|  |  |  |
| --- | --- | --- |
| **编译原理课程设计** | | |
|  | 题 目: | 文法三-词法分析、语法分析、语义分析 |
|  | 姓 名: | 黄子琰、董自经、孙伟嘉 |
|  | 学 号: | 19222125、19222126、19222111 |
|  | 学 院: | 人工智能学院 |
|  | 专 业： | 计算机科学与技术 |
|  | 班 级: | 计科221 |
|  | 指导教师: | 黄 芬 |
|  | 成绩评定: |  |
| 2024年12月9日 | | |

目录

[一、工作目标及介绍 2](#_Toc184670200)

[二、设计思路 3](#_Toc184670201)

[2.1编译程序概述 3](#_Toc184670202)

[2.2总体设计 4](#_Toc184670203)

[2.3 词法分析设计 5](#_Toc184670204)

[2.5语义分析 13](#_Toc184670205)

[三、核心代码 16](#_Toc184670206)

[3.1 文件读取模块(伪代码) 16](#_Toc184670207)

[3.2词法分析（伪代码） 17](#_Toc184670208)

[3.3 语法分析（伪代码） 19](#_Toc184670209)

[3.4语义分析（伪代码） 20](#_Toc184670210)

[四、测试用例及结果分析 20](#_Toc184670211)

[4.1 测试用例1 20](#_Toc184670212)

[4.2 测试用例2（测试用例1去掉了else） 22](#_Toc184670213)

[4.3 测试用例3 22](#_Toc184670214)

[4.4 测试用例4（测试用例3去掉一个;） 28](#_Toc184670215)

[4.5 测试用例5（测试用例3去掉#） 28](#_Toc184670216)

[五、遇到的问题 29](#_Toc184670217)

[5.1 词法分析遇到的问题 29](#_Toc184670218)

[5.2 语法分析遇到的问题 29](#_Toc184670219)

[5.3语义分析遇到的问题 29](#_Toc184670220)

[六、心得体会 29](#_Toc184670221)

一、工作目标及介绍

1、本次编译原理课程的设计，旨在通过词法、语法分析工具的设计，加深我们对编辑原理课程理论的理解，更好地理解计算机科学与技术的学科思维。同时也锻炼我们的逻辑思考能力。

2、采用SLR(1)语法分析方法，设计、开发所选文法描述语言的语法分析程序，加深对课堂相关理论教学内容的认识与理解，提高词法分析、语法分析、语义分析等诸多方法的应用与实践能力。

3、参照书中相应文法的翻译方案，采用语法制导的翻译技术，编写语义分析程序，实现对任务3所选择文法的语义分析，输出四元式中间代码，加深对课堂相关理论教学内容的认识与理解，提高词法分析、语法分析、语义分析等诸多方法的应用与实践能力。

4、用C/C++语言来完成本次课程设计。通过对理论知识的实践运用，深刻强化了对词法和语法分析的理解。编译器是现代计算机系统中最基本的组成部分之一，编译器是一种语言翻译程序，它把高级语言程序翻译成低级语言程序，编译器的重要性体现在它使大多数计算机用户不需要考虑与机器相关的繁琐细节，对理解编译器对软件开发人员能力的提高有很大帮助。

本次课设选题，我们选择的是文法三控制结构语句的词法分析+语法分析+语义分析，对于文法三是三个文法中较为困难的文法，主要内容为控制语句。

二、设计思路

2.1编译程序概述

编译程序完成从源程序到目标程序的翻译工作，是一个复杂的整体的过程。从概念上来讲，一个编译程序的整个工作过程是划分成阶段进行的，每个阶段将源程序的一种表示形式转换成另一种表示形式，各个阶段进行的操作在逻辑上是紧密连接在一起的。一般一个编译过程划分成词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成六个阶段。

上述编译过程的六个阶段的任务，可以分别由六个模块完成，它们称作词法分析程序、语法分析程序、语义分析程序、中间代码生成程序、代码优化程序和目标代码生成程序。

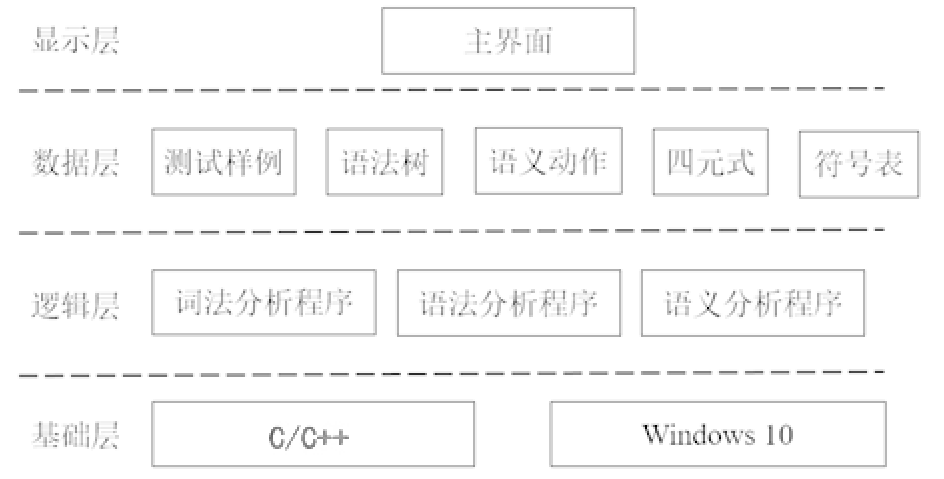


图2-1 系统框架图

图示

描述已自动生成

图2-2编译程序结构框图

2.2总体设计

对于课设整体的设计思路我们大致遵循以下的流程。在进行理论分析后，大致分为三个阶段，第一是从文件中读取含有测试语句的文件，接下来就是对读进来的语句进行词法分析，将语句分割成单词并生成一个识别序列，将词法分割生成的序列作为输入串按照SLR(1)分析表过程进行语法分析，同时对语法分析过的语句进行语法制导的翻译方案，即语义分析。

图示

描述已自动生成

图2-3编译程序结构框图

首先是读取文件部分的设计，我设计读取文件的主要目的是为了方便进行批量的测试，而不用直接修改源程序。其次是规范编程，提供标准的输入输出测试，便于源程序的维护。将文法相应的测试语句(正确和不正确都可以)从文件中读取，对语句进行词法分析，若读入的词句有误，则程序会报错。

在做词法、语法分析之前需要进行DFA构造，并根据绘制的DFA绘制SLR(1)分析表，并且根据文法以及运算符优先级来解决冲突，生成无冲突的SLR(1)分析表。

在词法分析过程中，需要根据读入的文件对单词进行分割，最后根据ACTION表中的终结符比如：end、if、then等与分割的单词进行比对，来生成语句的对应终结符的数字串，作为语法分析的输入串。

在语法分析过程中，需要用到SLR(1)分析表、对词法分析的输入串进行分析，步骤就是书上和上课讲过的理论步骤，但是实际执行的时候还是有一点难度的。语法分析的简单设计步骤就是根据SLR(1)分析表和输入串使用状态栈、符号栈、以及ACTION、GOTO等来识别活前缀，并及时即可规约的活前缀进行规约。这也是我们构造的DFA的功能就是识别活前缀。最后对输出输入串进行识别和分析，识别成功或者报错。

2.3 词法分析设计

词法分析程序的功能是从左到右逐字逐句扫描源程序，识别来自字符流中的源程序中的单词，实现一些功能，如：过滤源程序中的注释和空格，记录新输入字符行的信息，识别关键字、标识符、常数、运算符、界符等，辅助完成语法分析，将所发现的错误信息与源程序联系起来。在本次课程设计中，对关键字、标识符、界符、运算符、数字等进行识别。

其设计思想如下：程序可以实现从文本文件中逐行读取字符串，行读后对每行逐字符进行分析，我们将可能出现的字符串做了四大分类：常量，运算符，标识符，保留字，根据预先存放的token表或重构后的token表进行分类和标识，并用symbol\_table存放出现过的标识符,整体扫描一遍后,syn\_arr容器存放标识数字,scan\_arr 容器存放分割后的字符串。关键在于字符的准确分割和判断。

2.3.1 词法分析流程图

图片包含 钟表, 照片, 游戏机, 房间

描述已自动生成

图2-4词法分析步骤

2.3.2关键字和标识符

程序设计时因输入序列较长，因此我们组选择，将关键字和标识符转化为数字，关键字和标识符所对应的数字如下：

1 while

2 if

3 then

4 id

5 ;

6 :=

7 +

8 \*

9 -

10 (

11 )

12 do

13 else

15 rop

16 True

17 False

18 #

19 S

20 M

21 N

22 E

23 ||

24 &&

25 !

