

编译原理 I

课程设计报告

班级： 1618104

学号： 161810225

姓名： 王烨文

2020-12

1. 设计任务

● 课程设计题目

一个 PASCAL 语言子集 (PL/0) 编译器的设计与实现

● PL/0 语言的 BNF 描述 (扩充的巴克斯范式表示法)

```

<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> → 1{1|d}    (注: 1 表示字母)
<integer> → d{d}

```

注释:

<prog>: 程序 ; <block>: 块、程序体 ; <condecl>: 常量说明 ; <const>: 常量;
 <vardecl>: 变量说明 ; <proc>: 分程序 ; <body>: 复合语句 ; <statement>: 语句;
 <exp>: 表达式 ; <lexp>: 条件 ; <term>: 项 ; <factor>: 因子 ; <aop>: 加法运算符;
 <mop>: 乘法运算符; <lop>: 关系运算符。

● 假想目标机的代码

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

代码的具体形式:

F	L	A
---	---	---

其中：F 段代表伪操作码

L 段代表调用层与说明层的层差值

A 段代表位移量（相对地址）

进一步说明：

INT：为被调用的过程（包括主过程）在运行栈 S 中开辟数据区，这时 A 段为所需数据单元个数（包括三个连接数据）；L 段恒为 0。

CAL：调用过程，这时 A 段为被调用过程的过程体（过程体之前一条指令）在目标程序区的入口地址。

LIT：将常量送到运行栈 S 的栈顶，这时 A 段为常量值。

LOD：将变量送到运行栈 S 的栈顶，这时 A 段为变量所在说明层中的相对位置。

STO：将运行栈 S 的栈顶内容送入某个变量单元中，A 段为变量所在说明层中的相对位置。

JMP：无条件转移，这时 A 段为转向地址（目标程序）。

JPC：条件转移，当运行栈 S 的栈顶的布尔值为假（0）时，则转向 A 段所指目标程序地址；否则顺序执行。

OPR：关系或算术运算，A 段指明具体运算，例如 A=2 代表算术运算“+”；A=12 代表关系运算“>”；A=16 代表“读入”操作等等。运算对象取自运行栈 S 的栈顶及次栈顶。

● 假想机的结构

两个存储器：存储器 CODE，用来存放 P 的代码

数据存储器 STACK（栈）用来动态分配数据空间

四个寄存器：

一个指令寄存器 I：存放当前要执行的代码

一个栈顶指示器寄存器 T：指向数据栈 STACK 的栈顶

一个基地址寄存器 B：存放当前运行过程的数据区在 STACK 中的起始地址

一个程序地址寄存器 P：存放下一条要执行的指令地址

该假想机没有供运算用的寄存器。所有运算都要在数据栈 STACK 的栈顶两个单元之间进行，并用运算结果取代原来的两个运算对象而保留在栈顶。

● 活动记录：

RA
DL
SL

RA：返回地址

DL：调用者的活动记录首地址

SL：保存该过程直接外层的活动记录首地址

过程返回可以看成是执行一个特殊的 OPR 运算

注意：层次差为调用层次与定义层次的差值

● 程序实现要求：

PL/0 语言可以看成 PASCAL 语言的子集，它的编译程序是一个编译解释执行系统。PL/0 的目标程序为假想栈式计算机的汇编语言，与具体计算机无关。

PL/0 的编译程序和目标程序的解释执行程序可以采用 C、C++、Java 等高级语言书写。

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。

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。

2. 系统设计

● 词法分析器的实现

词法分析器是指代码在初始进行编译过程时，通过扫描代码所有文字对所有文字进行分类填表后，比如说对应于数字的 NUM 类型、对应于变量命名的 IDENTITY 类型以及对于保留字如 PROCEDURE、VAR 等的保留字类型，其目的是为了后面语法分析器对于语法分析时的方便。

具体的实现方式是，设计了三个获取字符的函数，分别是 GetChar（获取下一个字符），GetBC（获取下一个非空字符），backchar（获取前一个字符），在进行字符扫描时，对于保留字如果这一字符串符合保留字条件则将其表示为保留字，如果不符合则对他进行变量的分析，如果其开头为数字则表明这是一个错误的变量命名，反之则表明这是一个合法的变量命名。其他的分析和这一分析类似。并通过全局变量 row 与 col 来定义这一变量命名的位置，从而在语法分析与语义分析时可以直接报错定位。

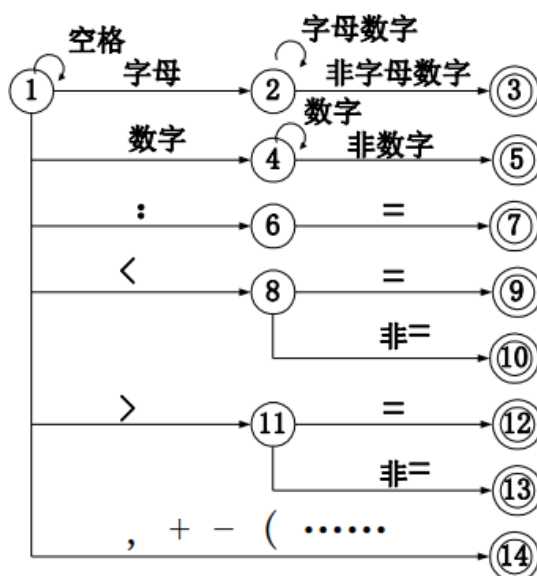


图 1.1 词法分析过程

如图 1.1 所示区分出了不同的保留字，数字，变量命名以及符号。

在进行 pl/0 语言分析时，我们先进行词法分析，将词法分析所得到的单词存入图 1.2 所示的单词表中，再到语法分析中进行单词的语法分析。

```

struct syntax {
    string symbol;
    string identity;
    int line;
    int col;
} Syn[10000];
  
```

图 1.2 词法分析单词表结构

```
Microsoft Visual Studio 调试控制台
8 8 x--identity
8 9 >--lop
8 11 0--num
8 13 do--do
9 2 begin--begin
10 3 if--if
10 6 odd--odd
10 10 x--identity
10 12 then--then
11 4 sum--identity
11 7 :=--Assign
11 9 sum--identity
11 12 +--aop
11 13 x--identity
12 3 else--else
13 4 sum--identity
13 7 :=--Assign
13 9 sum--identity
13 12 ---aop
13 13 x--identity
13 14 ;--semicolon
14 3 x--identity
14 4 :=--Assign
14 6 x--identity
14 7 ---aop
14 8 1--num
15 2 end--end
15 5 ;--semicolon
16 2 flag--identity
16 6 :=--Assign
```

