实验目的与要求：

一、理解并掌握 TINY 语言的语法规则，包括其基本语句结构和语法树构建方式。

二、熟悉 TINY 语法分析器的运行流程，包括如何编写 TINY 程序、运行 TINYParser 进行语法分析，以及观察得到正确的运行结果。

三、了解 TINY+语言相较于 TINY 语言的扩展，包括新增的声明语句和 while 循环语句，并掌握 TINY+的语法规则。

四、修改 TINY 语法分析器以适应 TINY+语言的语法规则，实现对 TINY+程序的语法分析，包括编写新的测试程序验证语法分析器的正确性。

五、综合实验内容，提升对编程语言语法分析器设计与实现的能力，加深对上下文无关文法的理解，并能将其应用于实际的语言处理任务中。

方法、步骤：

1、任务一：运行 TINY 语言的语法分析程序 TINYParser，理解 TINY 语言语法分析器的实现。

其中，TINY 语言的词法与实验二相同，TINY 语言的文法描述如下：

```
program -> stmt-seq
stmt-seq -> stmt-seq;stmt | stmt
stmt -> if-stmt|repeat-stmt|assign-stmt|read-stmt | write-stmt
if-stmt -> if exp then stmt-seq end | if exp then stmt-seq else stmt-seq end
repeat-stmt -> repeat stmt-seq until exp
assign-stmt -> id:= exp
read-stmt -> read id
write-stmt -> write exp
exp -> simp-exp cop simp-exp | simp-exp
cop -> < | =
simp-exp-> simp-exp addop term |term
term -> term mulop factor | factor
factor -> (exp) |num |id
addop -> + | -
mulop -> * | /
```

具体的语法树结构在 TINY_Syntax.pptx 里面描述，结合 TINYParser 代码理解语法树构造。

2、任务一要求：根据 TINY 语法，自己编写至少一个另外的 TINY 测试程序，运行 TINYParser 语法分析器，观察程序运行流程，得到正确的运行结果。

3、任务二：基于 TinyParser 语法分析器，实现拓展语言 TINY+的语法分析器。

其中，TINY+语言的词法与实验二相同，TINY+语言的文法描述如下（注：此处为了描述方便，对上下文无关文法的产生式表示进行了扩充，允许在产生式右部使用类似正则表达式的表示，例如第 5 条产生式右部花括号{ , identifier }代表*闭包），其中红色部分为 TINY+文法更新的部分，其余部分为 TINY 文法原有的产生式：）

```
1  program      -> declarations stmt-sequence
2  declarations -> decl ; declarations | ε
3  decl         -> type-specifier varlist
4  type-specifier-> int | bool | string | float | double

5  varlist      -> identifier { , identifier }
6  stmt-sequence    -> statement { ; statement }
7  statement    -> if-stmt | repeat-stmt | assign-stmt | read-stmt |
   write-stmt | while-stmt
8  while-stmt -> do stmt-sequence while bool-exp
9  if-stmt    -> if exp then stmt-seq end | if exp then stmt-seq else stmt-seq
   end
10 repeat-stmt  -> repeat stmt-sequence until exp
11 assign-stmt  -> identifier:=exp
12 read-stmt    -> read identifier
13 write-stmt   -> write exp
14 exp          -> simp-exp cop simp-exp | simp-exp
15 cop          ->   < | =
16 simp-exp->  simp-exp addop term |term
17 term      ->  term mulop factor | factor
18 factor    ->  (exp) |num |id
19 addop ->   + | -
20 mulop ->   * | /
```

TINY+语言的文法主要添加了声明语句及 while 语句

4、任务二要求：根据 TINY+语法，修改给定的 TINY 语法分析器，实现更新的 TINY+语法分析器，成功实现对上述示例程序的语法分析。并根据 TINY+文法的定义，编写至少一个另外的 TINY+测试程序，对该测试程序完成语法分析，得到正确的语法分析结果。

5、实验要求：

· 完成任务一及任务二的要求；

- 使用实验所提供的模板撰写实验报告，要求内容详实，有具体的设计描述、关键的代码片段、及实验结果屏幕截图；
- 在截止日期前将代码、实验报告、测试文件（如有）等所有实验相关文件压缩到一个压缩包姓名_学号_实验三.rar 上传至 Blackboard。

实验过程及内容：

1、理解 TINY 语言语法分析器的实现

1.1、逐句解析 TINY 语言的语法，如下所示：

①、程序 (program):

program -> stmt-seq

程序由一系列语句组成。

②、语句序列 (stmt-seq):

stmt-seq -> stmt-seq ; stmt | stmt

语句序列可以是一个语句，也可以是多个语句以分号分隔。

③、语句 (stmt):

stmt -> if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt

语句可以是 if 语句、repeat 语句、赋值语句、读取语句或写入语句。

④、if 语句 (if-stmt):

if-stmt -> if exp then stmt-seq end | if exp then stmt-seq else stmt-seq end

if 语句由一个条件表达式和一个语句序列组成，并且可以包含一个可选的 else 分支。

⑤、repeat 语句 (repeat-stmt):

repeat-stmt -> repeat stmt-seq until exp

repeat 语句由一个语句序列和一个直到条件组成。

⑥、赋值语句 (assign-stmt):

assign-stmt -> id := exp

赋值语句将一个表达式的值赋给一个标识符。

⑦、读取语句 (read-stmt):

read-stmt -> read id

读取语句从输入中读取一个值并赋给一个标识符。

⑧、写入语句 (write-stmt):

write-stmt -> write exp

写入语句将一个表达式的值输出。

⑨、表达式 (exp):

exp -> simp-exp cop simp-exp | simp-exp

表达式可以是一个简单表达式，也可以是两个简单表达式通过比较运算符组合而成。

⑩、比较运算符 (cop):

cop -> < | =

比较运算符包括小于和等于。

⑪、简单表达式 (simp-exp):

simp-exp -> simp-exp addop term | term

简单表达式可以是一个项，也可以是多个项通过加法运算符组合而成。

⑫、项 (term):

term -> term mulop factor | factor

项可以是一个因子，也可以是多个因子通过乘法运算符组合而成。

⑬、因子 (factor):

factor -> (exp) | num | id

因子可以是一个表达式、一个数字或一个标识符。

⑭、加法运算符 (addop):

addop -> + | -

加法运算符包括加号和减号。

⑮、乘法运算符 (mulop):

