编译原理课程实验报告

大作业

姓名	张恒		院系	计算机		学号	1160300620	
任课教师	辛明影			指导教师	辛明影			
实验地点	格物 208			实验时间	2019.5.1			
实验课表现	出勤、表现得分			实验报告得分		实验总分		
	操作结果得分							

一、需求分析 得分

本次大作业主要工作是整合词法分析、语法分析、语义分析三个模块，使这三个模块成为一个完整的应用程序。除此之外，本次大作业中，我额外添加了一些简单的功能，使得词法分析器、语法分析器、语义分析器三个模块更加完整。这些额外的工作包括：

（1） 数组的定义与引用；
（2） 过程定义与调用

下面就是整合的内容。

首先是作为一个完整的应用程序，需要完成的功能：
（1） 能够以一个友好的用户界面与用户交互；
（2） 用户能够自由选择是使用词法分析、语法分析还是语义分析。
（3） 能够优雅的退出程序

词法分析器应该完成的功能：
（1） 能够以文件方式输入源程序；
（2） 能够展示分词分析的结果，并且能够保存这个结果，用于语法分析阶段；词法分析器的输入和输出的格式如图 1-1 所示：

- **输入**
  - **while(num!=100){num++;}**
- **输出**

```
while    <WHILE,  _   >
(        <  SLP ,  _   >
num      <  IDN ,  num >
!=       <   NE ,  _   >
100      <CONST,  100 >
)        <  SRP ,  _   >
{        <   LP ,  _   >
num      <  IDN ,  num >
++       <  INC ,  _   >
;        < SEMI ,  _   >
}        <   RP ,  _   >
```

图 1-1 词法分析器的输入与输出实例

（3） 界面需要美观、人性化，具有良好的演示效果；

（4） 能够根据词法规则识别以及组合单词。这里的单词包括五类，分别为关键字，也称为基本字；标识符，由用户定义，表示各种名字；常数，整常数、实常数、布尔常数等；运算符，即算数运算符、逻辑运算符、和关系运算符；分界符，逗号、句号、括号等；

（5） 对数字常数完成数字字符串到二进制数值的转换；

（6） 查填符号表。在词法分析阶段，主要是构建符号表，这里不要求输出完整的符号表，能够生成符号表供下一阶段使用即可；

（7） 删去空格字符和注释；

（8） 错误检查。词法分析阶段的错误检查主要是检查词法错误，即非法的字符。本词法分析器要求做简单的不封闭错误检查；

（9） 关于词法分析器输出的 token 序列，对于标识符对应的属性值应当是符号表中该标识符的入口地址，在本词法分析器中暂不要求分配具体地址；

（10） 词法分析器应当基于 DFA 技术；

语法分析器应该完成的功能：

（1） 使用句法分析技术 LR（1）对类高级语言中的基本语句进行句法分析；

（2） 语法分析(syntax analysis)是编译程序的核心部分，其任务是检查词法分析器输出的单词序列是否是源语言中的句子，亦即是否符合源语言的语法规则。语法分析器的功能位置如图 1-1 所示：

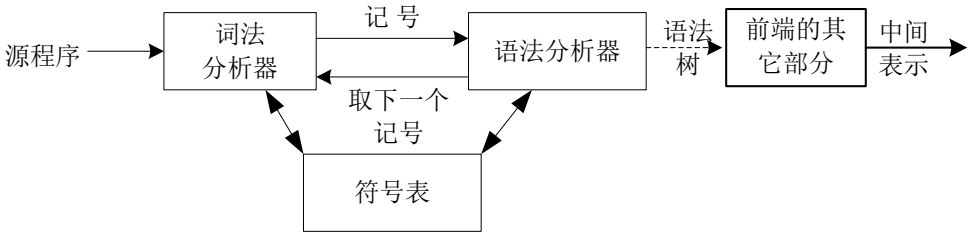

图 1-1 语法分析器的功能位置

（3） 能识别几类基本的语句，即声明语句（变量声明）、表达式及赋值语句（简单赋值）、分支语句（if_then_else）以及循环语句（do_while）；

（4） 在随意给出一个文法的情况下，能够自动计算相应的闭包等，并且构造出 LR（1）分析表；

（5） 具备简单语法错误处理能力，能准确给出错误所在位置，并采用可行的错误恢复策略。输出的错误提示信息格式为：Error at Line [行号]：[说明文字]；

（6） 能够通过文件导入文法和测试用例，其中测试用例要能够涵盖前面所说的四种语句，并需要设置一些语法错误；

（7） 系统的输出部分，需要打印出语法分析器的 LR（1）分析表，以及语法分析的结果，即规约时的产生式序列；

语义分析器应该完成的功能：

（1） 语法分析系统的核心任务，是在语法分析的基础上，检查源代码是否符合语义规范，并能够对不符合语义规范的错误进行提示，对符合语义规范的代码生成中间代码表示；

（2） 具体来说，本语义分析系统还需要满足：能够以文件的方式导入源代码；
（3） 能够识别声明语句、表达式及赋值语句、分支语句和循环语句；
（4） 具备语义错误处理能力，包括变量或函数重复声明、变量或函数引用前未声明、运算符和运算分量之间的类型不匹配（如整型变量与数组变量相加减）等错误，能准确给出错误所在位置，并采用可行的错误恢复策略；
（5） 能够输出符号表；
（6） 能够输出代码的中间代码表示，比如三地址指令或者是四元式；
（7） 能够完成数据类型的自动转换；

二、文法设计

得分

文法设计包括三个部分，分别是词法分析阶段的词法规则、语法分析阶段设计的一个用于测试的 LR（1）文法，以及语义分析阶段添加了语法制导翻译模式的文法。

（1） 词法分析阶段

词法规则

词法规则主要包括六个部分，即关键字、标识符、常数、运算符、界符和注释。其中，关键字和逻辑运算符都是标识符中的特殊部分，这个部分被称为保留字。在具体实现的时候，采用优先判断是否为保留字，来区分保留字和普通的标识符。词法规则可以用正则文法描述如表 1 所示：

规则	说明
S→keyword|indentifier|arithmetic|logistic|const|note|delimiter|compare	词法的总规则，由 8 类词组成
keyword→int|float|bool|if|else|do|while	关键词组成规则，包含在标识符组成规则中，实现时，采用优先识别关键字的方式来区分这两个情况
letter→a|b|c|…|z|A|B|C|…|Z digit→0|1|2|3|4|5|6|7|8|9 identifier→(letter|_)(letter|digit|_)*	标识符组成规则，
arithmetic→+|-|*|/|=	算数运算符和赋值号规则
logistic→and|or|not	逻辑运算符组成，包含在标识符组成规则中，实现时，采用优先识别关键字的方式来区分这两个情况
compare→>|>=|<|<=|==	比较运算符的词法规则
const→consti|constf consti→digit(digit)* constf=digit(digit)*.(digit)*	常数的词法规则，包括正整数和浮点数
delimiter→,|;|(|)|{|}|:	界符的词法规则
note→/*…*/	这个表达式中，表示由两个“/*”个中间的任何字符组成的字串都是注释

