形式语言与编译原理期末大作业

——设计简单语言 QL 的编译程序

赵宋文 王奕辰 冯志远 朱天宇 刘嘉政

问题综述

设计并实现一个简单语言 QL 的编译程序。

语言具体如下：

$$
\begin{aligned}
&P \longrightarrow Ď\ Š\\
&Ď \longrightarrow \varepsilon \mid Ď\ D\ ;\\
&Š \longrightarrow S \mid Š\ ;\ S\\
&D \longrightarrow T\ d \mid T\ d[Ǐ] \mid T\ d\ (Ď)\ \{Ď\ Š\}\\
&T \longrightarrow \text{int} \mid \text{float} \mid \text{void}\\
&Ǐ \longrightarrow i \mid Ǐ\ ,\ i\\
&S \longrightarrow d = E \mid \text{if}\ (B)\ S \mid \text{if}\ (B)\ S\\
&\text{else}\ S \mid \text{while}\ (B)\ S \mid \text{return}\ E \mid\\
&\{Š\} \mid d(Ě)\\
&Ě \longrightarrow E \mid Ě\ ,\ E\\
&E \longrightarrow i \mid d \mid d[Ě] \mid d(Ě) \mid E + E \mid\\
&E * E\\
&B \longrightarrow E\ r\ E \mid E
\end{aligned}
$$

语法

```
d = (ID, name)
ID = [a-z][a-z0-9]*
i = (INT, value)
INT = [+-]?[0-9]+
f = (FLOAT, value)
FLOAT = [+-]?[0-9]*. [0-9]+|[+-]?[0-9]+. [0-9]*
r = (ROP, name)
ROP = {<, ==, !=}
* = (MUL, -)
MUL = {*}
+=(ADD, -)
ADD={+}
\= = (ASG, -)
ASG         = {\=}
+\= = (AASG, -)
AASG = {+\=}
\( = (LPAR, -)
LPAR = {\(}
\) = (RPAR, -)
RPAR = {\)}
[ = (LBK, -)
LBK = {[}
] = (RBK, -)
RBK = {]}
\{ = (LBR, -)
LBR = {\{}
\} = (RBR, -)
RBR = {\}}
\, = (CMA, -)
CMA = {\,}
\; = (SEMI, -)
SEMI = {\;}
if = (IF, -)
else = (ELSE, -)
while = (WHILE, -)
return = (RETURN, -)
input = (INPUT, -)
print = (PRINT, -)
```

词法

程序组成部分：

1、 词法分析：组合 DFA
2、 词法分析：Scanner()
3、 语法分析：LR(0)规范集及冲突消解规则
4、 语法分析：SLR(1)分析表
5、 语义分析：SLR 引导的语义分析框架的程序实现
6、 中间代码生成：属性文法、符号表
7、 设计目标程序的存储映像格式及文件存储格式、构建存储映像
8、 产生过程调用代码序列及返回代码序列，构成过程的 “可执行代码” 。

实现

1. 词法分析

代码为 lexical_analyzer.cpp

本词法分析器对输入具有严格的要求，具体格式如下：

- 整数：整数仅由数字构成。例：123
- 浮点数：浮点数含有一个小数点，除此以外均由数字构成。例：456.789
- 标识符：标识符由非数字开头的合法字符（仅包括大小写字母、数字、下划线）表示。合法字符将在下文定义。例：a3b，temp。注：3ae 由数字开头，将被认为不合法。
- 字符串：字符串为由双引号围起来的由任意合法字符组成的串。例： “w3ad()%”
- 运算符：与关键字将在下表定义。

可识别的关键字和操作符共包括如下21+21=42种：

KEYWORD 关键字	if	else	void	return	while	for	do
	int	char	double	float	unsigned	main	string
	switch	case	cin	cout	break	const	bool
OPERATOR 操作符	+	-	*	/	<	>	=
	!	==	!=	<=	>=	%	||
	&&	++	--	<<	>>	+=	-=
DELIMITER 界符	,	;	(	)	[	]	{
	}	“	:				

关键字/操作符/界符表

注：在使用本词法分析器时，请使用上表中出现的符号，若使用其他未出现在上表中的关键字/操作符/界符，譬如^，|，&等，则程序将认为该字符无法识别；且请使用合规的字符（包括阿拉伯数字，大小写字母，各种上表中出现的算符与判断符），除此之外如#、$、\等都作为无法识别的字符处理。

