编译原理课程设计

学 院（系）：电子信息与电器工程学部

学 生 姓 名：黄楚冰

学　　号：201281724

班　　级：计日 1201

同 组 人 ：黄楚冰

大连理工大学

Dalian University of Technology

目　　录

1 源语言定义

1.1 样本语言文法定义

1.1.1 语言定义

非终结符集合 : A B C D E F F1 F2 FR L M P S T

终结符集合 : ( ) * + - / ; = cmp else for id if main num rel return type void while { }

表 1.1 样本语言产生式文法定义

产生式	
<主程序 M>	-> void main ( ) { <子程序 P> return ; }
<子程序 P>	-> <声明 D> <语句 S>
<声明 D>	-> <声明 D> type id ; | type id ;
<语句 S>	-> <赋值语句 A> | { <语句序列 L> }
<语句 S>	-> if (<布尔表达式 B>) { L } else { L }
<语句 S>	-> if (<布尔表达式 B>) { <语句序列 L> }
<语句 S>	-> while (<布尔表达式 B>) { L } | FR
<for 语句 FR>	-> for (<赋值语句 A> ; <布尔表达式 B> ; < F1 > < F2 >
<F1>	-> <赋值语句 A>) {
<F2>	-> <语句序列 L> }
<赋值语句 A>	-> id = <表达式 E>
<语句序列 L>	-> <语句 S> ; <语句序列 L> | <语句 S> | 空
<布尔表达式 B>	-> <布尔表达式 B> rel <比较表达式 C> | <比较表达式 C>
<比较表达式 C>	-> id cmp id | id cmp num | id
<表达式 E>	-> <项 T> | <表达式 E> + <项 T> | <表达式 E> - <项 T>
<项 T>	-> <因子 F> | <项 T> * <因子 F> | <项 T> / <因子 F>
<因子 F>	-> num | id | (<表达式 E>)

1.1.2 增广文法

0 $ -> M
1 M -> void main ( ) { P return ; }
2 P -> D S
3 D -> D type id ;
4 D -> type id ;
5 S -> A
6 S -> { L }
7 S -> if (B) { L } else { L }
8 S -> if (B) { L }
9 S -> while (B) { L }

10 S -> FR
11 FR -> for (A ; B ; F1 F2
12 F1 -> A) {
13 F2 -> L }
14 A -> id = E
15 L -> S ; L
16 L -> S
17 L -> #
18 B -> B rel C
19 B -> C
20 C -> id cmp id
21 C -> id cmp num
22 C -> id
23 E -> T
24 E -> E + T
25 E -> E - T
26 T -> F
27 T -> T * F
28 T -> T / F
29 F -> num
30 F -> id
31 F -> (E)

1.2 单词的识别模型-有穷自动机 DFA

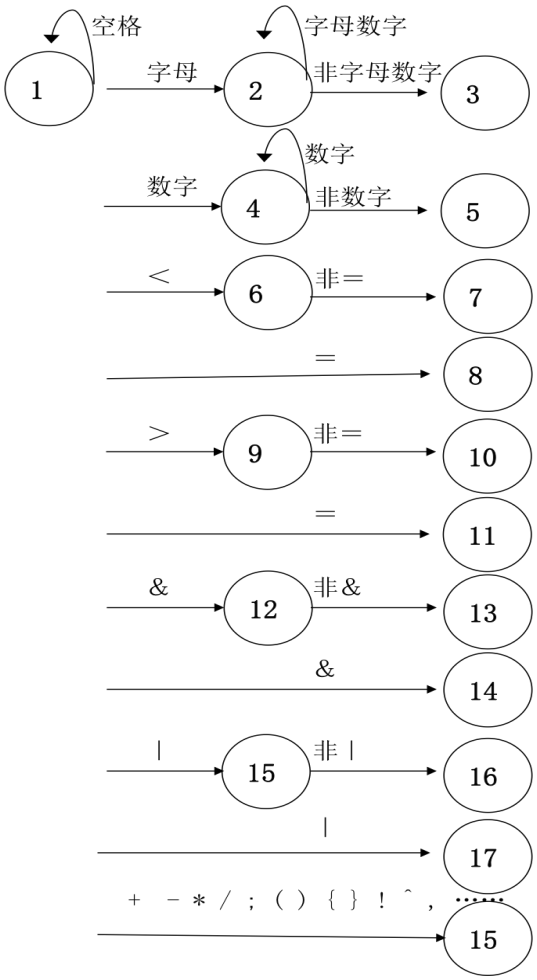

图 1.1 有穷自动机 DFA 状态转换图

2　词法分析程序的实现

2.1　功能说明

词法分析器能够分析整数、标识符、主要运算符和主要关键字，生成助记符标签，在语法分析程序中应用。

词法分析器从头到尾，从左到右扫描字符，并分解，识别单词或者符号。最后生成每个单词或者符号对应的助记符。

2.2　状态转换 DFA 及词法定义

2.2.1　状态转换 DFA

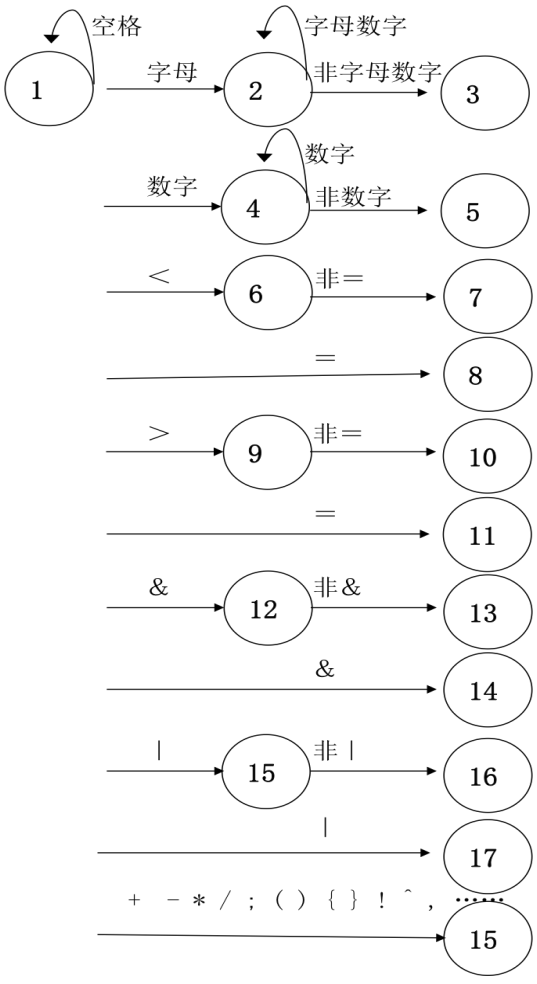

图 2.1　有穷自动机 DFA 状态转换图

2.2.2 词法定义

助记符	单词或符号
id	标识符
num	整数
for	for
while	while
do	do
if	if
else	else
void	void
const	const
return	return
main	main
include	include
type	int
type	double
type	float
type	char
+	+
-	-
*	*
/	/
%	%