表 1 词法规则

单词的编码表

各类单词的编码方案如表 2 所示。

单词	种别码	标记符	属性值
错误字符	1	ERROR	错误的字符
int	2	INT	-
float	3	FLOAT	-
bool	4	BOOL	-
record	5	RECORD	-
if	6	IF	-
else	7	ELSE	-
do	8	DO	-
while	9	WHILE	-
+	10	ADD	-
-	11	SUB	-
*	12	MUL	-
/	13	DIV	-
<>	14	NE	-
>	15	G	-
<	16	L	-
>=	17	GE	-
<=	18	LE	-
==	19	E	-
and	20	AND	-
or	21	OR	-
not	22	NOT	-
=	23	ASSIGN	-
;	24	SEMI	-
,	25	COMMA	-
(letter|_)(letter|digit|letter|_)*	26	ID	符号表相应地址
(	27	LB	-
)	28	RB	-
{	29	LCB	-
}	30	RCB	-
:	31	COLON	-
/*…*/	32	NOTE	注释字符串
digit(digit)*	33	CONSTI	相应整数
digit(digit)*.(digit)*	34	CONSTF	相应浮点数
true	35	TRUE	-
false	36	FALSE	-

表 2 单词的编码表

识别各类单词的 NFA

关键字的识别比较麻烦，因为关键并没有规律可言，只能一个一个识别。

识别关键字和逻辑运算符（即保留字）的 NFA 如图 2-1 所示：

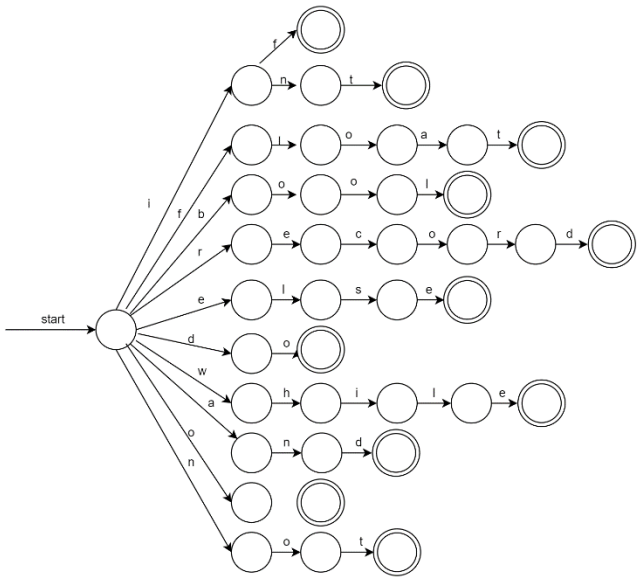

图 2-1 识别保留字的 NFA

识别标识符的 NFA 如图 2-2 所示：

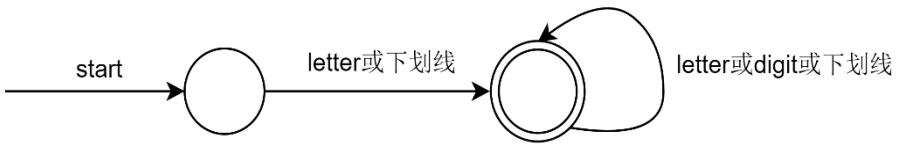

图 2-2 识别标识符的 NFA

识别正整数和浮点数常数的 DFA 如图 2-3 所示：

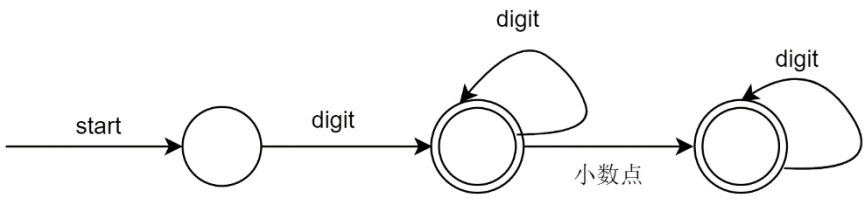

图 2-3 识别常数的 NFA

识别其余有效字符的 NFA 如图 2-4 所示：

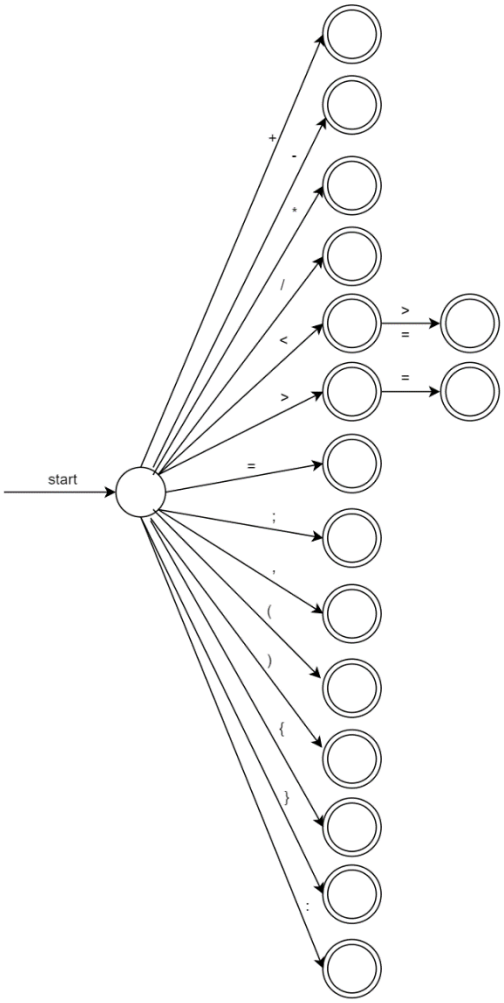

图 2-4 识别其余有效字符的 NFA

识别注释的 NFA 如图 2-5 所示：

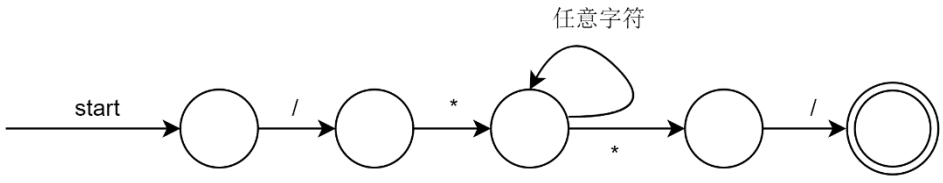

图 2-5 识别注释的 NFA

有了 NFA 后，就可以很简单的得到每个转换图的 DFA，比如将每个非法输入引入到一个错误状态，识别为错误字符即可。得到 NFA，就可以很容易进行词法分析了。

（2） 语法分析阶段

语法分析器采用 LR（1）分析方法，并且采用自动构建 LR（1）分析表的方式，因此，理论上来说，本语法分析器能够自动处理任意一种 LR（1）文法，并根据给定的文法对源程序进行语法分析。

