1.1 编写一个正确的 Oberon-0 源程序

1.1.1 代码编写

- 根据实验要求，首先给出一个正确的程序，尽量覆盖了Oberon-0语言提供的模块、声明（类型、常量、变量等）、过程声明与调用、语句、表达式等各种构造
- 下面的代码实现了指数运算过程`Power`、质因数分解并判断质合性奇偶性过程`PrimeFactor`

```
(* 正确版本 *)

MODULE Main;
    CONST zero = 0;
    TYPE bool = BOOLEAN;
    VAR num, pow, result: INTEGER;

    (* 计算num^pow *)
    PROCEDURE Power(num, pow: INTEGER);
        VAR i, result: INTEGER;
    BEGIN
        result := 1;
        i := 0;

        WHILE i < pow DO
            result := result * num;
            i := i + 1
        END;

        WRITE(result)
    END Power;

    (* 计算num的质因数，并以第一个质因数判断num质合性、奇偶性 *)
    PROCEDURE PrimeFactor(num : INTEGER);
        VAR fac : ARRAY 100 OF RECORD
                value : INTEGER;
                evenflag : bool;
            END;
            i, j, temp : INTEGER;
    BEGIN
        i := 2;
        j := zero;
        temp := num;
        WHILE ~(temp <= 1) DO
            WHILE temp MOD i = zero DO
                j := j + 1;
                fac[j].value := i;
                fac[j].evenflag := (i = 2);
                temp := temp div i
            END;
            i := i + 1
        END;

        (* 如果num为质数输出0 *)
        IF (fac[0].value = num) OR (fac[0].value = 1) THEN
```

```
            WRITE(0)

        (* 如果num为奇合数输出1 *)
        ELSIF ~fac[ZERO].evenflag & ~(fac[ZERO].value = 2) THEN
            write(1)

        (* 如果num为偶合数输出2 *)
        ELSE
            WriteLN(2)
        END
    END PrimeFactor;

    BEGIN
        Read(num);
        Read(pow);

        Power(num, pow);
        PrimeFactor(num)
    END Main.
```

1.1.2 包含内容

- 上述代码经过词法语法语义的验证、以及后面实验的验证，可以基本认为是正确的
- 上述代码尽量覆盖了Oberon-0的各种构造方法，包括：
  - **模块**

```
MODULE Main;
    ...
    END Main.
```

  - **声明**

```
CONST zero = 0;
TYPE bool = BOOLEAN;
VAR num, pow, result: INTEGER;
```

  - **类型**

```
VAR fac : ARRAY 100 OF RECORD
                value : INTEGER;
                evenflag : bool;
            END;
            i, j, temp : INTEGER;
```

  - **过程声明与调用**
    - 过程声明

```
PROCEDURE Power
...
END Power;

PROCEDURE PrimeFactor
...
END PrimeFactor;
```

- 调用

```
Power(num, pow);
PrimeFactor(num)
```

- **语句**
  - 循环

```
WHILE...DO
...
END;
```

  - 判断

```
IF...THEN
...
ELSIF...THEN
...
ELSE
...
END
```

  - 注释

```
(* ... *)
```

- **表达式**
  - 整形运算

```
result := result * num;
i := i + 1
temp MOD i = zero
temp := temp div i
```

  - 布尔运算

```
~(temp <= 1)
~fac[ZERO].evenflag & ~(fac[ZERO].value = 2)
```

  - 数组和记录引用

```
fac[j].value := i;
fac[j].evenflag := (i = 2);
```

1.2 编写上述 Oberon-0 源程序的变异程序

1.2.1 异常分类

- 由于实验软装置并没有给出异常源代码，故参考实验2中的异常分类自行给出各种可能的异常
- 根据编译器前端的性质，可以给出如下两层粗细粒度的异常分类
  - `LexicalException`
    - `IllegalIdentifierLengthException`
    - `IllegalIntegerException`
    - `IllegalIntegerRangeException`
    - `IllegalOctalException`
    - `IllegalSymbolException`
    - `MismatchedCommentException`
  - `SemanticException`
    - `MissingLeftParenthesisException`
    - `MissingOperandException`
    - `MissingOperatorException`
    - `MissingRightParenthesisException`
  - `SyntacticException`
    - `ParameterMismatchedException`
    - `TypeMismatchedException`