2.3.3相关词法规则

<标识符>::=<字母>

<标识符>::=<标识符><字母>

<字母>::=a|b|c|……|x|y|z

<加法运算符>::＝+|-

<乘法运算符>::＝\*

<关系运算符>::＝<|>|!=|>=|<=|==

<赋值运算符>::＝:=

<分界符>::＝;|(|)

<保留字>::＝while|if|then|do|else|True|False|id

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词(也称单词符号或符号)。词法分析程序实现这个任务。

2.4语法分析

语法分析就算的主要功能就是对从词法分析得到的对输入串转换之后的数字序列按照已经构造好的SLR(1)分析表进行相应的分析，判断输入语句是否合法，同时生成相应的语法树，为语义分析做准备。在本报告中，语义分析与语法分析是同步进行的，所以并未画出具体的语法树。

语法分析的主要原理如下：先设置相应的状态栈、符号栈、输入串，分别用于存储当前的状态序列、符号序列以及输入序列，输入序列会伴随着移进操作逐步进入符号栈，符号栈中的符号会在规约操作时出栈，并将相应的非终结符压入栈。状态栈中的数字序列会伴随着移进、规约操作而与输入序列、符号栈同步变化。

一开始分别将0、#压入状态栈和符号栈，之后读入输入串的首位元素，将状态栈的栈顶元素作为索引的第一部分key1，输入串的首位元素作为索引的第二部分key2，进入SLR(1)分析表中进行查找，如果根据两个索引可以找到相应的操作，则根据索引得到的结果action中的第一部分flag判断是移进还是规约操作。

如果是规约操作，先将状态栈和符号栈都做相应产生式右侧字符数量的出栈操作，之后将新的状态栈的栈顶元素作为索引的第一部分key1，规约之后的非终结符作为索引的额第二部分key2，继续根据SLR(1)分析表进行操作，但此时仅需要对状态栈进行操作，符号栈没有操作。

重复上述步骤直至输入字符串为空同时状态栈仅剩余01，符号栈仅剩余S即可判定输入字符串符合语法，输出acc。

2.4.1语法分析流程图

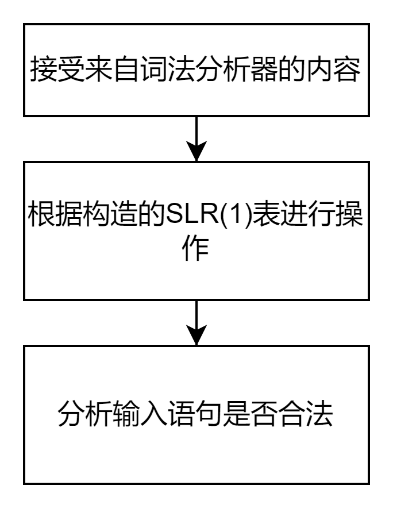


图2-5语法分析流程图

2.4.2文法产生式以及DFA状态

文法产生式：

0 S’->S

1 S->id := E

2 S->if E then M S

3 S->if E then M S N else M S

4 S->while M E do M S

5 S->S; M S

6 M->ε

7 N->ε

8 E->E + E

9 E->E \* E

10 E->- E

11 E->id

12 E->E || M E

13 E->E && M E

14 E->not E

15 E->(E)

16 E->E rop E

17 E->True

18 E->False

其中：S为语句，M、N为语义输出时必须符号，E为复合控制语句

DFA状态图：

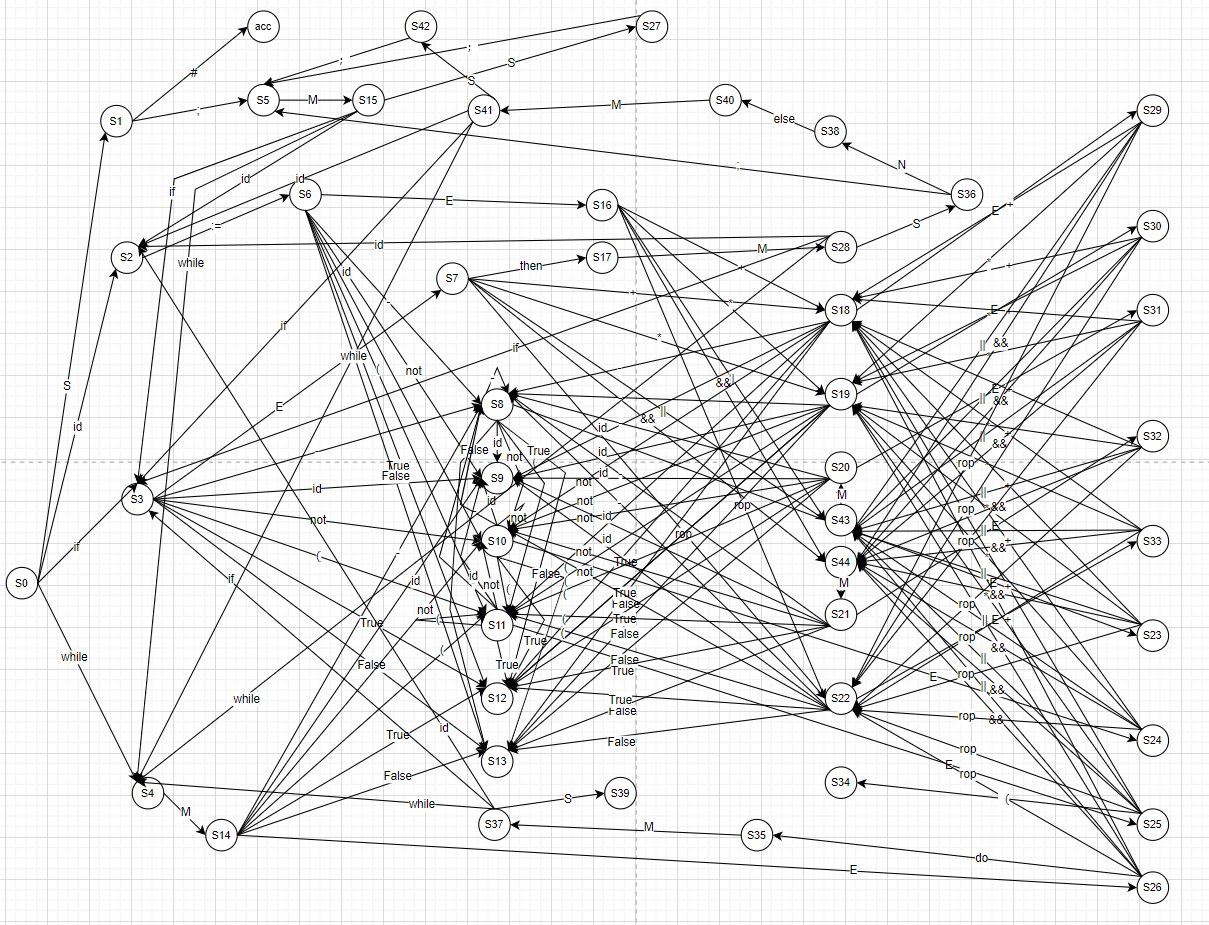


图2-6 DFA

其中：

S0:

S’->.S

S->.id:=E

S->.if E then M S

S->.if E then M S N else M S

S->.while M E do M S

S->.S;M S

S1:

S’->S.

S->S.;M S

S2:

S->id.:=E

S3:

S->if. E then M S

S->if. E then M S N else M S

E->.E + E

E->.E \* E

E-> .- E

E->.id

E->.E || M E

E->.E && E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S4:

S->while. M E do M S

M->.ε

M->ε.

S5:

S->S;. M S

M->.ε

M->ε.

SS6:

S->id :=.E

E->.E + E

E->.E \* E

E->.- E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S7:

S->if E .then M S

S->if E .then M S N else M S

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && E

E->E. rop E

S8:

E->-.E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S9:

E->id.

S10:

E->not .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S11:

E->(.E)

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E || M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S12:

E->True.

S13:

E->False.

S14:

S->while M. E do M S

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S15:

S->S; M .S

S->.id := E

S->.if E then M S

S->.if E then M S N else M S

S->.while M E do M S

S16:

S->id := E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S17:

S->if E then .M S

S->if E then .M S N else M S

M->.ε

M->ε.

S18:

E->E + .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S19:

E->E \* .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S20:

E->E | | M .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S21:

E->E && M .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E || E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S22:

E->E rop .E

E->.E + E

E->.E \* E

E->.-E

E->.id

E->.E | | M E

E->.E && M E

E->.not E

E->.(E)

E->.E rop E

E->.True

E->.False

S23:

E->-E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S24:

E->not E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S25:

E->(E.)

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S26:

S->while M E. do M S

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S27:

S->S; M S.

S->S.; M S

S28:

S->if E then M .S

S->if E then M .S N else M S

S->.id := E

S->.if E then M S

S->.if E then M S N else M S

S->.while M E do M S

S->.S; M S

S29：

E->E + E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S30:

E->E \* E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S31:

E->E | | M E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S32:

E->E && M E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S33:

E->E rop E.

E->E. + E

E->E. \* E

E->E. | | M E

E->E. && M E

E->E. rop E

S34:

E->(E).

S35:

S->while M E do .M S

M->.ε

M->ε.

S36:

S->if E then M S.

S->if E then M S. N else M S

S->S.; M S

N->.ε

N->ε.

S37:

S->while M E do M .S

S->.id := E

S->.if E then M S

S->.if E then M S N else M S

S->.while M E do M S

S->.S; M S

S38:

S->if E then M S N .else M S

S39:

S->while M E do M S.

S->S.; M S

S40：

S->if E then M S N else .M S

M->.ε

M->ε.