如表 3 所示，是一个用于测试的 LR（1）文法。对于文法的定义有如下的说明：

① 文法中的非终结符均包含的字母符号均是大写字母，终结符包含的字母符号均是小写字母；

② 文法中每一行只包含一个推导式；

③ 如果一个推导式中包含有多个符号（终结符或者非终结符），符号间用空格隔开；

④ 空串用“epsilon”表示；

⑤ 文法必须是拓广文法；

说明	产生式
程序总结构。 描述程序中几种结构的语句可能的顺序	PP->P P->D P->S P->D S S->S S
声明语句	D->D D D->proc id ; D S D->T id ; T->X C T->record D X->int X->float X->bool C->[consti] C C->epsilon
表达式及赋值语句	S->id = E ; S->L = E ; E->E + T E->E - T E->T T->T * F T->F T->id T->consti T->constf T->L L->id [E] L->L [E]
	S->UNMATCHS S->MATCHS

控制语句，即 条件语句 循环语句	MATCHS->if (B) then MATCHS else MATCHS UNMATCHS->if (B) then S UNMATCHS->if (B) then MATCHS else UNMATCHS S->while B do S B->B or B B->B and B B->not B B->(B) B->E RELOP E B->true B->false RELOP->< RELOP-><= RELOP->== RELOP->!= RELOP->> RELOP->>=
过程调用	S->call id (ELIST) ELIST->ELIST , E ELIST->E

表 3 一个用于测试的 LR（1）文法

（3） 语义分析阶段

语义分析阶段需要一个添加了语法制导翻译模式的文法，因此不能像语法分析模块一样可以自己选择文法，这里需要固定一个文法。基本文法沿用语法分析模块中的测试文法，在此文法的基础上添加语法制导翻译模式。结果如表 4 所示。

表达式类型	文法	语法制导翻译模式
文法的入口	P->S	offset=0;
声明语句	S->D	
	D->D D	
	D->T id ;	entry(id.lexeme, T.type, offset);
	T->int	T.type=int;
	T->float	T.type=float;
	T->bool	T.type=bool;

	S->id = E ;	p=lookup(id.lexeme);
		if p == nil then error;
		gen(p ‘=’ E.addr);

	S->L = E ;	gen(L.array ‘[’ L.offset ‘]’ ‘=’ E.addr);
	E->E1 + EE	E.addr=newtemp; gen(E.addr ’=’ E1.addr+EE.addr);
	E->E – EE	E.addr=newtemp;

表达式及赋值语句		gen(E.addr ’=’ E1.addr-EE.addr);
	E->EE	E.addr=EE.addr;
	EE->EE1 * EEE	EE.addr=newtemp; gen(EE.addr ‘=’ EE1.addr*EEE.addr);
	EE->EEE	EE.addr=EEE.addr;
	EEE->(E)	EEE.addr=E.addr;
	EEE->L	E.addr=newtemp(); gen(E.addr ‘=’ L.array ‘[’ L.offset ‘]’);
	EEE->consti	EEE.addr=loolup(consti.lexeme); if EEE.addr == nil then error;
	EEE->constf	EEE.addr=loolup(constf.lexeme); if EEE.addr == nil then error;
	EEE->id	EEE.addr=loolup(id.lexeme); if EEE.addr == nil then error;
	L->id [E]	L.array=lookup(id.lexeme); if L.array==nil then error; L.type=L.array.type.elem; L.offset=newtemp(); gen(L.offset ‘=’ E.addr ‘*’ L.type.width);
	L->L1 [E]	L.array=L1.array; L.type=L1.type.elem; t=newtemp(); gen(t ‘=’ E.addr ‘*’ L.array.width); L.offset=newtemp(); gen(L.offset ‘=’ L1.offset ‘+’ t);
控制流语句	S->S1 M S2	backpatch(S1.nextlist, M.quad); S.nextlist=S2.nextlist;

	S->if (B) then M1 { S1 } N else M2 { S2 }	backpatch(B.truelist, M1.quad;
		backpatch(B.falselist, M2.quad;
		S.nextlist=merge(merge(S1.nextlist,
		N.nextlist), S2.nextlist);

	N->ε	N.nextlist=makelist(nextquad); gen(‘goto _’);
	M-> ε	M.quad=nextquad;

	S->while M1 (B) do M2 { S1 }	backpatch(S1.nextlist, M1.nextquad;
		backpatch(B.truelist, M2.nextquad;
		S.nextlist=B.falselist;
		gen(‘goto’ M1.nextquad);

	B->B1 or M H	backpatch(B1.falselist, M.quad); B.truelist=merge(B1.truelist, H.truelist); B.falselist=H.falselist;
	B->H	B.truelist=H.truelist; B.falselist=H.falselist;
		backpatch(H1.truelist, M.quad);

布尔表达式	H->H1 and M I	H.truelist=I.truelist; H.falselist=merge(H1.falselist, I.falselist);
	H->I	H.truelist=I.truelist; H.falselist=I.falselist;
	I->not I1	I.truelist=I1.falselist; I.falselist=I1.truelist;
	I->(B)	I.truelist=B.truelist; I.falselist=B.falselist;

	I->EEE1 RELOP EEE2	I.truelist=makelist(nextquad);
		I.falselist=makelist(nextquad+1);
		gen('if' EEE1.addr RELOP EEE2.addr 'goto _');
		gen('goto _');

	I->true	I.truelist=makelist(nextquad); gen('goto _')
	I->false	I.truelist=makelist(nextquad) gen('goto _')
	RELOP-><	
	RELOP-><=	
	RELOP->>	
	RELOP->>=	
	RELOP->==	
	RELOP->!=	

过程调用	S->call id (ELIST)	n=0;
		for q 中的每个 t do
		{
		gen('param' t);
		n=n+1;
		}
		gen('call' id.addr ',' n);

	ELIST->E	将 q 初始化为只包含 E.addr;
	ELIST->ELIST1, E	将 E.addr 添加到 q 的队尾;

表 4 语法制导翻译模式

三、系统设计

得分

系统实际上是分为三个模块组合而成，每个模块都是独立出来的，但是相互之间又有联系。具体来说，是语义分析器包含了语法分析器，语法分析器又包含了词法分析器，因此分别对三个模块进行讨论，会显得十分冗余，这里直接考虑整体的结构。

本程序的主要功能为对给定的源代码进行词法、语法、语义分析，同时，也提供单独的词法、语法或是语义分析模块，可以决定之分析到某一步骤就停止，而不必做完整的分析。

（1） 系统概要设计

系统框架图

系统的框架如图 3-1 所示。用户界面是用户与分析器的交互媒介，程序的三个功能，词法分析、语法分析、语义分析，用户可分别通过调用词法分析模块、语法分析模块和语法分析模块来完成。

