南京航空航天大学

编译原理
课程设计

目 一个 PASCAL 语言子集（PL/0）
编译器的设计与实现

学生姓名 杨通达

学　　号 051810217

班　　级 1618103

学　　院 计算机科学与技术学院

专　　业 计算机科学与技术

指导教师 谢

2021 年 1 月 4 日

目录

1 设计任务 **1**

2 系统设计 **3**

2.1 词法分析器 4

2.1.1 任务 4

2.1.2 设计思路 4

2.2 语法分析器 4

2.2.1 任务 4

2.2.2 设计思路 4

2.3 语义分析与中间代码产生器 6

2.3.1 任务 6

2.3.2 设计思路 6

2.4 目标代码生成器 9

2.4.1 任务 9

2.4.2 设计思路 10

2.5 解释器 12

2.5.1 任务 12

2.5.2 设计思路 12

2.6 错误处理 14

2.6.1 词法分析阶段 14

2.6.2 语法分析阶段 14

2.6.3 语义分析阶段 15

3 课设总结 **15**

1 设计任务

一个 PASCAL 语言子集（PL/0）编译器的设计与实现。

```
<prog> → program <id>; <block>
<block> → [<condecl>][<vardecl>][<proc>]<body>
<condecl> → const <const>{,<const>};
<const> → <id>:=<integer>
<vardecl> → var <id>{,<id>};
<proc> → procedure <id> ([<id>{,<id>}]) ;<block>{;<proc>}
<body> → begin <statement>{;<statement>}end
<statement> → <id> := <exp>
|if <lexp> then <statement>[else <statement>]
             |while <lexp> do <statement>
             |call <id> ([<exp>{,<exp>}])
             |<body>
             |read (<id>{, <id>})
             |write (<exp>{,<exp>})
<lexp> → <exp> <lop> <exp>|odd <exp>
<exp> → [+|-]<term>{<aop><term>}
<term> → <factor>{<mop><factor>}
<factor>→<id>|<integer>|(<exp>)
<lop> → =|<>|<|<=|>|>=
<aop> → +|-
<mop> → *|/
<id> → l{l|d}  (注：l表示字母)
<integer> → d{d}
```

其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需要生成相应的目标代码时，则调用代码生成程序。

```
LIT 0 , a 取常量a放入数据栈栈顶
OPR 0 , a 执行运算，a表示执行某种运算
LOD L , a 取变量（相对地址为a，层差为L）放到数据栈的栈顶
STO L , a 将数据栈栈顶的内容存入变量（相对地址为a，层次差为L）
CAL L , a 调用过程（转子指令）（入口地址为a，层次差为L）
INT 0 , a 数据栈栈顶指针增加a
JMP 0 , a无条件转移到地址为a的指令
JPC 0 , a 条件转移指令，转移到地址为a的指令
RED L , a 读数据并存入变量（相对地址为a，层次差为L）
WRT 0 , 0 将栈顶内容输出
```

用表格管理程序建立变量、常量和过程标识符的说明与引用之间的信息联系。

用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质。
当源程序编译正确时，PL/0 编译程序自动调用解释执行程序，对目标代码进行解释执行，并按用户程序的要求输入数据和输出运行结果。
假想机结构为：

1. 两个存储器（code 存储 P 代码，stack 数据栈）
2. 寄存器 I 存储当前运行的代码
3. 寄存器 T 存储当前数据栈栈顶
4. 寄存器 B 存储当前活动记录基地址
5. 寄存器 P 存储下一条要运行的代码

