

# 编译原理课程设计

学 院（系）：电子信息与电气工程学部

学 生 姓 名：赵裕

学 号：201445004

班 级：电计 1403

同 组 人：无

大连理工大学

Dalian University of Technology

## 目 录

1	程序简介与设计.....	1
1.1	程序架构.....	1
1.2	代码结构.....	2
2	词法分析程序的实现.....	3
2.1	标识符.....	3
2.2	单词的识别模型-有穷状态自动机 DFA.....	3
2.3	词法分析实现.....	4
2.4	词法分析测试.....	5
3	递归下降法实现语法分析程序.....	7
3.1	语法定义.....	7
3.2	程序实现.....	13
3.3	语法分析测试.....	14
4	中间代码与解释程序.....	16
4.1	中间代码.....	16
4.1.1	表达式的中间代码.....	16
4.1.2	条件语句的中间代码.....	17
4.1.3	循环语句的中间代码.....	17
4.1.4	选择语句的中间代码.....	18
4.2	符号表.....	18
4.3	程序实现.....	20
4.4	测试用例.....	21
4.4.1	表达式.....	21
4.4.2	条件语句.....	22
4.4.3	循环语句.....	23
4.4.4	分支语句.....	24
4.4.5	函数调用.....	25

4.4.6	数据类型.....	27
5	错误处理.....	29
5.1	词法解析错误.....	29
5.1.1	错误用例 1.....	30
5.1.2	错误用例 2.....	30
5.2	语法解析错误.....	31
5.2.1	表达式缺少成分.....	31
5.2.2	类型不匹配.....	32
5.3	运行时错误.....	32
5.4	错误处理的实现.....	33
5.4.1	错误定位的实现.....	33
5.4.2	类型检查的实现.....	33
5.4.3	错误的恢复.....	35
5.4.4	错误类型.....	36
6	调试器.....	37
7	图形界面.....	40
8.1	课设总结.....	41
8.2	收获.....	41
9	附录.....	42

# 1 程序简介与设计

为描述方便，以下简称该语言为 ToyPascal。

## 1.1 程序架构

ToyPascal 包括词法分析，语法、语义分析，中间代码生成，解释执行，调试功能以及图形界面。

编译程序主体如下图

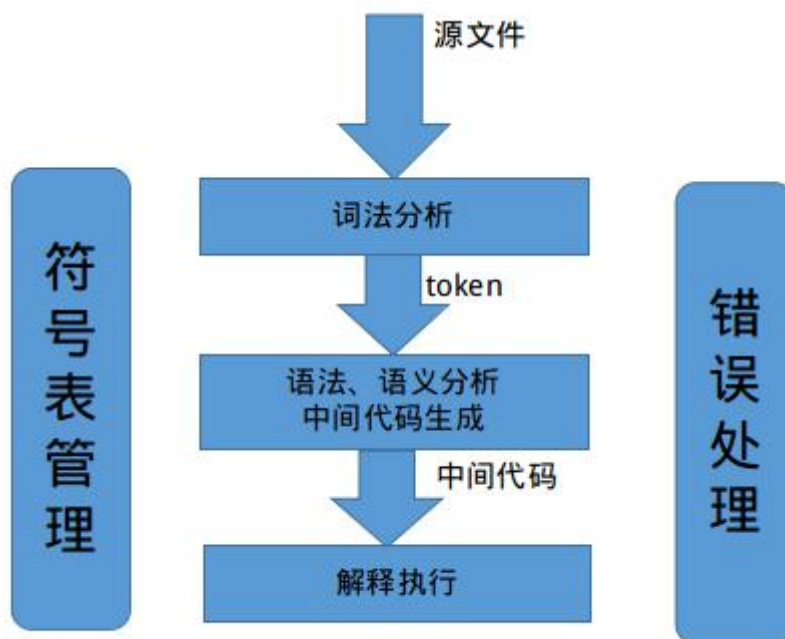


图 1.1 程序架构

调试程序的实现较为复杂，后文做介绍。GUI 采用 **C/S** 模式，图形界面作为发送消息的外壳：

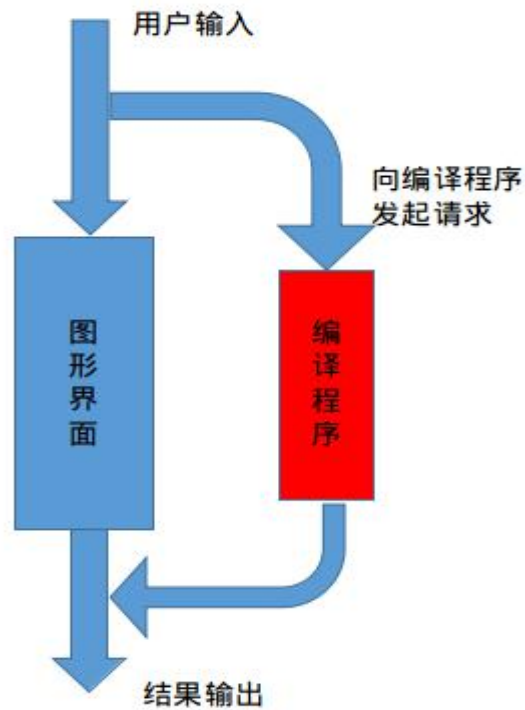
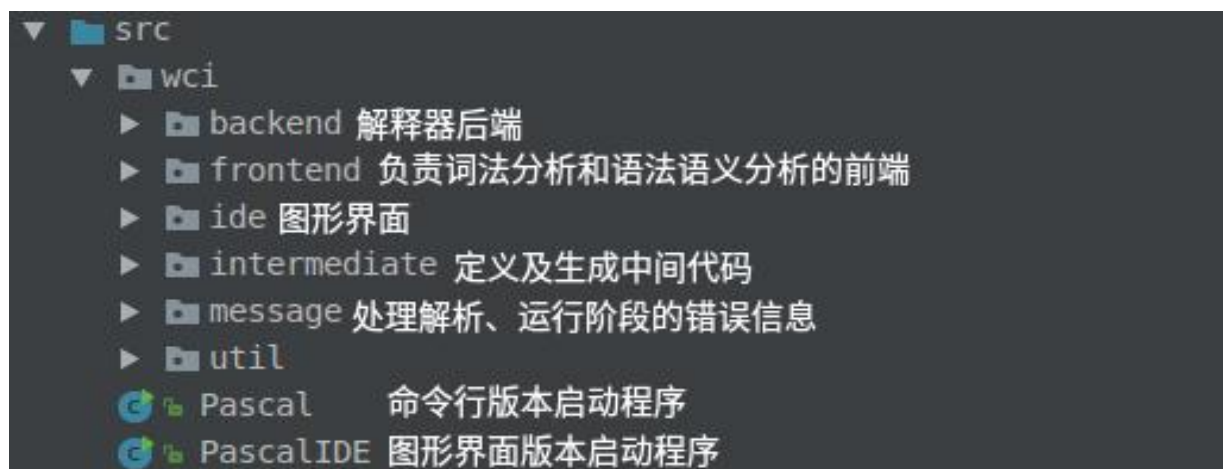


图 1.2 图形界面设计

## 1.2 代码结构



1.3 代码结构

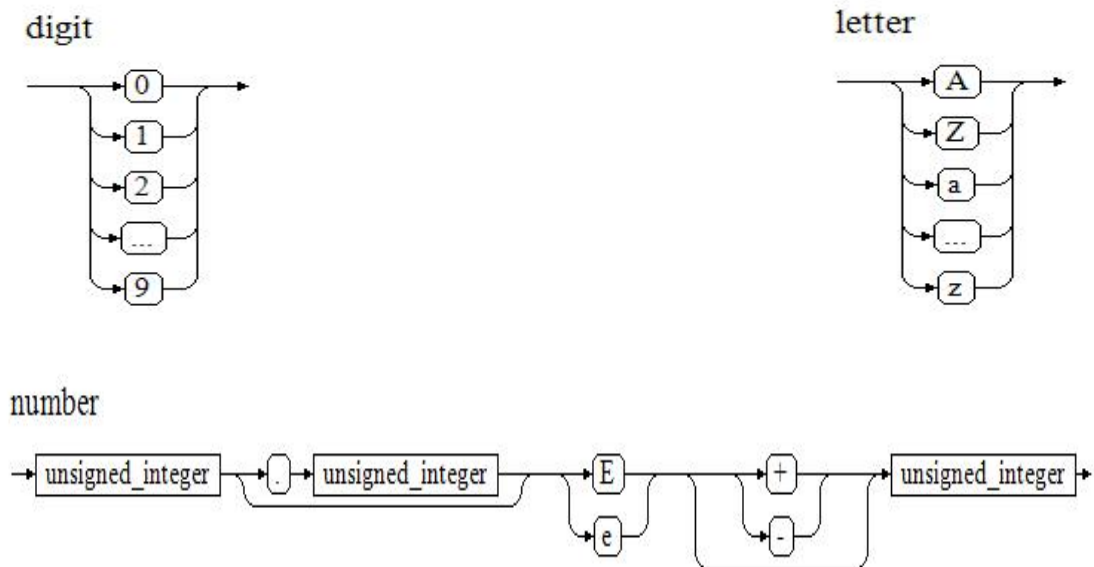
## 2 词法分析程序的实现

### 2.1 标识符

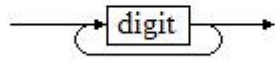
主要包括**保留字**和**运算符**，关于字符串、标志符见 1.1 文法定义。

关键字	AND、ARRAY、BEGIN、CASE、CONST、DIV、DO、DOWNT ELSE、END、FOR、FUNCTION、IF、MOD、NIL、NOT、 OF、OR、PROCEDURE、PROGRAM、RECORD、REPEAT、 SET、THEN、TO、TYPE、UNTIL、VAR、WHILE、WITH
运算符	+, -, *, /, MOD, DIV, ..., ., >=, <>, <=, >, <, =, :=, ,, AND, OR, NOT, :