S41：

S->if E then M S N else M .S

S->.id := E

S->.if E then M S

S->.if E then M S N else M S

S->.while M E do M S

S->.S; M S

S42：

S->if E then M S N else M S

S->S; M S

S43：

E->E | | .M E

M->.ε

N->ε.

S44:

E->E && .M E

M->.ε

N->ε.

2.4.3 SLR(1)分析表

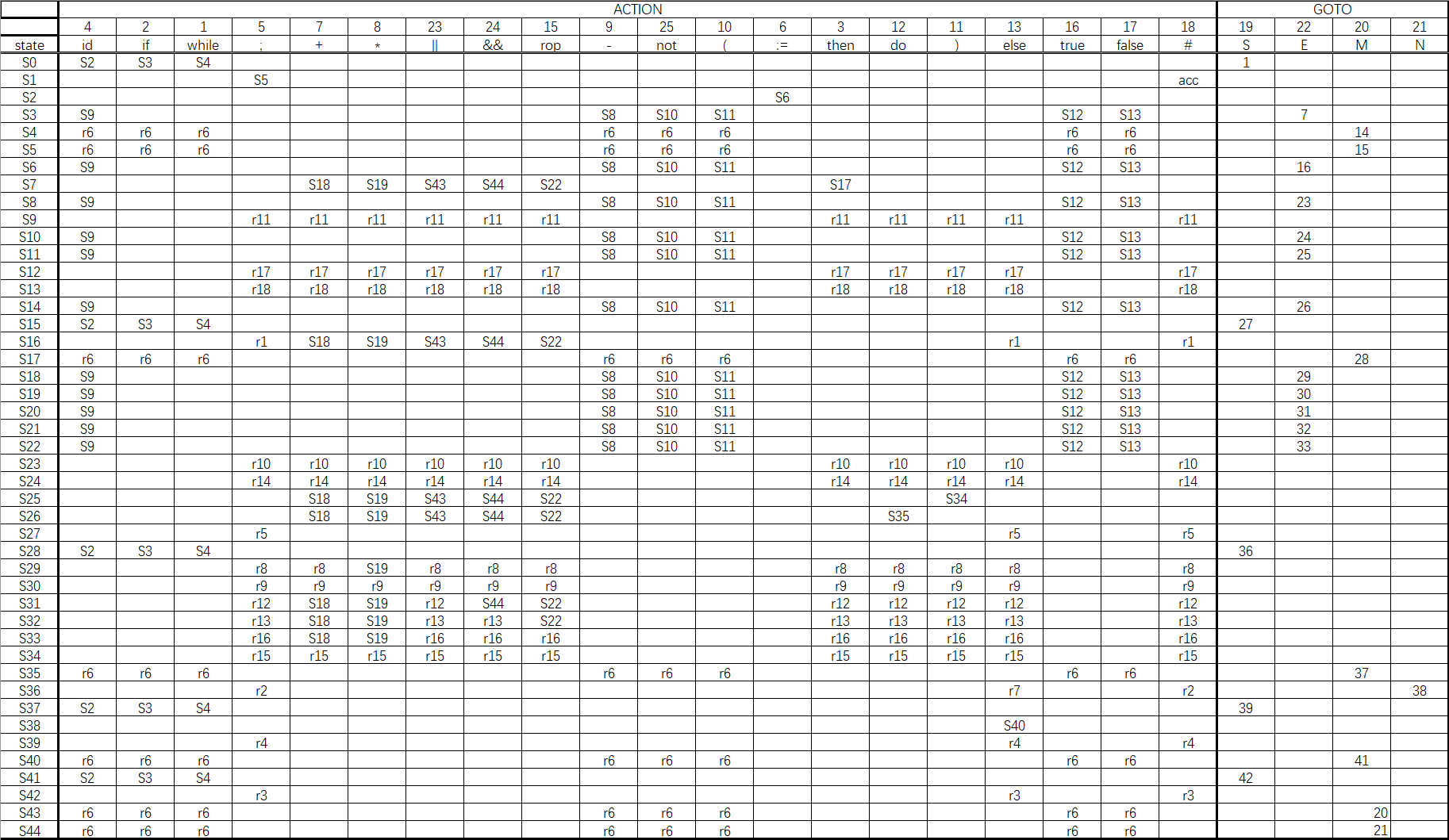


图2-7 SLR(1)分析表

2.4.4 语法分析需要的数据

对于语法分析而言，SLR(1)分析表是不可或缺的一部分，在本次课程设计中，采用了map<pair<key1,key2>,action>的数据结构来表示SLR(1)分析表，其中action为结构体含有flag标记操作类型以及，val对应操作的第几条。

同时采用了一个二维数组grammar来记录文法，由于语法分析仅需要知道规约后形成了什么非终结符以及需要规约几个字符，所以grammar中进存储了这两个信息，其中grammar[i][0]是相应产生式左侧的非终结符，grammar[i][1]是产生式右侧的字符数量。

2.5语义分析

2.5.1语义分析设计思路

语义分析的功能是在自下而上语法分析的过程中，在执行归约操作的时候进行对应的语义计算，生成中间代码。

语义分析并不是一个独立的模块。进行语义分析时，需要调用词法分析中生成的符号表symbol\_table，以及符号表的新地址newtemp。同时，还需要使用语法分析中生成的符号栈symbol，以及对应的栈顶指针top，我们设定符号栈存储的是Node类型的元素。

语义分析的大致流程如下：在语法分析执行归约操作前，调用语义分析操作，同时传入归约的是第几个操作以及一个新的Node类型元素P，表示归约后的要入符号栈的新的非终结符。识别到具体执行哪一个归约操作后，执行对应的S属性文法翻译方案，进行对应的代码回填、拉链、合并链、给P初始化等操作，同时在符号表中新增P的地址。如果当前归约操作需要，就在四元式代码中新增若干行四元式，格式为：（操作符，操作数1在符号表中的位置，操作数2在符号表中的位置，要跳转到的四元式地址）。等语法分析结束后，就生成了该程序段对应的四元式中间代码了，这就是大致的语义分析思路。

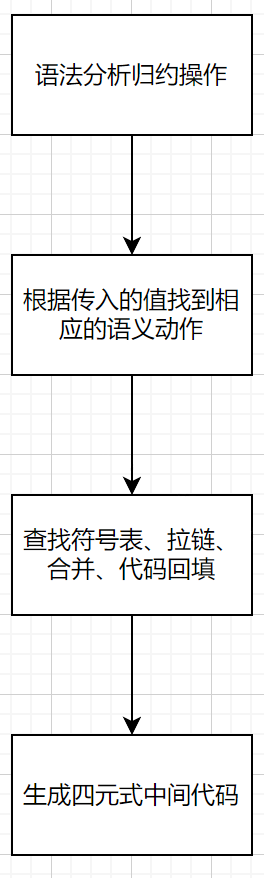
2.5.2 语义分析流程图

图2-8 语义分析流程图

2.5.3 语义属性

id.name：id对应的标志符名字

id.place：id对应的存储位置

E.place：用来存放E的值的存储位置

E.code：E求值的TAC语句序列

S.code：对应于S的TAC语句序列

E.truelist：“真链”，链表中的元素是一系列跳转语句的地址，这些跳转语句的目标标号应为布尔表达式 E 为“真”应该跳转的目标标号

E.falselist：“假链”，链表中的元素是一系列跳转语句的地址，这些跳转语句的目标标号应为布尔表达式 E 为“假”应该跳转的目标标号

S.nextlist：“next 链”，链表中的元素是一系列跳转语句的地址，这些跳转语句的目标标号是在执行序列中紧跟在 S 后的下条TAC语句的标号

M.gotostm：记录处理到M时下一条待生成语句的标号

产生式、语义规则、翻译模式

根据每个产生式的语义规则，设计相应的语义动作

属性文法、属性计算、翻译控制语句

2.5.4 语义函数/过程

lookup(id.name)：查表检查该标志符是否声明过

nextstm：下一条TAC 语句的地址

emit(op ,arg1, arg2,t)：生成一条 TAC 语句,并使 nextstm 加1

newtemp：在符号表中新建一个从未使用过的名字，并返回该名字的存储位置

makelist(i): 创建只有一个结点 i 的表，对应下一条跳转语句的地址

merge(p1,p2): 连接两个链表 p1 和 p2 ，返回结果链表

backpatch(p,i): 将链表 p 中每个元素所指向的跳转语句的标号置为 i

2.5.5 翻译模式

(1) S → id := E

{

P:=lookup(id.name);

if P ≠nil

then emit( :=, E.place,, P);

else error;

}

(2)S → if E then M S1

{

backpatch(E.truelist,M.gotostm) ;

S.nextlist := merge(E.falselist, S1.nextlist);

}

(3) S → if E then M1 S1 N else M2 S2

{

backpatch(E.truelist, M1.gotostm) ;

backpatch(E.falselist, M2.gotostm) ;

S.nextlist := merge(S1.nextlist, merge(N.nextlist, S2.nextlist) ) ;

}

(4)S → while M1 E do M2 S1

{

backpatch(E.truelist, M2.gotostm) ;