观察需求，需要能识别不同种属的单词。因此首先要对单词进行划分。在要求中，已经给定了比较明确的分类标准：整数、浮点数、标识符、字符串、运算符。所以，为了能正确区分这几个种属，需要对各个类型做出界定。

在明确各个种属的不同之后，就需要对整个词法分析器程序做出规划：首先，我们需要获取输入，在本实验中采取从路径下读取文本内内容的方式获取输入；其次，我们需要对输入进行处理，这就需要合理的循环与条件判断，来将输入划分到具体的类别；最后是输出，根据已经分析处的单词类型，输出合适的内容，对不合规的单词也给出错误回馈。

由上述分析可知，我们大体需要三个部分，函数的设计也是从这个角度出发的。

- 输入函数：本实验中，并没有提供命令行输入的功能，而是提供文本路径，直接从文本中获取输入。在 main 函数中，创建了一个文件指针指向用户输入的文件，并将该指针传递给之后进行词法分析的函数。
- 分析函数 :在该函数中，需要对文本中的内容进行逐个读入并分析各个单词的含义。特别需要注意的是，空白符（包括空格，制表符’\t’，换行符’\n’和回车’\r’）不算单词，可以通过预处理直接去掉。但在本实现中，则是集成在整个文件的读取

循环中进行特判处理。在每次读取完一个新单词后，都会将之后的空白符全部跳过，直到读入下一个单词。处理完空白符，就需要对单词进行种属的划分，而这则是分析函数的绝大部分内容。

- 可复用函数：这是一类函数，包括输出，判断字符类型等函数。因为他们将在分析函数中被多次调用，因此称为“可复用函数”。
  - 输出函数：我们需要输出的是这个单词所属的 ID，这个单词的内容与单词属于的种属名。为了让程序看上去简洁清晰，可以单独完成一个 output 函数进行相关的处理，在函数内部完成输出格式的控制，以及由 ID 来确认这个单词的种属等工作。
  - 判断函数：本实验中有两个用于判断的函数，一个为判断输入的字符串是否为关键字的 isKeyword 函数，另一个是用于判断输入的字符是否合法的 isLegalChar 函数。

在这个部分，将对部分较为复杂的编程实现部分进行说明。完整的程序将在压缩包下的 Analyzer.cpp 中呈现，程序中附有较为详实的注释，可以对各部分的功能有清晰的了解。

main 函数之前的部分是在引入头文件、进行函数声明以及最重要的：对每个种属进行编号的工作。三个 enum 枚举类型分别对“关键字”、“操作符”和“界符”进行了划定，并赋予每个单词/符号单独的 ID。四个 const int 则对其余四种（整数、浮点数、字符串和标识符）有无穷可能的种属进行了 ID 的统一。

文件名输入相关

```
while(true) {
    std::cin >> File;
    if((fd = fopen(File, "r")) != NULL) {
        break;
    } else {
        std::cout << "文件名有误或文件不存在，请重新输入合规
                  的文件名：" << std::endl;
        std::cin.clear();
        std::cin.sync();
    }
}
```

main 函数中进行文件指向

这段代码中，最后两行 cin.clear()和 cin.sync()的作用是为了防止用户恶意输入（譬如 Ctrl+Z，输入中文汉字等）导致程序进入死循环，无法继续等待其他输入，只会不断报错。

注释处理

代码中，注释的部分应该剔除，本程序剔除注释的部分如下：

```
char temp = fgetc(fd);
if(temp == EOF) {
    break;
} else
if(temp == '/') {
    while((temp = fgetc(fd)) != EOF
        && (temp = fgetc(fd)) != '\n');
```

```
    } // 最后读入 '\n' 所以回到 case '\n': 情况，break 下沿
    // /* */ 注释
    else if(temp == '*') {
        bool flag = true; // 用于判断是否检索到 */
        while(temp != EOF && (temp == '*' || flag)) {
            if(temp == '*') {
                if((temp = fgetc(fd)) == '/'){
                    flag = false;
                    break;
                } /* 如果扫描到 * / 则结束循环，否则继续读取 */
                else fseek(fd, -1, SEEK_CUR);
            }
            temp = fgetc(fd);
        }
    } else {
        blank_flag = false;
        fseek(fd, -1, SEEK_CUR);
    }
```