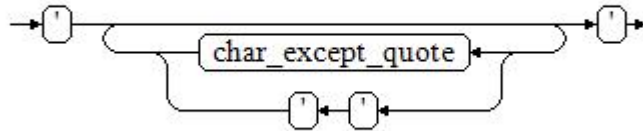
### 2.2 单词的识别模型-有穷状态自动机 DFA



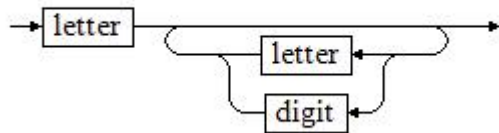
unsigned\_integer



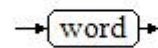
string



word



identifier



## 2.3 词法分析实现

`front.pascal.token` 包下定义了各种标志符：



图 2.1 词法分析程序

## 2.4 词法分析测试

词法分析程序在包 `wci.parser` 下。

在 GUI 中，通过复用词法分析部分代码，可以直观看到词法分析的效果，如下图，通过词法分析识别不同词法成分，并设置不同高亮：

```

1  {ToyPascal支持的基本数据类型}
2  PROGRAM BasicType;
3
4  {常量的类型由右值决定}
5  CONST
6      WEEKDAY = 7; {整型常量}
7      PI = 3.1415926; {浮点常量}
8      X = 'x'; {字符常量}
9      ERROR = 'There is an error!'; {字符串常量}
10     YES = true; {布尔常量}
11 {TYPE可以命名新的常量,类似C语言的typedef}
12 TYPE
13     string = char;
14     bool = boolean;
15 {必须在变量定义后指定类型}
16 VAR
17     message : char;
```

图 2.2 图形界面的代码高亮

以以下代码为例：

```

1. PROGRAM test;
2. VAR
3.     sum, n:integer;
4.
5. PROCEDURE recurSum(VAR sum, n: integer); forward;{前置声明}
6.
7. PROCEDURE recurSum;
8.     BEGIN
9.         IF n < 0 THEN BEGIN
10.             sum := sum + n;
11.             n := n - 1;
12.             recurSum(sum, n);{递归调用}
13.         END;
14.     END;
```



```
15. BEGIN{主程序}
16.  sum := 0;
17.  n := 100;
18.  recurSum(sum, n);
19.  writeln('1+2+...+100 = ', sum);
20. END.
```

词法分析结果为：

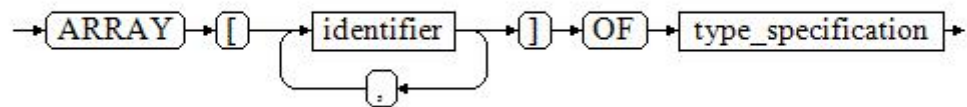
```
PROGRAM test ; VAR sum , n : integer ; PROCEDURE
recurSum ( VAR sum , n : integer ) ; forward ;
PROCEDURE recurSum ; BEGIN IF n > 0 THEN BEGIN sum
:= sum + n ; n := n - 1 ; recurSum ( sum , n )
; END ; END ; BEGIN sum := 0 ; n := 100 ;
recurSum ( sum , n ) ; writeln ( '1+2+...+100 = ',
sum ) ; END .
```