S.nextlist := E.falselist;

emit(‘goto’, , ,M1.gotostm);

backpatch(S1.nextlist, M1.gotostm) ;

}

(5) S → S1 ; M S2

{

backpatch(S1.nextlist, M1.gotostm) ;

S.nextlist := S2.nextlist ;

}

(6) M → ε

{

M.gotostm := nextstm;

}

(7) N → ε

{

N.nextlist := makelist(nextstm);

emit(goto , , , \_);

}

(8) E → E + E

{

E.place:= newtemp;

emit(+, E1.place,E2.place, E.place) ;

}

(9) E → E \* E

{

E.place:= newtemp ;

emit(\*, E1.place,E2.place, E.place);

}

(10) E → - E

{

E.place:=newtemp;

emit(-,E1.place,,E.place);

}

(11) E → id

{

E.place:=newtemp;

P:=lookup(id.name);

if P≠nil then E.place:=P;

else error;

}

(12) E → E || M E

{

Backpatch(E1.falselist,M.gotostm) ;

E.truelist := merge(E1.truelist, E2.truelist) ;

E.falselist := E2.falselist ;

}

(13) E → E && M E

{

backpatch(E1.truelist,M.gotostm);

E.falselist := merge(E1.falselist,E2.falselist) ;

E.truelist := E2.truelist ;

}

(14) E → ¬ E

{

E.truelist := E1.falselist ;

E.falselist := E1.truelist ;

}

(15) E → ( E)

{

E.truelist := E1.truelist ;

E.falselist := E1.falselist

}

(16) E → id rop id

{

E.truelist := makelist(nextstm);

E.falselist := makelist(nextstm+1);

emit (rop.op,id1.place,id2.place,\_);

emit (j, , ,\_) ;

}

(17) E → true

{

E.truelist := makelist(nextstm);

emit (j, , ,\_) ;

}

(18) E → false

{

E.falselist := makelist(nextstm);

emit (j, , ,\_) ;

}

三、核心代码

3.1 文件读取模块(伪代码)

为方便修改和测试，我选择将测试的内容放在文件中，程序从文件中读取测试内容进行测试。技术的实现主要是用C++的iofstream相关类实现对文件的读取，并对文件中的内容进行词法分析。

void LexicalAnalyzer::readCode(string path) {

string line;//行读缓冲区

ifstream inputFile(路径);//打开文件

if (!inputFile) {

cerr<<"无法打开文件 " << path << endl;

return;

}

while (getline(inputFile, line)) {

if (this->Analyse(line) == false) {//进行词法分析

cout << this->line\_count << "error" << endl;

}

if (line == "#") {

cout << "catch end" << endl;

break;

}

}

关闭文件

}

词法分析的主要目的就是将读入的语句，进行单词的分割，并与SLR(1)表中的保留字进行对比确定属于文法中的那个成分，比如 if a>b end# 就要把if end 等分割出来，然后就行对比确定属于SLR(1)分析表中的那个部分，最后得到一个分析后的单词序列。需要补充说明的是,在词法分析中，我们提供了一个类CreateToken允许通过读入token.txt文件的方式自动化的构造token表。这样做的好处是大大降低了程序的耦合性。当更换文法时不必“牵一发而动全身”。

3.2词法分析（伪代码）

语法分析就是按照书上的理论知识，来编程实现，主要是进行规约步骤的时候较为复杂。需要确定适用哪条产生式进行分析。

bool LexicalAnalyzer::Analyse(string str){

将行计数器 line\_count 加 1

初始化 value 为 0

遍历字符串 str 中的每个字符:

判断当前字符:

如果是 ' ' 或 '\t':

跳过 (继续下一个字符)

如果是 ':':

检查下一个字符:

如果是 '=':

设置 value 为 ":=" 的标记值

将 ":=" 添加到扫描表 scan\_table 并记录到输

出数组

否则:

设置 value 为 ":" 的标记值

将 ":" 添加到扫描表 scan\_table 并记录到输出

数组

如果是 'w' 且后续字符构成 "while":

设置 value 为 "while" 的标记值

将 "while" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "while" 的长度

如果是 'i' 且后续字符构成 "if":

设置 value 为 "if" 的标记值

将 "if" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "if" 的长度

如果是 't' 且后续字符构成 "then":

设置 value 为 "then" 的标记值

将 "then" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "then" 的长度

如果是 ';', '+', '\*', '-', '(', ')':

设置 value 为相应符号的标记值

将符号添加到扫描表 scan\_table 并记录到输出数组

如果是 '>' 或 '<':

检查下一个字符:

如果是 '=', '>' 或 '<>':

设置 value 为关系运算符的标记值

将运算符添加到扫描表 scan\_table 并记录到输

出数组

跳过运算符的长度

否则:

设置 value 为单个运算符的标记值

将运算符添加到扫描表 scan\_table 并记录到输

出数组

如果是 'T' 且后续字符构成 "True":

设置 value 为 "True" 的标记值

将 "True" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "True" 的长度

如果是 'F' 且后续字符构成 "False":

设置 value 为 "False" 的标记值

将 "False" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "False" 的长度

如果是 '|', '&', '!':

如果符号与下一个字符组成有效的双字符运算符:

设置 value 为相应运算符的标记值

将运算符添加到扫描表 scan\_table 并记录到输出数

组

跳过运算符的长度

如果是 'd' 且后续字符构成 "do":

设置 value 为 "do" 的标记值

将 "do" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "do" 的长度

如果是 'e' 且后续字符构成 "else":

设置 value 为 "else" 的标记值

将 "else" 添加到扫描表 scan\_table 并记录到输出数组

跳过 "else" 的长度

如果是字母或下划线:

提取完整的标识符:

如果标识符是关键字:

设置 value 为关键字的标记值

将关键字添加到扫描表 scan\_table 并记录到输

出数组

否则:

设置 value 为 "id" 的标记值

如果标识符未在符号表 symbol\_table 中:

添加到符号表 symbol\_table

将标识符记录到输出数组

其他情况:

跳过该字符

return True //表示分析完成

}

3.3 语法分析（伪代码）

语法分析器的操作只有移进和规约，所以对应的数据结构操作只有进出栈，在根据key1和key2查找到相应的动作后进行相应的操作即可。

void SyntaxAnalyzer(){

while(输入串不为空){

输出符号栈、状态栈、输入串

取输入串的第一个元素作为key2;

取状态栈栈顶元素作为key1;

根据(key1,key2)在SLR(1)分析表中进行查询;

if(查询结果不为空)

if(移进操作){

移进

更新状态栈、符号栈

打印移进操作

if(符合结束条件)

打印acc

}

else if(规约操作){

根据key2查找grammer数组，确定规约几位

规约成什么非终结符

更新状态栈、符号栈

打印规约操作、GOTO操作

}

else{

打印错误信息

}

}

}

3.4语义分析（伪代码）

语义分析的功能是在自下而上语法分析的过程中，在执行归约操作的时候进行对应的语义计算，生成中间代码。以下是语义分析核心模块的伪代码。

void Semantic\_Analyzer::reduction(int type, Node& P) // r1~r18,语义分析

{

语义分析函数的参数为要执行哪个产生式的归约动作type,以及归约成的非终结符P

if (type == 1) // S -> id := E

{

找到对应的语义动作并执行

对于id类型变量，查找其在符号表中的位置，判断合法性

如果有新的临时变量生成，则在符号表中新增元素，为其分配位置

处理好对应的链，执行拉链、代码回填等操作

生成对应的中间代码

}

else if (type == 2) // S -> if E then M S

…

}

每次进行语法分析的归约操作时，都需调用语义分析函数，为符号栈中的元素赋好相应的符号表位置、四元式代码位置、truelist、falselist、nextlist等相关语义属性信息，最后完成自下而上语法分析时，可以实现输出程序段的中间代码。