助记符	单词或符号
&	&
|	|
^	^
~	~
!	!
=	=
cmp	<
cmp	<=
cmp	>
cmp	>=
cmp	==
cmp	!=
rel	&&
rel	||
,	,
;	;
(	(
)	)
{	{
}	}

2.3 代码实现

2.3.1 输入及输出

输入是正常的代码，按照字符开始扫描，按照自动机的状态转移，分解成单词或符号。

输出是一个容器，容器里存放了 Word 结构体，每个 Word 里记录了这个单词或者符号在哪一行，助记符，原本的名称或者符号。

2.3.2 实现过程及解释

（1） Word 结构体：

包含行号 line，助记符 label，原本的单词或者符号 name。

如果是数字，val 设置为数字的值，否的 val 值为空

（2） Lexer 类：

a) 存储结构包括：用于记录行数的 line，存储当前字符的 ch，和存储生成的单词 Word 的 vector 容器 words。

b) 函数包括：初始化函数，移除空格函数 remove_blank 对应自动机空格的转移，分析数字函数 number，分析单词函数 word，分析其他符号和字符函数 others，用于存 val 的将字符串转换数字的函数 char2num，检查是否是关键字函数 check_key，检查是否是运算符和其他合法符号的函数 judgeOpt，运行词法分析的入口函数 runLexer，输出结果的函数 print。

2.4 测试结果

2.4.1 测试 1

（1） 输入：

```
void main()
{
    int a,b;
    if(a!=b || b==0)
    {
        a=a-b;
    }
    if(b>=a && a<0)
        a=b=0;
    while(a<0)
    {
        a=a+2;
    }
    return;
}
```

图 2.2 词法分析测试输入图

（2） 输出：

行号	助记符	name	val 值
1	void	void	0
1	main	main	0
1	(	(	0
1	)	)	0
2	{	{	0
3	type	int	0
3	id	a	0
3	,	,	0
3	id	b	0
3	;	;	0
4	if	if	0
4	(	(	0
4	id	a	0
4	cmp	!=	0
4	id	b	0
4	rel	||	0

4	id	b	0
4	cmp	==	0
4	num	0	0
4	)	)	0
5	{	{	0
6	id	a	0
6	=	=	0
6	id	a	0
6	-	-	0
6	id	b	0
6	;	;	0
7	}	}	0
8	if	if	0
8	(	(	0
8	id	b	0
8	cmp	>=	0
8	id	a	0
8	rel	&&	0
8	id	a	0
8	cmp	<	0
8	num	0	0
8	)	)	0
9	id	a	0
9	=	=	0
9	id	b	0
9	=	=	0
9	num	0	0
9	;	;	0
10	while	while	0
10	(	(	0
10	id	a	0
10	cmp	<	0
10	num	0	0
10	)	)	0
11	{	{	0
12	id	a	0
12	=	=	0
12	id	a	0
12	+	+	0
12	num	2	2
12	;	;	0
13	}	}	0
14	return	return	0
14	;	;	0
15	}	}	0

3 自底向上实现语法语义分析程序

3.1 LR(1)语法分析

3.1.1 LR(1)语法分析方法概述

自底向上的 LR(1)分析方法，适用文法广，允许左递归，能解决 SLR(1)解决不了的冲突，分析能力强，分析速度快。

L 指自左至右扫描输入符号串

R 指构造一个最右推导的逆过程（最左归约）

1 指在作出分析决定前要向前看的输入符号个数为 1

自底向上分析法，也称移进-归约分析法。它的实现思想是对输入符号串自左向右进行扫描，并将输入符逐个移入一个后进先出栈中，边移入边分析，一旦栈顶符号串形

成某个句型的句柄时，(该句柄对应某产生式的右部)，就用该产生式的左部非终结符代替相应右部的文法符号串，这称为一步归约。重复这一过程直到归约到栈中只剩文法的开始符号时则为分析成功，也就确认输入串是文法的句子。

3.1.2 first 集的生成

程序代码的实现过程按照下述算法。

求 First 集的步骤：

若 X->a..，则将终结符 a 加入 FIRST(X)中;

若 X->e ,则将终结符 e 加入 FIRST(X)中(e 表示空集)；

若 X->BC..D,则将 First（B）所有元素（除了空集）加入 First（A），然后检测 First（B），若 First（B）中不存在空集, 即 e,则停止，若存在则向 B 的后面查看，将 First（C）中所有元素（除了空集）加入 First（A），然后再检测 First（C）中是否有 e...直 到最后，若 D 之前的所有非终结符的 First 集中都含有 e,则检测到 D 时，将 First（D）也加入 First（A），若 First（D）中含有 e,则将 e 加入 First（A）。

终结符：通俗的说就是不能单独出现在推导式左边的符号，也就是说终结符不能再进行推导。

非终结符：不是终结符的都是非终结符。如：A->B，则 A 是非终结符；A->id,则 id 是终结符。（一般终结符用小写，非终结符大写。）

3.1.3 项目闭包的生成

程序代码的实现过程按照下述算法。

把 S′ →·S，#放于初始项目集中，把'#'号作为向前搜索符，表示活前缀为γ(若γ是有关 S 产生式的某一右部)要归约成 S 时，必须面临输入符为'#'号才行。因此对初始项目 S′ →·S，# 求闭包后再用转换函数逐步求出整个文法的 LR(1)项目集族。