### 3 递归下降法实现语法分析程序

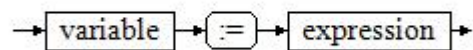
#### 3.1 语法定义

见下图：

array\_type



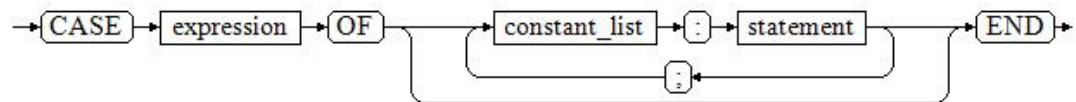
assignment\_statement



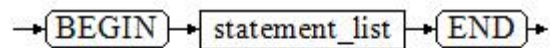
block



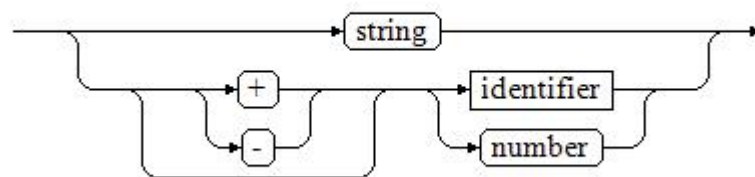
CASE\_statement



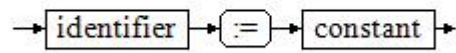
compound\_statement



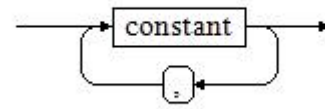
constant



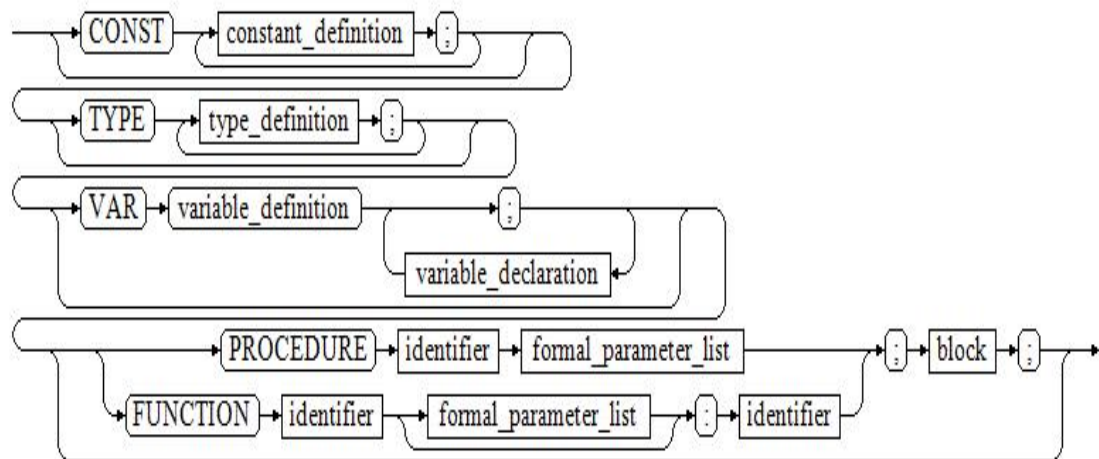
### constant\_definition



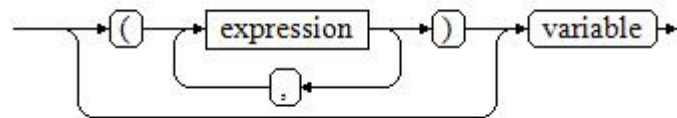
### constant\_list



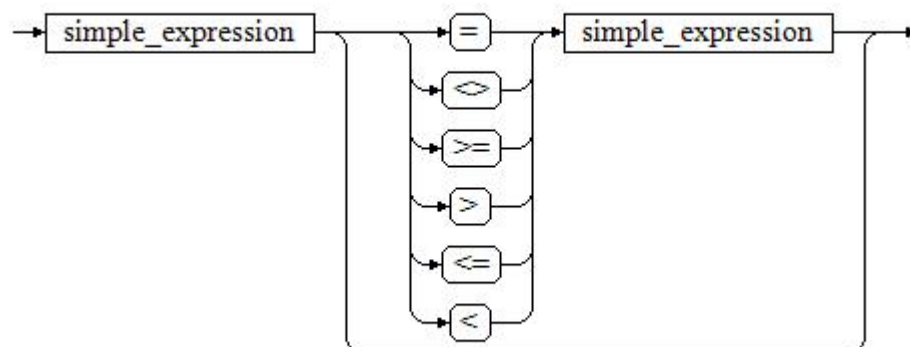
### declarations



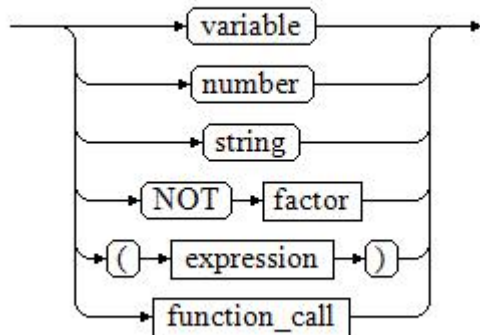
### declared\_routine\_call



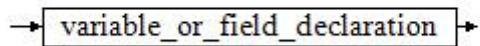
### expression



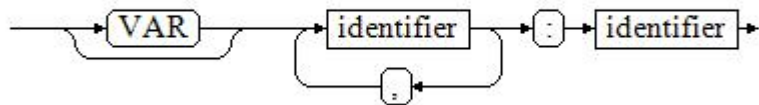
factor



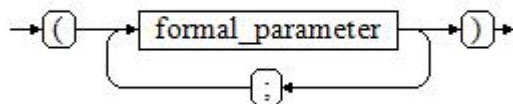
field\_declaration



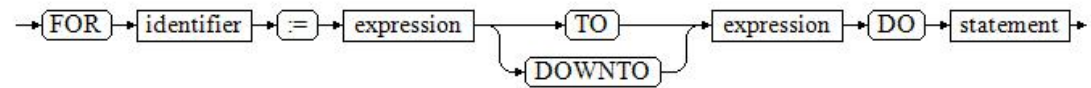
formal\_parameter



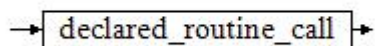
formal\_parameter\_list



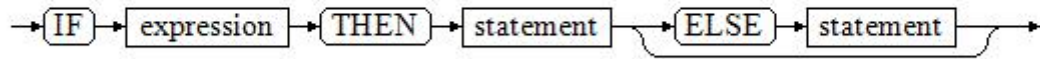
FOR\_statement



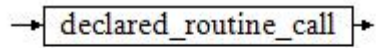
function\_call



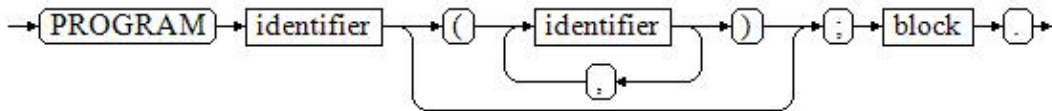
IF\_statement



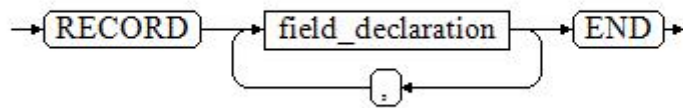
procedure\_call



program



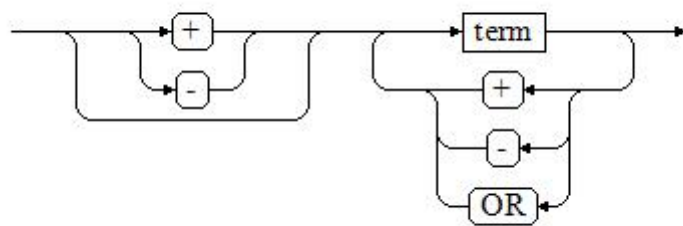
record\_type



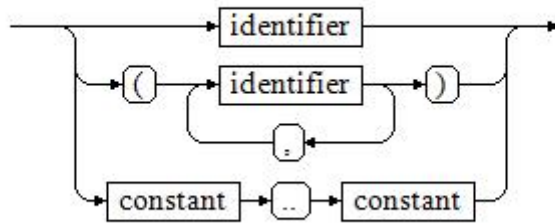
REPEAT\_statement



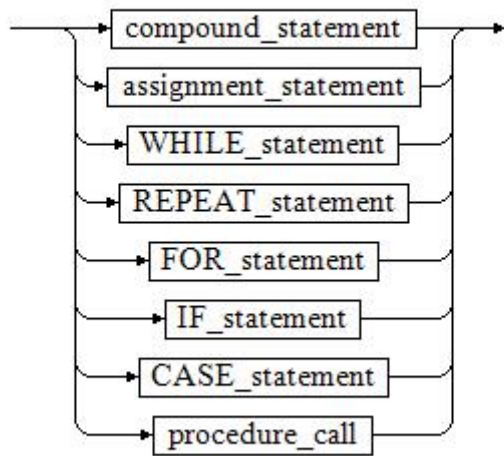
simple\_expression



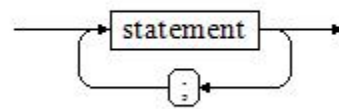
simple\_type



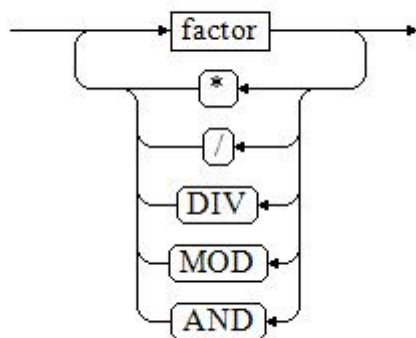
statement



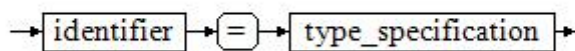
statement\_list



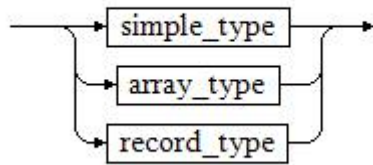
term



type\_definition



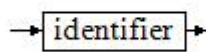
type\_specification



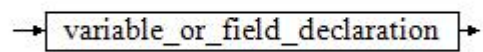
unsigned\_integer



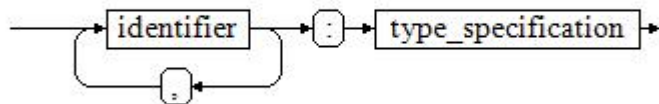
variable



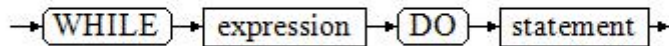
variable\_declaration



variable\_or\_field\_declaration



WHILE\_statement



## 3.2 程序实现

语法分析主要在 `front.pascal.parser` 包下（图中标出了一部分类的功能，其他类根据名字也很容易知道）：



图 3.1 语法、语义分析程序



### 3.3 语法分析测试

代码：

```
1 PROGRAM LoopTest;
2
3 VAR
4     root, number : real;
5     i, sum : integer;
6     n, pi : real;
7
8 BEGIN
9
10    {求阶乘5! : 测试FOR循环}
11    sum := 1;
12    FOR i := 1 TO 5 DO BEGIN
13        sum := sum * i;
14    END;
15    writeln('5! = ', sum); {120}
16
17
18 END.
```

图 3.2 语法分析测试代码

语法分析信息：

运行信息	输出	中间代码
18 source lines, 0 syntax errors, 0.04 seconds total parsing time.		
19 statements executed, 0 runtime errors, 0.00 seconds total execution time.		

图 3.3 语法分析信息

中间代码：



```
*** PROGRAM looptest ***
<COMPOUND line="8">
  <ASSIGN line="11" type_id="integer">
    <VARIABLE id="sum" level="1" type_id="integer" />
    <INTEGER_CONSTANT value="1" type_id="integer" />
  </ASSIGN>
  <COMPOUND line="12">
    <ASSIGN line="12" type_id="integer">
      <VARIABLE id="i" level="1" type_id="integer" />
      <INTEGER_CONSTANT value="1" type_id="integer" />
    </ASSIGN>
    <LOOP>
      <TEST>
        <GT type_id="boolean">
          <VARIABLE id="i" level="1" type_id="integer" />
          <INTEGER_CONSTANT value="5" type_id="integer" />
        </GT>
      </TEST>
      <COMPOUND line="12">
        <ASSIGN line="13" type_id="integer">
          <VARIABLE id="sum" level="1" type_id="integer" />
          <MULTIPLY type_id="integer">
            <VARIABLE id="sum" level="1" type_id="integer" />
          </MULTIPLY>
        </ASSIGN>
      </COMPOUND>
    </LOOP>
  </COMPOUND>
</COMPOUND>
```