mulop -> * | /

乘法运算符包括乘号和除号。

1.2、整理 TINY 语法的特点：

①、TINY 语言是由一系列语句组成，语句由分号分割。

②、语句可以是 if 语句、repeat 语句、赋值语句、读取语句或写入语句。

③、TINY 语言没有声明。

④、数据类型只有整形，使用赋值语句对其声明。

⑤、表达式为布尔计算和整形计算。

⑥、花括号内为注释内容。

1.3、解析代码：递归下降解析器使用一组递归函数来处理每个语法规则。

代码中每个函数负责解析特定的语法规则，并将输入令牌流转换为抽象语法树 (AST) 的相应部分。

下图为我分析整理后得出的 TINY 语法分析结构图。

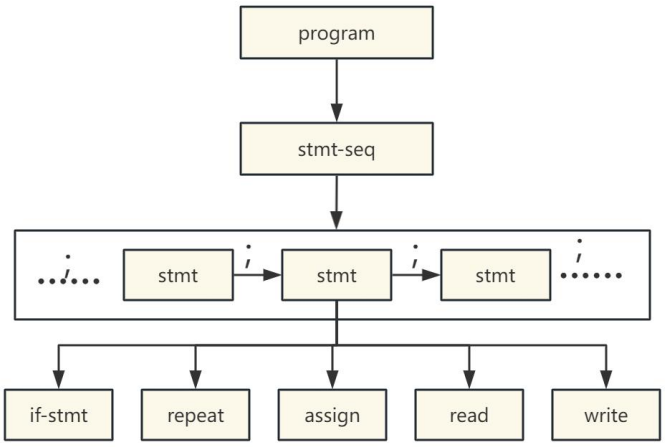


图 1 TINY 语法分析结构图

6 种语句对应树状结构如下图所示：

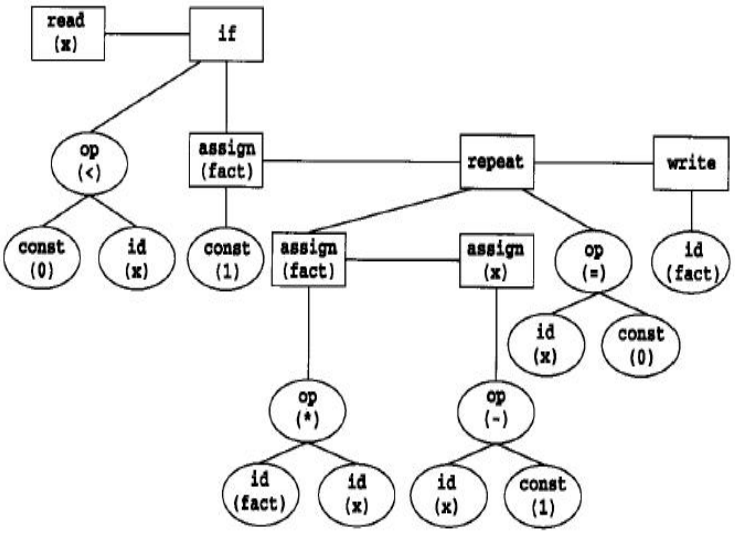


图 2 6 种语句的结构

1.4、分析 TINY Parser 代码实现逻辑

①、程序入口

TINYParser 的主函数 main.c 是程序的入口点。它接收一个参数，即要解析的文件名，然后打开这个文件。由于我们需要进行语法分析，所以常量 NO_PARSE 的值被设置为 FALSE。然后，调用 parse()函数进行语法分析，生成的语法树的根节点被保存在 syntaxTree 变量中。

```
50  #if NO_PARSE
51    while (getToken()!=ENDFILE);
52  #else
53    // 词法和语法分析入口，获得语法树
54    syntaxTree = parse();
55    if (TraceParse) {
56      fprintf(listing,"\nSyntax tree:\n");
57      printTree(syntaxTree);
58    }
```

图 3 程序入口

查看 parse.c 文件中的 parse 函数，分析逻辑：获取一个令牌（token），然后解析一系列语句并生成语法树。它检查是否已经到达文件的末尾，如果没有，那么就报告一个语法错误。最后，返回生成的语法树。

```
208  TreeNode * parse(void)
209  { TreeNode * t;
210    token = getToken();
211    t = stmt_sequence();
212    if (token!=ENDFILE)
213      syntaxError("Code ends before file\n");
214    return t;
215  }
```

图 4 parse 函数

②、分析各个语法结构

A、首先 stmt-seq 表示 A sequence of statements，根据产生式，遇到 stmt-seq 继续递归，遇到 stmt 就下降，得到的树结构为一个 stmt 序列。

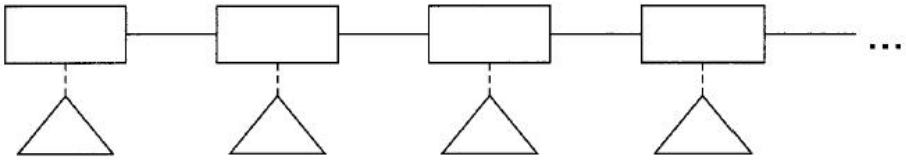


图 5 stmt-seq 对应的语法树结构

对应的程序如下图所示，首先，解析一个语句并将结果赋值给 t 和 p。然后，只要当前的令牌不是文件结束、语句结束、else 或 until，就会继续解析。每次循环，都会匹配一个分号，然后解析下一个语句并将结果赋值给 q。如果 q 不为空，会将 q 添加到语法树的末尾。最后，返回生成的语法树。

```
43  TreeNode * stmt_sequence(void)
44  { TreeNode * t = statement();
45    TreeNode * p = t;
46    while ((token!=ENDFILE) && (token!=END) &&
47           (token!=ELSE) && (token!=UNTIL))
48    { TreeNode * q;
49      match(SEMI);
50      q = statement();
51      if (q!=NULL) {
52        if (t==NULL) t = p = q;
53        else /* now p cannot be NULL either */
54        { p->sibling = q;
55          p = q;
56        }
57      }
58    }
59    return t;
60  }
```

图 6 stmt_sequence 生成语句序列

B、stmt 又分为 if-stmt，repeat-stmt，assign-stmt，read-stmt，write-stmt。对应的程序如下图所示：

```
62  TreeNode * statement(void)
63  { TreeNode * t = NULL;
64    switch (token) {
65      case IF : t = if_stmt(); break;
66      case REPEAT : t = repeat_stmt(); break;
67      case ID : t = assign_stmt(); break;
68      case READ : t = read_stmt(); break;
69      case WRITE : t = write_stmt(); break;
70       default : syntaxError("unexpected token -> ");
71                 printToken(token,tokenString);
72                 token = getToken();
73                 break;
74    } /* end case */
75    return t;
76  }
```

图 7 statement 匹配每种语句

C、if-stmt 对应的语法树结构如下图 8 所示，对应的程序如下图 9 所示。if_stmt 的函数用于解析 if 语句并生成一个语法树。首先，创建一个新的语句节点 t，然后匹配 if 关键字。如果 t 不为空，解析表达式并将结果赋值给 t 的第一个子节点。然后，匹配 then 关键字，并解析语句序列，将结果赋值给 t 的第二个子节点。如果当前令牌是 else，匹配

else 关键字，并解析语句序列，将结果赋值给 t 的第三个子节点。最后，匹配 end 关键字，并返回生成的语法树。

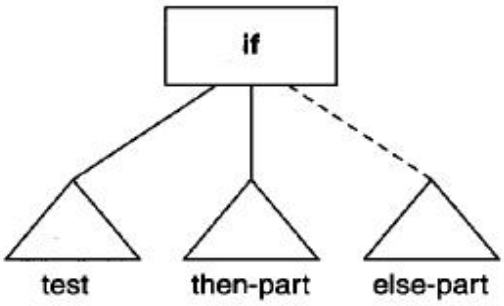


图 8 if-stmt 对应的语法树结构

```
78  TreeNode * if_stmt(void)
79  { TreeNode * t = newStmtNode(IfK);
80    match(IF);
81    if (t!=NULL) t->child[0] = exp();
82    match(THEN);
83    if (t!=NULL) t->child[1] = stmt_sequence();
84    if (token==ELSE) {
85      match(ELSE);
86      if (t!=NULL) t->child[2] = stmt_sequence();
87    }
88    match(END);
89    return t;
90  }
```