四、测试用例及结果分析

4.1 测试用例1

if True then a:=c else a:=b

#

测试结果：

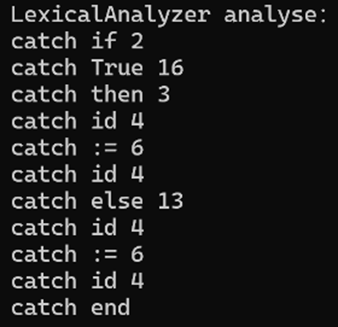


图4-1测试用例1词法分析

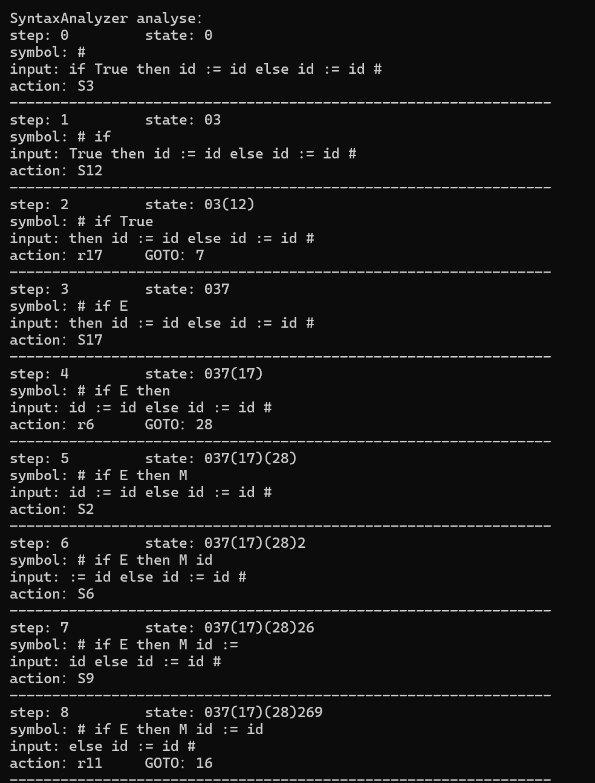
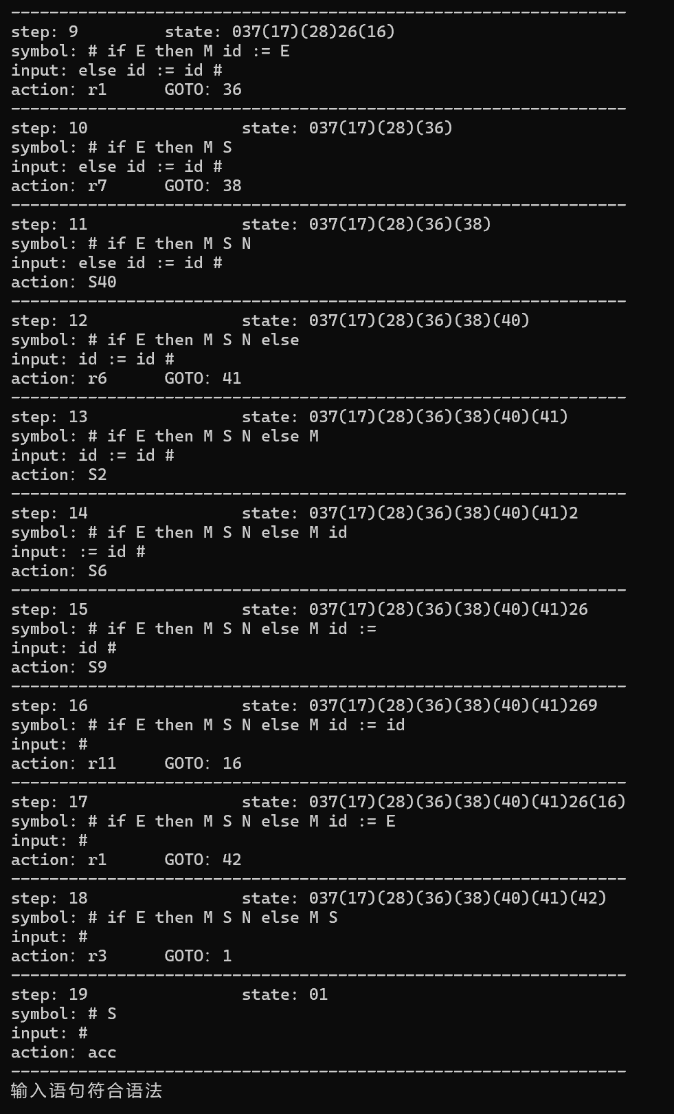
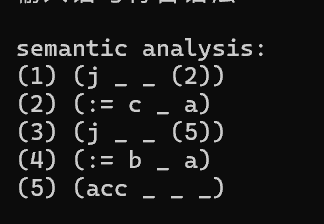
图4-2测试用例1语法分析1

图4-3测试用例1语法分析2

图4-4测试用例1语义分析

4.2 测试用例2（测试用例1去掉了else）

if True then a:=c a:=b

#

测试结果：

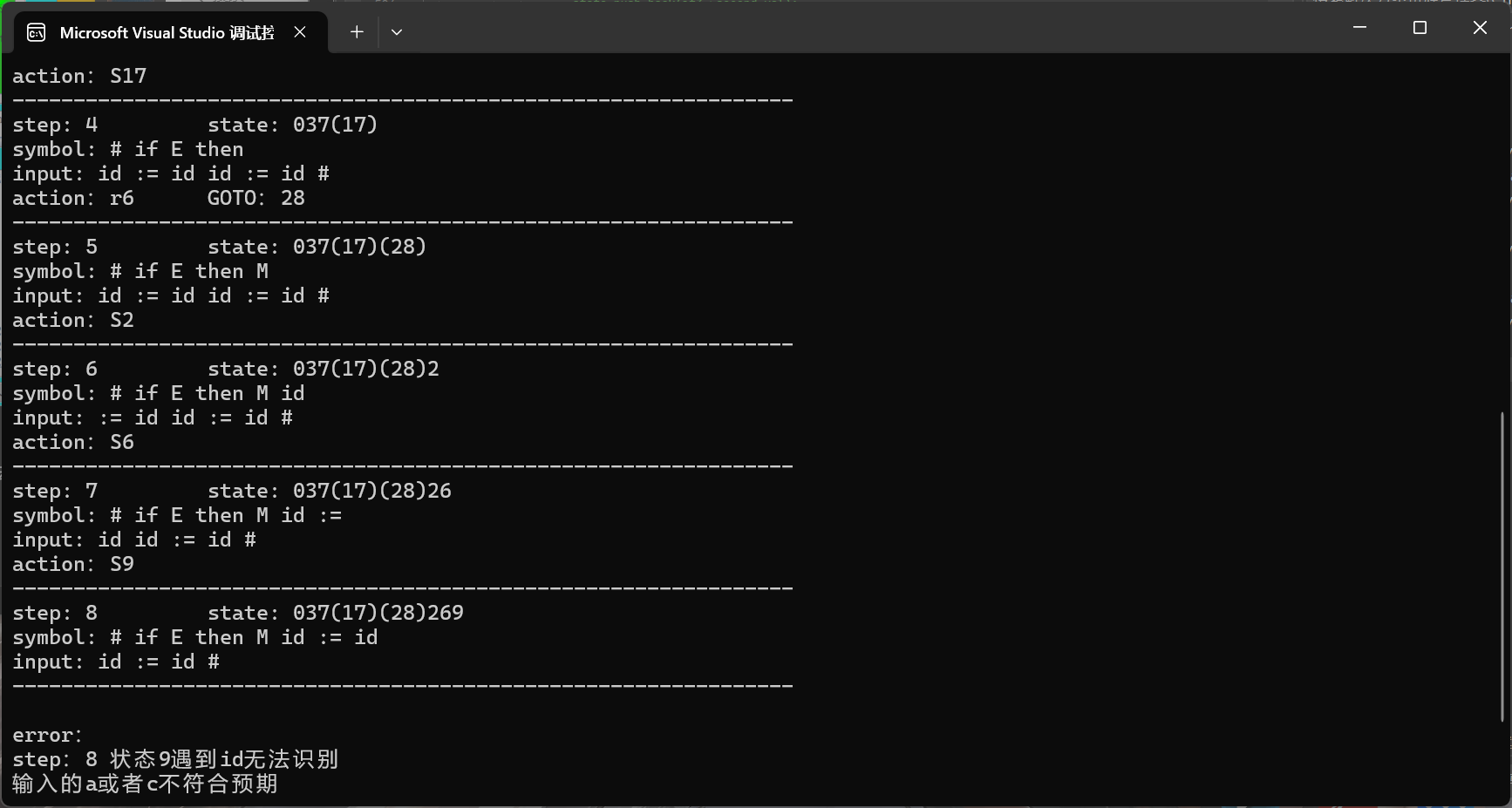


图4-5测试用例2-显示错误信息

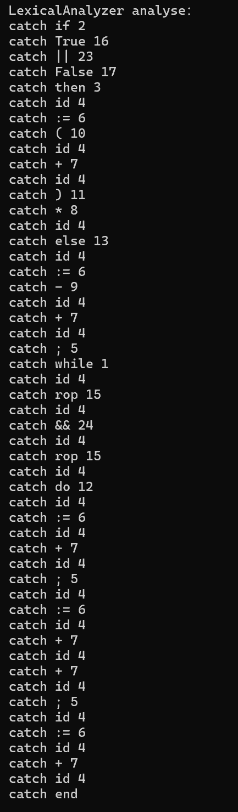
从输出结果中发现输入有错误，在原本应该时else的区域输入了不合时宜的a，导致后序步骤无法进行。

4.3 测试用例3

if True || False then a:=(b+c)\*a else a:=-b+c;