图 1.3 词法分析所得单词。

- 语法分析器的实现

- 语法分析的主要过程

首先语法分析采用的是书上的递归下降分析，其主要实现方法是通过非终结符进行一个函数的设计用来检验某一非终结符中的语法是否符合规范，若其符合规范则返回到当前非终结符中检验接下来的子串。通过递归调用的方法，减少了程序的代码量。

其具体的语法图如下，

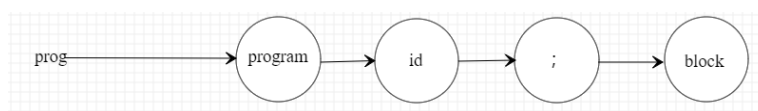


图 2.1 prog 语法图

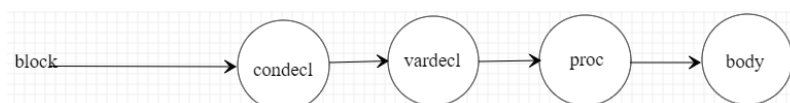


图 2.2 block 语法图

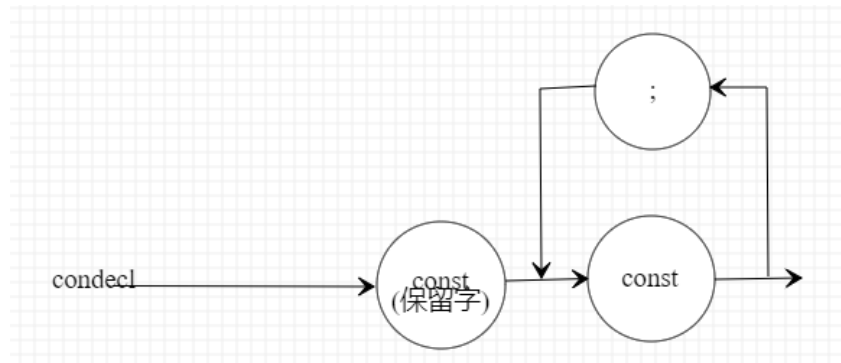


图 2.3 condecl 语法图

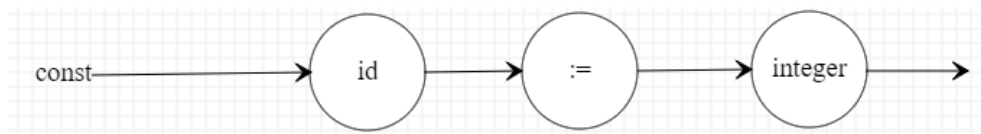


图 2.4 const 语法图

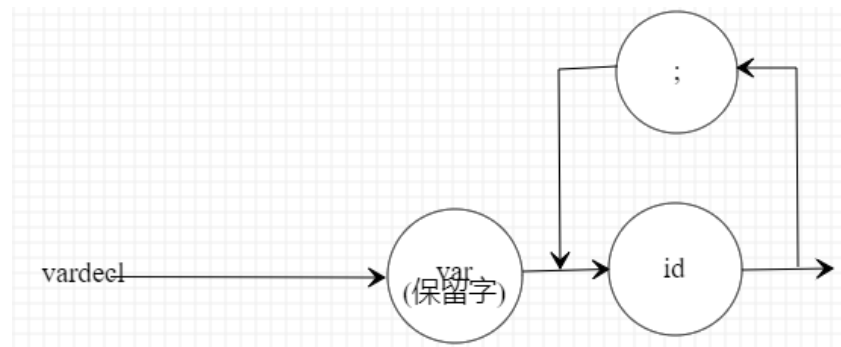


图 2.5 vardecl 语法图

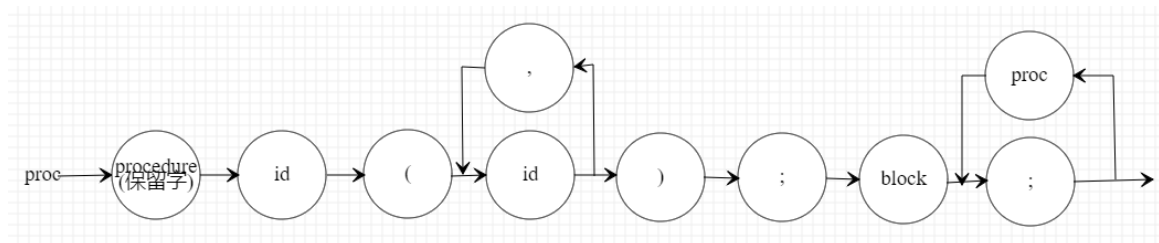


图 2.6 proc 语法图

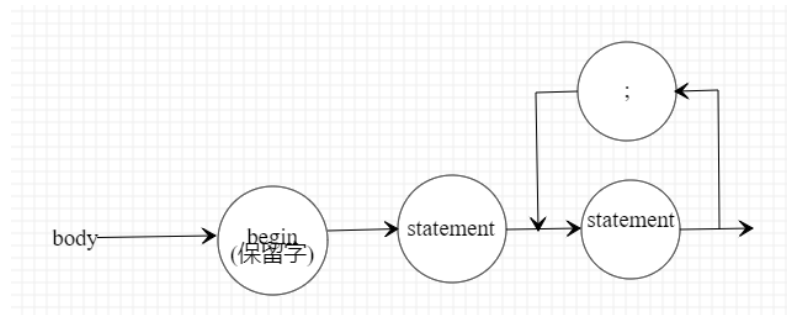


图 2.7 body 语法图