图 9 if_stmt 程序解析 if 语句并生成对应的语法树

D、repeat-stmt 对应的语法树结构如下图 10 所示，对应的程序如下图 11 所示。首先，创建一个新的语句节点 t，然后匹配 repeat 关键字。如果 t 不为空，解析语句序列并将结果赋值给 t 的第一个子节点。然后，匹配 until 关键字，并解析表达式，将结果赋值给 t 的第二个子节点。最后，返回生成的语法树。

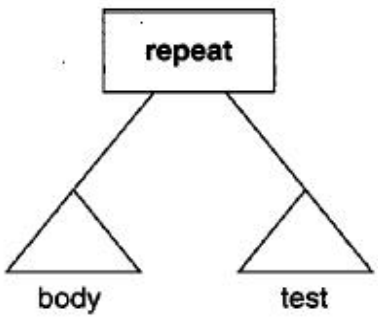


图 10 repeat-stmt 对应的语法树结构

```
92  TreeNode * repeat_stmt(void)
93  { TreeNode * t = newStmtNode(RepeatK);
94    match(REPEAT);
95    if (t!=NULL) t->child[0] = stmt_sequence();
96    match(UNTIL);
97    if (t!=NULL) t->child[1] = exp();
98    return t;
99  }
```

图 11 repeat-stmt 程序解析 repeat 语句并生成对应的语法树

E、assign-stmt 对应的语法树结构如下图 11 所示，对应的程序如下图 12 所示。首先，创建一个新的语句节点 t，然后如果 t 不为空且当前令牌是 ID，它将令牌字符串复制到 t 的属性名中。然后，匹配 ID 和赋值符号。如果 t 不为空，解析表达式并将结果赋值给 t 的第一个子节点。最后，返回生成的语法树。

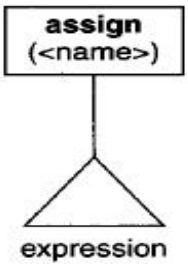


图 12 assign-stmt 对应的语法树结构

```
101   TreeNode * assign_stmt(void)
102   { TreeNode * t = newStmtNode(AssignK);
103     if ((t!=NULL) && (token==ID))
104       t->attr.name = copyString(tokenString);
105     match(ID);
106     match(ASSIGN);
107     if (t!=NULL) t->child[0] = exp();
108     return t;
109   }
```

图 13 assign-stmt 程序解析 assign 语句并生成对应的语法树

F、read_stmt 函数用于解析 read 语句并生成一个语法树。首先，它创建一个新的语句节点 t，然后匹配 read 关键字。如果 t 不为空且当前令牌是 ID，它将令牌字符串复制到 t 的属性名中。然后，它匹配 ID。最后，返回生成的语法树。

```
111   TreeNode * read_stmt(void)
112   { TreeNode * t = newStmtNode(ReadK);
113     match(READ);
114     if ((t!=NULL) && (token==ID))
115       t->attr.name = copyString(tokenString);
116     match(ID);
117     return t;
118   }
```

图 14 read-stmt 程序解析 read 语句并生成对应的语法树

G、write-stmt 对应的语法树结构如下图 12 所示，对应的程序如下图 13 所示。首先，创建一个新的语句节点 t，然后匹配 write 关键字。如果 t 不为空，解析表达式并将结果赋值给 t 的第一个子节点。最后，返回生成的语法树。

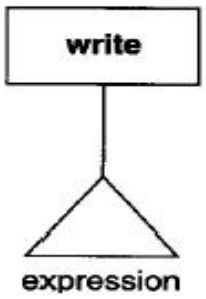


图 15 write-stmt 对应的语法树结构

```
120   TreeNode * write_stmt(void)
121   { TreeNode * t = newStmtNode(WriteK);
122     match(WRITE);
123     if (t!=NULL) t->child[0] = exp();
124     return t;
125   }
```

图 16 write-stmt 程序解析 write 语句并生成对应的语法树

H、对于表达式 exp，它由一条简单表达式加连接运算符加一条简单表达式或者一条简单表达式组成。运算符连接的表达式对应的结构树如下：

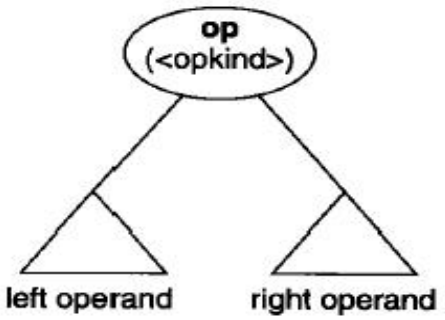


图 17 表达式对应的树结构

exp 函数如下图所示。首先，调用 simple_exp 函数解析一个简单的表达式，并将结果赋值给 t。如果当前的令牌是小于符号（LT）或等于符号（EQ），创建一个新的表达式节点 p，并将 t 和当前的令牌分别赋值给 p 的第一个子节点和属性。然后，匹配当前的令牌，并解析下一个简单的表达式，将结果赋值给 t 的第二个子节点。最后，返回生成的语法树。

```
127   TreeNode * exp(void)
128   { TreeNode * t = simple_exp();
129     if ((token==LT)||(token==EQ)) {
130       TreeNode * p = newExpNode(OpK);
131       if (p!=NULL) {
132         p->child[0] = t;
133         p->attr.op = token;
134         t = p;
135       }
136       match(token);
137       if (t!=NULL)
138         t->child[1] = simple_exp();
139     }
140     return t;
141   }
```

图 18 exp 函数解析表达式语句并生成对应的语法树

以上是对 TINY 程序代码的理解和介绍。

1.5、运行 TINY 程序，测试 tiny.txt 文件。

使用 linux 系统，终端输入指令 gcc main.c parse.c scan.c util.c -o main 编译程序，再使用./main 文件名 测试 tiny 文件。

测试结果如下图所示，可以正确进程词法分析和语法分析，构建语法树。

```
zhangweijing_2021152011@ubuntu-2204:~/桌面/tiny$ ./main tiny.txt

TINY COMPILATION: tiny.txt
   1: {A sample TINY program}
   2: read x;
        2: reserved word: read
        2: ID, name= x
        2: ;
   3: if 0<x then
        3: reserved word: if
        3: NUM, val= 0
        3: <
        3: ID, name= x
        3: reserved word: then
   4: fact:=1;
        4: ID, name= fact
        4: :=
        4: NUM, val= 1
        4: ;
   5: repeat
        5: reserved word: repeat
   6:   fact:=fact*x;
        6: ID, name= fact
        6: :=
        6: ID, name= fact
        6: *
```

图 19 测试 tiny.txt 文件，词法分析正确

```
Syntax tree:
  Read: x
  If
    Op: <
      Const: 0
      Id: x
    Assign to: fact
      Const: 1
    Repeat
      Assign to: fact
        Op: *
          Id: fact
          Id: x
      Assign to: x
        Op: -
          Id: x
          Const: 1
      Op: =
        Id: x
        Const: 0
    Write
      Id: fact
sh: 1: pause: not found
```