设计的活动记录为：

变量
形参
n（形参个数）
RA（返回地址）
DL（调用该过程的调用这的活动记录首地址）|
SL（该过程直接外层的活动记录首地址）

2 系统设计

编译程序就是能够把源语言程序转换为目标语言程序的程序。编译器的结构主要包括：词法分析器、语法分析器、语义分析与中间代码产生器、优化器以及目标代码生成器。

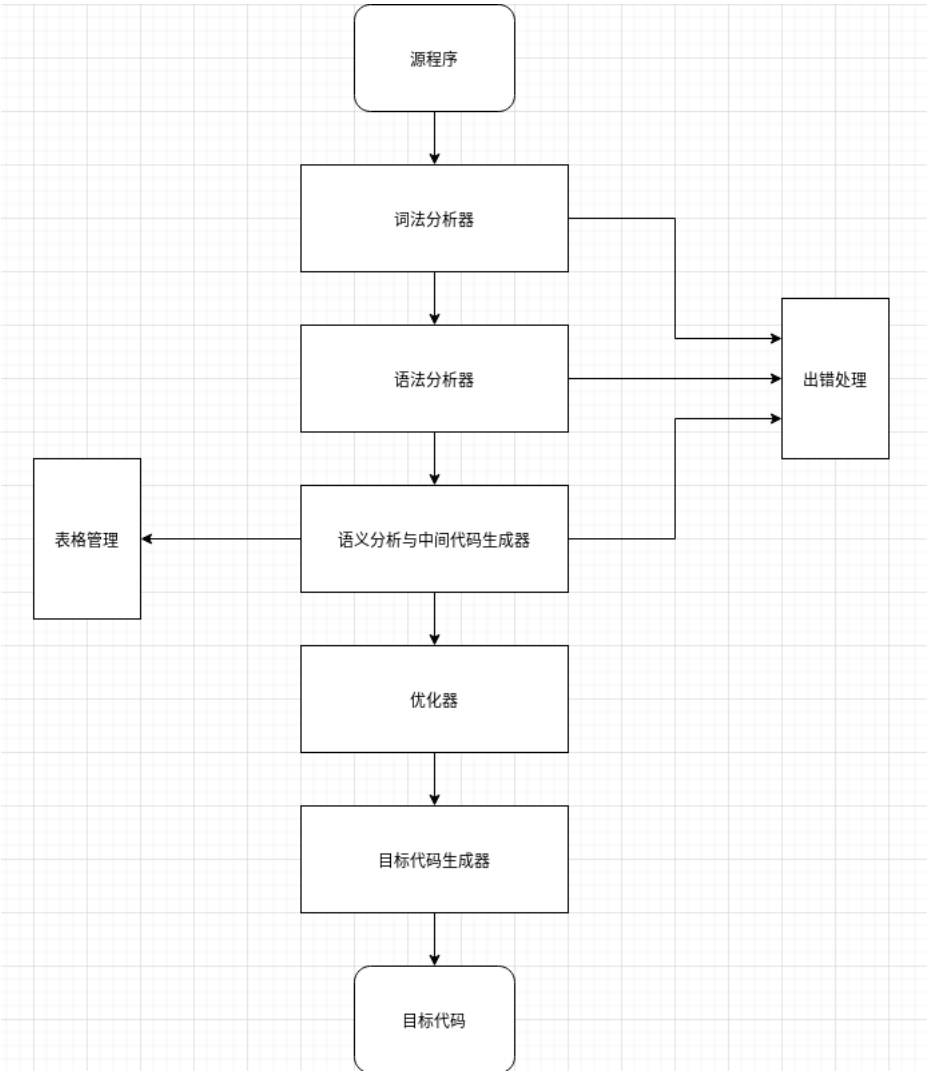

图 1: 编译器结构

PL/0 语言的编译程序包括词法分析器、语法分析器、语义分析与中间代码产生器以及目标代码生成器，同时设计一个解释器对生成的 P 代码进行解释运行。PL/0 语言编译程序采用那个以语法分析为核心，词法分析和语法分析及中间代码生成器作为独立的子程序共语法分析程序调用，成功生成中间代码后，调用后端目标代码生成器将中间代码翻译为 P 代码，在调用解释器对其解释运行。接下来对程序中的每个部分进行介绍。

对于测试用例：

```
program xi;
const a:=5;
var j,sum,x;
   procedure suml(x);
   var j;
```

```
  begin
    j:=10;
    sum:=0;
    while j<=x do
    begin
      sum:=sum+j;
      j:=j+1
    end;
    write(sum, a)
  end
begin
  read(x);
  write(x);
  read(j);
  write(j);
  call suml(j);
  write(j)
end
```