图 3.4 语法分析中间代码

错误用例见 5 错误处理

## 4 中间代码与解释程序

### 4.1 中间代码

本程序的中间代码和常规中间代码稍有不同，因为是解释执行，所以没有通过符号表生成三元式而是在执行过程中查找符号表实现解释执行。

#### 4.1.1 表达式的中间代码

中间代码的本质就是一个抽象语法树（AST）

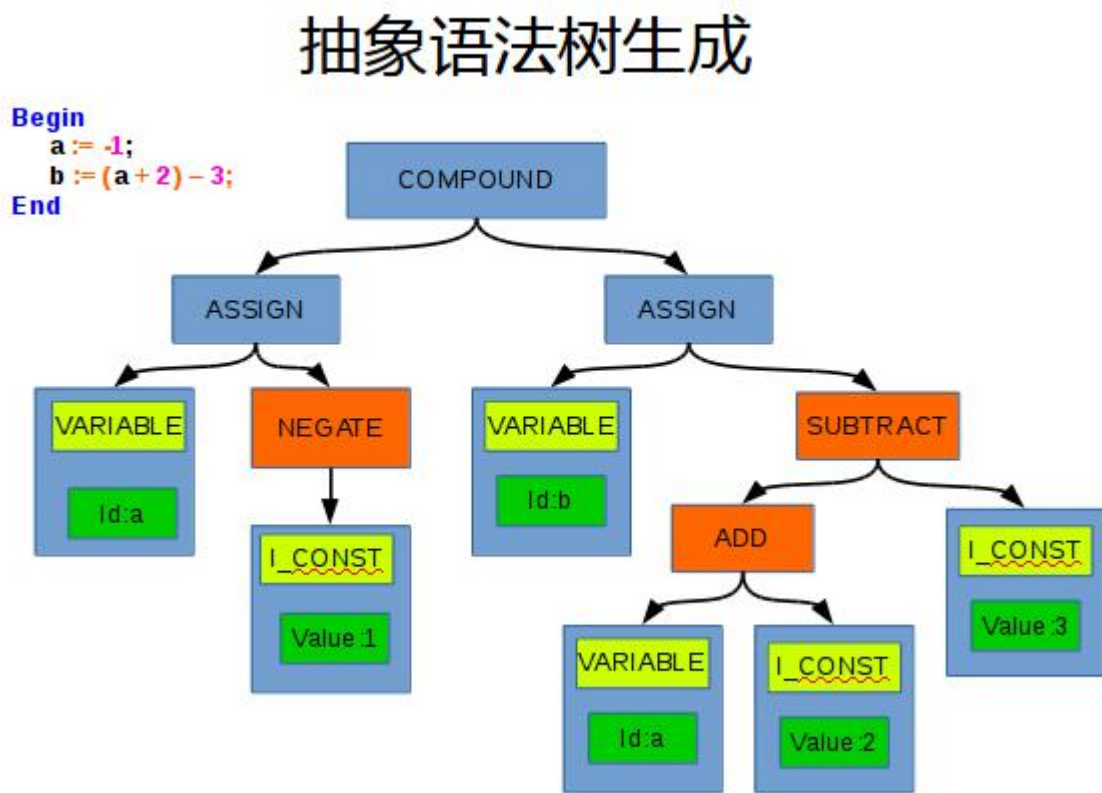


图 4.1 表达式的抽象语法树

## 4.1.2 条件语句的中间代码

## 举例：IF-ELSE 结构

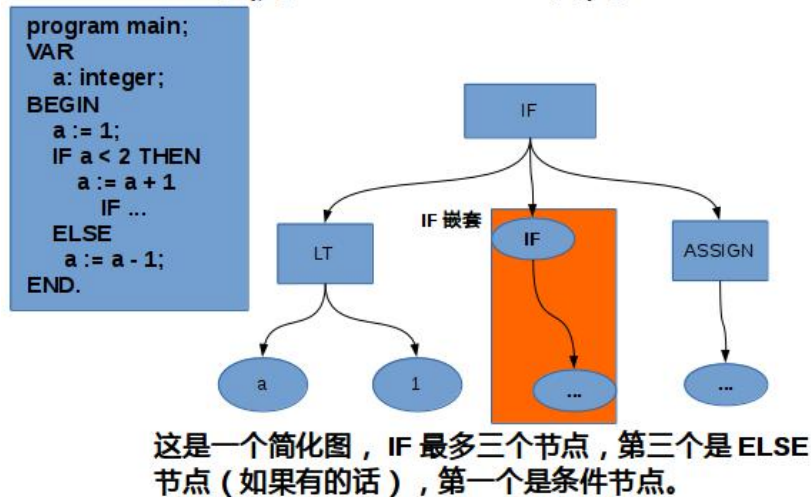


图 4.2 条件语句的抽象语法树

## 4.1.3 循环语句的中间代码

## 举例：循环结构

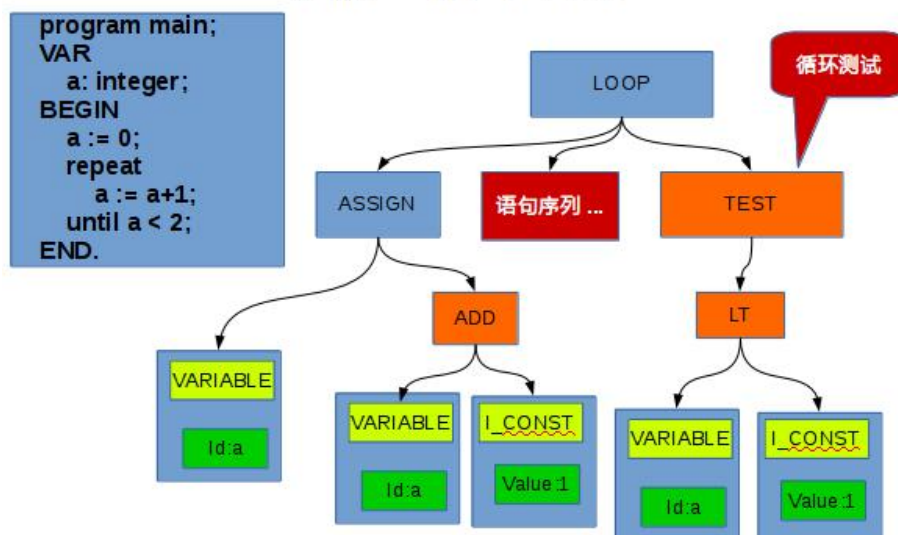


图 4.3 循环结构的抽象语法树

#### 4.1.4 选择语句的中间代码

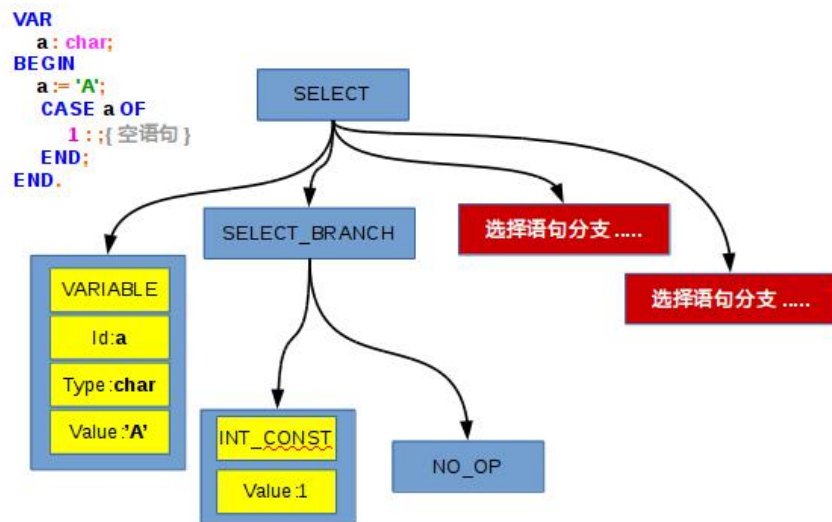


图 4.4 选择语句的抽象语法树

## 4.2 符号表

符号表负责管理类型声明和作用域：

### 符号表管理 (类型声明)

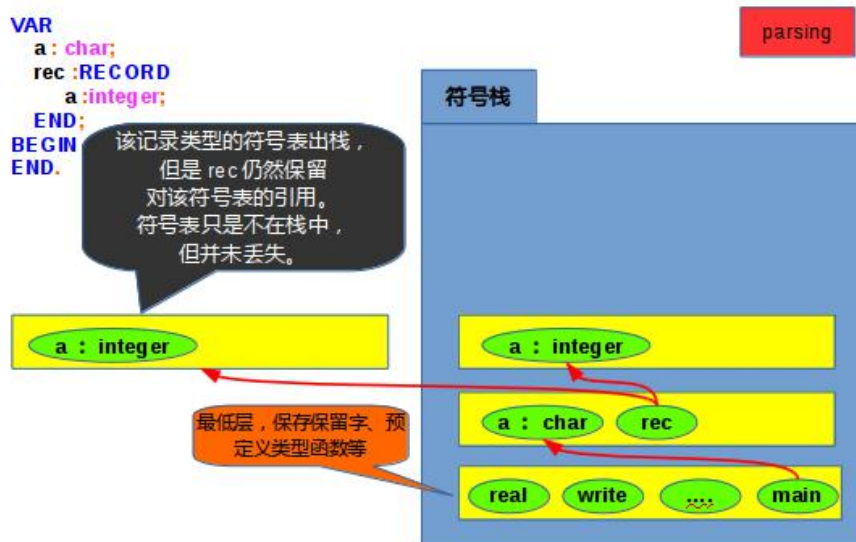


图 4.5 类型声明的符号表

## 符号表管理 (过程 / 函数)

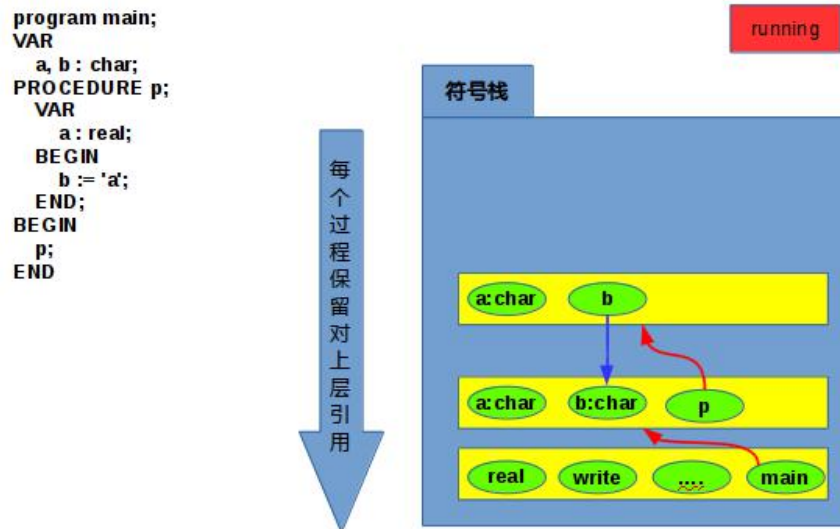


图 4.6 过程调用的符号表

## 作用域

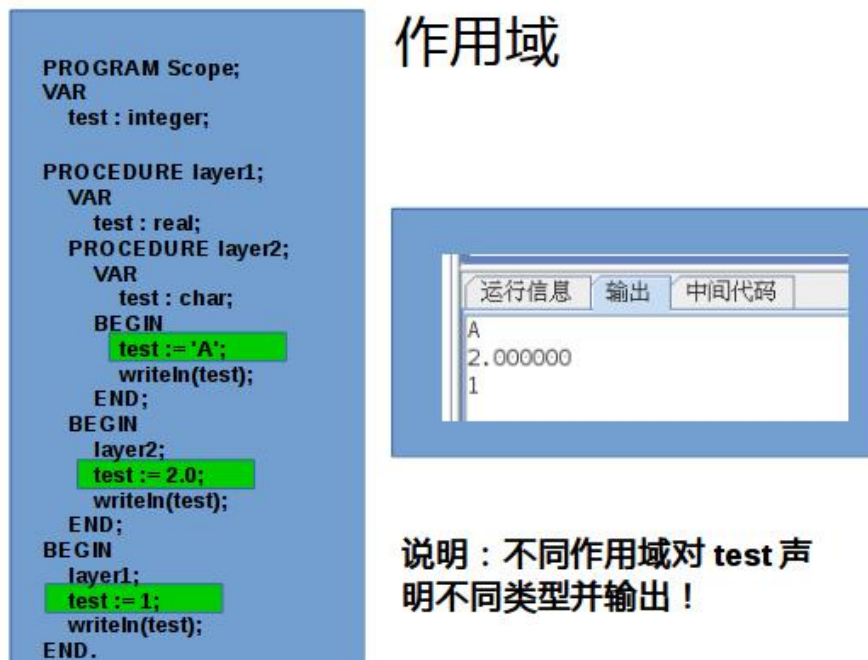


图 4.7 不同符号表实现作用域



### 4.3 程序实现

中间代码：

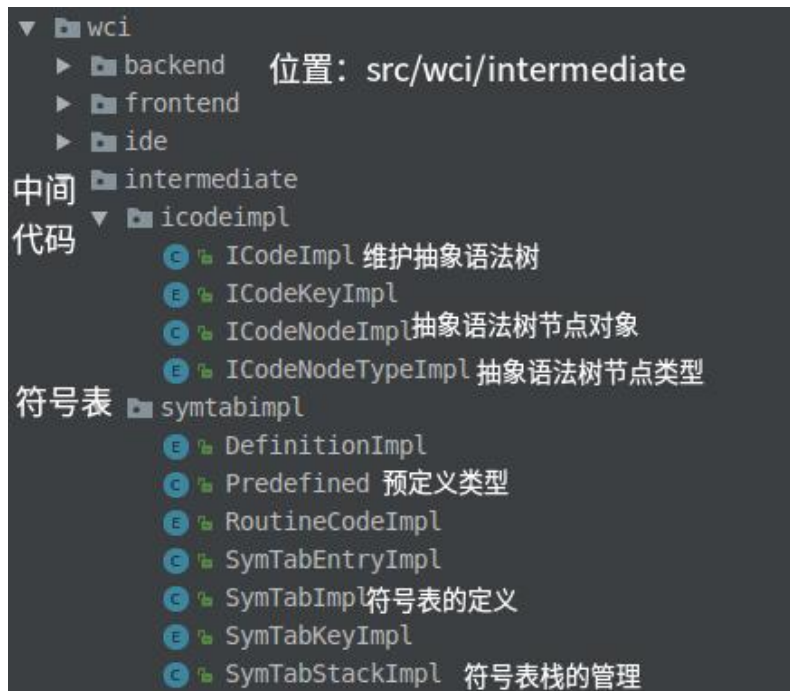


图 4.8 Toy Pascal 的中间代码实现

解释程序：

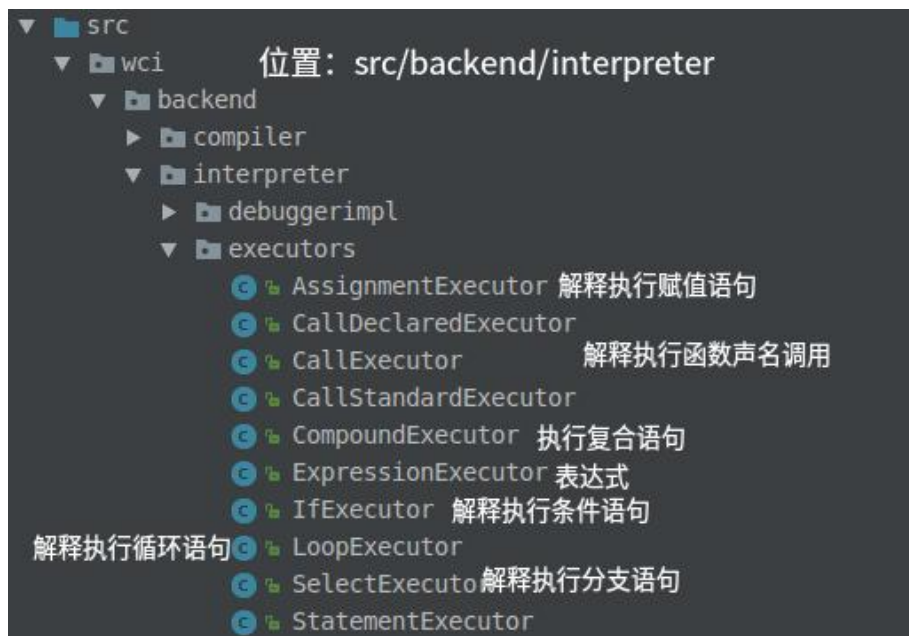


图 4.9 Toy Pascal 的解释执行代码实现

## 4.4 测试用例

### 4.4.1 表达式

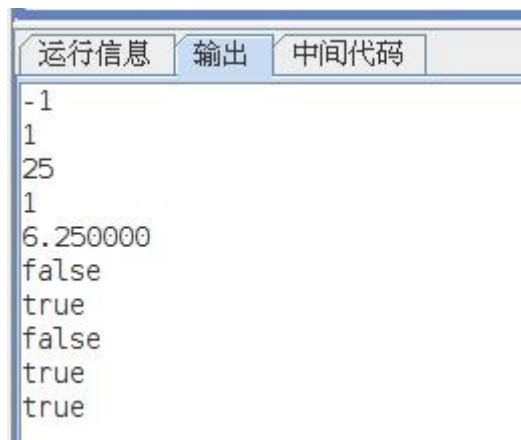
支持**整型**、**浮点型**运算以及**逻辑运算**。代码：

```
1. PROGRAM Expression;
2. VAR
3.   num1, num2, num3 : integer;
4.   num4 : real;
5.   b1, b2, b3 : boolean;
6.
7. BEGIN
8.   num1 := -1;
9.   writeln(num1);
10.
11.  num2 := -num1;
12.  writeln(num2);
13.
14.  num3 := (10 + 10) DIV 4 * 5;
15.  writeln(num3);
16.
17.  writeln(num3 MOD 3);
18.
19.  num4 := (1.4 + 3.6) * 5 / 4;
20.  writeln(num4);
21.
22.  b1 := true;
23.  b2 := true;
24.  b3 := false;
25.
26.  writeln(NOT b1);
```



```
27. writeln(b1 AND b2);
28. writeln(b1 AND b3);
29. writeln(b1 OR b2);
30. writeln(NOT b1 OR (b2 AND NOT b3));
31.
32. END.
```

运行结果：



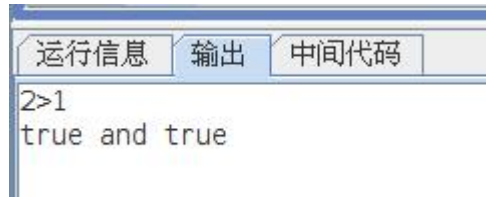
#### 4.4.2 条件语句

支持嵌套 IF 语句。代码：

```
1. PROGRAM IfTest;
2.
3. BEGIN
4.   {简单 IF 语句}
5.   IF 2 < 1 THEN
6.     writeln('2<1')
7.   ELSE
8.     writeln('2<=1');
9.
10.  {IF 嵌套}
11.  IF true THEN
12.    IF NOT false THEN
13.      writeln('true and true')
14.    ELSE
```

```
15.   writeln('true and false')
16.   ELSE
17.   writeln(false);
18. END.
```

结果：



#### 4.4.3 循环语句

支持 FOR、WHILE、REPEAT 三种循环：

```
1. {测试 3 中循环结构}
2. PROGRAM LoopTest;
3.
4. VAR
5.   root, number : real;
6.   i, sum : integer;
7.   n, pi : real;
8.
9. BEGIN
10. {牛顿法求平方根：测试 while 循环}
11.   number := 2;
12.   root := number;
13.   WHILE root*root - number > 0.00001 DO BEGIN
14.     root := (number/root + root)/2;
15.   END;
16.   writeln('root of 2: ', root);
17.
18. {求阶乘 5!：测试 FOR 循环}
19.   sum := 1;
20.   FOR i := 1 TO 5 DO BEGIN
```

```
21.    sum := sum * i;
22.    END;
23.    writeln('5! = ', sum);{120}
24.
25.    {利用公式求圆周率：测试 REPEAT 循环}
26.    n := 1;
27.    pi := 0;
28.    REPEAT
29.        pi := pi + (1/(4*n-3) - 1/(4*n-1));
30.        n := n+1;
31.    UNTIL n = 1000;
32.    pi := 4*pi;
33.    writeln('PI = ', pi);