语法分析和语义分析是同步实现的，即通过一边扫描，同时实现语法分析和语义分析，并且是以语法分析器为核心，因此可以看到语法分析器在分析过程中，会调用语义分析器。在这里还需要一个辅助的模块，即操作系统提供的文件选择 API。

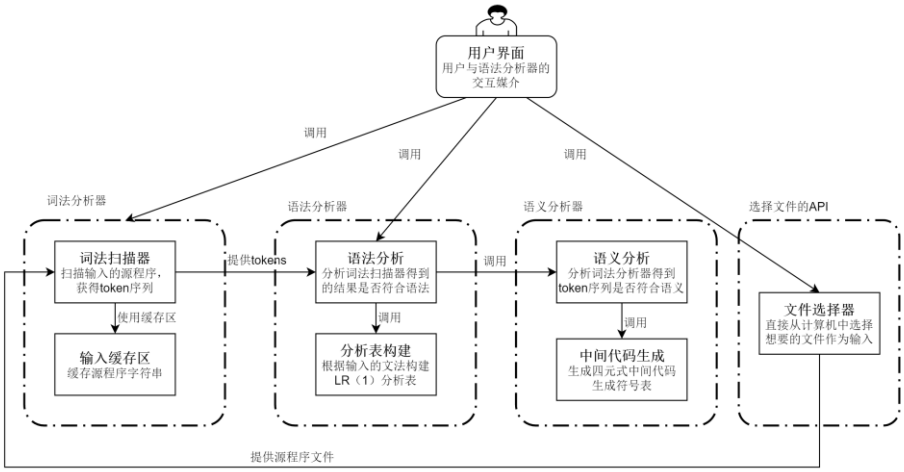

图 3-1 系统框架图

数据流图

在使用本程序提供的三个不同的功能时，程序拥有不同的数据流。其中，做词法分析时最简单，数据流的起始是用户输入的源程序，经过词法分析器的处理，得到输出给用户的数据流，Token 序列。因此，在这里对词法分析的数据流不赘述。

语法分析的数据流如图 3-2 所示。此时程序的数据流起始与用户的输入，先分别用此法处理程序处理源程序输入流，得到 token 流，用语法处理程序中的构建 LR（1）分析表的模块得到 LR（1）分析表。然后语法分析器的分析模块处理这两个输入流就能得到结果。结果可能会有两种情况，在分析成功时，得到的是分析中的规约式；分析失败时，得到的是分析中得到的错误信息。

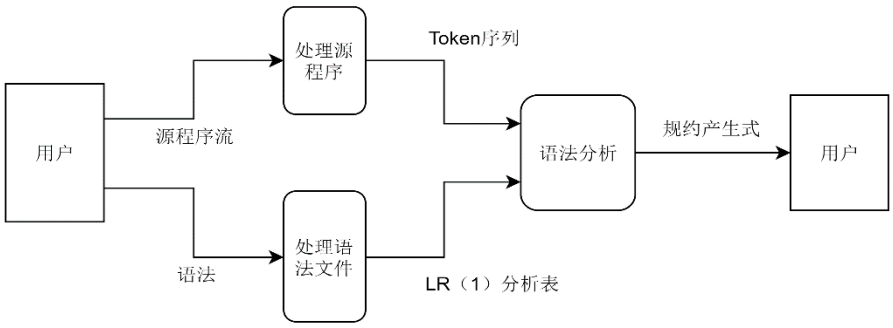

图 3-2 语法分析器的数据流图

语义分析的数据流如图 3-3 所示，此时程序的数据流起始于用户的输入，先分别用词法分析程序处理源程序得到 Token 序列，用语法分析程序处理语法规则文件得到 LR（1）分析表，然后利用语法分析模块处理这两个数据，得到一系列推导式，通过这些推导式可以得到源程序。然后数据流 Token 序列和推导式到达语义分析程序，语义分析程序据此得到四元式的中间代码和符号表。

实际上，在本程序中，语义分析和语法分析实际上是一起进行的，具体来说，是语法分析器处理一部分 token 后，就将结果交给语义分析器处理。这样，实际上语义分析的工作是在语法分析之后，因此我将其抽象成如图 3-3 所示的样子，便于理解和展示。

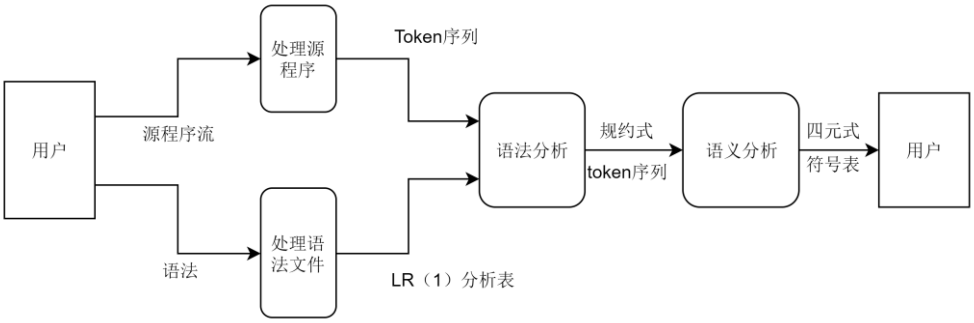

图 3-3 语义分析器的数据流图

功能模块图

程序的模块如图 3-4 所示。因为之前说到的三个功能之间的包含关系，所以程序的模块在三个功能中有很大程度的共用，图 3-4 中仅展示程序拥有的模块。程序主要分为三个模块，一是词法分析器，一是语法分析器，一是语义分析器。最后，还需要一个辅助的模块，即文件选择模块。

每个模块都完成各自的功能，并为下一层的模块提供接口，共同完成中间代码生成的任务。

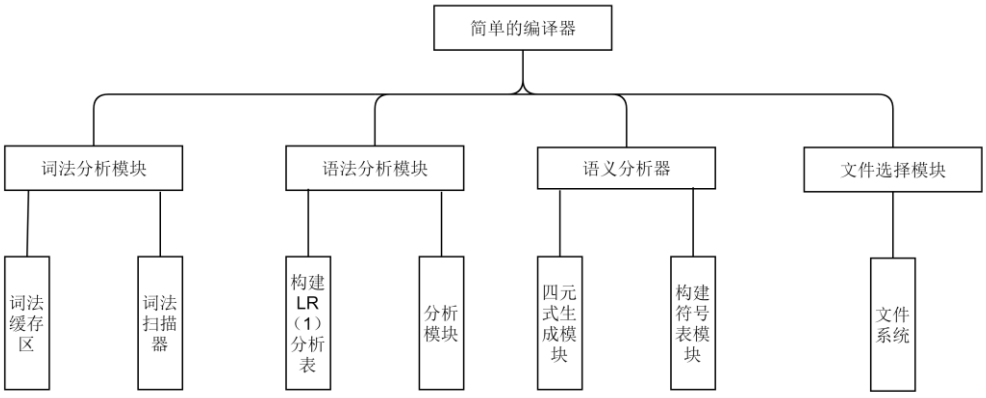

图 3-4 程序的功能模块图

（2） 系统详细设计

核心数据结构的设计

程序的结构可以分为三个模块，下面依次分模块介绍程序中涉及到的核心数据结构。

词法分析：

输入序列缓存区。输入缓存区的主要功能就是缓存源程序，并记录程序中的字符在源文件中位置，以便后续的分析能够定位错误的位置。我新建了一个 LexicalBuffer.java 的类来表示缓存区，这个的属性设置如图 3-5 所示：

```
private char[] buf = new char[BUF_SIZE]; // 缓存区数组
private int rawNum = 1; //  当前处理的行号
private int colNum = 1; //  当前处理的列号
private int current = 0; // 当前处理到的字符位置
private int len; // 缓存数组中有效数据的长度
private BufferedReader reader = null; // 读取文件的reader
private boolean _available; // 标记文件是否读完
```