2.1 词法分析器

2.1.1 任务

词法分析的任务是输入源程序，对构成源程序的字符串进行扫描和分解，识别出一个一个的单词。识别出的单词用于后续的语法分析阶段。

2.1.2 设计思路

词法分析器的设计思想主要是利用有限状态机对输入的字符进行处理，得到单词符号或产生错误类型。当有限自动机成功识别出字母、PL/0 语言所含的符号以及数字后，输出得到的单词，否则调用错误输出程序 errorprint () 输出对应的错误。每次语法分析器对一个单词进行匹配以后，都调用词法分析器对源程序进行分析，得到下一个单词。

2.2 语法分析器

2.2.1 任务

语法分析的任务是在词法分析的基础上，根据语言的语法规则，把单词符号串分解成各类语法单位（语法范畴），如“短语”、“子句”、“句子”和“程序”等。

2.2.2 设计思路

在语法分析阶段，采用自顶向下的语法分析模式。其主要步骤是：

1. 消除左递归

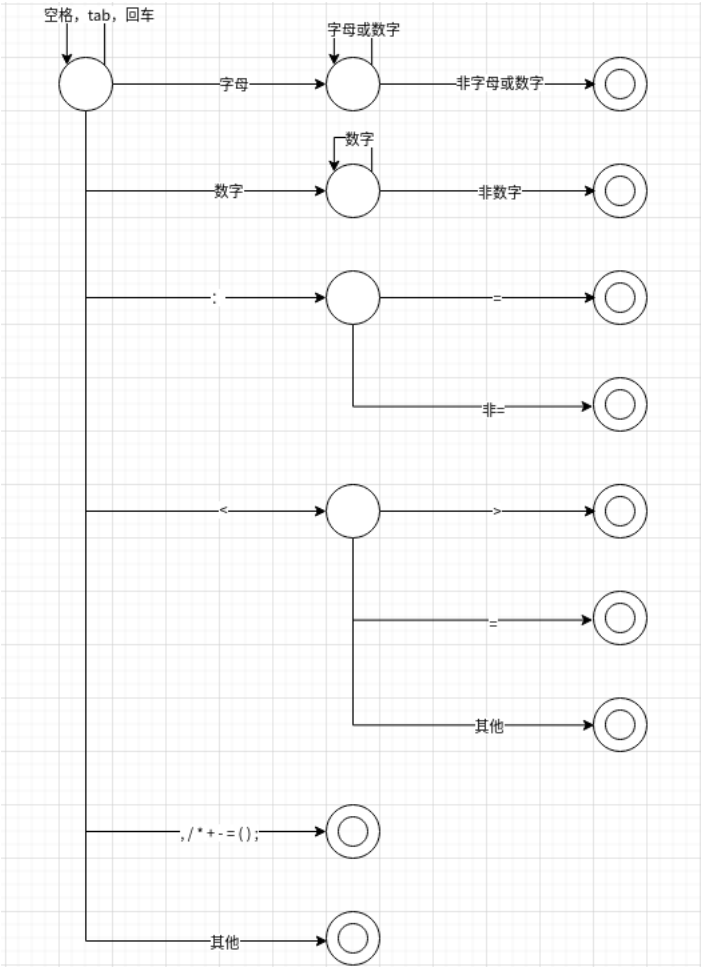

图 2: 词法分析器

2. 提取公共左因子
3. 求解 First 集合
4. 对读进的单词进行匹配

由于 PL/0 语言不存在左递归以及公共左因子，我们求解 PL/0 语言的 First 集合：

```
<prog> = {'program'}
<block> = {'const','var','procedure','begin'}
<condecl> = {'const'}
<const> = {'l'}
<vardecl> = {'var'}
<body> = {'begin'}
<proc> = {'procedure'}
<statement> = {'if','while','call','read','write','l','begin'}
<lexp> = {'odd','(','l','d'}
<exp> = {'+','-','(','l','d'}
<term> = {'(','l','d'}
<factor> = {'(','l','d'}
```

```
<lop> = {'=','<>','<','<=','>','>='}
<aop> = {'+','-'}
<mop> = {'*','/'}
<id> = {'l'}
<integer> = {'d'}
```