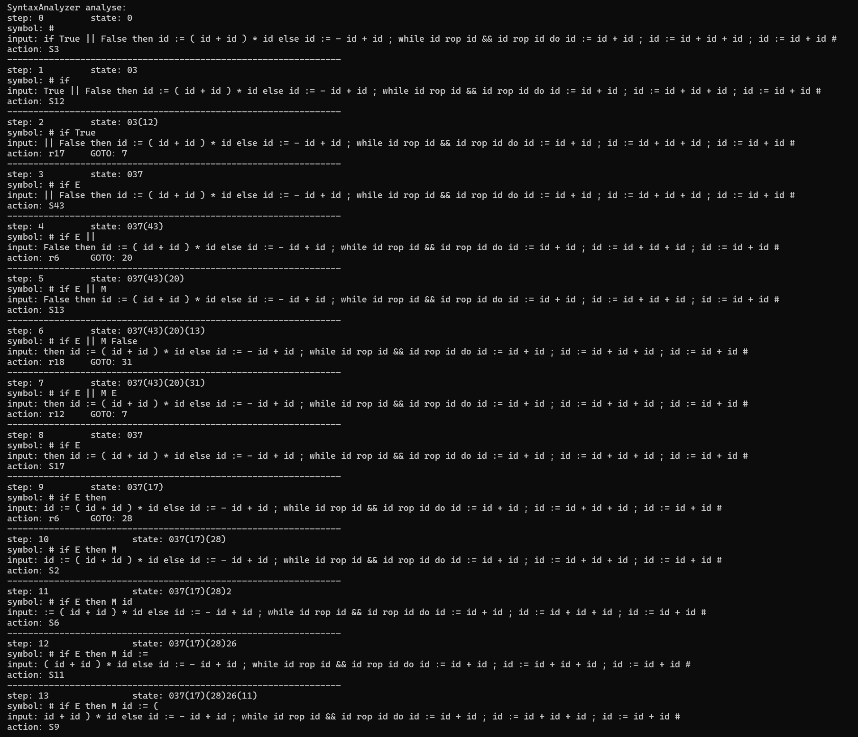
while b<=c && a<>c do b:=b+a;

ans:=a+b+c;ans:=ans+d

#

测试结果：

图4-6测试用例3词法分析

图4-7测试用例3语法分析1

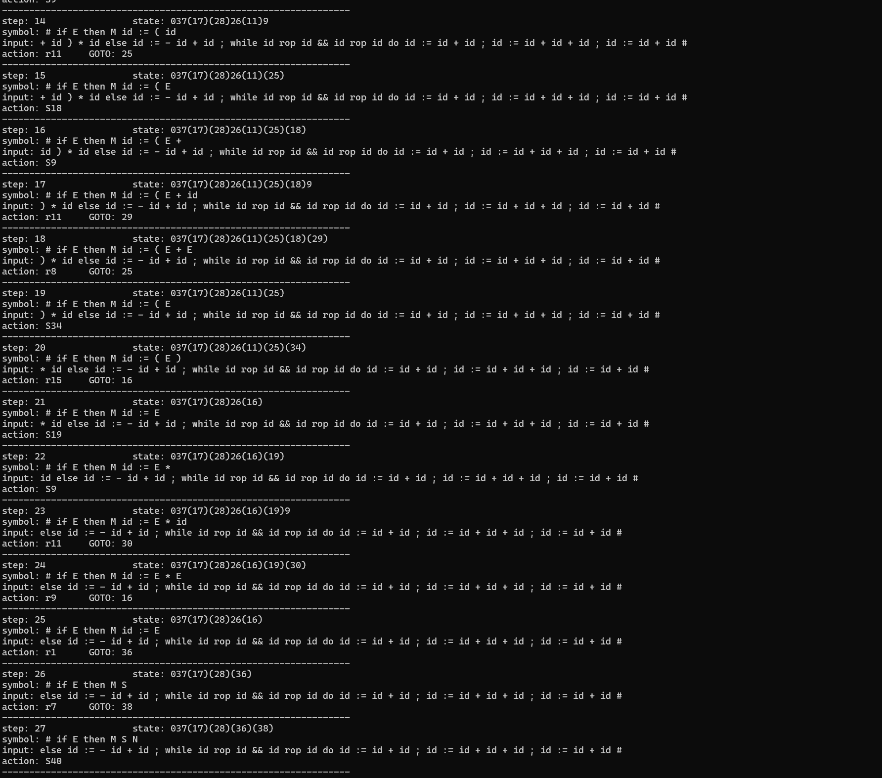


图4-8测试用例3语法分析2

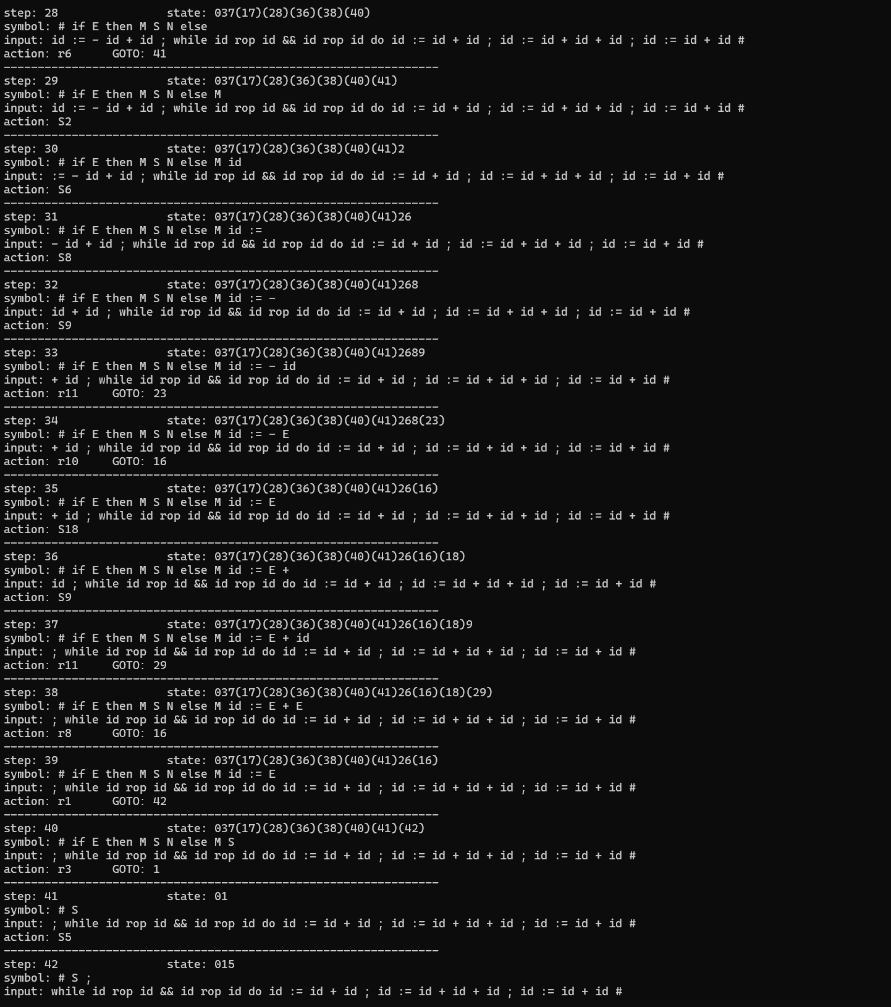
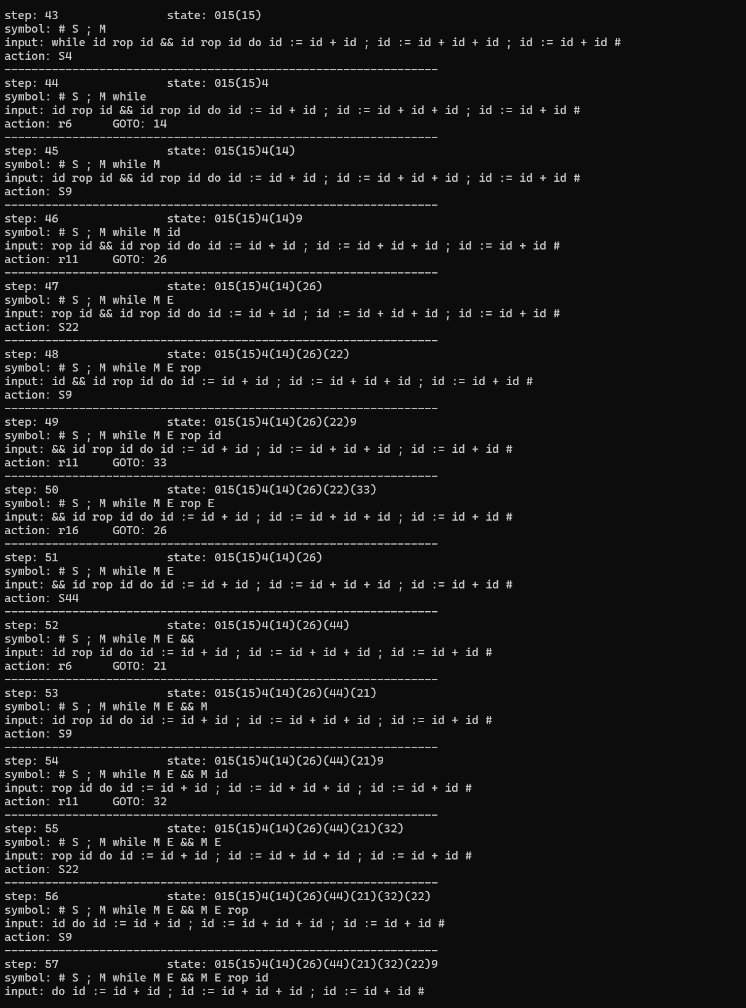
图4-9测试用例3语法分析3