图 20 测试 tiny.txt 文件，成功构建语法树

1.6、自己编写一个测试文件测试。

自己编写测试文件时需要注意，repeat 子句里面的最后一条语句是不能加分号的。观察 stmt_sequence 这个函数可以发现，输入最后一条语句后，如果 token 是 until 就直接

返回了，不会再读取分号，同样的，如果一条语句后面是文件末尾、END 关键字、ELSE 关键字，那么也不能在这条语句的末尾写分号，这也解释了 tiny1.txt 中的 write 语句末尾不写分号的原因。

我编写的 tiny 测试文件如下所示：

```
{My TINY+ program}
read x;  { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

图 21 mytiny.txt 文件

mytiny.txt 文件的词法分析结果如下图，可以正确进行词法分析没有报错。

```
TINY COMPILATION: mytiny.txt
  1: {My TINY+ program}
  2: read x;  { input an integer }
       2: reserved word: read
       2: ID, name= x
       2: ;
  3: if 0 < x then { don't compute if x <= 0 }
       3: reserved word: if
       3: NUM, val= 0
       3: <
       3: ID, name= x
       3: reserved word: then
  4:   fact := 1;
       4: ID, name= fact
       4: :=
       4: NUM, val= 1
       4: ;
  5:   repeat
       5: reserved word: repeat
  6:     fact := fact * x;
       6: ID, name= fact
       6: :=
       6: ID, name= fact
       6: *
       6: ID, name= x
       6: ;
  7:     x := x - 1
       7: ID, name= x
       7: :=
       7: ID, name= x
       7: -
       7: NUM, val= 1
  8:   until x = 0;
       8: reserved word: until
       8: ID, name= x
       8: =
       8: NUM, val= 0
       8: ;
  9:   write fact { output factorial of x }
       9: reserved word: write
       9: ID, name= fact
 10: end
      10: reserved word: end
      11: EOF
```

图 22 mytiny.txt 文件的词法分析结果

mytiny.txt 文件的语法分析结果如下图，可以正确进行的语法分析，打印出正确的语法树。

```
Syntax tree:
  Program:
    Read: x
    If
      Op: <
        Const: Integer: 0
        Id: x
      Assign to: fact
        Const: Integer: 1
      Repeat
        Assign to: fact
          Op: *
            Id: fact
            Id: x
        Assign to: x
          Op: -
            Id: x
            Const: Integer: 1
        Op: =
          Id: x
          Const: Integer: 0
      Write
        Id: fact
sh: 1: pause: not found
```

图 23 mytiny.txt 文件的语法分析结果

2、基于 TinyParser 语法分析器，实现拓展语言 TINY+的语法分析器。

2.1、理解 TINY+语法

TINY+语法与 TINY 语言类似，不同之处：

①、TINY+增加了声明语句，语法规定是先声明变量（declarations），然后接着语句序列（stmt-sequence），这两部分的顺序是固定的；根据 TINY+语言的文法描述，声明部分是可选的，因此可以不用声明。

```
//一个程序由声明部分（declarations）和语句序列部分（stmt-sequence）组成
program -> declarations stmt-sequence
//声明部分由一个或多个声明（decl）组成，每个声明用分号分隔。声明部分可以为空（ε）。
declarations -> decl ; declarations | ε
```

图 24 TINY+增加了声明语句

②、一个声明由类型说明符（type-specifier）和变量列表（varlist）组成。类型说明符表示变量的类型，可以是 int、bool、string、float 或 double。变量列表由一个或多个标识符（identifier）组成，多个标识符之间用逗号（,）分隔。

```
// 一个声明由类型说明符（type-specifier）和变量列表（varlist）组成。
decl -> type-specifier varlist
// 类型说明符表示变量的类型，可以是int、bool、string、float或double。
type-specifier -> int | bool | string | float | double
// 变量列表由一个或多个标识符（identifier）组成，多个标识符之间用逗号（,）分隔。
varlist -> identifier { , identifier }
```

图 25 一个声明由类型说明符和变量列表组成

③、TINY+语言增加了 while 循环语句，使得可以使用 do-while 结构。While 语句的基本形式如下：

```
while-stmt -> do stmt-sequence while bool-exp
do                          // 循环体的开始
  <stmt-sequence>           // 在do和while之间的语句序列
while <bool-exp>            // bool-exp表示布尔表达式
```

图 26 while 循环语句

根据以上分析，重新构建 program 的语法树，如下图所示，红色框部分即为 TINY+语法分析器需要添加的内容：

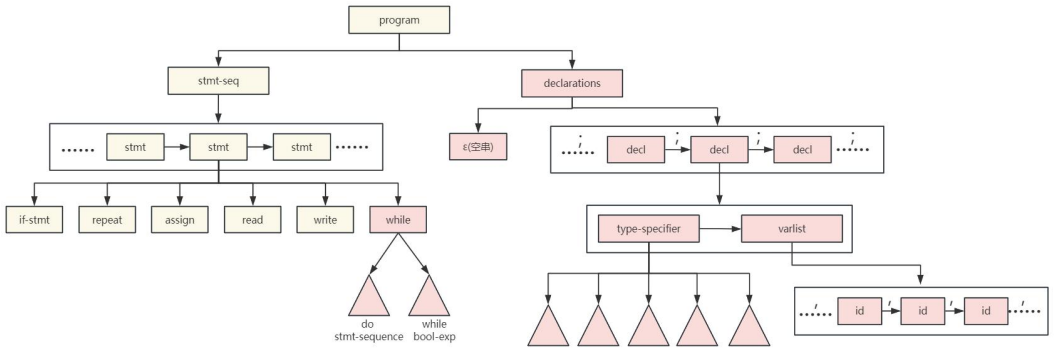


图 27 TINY+的语法结构

2.2、实现 TINY+语法分析器

①、添加 TINYParse 中缺少的词法分析部分

首先在 globals.h 文件中添加 double 和 float 两个关键字，并且修改 MAXRESERVED 的数值，代表关键字的数量。如下图所示：

```
/* MAXRESERVED = the number of reserved words */
#define MAXRESERVED 20          ②、修改关键字的数量

typedef enum
    /* book-keeping tokens */
   {ENDFILE,ERROR,
    /* reserved words */
    IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,   ①、新增FLOAT和DOUBLE
  T_TRUE,T_FALSE,OR,AND,NOT,INT,BOOL,STRING,DO,WHILE FLOAT,DOUBLE,
    /* multicharacter tokens */
    ID,NUM,STR,
    /* special symbols */
    ASSIGN,EQ,LT,GT,LTE,GTE,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI,COMMA,SQM

   } TokenType;
```

图 28 在 globals.h 文件中添加 double 和 float 两个关键字

还需要在 scan.c 中的关键字查找表中，添加 double 和 float 两个对应的关键字。

```
    {"string",STRING},
    {"bool",BOOL},
    {"float", FLOAT},    // 新增
    {"double", DOUBLE}, // 新增
    {"do",DO},
    {"while",WHILE}
    };
```

图 29 在 scan.c 中的关键字查找表中，添加 double 和 float

最后在 util.c 的打印函数中添加对这两种关键字的处理，以实现正确的词法分析。

