    {
        int patab = ib.y;
        int index = st[tp];
        if (index < 0) {
            errorInter("RUNTIME:array index below 0 !");
            return 1;
        }
        if (atab[patab].bound <= index) {
            errorInter("RUNTIME:array subscript out of bound");
            return 1; //out of bound
```

```
        }
        tp--; //discard the index
        st[tp] += index;
        break;
    }
```

10. 出错处理 Error Handling

A. 出错处理方案：

错误来源主要有三个方面，词法分析程序（语法错误）和语法分析程序（包括语法错误，语义错误）和解释执行程序（运行时错误）。

对词法分析程序我们主要检测源程序是否由正常字符组成，目前只接受常规可打印 ASCII 码，其他超出范围的字符将触发词法错误，导致编译停止。

词法分析程序还处理一些其他构成 Token 时遇到的错误，例如对实型数和字符串的组成中的识别冲突。

语法分析程序处理的是在 Token 序列的组合中可能发生的语法错误，例如缺少关键字，以及语法解析之后显现出的语义错误，例如类型不一致，各类符号表的溢出等。

对于语法语义错误，尽可能恢复丢失语法成分或者跳过部分原程序后继续，对于语义错误将产生警告提示，但是尽量保证继续编译或者继续生成中间代码。

解释执行程序主要是检测运行时错误，例如数组越界或者运行栈溢出。对于运行时错误，处理方案是打印出错信息停止执行。

B. 错误信息和含义：

错误处理程序主要检查的有：

1. 词法分析时的语法错误：

超出范围ASCII码错误， "the char in the file is wrong.

实型数组成错误，"at least one digit expected after period'.'!

字符常量识别错误，"literal expected!! found--> " + peek.

字符常量终结错误，"literal terminator ''' expected! found --> ") + peek。

字符串终结错误，string terminator expected before any char outside ascii 32-126!"

2. 语法分析器的语法/语义错误：

符号表溢出。"table full!!"

标识符重复定义。"symbol" + lexeme + "redefined!"

标识符未找到。"id:" + lexeme + "not found!"

赋值类型不一致并且不可向上转换。"assignment type conflict!"

字符常量和无符号数常量混乱，"you have sign bit already, no char allowed!".

缺少常量 "constant expected!"。

数组界限声明有问题。"array bound too high"/ "array bound less or equal 0"

非基本类型数组，例如嵌套数组。"array of basic type expected!"

缺少类型。"type expected!"

缺少相应的关键字，例如分号，赋值号，逗号等。

语句开始符号错误。"Wrong start id " + tab[i].name + " for a statement!"

缺少逻辑运算符号。"No comparision operator found."

逻辑运算两侧表达式类型不一致。"warnning: inconsistant type in relation!"

数组元素下标不是整数。"subindex type not unsighed int!"

对非数组取下标。"not a array when trying to subaddress!"

for循环使用常量作为循环变量。"should be a variable."

调用函数/过程时参数太多。"Too many param."

不是地址实参。"we need a variable for var param..."

实际参数太少。"too few actual params."

运行时：

运行栈是否溢出。"RUNTIME:stack full"

乘除法时除数是否为0。"RUNTIME:divided by 0"

数组的下表表达式值是否越界。"RUNTIME:array subscript out of bound"/ "RUNTIME:array index below 0 !"

遇到无法识别的指令。>>>>Note:正常情况不能产生该错误，除非编译器实现有错误。<<<<"RUNTIME:instruction error or not implemented"

三．操作说明 Operative Specification

1．运行环境

IDE 环境：

Blend for Microsoft Visual Studio Enterprise 2015
Version 14.0.24720.00 Update 1

套件中的 Microsoft Visual C++ 2015

OS 环境：

Windows 10 Pro

硬件环境：

CPU i7-4850HQ 2.3GHz

RAM 16GB

2．操作步骤

将所有工程源代码文件编译并链接 C++2011 标准库后即可编译器生成可执行文件。

点击执行，在控制台窗口输入源代码文件路径，即可读取该文件，开始编译过程，并显示语法解析完成信息。

然后，该目标程序将自动执行，并将通过控制台的标准输入输出流进行目标程序的输入输出操作。

四．测试报告 Test Report

1．测试程序及测试结果

应考核要求,测试程序放在单独文件夹中,编号 test1-5[error]。

测试结果（以下分为五组，每组分别包含正确程序和错误程序)：

1. 输入 1： 23 12

输出 1：please write number 1 and number 2---> 23 12

The greatest common divider of 23 and 12 is : 1 .

Their lowest common multiple is: 276. ->Program

complete. bye!

输入 2：23 0

输出 2：please write number 1 and number 2---> 23 0

XXXXXXXXXXXXXXXXXXXX The num 2 is not positive.

Program halted! bye!

错误程序编译结果：

line 62 pos 35 type dismatch!!

```
L60->          begin
L61->              ra[0]:=ia[1];
L62->              result[0] := gcd(ra[0], ia[2]);
                                        ^(approximately)
line 63 pos 32 we need a variable for var param...