构造 LR(1)项目集的闭包函数：

a) I 的任何项目都属于 CLOSURE(I)

b) 若有项目[A→α·Bβ,a]属于 CLOSURE(I)，B→γ是文法中的产生式，β∈V*，b∈FIRST(βa)， 则[B→·γ,b]也属于 CLOSURE(I)中。

c) 重复 b)直到 CLOSURE(I)不再增大为止。

3.1.4 项目集规范簇的生成

程序代码的实现过程按照下述算法。

把 S′ →·S，#属于初始项目集中，对初始项目 S′ →·S，# 求闭包后再用转换函数逐步求出整个文法的 LR(1)项目集族。

转换函数的构造：

a) LR(1)转换函数的构造与 LR(0)的相似，GO(I,X)＝CLOSURE(J)

b) 其中 I 是 LR(1)的项目集，X 是文法符号，J={任何形如[A→αX·β,a]的项目 | [A→α·Xβ,a]∈I}

c) 对文法 G′ 的 LR(1)项目集族的构造仍以[S′ →·S,#]为初态集的初始项目，然后对其求闭包和转换函数，直到项目集不再增大。

3.1.2 LR(1)分析表的生成

程序代码的实现过程按照下述算法。

若已构造出某文法的 LR(1)项目集族 C：C＝{I0,I1,…,In}，其中 Ik 的 k 为分析器的状态，则动作 ACTION 表和状态转换 GOTO 表构造方法如下：

a) 若项目[A→α·aβ,b]属于 Ik，且 GO（Ik,a）=Ij，其中 a∈VT,则置 ACTION[k,a]=Sj。其 Sj 的含义是把输入符号 a 和状态 j 分别移入文法符号栈和状态栈。

b) 若项目[A→α·,a]属于 Ik，则置 ACTION[k,a]= rj 其中 a∈VT，rj 的含义为把当前栈顶符号串 α 归约为 A(即用产生式 A→α 归约)。j 为在文法中对产生式 A→α 的编号。

c) 若项目[S′ →S·，#]属于 Ik，则置 ACTION[k，#]＝"acc"，表示"接受"。

d) 若 GO(Ik，A)＝Ij，其中 A∈VN，则置 GOTO[k,A]＝j。表示转入 j 状态，置当前文法符号栈顶为 A，状态栈顶为 j。

e) 凡不能用规则 a) ～ d)填入分析表中的元素，均置"报错标志"。程序中用(F, -1)表示，用户界面中用空白表示。

3.2 属性的传递

3.2.1 属性文法

属性文法指为每个文法符号引入一个属性集合，反映相应语言结构的语义信息，如标识符的类型、常量的值、变量的存储地址等。

属性的类型包括综合属性和继承属性：

a) 综合属性：属性值是分析树中该结点的子结点的属性值的函数。

b) 继承属性：属性值是分析树中该结点的父结点和或兄弟结点的属性值的函数。

非终结符既可有综合属性也可有继承属性。

文法开始符号没有继承属性。

终结符号只有综合属性。

3.2.2 程序中属性的传递方式

在进行移进操作的时候赋予语法树叶子结点相应的综合属性，在进行规约的时候栈中待规约项目的继承属性传递给兄弟节点。

3.3 语义分析及四元式生成

3.3.1 分析方法及语义规则

在进行移进操作的时候赋予语法树叶子结点相应的综合属性，在进行规约的时候栈中待规约项目的继承属性传递给兄弟节点。

语义规则基本按照书上的语义规则实现。

语义分析一旦遇到错误 1 处则输出错误退出语义分析程序不再继续分析。

3.3.2 拉链回添

当四元式中的转向不确定时，将所有转向同一地址的四元式地址码拉成一个链；

一旦所转向的地址被确定，则沿此链向所有的四元式地址码中回填入此地址

3.4 类型表和地址表的实现

类型表和地址表通过 C++STL 中的 map 容器实现。map 提供了很好一对一的关系，它存储的是一个键－值对，键和值的数据类型可以是不相同的。它能提供对 T 类型的数据快速和高效的检索。

类型表的地址表是定义在 Semantic 类中的 typeTable 和 placeTable。typeTable 的健和值都是 string 类型，分别代表变量名字和变量类型，通过变量的名字可以直接查到变量对应的类型。placeTable 的健是 string 类型，值是语法树节点，通过变量名称能快速检索到对应的语法树节点，改节点包含其各种属性。

3.5 可检查的错误类型

未声明主函数 void main { return; }

非法的程序定义

缺少 return

缺少 ;

缺少 (

缺少)

缺少 {

缺少 }

使用未声明的变量

声明结束不能直接结束程序，需要有程序语句	布尔语句不符合语言规则
条件控制语句缺少条件	非法符号
	非法语句

3.6 代码实现

3.6.1 语法分析器

语法分析器的功能包括：

（1） 能自动生成非终结符集，终结符集，增广文法

（2） 能自动生成 First 集

（3） 能自动生成项目集规范簇

（4） 能自动生成 LR(1)分析表

上述功能按照 3.1 节陈述的算法步骤代码模拟实现。

（1） 自己定义了语法产生式规则的结构体 Rule：

a) 存储信息包括：规则编号，左边的非终结符，右侧的所有单词或者符号。

b) 函数包括：构造函数，打印函数和运算符重载函数。

（2） Grammar 类：

自己定义了项目闭包的类 Item，其中定义了存储结构 State。

a) State 包括：

存储信息包括：原点位置 dot，向前查看的输入符号 ahead。

函数包括：构造函数和运算符重载函数。

b) Item 包括：

存储结构包括项目闭包集合 itemSet。

函数包括运算符重载函数和打印函数。

c) Grammar 的存储结构：

空符号 ept，文法开始符号 startSym，非终结符号集合 vn，终结符号集合 vt，文法规则的存储容器 rules，first 集 first，存储临时闭包的 item，规范项目簇 closure，Goto 表 go_to，Action 表 action。

d) Grammar 的函数：

构造函数 Grammar，生成 first 集函数 geneFirst，判断是否是终结符函数 isVt，判断是否空规则函数 hasNotVancancy，初始化项目集规范簇 I0 函数 initClosure，初始化每个规范簇的 goto 和 action 表函数 initTable，在生成项目集规范簇时取得后面字符和检查字符的 first 集函数 getFirst，生成项目集闭包函数 geneSet，生成项

目集规范簇，用于事先的 debug 函数 geneClosure，检查当前生成的项目集闭包是否已经存在的函数 findSet，判断 2 个项目集闭包是否相等函数 eqlSet，生成项目集规范簇的同时生成 goto 和 action 表的函数 buildLR1table，找到需要规约时候的规则号，用于生成 action 表的函数 getRuleNum，初始化语义分析所用 Vn, Vt, First 集和 LR1 表函数 initSyntex 和打印函数 print。