其中的注释已经说明了部分的功能。因为 c++中的注释主要为“/**/”或“//”，因此，都需要进行提前搜索。由于 FILE 类型并未提供强大的 peek 函数，因此在本词法分析器中采取的方法是首先额外读入一个字节，在判断是否为两种注释类型其一后进行对应的处理（比如去除整行，一直跳过直到匹配的“*/”出现），若都不是，那么该字符“/”就是作为操作符的触发出现。此时则需要将读进来的额外一个字符给“吐”回去。fseek 函数提供了这个功能，-1表示回退一个单位，SEEK_CUR 表示从指针当前位置开始偏移“-1”个单位。

除了上述附出代码的两部分之外，其余部分都较为直观，譬如用于判断输入字符的 switch 开关函数，或者是用于输出的 output 函数等，都有较为直接的功能与易读的代码。

2. FOLLOW 集、FIRST 集、LR0 分析表与 SLR1 分析表

代码为 FIRST-FOLLOW.cpp

首先构造文法的规范 LR（0）项集族 C={I_0，I_1，I_2，…，I_n}，初始的项集族 C 中只包含一个初始项目的项目集闭包，对于 C 中的每一个项目集 I，对于每一个文法符号 X，如果 GOTO（I，X）【即项目 I 关于文法符号 X 的后继项目闭包】非空且不在 C 中，将其加入 C 中，重复执行该操作，至没有新的项目集加入 C。

然后根据得出的规范 LR（0）项集族 C，通过下面三种原则分析出各个状态的语法分析动作。

① 状态I_i中存在一个移入项目，移入的为终结符即 A->α·aβ，并且 GOTO（I_i，a）=I_j，则在分析表中 ACTION[i，a]=sj

② 状态I_i中存在一个移入项目，移入的为非终结符即 A->α·Bβ，并且 GOTO（I_i，B）=I_j，则在分析表中 GOTO[i , B]=j

③ 状态I_i中存在一个规约项目，即 A->α· ,并且 A 不是初始项目,那么将在分析表中对于 A 的所有 FOLLOW 集的元素,都将采取规约动作 ACTION[i，k]=rj 其中 k

为 A 的 FOLLOW 集元素,j 为产生式 A->α的标号

④ 状态I_i中为初始项目,S' ->S,那么该状态只接受句末终结符$,即 ACTION[i，$]=acc

⑤ 剩下没有定义的均视为 error

最后处理规约-移进冲突:

已知项目集 I 中有 m 个规约项目,n 个移进项目

$$\begin{cases} A_1 \to \alpha_1 \cdot a_1 \beta_1 \\ A_2 \to \alpha_2 \cdot a_2 \beta_2 \\ \quad\vdots \\ A_m \to \alpha_m \cdot a_m \beta_m \\ B_1 \to \gamma_1 \cdot \\ B_2 \to \gamma_2 \cdot \\ \quad\vdots \\ B_n \to \gamma_n \cdot \end{cases}$$

如果集合$\{a_1, a_2, \dots, a_m\}$与 FOLLOW(B_1), FOLLOW(B_2), …, FOLLOW(B_n)互不相交则该冲突可以通过以下的规则解决,如果 a 是下一个输入符号,①a∈$\{a_1, a_2, \dots, a_m\}$，则采用移进 ②a∈FOLLOW(B_i)则采取规约$B_i \to \gamma_i$ ③否则报错。

具体的 SLR(1)转移表在消除移进归约冲突的代码实现，我将以一下伪代码形式进行复现与展示：

```
BuildTable(string Exps)
{
    int k = FindFamily(Exps);
    //第一种情况
    if( IsVts(NextStat(Exps)) )//当前产生式将要读入终结符，NextStat 表示规范族预处理时设定的探查函数，返回要读入的下一个字符
    {
        string ch = NextStat(Exps);//存储将要读入的终结符
        int j = GO[k][Num(ch)];//Num 表示字符->数字的映射 GO 数组已经借助 DFA 预处理完成
        //构建预测分析表->移进
        ACTION[k][Num(ch)] = 's';
        state[k][Num(ch)] = j;
    }
    else if(NextStat(Exps) == '~')//产生式读入完毕，将要进行归约动作
    {
        int j = ExpsNum(Exps);//是第 j 个产生式

        if(j==1)//如果是第一个产生式：acc
        {
            ACTION[k][Num('#')] = 'acc';
            return ;
        }

        for(string ch : follow(Exps))
```

```
        {
            ACTION[k][Num(ch)] = 'r';
            state[k][Num(ch)] = j;
        }
    }
    else if( IsVns(NextStat(Exps)) )//读入非终结符
    {
        string ch = NextStat(Exps);
        int j = GO[k][Num(ch)];
        GOTO[k][Num(ch)] = j;
    }
}
```