34.
35. END.
```

结果：



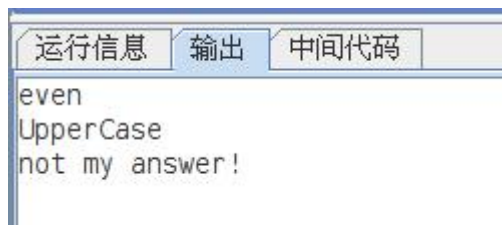
#### 4.4.4 分支语句

CASE 语句能识别单个字符和整型数字和嵌套 CASE 语句：

```
1.    {测试 Case 语句}
2.    PROGRAM CaseTst;
3.    VAR
4.        ch : char;
5.        num : integer;
6.    BEGIN
7.        {测试用例 1}
8.        CASE 10 DIV 4 OF{10/4 的结果是实型}
```

```
9.      1, 3, 5, 7: writeln('odd');
10.     2, 4, 6, 8: writeln('even')
11.  END;
12.  {测试用例 2}
13.  ch := 'B';
14.  CASE ch OF
15.    'A', 'B', 'C': writeln('UpperCase');
16.    'a', 'b', 'c': writeln('LowerCase')
17.  END;
18.
19.  {测试用例 3:嵌套}
20.  num := 3;
21.  CASE num OF
22.    1, 3, 5: writeln('not my answer!');
23.    2, 4, 6:
24.      CASE num OF
25.        2: writeln(2);
26.        4: writeln(4);
27.        6: writeln(6);
28.      END;
29.  END;
30. END.
```

结果：



#### 4.4.5 函数调用

```
1. PROGRAM test;
```

```
2.
3.  VAR
4.    sum, n:integer;
5.
6.  PROCEDURE recurSum(VAR sum, n: integer); forward;{前置声明}
7.
8.  PROCEDURE recurSum;
9.    BEGIN
10.     IF n <= 0 THEN BEGIN
11.       sum := sum + n;
12.       n := n - 1;
13.       recurSum(sum, n);{递归调用}
14.     END;
15.  END;
16.
17.
18. BEGIN{主程序}
19.   sum := 0;
20.   n := 100;
21.   recurSum(sum, n);
22.   writeln('1+2+...+100 = ', sum);
23. END.
```

结果：



#### 4.4.6 数据类型

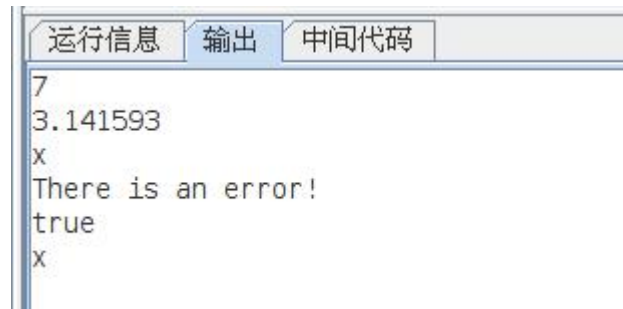
支持基本类型：integer,real,char,boolean 和复杂类型：枚举，子域，记录，数组。

代码 1：

```
1. {ToyPascal 支持的基本数据类型}
2. PROGRAM BasicType;
3.
4. {常量的类型由右值决定}
5. CONST
6.     WEEKDAY = 7;{整型常量}
7.     PI = 3.1415926;{浮点常量}
8.     X = 'x';{字符常量}
9.     ERROR = 'There is an error!';{字符串常量}
10.    YES = true;{布尔常量}
11. {TYPE 可以命名新的常量,类似 C 语言的 typedef}
12. TYPE
13.     string = char;
14.     bool = boolean;
15. {必须在变量定义后指定类型}
16. VAR
17.     message : char;
18.     grade : real;
19.     age : integer;
20.     result : boolean;
21.
22. BEGIN
23.     writeln(WEEKDAY);
24.     writeln(PI);
25.     writeln(X);
26.     writeln(ERROR);
27.     writeln(YES);
```

```
28. message := X;
29. writeln(message);
30. END.
```

结果 1:

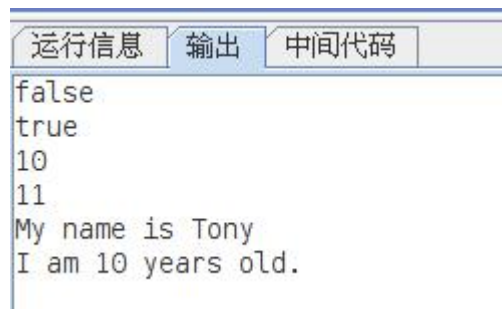


代码 2 (测试数组等结构) :

```
1. {ToyPascal 支持的高级数据类型}
2. PROGRAM ConstructType;
3.
4. VAR
5.   enum : (one, two, three, four);{枚举}
6.   subrange : 1..100;{子域}
7.   arr : ARRAY[1..10] OF integer;{数组}
8.   rec : RECORD{记录类型}
9.     name : ARRAY[1..10] OF char;
10.    age : integer;
11.    isStudent : boolean;
12.  END;
13.
14. BEGIN
15.   enum := two;
16.   writeln(three < enum);{枚举不能直接输出}
17.   enum := four;
18.   writeln(three < enum);
19.
20.   subrange := 10;{注意赋值不能越界}
```

```
21. writeln(subrange);
22.
23. arr[1] := 11;{注意从 1 开始}
24. writeln(arr[1]);{只能操作已经定义的值,输出 arr[2]会导致错误}
25. rec.name := 'Tony';
26. rec.age := 10;
27. writeln('My name is ', rec.name);
28. writeln('I am ', rec.age, ' years old. ');
29. END.
```