L61->              ra[0]:=ia[1];
L62->              result[0] := gcd(ra[0], ia[2]);
L63->              lcm(num1, num2, fir);
```

^(approximately)

2. 输出：

160.59506d3.3-6.3hahaha

错误程序编译输出：

line 24 pos 17 ->literal<- is a bad factor!!!

L22-> write(3.3);

;23-> write(hehe)

L24-> write('h')

^(approximately)

line 24 pos 17 expect basic type when writing.

L22-> write(3.3);

;23-> write(hehe)

L24-> write('h')

^(approximately)

3. 输出：

ABCDEF6

错误程序编译输出：

line 28 pos 9 Wrong start id c1 for a statement!

L26-> c2 := r1 - i1;

L27->

L28-> c1 := 2;

^(approximately)

4. 输出：

   ABCDEF6

   错误程序运行输出：

   ABCDEF--->>error at code[152] = f: 20 x: 0 y:

   1 reason:RUNTIME:array index below 0 !

5. 输出：

   in level 1: -52in level2: -5-4-3Ain level22: 2

   错误程序编译输出：

   line 14 pos 17 symbolpar3redefined!

   L12-> end;

   L13-> procedure l22(par3, par5 :integer);

   L14-> var par1,par3 : real;

                  ^(approximately)

   Sytax parser finished!!!

   in level 1: -52in level2: -5-4-3Ain level22: 2

2．测试结果分析

1. 本测试程序主要覆盖了常量定义，字符串，变量说明，数组的使用，过程函数说明，形式参数，传地址参数，if，while，for 语句结构，复合语句，赋值语句，函数返回值，表达式，项，因子，函数调用，条件，读写语句等语法结构。

本程序用于计算最大公约数和最小公倍数，如果输入的两个数都是正整数，那么输出正确计算结果。如果任何一个数为负，输出错误提示信息。如果输入其他内容，输出和行为将未定义。

2. 本测试程序主要覆盖了常变量声明，不同类型数字间的混合运算，函数调用，复杂表达式的求值，输出各种类型的表达式。

   输出结果为正确的计算结果。

3. 本测试程序主要覆盖了不同类型数字间的混合运算，不同类型间表达式的赋值，复杂表达式的求值。

   输出包括了冲突类型间的比较和赋值时的提示信息。

   以及通过的测试个数。

4. 本测试程序主要覆盖了各种类型的数组元素的参与运算和赋值，以及不同形式的表达式来计算数组下标。

   输出包括了冲突类型间的比较和赋值时的提示信息。

   以及通过的测试个数。

5. 本测试主要覆盖了嵌套子程序的外层和内层间相互调用，内存函数访问外层函数局部变量值，内层函数定义同名变量隐藏外层变量的功能。

   输出是外层到内层逐步调用后，因为作用域不同而同名或不同层次的变量值的情况。

五．总结感想 Summary

All good things must come to an end.

转眼间，持续两个月的《编译技术课程设计》课程就到最后一个阶段了。

这一次的课设主要还是集中在前半段的时间，工作量也在前半段，后半段时间主要坐等每周的测试，对于我的中难度文法来说较为轻松。

我的程序编写大概经过以下流程：

A. 第一阶段是初步认识文法，做词法分析器，把源程序拆成单词。

B. 第二阶段是构造剩余的一切。我做语法分析的时候就同步做了符号表，做了语法分析，每个子程序编写时就同时插入语义动作。

C. 做完上述模型后，我就试着把测试源程序翻译成了 P-code 代码，然后对照着代码表手工反汇编回语句，比对是否和源程序语义一致。这是个稍耗时的过程，不过做一遍就可以解决掉严重的结构问题和大部分的粗心错误了。

D. 接下来我就花了较少时间把解释器给做好了。然后就可以执行看结果来判断对不对了。

E. 从此之后我的编译器没有花太多时间再去维护。可能是写语法解析器的时候就下意识地把各个层次的问题和可能的边界情况考虑了。

对于大型系统这种深度优先的办法显然不可行，但是感觉对照着书上 Pascal-S 源码还是可以 hold 住的。

编译课设让我们通过实际构造一个完整的编译器，加深了编译器实现原理的理解，了解完整编译系统的构造方法。对于文法上，个人感觉中难度的文法还是覆盖到了现代高级程序设计语言的主要语法成分，不过没有实现 record 这种类似 struct 的语法结构，但是在书上源码有进行相应处理，类似于定义了一个过程。

文法中没有涉及到面向对象的类和对象的概念，可能这个实现起来的时候，更多的是一种程序的数据结构组织上的工作吧（除了那些多态等特征之外）。如果考虑上继承等元素，可能复杂度过大吧。

对于难度上，这次比较保险地选择了中难度。后来还是感觉得到的锻炼程度比较有限，属于一个比较通用/基础的层次。因为 P-code 的模式比较简单，栈式虚拟机直接可以执行中间代码，省却了再次从中间代码生成目标体系的代码的步骤。其实这其中还是有很多需要考量的地方。

最后非常感谢老师和助教在这个过程中的指导和付出。这次的编译课设让我对编译的过程有了更深入的理解。