图 3-5 词法缓存区的属性

词法分析器输出的 Token。词法分析器每分析出一个单词，就做成一个 token，token 的原定义是一个二元组，即单词的编码（或是编码的助记符）和对应的属性值。为了记录 token 在源文件中的位置，我添加了一个行号的属性。我建立一个 LexicalToken.java 类来表示这个数据结构，数据结构的属性如图 3-6 所示。

```
private StringProperty word; // 合法的 token 序列
private StringProperty category; // token 序列对应的类别标记符
private StringProperty value; // token 序列对应的属性值
private int lineNum = -1;   //  Token 出现的行号
```

图 3-6 Token 的属性

语法分析：

文法的表示。文法的表示包括推导式、终结符集合和非终结符集合。在构建 LR（1）分析表时，需要用表格的形式来存储，这就需要将每个终结符和非终结符以及推导式对应于一个唯一的序号，因此采用顺序表的方式，用 list 实现。其中每个符号用一个 String 表示。

推导式和 LR 项目的表示。LR 的项目可以分为四个部分，推导式左部、推导式右部、圆点位置和后继符号集合。前两者是不需要改变的，但是后两者需要频繁改变，因此我新建了一个数据结构用来表示一个项目，并且将文法中的推导式视为一个特殊的项目。这个数据结构的属性列表如图 3-7 所示：

```
private String left; // 左部非终结符
private List<String> right = new ArrayList<String>(); // 右部符号集合
private int dotPosition; // 圆点的位置
private Set<String> successor = new HashSet<String>();  //  后继符号集合
```

图 3-7 LR（1）项目的数据结构的属性

FIRST 集的表示。在计算时会频繁用来某个非终结符的 FIRST 集，因此我用字典，也就是 java 中的 Map 来实现非终结符和其 FIRST 的对应。并且考虑到 FIRST 需要频繁的做一些集合的操作，且不需要在意集合中符号的顺序，因此每个 FIRST 集我用一个集合 Set 来表示。

项目集族的表示。对每个项目集，可以不用关心其中每个项目的顺序，因此我用集合 Set 来表示一个项目集。但是在分析表中，需要得到每个项目集和一个唯一序号的对应，因此项目集族就是项目集的一个有序列表，用 List 实现。

分析表。分析表是一个二维的表，其中涉及到集中数据结构，这是难以实现的，因此我将涉及到的数据全部映射为一个整数。也就是说，我用一个 int 类型的二维数组来表示分析表，其中的行是项目集的编号，列是非终结符或者是终结符的编号。具体来说，我将分析表分为 action 表和 goto 表两个表。

整个语法分析器涉及到的数据结构如图 3-8 所示。

```
public static int acc = 100000; //  接收状态的编码
private List<String> terminals = new ArrayList<String>();   //  终结符的集合
private List<String> varibles = new ArrayList<String>();    //  非终结符的集合
private List<LR1Item> grammers = new ArrayList<LR1Item>();  //  推导式额集合
private Map<String, Set<String>> firsts = new HashMap<String, Set<String>>();   //  非终结符和FIRST集的对应
private List<Set<LR1Item>> LRfamily = new ArrayList<Set<LR1Item>>();    //  项目集族
private int[][] actionTable;    //  action 表
private int[][] gotoTable;  //  goto 表
private List<String> errors = new ArrayList<String>();  //  语法分析的错误信息列表
private List<LR1Item> rules = new ArrayList<LR1Item>(); //  语法分析中的规约式集合
```

图 3-8 语法分析器设计到的数据结构概览

语义分析：

四元式的中间代码。这是中间代码的一种表现形式，共分为四个部分，第一部分是操作类型，第二部分是参数一，第三部分是参数二，第四部分是结果。在实现时，我新建了一个类，用如图 3-9 所示的内部属性表示一个四元式。

```
private int offset; //  代码的首地址
private String actionType = null;   //  操作类型
private String arg1 = null; //  参数一
private String arg2 = null; //  参数二
private String result = null;   //  结果
```

图 3-9 四元式的表示

符号表的表示。符号表中可以包含各种类型，我这里在符号表中存入了常数和变量。符号表整体由一个字典组成，关键在于每一个表项的表示。这里，我新建了一个类，用于表示一条表项。每条表项由四部分组成，一是关键字，二是符号类型，三是符号对应的变量的类型，四是在符号表中偏移量。如图 3-10 是该类的内部属性表示。

```
private String name;     //  关键字
private String category;     //  类别
private String type;     //  对应的变量类型
private int addr;    //  偏移量，也就是地址
```

图 3-10 符号表项的表示

除了上面的两个最重要的数据结构，还有一个数据结构也值得一提，那就是在语义分析过程中，代表中间节点的变量。每个中间节点可能包括不同的属性，如代表布尔表达式的节点拥有 truelist 和 falselist 等属性，我设计了一个 Variable.java 类来表示一个节点，类的属性如图 3-11 所示。

```
private String name;     //  节点代表的名字，如变量的名称
private String type = null; //  节点代表的变量的类型
private String value = null;     //  节点的值
private String begin;    //  节点代表的代码块的起始位置
private String next;     //  节点代表的代码块之后的位置
private List<Integer> nextList=new ArrayList<Integer>();     //  节点的 nextlist 列表
private List<Integer> trueList=new ArrayList<Integer>();     //  节点的 truelist 列表
private List<Integer> falseList= new ArrayList<Integer>();  //  节点的 falselist 列表
```

图 3-11 节点的表示

主要功能函数说明

下面依次分模块介绍程序中涉及到的核心数据结构。

词法分析：

next。LexicalBuffer 中的函数，用以读取下一个字符；

rollback。LexicalBuffer 中的函数，用以回滚一个字符；

removeSpace。LexicalScanner 的函数，用以出去连续的空格；

recognizeV。LexicalScanner 的函数，用以识别保留字和标识符；

recognizeN。LexicalScanner 的函数，用以识别常数；

recognizeO。LexicalScanner 的函数，用以识别其他的有效字符；

语法分析：

private void computeFirsts()。构建非终结符的 FIRST 集。

private Set<LR1Item> computeClosure(LR1Item item)。计算给定项目的项目集闭包。

private void computeLRfamily()。计算项目集族。

public void buildAnalysisTable(String grammer)。构建所给文法的 LR（1）分析表。

public void parse(List<LexicalToken> tokens)。分析所给 token 序列是否符合文法规则。

语义分析：

语义分析器的核心在于一个函数，其定义如下：

public void execAction(int grammerNum);