最终我们得到的 LR(0)与 SLR(1)结果将在下文展示。

首先将原 QL 语言文法扩充成增广文法，增广文法如下：

P’ ->P
P->D’ S’
D’ ->ε | D’ D
S’ ->S | S’ ;S
D->T d | T d[I’] | T d(D’){D’ S’ }
T->int | float | void
I’ ->i | I’ ,i
S->d=E | if(B) S | if(B) S else S | while(B) S | return E | {S’ } | d(E’)
E’ ->E | E’ ,E
E->i | d | d[E’] | d(E’) | E+E | E*E
B->E r E | E

从而确定 LR（0）规范集，并使用 SLR 分析法，消解冲突。LR（0）规范集见附件。

构造 LR（0）分析表、构造 SLR（1）分析表

根据书上提供的算法，构建 LR（0）分析表，并处理移入-规约冲突。

```
closure(C) // C 是某状态对应的闭包
    while (C is still changing)
        for (item i: C) // example: i = A-> beta · B gamma
            C += {B -> ...}

Goto(C, x) //  求 GOTO 得到的新状态
    tmp = {}
    for (item i: C) // example: i = A-> beta · B gamma
        tmp += {A-> beta B · gamma}
    return closure(C)

LR0_table()
    C0 = closure(S'->·S$)
    state_set = {C0}
    Q = enqueue(C0)
    while (Q isn't empty)
        C = dequeue(Q)
        for (x: N+T) // N 是非终结符集，T 是终结符集
            D = Goto(C, x)
            if (x in T)
                ACTION[C][x] = D
            else
                GOTO[C][x] = D
            if (D isn't in state_set)
                state_set.add(D)
                enqueue(D)
```

然后通过分析 FOLLOW 集的方法，正确处理规约动作,将 LR（0）分析表转成 SLR（1）分析表：

FOLLOW(P)={$}

FOLLOW(D’)={d,if,else,while,return,{,int,float,void,)}

FOLLOW(S’)={$, ; , } }

FOLLOW(D)={d,if,else,while,return,{,int,float,void,),}

FOLLOW(S)={$, ; , } ,else }

FOLLOW(T)={d}

FOLLOW(I’)={] , , }

FOLLOW(E)={$, ; , } , else ,) , , , + , * ,] , r }

FOLLOW(B)={) }

FOLLOW(E’)={) , , ,] }

3. 语法分析

在已有的 token 流与 SLR(1)转移表的基础上，我们首先可以构建一个树结构来存放语法树，即代码中以 13 种 struct 的地址为节点的，通过链表方式构造出树型结构。对于每一格 SLR 表中内容，存在 s 移进操作和 r 归约操作和 goto 操作，我们需要分别讨论以获得具体的内容。特别是对于归约操作，对于每一条归约式，都存在不同的结点跳转情况、和 stack 出入栈情况。因此我们构建了一一对应的子函数，这种操作模式，也为后文中的属性文法的添加，增添了便利。

movein()函数代表移进，需要的参数分别是：移进后的状态 motion.second，由查询 slr1 表得出，其作为 token 流文法标志至 slr1 转移表内容的转换中介；tk 是输入 token 的类型，由 transfer_token()函数计算而来，其本质数据来自于文法分析的结果；iter->second 是每个文法标志的具体内容，例如 INT 的具体数值，ID 的具体 string 名称。在 movein 后，函数会为起建立单独的 struct 保存数据类型和值作为树的叶子节点。

```
state.push(0); //初始状态为 0
long tempIdol = long(new string("$"));
symb.push(tempIdol); //符号表预先留一个$
for (auto iter = tokens.begin(); iter != tokens.end(); iter++)
{
    int tk = transfer_token(*iter);
    pair<char, int> motion = slrp[sit + 3][tk];
    if (motion.first == 's')
    {
        movein(motion.second, tk, iter->second);
    }
    else if (motion.first == 'r')
    {
        reduce(motion.second);
    }
    else
    {
        cout << "SLR(1) table error!" << endl;
    }
}
```