图4-10测试用例3语法分析4

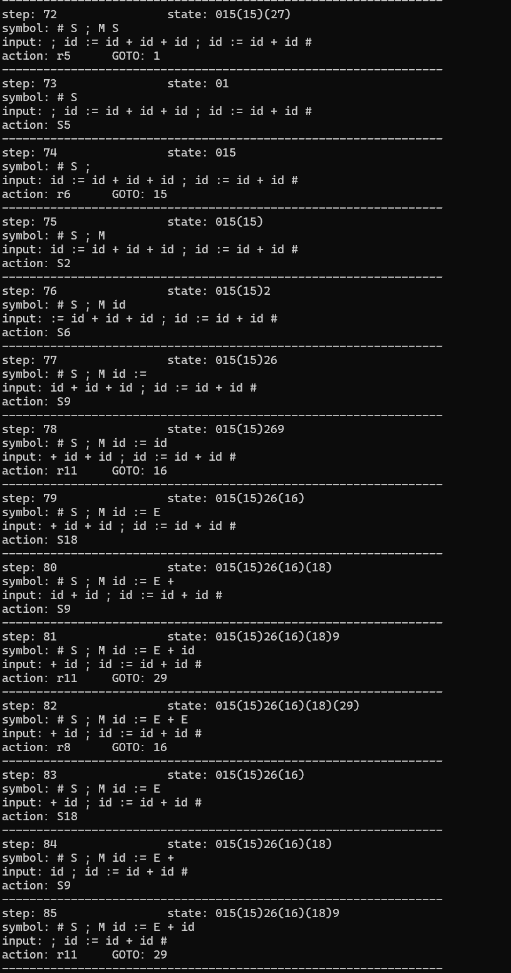
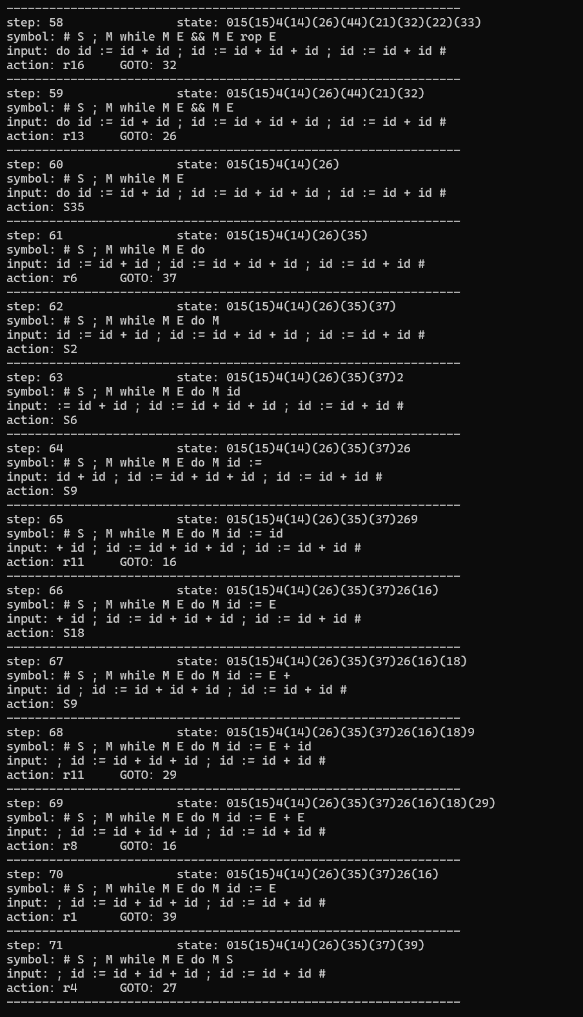
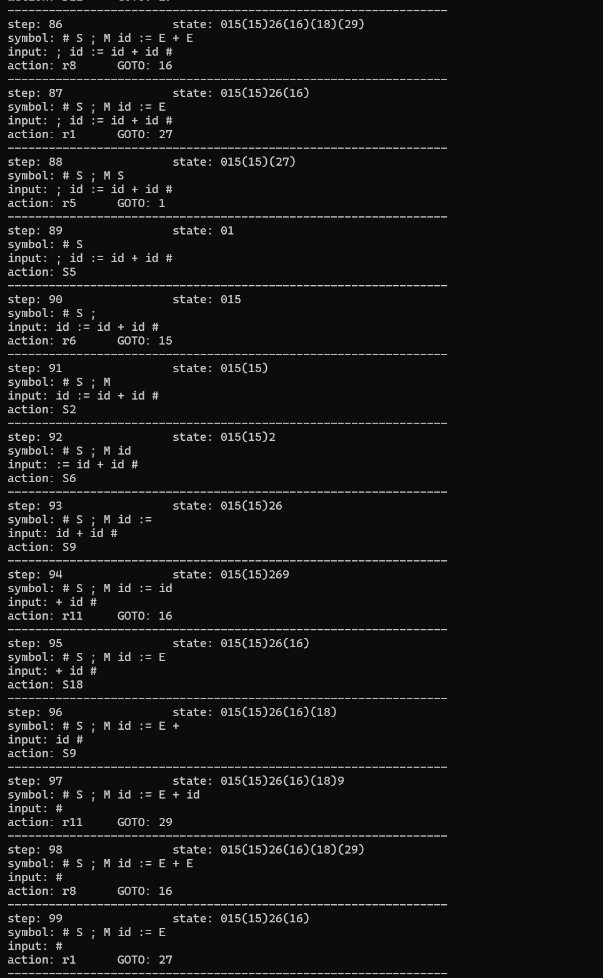
图4-11测试用例3语法分析5

图4-12测试用例3语法分析6

图4-13测试用例3语法分析7

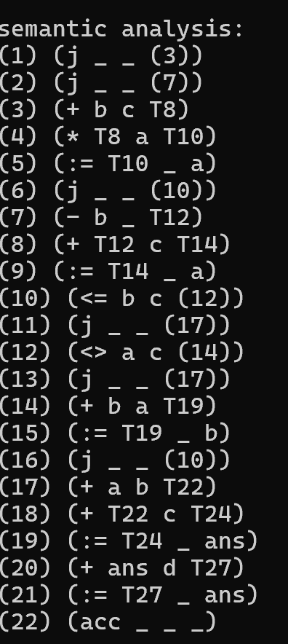
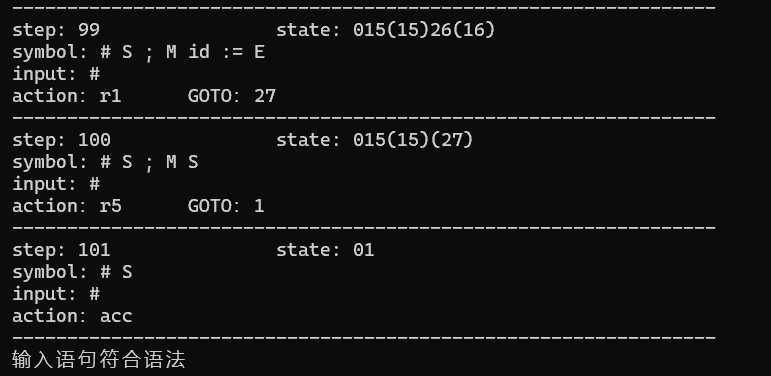
图4-14测试用例3语法分析8

图4-15测试用例3语义分析

4.4 测试用例4（测试用例3去掉一个;）

if True || False then a:=(b+c)\*a else a:=-b+c;