1.2.2 代码编写

- 根据实验要求，在每个变异版本的代码顶部指明错误类型，同时在错误的代码段处添加了对应注释
- 代码形式与下面给出的类似，其中`main.obr`为正确版本，`main.*`为变异版本，下面给出`main.001`作为示例：

```
(* 变异版本 LexicalException--IllegalSymbolException *)

MODULE Main;
    CONST zero = 0;
    TYPE bool = BOOLEAN;
    VAR num, pow, result: INTEGER;

    (* 计算num^pow *)
    PROCEDURE Power(num, pow: INTEGER);
        VAR i, result: INTEGER;
    BEGIN
        result := 1;
        i := 0;
```

```
14
15          WHILE i < pow DO
16              result := result * num;
17              i := i + 1
18          END;
19
20          WRITE(result)
21      END Power;
22
23      (* 计算num的质因数，并以第一个质因数判断num质合性、奇偶性 *)
24      PROCEDURE PrimeFactor(num : INTEGER);
25          VAR fac : ARRAY 100 OF RECORD
26                  value : INTEGER;
27                  evenflag : bool;
28              END;
29              i, j, temp : INTEGER;
30      BEGIN
31          i := 2;
32          j := zero;
33          temp := num;
34          WHILE ~(temp <= 1) DO
35              WHILE temp % i = zero DO    (* 非法操作符 *)
36                  j := j + 1;
37                  fac[j].value := i;
38                  fac[j].evenflag := (i = 2);
39                  temp := temp div i
40              END;
41              i := i + 1
42          END;
43
44          (* 如果num为质数输出0 *)
45          IF (fac[0].value = num) OR (fac[0].v1a2l3u4e5 = 1) THEN (* 非法
    标识符 *)
46              WRITE(0)
47
48          (* 如果num为奇合数输出1 *)
49          ELSIF ~fac[ZERO].evenflag & ~(fac[ZERO].value = 2) THEN
50              write(1)
51
52          (* 如果num为偶合数输出2 *)
53          ELSE
54              WriteLN(2)
55          END
56      END PrimeFactor;
57
58      BEGIN
59          Read(num);
60          Read(pow);
61
62          Power(num, pow);
63          PrimeFactor(num)
64      END Main.
```

1.3 讨论 Oberon-0 语言的特点

1.3.1 保留字和关键字的区别

保留字和关键字核心思想在于区分语言的语法结构和语言预定义的标准功能，本质上是为了方便程序员在不同粒度上使用语言，在保证语言正确性稳定性的基础上、允许程序员自由编程

1.3.1.1 保留字

- 保留字是构成语言语法基础的词，它们是语言结构的一部分，用于定义程序的控制流、声明结构、模块边界等
- 保留字的含义在编译器内部是写死的，在进行语法分析时，编译器会将保留字识别为特定的语法符号
- 保留字是BNF或EBNF中严格定义的终结符，直接决定了句子的类型和结构
    - `MODULE`标志着模块的开始
    - `PROCEDURE`标志着过程的开始
    - `IF`标志着条件语句的开始
    - `WHILE`标志着循环语句的开始
    - ...
- 保留字不能被使用作为变量名、常量名、过程名、模块名等标识符，也不能被重定义

1.3.1.2 关键字

- 关键字也被称作标准标识符、预定义标识符
- 关键字是语言标准库或环境中预先定义好的标识符，通常代表了基本数据类型、内建常量、标准过程和函数等
- 关键字本质上是预先在全局作用域声明好的标识符，拥有默认的含义
    - `INTEGER` 是预定义的类型标识符
    - `TRUE` 是预定义的布尔常量
    - `WRITE` 是预定义的标准过程
    - ...
- 关键字在语法上可以在局部作用域中被重新声明，这是与保留字最大的不同，比如上面代码中的：

```
1  TYPE bool = BOOLEAN;
```

1.3.2 表达式语法的区别

1.3.2.1 符号区别

- 赋值、相等、不等号、逻辑与、逻辑或、逻辑非等符号具有同样的表达式语义，但是两种语法的对应符号不同
- Oberon-0：`:=`、`=`、`#`、 `AND`、 `OR`、 `~`、...
- C++/Java：`=`、`==`、`!=`、 `&&`、 `||`、 `!`、...