```
34      case STRING:
35      case FLOAT:
36      case DOUBLE:
37      case DO:
38      case WHILE:
39          fprintf(listing,
40            "reserved word: %s\n",tokenString);
```

图 30 在 util.c 的打印函数中添加对这两种关键字的处理

接下来进行语法分析拓展，从递归的最底层开始添加代码。

②、while-stmt 解析 while 语句并生成对应的语法树

while-stmt -> do stmt-sequence while bool-exp 这条产生式，它的语法树如下：

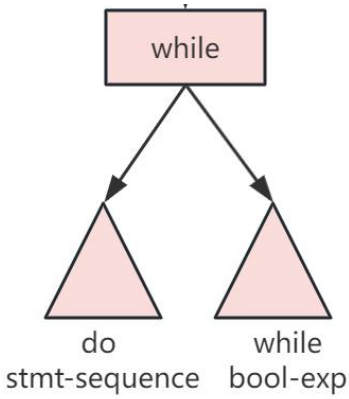


图 31 while-stmt 对应的语法树

首先在 globals.h 中添加语句类型节点 WhileK，如下图所示：

```
50  // 新增
51  typedef enum {StmtK,ExpK,DeclK,ProK} NodeKind;//节点类型
52  typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK,WhileK} StmtKind;//语句节点
53  typedef enum {OpK,ConstK,IdK,SpecifierK,StringK} ExpKind;//表达式节点
```

图 32 添加语句类型节点 WhileK

```
73  // 判断关键字，据当前的令牌（token）来决定调用哪个函数来解析语句。
74  TreeNode * statement(void)
75  { TreeNode * t = NULL;
76    switch (token) {
77      case IF : t = if_stmt(); break;
78      case REPEAT : t = repeat_stmt(); break;
79      case ID : t = assign_stmt(); break;
80      case READ : t = read_stmt(); break;
81      case WRITE : t = write_stmt(); break;
82      // 新增
83      case DO : t = while_stmt(); break;
84      default : syntaxError("unexpected token -> ");
85                printToken(token,tokenString);
86                token = getToken();
87                break;
88    } /* end case */
89    return t;
90  }
```

图 33 在 statement 函数中添加 while_stmt 这一分支

然后在 statement 函数中添加 while_stmt 这一分支，当读取到 DO 时，进入 while_stmt 函数进行解析，代码如图 33 所示。

最后还需要在 util.c 的打印函数中添加 WhileK 节点，如下图所示：

```
216         case WhileK:
217           fprintf(listing,"While\n");
218           break;
219         default:
220           fprintf(listing,"Unknown ExpNode kind\n");
221           break;
```

图 34 在 util.c 的打印函数中添加 whileK 节点

④、type-specifier 解析类型说明符。

type-specifier -> int | bool | string | float | double。

在 Tiny+语言中，类型说明符可以是 int、bool、string、float 或 double。

为了实现以上语法，需要在 globals.h 文件中添加表达式节点 SpecifierK 和表达式类型，如下图所示。

```
50  // 新增
51  typedef enum {StmtK,ExpK,DeclK,ProK} NodeKind;//节点类型
52  typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK,DoK,WhileK} StmtKind;//语句节点
53  typedef enum {OpK,ConstK,IdK,SpecifierK} ExpKind;//表达式节点
54
55  // 新增
56  /* ExpType is used for type checking */
57  typedef enum {Void,Integer,Boolean,Double,Float,String} ExpType; //表达式类型
```

图 35 添加表达式节点和表达式类型

编写 type_specifier 函数，首先创建一个新的表达式节点，然后根据当前的令牌（token）来确定类型说明符的类型，可能的类型包括 Integer、Boolean、String、Float 或 Double。如果遇到意外的令牌，函数会打印一个语法错误，并获取下一个令牌。最后，函数返回解析出的类型说明符。

```
238 //新增，类型说明符
239 TreeNode* type_specifier(void){
240   TreeNode *t = newExpNode(SpecifierK);
241   switch (token){
242     case INT:t->type = Integer;match(INT);break;
243     case BOOL:t->type = Boolean;match(BOOL);break;
244     case STRING:t->type = String;match(STRING);break;
245     case FLOAT:t->type = Float;match(FLOAT);break;
246     case DOUBLE:t->type = Double;match(DOUBLE);break;
247     default:
248       syntaxError("unexpected token -> ");
249       printToken(token,tokenString);
250       token = getToken();
251       break;
252     }
253     return t;
254 }
```

图 36 类型说明符

接着在 util.c 的打印函数中添加对应的内容，如下图所示：

```
238         case IdK:
239           fprintf(listing,"Id: %s\n",tree->attr.name);
240           break;
241         case SpecifierK:
242           switch(tree->type){
243             case Integer:
244               fprintf(listing,"Type: int\n");
245               break;
246             case Boolean:
247               fprintf(listing, "Type: bool\n");
248               break;
249             case String:
250               fprintf(listing, "Type: string\n");
251               break;
252             case Double:
253               fprintf(listing, "Type: double\n");
254               break;
255             case Float:
256               fprintf(listing, "Type: float\n");
257               break;
258           }
259           break;
```

图 37 在 util.c 的打印函数中添加对应的内容

⑤、varlist 解析类型说明符。

varlist -> id {, id}

该产生式示一个变量列表，可能存在一个或多个变量，每个变量之间用逗号隔开，直到遇到分号为止。依照 stmt_sequence 函数结构，可以得到对应的 varlist 函数，如下图所示。

函数首先创建一个新的标识符节点，并将其名称设置为当前令牌的字符串。然后，进入一个循环，不断解析并创建新的标识符节点，直到遇到分号为止。**在每次循环中，函数都会创建一个新的标识符节点，并将其添加到前一个节点的子节点中**。最后，函数返回解析出的变量列表的根节点。

```
256 TreeNode* varlist(void){
257   TreeNode *t = newExpNode(IdK);
258   if(t!=NULL && token==ID) t->attr.name = copyString(tokenString);
259   match(ID);
260   TreeNode *p = t;
261   while((token!=SEMI)){
262     TreeNode *q;
263     match(COMMA);
264     q = newExpNode(IdK);
265     if(q!=NULL && token==ID) q->attr.name = copyString(tokenString);
266     match(ID);
267     if(q != NULL){
268       if(t==NULL) t = p = q;
269       else{
270         p->child[0] = q;
271         p = q;
272       }
273     }
274   }
275   return t;
276 }
```

图 38 varlist 解析类型说明符

⑥、decl 解析一个声明

decl -> type-specifier varlist

声明由一个类型说明符和变量列表组成。

首先创建一个 decl 的节点，在 globals.h 的节点类型中添加 DeclK 类型，表示声明，如下图所示：

```
50  // 新增
51  typedef enum {StmtK,ExpK,DeclK,ProK} NodeKind;//节点类型
52  typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK,DoK,WhileK} StmtKind;//语句节点
53  typedef enum {OpK,ConstK,IdK,SpecifierK} ExpKind;//表达式节点
54
55  // 新增
56  /* ExpType is used for type checking */
57  typedef enum {Void,Integer,Boolean,Double,Float,String} ExpType; //表达式类型
```

图 39 在 globals.h 的节点类型中添加 DeclK 类型

接着在 util.c 中添加创建 DeclK 节点的方法，依据 StmtK 类型和 ExpK 类型的创建方法即可，代码如下图所示。函数首先为新的 TreeNode 结构体分配内存，然后检查是否成功。如果内存分配失败，函数会打印一条错误信息。如果成功，函数会初始化新节点的所有子节点为 NULL，设置其兄弟节点为 NULL，节点类型为 DeclK，并记录其在源代码中的行号。最后，函数返回新创建的节点。