3.6.2 语义分析及中间代码生成器

语义分析器的功能包括：

（1） 能分析经过词法分析器分析后的代码序列是否符合样本语言的文法定义，并返回错误。

（2） 在分析的同时生成语法树和四元式代码

（3） 错误检查。语义分析一旦遇到错误 1 处则输出错误退出语义分析程序不再继续分析。

上述功能按照 3.1 节陈述的算法步骤代码模拟实现。

（1） 自己定义了中间代码结构 MidCode：

存储信息包括：中间代码的 id，中间代码的值 data。

函数包括：构造函数和打印函数。

（2） 语法树 syntaxTree 类：

a) 自己定义了语法树节点 TreeNode 结构体：

存储信息包括：

符号标签（终结符的 label 或非终结符）label，父亲节点 fa，孩子节点 child，用于画树的 pos 和 size

属性信息包括：

节点名字 name，节点类型 type，节点的数值 val，代码出现的行号 line, 节点代码开始的位置 place, 节点为真情况下代码开始的位置 tr, 节点为假情况下代码开始的位置 fl;

函数包括：

构造函数，设置父节点函数 setFa。

b) syntaxTree 的存储信息：

四元式标号记录 labels，语法树根结点 root。

c) syntaxTree 的函数：

构造函数，生成新四元式标号函数 newLabel，打印函数。

（3） Semantic 类：

存储信息包括：四元式代码的存储容器 Code，类型表 typeTable，地址表 placeTable。函数包括：构造函数 Semantic，语义分析程序函数 semantic，用于添四元式信息的将数字转换成字符串函数 num2Char，拉链回填函数 backpatch，输出四元式函数 emitMidCode，开始语法语义分析的接口函数 start。

3.7 测试结果

3.7.1 语法分析器

（1） 测试 1 及 LR(1)分析表

S->E
E->T | E + T
T->P | T * P
P->num
P->(E)

LR1 Table :

	Action						GOTO			
	(	)	*	+	num	#	E	P	S	T
0	S5	F	F	F	S6	F	2	4	1	3
1	F	F	F	F	F	A	-1	-1	-1	-1
2	F	F	F	S7	F	r1	-1	-1	-1	-1
3	F	F	S8	r2	F	r2	-1	-1	-1	-1
4	F	F	r4	r4	F	r4	-1	-1	-1	-1
5	S12	F	F	F	S13	F	9	11	-1	10
6	F	F	r6	r6	F	r6	-1	-1	-1	-1
7	S5	F	F	F	S6	F	-1	4	-1	14
8	S5	F	F	F	S6	F	-1	15	-1	-1
9	F	S17	F	S16	F	F	-1	-1	-1	-1
10	F	r2	S18	r2	F	F	-1	-1	-1	-1
11	F	r4	r4	r4	F	F	-1	-1	-1	-1
12	S12	F	F	F	S13	F	19	11	-1	10
13	F	r6	r6	r6	F	F	-1	-1	-1	-1
14	F	F	S8	r3	F	r3	-1	-1	-1	-1
15	F	F	r5	r5	F	r5	-1	-1	-1	-1
16	S12	F	F	F	S13	F	-1	11	-1	20
17	F	F	r7	r7	F	r7	-1	-1	-1	-1
18	S12	F	F	F	S13	F	-1	21	-1	-1
19	F	S22	F	S16	F	F	-1	-1	-1	-1
20	F	r3	S18	r3	F	F	-1	-1	-1	-1
21	F	r5	r5	r5	F	F	-1	-1	-1	-1
22	F	r7	r7	r7	F	F	-1	-1	-1	-1

（2） 测试 2 及 LR(1)分析表

S->A d D
S->#
A->a A d
A->#
D->D d A
D->b
D->#

LR1 Table :

	Action					GOTO	
	a	b	d	#	A	D	S
0	S3	F	r4	r2	2	-1	1
1	F	F	F	A	-1	-1	-1
2	F	F	S4	F	-1	-1	-1
3	S3	F	r4	F	5	-1	-1
4	F	S7	r7	r7	-1	6	-1
5	F	F	S8	F	-1	-1	-1
6	F	F	S9	r1	-1	-1	-1
7	F	F	r6	r6	-1	-1	-1
8	F	F	r3	F	-1	-1	-1
9	S10	F	r4	r4	11	-1	-1
10	S3	F	r4	F	12	-1	-1
11	F	F	r5	r5	-1	-1	-1
12	F	F	S13	F	-1	-1	-1
13	F	F	r3	r3	-1	-1	-1

其他的测试样例在 testExample/parserTest 里，这里不一一展示。

3.7.2 语法树样例

测试样例：

```
void main()
{
    int a;
    int b;
    if(a>b || b<0)
    {
        a=a-b;
    }
    return;
}
```