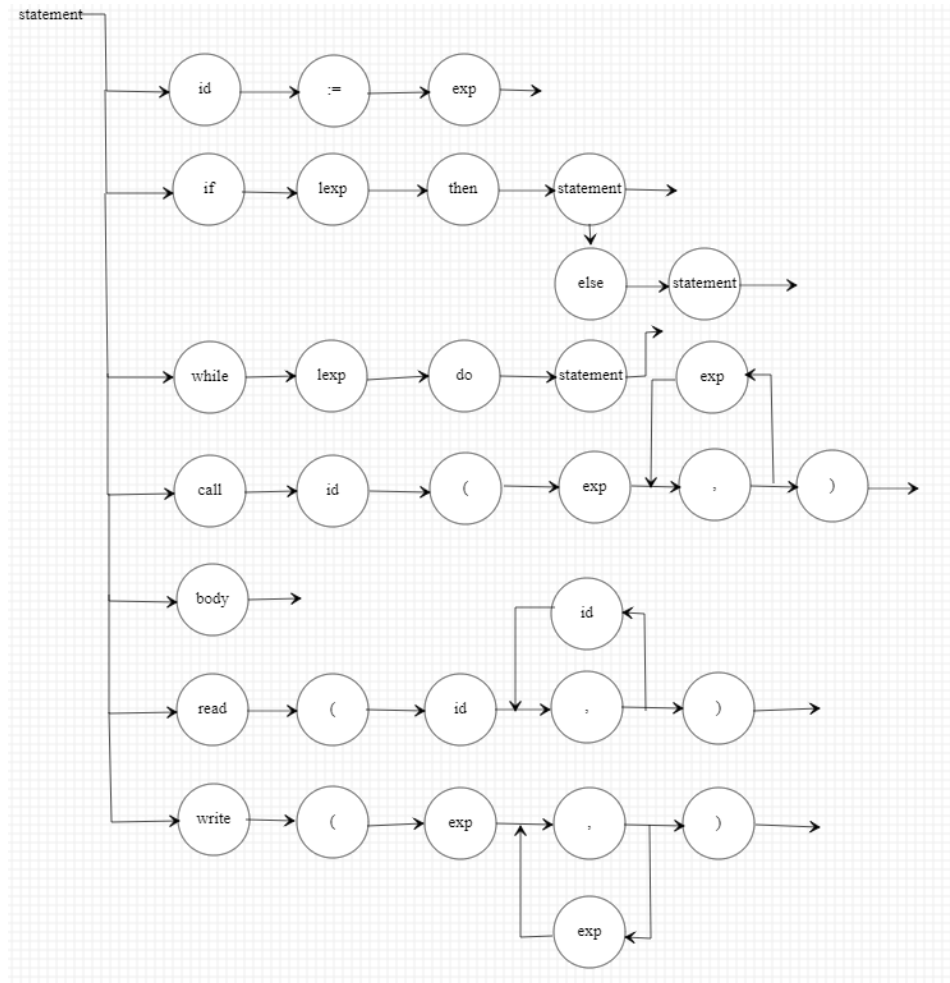


图 2.8 statement 语法图

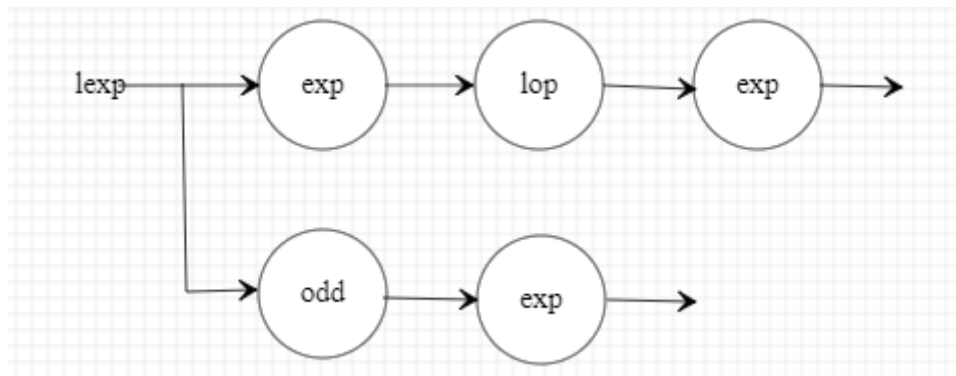


图 2.9 lexp 语法图

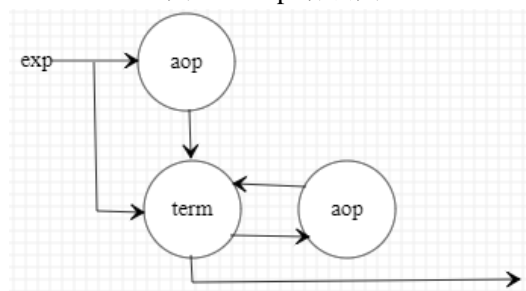


图 2.10 exp 语法图

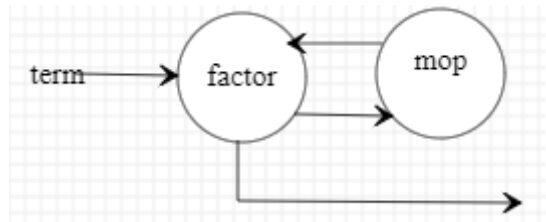


图 2.11 term 语法图

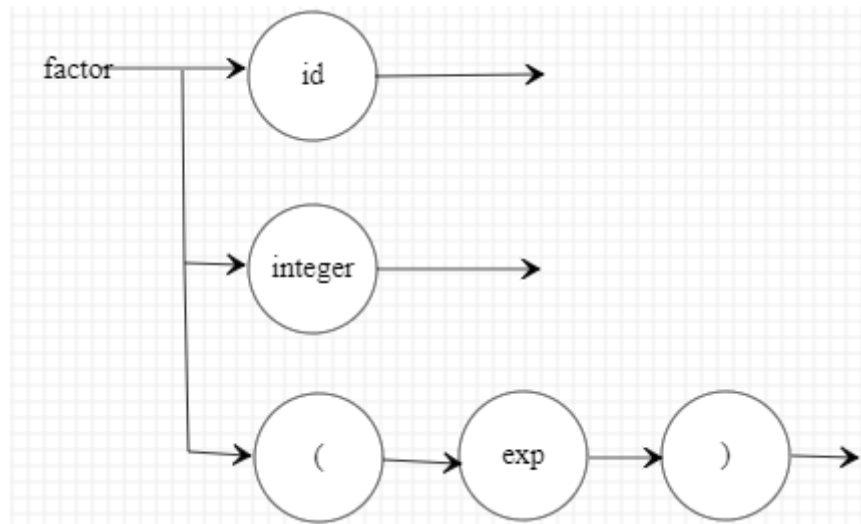


图 2.12 factor 语法图

lop 语法图

=,<,<=,>,>=,<>

aop 语法图

+, -

mop 语法图

/, *

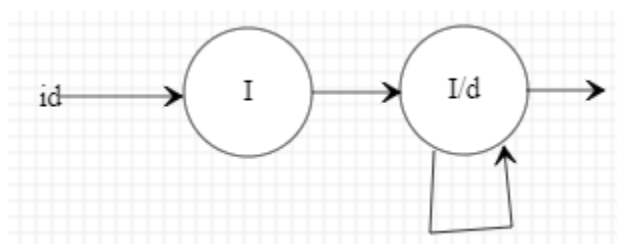


图 2.13 id 语法图

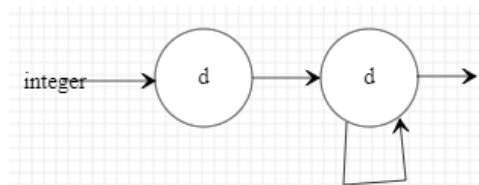


图 2.14 integer 语法图

■ 语法分析纠错的主要方法

本次课设着重处理的错误情况主要分为两种，一种是符号的缺失，一种是符号的错误，比如 `statement` 中的 `id:=exp`，如果 `:=` 缺失则报错 `:=`，如果 `:=` 错误则报错错误。