其中参数 grammerNum 是规约式对应的序号，程序内部有一个规约式与相应的语义动作的对应，函数会根据规约式执行相应的语义动作。

程序核心部分的流程图

程序主要外层的流程是一个模块选择，就是选择使用的功能。这个流程比较简单，下面就分别说说几个功能的流程。

词法分析的功能流程如图 3-12 所示。

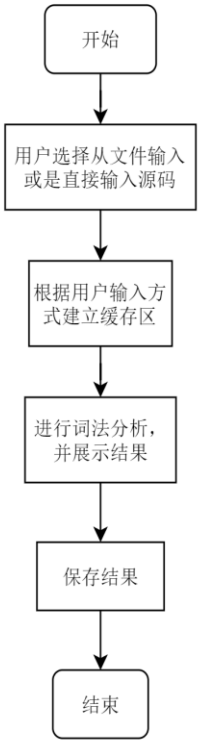

图 3-12 系统每次词法分析的流程图

语法分析的流程如图 3-13 所示。

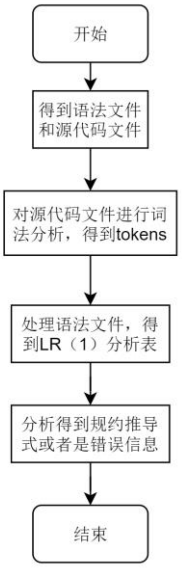

图 3-13 语法分析的流程图

语义分析的流程如图 3-14 所示。

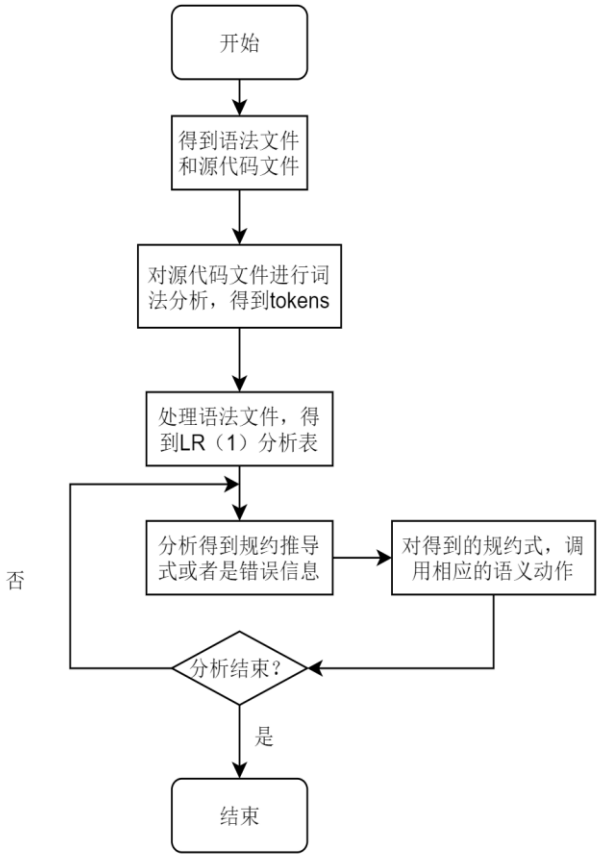

图 3-14 语义分析的流程图

四、系统实现及结果分析	得分	

（1） 测试结果

如图 4-1 是程序的初始化界面。

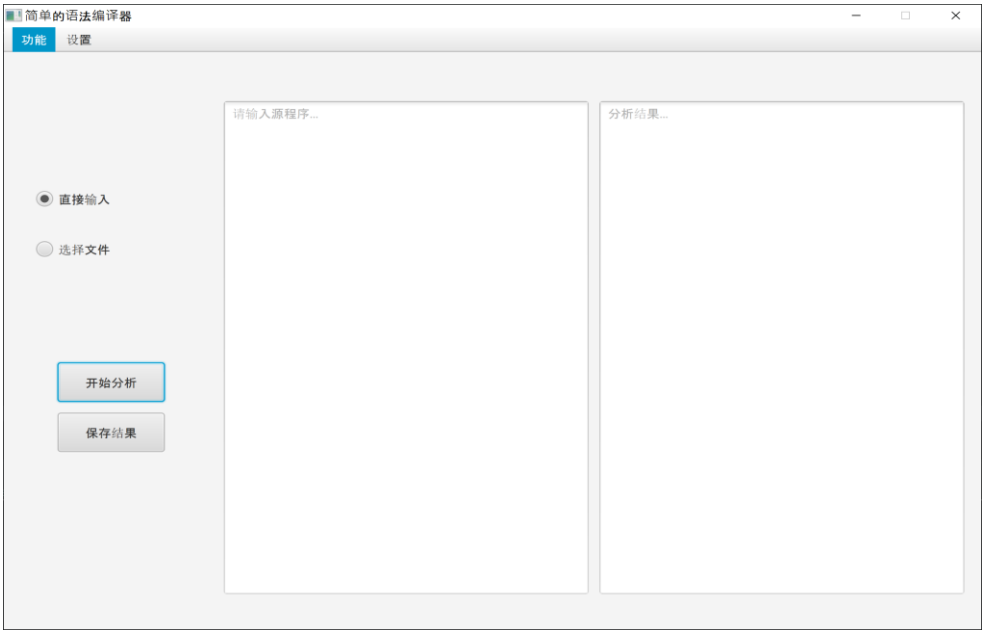

图 4-1 程序的初始化界面