图 3.1 语法树测试样例

输出结果：

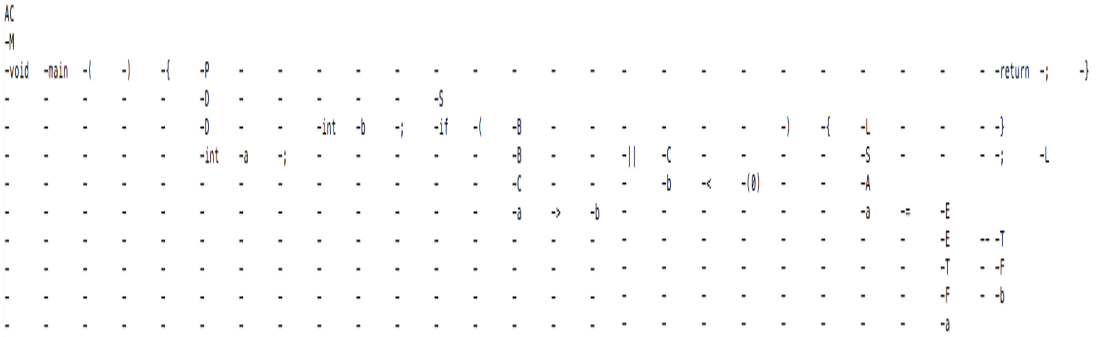

图 3.2　语法树图

4　综合测试

4.1　测试样例 1 及结果与解释

```
1  void main()
2  {
3      int a;     L0
4      int n;     L1
5      int i;     L2
6      {
7          a = 1 + 2 * 3;     L3 - L8
8          n = 5;             L9-L10
9          if( a >= n && a > 0 || n > 0)
10         {
11             a = 0;         L17-L18
12         }
13         else
14         {
15         }
16         ;
17         for(i = 0; i < n; i = i + 1)
18         {
19             a = a + 1 ;    L27-L29
20         }
21     }
22     return ;
23 }
```

```
ACC
L0   declare int - a
L1   declare int - n
L2   declare int - i
L3   num F := 1
L4   num F := 2
L5   num F := 3
L6   T L4 * L5
L7   E L3 + L6
L8   a := - L7
L9   num F := 5
L10  n := - L9
L11  if a>=n goto L13   如果a>=n接着判断a>0
L12  goto L16 - -       否则接着判断n>0
L13  if a>0 goto L17    如果a>=0执行a = 0
L14  goto L15 - -       否则接着判断n>0
L15  if n>0 goto L17    如果n>0执行a=0
L16  goto L19 - -       否则进入else,else为空执行下面的for
L17  num F := 0
L18  a := - L17
L19  num F := 0         for最开始另i = 1
L20  i := - L19
L21  if i<n goto L28    如果i<n执行a=a+1
L22  goto L31 - -       否则跳出循环
L23  num F := 1
L24  E L2 + L23         i = i+1
L25  i := - L24
L26  goto L21 - -       i返回判断是否i<n
L27  num F := 1
L28  E L0 + L27
L29  a := - L28
L30  goto L24 - -       返回执行i=i+1
L31  - - - -
```

图 4.1　测试样例 1 及结果与解释图

4.2 测试样例 2 及结果

该结果的四元式模式和测试样例 1 相似，L19-L22 是 else 语句中嵌套的 if 语句的四元式表示。

```
void main()
{
    int a;
    int n;
    int i;
    {
        a = 1 + 2 * 3;
        n = 5;
        if( a >= n && a > 0 || n > 0)
        {
            a = 0;
        }
        else
        {
            if(a > 3)
            {
                a = 6;
            }
        }
        ;
        for(i = 0; i < n; i = i + 1)
        {
            a = a + 1 ;
        }
    }
    return ;
}
```

```
ACC
L0   declare int - a
L1   declare int - n
L2   declare int - i
L3   num F := 1
L4   num F := 2
L5   num F := 3
L6   T L4 * L5
L7   E L3 + L6
L8   a := - L7
L9   num F := 5
L10  n := - L9
L11  if a>=n goto L13
L12  goto L16 - -
L13  if a>0 goto L17
L14  goto L15 - -
L15  if n>0 goto L17
L16  goto L19 - -
L17  num F := 0
L18  a := - L17
L19  if a>3 goto L21
L20  goto L23 - -
L21  num F := 6
L22  a := - L21
L23  num F := 0
L24  i := - L23
L25  if i<n goto L32
L26  goto L35 - -
L27  num F := 1
L28  E L2 + L27
L29  i := - L28
L30  goto L25 - -
L31  num F := 1
L32  E L0 + L31
L33  a := - L32
L34  goto L28 - -
L35  - - - -
```