实现这一目的的主要方法是通过先计算出非终结符的 `First` 集合与 `Follow` 集合，然后再进行递归程序语法分析时，判断跟在当前单词之后的是否为下一个合理的非终结符的 `First` 集合或者是合理的终结符。如果当前是，则是出现了符号丢失的问题，如果下一个单词才是，则表明是符号错误的问题。

对于整个非终结符的缺失主要做了 `term` 中 `factor` 的缺失，其方法与上述大同小异，是通过判断当前符号之后的符号是否为该非终结符的 `Follow` 集或者是合理的终结符。

```
program xi
const a:=10;
var b,c;
procedure p(x);
begin
    c:=+a
    write(c);
    c:=x+y;
    write(c)
end
begin
    read(b)
    while b<>0 do
    begin
        call p(1,2);
        read(c);
        write(2*);
        read(b)
    end
end
```

图 2.15 测试代码用例图

```
C:\Users\44165\source\repos\Project7\Debug\Project7.exe
15 8 (---leftbrackets
15 9 l--num
15 10 ,--comma
15 11 2--num
15 12 )--rightbrackets
15 13 ;--semicolon
16 2 read--read
16 6 (---leftbrackets
16 7 c--identity
16 8 )--rightbrackets
16 9 ;--semicolon
17 2 write--write
17 7 (---leftbrackets
17 8 2--num
17 9 *--mop
17 10 )--rightbrackets
17 11 ;--semicolon
18 2 read--read
18 6 (---leftbrackets
18 7 b--identity
18 8 )--rightbrackets
19 1 end--end
20 0 end--end
2行0列语法错误, 缺少;!
4行14列语法错误, 缺少id!
7行1列语法错误, 缺少;!
8行6列变量y未定义!
13行1列语法错误, 缺少;!
17行10列语法错误, 缺少factor因子!
请输入数据:
```

图 2.16 测试代码结果

如图 2.15 与图 2.16 所示为上述代码在语法分析中的表现，可以看出缺失 `program` 之后缺少 `;`, `procedure` 中间缺少参数，`c:=+a` 缺少分号，`y` 在变量中没有定义(这是语义分析的处理)，`read(b)` 缺少分号，`write(2*)` 在乘号之后缺少 `factor`

因子。

在经过了各自模块的语法分析之后，我们通过一个递归表达来找到当前非终结符最近的 Follow 集中的终结符，从而保证语法分析的流畅性。当然这一要求无法检验出程序第一个字符是否存在问题。这是一点遗憾的地方，这是我在设计语法分析时没有考虑到的地方。

● 语义分析器的实现

■ 符号表的设计

符号表的设计采用树形结构，在一开始进行符号表设计时本来想采用线性结构并用层次嵌套关系进行比对，但在进行演算后还是决定采用树形结构，这一结构数据结构简单，但对于推算并不友好。

首先是图 3.1 所示树形符号表的详细数据结构。

```
struct table {
    string identity; //定义名称
    string type; //类型-const等
    int val; //值
    int addr; //偏移量
};

struct tablelink {
    vector<table> level_table;
    int level;
}Tablelink[200]; //变量表
```

图 3.1 树符号表结构

采用的是显示建立了一个普遍使用的符号信息结构体，符号信息结构体中，identity 存的是变量定义的名称，用于后面进行变量重名的比较，type 存的是变量的类型，比如 const，var，procedure，如果是 const 的话在进行翻译模式的时候可以直接使用 LIT 存入数据，而不需要 LOD 在目标代码运行时查找变量。Tablelink 则是整张符号表的树形结构。

```
int fa_table[100]; //父表
```

图 3.2 父节点存储结构

其中，我通过图 3.2 所示并查集思想的数据结构存储了数据表的父表位置。

符号表添加的思路是通过 block 和 proc 的嵌套传输当前的层次，对于一个子程序的建立，一定建立在先后顺序，所以不存在先建立后面程序的符号表再建立前面程序的符号表，或者重复建表，所以我们可以大胆地对每个子程序进行符号表的建立，在建立了符号表后，我们用一个 now_level 变量来确定当前 proc 的嵌套深度（跟随程序嵌套加深减小），用 now_table 标识这是建立的第几张表，建立的表与已有的表建立父子关系的方法，是通过函数嵌套传入的 pre_table，因为程序是顺序进行表的建立的，所以我们可以每次嵌套的 pre_table 一定是定义该子程序的程序所在的表，所以我们将子表指向父表，即可完成父子关系的建立。

通过这些方法，我建立起来了属性的符号表结构，从而大幅度减轻了线性

符号表所带来的顺序查找的时间太久等问题,以及难以判断子程序与其他子程序的子程序如果具有重名却报错的问题。

如图 3.3 所示的程序,wyw3 中定义了 wyw4, wyw6 作为 wyw3 的同级程序也定义了 wyw4,这是合理的函数定义方式。其编译成功,表明树形符号表建立完成且正确!

```

58 1 d--identity
58 2 :--Assign
58 4 1--num
58 5 ;--semicolon
59 1 while--while
59 7 d--identity
59 8 <--lop
59 10 c--identity
59 12 do--do
60 1 begin--begin
61 2 call--call
61 7 wyw2--identity
61 11 (--leftbrackets
61 12 d--identity
61 13 )--rightbrackets
61 14 ;--semicolon
62 2 write--write
62 7 (--leftbrackets
62 8 flag--identity
62 12 )--rightbrackets
62 13 ;--semicolon
63 2 d--identity
63 3 :--Assign
63 5 d--identity
63 6 +--aop
63 7 1--num
64 1 end--end
65 0 end--end
编译成功
请输入数据:

```

```

6.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

x1:=x1-1
end;
flag:=sum1
end;
procedure wyw2(q);
  procedure wyw3(p);
    procedure wyw4(o);
      procedure wyw5(r);
        begin
          call wywnb(r);
        end
      end
    begin
      call wyw5(o);
    end
  begin
    call wyw4(p)
  end;
procedure wyw6(l);
  procedure wyw4(s);
    begin
      call wyw3(s);
    end
  begin
    call wyw4(l);
  end
end

```

图 3.3 复杂层级测试代码

```

58 1 d--identity
58 2 :--Assign
58 4 1--num
58 5 ;--semicolon
59 1 while--while
59 7 d--identity
59 8 <--lop
59 10 c--identity
59 12 do--do
60 1 begin--begin
61 2 call--call
61 7 wyw2--identity
61 11 (--leftbrackets
61 12 d--identity
61 13 )--rightbrackets
61 14 ;--semicolon
62 2 write--write
62 7 (--leftbrackets
62 8 flag--identity
62 12 )--rightbrackets
62 13 ;--semicolon
63 2 d--identity
63 3 :--Assign
63 5 d--identity
63 6 +--aop
63 7 1--num
64 1 end--end
65 0 end--end
18行10列变量wywnb重复定义!
请输入数据:

```

```

6.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

procedure wywnb(x);
  var sum;
  begin
    sum:=0;
    while x>=0 do
      begin
        if odd x then
          sum:=sum+x
        else
          sum:=sum-x;
        x:=x-1
      end;
      flag:=sum
    end;
  procedure wywnb(x1);
    var sum1;
    begin
      sum1:=0;
      while x1>=0 do
        begin
          if odd x1 then
            sum1:=sum1+x1
          else
            sum1:=sum1-x1;
          x1:=x1-1
        end;
        flag:=sum1
      end;
    end;
  end;
end;

```

图 3.4 纠错代码测试

图 3.4 是重复变量定义。(同样适用于参数)

在设计符号表层次的时候有过一个难点，就是同级程序的符号表，由于符号表的顺序建立，其查找成了个较为麻烦的事情，于是我定义了一个变量为 `check_table`，这一变量 `proc` 模块中进行新表建立后，我们只需要通过 `now_table` 查找 `pre_table`，于是此时的 `check_table` 就是 `now_table`，而在 `proc` 的 `block` 语句结束后，我们则需要回到父表，相当于层次减一，做法是通过递归调用传参 `pre_table` 实现。

■ 翻译模式的设计

翻译模式的设计相较于课上讲的较为简单，我增加了两个目标代码，一个是 `CALL`，一个是 `RET`，当然这两个操作都已通过 `OPR` 来实现，`CALL` 的作用是当第一次遇到 `call` 语句时，就进行活动记录栈顶的静态链，动态链和 `sp` 的入栈，然后再在栈顶对函数传参的表达式进行栈顶计算，因为对于一个函数调用数据，在堆栈中，它的结构特殊性决定了他是属于符号表最前面的那几个变量，于是就是静态链之上的数据空间，于是我们直接在栈顶进行计算就能完成函数的传参。

`RET` 语句则是通过当前栈顶的活动记录进行该函数的返回。

其他的主要是目标代码的改造，其中最大的改造是 `CAL` 语句的改造，在符号表阶段我通过 `now_code` 变量记录目标代码的数目，当进行到 `procedure` 时，我通过确定当前代码的生成数目，直接填入 `CAL` 的 `a` 数据段，然后也存入符号表中函数定义变量的 `val` 属性之中，从而在后面进行下一层次时，仍然可以通过符号表的父子关系得到需要跳转到的目标代码位置。

其余方面和课上讲的几乎没有差别。

上面所述的将 `const` 的 `LOD` 改变为 `LIT` 也是我设计的一种方法。

■ 活动记录的设计

活动记录的设计按照书上的静态链结构进行设计，其方法是通过调用子程序，将子程序指向其能作用的已有活动记录的最近一层。

所以在此设计过程中我将函数调用分为三类来进行推算。

1. 同级调用。

同级静态链调用方法较为简单。如下图代码

Proc A

.....

Proc B

.....

Call A

在 `B` 子程序中调用 `A` 程序，`B` 和 `A` 所属 `level` 层次相同。我们在进入 `B` 程序后，`B` 必定会指向他所作用的静态链，而他所作用的静态链和 `A` 所作用的静态链由于 `AB` 所属层次相同，所以 `B` 调用 `A` 后直接将 `A` 的静态链指向 `B` 的静态链所指向的栈内地址即可。如图 3.5 所示。

```

sp = top; // 栈顶
act[top] = lsp; // 动态链
top++;
act[top] = top - 2; // 返回地址
top++;
if (level_proc[Code[T].a] == s3.top()) // 同级调用
    act[top] = act[lsp + 2]; // 静态链

```

图 3.5 同级调用代码

2. 层次低函数调用层次高函数。即调用子程序。

由于 p1/0 代码的特殊性，在一个定义的子程序中，他所能调用的比该子程序低的程序只能是其自身的子程序，他无法调用其他同级程序之中的子程序，且无法调用其自身的子程序中的子程序，所以这一设计十分简单，直接指向调用他的活动记录静态链的地址（与第一种不同，第一种指向的是静态链中的内容，这一种指向的是静态链所存在的位置的下标）即可。

```

else if (level_proc[Code[T].a] > s3.top()) // 调用子程序
    act[top] = lsp + 2;

```

图 3.6 子级调用代码

3. 层次高的函数调用前面已经设计并能为之调用的层次低的函数。

```

procedure wyw2(q);
    procedure wyw3(p);
        procedure wyw4(o);
            procedure wyw5(r);
                begin
                    call wywnb(r);
                end
            begin
                call wyw5(o);
            end
        begin
            call wyw4(p)
        end;
    procedure wyw6(l);
        procedure wyw4(s);
            begin
                call wyw3(s);
            end
        begin
            call wyw4(l);
        end
    begin
        call wyw6(q)
    end
end

```

图 3.7 复杂层次测试代码

图 3.7 代码中的体现是 wyw6 的子程序 wyw4 调用了 wyw6 的同级程序 wyw3。

这一设计较为繁琐，我的设计思路是，首先我们先得从被调用的 wyw3 找

到 `wyw4` 的静态链，当作一次层次低函数调用层次高的函数，然后根据 `wyw3` 和 `wyw4` 的层次差，向前找层次差次静态链，这里就是前两种情况的结合。所以最后综合得到的办法就是向前找 `wyw3` 和 `wyw4` 层次差+1 次静态链指向。

