1. 语义分析与类型检查必做样例

样例1：

输入：

```
1 int main() 2  {
3       int i = 0;
4       j = i + 1;
5  }
```

输出：

样例输入中变量“j”未定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 1 at Line 4: Undefined variable "j".
```

样例2：

输入：

```
1 int main() 2  {
3       int i = 0;
4       inc(i);
5  }
```

输出：

样例输入中函数“inc”未定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 2 at Line 4: Undefined function "inc".
```

样例3：

输入：

```
1 int main() 2  {
3       int i, j;
4       int i;
5  }
```

输出：

样例输入中变量“i”被重复定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 3 at Line 4: Redefined variable "i".
```

样例4：

输入：

```
 1 int func(int i) 2  {
 3       return i;
 4  }
 5
 6 int func() 7  {
 8       return 0;
 9  }
10
11 int main() 12  {
13  }
```

输出：

样例输入中函数“func”被重复定义，因此你的程序可以输出如下的错误提示信息：

```
Error type 4 at Line 6: Redefined function "func".
```

样例5：

输入：

```
1 int main() 2 {
3       int i;
4       i = 3.7;
5 }
```

输出：

样例输入中错将一个浮点常数赋值给一个整型变量，因此你的程序可以输出如下的错误提示信息：

```
Error type 5 at Line 4: Type mismatched for assignment.
```

样例6：

输入：

```
1 int main() 2 {
3       int i;
4       10 = i;
5 }
```

输出：

样例输入中整数“10”出现在了赋值号的左边，因此你的程序可以输出如下的错误提示信息：

```
Error type 6 at Line 4: The left-hand side of an assignment must be a varia- ble.
```

样例7：

输入：

```
1 int main() 2 {
3       float j;
4       10 + j;
5 }
```

输出：

样例输入中表达式“10 + j”的两个操作数的类型不匹配，因此你的程序可以输出如下的错误提示信息：

```
Error type 7 at Line 4: Type mismatched for operands.
```

样例8：

输入：

```
1 int main() 2 {
3       float j = 1.7;
4       return j;
5 }
```

输出：

样例输入中“main”函数返回值的类型不正确，因此你的程序可以输出如下的错误提示信息：

```
Error type 8 at Line 4: Type mismatched for return.
```

样例9：

输入：

```
1 int func(int i) 2  {
3       return i;
4  }
5
6 int main() 7  {
8       func(1, 2);
9  }
```

输出：

样例输入中调用函数“func”时实参数目不正确，因此你的程序可以输出如下的错误提示信息：Error type 9 at Line 8: Function "func(int)" is not applicable for arguments "(int, int)".

样例10：

输入：

```
1 int main() 2  {
3       int i;
4       i[0];
5  }
```

输出：

样例输入中变量“i”非数组型变量，因此你的程序可以输出如下的错误提示信息：

```
Error type 10 at Line 4: "i" is not an array.
```

样例11：

输入：

```
1 int main() 2  {
3       int i;
4       i(10);
5  }
```

输出：

样例输入中变量“i”不是函数，因此你的程序可以输出如下的错误提示信息：

```
Error type 11 at Line 4: "i" is not a function.
```

样例12：

输入：

```
1 int main() 2  {
3       int i[10];
4       i[1.5] = 10;
5  }
```

输出：

样例输入中数组访问符中出现了非整型常数“1.5”，因此你的程序可以输出如下的错误提示信息：

```
Error type 12 at Line 4: "1.5" is not an integer.
```

样例13：

输入：

```
1 struct Position2 {
3       float x, y;
4 };
5
6 int main() 7 {
8       int i;
9       i.x;
10 }
```

输出：

样例输入中变量“i”非结构体类型变量，因此你的程序可以输出如下的错误提示信息：

```
Error type 13 at Line 9: Illegal use of ".".
```

样例14：

输入：

```
1 struct Position 2 {
3       float x, y;
4 };
5
6 int main() 7 {
8       struct Position p; 9       if
(p.n == 3.7)
10          return 0;
11 }
```

输出：

样例输入中结构体变量“p”访问了未定义的域“n”，因此你的程序可以输出如下的错误提示信息：

```
Error type 14 at Line 9: Non-existent field "n".
```

样例15：

输入：

```
1 struct Position 2 {
3       float x, y;
4       int x;
5 };
6
7 int main() 8 {
9 }
```

输出：

样例输入中结构体的域“x”被重复定义，因此你的程序可以输出如下的错误信息：