```
123  // 新增
124  /* Function newDeclNode creates a new declaration
125   * node for syntax tree construction
126   */
127  TreeNode* newDeclNode(){
128    TreeNode *t = (TreeNode *)malloc(sizeof(TreeNode));
129    int i;
130    if(t == NULL){
131      fprintf(listing, "Out of memory error at line %d\n", lineno);
132    }else{
133      for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
134      t->sibling = NULL;
135      t->nodekind = DeclK;
136      t->lineno = lineno;
137    }
138    return t;
139  }
```

图 40 在 util.c 中添加创建 DeclK 节点的方法

最后在 parse.c 中添加 decl 方法。首先调用 type_specifier 函数解析类型说明符，并创建一个新的节点 t。如果 t 不为 NULL，函数会调用 varlist 函数解析变量列表，并将结果存储在 t 的第一个子节点中。最后，函数返回 t，即解析出的声明。

```
277  // 新增，解析声明
278  TreeNode* decl(void){
279    TreeNode *t = type_specifier();
280    if (t!=NULL) t->child[0] = varlist();
281    return t;
282  }
```

图 41 decl 函数

⑦、declarations 解析多个声明或空声明

declarations -> decl ; declarations | ε

该产生式表示声明语句，可以为空，也可以是一个或多个声明，不同的声明语句之间用分号隔开，直到遇到文件末尾或者遇到非类型说明符为止。

根据以上信息，编写 declarations 函数，如下图所示。首先创建一个新的声明节点 t，

然后进入一个循环，**不断解析新的声明，直到遇到文件结束符或者不是类型说明符的令牌为止**。在每次循环中，函数都会创建一个新的声明节点，**并将其添加到前一个节点的兄弟节点中**。如果没有成功解析出任何声明，函数会返回 NULL，否则返回解析出的声明列表的根节点。

```
283  TreeNode* declarations(void){
284    TreeNode *t = newDeclNode();
285    TreeNode *p = t;
286    int notNull = FALSE;
287    while((token!=ENDFILE)&&(token == INT || token == BOOL || token == STRING ||
288                token == FLOAT || token == DOUBLE)){
289      notNull = TRUE;
290      TreeNode *q = decl();
291      match(SEMI);
292      if(q!=NULL){
293        if(t==NULL) t = p = q;
294        else{
295          p->sibling = q;
296          p = q;
297        }
298      }
299    }
300    return notNull ? t : NULL;
301  }
```

图 42 declarations 解析多个声明或空声明

⑧、修改 parse 函数

program -> declarations stmt-sequence

该产生式，由两个字节点构成，声明序列和语句序列，对应 parse 函数。TINY 语法分析器中，program 只对应一个节点，因此 TINY+需要新建一个节点。

首先在 globals.h 中添加 ProK 节点类型，如下图所示。

```
50  // 新增
51  typedef enum {StmtK,ExpK,DeclK,ProK} NodeKind;//节点类型
52  typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK,DoK,WhileK} StmtKind;//语句节点
53  typedef enum {OpK,ConstK,IdK,SpecifierK} ExpKind;//表达式节点
```

图 43 在 globals.h 中添加 ProK 节点类型

接着在 util.c 中添加创建 ProK 节点的方法，如下图所示：

```
141  TreeNode* newProNode(){
142    TreeNode *t = (TreeNode *)malloc(sizeof(TreeNode));
143    int i;
144    if(t == NULL){
145      fprintf(listing, "Out of memory error at line %d\n", lineno);
146    }else{
147      for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
148      t->sibling = NULL;
149      t->nodekind = ProK;
150      t->lineno = lineno;
151    }
152    return t;
153  }
```

图 44 在 util.c 中添加创建 ProK 节点的方法

修改 util.c 的打印函数，添加 ProK 节点打印项并处理 Declk 节点，在遇到 Declk 节点时不打印空格，而其他节点需要打印，使得最终打印结果符合实验指导要求。

```
267     // 新增
268     else if(tree->nodekind == ProK){
269       printSpaces();
270       fprintf(listing, "Program:\n");
271     }
272     else if(tree->nodekind == DeclK){
273      //取消回退缩进
274
275     }
```

图 45 修改 util.c 的打印函数，添加 ProK 节点打印项并处理 Declk 节点

最后修改 parse 函数，首先创建一个新的程序节点 t，然后获取第一个令牌。接着，调用 declarations 函数解析所有的声明，并将结果存储在 t 的第一个子节点中，然后调用 stmt_sequence 函数解析所有的语句序列，并将结果存储在 t 的第二个子节点中。如果在解析完所有的声明和语句序列后，令牌不是文件结束符，会打印一个语法错误。最后，函数返回 t，即解析出的程序节点。

```
310   TreeNode * parse(void)
311   {
312     TreeNode *t = newProNode();
313     token = getToken();
314     t->child[0] = declarations();
315     t->child[1] = stmt_sequence();
316     if (token!=ENDFILE)
317       syntaxError("Code ends before file\n");
318     return t;
319   }
```

图 46 修改 parse 函数

⑨、添加函数声明

为之前添加的新函数添加函数声明。

Parse.c 中添加如下声明：

```
27    // 新增
28    static TreeNode * while_stmt(void);
29    static TreeNode * type_specifier(void);
30    static TreeNode * varlist(void);
31    static TreeNode * decl(void);
32    static TreeNode * declarations(void);
```

图 47 Parse.c 中添加函数声明

Util.h 中添加如下声明：

```
26  // 新增
27  /* Function newDeclNode creates a new declaration
28   * node for syntax tree construction
29   */
30  TreeNode * newDeclNode();
31  // 新增
32  /* Function newProNode creates a new program
33   * node for syntax tree construction
34   */
35  TreeNode * newProNode();
```

图 48 Util.h 中添加函数声明

2.3、调试并完善 TINY+语法分析器，完成对测试文件的测试

用 tiny+.txt 文件测试程序。tiny+.txt 文件内容如下图所示。

```
1 {this is an example}
2 int A,B;
3 bool C;
4 string D;
5 D:= 'scanner';
6 C:=A + B;
7 do
8 A:=A*2
9 while A<=D
```

图 49 tiny+.txt 文件

①、补充对 string 字符串赋值

运行程序，首先发现第五行报错，意思是扫描器无法识别 string 字符串，并且这一行是赋值语句。

```
   5: D:= 'scanner';
        5: ID, name= D
        5: :=
        5: STR,name= 'scanner'

>>> Syntax error at line 5: unexpected token -> STR,name= 'scanner'
```

图 50 词法分析报错 1：无法识别 string 字符串

找到 assign_stmt 函数，发现 match 赋值符号后，在 factor 函数中匹配 string，但是之前并没有在 factor 函数中添加 string 字符串相关的处理。因此添加处理 STR 的情况，如下图所示，依据赋值数字的代码编写，其中都需要创建 ConstK 节点，这里将节点的类型设置为 String，表示赋值类型为字符串。