我们可以通过数学归纳法证明上面方法的正确性。

首先最普遍的情况，最简单的情況，就如上面情况肯定正确。

其次我们假设 `B` 调用 `A` 为正确的。我们通过 `B` 调用 `A`，`A` 调用 `C`，我们无论哪种情况都可以先让 `C` 返回找到 `A`，然后问题又变回 `B` 调用 `A` 的子问题，从而证明正确。（但其实不严谨，我们可以通过画图证明）

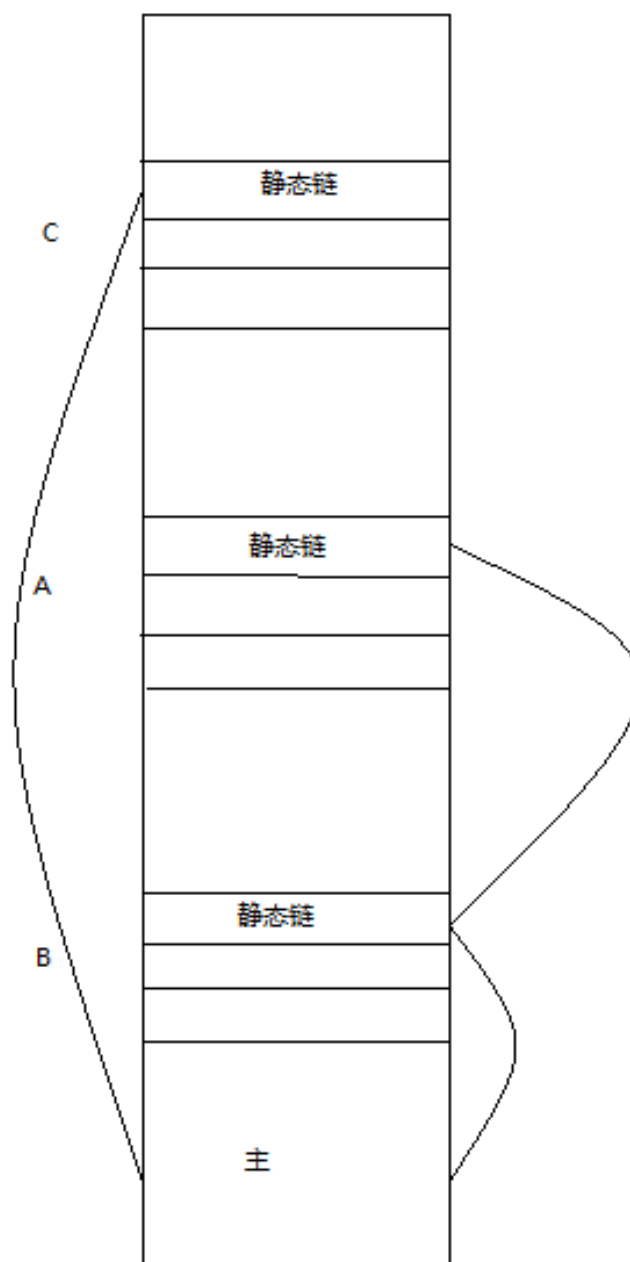


图 3.8 复杂层次调用图示

如图 3.8 所示, 假设 B 调用了 A, C 与 B 是同级程序, 我们可以通过 C 与 A 的层差从 A 找到 B 的作用范围, 就找到了 C 的作用范围, 于是我们只需要 C 与 A 层差+1 次向前搜索就能完成静态链的处理。同样, 如果中间出现了其他不同层次的函数, 但由于我们在调用该程序之前的所有活动记录都指向到了正确的作用范围, 所以完全不用担心找不到或者找错。这也是一个数学归纳法的运用! 最后我将这种方法归为本质是寻找同级程序的过程, 因为一个程序被调用必然在之前会被他层次相同会更低的调用过。我们只要找到与这个函数层次相同的函数就可以完成静态链的正确指向。

代码体现为图 3.9

```

act[top] = lsp + 2;
else if (level_proc[Code[T].a] < s3.top()) { //姊姊调用子
int t = s3.top() - level_proc[Code[T].a] + 1;
int m = lsp + 2;
while (t-->0) {
m = act[m];
}
act[top] = m;
}
s3.push(level_proc[Code[T].a]);