移进函数的具体实现：

```
void movein(int stP, int tkP, string s) //移进
{
    //读 token，new 出的终结符节点入栈，计算下一个状态，下一个状态入栈
    state.push(stP);
    long temp_pt;
    switch (tkP)
    {
    case 1:
        i *tempp = new i();
        tempp->value = atoi(s.c_str());
        temp_pt = long(tempp);
        break;
    case 2:
        d *tempp = new d();
        tempp->name = s;
        temp_pt = long(tempp);
        break;
    case 14:
        r *tempp = new r();
        tempp->name = s;
        temp_pt = long(tempp);
        break;
    default:
        string *tempp = new string(s);
        temp_pt = long(tempp);
    }
    temp_son.push_back(temp_pt);
    symb.push(temp_pt);
}
```

归约操作，以第一条归约式为样例：一个递归的实现必须同时兼顾栈的维护，树节点的建立，和树关系的创立，还需要考虑到之后属性文法的计算过程，所以每一个归约式在归约

```
int n; //归约式右侧元素数
vector<int> s_red(10);
vector<long> y_red(10);
long newsymbol;
switch (idP)
{
case 1:
    n = 2;
    for (int i = 0; i < n; i++)
    {
        s_red[i] = state.top();
        state.pop();
        y_red[i] = symb.top();
        symb.pop();
    }
    sit = state.top();
    P *red_fa = new P();
    red_fa->son_num = n;
    for (int i = 0; i < n; i++)
    {
        red_fa->son.push_back(y_red[i]);
    }
    ATTR::P1(y_red[1], y_red[0]);
    newsymbol = long(red_fa);
    symb.push(newsymbol);
    state.push(atoi(slr[sit + 3][23].c_str()));
    break;
```

的过程中必须按顺序执行一下操作：

1. N 设为归约式右侧的符号个数
2. 从状态栈中弹出 n 个状态序数
3. 从符号栈中弹出 n 个符号/变元的结构体地址
4. New 一个新的结构体，其类型为归约式左侧的变元
5. 将弹出的 n 个符号/变元的结构体地址作为 son 属性的内容，也就是将新建节点定位其父节点
6. 执行属性文法的计算（继承属性）和中间代码的合成
7. 根据生成的新变元和旧状态查询 slr 表中的新状态，pop 入 stack 栈中。
8. 新变元 pop 入符号栈 symb 中
   在代码中出现的两个栈 state 为状态栈，symb 为符号栈，两者的存在一方面是便于文本状态的记录与回溯，另一方面是中间过程更加直接明了。

4. 语法制导翻译

代码 parser.cpp

首先需要解决属性定义，对 QL 文法中的每一个变元和终结符定义相应的综合属性，使得在构建语法树的过程中能通过计算各个树节点的属性值生成中间代码和符号表。各文法符号的属性定义说明如下。

属性定义

变元 ǐ 有属性 place。ǐ.place 为一个整型列表，存储整数列表 ǐ 中的所有整数，在 C++中实现为 vector<int>。

变元 E 有属性 place 和 code。E.place 为一个字符串，记录存储表达式 E 的值的变量的名称，C++中实现为 string；E.code 也为字符串，记录计算表达式 E 的值所需的所有中间代码。

变元 ě 有属性 place 和 code。其分别为表达式列表 ě 包含的所有表达式 E 的 E.place 和 E.code 的列表，实现为 vector<string>。

变元 B 有属性 tc，fc 和 code。B.tc 为一个字符串，记录若表达式 B 为真，中间代码将会跳转到的标号的名称；B.fc 为一个字符串，记录若表达式 B 为假，中间代码将会跳转到的标号的名称；B.code 为一个字符串，存储判断表达式 B 的真假所需的中间代码。

变元 S 有属性 code，其为一个字符串，记录语句 S 对应的所有中间代码；变元 š 有属性 code，其为字符串列表，是语句段 š 内所有语句 S 的 S.code 的列表。

变元 T 有属性 type 和 width。T.type 为一个字符串，可以是 INT，FLO 或 VOID；T.width 为一个整型，是单个 INT，FLO 或 VOID 类型的变量所需要占用的字节数。