图 4.2　测试样例 2 及结果图

4.3 测试样例 3 及结果

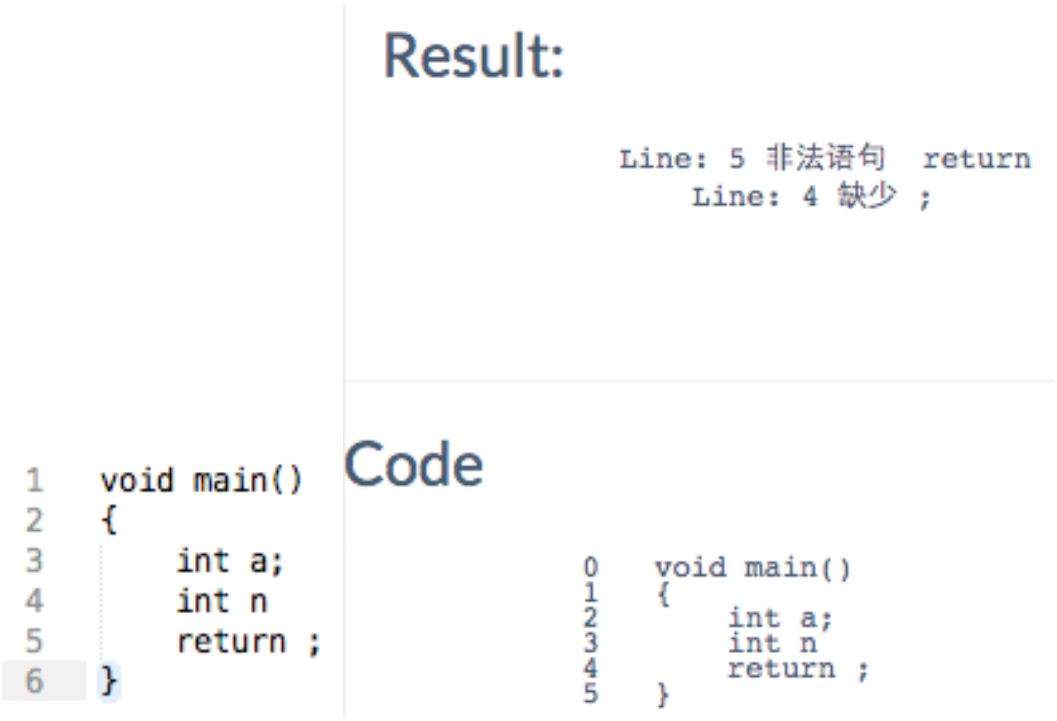

图 4.3 测试样例 3 及结果图

4.4 测试样例 4 及结果

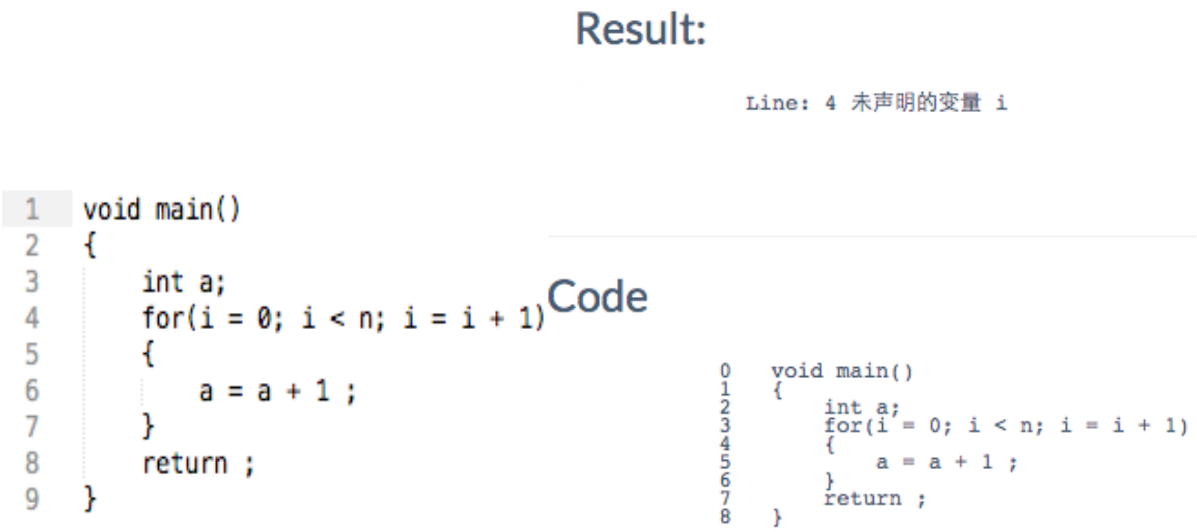

图 4.4 测试样例 4 及结果图

5 用户界面

5.1 程序概述

5.1.1 功能说明

用户界面能够实现提交代码，返回四元式结果，查看单词类别及其内部表示，查看语言定义，Goto 表，Action 表，完整的 LR(1)表和项目集规范簇。

表 5.1 功能描述

功能	描述
提交代码	用户提交自己写的代码，后台运行编译程序并返回结果
查看结果	将四元式结果返回给用户查看，并附带用户提交的代码
查看单词类别及内部表示	查看助记符及其对应的语法单位名称，如关键字、算数符号等。
查看语言定义	查看样本语言的定义
查看 Goto 表	查看语法语义分析时候用到的 Goto 表
查看 Action 表	查看语法语义分析时候用到的 Action 表
查看完整 LR(1)分析表	查看语法语义分析时候用到的完整 LR(1)分析表
查看项目集规范簇	查看语法分析时候生成的项目集规范簇

5.1.2 实现方法

为了方便不同操作系统的使用，我使用 Python 语言搭建了一个网页形式的用户界面。

该界面的后台运用 Python 的 Flask 框架，前端运用 HTML 配合已有的前端库 Bootstrap 实现。在运行编译程序时候通过 Python 调用系统命令，在 shell 中执行运行已经编译好的 C++程序。

5.2 界面概览

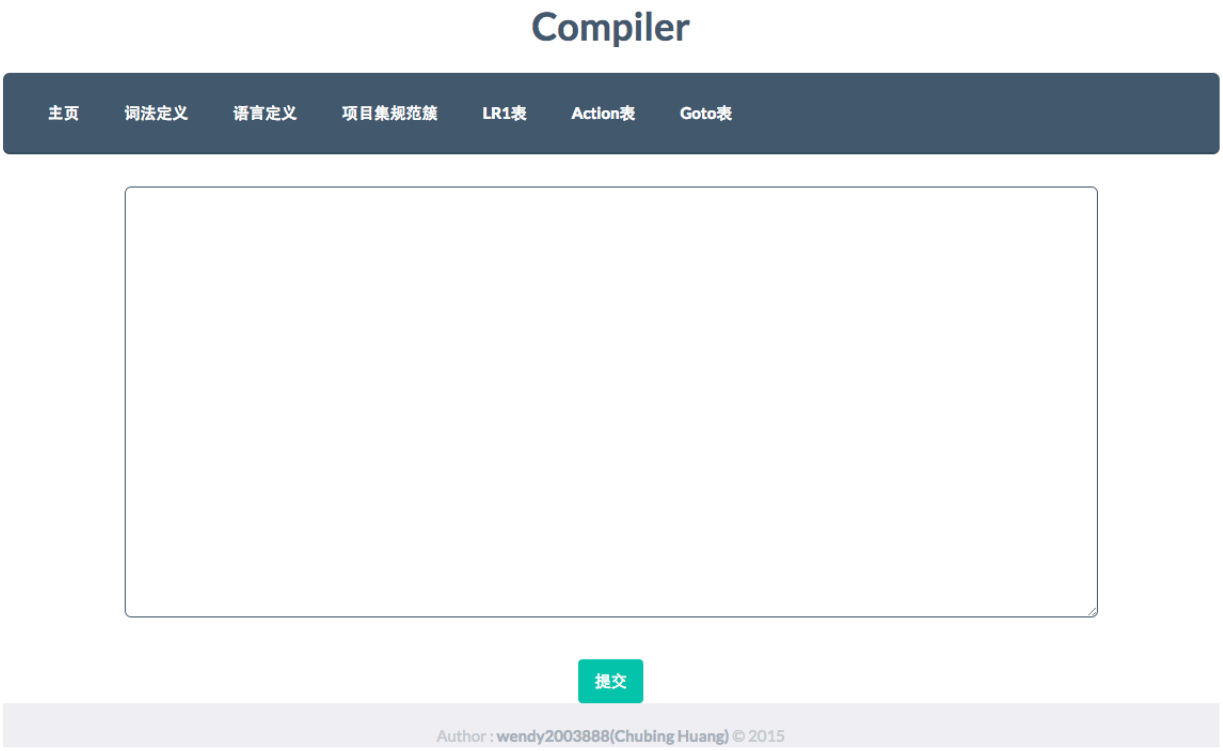

图 5.1 用户提交代码的主页面

助记符	单词符号
id	标识符
num	整数
for	for
while	while
do	do
if	if
else	else
void	void
const	const
return	return
main	main
include	include
type	int
type	double
type	float

图 5.2 查看单词类别及内部表示

图 5.3　结果页面(上)