```

图 3.9 复杂调用代码

我设立了一个堆栈用于存储每一个进行的子程序的层次, 由于程序的顺序调用, 不用担心计算层次差出现错误。

■ 整体的设计

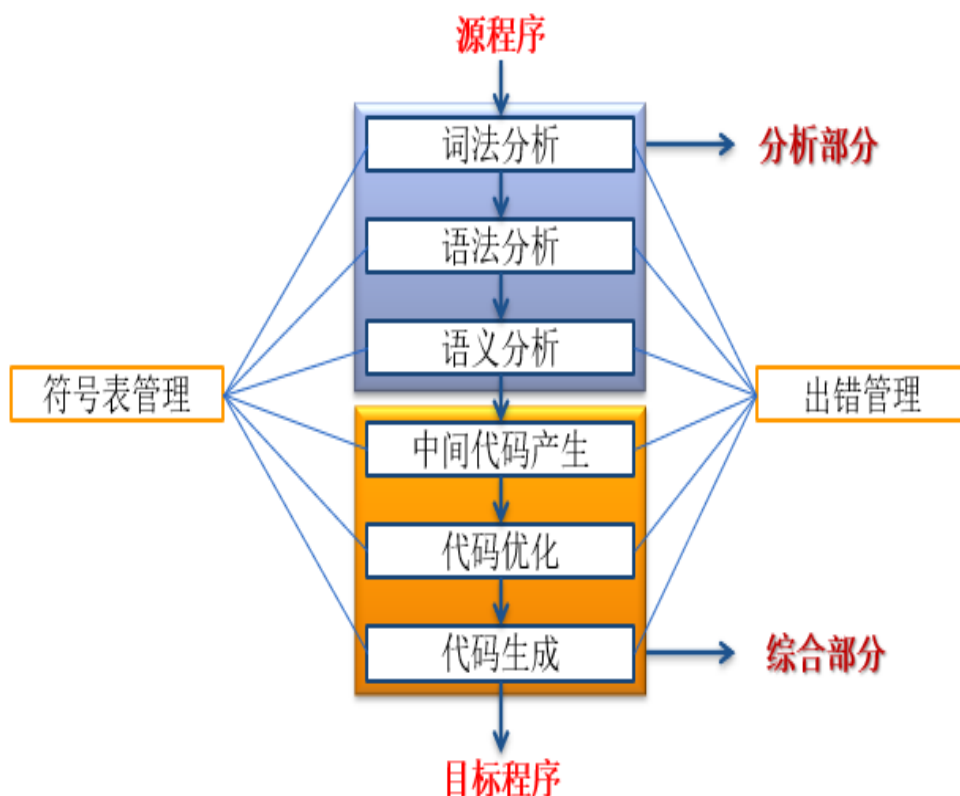


图 4.1 整体结构的设计

如图 4.1 是整体结构的设计。

3. 课设总结

本次课程设计从一开始的词法分析器到语法分析器，然后再到最后的语义分析器，目标代码运行程序，这一套设计流程让我对编译原理课程有了更深入的了解。从一开始老师上课讲 Pascal 语言写的词法分析器，我盲人摸象着使用 C++写了个实现单词鉴别的词法分析器，然后到语法分析器的时候，由于一开始没有进行良好的逻辑整理，对于报错的处理我并没有很好的设计，好在后来我通过改变思路，主要处理缺失和错误两种错误，灵活使用 First 与 Follow 集合进行语法分析，从而使得完成了语法分析的主要纠错。在这一段时间中，由于自己立场不坚定，写出来了两份不同的语法分析器代码，在经历取舍后选择了一份进行最后的完成。在课设阶段，由于课上对于符号表设计比较有思路，直接抛弃了线性符号表的设计，采用树形结构，这一部分灵感来源于老师上课讲的遇到一个 Procedure 就创建一个新表。然后设计翻译模式，这边的翻译模式相较于书上的较为简单，回填也更方便。最后是活动记录这一模块，这一模块对于我来说是花费时间最长的，因为我在写第一次的时候发现我对静态链的概念出现了问题，我将静态链永远指向前一个调用程序，使得我出现了许多错误，好在认真看书之后，经过画图和数据刻画之后完成了严谨的逻辑推导和数学演算。最后实现了整个课设。

最后感谢李皓琨同学的测试代码，以及谢强老师课上的悉心讲解。

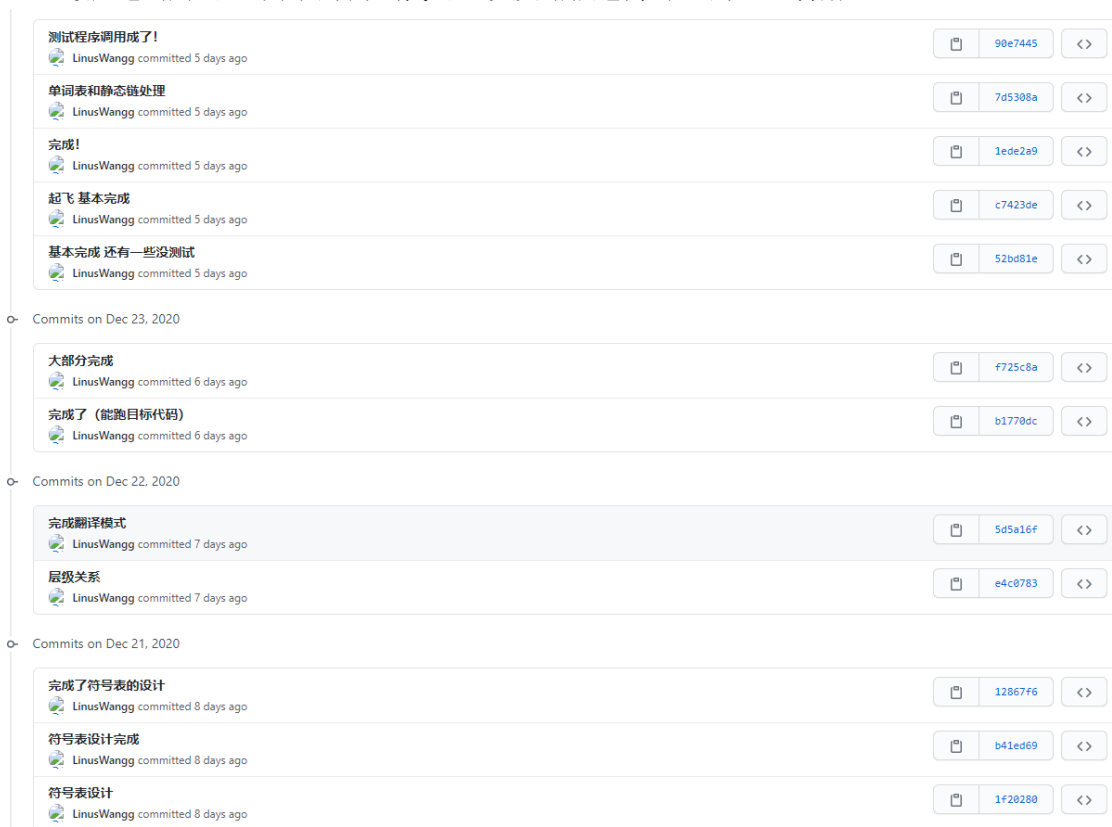


图 5.1 代码记录

最后感谢 github 对我代码记录和 commit 记录的保存。经过四天，每天三小时到五小时的思路设计和 bug 修复，我对编译原理有了更深入的理解，也对自己的代码能力有了提高。