如图 4-2，是词法分析的测试结果。

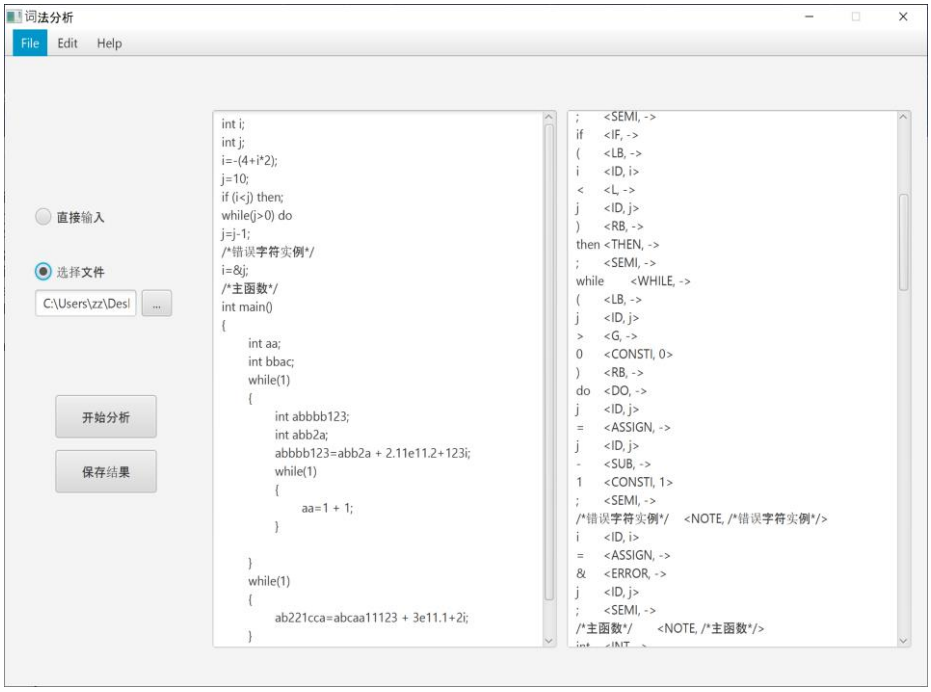

图 4-2 词法分析的结果

语法分析测试正确的源程序的结果如图 4-3 所示。

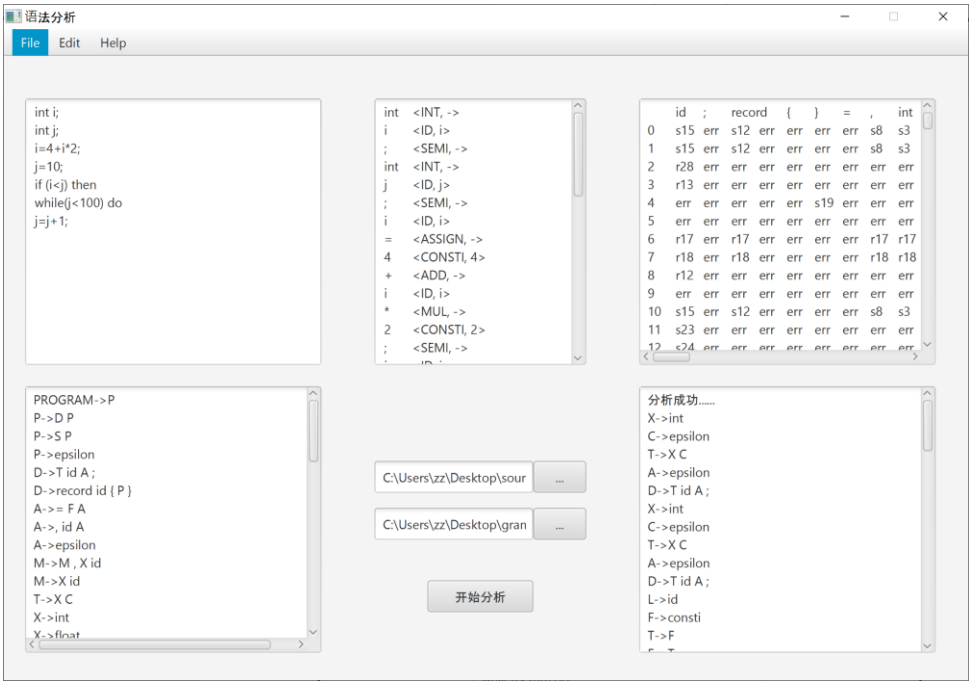

图 4-3 语法分析测试正确源程序的分析结果

语法分析测试错误的源程序的结果如图 4-4 所示。

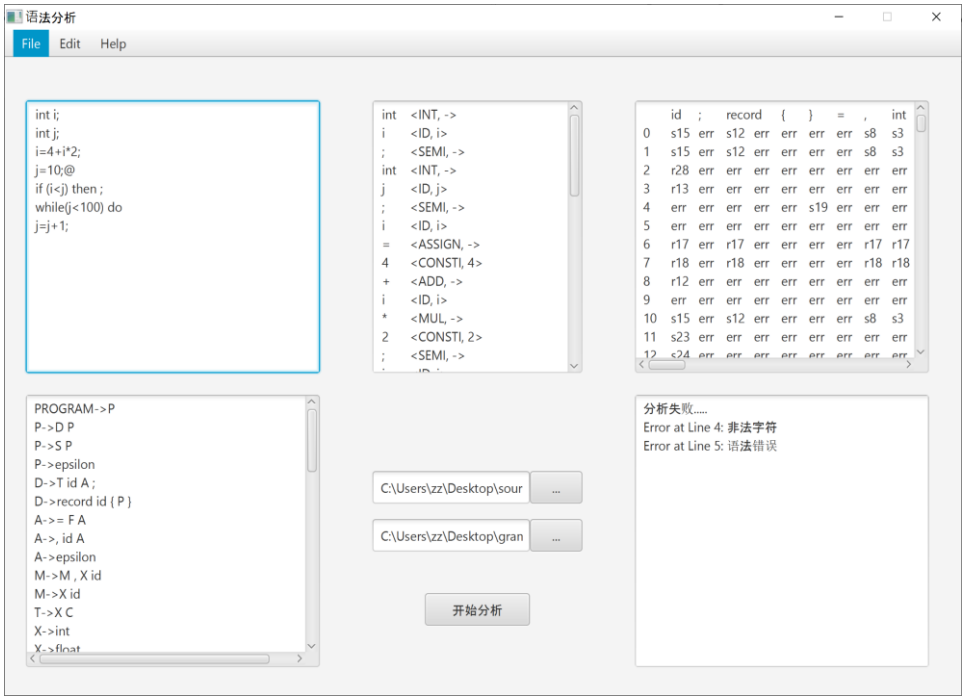

图 4-4 语法分析测试错误源程序的分析结果

语义分析针对的是已经没有语法和词法错误的源程序，因此，进行该测试的源程序已经通过了词法和语法检查。

如图 4-5 所示是一段通过语义分析的正确的源代码。

```
int a;
int b;
int c;
int d;
float x;
float y;
int z;
while (a<b) do{
    if (c<d) then{
        x=y+z;
    }else{
        x=y-z;
    }
}
```