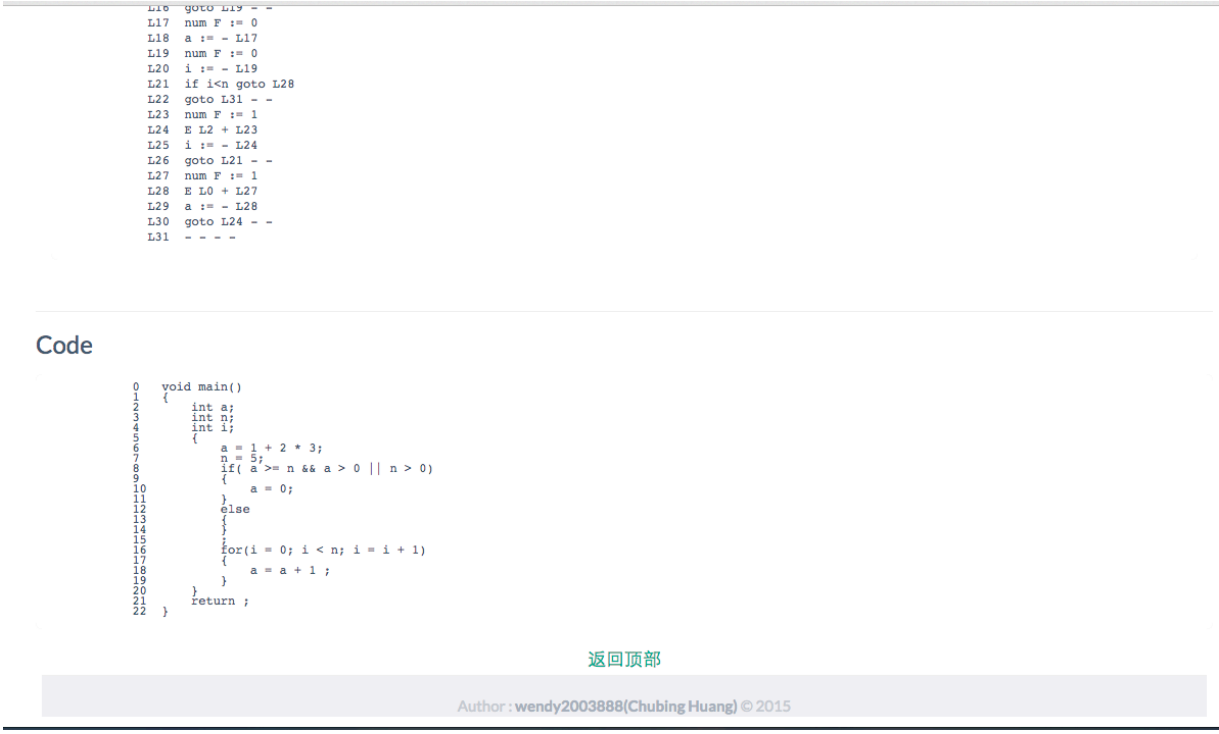

图 5.4　结果页面(下)

图 5.5　文法定义页面

图 5.6　Goto 表页面

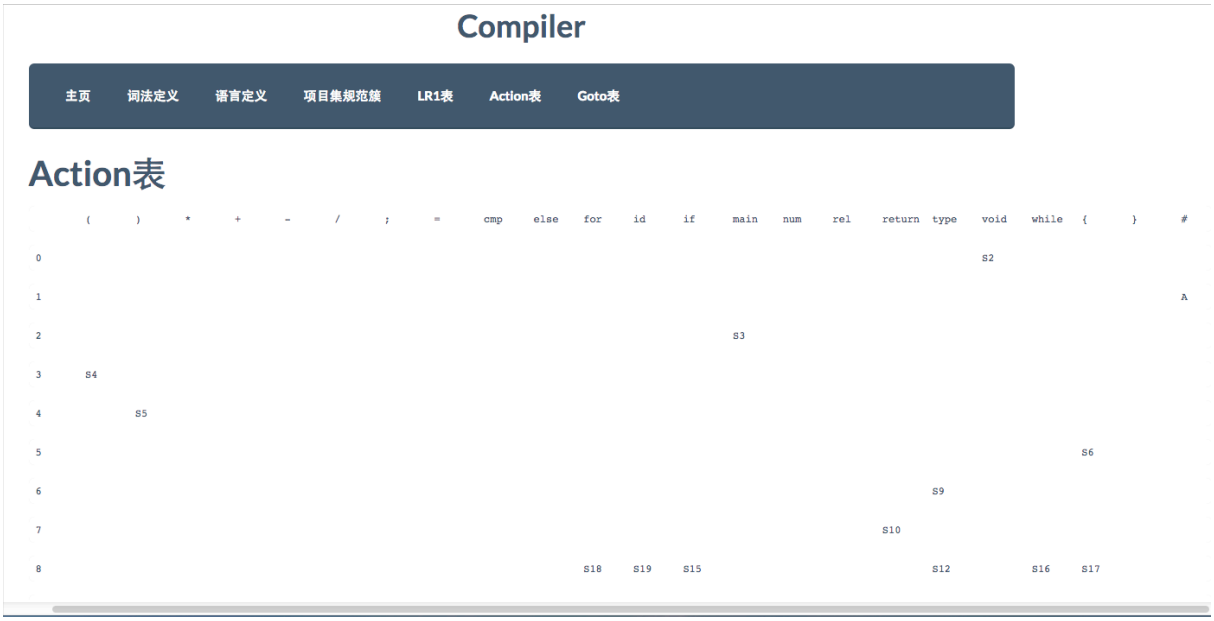

图 5.7　Action 表页面

图 5.8　项目集规范簇页面

6 程序说明

6.1 整体代码结构及介绍

CompilerWeb 文件夹包括用户界面的构成程序以及实现的编译器程序。

单独的编译程序在 Compiler 文件下，其中包括编译好的可执行程序 main，C++主程序 main.cpp，分模块 lexer.cpp、parser.cpp、semantic.cpp 和 syntaxTree.cpp，他们分别对应 lexer.h、parser.h、semantic.h 和 syntaxTree.h 头文件中的声明。definition.txt 文件是样本语言的定义文件，build.sh 可以重新编译并连接所有文件，如果代码更改可以运行 build.sh 重新编译并生成可执行程序。

用户界面的 Python 程序代码在 CompilerWeb 和 app 文件夹里。CompilerWeb 包括运行网页文件 start.py，初始化文件 init.py，配置文件 config.py，运行编译器程序文件 judge.py。app 文件夹包括初始化 app 包文件__init__.py，后台操作视图文件 view.py。