变元 D 有属性 tab，place 和 func_or_array。D.tab 为登记有声明语句 D 声明的变量的轻符号表；D.place 为一个记录声明语句 D 声明的变量名称的字符串；D.func_or_array 为一个 bool 型变量，记录 D 定义的变量是否为数组或过程。

变元 Ď 有属性 place 和 func_or_array。Ď.place 为声明语句段中声明的所有变量名称的列表，Ď.func_or_array 记录声明语句段中是否声明了数组或过程。

终结符的属性包括 d.name，i.value 和 r.name，其分别为标识符 d 的名称，常量 i 的值，关系运算符 r 的名称。d 还有其他属性，如 type，offset 等等，这些属性声明 d 时在符号表中登记，使用 d 时通过符号表来查找。

有了属性定义，需要为文法中每一个产生式编写相应的语义代码，来完成属性计算。属性计算的思路和代码实现就不过多赘述，参见附录。在此仅对程序中定义的符号表及其操作、声明语句的语义代码中符号表的构建做以说明。

符号表及其操作

程序中定义符号表类 Tab。包含表头信息 outer，width，code，rtype，args，arglist，以及所有登记项的列表 list，list 的类型为 vector<Item>。

```
struct Tab;
struct Item
{
    string name,type;
    int offset; Tab* mytab;
    string etype; int dims,dim[5],base; //至多 6 维数组
};
struct Tab
```

```
{
    Tab* outer;
    int width;
    string code;
    string rtype;
    int args; vector<string> arglist;
    vector<Item> list;
    Tab(){outer=NULL,width=0,code="",args=0;}
};
```

符号表的操作包括一下几个函数：

void bind(Tab &tb,string name,string type)，在符号表 tb 中添加登记项，名字域为 name，类型域为 type。

void lookup_c(Tab &tb,string name,string seg_name,string str="",int it=0,Tab* add=NULL)，在符号表 tb 中进行修改操作，找到名字域为 name 的登记项，将其 seg_name 域修改为对应的值，seg_name 域为字符串类型则改为 str，为整型则改为 it，为 Tab*型则改为 add。

string lookup_t(string name)，在 symtab 栈顶的前2个符号表里查名字域为 name 的登记项，找到则返回其类型，没有找到则返回 UNBOUND。

int lookup_d(string name,int i=-1)，在 symtab 栈顶的前2个符号表里查名字域为 name 的登记项。若参数 i 缺省则返回其 dims 域的值，否则返回其 dim[i]域的值。找不到该登记项则返回-1。

int lookup_b(string name)，在 symtab 栈顶的前2个符号表里查名字域为 name 的登记项，返回其 base 域的值。找不到该登记项则返回-1。

void merge(Tab &A,Tab &B)，合并符号表 A 和符号表 B，结果存在符号表 A 中。

符号表的构建

QL 语言文法允许过程定义和过程嵌套定义，因此需要用一个符号表栈 symtab 来记录当前过程或全局的符号表，另外在开一个符号表列表 tablist 存储构建好的符号表。此处 symtab 和 tablist 存储的都是符号表的地址，真正的符号表是通过 new 创建。

对于 D′ ⟶ ε，创建一个空符号表，并将其地址压入 symtab 栈。每次 D′ ⟶ D′D;时将轻符号表 D.tab 并入 symtab 栈顶的符号表。不难看出，当前 D′ 中声明的所有变量的符号表就是 symtab 栈顶的符号表。

对于 D ⟶ T d (D′) {D′š}，symtab 栈顶的前2个符号表即为第一个 D′ 和第二个 D′ 的符号表，分别记为 tab 和 tab_inner。将 tab 和 tab_inner 出栈。将 tab_inner 并入 tab，删除 tab_inner。将 š.code 存入 tab 的 code 域，第一个 D′.place 存入 tab 的 arglist 域，tab 的 outer 域指向 symtab 栈顶符号表。tab 即为过程 d 的符号表，将其存入 tablist 中。

对于 P ⟶ D′š，symtab 的栈顶符号表即为 D′ 的符号表，记为 tab。将 tab 出栈。将 š.code 存入 tab 的 code 域，tab 的 arglist 置为空，tab 的 outer 域指向 NULL。tab 即为全局符号表，将其存入 tablist 中。