结果 2:



## 5 错误处理

由于本程序处理错误的能力较强，故单错误处理独作为一章。本程序的错误处理有两个特点（详细见测试用例）：

1. 错误类型详细;
2. 错误定位准确;

### 5.1 词法解析错误

前面提到，GUI 的语法高亮功能复用了词法分析的代码，所以，即使不编译，也能实时现实词法解析的错误。



### 5.1.1 错误用例 1

如下图，对于 ToyPascal 未定义的 Token 显示为红色：

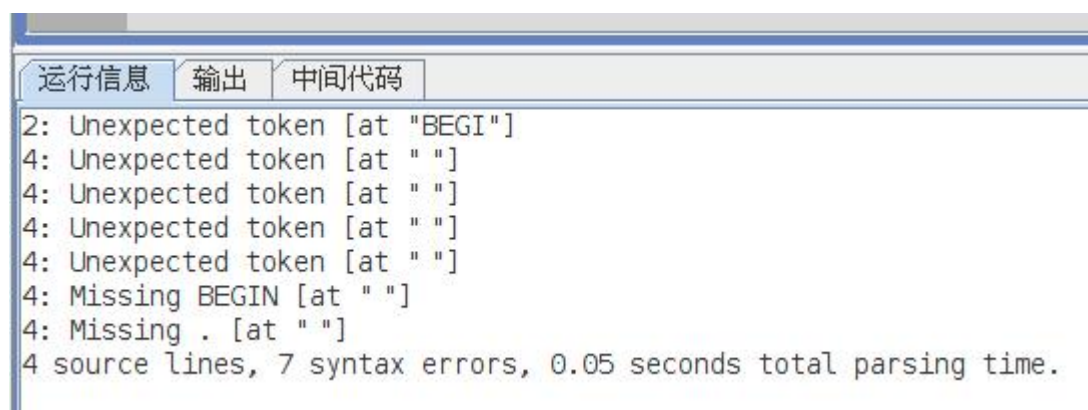
```
1 PROGRAM HelloWorld; {第一个ToyPascal程序}
2 BEGIN
3 $$$_{非法标志符现实为红色}
4     writeln('Hello World!')
5 END.
```

### 5.1.2 错误用例 2

也可以编译程序获取更具体的错误信息。如程序：

```
1 PROGRAM HelloWorld; {第一个ToyPascal程序}
2 BEGI {缺少字符N,报错}
3     writeln('Hello World!'{缺少有括号}
4 END.
```

会先显示出错行，后现显示错误类型，由于上一个错误会导致一系列**连锁错误**，而 ToyPascal 在遇到错误时会尝试**修复错误**（见 5.4 错误处理的实现），而不是简单退出，所以会显示多余两条错误（上面代码实际错误数量）。根据报错信息修复错误是应该从最上面的错误修复。



如过现在修复 BEGIN 的错误：

```
1 PROGRAM HelloWorld; {第一个ToyPascal程序}
2 BEGIN {修复}
3     writeln('Hello World!') {缺少有括号}
4 END.
```

那么报错会急剧减少，



## 5.2 语法解析错误

语法错误，指每个标志符合法，但语法结构错误。以下是几个例子

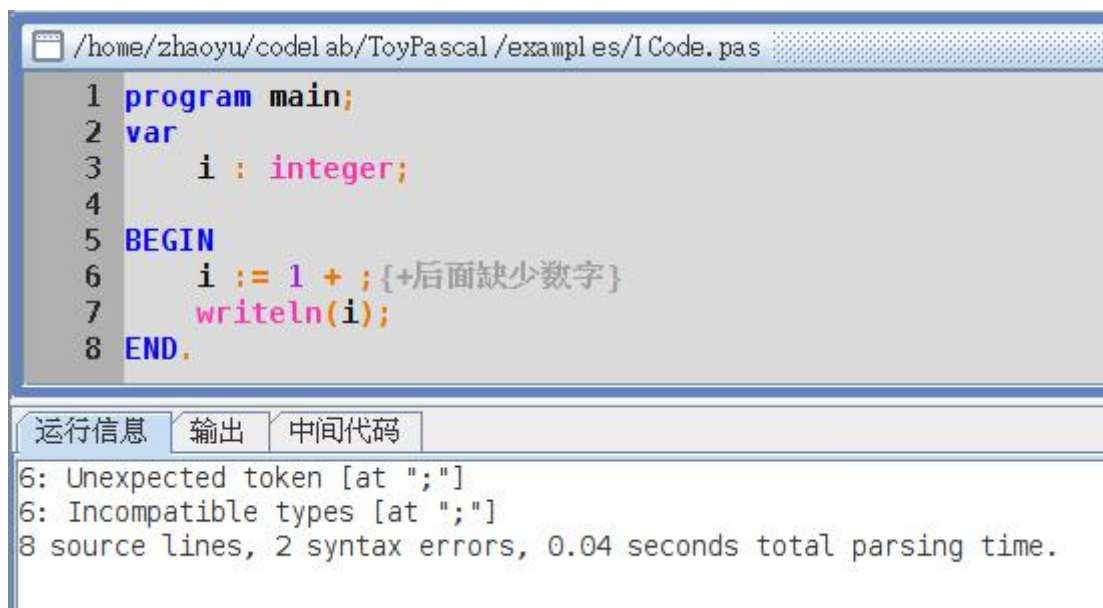
### 5.2.1 表达式缺少成分

第 6 行缺少一个数字，编译过程中会报错，并指出可能的错误类型。以

1. 6: Unexpected token [at ‘;’]

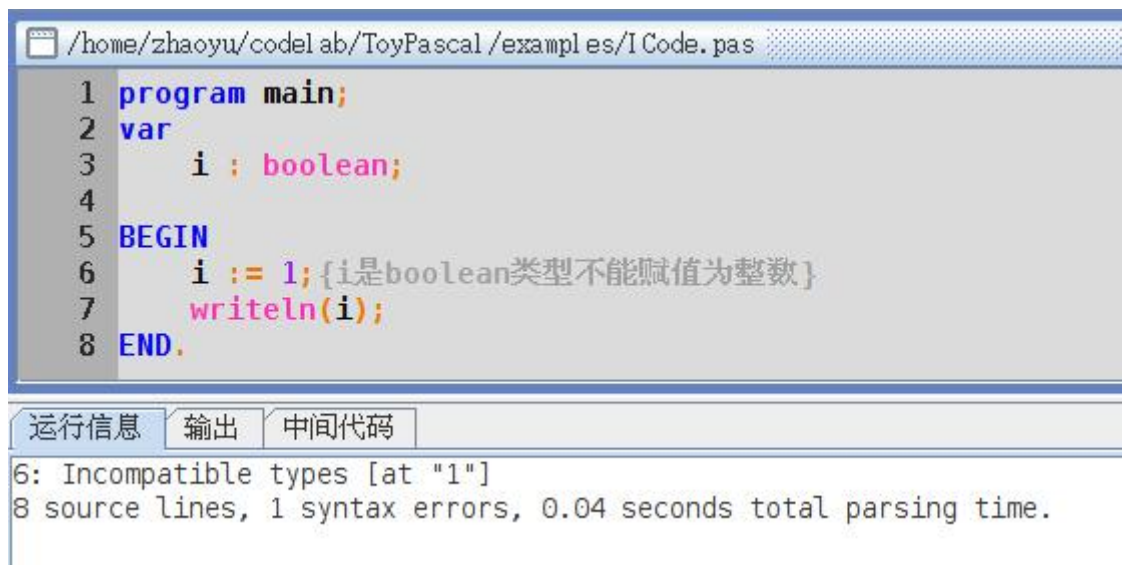
为例（详细见下图），**6** 指出了出错的行，**[at ‘;’]** 进一步指出了错误的具体位置。

**Unexpected token** 指出+后面出现了一个与文法不匹配的标记符。



### 5.2.2 类型不匹配

如下，当尝试给 boolean 型赋值整数是类型报错（**Incompatible types**）



```
/home/zhaoyu/code lab/ToyPascal/examples/I Code.pas
1 program main;
2 var
3     i : boolean;
4
5 BEGIN
6     i := 1; {i是boolean类型不能赋值为整数}
7     writeln(i);
8 END.
```

运行信息 输出 中间代码

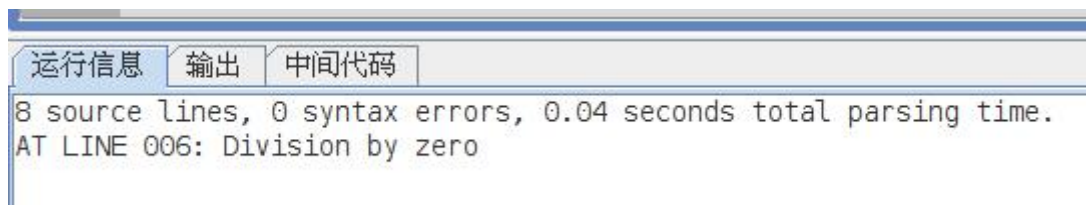
6: Incompatible types [at "1"]  
8 source lines, 1 syntax errors, 0.04 seconds total parsing time.