while b<=c && a<>c do b:=b+a

ans:=a+b+c;ans:=ans+d#

测试结果：

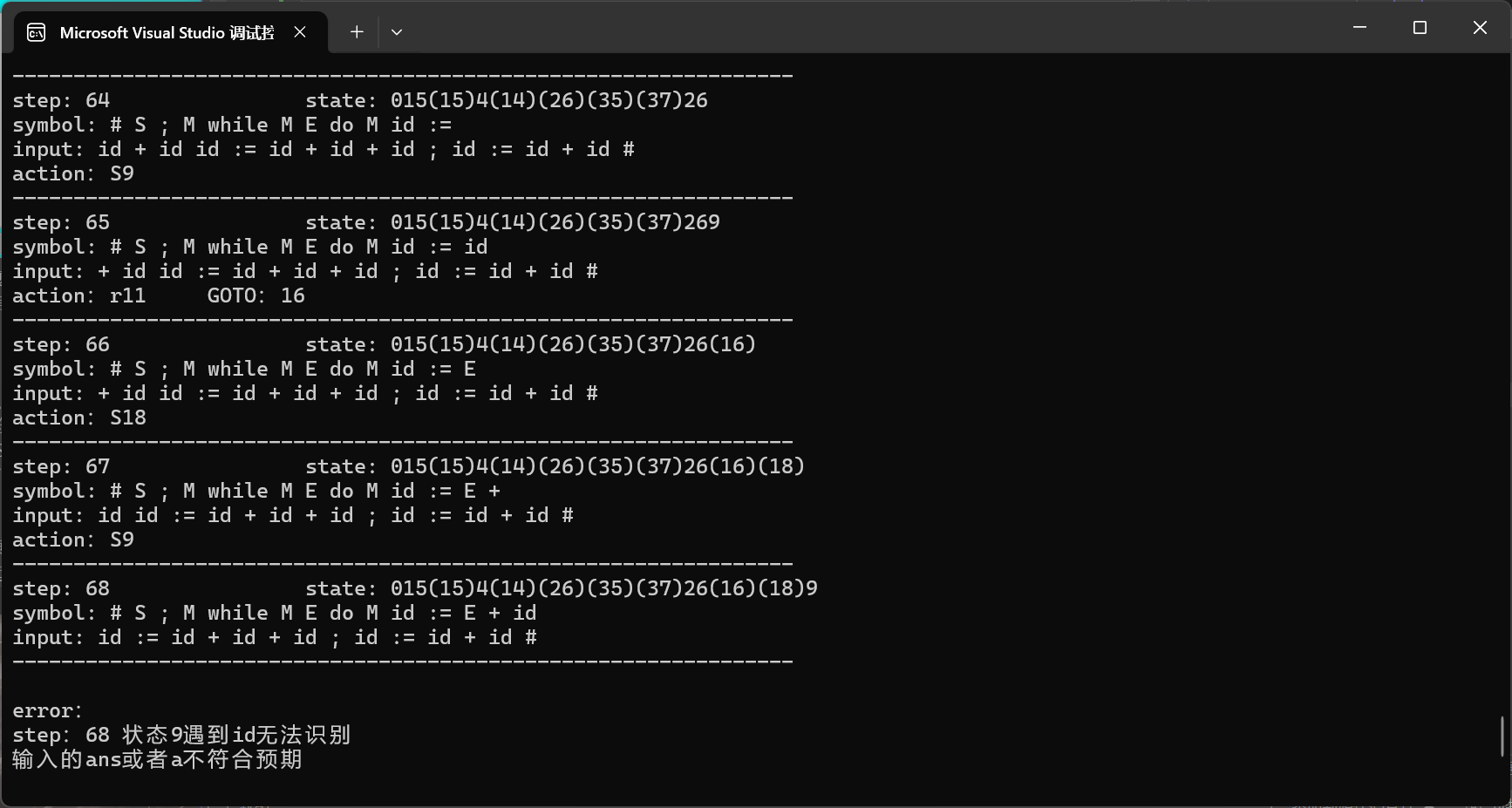


图4-16测试用例4-显示错误信息

在输出结果中发现原本应该时;的位置输入了不符合语法的ans，导致后序步骤无法进行。

4.5 测试用例5（测试用例3去掉#）

if True || False then a:=(b+c)\*a else a:=-b+c;

while b<=c && a<>c do b:=b+a;

ans:=a+b+c;ans:=ans+d

测试结果：

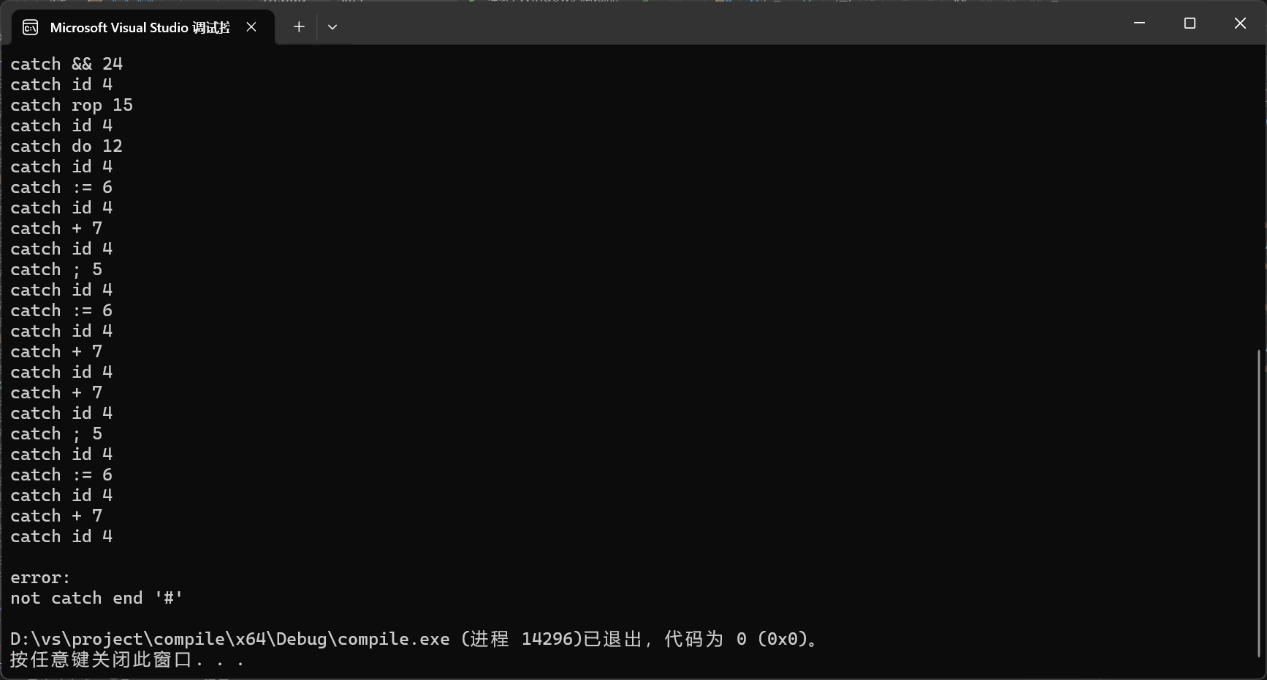


图4-17 测试用例5-显示错误信息

输入语句末尾没有#在词法分析阶段直接被判断为不符合词法而停止程序的继续执行。

五、遇到的问题

5.1 词法分析遇到的问题

词法分析的过程中,我遇到了几个困难,一是通过什么样的方式对输入的字符串进行单词分割,通过怎么样的手段达成什么样的结果,尤其是对保留字的识别方式。我构思过很多种方法去识别保留字等匹配长度大于一位的符号串，包括但不限于利用正则表达式前缀匹配，栈型结构等，最后通过上网查资料、与同学老师交流、自己思考等方式最终找到了最适合进行单词分割的手段。

二是如何降低程序的耦合性，让整个分析器不至于更换文法后整体报废。如何使程序可以适用于不同的文法，最终我想到了借用Spring架构的思想对程序整体进行抽象和解耦，将整个分析程序抽象成几部分，在词法分析阶段使用配置文件的方式构造token表，同样思想也运用在阅读code文件时，这样就可以解决程序耦合度高的问题。

5.2 语法分析遇到的问题

语法分析遇到的第一个问题就是二义性，对于同时具有移进、规约操作的状态，初步可以根据相应的FOLLOW集合进行解决，但是仍然存在少量存在二义操作的状态，此时根据算符优先级以及左结合的原则可以将其消除。

其次，由于语法分析是与语义操作同时进行的，所以语法分析就不能像往常一样仅仅进行语法分析，而是要在进行语法分析的同时，向语义分析传递相应的参数，并对这些参数及时进行修改、维护以确保语义分析的正确性。

5.3语义分析遇到的问题

语义分析时往往要综合词法分析器和语法分析器，统一三者具体的维护信息以及维护类型，在进行词法或语法分析的过程中，都要为语义分析做好铺垫，这样才能确保语义分析的正确运行。也正是因为语义分析的综合性，使得设计时需要有全局思维，而且在具体实现的过程中调试的难度较大。

六、心得体会

**黄子琰**

这次的编译原理课设还是富有挑战性的，这是第一次多人一起写一个代码量较大的一个工程，对我来说算是一次很好的历练过程。多人共享写代码如果分工明确，配合默契的话，效率会提高很多，但是因为是第一次合作完成代码，加上编译原理的实现代码耦合性比较大，总体来说过程还是比较磕磕绊绊的。总得来说，这次课设还是很有收获的，不仅再次加深了对编译原理工作流程的理解，提高了语义分析的熟练度，也增长了多人合作完成项目的经验。

**董自经**

此次编译原理课程设计加深了我使用自顶向下（递归下降）和自底向上（LR解析）的方法进行语法分析。这些技术可以帮助我正确解析输入，还让我能够构建解析树和抽象语法树（AST），为后续的语义分析和代码生成打下基础。通过不断调试和优化我的语法分析器，我体验到了设计简单而高效的文法所带来的挑战与乐趣。

同时，我的编程能力得到了显著提升。通过优化代码结构，我提高了代码的可读性，使得其他人能够更容易理解我的实现逻辑。而且，团队合作的经历让我明白了协作的价值。与同学们讨论语法规则和解析算法时，我学会了更好地表达自己的想法并注意聆听他人的见解，这使我们的代码质量得到了提升。在这个过程中，代码审查和相互支持也进一步激励了我的成长。

**孙伟嘉**

在本次编译原理的课程设计中，我在编程实践中得到了切实而宝贵的一课，我深刻理解了理论知识和编程实践能力的紧密结合，对于计算机科学与技术专业学生二者缺一不可，在编码过程中，我一边通过编程锻炼自己的实践能力又同时对自身编译原理的理论知识进行查漏补缺。

另外，我还在本次编译原理课程设计中，在与同小组同学的紧密合作中得到了沟通能力与团队合作能力的显著提升，深刻认识了不同同学编码的特点，认真学习了各同学编码的长处。