整体结构概览图如下：

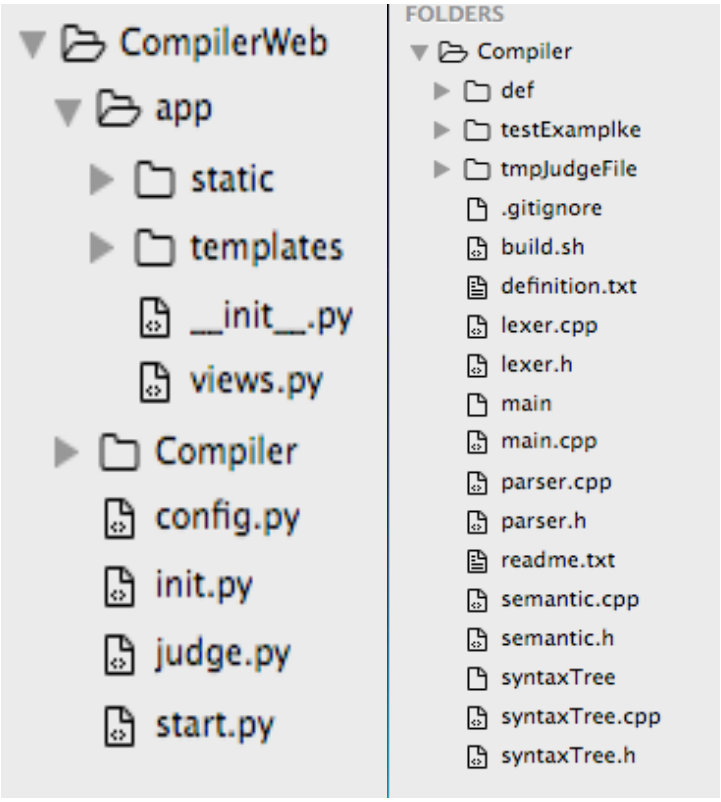

图 6.1 文件结构概览图

6.2 运行说明

安装 Python 2.7(.9)

pip install flask==0.10.1

在终端进入到 CompilerWeb 文件夹中,运行 python start.py。

访问 localhost:8080 即可看到用户界面

感　　想

通过这次编译原理课程设计，我能更好地理解编译器的工作原理，巩固并更深入地理解书本上的知识。

通过实现词法分析器，我学习了自动机的代码实现过程。在实现语法分析器的时候，我选择了自底向上的 LR(1)方法，该方法在课堂上并未做特别详细的讲解，并且相对于自顶向下的语法分析资料较少。通过自己学习书上的算法，查阅资料，询问老师，一步一步实现了自底向上的语法分析和语义分析，并且生成了四元式。

整个课程设计的过程锻炼了我的代码能力，提高了我查阅资料，自己学习，自己思考的能力。

最后，感谢老师，整个程序的完成离不开老师的指导和耐心的解答。

附录 A　数据结构及函数一览

```
struct Word
{
    string label, name;
    int line, type;
    long long val;
    Word(int ln = -1, string lb = "", string nm = "", long long v = 0);
    void print();
};
class Lexer {
public:
    int line;
    char ch;
    vector<Word> words;   //结果存储

    Lexer();       //构造函数
    void initial();       //初始化行号和关键字个数
    char remove_blank(char ch);       //移去空格，缩进，并计算行号
    char number(char ch);       //分析数字
    char word(char ch);       //分析单词
    char others(char ch);       //分析单词和数字以外的字符
    long long char2num(string num);       //将字符串转换成数字
    bool check_key(string s);       //检查是否是关键字
    bool judgeOpt(string s,int k);   //判断是否是运算符等
    void runLexer();   //运行词法分析器
    void print();       //输出结果
};
struct Rule {
    int id;   //规则编号
    string lelm;   //leftElement 左边的非终结符
    vector<string> relm;   //rightElement 右侧的规则符号
    Rule(int i = 0, string lel = "");       //构造函数
    void print();       //打印规则
```

```
        //运算符重载
        bool operator < (const Rule &x) const;
        bool operator == (const Rule &x) const;
        bool operator != (const Rule &x) const;
};
class Grammar {
public:
        const string ept;      //定义空 empty    "#"

        class Item {
        public:
                struct State
                {
                        int dot;
                        string ahead;
                        State(int d = 0, string s = "#")
                        bool operator < (const State &x) const
                        bool operator == (const State &x) const
                        bool operator != (const State &x) const
                };
                set<pair<Rule, State> > itemSet; //增广文法项目
                 //运算符重载
                bool operator < (const Item &x) const;
                bool operator == (const Item &x) const;
                bool operator != (const Item &x) const;
                void print();
        };

        set<string> vn;
        set<string> vt;
        string startSym;
        vector<Rule> rules;
        map<string, set<string> > first; //firstSet
        Item item;
        vector<set<pair<Rule, Item::State> > > closure; //项目集族
        map<int, map<string, int> > go_to;
```

```
    map<int, map<string, pair<char, int> > > action;

    Grammar();
    void initSyntex();  //初始化语法分析所用 Vn, Vt, First 集和 LR1 表
    map<string, set<string> > geneFirst();  //生成 first 集
    bool isVt(string s);  //判断是否是终结符
    bool hasNotVancancy(string s);  //判断是否空规则
    void initClosure();  //初始化项目集规范簇
    void initTable(int x);  //初始化每个规范簇的表
    set<string> getFirst(string s, string ahead);  //在生成项目集规范簇时取得后面字符和检查字符的 first 集
    void geneSet();  //生成项目集闭包
    void geneClosure();  //生成项目集规范簇，用于事先的 debug
    int findSet(set<pair<Rule, Item::State> > s);  //检查当前生成的项目集闭包是否已经存在
    bool eqlSet(const set<pair<Rule, Item::State> > &a, const set<pair<Rule, Item::State> > &b);  //判断 2 个项目集闭包是否相等
    void buildLR1table();  //生成项目集规范簇的同时生成 goto 和 action 表
    int getRuleNum(Rule r);  //找到需要规约时候的规则号，用于生成 action 表
    void print();  //打印与输出 first 集，
};
struct MidCode {
    int id;
    string data[4];
    MidCode(int x = 0, string a = "-", string b = "-", string c = "-", string d = "-");
    void print();
};
class syntaxTree {
public:
    struct TreeNode {
        string label;
        int pos, size;
        TreeNode *fa;
        //属性信息
```

```
        string name, type;
        long long val;
        int line, place, tr, fl;

        vector<TreeNode*> child;
        TreeNode(string lb, int p = 0, int sz = 0, vector<TreeNode*> ch =
vector<TreeNode*>(0), int ln = -1);
        void setFa(TreeNode *f);
        void emit(string s, int t, int f);
    };

    int labels;
    TreeNode *root;

    syntaxTree();
    int newLabel();
    void print(TreeNode *root);
};
class Semantic {
public:
    map<string, string> typeTable;
    map<string, syntaxTree::TreeNode*> placeTable;
    map<int, MidCode> Code;

    Semantic();
    bool semantic(vector<Word> input, syntaxTree &synTree, Grammar G);  //语义分
析程序
    string num2Char(int a);  //将数字转换成字母
    void backpatch(syntaxTree::TreeNode &a, int place, int flag);  //拉链回填
    void emitMidCode(syntaxTree synTree);  //输出 4 元式
    bool start(vector<Word> input, syntaxTree &synTree, Grammar G);  //开始语法
语义分析并生成语法树和中间代码(四元式)
};
```