4. 实验结果

a) 斐波那契数列

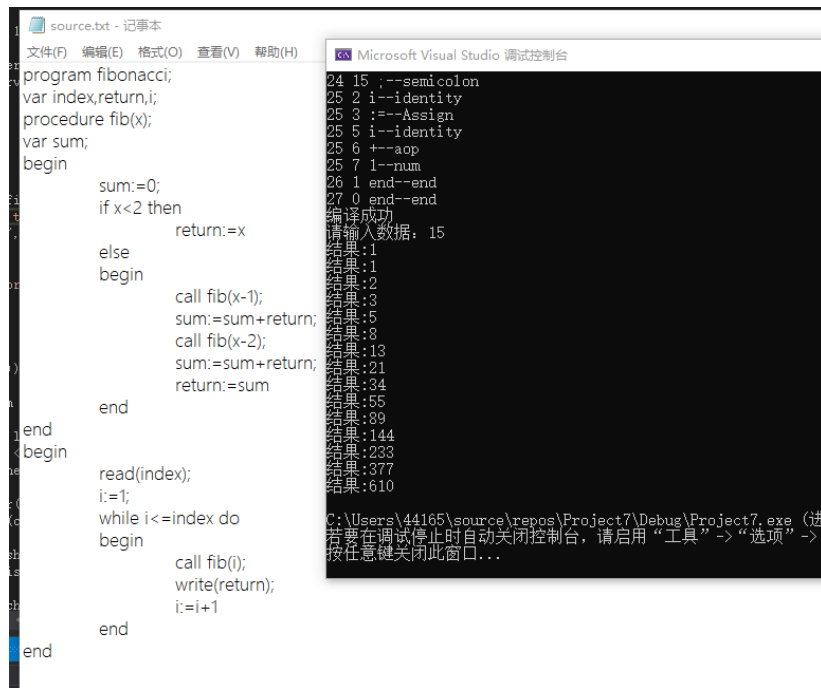


图 6.1 斐波那契数列测试

b) 复杂程序调用代码测试

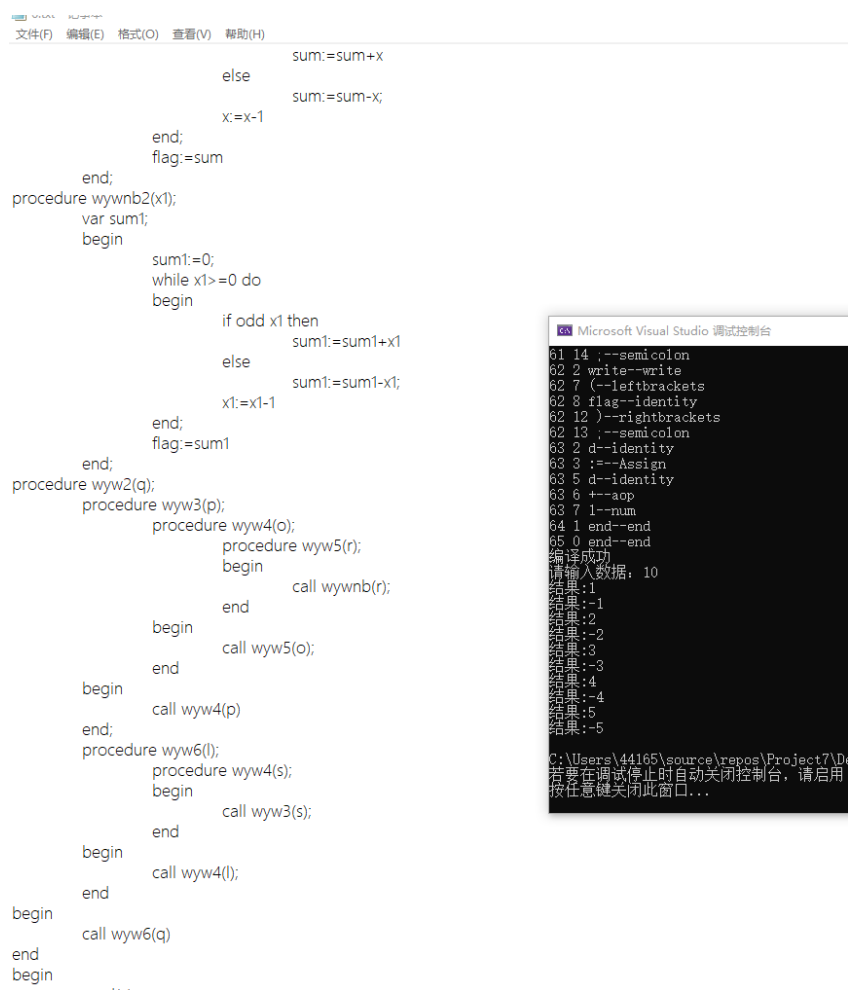
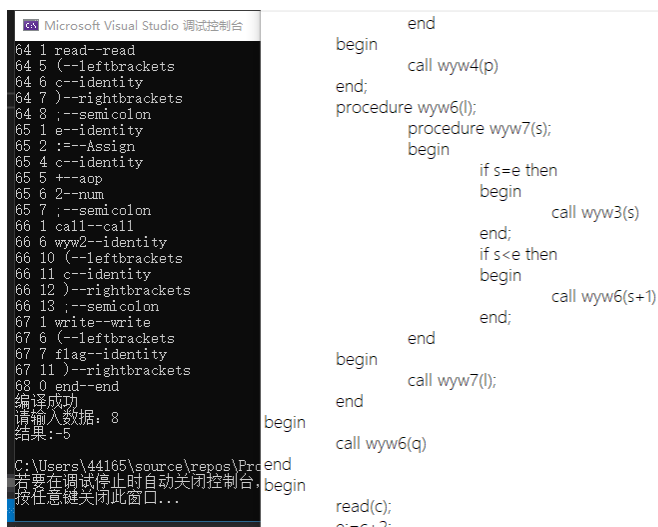


图 6.2 复杂代码测试

c) 更复杂程序调用代码测试



The screenshot shows the Microsoft Visual Studio 调试控制台 (Debug Console) with the following content:

```
64 1 read--read
64 5 (--leftbrackets
64 6 c--identity
64 7)--rightbrackets
64 8 ;--semicolon
65 1 e--identity
65 2 :--Assign
65 4 c--identity
65 5 +--aop
65 6 2--num
65 7 ;--semicolon
66 1 call--call
66 6 wyw2--identity
66 10 (--leftbrackets
66 11 c--identity
66 12)--rightbrackets
66 13 ;--semicolon
67 1 write--write
67 6 (--leftbrackets
67 7 flag--identity
67 11)--rightbrackets
68 0 end--end
编译成功
请输入数据: 8
结果: -5
C:\Users\44165\source\repos\Proend
若要在调试停止时自动关闭控制台,
按任意键关闭此窗口...
```

On the right side of the image, the corresponding Pascal code is shown:

```
end
begin
    call wyw4(p)
end;
procedure wyw6(l);
    procedure wyw7(s);
        begin
            if s=e then
                begin
                    call wyw3(s)
                end;
            if s<e then
                begin
                    call wyw6(s+1)
                end;
        end
    end
begin
    call wyw7(l);
end
call wyw6(q)
end
begin
    read(c);
    e:=c+2;
```

图 6.3 更复杂代码测试

(对应于检验不同静态链问题处理的代码) 第一个程序为斐波那契数列, 第二个是输出 $1-n$ 的交错上升数, 第三个是输出 $n+2$ 的交错上升数。

5. 参考资料

《编译原理》(第三版) 陈火旺 刘春林