```
Error type 15 at Line 4: Redefined field "x".
```

样例16：

输入：

```
1 struct Position 2 {
3       float x;
4 };
5
6 struct Position
7 {
```

```
 8      int y;
 9  };
10
11 int main()
12  {
13  }
```

输入：

样例输入中两个结构体的名字重复，因此你的程序可以输出如下的错误信息：

Error type 16 at Line 6: Duplicated name "Position".

样例17：

输入：

```
 1 int main()
 2 {
 3       struct Position pos;
 4 }
```

输出：

样例输入中结构体“Position”未经定义，因此你的程序可以输出如下的错误信息：

Error type 17 at Line 3: Undefined structure "Position".

2. 中间代码生成必做样例

样例1：

输入：

```
1 int main()
2  {
3    int n;
4    n = read();
5    if (n > 0) write(1);
6    else if (n < 0) write (-1);
7    else write(0);
8    return 0;
9  }
```

输出：

这段程序读入一个整数n，然后计算并输出符号函数sgn(x)。它所对应的中间代码可以是这样的：

```
 1  FUNCTION main :
 2  READ t1
 3 v1 := t1
 4 t2 := #0
 5  IF v1 > t2 GOTO label1
 6  GOTO label2
 7  LABEL label1 :
 8 t3 := #1
 9  WRITE t3
10  GOTO label3
11  LABEL label2 :
12 t4 := #0
13  IF v1 < t4 GOTO label4
14  GOTO label5
15  LABEL label4 :
```

```
16 t5 := #1
17 t6 := #0 - t5
18  WRITE t6
19  GOTO label6
20  LABEL label5 :
21 t7 := #0
22  WRITE t7
23  LABEL label6 :
24  LABEL label3 :
25  t8 := #0
26  RETURN t8
```

需要注意的是，虽然样例输出中使用的变量遵循着字母t后跟一个数字（如t1、v1等）的方式，标号也遵循着label后跟一个数字的方式，但这并不是强制要求的。也就是说，你的程序输出完全可以使用其它符合变量名定义的方式而不会影响虚拟机小程序的运行。

可以发现，这段中间代码中存在很多可以优化的地方。首先，0这个常数我们将其赋给了t2、t4、t7、t8这四个临时变量，实际上赋值一次就可以了。其次，对于t6的赋值我们可以直接写成t6 := #–1而不必多进行一次减法运算。另外，程序中的标号也有些冗余。如果你的程序足够“聪明”，可能会将上述中间代码优化成这样：

```
 1  FUNCTION main :
 2  READ t1
 3 v1 := t1
 4 t2 := #0
 5  IF v1 > t2 GOTO label1
 6  IF v1 < t2 GOTO label2
 7  WRITE t2
 8  GOTO label3
 9  LABEL label1 :
10 t3 := #1
11  WRITE t3
12  GOTO label3
13  LABEL label2 :
14 t6 := #-1
15  WRITE t6
16  LABEL label3 :
17  RETURN t2
```

样例2：

输入：

```
 1 int fact(int n)
 2 {
 3   if (n == 1)
 4     return n;
 5   else
 6     return (n * fact(n - 1));
 7 }
 8 int main()
 9 {
10   int m, result;
11   m = read();
12   if (m > 1)
13     result = fact(m);
14   else
15     result = 1;
16   write(result);
17   return 0;
18 }
```

输出：

这是一个读入m并输出m的阶乘的小程序，其对应的中间代码可以是：

```
 1  FUNCTION fact :
 2  PARAM v1
 3  IF v1 == #1 GOTO label1
 4  GOTO label2
 5  LABEL label1 :
 6  RETURN v1
 7  LABEL label2 :
 8  t1 := v1 - #1
 9  ARG t1
10  t2 := CALL fact
11 t3 := v1 * t2
12 RETURN t3
13
14  FUNCTION main :
15  READ t4
16 v2 := t4
17  IF v2 > #1 GOTO label3
18  GOTO label4
19  LABEL label3 :
20  ARG v2
21  t5 := CALL fact
22 v3 := t5
23  GOTO label5
24  LABEL label4 :
25 v3 := #1
26  LABEL label5 :
27  WRITE v3
28  RETURN #0
```

这个样例主要展示如何处理包含多个函数以及函数调用的输入文件