属性计算完成后，tablist 即为所有构建完成的符号表，符号表的 code 域即为中间代码。

语法制导翻译的实现

语法制导翻译，就是要在构建语法树的同时进行属性计算，生成符号表和中间代码。每个文法符号的结构体中包含了该文法符号的所有属性。对于每一个产生式，将产生式

的语义代码编写为一个函数。函数的形参依次为属性计算需要用到的文法符号的结构体类型。函数没有返回值，形参为传地址，函数中计算出形参的属性值即可。例如，产生式 B⟶ E r E 的函数定义如下：

```
void P29(B &B, E &E0, r r, E &E1) //29 为产生式编号
{
    string l1 = newlabel(), l2 = newlabel();
    B.tc = l1;
    B.fc = l2;
    B.code = E0.code + E1.code + gen("if", E0.place,
r.name, E1.place, "then", l1, "else", l2);
}
```

在 SLR 分析的每一步归约时，生成新变元的语法树节点，先构建语法树。之后只需以新生成的语法树节点和他的儿子节点的地址为实参，调用一次归约产生式对应的函数即可完成属性计算。

错误处理

语法制导翻译的有三种可能的终止情况：语法错误、语义错误、编译成功。程序分别对这三种情况进行处理，如果发现错误则输出错误信息，编译成功则输出符号表及中间代码。

- ---GRAMMATICAL ERROR---
  - SLR 分析中，当前状态和下一个 token 在 SLR 分析表中的对应位置为空
  - SLR 分析中，还没有走到 acc 状态就消耗完了输入 token 流
- ---SEMANTIC ERROR---
  - 未声明或声明类型与调用方式不匹配：使用标识符 d 时，在 symtab 栈顶的前2个符号表中找不到 d 的登记项，或者登记的类型与器调用方式不一致
  - 过程和数组不能作为其他过程的参数：作为函数参数声明语句段的 D˝ 的 func_or_array 属性为真。之所以有这条规则是因为 QL 文法中调用过程的实参只能是表达式
  - 数组维数不能超过6：声明数组是表达式列表长度超过6
  - 数组维数不能小于1：声明数组是表达式列表为空
- ---COMPILATION SUCCESS---
  - 没有发现语法错误或语义错误，则输出符号表及其对应的中间代码。

5. 目标代码生成与其存储格式

栈式存储分配：当一个过程被调用，该过程的活动记录被压入栈；当过程结束时，该活动记录被弹出栈。这个过程的活动序列的入栈的顺序一般是：

1.实际参数
2.控制链：指向被调用者的活动记录
3.访问链：用来访问存放于其他活动记录中的非局部数据
4.保存的机器状态
5.局部数据
6.临时变量

调用序列：实现过程调用的代码段，为一个活动记录在栈中分配空间，并在此记录的字段中填写信息（入栈）

调用者计算实际参数的值，将返回地址（程序计数器的下一个地址）放到被调用者的机器状态字中，将原来的栈顶指针放到被调用者的控制链中，然后，增加栈顶指针的值，使其指向被调用者局部数据开始位置（此时栈顶指针指向自身），被调用者保存寄存器值和其他状态信息，被调用者初始化其局部数据并开始执行。

返回序列 :回复机器状态，使得调用过程能够在调用结束之后继续执行（出栈）

被调用者将返回这放到与参数相邻的位置，使用机器状态字段中的信息，被调用者恢复栈顶指针和其他寄存器，被调用者跳转到恢复由调用者放在及其状态字段中的返回地址，尽管栈顶指针已经被减小，但调用者仍然知道返回值相对于当前栈顶指针值的位置，因此，调用者可以使用那个返回值。从而返回原来的位置。

运行结果

1. 词法分析

使用说明：请注意输入文件地址时，我们预写的测试源代码在 eval_input 文件夹中，代码输出格式与编码格式均为 GB2312.

t0 输入：

```
k = k * 2 – p
```

输出:

```
词法分析的结果为：
300          k   ID
 56          =   =
300          k   ID
 52          *   *
200          2   INT
 51          -   -
300          p   ID
```

t2 输入：

```
int a1;
int a2;
int a3;
int p(){return 1}
a1 = 1+2;
a2 = a1+4;
a3 = 5*2;
if(a3<a2){
if(a2<a1)
{
a3 = 100;
return a3
}
else a1 = a2;
}
else a1 = p()
}
```