图 4-5 语义分析测试中正确的源代码

如图 4-6 所示是图 4-5 所示代码的测试结果。

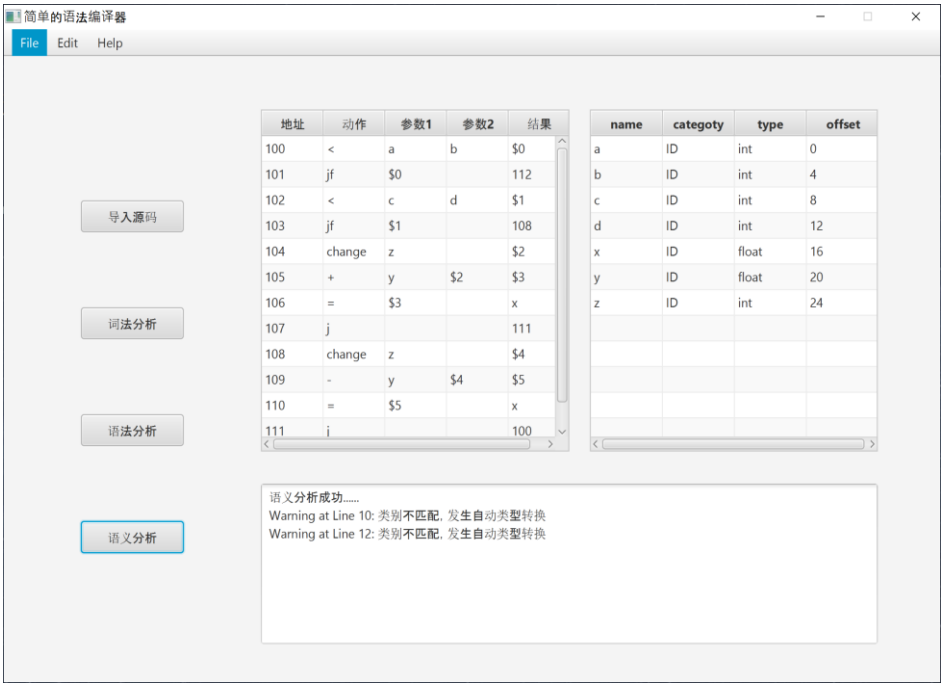

图 4-6 语义分析测试正确代码的结果

如图 4-7 所示是一段只包含语义错误的代码。注意这段代码中可以测试到的点包括简单的错误恢复机制，因此可以在一次分析中，发现多个错误。

```
int i;
int i;
bool b;
float k;
i=i+b;
i=j;
i=i+k;
```

图 4-7 语义分析测试中的错误的源程序

如图 4-8 所示是图 4-7 所示代码的测试结果。

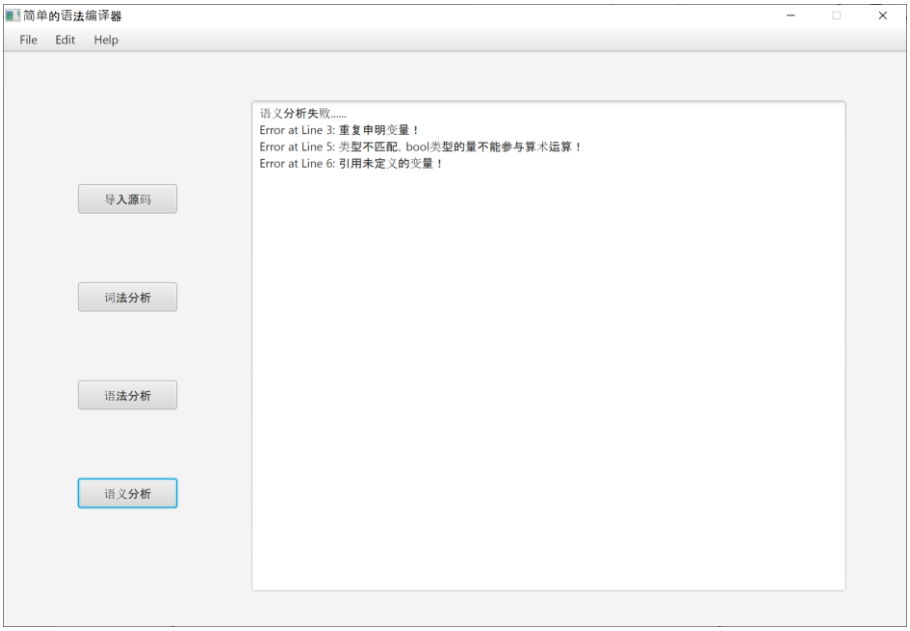

图 4-8 语义分析测试错误代码的结果

（2） 遇到的问题

在实现的过程中，遇到的问题也比较多。

① 在系统实现的过程中首先遇到的问题就是种别码的定义，作为一个初学者，我希望能够尽量完整的实现一个编译系统，这就要求要编译的语言不能太过复杂，否则在语义分析阶段难以完成。所以我必须选择一个尽量精简的词法，但是这个词法又要足够完整。

② 系统结构的设计。我希望能够有一个清晰高效的系统结构，如果有必要，在继续写语法语义分析时可以快速修改词法实现。

③ 识别注释时后，需要判断注释的开始和结束，而注释的第一个字符是“/”，与算数运算符中乘号冲突了，在注释结束时也是用连个连续的字符来标识，因此，无论是在注释的开始还是在注释的结束，都需要预取一个字符来判断，稍微有点小麻烦。

④ token 的表示，以及 token 中种别码和助记符的映射。Token 的定义是由种别码和属性值对组成，但是仅有这些信息在后续分析中无法给出错误的定位，因此需要一个方式来记录位置信息。我想到两种方式，一是在 token 序列中插入一种表示位置信息的特殊的 token 序列，在每一行开始的 token 之前，用以标识这是新的一行；二是为每个 token 加上位置信息。我这里选择的是第二种方式。

⑤ 求 FIRST 集和闭包以及项目集族时，在课件上是可以找到相应的例子的，但是在计算分析表的时候，课件上涉及到的一些例子均有一些小问题，在排除 bug 的过程中因此花费了较多的时间。为此，我手工对一些例子进行了求解。

⑥ 文法的设计。LR（1）文法是一种没有二义性的文法，要构造一个包含基本语句的文法比较有难度。实验指导中给出了一种文法，但是这是一个有二义性的文法，需要对其进行改造。

⑦ 语法分析器中涉及的数据的表示。涉及到比较多的数据，包括项目集族等。为了简单表示这些数据机构，并且兼顾快速的处理，并且能够尽量少使用存储空间。这是一个问题。

⑧ 错误处理。错误处理我使用的是恐慌模式，在进行弹栈寻找合适的状态后，恰好不需要丢弃输入符号，此时可能陷入死循环，为此，我规定错误处理时至少丢弃一个输入终结符。

⑨ 语义分析模式的选择。目前存在两种语义分析的模式，一种是与语法分析一起进行；另一种是在语法分析阶段生成语法树，作为一种中间表示，然后语义分析器使用语法树，完成语义分析。两种各有优缺点，前者资源使用少但是与语法分析的耦合度比较高，后者和语法分析的耦合低，但是速度更慢，使用存储控件更多。最终我选择了前者。

⑩ 翻译模式的设计。为了使用上面选择的语义分析模式，就要保证翻译模式中所有的语义动作均要在最后执行，因此需要修改文法。

（3） 结果分析

词法分析、语法分析、语义分析均能正确完成所设计的工作。

但是程序中也存在诸多不足，比如可能存在可能的结构不严谨的问题。而且，完整的编译器还应该具备代码优化和目标代码生成的部分。

指导教师评语：

日期：