```
209     // 新增，匹配string
210     case STR:
211       t = newExpNode(ConstK);
212       if(t!=NULL && token==STR) {
213         t->attr.name = copyString(tokenString);
214         t->type = String;
215       }
216       match(STR);
217       break;
```

图 51 添加处理 STR 的情况

在 util.c 的打印函数中添加字符串赋值的处理，如下图所示。修改 ConstK 的情况，当 type 为 string 时输出字符串赋值信息，否则输出整数赋值信息。

```
232     case ConstK:
233       if(tree->type == String)
234         fprintf(listing,"Const: string: %s\n",tree->attr.name);
235       else
236         fprintf(listing,"Const: Integer: %d\n",tree->attr.val);
237       break;
```

图 52 在 util.c 的打印函数中添加字符串赋值的处理

再次运行程序原来的报错解决。

②、while 语句的处理

运行程序发现，第九行出现报错，出现了不希望出现的令牌“while”。

因为根据之前的程序，在 stmt_sequence 函数中，语句以分号分割，但是观察 do-while 语句，发现 while 之前出现的前一条语句末尾是不加分号的。如下图所示：

```
7 do
8 A:=A*2
9 while A<=D
```

图 53 do-while 语句

因此需要修改 stmt_sequence 函数，添加对 while 语句的判断，使得 while 关键字的前一条语句末尾不带分号，如下图所示：

```
50  // 语句序列，即多个语句的集合
51  TreeNode * stmt_sequence(void)
52  { TreeNode * t = statement();
53    TreeNode * p = t;
54    // 当token不是ENDFILE、END、ELSE、UNTIL、WHILE时，继续解析语句，因为这些语句不用；分割
55    while ((token!=ENDFILE) && (token!=END) &&
56           (token!=ELSE) && (token!=UNTIL)&&(token!=WHILE))
57    { TreeNode * q;
58    // 匹配一个分号（SEMI），Tiny+语言中语句之间的分隔
59      match(SEMI);
60      q = statement();
61      if (q!=NULL) {
62        // 如果t为NULL，说明这是第一个成功解析的语句，函数会将t和p都设置为q
63        if (t==NULL) t = p = q;
64        else /* now p cannot be NULL either */
65        { p->sibling = q;
66          p = q;
67        }
68      }
69    }
70    return t;
71  }
```

添加

图 54 修改 stmt_sequence 函数，添加对 while 语句的判断

修改后运行程序，不会出现原来的报错。

③、表达式的处理

继续运行程序，出现如下报错。表示<=符号不能识别。

```
   9: while A<=D
        9: reserved word: while
        9: ID, name= A
        9: <=

>>> Syntax error at line 9: unexpected token -> <=

>>> Syntax error at line 9: unexpected token -> <=
        9: ID, name= D

>>> Syntax error at line 9: unexpected token -> ID, name= D
        10: EOF
```

图 55 词法错误 3：表达式符号无法识别

表达式由函数 exp 处理，查看 exp 函数，发现当前只支持 < 和 = 两个符号，所以我们需要加上其他运算符，如下图所示：

```
// 解析表达式
TreeNode * exp(void)
{ TreeNode * t = simple_exp();
  // 如果是比较运算符
  if ((token==LT)||(token==EQ)||(token==GT)||(token==LTE)||(token==GTE)) {
    // 操作符
    TreeNode * p = newExpNode(OpK);
    if (p!=NULL) {
      p->child[0] = t;
      p->attr.op = token;
      t = p;
    }
    match(token);
    if (t!=NULL)
      t->child[1] = simple_exp();
  }
  return t;
}
```

添加其他符号

图 56 添加其他表达式运算符

再次运行程序，成功解决所有词法分析报错。

④、tiny+.txt 文件测试结果

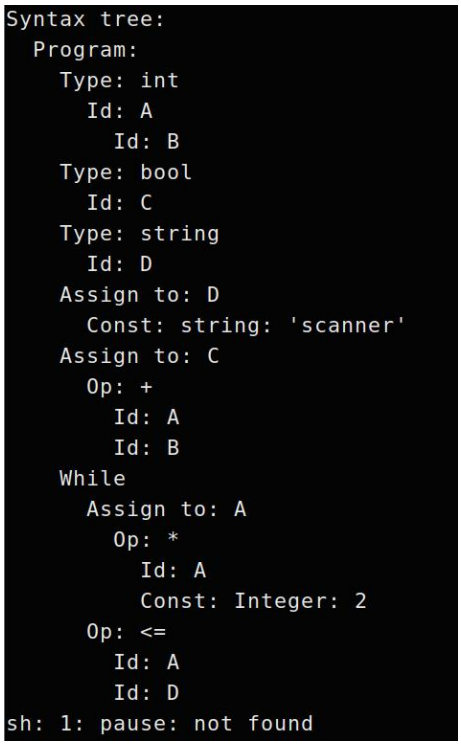
最终测试结果如下图所示，成功进行词法分析和打印出语法树。

```
TINY COMPILATION: tiny+.txt
   1: {this is an example}
   2: int A,B;
        2: reserved word: int
        2: ID, name= A
        2: ,
        2: ID, name= B
        2: ;
   3: bool C;
        3: reserved word: bool
        3: ID, name= C
        3: ;
   4: string D;
        4: reserved word: string
        4: ID, name= D
        4: ;
   5: D:= 'scanner';
        5: ID, name= D
        5: :=
        5: STR,name= 'scanner'
        5: ;
```

```
   6: C:=A + B;
        6: ID, name= C
        6: :=
        6: ID, name= A
        6: +
        6: ID, name= B
        6: ;
   7: do
        7: reserved word: do
   8: A:=A*2
        8: ID, name= A
        8: :=
        8: ID, name= A
        8: *
        8: NUM, val= 2
   9: while A<=D
        9: reserved word: while
        9: ID, name= A
        9: <=
        9: ID, name= D
        10: EOF
```

图 57 tiny+.txt 文件的词法分析结果

下图为我编写的 TINY+语法分析器结果（左）和实验指导中的结果示例（右），我编写的程序运行结果与实验指导的要求一致。成功完成实验要求。



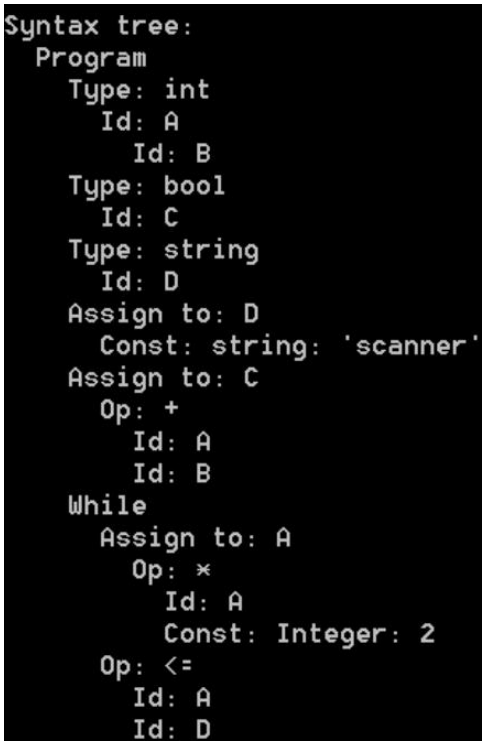
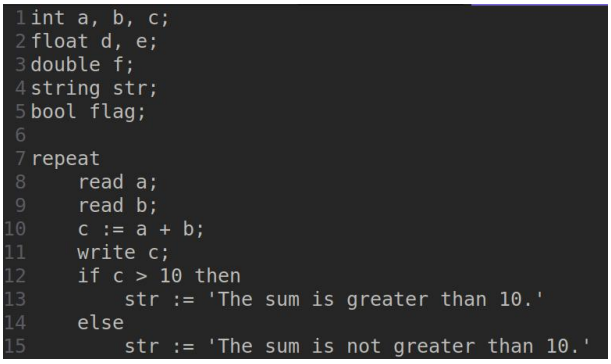


图 58 TINY+语法树结构

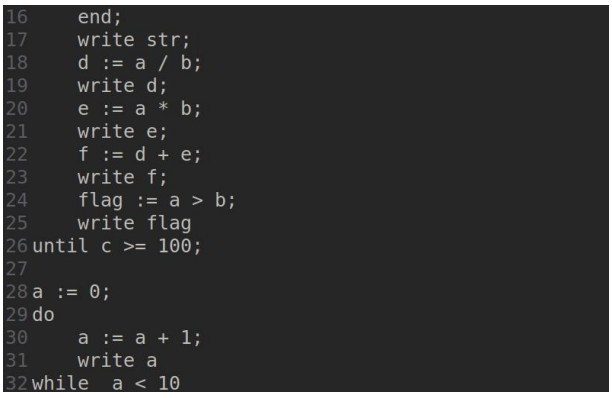
2.4、编写一个 TINY+文件进行测试

自行编写一个 TINY+文件再次进行测试。在编写测试文件的过程中，需要注意语句后是否需要添加分号，例如 while 语句的前一句语句就不需要添加分号。我编写的 mytiny+.txt 文件如下图所示：