### 5.3 运行时错误

对于运行是错误，比如 **1/0** 编译通过，但运行是报错。

```
1 program main;
2 var
3     i : real;
4
5 BEGIN
6     i := 1 / 0;
7     writeln(i);
8 END.
```

编译信息（**AT LINE 006...**是运行时错误，第一行表明词法、语法分析通过）



运行信息 输出 中间代码

8 source lines, 0 syntax errors, 0.04 seconds total parsing time.  
AT LINE 006: Division by zero

## 运行信息

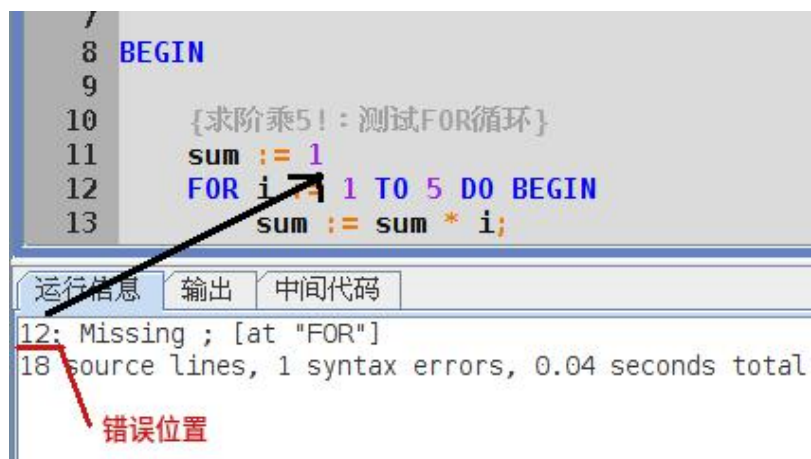


## 5.4 错误处理的实现

错误处理的实现比较复杂，这里做简要介绍。

### 5.4.1 错误定位的实现

所谓定位就是指明错误出现的行数，简单来说可以通过记录（通过符号表）每个 token 标志符的行信息（在词法分析时），并在循环、函数调用等**不按顺序执行**的语句中记录跳转行数来实现：



### 5.4.2 类型检查的实现

类型检查利用了**属性文法**和符号表的信息：

## 类型检查

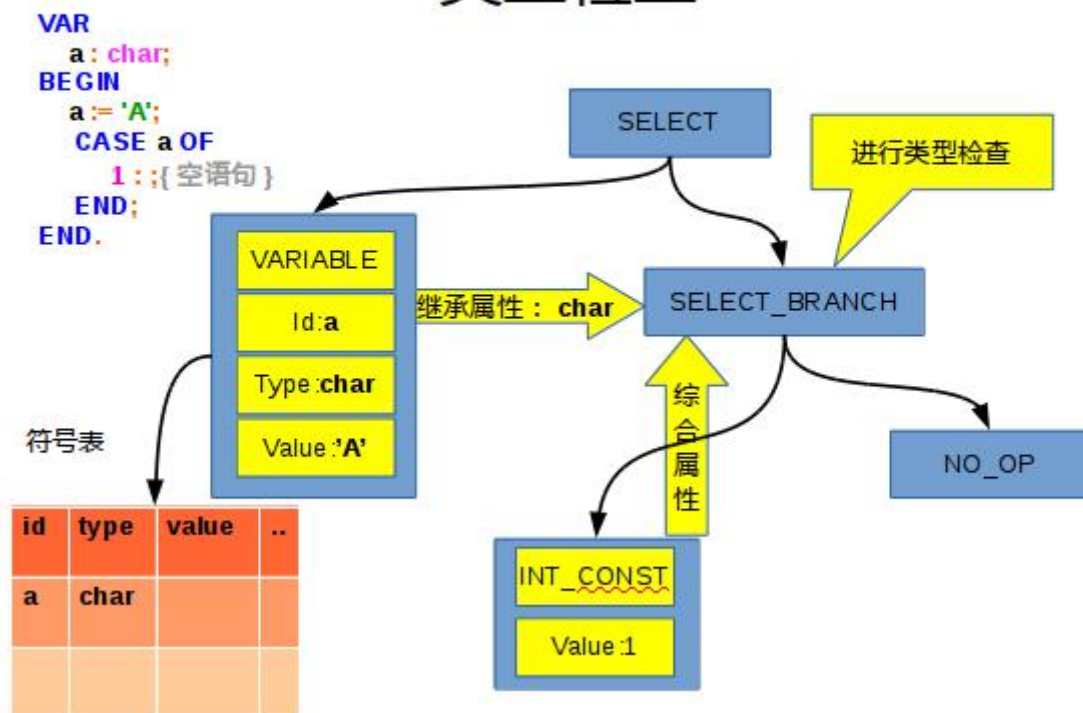


图 5.1 类型检查

例子:

```
/home/zhaoyu/code/ab/ToyPascal/examples/1Code.pa
1 PROGRAM Error;
2 VAR
3   a : char;
4 BEGIN
5   a := 'A';
6   CASE a OF
7     1 :: {空语句}
8   END;
9 END.
```

类型冲突

运行信息 输出 中间代码

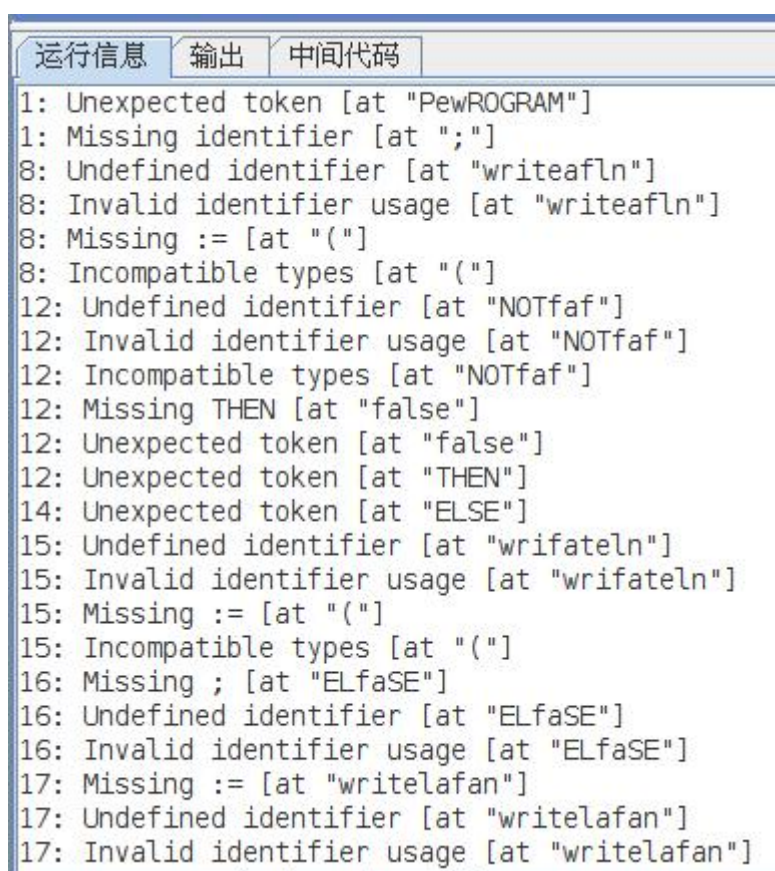
7: Incompatible types [at "1"]  
9 source lines, 1 syntax errors, 0.04 seconds

图 5.2 类型冲突的例子



### 5.4.3 错误的恢复

一般来说，为了实现简单在遇到错误时直接退出，本程序采取了一定的恢复机制，比如：`i := 1 + ;`是一个有错误的语句。但`;`是`+`**FOLLOW**集的**FOLLOW**集。所以可以认为这个错误不会影响下一个语句的分析。比起遇到错误就退出，进行错误恢复并继续分析可以提供更多更丰富的错误信息：



```
运行信息  输出  中间代码
1: Unexpected token [at "PewROGRAM"]
1: Missing identifier [at ";"]
8: Undefined identifier [at "writeafln"]
8: Invalid identifier usage [at "writeafln"]
8: Missing := [at "("]
8: Incompatible types [at "("]
12: Undefined identifier [at "NOTfaf"]
12: Invalid identifier usage [at "NOTfaf"]
12: Incompatible types [at "NOTfaf"]
12: Missing THEN [at "false"]
12: Unexpected token [at "false"]
12: Unexpected token [at "THEN"]
14: Unexpected token [at "ELSE"]
15: Undefined identifier [at "wriafatefn"]
15: Invalid identifier usage [at "wriafatefn"]
15: Missing := [at "("]
15: Incompatible types [at "("]
16: Missing ; [at "ELfaSE"]
16: Undefined identifier [at "ELfaSE"]
16: Invalid identifier usage [at "ELfaSE"]
17: Missing := [at "writelafan"]
17: Undefined identifier [at "writelafan"]
17: Invalid identifier usage [at "writelafan"]
```

图 5.3 通过尝试恢复错误提供丰富的错误提示

#### 5.4.4 错误类型

**PascalErrorCode** 类枚举了所有可能的错误类型：

```
public enum PascalErrorCode {  
    ALREADY_FORWARDED("Already specified in FORWARD"),  
    CASE_CONSTANT_REUSED("CASE constant reused"),  
    IDENTIFIER_REDEFINED("Redefined identifier"),  
    IDENTIFIER_UNDEFINED("Undefined identifier"),  
    INCOMPATIBLE_ASSIGNMENT("Incompatible assignment"),  
    INCOMPATIBLE_TYPES("Incompatible types"),  
    INVALID_ASSIGNMENT("Invalid assignment statement"),  
    INVALID_CHARACTER("Invalid character"),  
    INVALID_CONSTANT("Invalid constant"),  
    INVALID_EXPONENT("Invalid exponent"),  
    INVALID_EXPRESSION("Invalid expression"),  
    INVALID_FIELD("Invalid field"),  
    INVALID_FRACTION("Invalid fraction"),  
    INVALID_IDENTIFIER_USAGE("Invalid identifier usage"),  
    INVALID_INDEX_TYPE("Invalid index type"),  
    INVALID_NUMBER("Invalid number"),  
    INVALID_STATEMENT("Invalid statement"),  
    INVALID_SUBRANGE_TYPE("Invalid subrange type"),  
    INVALID_TARGET("Invalid assignment target"),  
    INVALID_TYPE("Invalid type"),  
}
```

#### 5.4 PascalErrorCode 中的错误类型

## 6 调试器

调试器主要利用了符号表收集的信息，不属于要求完成的内容。这里做一个简单的演示：

如下代码：

```
1. . PROGRAM Scope;
2.  VAR
3.    test : integer;
4.
5.  PROCEDURE layer1;
6.    VAR
7.      test : real;
8.  PROCEDURE layer2;
9.    VAR
10.     test : char;
11.  BEGIN
12.    test := 'A';
13.    writeln(test);
14.  END;
15. BEGIN
16.  layer2;
17.  test := 2.0;
18.  writeln(test);
19.  END;
20. BEGIN
21.  layer1;
22.  test := 1;
23.  writeln(test);
24. END.
```



启动调试器：（在提交代码的 release 目录下）

```
$ java -jar Debugger.jar execute Scope.pas
```

设置断点并运行：

```
At line20  
(pdb) break 13;  
(pdb) go;  
  
Breakpoint at line 13  
(pdb) █
```

查看调用栈（test 在不同作用域有不同值）：

```
(pdb) stack;  
2: PROCEDURE layer2  
  test: 'A'  
1: PROCEDURE layer1  
  test: 2.0  
0: PROGRAM scope  
  test: 1  
(pdb) █
```