l 代表字母，d 代表数字。
接着对得到的单词进行匹配。

1. 获取单词，存到全局 alpha 中，若获取成功，跳转到 2, 否则跳转到 4
2. 对于每个词法分析器得到的字母 a，当前非终结符 A，若 $a \in FIRST(\alpha_i)$（α_i 为非终结符 A 的产生式），则用 α_i 去匹配 a。跳转到 1
3. 若不存在，由于 PL/0 语言不存在产生式为空，所以这是一种语法错误，输出对应的错误。跳转到 1
4. 结束

在语法分析阶段，每一个非终结符都是一个过程，我们递归的调用非终结符形成的过程，对每个字母进行匹配。

2.3 语义分析与中间代码产生器

2.3.1 任务

语法分析的任务是对语法分析所识别出的各类语法范畴分析其含义，并进行初步翻译，产生中间代码。包括两方面的工作：语法分析和中间代码的生成。

2.3.2 设计思路

该阶段主要借助属性文法和翻译模式进行分析和产生代码。该阶段除了语法分析程序，还有表格管理程序对过程中的符号表进行变量、常量和过程标识符的说明与引用之间的信息联系。符号表用于记录该过程中的变量，常量，标识符等信息，具体我们的设计的符号表为：
表头：

前一张表（pre） 宽度（width） 层次（level） 程序起始地址（quad）

表项：

名字（name）	类型（type）	偏址（offset）	数值（value）

为过程定义几个函数：

```
mktable:
输入:
pre 指向一张先前创建的符号表
idname 该过程的名称
layer 该过程的层次
功能:
在表格管理程序中创建一张新的符号表，记录过程中的变量，常量等信息，放着表格栈的顶端

pop:
功能:
将符号表栈栈顶的符号表出栈

enter:
输入:
idname 变量/常量名
type 类型
offset 偏址
value 值
功能:
在表格栈顶端的表格中加入新的一项（idname，type，offset，value）

enterproc:
输入:
idname 过程名
offset 偏址
value 该过程的符号表
功能:
在表格栈顶端的表格中加入新的一项（idname，’ table ‘，value）

lookup:
输入:
idname 变量/常量名称
输出:
变量/常量对应的入口
功能:
寻找idname对应变量/常量的入口，以便对变量/常量的访问
具体实现:
1.l=0，t=符号表栈栈顶的符号表
2.若t=None，跳转到4，;否则在t中寻找该变量/常量，若存在，则offset=该变量/常量在符号表的位置，typ
3.返回offset，type，l
4.输出错误（未定义）

```

```
emit:
输入:
code 三地址代码
功能:
生成中间代码

newtemp:
输出:
新临时变量ti
功能:
创建临时变量

backpath:
输入:
nextlist 回填列表
quad 回填地址
功能:
将回填列表中的第四部分用quad替换
```

创建空转换:

```
用于创建第一个符号表，插入到prog过程中的block前
M -- $\epsilon$
{t = mkable(nil);
push(t, tblpter);
push(0, offset);}

用于创建新的符号表，插入到proc括号前
N -- $\epsilon$
{t = mkable(top(tblptr));
push(t, tblpter);
push(0, offset);}

用于获取下一条代码地址，插入到body的statement之后（每个）/当statement翻译为if时，插入到then之后
H -- $\epsilon$
{H.quad = nextquad;}
```

设计属性文法:
语法分析过程我们采用自顶向下的分析方法，每当一个过程分析完后，我们就知道了我们用那些单词对该过程进行归约，即可调用每个过程对应的语义分析来产生中间代码。
在这个任务中，对于 P 代码，我们生成的中间代码将访问常量直接翻译为常数，对于变量，翻译为相对于当前符号表的层差与偏址。同时在每个过程 body 部分的开始阶段和结束阶段设置标识符（idname、end）