```
1 int a, b, c;
2 float d, e;
3 double f;
4 string str;
5 bool flag;
6
7 repeat
8     read a;
9     read b;
10    c := a + b;
11    write c;
12    if c > 10 then
13        str := 'The sum is greater than 10.'
14    else
15        str := 'The sum is not greater than 10.'
```

图 59 mytiny+.txt 文件-part1



```
16    end;
17    write str;
18    d := a / b;
19    write d;
20    e := a * b;
21    write e;
22    f := d + e;
23    write f;
24    flag := a > b;
25    write flag
26 until c >= 100;
27
28 a := 0;
29 do
30    a := a + 1;
31    write a
32 while  a < 10
```

图 60 mytiny+.txt 文件-part2

mytiny+可以正确进行词法分析，没有报错。对应的语法树结构如下：

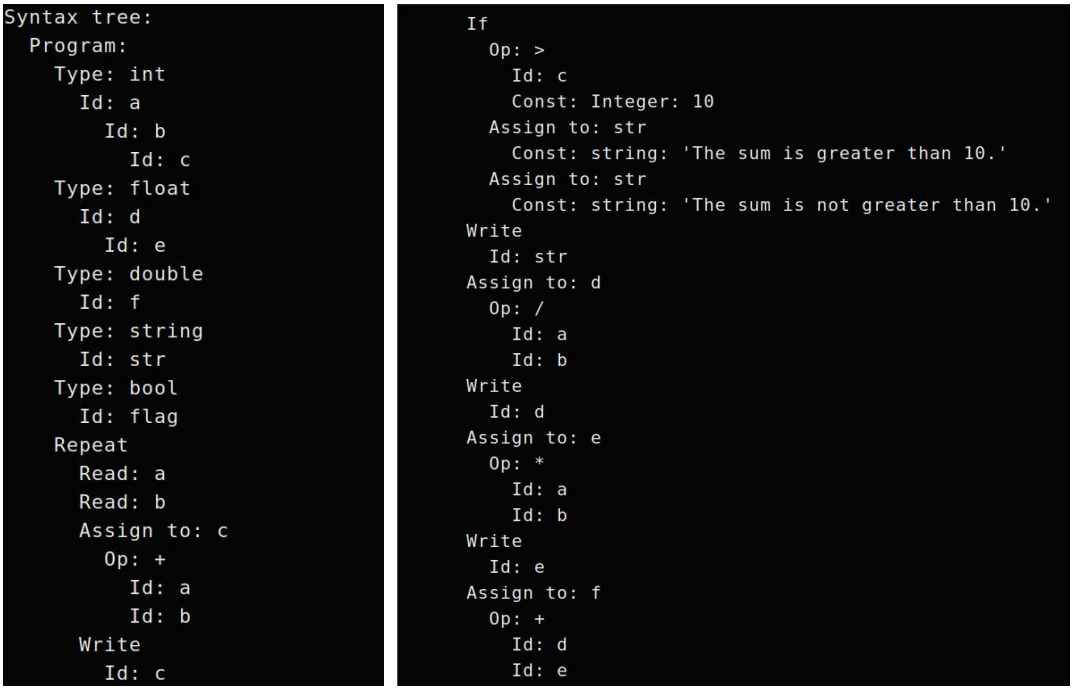


图 61 mytiny+语法树-part1

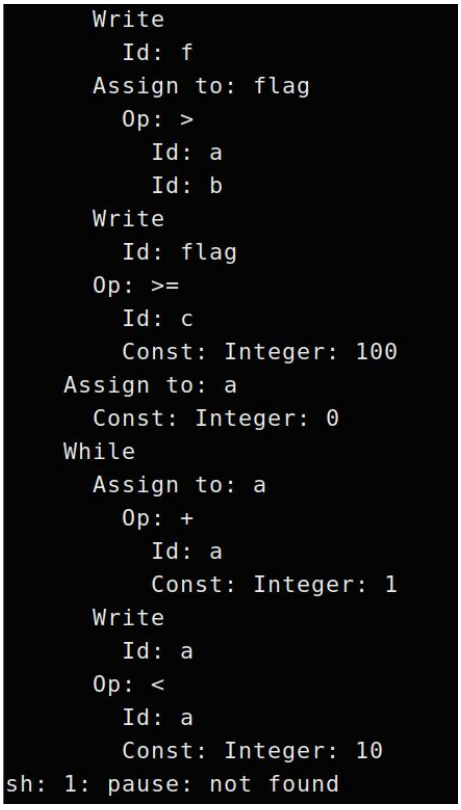


图 62 mytiny+语法树-part2

以上，成功用拓展的 TINY+语法分析器实现对 TINY+文件的词法分析和语法分析。

实验结论：

在任务一中，我成功运行了 TINY 语言的语法分析程序 TINYParser，并理解了其实现过程。通过编写并运行一个新的 TINY 测试程序，观察其解析过程和生成的语法树，我对 TINY 语言的语法规则和解析器的工作原理有了更深入的理解。

在任务二中，我修改了 TINY 语法分析器，以支持扩展的 TINY+ 语言语法。主要添加了声明语句和 while 语句的解析功能。根据 TINY+ 语言的语法定义，编写并运行了新的 TINY+ 测试程序，验证了修改后的语法分析器的正确性。

通过完成任务一和任务二，我成功运行并理解了 TINY 语言的语法分析程序，并在其基础上实现了对 TINY+ 语言的语法分析器的扩展。编写并运行了新的 TINY 和 TINY+ 测试程序，生成了正确的语法树，验证了语法分析器的正确性。本次实验帮助我加深了对语法分析器设计与实现的理解，提高了编程能力和解决实际问题的能力。