1.3.2.2 表达式性质区别

- Oberon-0中表达式服务于赋值语句，单独的表达式不能作为一个语句，只有赋值语句右部的表达式才是合法的表达式
- C++/Java单独的表达式能作为一个语句，这是因为在编译器内部实际上表达式是具有返回值的，任何表达式的语义动作并不直接依赖于表达式本身、而是依赖于表达式的返回值
- C++/Java的表达式语句比Oberon-0有更强的独立性，支持重载运算符、链式赋值`a = b = c`等复杂但是合理的操作

1.3.2.3 类型区别

- Oberon-0是强类型语言，任何类型不匹配的表达式都会被编译器抛出异常
- C++/Java是弱类型语言、允许隐式类型转换，大部分合法的类型不匹配的表达式会被编译器进行隐式类型转换

1.3.2.4 优先级区别

- Oberon-0操作符集合非常精简，优先级层次很少
- C++/Java操作符集合复杂庞大，具有十几个优先级层次

1.4 讨论 Oberon-0 文法定义的二义性

1.4.1 是否存在二义性

个人认为Oberon-0文法不具有二义性，下面主要讨论几个容易产生二义性的内容

1.4.1.1 悬挂`ELSE`

- 对应BNF:

```
if_statement = "IF" expression "THEN"
                    statement_sequence
               {"ELSIF" expression "THEN"
                    statement_sequence}
               ["ELSE"
                    statement_sequence]
               "END" ;
```

- 每个`IF`语句都必须由一个`END`来显示关闭，此时任何合法的句子不会出现`ELSE`悬挂、或者`IF`匹配到另外的条件分支
- 类比表达式中每个子表达式都使用括号时，即使没有优先级和结合性的约束，也不会发生二义性问题

1.4.1.2 操作符优先级与结合性

- 对应BNF:

```
expression = simple_expression [("=" | ...) simple_expression] ;
simple_expression = ["+" | "-"] term {("+" | "-" | "OR") term} ;
term = factor {("*" | "DIV" | "MOD" | "&") factor} ;
factor = ... ;
```

- 通过分层隐式地考虑了优先级：
  - 高优先级的操作符，推导时相对晚产生、归约时相对早被归约，比如`*`、`DIV`
  - 低优先级的操作符，推导时相对晚产生、归约时相对早被归约，比如`=`
  - 这样保证了在语法分析树上，高优先级运算符对应的表达式总是在低优先级运算符的下层、自然也就更早归约产生计算结果
- 通过`Y {op Y}`、`{Y op} Y`的形式隐式地考虑了结合性：
  - `Y {op Y}`对应左结合，在循环应用右侧`{op Y}`以进行匹配时，总是左边的表达式先被匹配
  - `{Y op} Y`对应右结合，在循环应用左侧侧`{Y op}`以进行匹配时，总是右边的表达式先被匹配
  - 先被匹配的表达式先产生结果，自然也就符合了结合性的计算结果

1.4.1.3 赋值语句与布尔表达式

- 对应BNF：

```
assignment = identifier selector ":=" expression ;
statement = [assignment | ... ] ;

expression = simple_expression ["=" simple_expression] ;
```

- 若采用`=`和`==`
  - 如果在词法分析时对两者加以区分，对现代大部分编译器都是从左到右进行字符匹配的，此时`=`和`==`显然会出现词法分析的二义性
  - 如果在词法分析时不对两者加以区分、而是把`==`当做两个`=`的话，对现代大部分编译器都是从左到右进行终结符匹配的，此时`=`和`==`第一个等于号显然会出现语法分析的二义性
- 若采用`:=`和`=`
  - 两者最左侧字符不一样，在词法分析时就可以简单的进行区分

1.4.2 对比其他高级程序设计语言

- 其他高级程序设计语言容易出现的二义性问题在上面也有提到，这些都是Oberon-0能够避免的
- 其他高级程序设计语言不对`IF`语句设置`END`来显示关闭，这可能导致 悬挂`ELSE`的二义性问题
- 其他高级程序设计语言往往不通过语法设计来避免表达式二义性，而是通过编译器内部的语义规则等进行比较复杂的优先级和结合性设置，但可能无法避免所有二义性情况
- 其他高级程序设计语言可能采用`=`和`==`作为赋值语句和布尔表达式，这就需要在词法分析或者语法分析进行额外的二义性处理