属性名	作用
<id>.name	变量/常量名称
<statement>.nextlist	回填列表
<lexp>.truelist	判断真出口
<lexp>.truelist	判断假出口
<exp>.place	运算量入口
<term>.place	运算量入口
<factor>.place	运算量入口
<lop>.c	判断
<aop>.c	加减运算符
<mop>.c	乘除运算符
<integer>.c	常数值

中间代码：

```
0 : suml
1 : j|v|1|0 = 10
2 : sum|v|2|1 = 0
3 : j<= j|v|1|0 x|v|0|0 5
4 : j - - 10
5 : t0 = sum|v|2|1 + j|v|1|0
6 : sum|v|2|1 = t0
7 : t1 = j|v|1|0 + 1
8 : j|v|1|0 = t1
9 : j - - 3
10 : write sum|v|2|1 a|c|5
11 : end
12 : xi
13 : read x|v|3|0
14 : write x|v|3|0
15 : read j|v|1|0
16 : write j|v|1|0
17 : param j|v|1|0
18 : call suml 1 1
19 : write j|v|1|0
20 : end
```

2.4 目标代码生成器

2.4.1 任务

将语义分析产生的中间代码转换为目标机上的目标代码。

2.4.2 设计思路

目标代码生成器通过识别每条中间代码，将中间代码转换为目标 P 代码。由于目标代码与中间代码并不是 11 对应的，所以在生成目标代码时候需要记录每个中间代码产生目标代码时对应的起点，以便于后续对跳转指令进行回填。
对于中间代码的几种形式，进行如下转换：
变量/常量提取代码:

1. 首先判断是否为临时变量，若是临时变量，则不生成代码
2. 若不是，根据输入判断类型
3. 若是常量，生成 LIT 代码
4. 若是变量，生成 LOD 代码

变量/常量存取代码:

1. 首先判断是否为临时变量，若是临时变量，则不生成代码
2. 若不是，根据输入判断类型，若是变量，生成 STO 代码

write 代码:
对于 write 后面的每个参数，先使用变量/常量提取代码，在生成 WRT 0 0 代码
read 代码:
对于 read 后面每个参数，生成 RED l a 代码
param 代码:
生成变量/常量提取代码
call 代码:

1. 生成 CAL l a 代码
2. 生成常量提取代码，将行参数放着数据栈顶
3. 生成无条件跳转指令

有条件跳转代码:

1. 生成两个数据提取代码
2. 根据判断类型生成 OPR 代码
3. 生成有条件跳转代码

赋值代码:

1. 生成变量/常量提取代码
2. 生产变量/常量存取代码

运算代码:

1. 生成两个变量/常量提取代码

2. 生成根据运算类型 OPR 代码

idname end:
在中间代码生成时，设计了过程开始与结束阶段的标识符 (idname, end)，当识别带 idname 时，生成 INT 0 a 代码，通过查询符号表来找到 a; 当识别到 end 时，生成 OPR 0 0 代码
运行结果：

```
0 : JMP 0 25
1 : INT 0 6
2 : LIT 0 10
3 : STO 0 1
4 : LIT 0 0
5 : STO 1 2
6 : LOD 0 1
7 : LOD 0 0
8 : OPR 0 13
9 : JPC 0 11
10 : JMP 0 20
11 : LOD 1 2
12 : LOD 0 1
13 : OPR 0 2
14 : STO 1 2
15 : LOD 0 1
16 : LIT 0 1
17 : OPR 0 2
18 : STO 0 1
19 : JMP 0 6
20 : LOD 1 2
21 : WRT 0 0
22 : LIT 0 5
23 : WRT 0 0
24 : OPR 0 0
25 : INT 0 7
26 : RED 0 3
27 : LOD 0 3
28 : WRT 0 0
29 : RED 0 1
30 : LOD 0 1
31 : WRT 0 0
32 : LOD 0 1
33 : CAL 1 1
34 : LIT 0 1
35 : JMP 0 1
36 : LOD 0 1
```

```
37 : WRT 0 0
38 : OPR 0 0
```

2.5 解释器

2.5.1 任务

将产生的目标代码进行解释运行.

