对于错误的源程序，错误处理输入输出及处理要求如下：

（1）结果文件中包含如下两种信息：错误所在的行号 错误的类别码 （行号与类别码之间只有一个空格，类别码严格按照表格中的小写英文字母）

其中错误类别码按下表中的定义输出，行号从1开始计数：

错误类型	错误类别码	解释	对应文法及出错符号（ … 表示省略该条规则后续部分）
非法符号	a	格式字符串中出现非法字符报错行号为 `<FormatString>` 所在行数。	`<FormatString> → '"'{<Char>}'"'`
名字重定义	b	函数名或者变量名在当前作用域下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为 `<Ident>` 所在行数。	`<ConstDef>→<Ident> … <VarDef>→<Ident> … <Ident> … <FuncDef>→<FuncType><Ident> … <FuncFParam> → <BType> <Ident> …`
未定义的名字	c	使用了未定义的标识符报错行号为 `<Ident>` 所在行数。	`<LVal>→<Ident> … <UnaryExp>→<Ident> …`
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的函数名所在行数。	`<UnaryExp>→<Ident>'('[<FuncRParams>]')'`
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的函数名所在行数。	`<UnaryExp>→<Ident>'('[<FuncRParams>]')'`

错误类型	错误类别码	解释	对应文法及出错符号（ … 表示省略该条规则后续部分）
无返回值的函数存在不匹配的return语句	f	报错行号为 `'return'` 所在行号。	`<Stmt>→'return' {'['<Exp>']'}';'`
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，无需考虑数据流。报错行号为函数结尾的'}' 所在行号。	`<FuncDef> → <FuncType> <Ident> '(' [<FuncFParams>] ')' <Block>` `<MainFuncDef> → 'int' 'main' '(' ')' <Block>`
不能改变常量的值	h	`<LVal>` 为常量时，不能对其修改。报错行号为 `<LVal>` 所在行号。	`<Stmt>→<LVal>'=' <Exp>';'` `<Stmt>→<LVal>'=' 'getint' '(' ')' ';'`
缺少分号	i	报错行号为分号前一个非终结符所在行号。	`<Stmt>`，`<ConstDecl>` 及 `<VarDecl>` 中的';'
缺少右小括号')'	j	报错行号为右小括号前一个非终结符所在行号。	函数调用（`<UnaryExp>`）、函数定义（`<FuncDef>`）及 `<Stmt>` 中的')'
缺少右中括号']'	k	报错行号为右中括号前一个非终结符所在行号。	数组定义（`<ConstDef>,<VarDef>,<FuncFParam>`）和使用（`<LVal>`）中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为 `'printf'` 所在行号。	`<Stmt> →'printf''('<FormatString>{,<Exp>}')'';'`
在非循环块中使用break和continue语句	m	报错行号为 `'break'` 与 `'continue'` 所在行号。	`<Stmt>→'break';' <Stmt>→'continue';'`

【注】错误 i，j，k 类型中的“前一个非终结符”强调的是在文法规则里出现在 `; ) ]` 之前的非终结符号，在分析中处理的是该非终结符产生的终结符号串的最后一个符号，也就是 `; ) ]` 本应该正常出现的位置的上一个单词。

（2）所有错误都不会出现恶意换行的情况，包括字符、字符串中的换行符、函数调用等等。

（3）其他类型的错误，错误的行号以能够断定发现出错的第一个符号的行号为准。例如有返回值的函数缺少返回语句的错误，只有当识别到函数末尾的}时仍未出现返回语句，才可以断定出错，报错行号即为}的行号。

为方便对照，下文给出了文法符号与可能存在的错误的对应关系：

```
编译单元    CompUnit → {Decl} {FuncDef} MainFuncDef
声明  Decl → ConstDecl | VarDecl
常量声明    ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
// i
基本类型    BType → 'int'
常数定义    ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
// b k
常量初值    ConstInitVal → ConstExp
    | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
变量声明    VarDecl → BType VarDef { ',' VarDef } ';' // i
变量定义    VarDef → Ident { '[' ConstExp ']' } // b
    | Ident { '[' ConstExp ']' } '=' InitVal // k
变量初值    InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
函数定义    FuncDef → FuncType Ident '(' [FuncFParams] ')' Block //
b g j
主函数定义   MainFuncDef → 'int' 'main' '(' ')' Block // g j
函数类型    FuncType → 'void' | 'int'
函数形参表   FuncFParams → FuncFParam { ',' FuncFParam }
函数形参    FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }]
//   b k
语句块     Block → '{' { BlockItem } '}'
语句块项    BlockItem → Decl | Stmt
语句  Stmt → LVal '=' Exp ';' | [Exp] ';' | Block // h i
    | 'if' '(' Cond ')' Stmt [ 'else' Stmt ] // j
    | 'for' '('[ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
    | 'break' ';' | 'continue' ';' // i m
    | 'return' [Exp] ';' // f i
    | LVal '=' 'getint''('')'';' // h i j
    | 'printf''('FormatString{,Exp}')'';' // i j l
语句 ForStmt → LVal '=' Exp   //h
表达式 Exp → AddExp 注: SysY 表达式是int 型表达式
条件表达式   Cond → LOrExp
左值表达式   LVal → Ident {'[' Exp ']'} // c k
基本表达式   PrimaryExp → '(' Exp ')' | LVal | Number
数值  Number → IntConst
一元表达式   UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' //
c d e j
        | UnaryOp UnaryExp
单目运算符   UnaryOp → '+' | '-' | '!' 注: '!'仅出现在条件表达式中
函数实参表   FuncRParams → Exp { ',' Exp }
```

```
乘除模表达式  MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
加减表达式   AddExp → MulExp | AddExp ('+' | '-') MulExp
关系表达式   RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=')
AddExp
相等性表达式  EqExp → RelExp | EqExp ('==' | '!=') RelExp
逻辑与表达式  LAndExp → EqExp | LAndExp '&&' EqExp
逻辑或表达式  LOrExp → LAndExp | LOrExp '||' LAndExp
常量表达式   ConstExp → AddExp 注：使用的Ident 必须是常量
格式字符串：
<FormatString> → '"'{<Char>}'"' // a
```

特别注意，为了避免因为测试程序中的错误导致出现语法二义性，使得语法树以错误的方式建立，我们保证：`for` 语句不会出现任何错误。