查看变量：（当前调用栈）

```
(pdb) show test;  
'A'  
(pdb) █
```

跟踪变量（当前调用栈），单步运行：

```
(pdb) watch test;  
(pdb) step;  
  
At line 13:test: A  
A  
  
At line18  
(pdb) █
```

退出调试:

```
(pdb) quit;  
Program terminated.
```

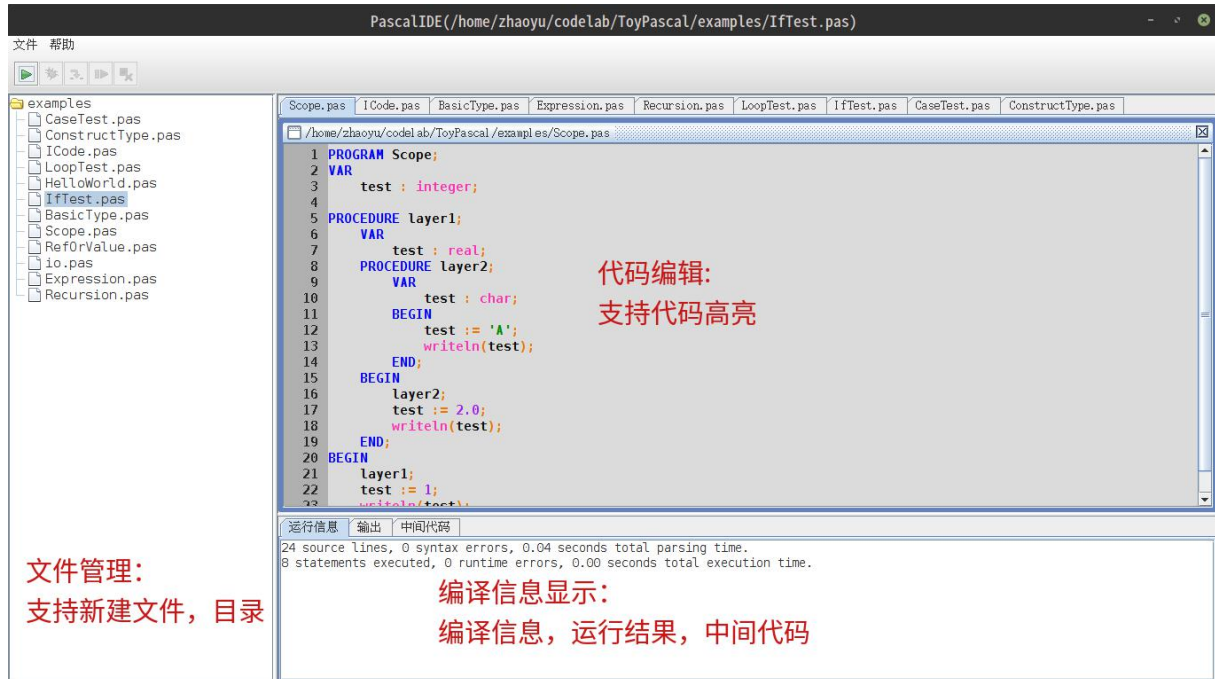
整体截图:

```
At line20  
(pdb) break 13;  
(pdb) go;  
  
Breakpoint at line 13  
(pdb) stack;  
2: PROCEDURE layer2  
  test: 'A'  
1: PROCEDURE layer1  
  test: 2.0  
0: PROGRAM scope  
  test: 1  
(pdb) show test;  
'A'  
(pdb) wacth test;  
!!! ERROR: Invalid command: 'wacth'.  
(pdb) watch test;  
(pdb) step;  
  
At line 13:test: A  
A  
  
At line18  
(pdb) quit;  
Program terminated.
```

图 6.1 调试器功能演示

## 7 图形界面

图形界面如下：



6 调试器中的功能目前未在图形界面中实现，因为涉及到 java 线程间的异步 IO。

## 8 感想

### 8.1 课设总结

本次课设由于提前开始准备，实现了比较多的功能。但也有一些预期的功能没有实现。

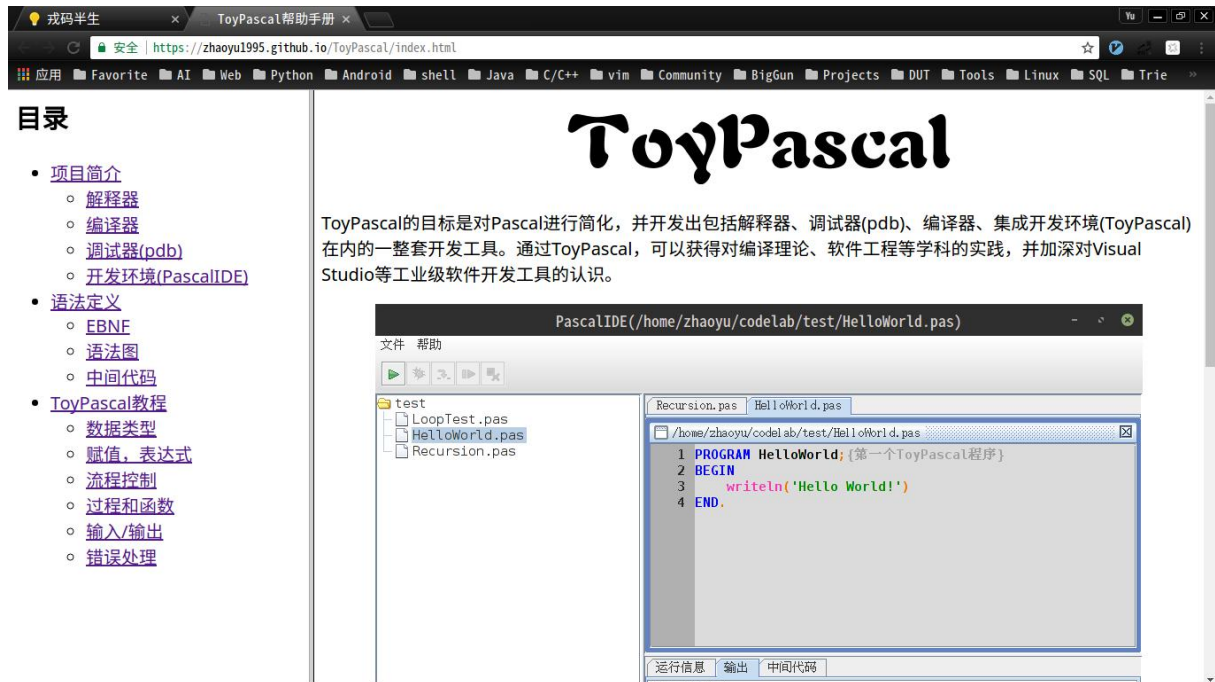
实现的功能	性质
词法分析	基本要求
语法分析	
中间代码生成	
解释执行	附加功能
循环语句	
枚举、数组类型	
函数调用	
图形界面	
调试器	

### 8.2 收获

通过本次课设（实际是断断续续几个月）对 ToyPascal 的开发。加深了对编译原理的理解，同时编译程序也是大学以来自己写的**最大规模**的程序，词法分析、语法分析、错误处理、解释执行等各部分代码的设计，整体程序的架构都让我得到了巨大的锻炼。

通过本次课设，也加深了我对编程语言和编译理论的兴趣，ToyPascal 将作为一个长期项目继续开发，在实践中不断加深对编译理论的理解。

ToyPascal 项目网站: <https://zhaoyu1995.github.io/ToyPascal/index.html>



## 9 附录

ToyPascal 的 EBNF:

```

program = PROGRAM identifier ["(" identifier {"," identifier"} ")"] ";" block ".".

block = declarations compound_statement.

declarations =      ["CONST" constant_definition ";" {constant_definition ";"}] \n
                    ["TYPE" type_definition ";" {type_definition ";"}] \n
                    ["VAR" variable_definition ";" {variable_declaration ";"}] \n
                    [{"PROCEDURE" identifier formal_parameter_list | \n
                        "FUNCTION" identifier [formal_parameter_list] ":" identifier} \n
                    ";" block ";"].

constant_definition = identifier "==" constant.

```

```
variable_declaration = variable_or_field_declaration.

type_definition = identifier "=" type_specification.

type_specification = simple_type | array_type | record_type.

simple_type = identifier | "(" identifier {"", identifier} ")" | (constant ".." constant).

array_type = "ARRAY" "[" identifier {"", identifier} "]" "OF" type_specification.

record_type = "RECORD" field_declaration {"", field_declaration} "END".

field_declaration = variable_or_field_declaration.

variable_or_field_declaration = identifier {"", identifier} ":" type_specification.

formal_parameter_list = "(" formal_parameter {"", formal_parameter} ")".

formal_parameter = ["VAR"] identifier {"", identifier} ":" identifier.

compound_statement = "BEGIN" statement_list "END".

statement_list = statement {"", statement}.

statement = compound_statement | assignment_statement | WHILE_statement |
REPEAT_statement | FOR_statement | IF_statement | CASE_statement | procedure_call.

assignment_statement = variable "!=" expression.

WHILE_statement = "WHILE" expression "DO" statement.

REPEAT_statement = "REPEAT" statement_list "UNTIL" expression.

FOR_statement = "FOR" identifier "!=" expression ("TO" | "DOWNTO") expression "DO" statement.

IF_statement = "IF" expression "THEN" statement ["ELSE" statement].

CASE_statement = "CASE" expression "OF" [constant_list ":" statement {"", constant_list ":"
statement}] "END".

constant_list = constant {"", constant}.

constant = string | (["+" | "-"] (identifier | number)).

expression = simple_expression [{"=" | "<" | ">=" | ">" | "<=" | "<"} simple_expression].

simple_expression = ["+" | "-"] term [{"+" | "-" | "OR"} term].
```

```
term = factor {("*" | "/" | "DIV" | "MOD" | "AND") factor}.

factor = variable | number | string | "NOT" factor | "(" expression ")" | function_call .

procedure_call = declared_routine_call.

function_call = declared_routine_call.

declared_routine_call = ["(" expression {"," expression} ")"].

variable = identifier.

identifier = word.

letter = A | Z | a | "." | z .

word = letter {letter | digit}.

string = "'" {[ char_except_quote | "'" "'] } "'".

unsigned_integer = digit {digit}.

number = unsigned_integer [ "." unsigned_integer ] ("E" | "e") ["+" | "-"] unsigned_integer .

digit = "0" | "1" | "2" | "." | "9" .
```