2.5.2 设计思路

解释器通过识别每一条 P 代码进行解释运行。运行环境为:

1. 两个存储器（code 存储 P 代码，stack 数据栈）
2. 寄存器 I 存储当前运行的代码
3. 寄存器 T 存储当前数据栈栈顶
4. 寄存器 B 存储当前活动记录基地址
5. 寄存器 P 存储下一条要运行的代码

具体语句解释思路:

指令	l	a	对应操作	I	T	B
LIT	0	a	将常数 a 直接放入数据栈栈顶	I=P+1	T=T+1	B=B
OPR	0	0	跳转回原过程	I=S[B+2]	T=B-S[B+3]	B=S[B+1]
		2	加法运算，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		3	减法运算，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		4	乘法运算，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		5	除法运算，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		6	奇数判断	I=P+1	T=T	B=B
		8	相等判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		9	不等判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B

		10	小于判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		11	大于等于判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		12	大于判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
		13	小于等于判断，取出栈顶两个数， 操作数 1 为次栈顶元素，操作数 2 为栈顶元素， 结果放到栈顶	I=P+1	T=T-1	B=B
LOD	l	a	递归寻找基地址 add: while l > 0 add=S[add],l=l-1 将 S[add + a + 3 + S[add + 3]] 处值放到栈顶	I=P+1	T=T+1	B=B
STO	l	a	递归寻找基地址 add: while l > 0: add=S[add],l=l-1 将栈顶值放到 S[add + a + 3 + S[add + 3]] 处	I=P+1	T=T-1	B=B
CAL	l	a	调用过程，a 为被调用过程的过程体 在目标程序区的入口地址，l 为层差， 当层差为 0 时， 被调用过程的直接外层与该过程相同 当层差为 0 时， 被调用过程的直接外层为该过程	I=P+1	T=T+4	B=T+1
INT	0	a	为过程开辟空间	I=P+1	T=B+a-1	B=B
JMP	0	a	无条件跳转	I=a	T=T	B=B
JPC	0	a	条件跳转，当栈顶为 1 时，跳转到 a， 否则不跳转	if top == 1: I=a else:I=P+1	T=T-1	B=B
RED	l	a	递归寻找基地址 add: while l > 0: add=S[add],l=l-1 将读入的值放到 S[add + a + 3 + S[add + 3]] 处	I=P+1	T=T	B=B
WRT	0	0	将栈顶处的值输出	I=P+1	T=T-1	B=B

2.6 错误处理

2.6.1 词法分析阶段

词法分析阶段运用超前搜索的思想，当读取到 < 时，向后在读一个元素，当该元素为 >，则为不相等判断符号 <>, 若为 =，则为 >=，否则退回到 <; 同理，当读到 > 也进行如此处理。
词法分析可能遇到的错误为：

1. 非法字符
2. 在：后为跟着 =
3. 数字中含有字母

2.6.2 语法分析阶段

语法分析阶段运用恐慌模式，当遇到读到的字母符号不在当前分析的 First 集时，先判断该字母符号是否在下一个要分析的 First 集，如果在，则输出错误缺少一个当前分析的 First 集中的字母符号，否则，则为输入错误，输出该错误，跳过该字符。
语法分析可能遇到的错误为：

1. 缺少;
2. 程序没有以 program 开始
3. 缺少 const
4. 缺少 var
5. ：= 输入错误
6. 缺少（
7. 缺少）
8. 缺少 begin
9. 缺少 end
10. 缺少 then
11. 缺少 do
12. 未完成 <statement> 的要求
13. 未完成 <factor> 的要求
14. 符号不属于 <lop>
15. 符号不属于 <aop>
16. 符号不属于 <mop>
17. 未完成 $< exp >$ 的要求