输出:

```
词法分析的结果为：
  7        int   i
300         a1   ID
101          ;   ;
  7        int   i
300         a2   ID
101          ;   ;
  7        int   i
300         a3   ID
101          ;   ;
  7        int   i
300          p   ID
102          (   (
103          )   )
106          {   {
  3    return   return
200          1   INT
107          }   }
300         a1   ID
 56          =   =
200          1   INT
 50          +   +
200          2   INT
101          ;   ;
300         a2   ID
 56          =   =
300         a1   ID
 50          +   +
200          4   INT
101          ;   ;
300         a3   ID
 56          =   =
200          5   INT
 52          *   *
200          2   INT
101          ;   ;
  0         if   if
102          (   (
300         a3   ID
 54          <   <
300         a2   ID
103          )   )
106          {   {
  0         if   if
102          (   (
300         a2   ID
 54          <   <
300         a1   ID
103          )   )
```

2. FOLLOW 集、FIRST 集、LR0 分析表与 SLR1 分析表

t0 输入：original_grammar.txt 即题目所给的语法规则的文档

```
P->Dp
P->Sp
Dp->~
Dp->Dp D
Sp->S
Sp->Sp ; S
D->T d
D->T d [ Ip ]
D->T d ( Dp ) { Dp Sp }
T->int
T->float
T->void
Ip->i
Ip->Ip , i
S->d = E
S->if ( B ) S
S->if ( B ) S else S
S->while ( B ) S
S->return E
S->{ Sp }
S->d ( Ep )
Ep->E
Ep->Ep , E
E->i
E->d
E->d [ Ep ]
E->d ( Ep )
E->E + E
E->E * E
B->E r E
B->E
#
```

输出:

```
input production (end with #):
FIRST():
B:d i
D:float int void
Dp:~
E:d i
Ep:d i
Ip:i
P:d if return while { ~
S:d if return while {
Sp:d if return while {
T:float int void
FOLLOW():
B:
D:
Dp:$
E:
Ep:
Ip:
P:$
S:$
Sp:$
T:
```

注 :与前文答案有所不同是因为原版文法不完善,我们自定义了一些出现的冲突的解决方法。对于每个产生式，由于其复杂的结构，无法确定其结束条件，因而无法读取右侧所有独立元素，若要读取每个元素，则要为每个产生式增添一个识别符。

3. 语法制导翻译

从词法分析的输出文件夹 output 中读入 token 流，构建语法树，计算属性值，处理语法错误和语义错误，生成符号表和中间代码。

t0 的编译结果

```
Read 98 lines
---LOAD SLR1 FINISHED---
---FINISHED TRANS SLR---
Read 5 lines
---LOAD TOKEN DOC FINISHED---
---COMPILATION SUCCESS---
t0 = 1
a1 = t0
```

t2 的编译结果

```
Read 98 lines
---LOAD SLR1 FINISHED---
---FINISHED TRANS SLR---
Read 66 lines
---LOAD TOKEN DOC FINISHED---
---COMPILATION SUCCESS---
t0 = 1
return t0


t1 = 1
t2 = 2
t3 = t1 + t2
a1 = t3
t4 = 4
t5 = a1 + t4
a2 = t5
t6 = 5
t7 = 2
t8 = t6 * t7
a3 = t8
if a3 < a2 then t0 else t1
label t0
if a2 < a1 then t2 else t3
label t2
t9 = 100
a3 = t9
goto t4
label t3
a1 = a2
label t4
goto t5
label t1
t10 = 3
t11 = a2 + t10
par t11
call p , 1
a1 = REG0
label t5
```

t2r1 把整型变量 a1 当作函数进行调用，其编译结果如下

```
Read 98 lines
---LOAD SLR1 FINISHED---
---FINISHED TRANS SLR---
Read 66 lines
---LOAD TOKEN DOC FINISHED---
---SEMANTIC ERROR---
a1未声明或声明类型与调用方式不匹配
```

t2r2 存在语法错误，其编译结果如下

```
Read 98 lines
---LOAD SLR1 FINISHED---
---FINISHED TRANS SLR---
Read 65 lines
---LOAD TOKEN DOC FINISHED---
---GRAMMATICAL ERROR---
```

附件 1：SLR(1)表
附件 2：LR（0）规范集
附件 3：属性定义、语义代码及符号表