18. 应该输入以字母开头的字符串，输入了数字
19. 关键字拼写错误（当输入满足关键字前缀表达式时）

2.6.3 语义分析阶段

语义分析阶段主要要进行静态的语义检查，如变量是否定义，常量是否在表达式左边等。语义分析可能遇到的错误为：

1. 变量/常量未定义
2. 过程未定义
3. 常量位于表达式的左边
4. 过程调用含有太多参数
5. 过程调用参数不足

3 课设总结

本次课程设计我深入了解了编译器的运行原理以及设计中可能遇到的难点。本次课程设计主要分为两个阶段。第一阶段的主要任务是设计词法分析器和语法分析器。由于上学期已经学习了《形式语言与自动机》这么课程，设计词法分析时遇到的困难比较小，基本上能看着自动机的图顺利的完成。在语法分析阶段，一开始能设计出不含错误处理的递归程序，但是对于恐慌模式的错误处理不太理解，所以花费时间比较久。随后，通过阅读一些资料，慢慢了解了恐慌模式在实际编译过程的运用。但是自己考虑的情况还是比较少，没有考虑到较全面的编程过程中可能出现的问题，所以对于有些问题不能指出并是程序正常的运行。
第二阶段我们主要设计编译器的语法分析与中间代码产生器，目标代码生成器以及解释器。首先设计语法分析与中间代码生成器，跟着上课讲的翻译模式可以理解大致的应该如何设计语法分析器，但是遇到了一个困难，就是 lookup 这个函数的返回值应该是什么。在上课中我们讲的是这个变量的入口，但是入口的形式应该是怎么样的并没有具体的讲解。后来通过看目标代码 P 代码，想到了我们在实际运行中是以层差和偏址来表示变量的位置的，所以就将层差和偏址设置为 lookup 函数的返回值。在这个过程中，也渐渐理解了后面讲解的静态链的实际意义，也明白了为什么要在创建符号表的时候在表头放入符号表的前一个符号表，是为了后续进行变量寻找的时候设计的。在解决了这个问题后，就很容易了解到符号表在设计的时候每个表头的信息都是有意义的，我们都会在后续的过程的运用到。而采用一些好的编程方式的优点就是当你发现前面忘记设计一些东西，后面进行更改的时候会比较容易。
接着是设计目标代码生成器。在这个阶段，因为自己对中间代码和 P 代码之间的关系并不是很理解，所以一开始翻译便出现了问题，不知道怎么下手。所以我先设计了解释器，通过设计解释器，来理解 P 代码的运行逻辑，以及他们之间的关系。设计解释器时，对静态链一直存在很大的困惑。理解静态链的作用以及理解静态链的使用方法是一个比较容易的过程。但是有一个难点就是不太理解静态链是如何生成的。虽然课程中有对静态链进行讲解和举例，但是在课程中对静态链的生成时，我们是直接假设，而不是通过 P 代码，所以存在一定问题。我们直接假设的时候，存在很多先验的知识，就是我们知道这个过程的直接外层是什么，知道他在栈中的位置，所以写的话可以很容易写出来。但是当让程序机械化运行的时

候，就比较迷惑，不知道如何使用。后来通过国防科技大学老师对静态链讲解的时候中，明白了静态链如何机械化的直接生成，解决了这个问题。设计完解释器后，就可以深入理解 P 代码，对于后端编译器的设计就比较简单就完成了。

我觉得本次课程设计比较大的困难就是很多知识课程中已经学习了，但是到实际的设计过程中，我们还比较难把这些知识串接起来，需要自己查阅比较多的资料来串接。第二个难点是每个模块虽然相对独立，但是又互相联系，所以前面设计的时候需要思考到后面的设计，如我产生中间代码的时候要考虑到中间代码怎么转换为目标代码。

通过这次课程设计，我逐渐明白了我们现在所使用的编译器是凝结了无数前人的智慧的成果。在编写报告的时候，需要自己回头对代码进行一些检查，可以发现自己在错误分析部分还有较大的缺陷，并不能很好的，很全面的发现问题。同时自己设计的编译器还有一些缺陷，没有对错误进行有效的处理。同时，我觉得设计测试用例也是一项比较有挑战性的工作，因为很多时候我们需要出错了才能知道自己错了。所以测试用例也是程序设计中重要的一部分。[1]

参考文献

[1] 钱家骅 and 孙永强. 程序设计语言编译原理. 国防工业出版社, 1984.