错误处理

一、分析

要求处理如下错误，整理如下。
可以将错误分成2类：语义不相关错误和语义相关错误。

1.1. 语义不相关错误

这类错误由于不涉及上下文语义，而只涉及文法等相关约束，因此判断时无需结合符号表，较为简单。

1.1.1. 非法符号

- 错误码：a
- 错误描述：FormatString 中出现非法字符，即【十进制编码为32,33,40-126的ASCII字符，''（编码92）出现当且仅当为'\n'，%d】以外的字符。
- 报错信息：FormatString 所在的行数。

应当在 `StmtPrintfParser` 中判断 FormatString 是否合法，若非法则添加到 ErrorTable 中。

1.1.2. 缺少分号

- 错误码：i
- 错误描述：文法中任何需要有 `;` 的部分缺失。
- 报错信息：分号前一个非终结符所在的行数。

应当在 `ConstDeclParser`、`VarDeclParser`、`StmtParser`、`StmtAssignParser`、`StmtBreakParser`、`StmtContinueParser`、`StmtExpParser`、`StmtGetIntParser`、`StmtPrintfParser`和`StmtReturnParser`中判断是否缺少分号，若缺少则将错误添加到 ErrorTable 中。

1.1.3. 缺少右小括号

- 错误码：j
- 错误描述：文法中任何需要有`)`的部分缺失。
- 报错信息：右小括号前一个非终结符所在的行数。

应当在 `UnaryExpFuncParser`、`FuncDefParser`、`MainFuncDefParser`、`StmtCondParser`、`StmtGetIntParser` 和 `StmtPrintfParser` 中判断是否缺少右小括号，若缺少则将错误添加到 ErrorTable 中。

1.1.4. 缺少右中括号

- 错误码：k
- 错误描述：文法中任何需要有`]`的部分缺失。
- 报错信息：右中括号前一个非终结符所在的行数。

应当在 `ConstDefParser`、`VarDefParser`、`LValParser` 和 `FuncFParamParser` 中判断是否缺少右中括号，若缺少则将错误添加到 ErrorTable 中。

1.2. 语义相关错误

这类错误需要结合符号表进行处理。

1.2.1. 名字重定义

- 错误码：b
- 错误描述：函数名或变量名在**当前作用域**下重复定义
- 报错信息：ident 所在行数

1.2.2. 未定义的名字

- 错误码：c
- 错误描述：使用了未定义的标识符
- 报错信息：Ident 所在行数。

1.2.3. 函数参数个数不匹配

- 错误码：d
- 错误描述：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
- 报错信息：函数调用语句的函数名所在行数

思路：从当前符号表中找到该函数的定义情况，在参数调用时对参数个数进行计数，两者进行匹配

1.2.4. 函数参数类型不匹配

- 错误码：e
- 错误描述：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。
- 报错信息：函数调用语句的函数名所在行数

思路：对每一个传入参数检查其维度，仅有二维数组、一维数组、整数、void和数组的部分维度情况，因此仅需考参数的维度是否匹配

二、Error 错误包

2.1. ErrorType 错误种类枚举类

枚举13种错误类型，作为`Error`对象的一个属性。

2.2. Error 错误类

描述错误的类，记录一个错误的信息（包括错误类型和所在行数），作为错误表的组成元素。

2.3. ErrorTable 错误表类

保存错误数据的容器。

三、Symbol 符号表包

符号表（Symbol Table）是编译器中的一种数据结构，用于存储源代码中出现的标识符（变量名、函数名等）以及其相关信息。

符号表的作用如下：

- 标识符管理：符号表记录了源代码中所有的标识符及其对应的属性信息，包括 **名称、类型、作用域、内存地址** 等。它提供了一个统一的地方来管理和维护标识符，使得编译器可以在需要时快速查找和访问标识符的相关信息。
- 语法语义分析：在编译过程中，语法和语义分析阶段需要对标识符进行识别和处理。符号表可以记录每个标识符在源代码中的出现位置，并与上下文进行关联。通过符号表，编译器可以检查标识符的声明和使用是否符合语法规则和语义规范。
- 冲突检测和解析：符号表还用于检测标识符的冲突情况，例如重复定义、重复声明等。编译器可以利用符号表进行检查，并给出相应的错误或警告信息。此外，符号表还可以帮助编译器解决不同作用域中的标识符名字冲突问题，确保引用正确的标识符。
- 代码生成和优化：符号表中的信息可以用于代码生成和优化过程。编译器可以根据标识符的类型和特性，生成相应的目标代码或进行代码优化。例如，根据变量类型和内存地址信息，编译器可以决定合适的寄存器分配策略或优化存取操作。

3.1. SymbolType 符号类型枚举类

枚举符号类型，作为 `Symbol` 对象的一个属性。
普通常量 CON、一维数组常量 CON1、二维数组常量 CON2、普通变量 VAR、一维数组变量 VAR1、二维数组变量 VAR2 和函数名 FUNC。

3.2. Symbol 符号类

记录单个符号的相关信息，包括该符号的名字和若干个属性。根据符号类型的不同，需要记录的属性也需要不同，但总有一些公共属性，是所有符号都需要记录的，如下。

3.2.1. name 符号名

该字段记录符号的名字。具体而言，name 应为标识符 Ident 的内容，即 `ident.getName()`。

3.2.2. lineNum 所在行数

从1开始。

3.2.3. symbolType 符号类型

包括普通常量 CON、一维数组常量 CON1、二维数组常量 CON2、普通变量 VAR、一维数组变量 VAR1、二维数组变量 VAR2 和函数名 FUNC。 具体而言：

- 对于 ConstDef 中的 Ident，symbolType 可能为 CON、CON1 或 CON2；
- 对于 VarDef 中的 Ident，symbolType 可能为 VAR、VAR1 或 VAR2；
- 对于 FuncDef 中的 Ident，symbolType 为 FUNC；
- 对于 FuncFParam 中的 Ident，symbolType可能为 VAR、VAR1 或 VAR2；
- 对于 MainFunc 中的 Ident【即main】，symbolType 为 FUNC；

对于不同类型的符号，对于常量符号、变量符号和函数符号，还有一些独有的属性需要记录。需要以示区分，如下。

3.2.4. SymbolCon 常量符号

3.2.4.1. initVal 常量初值

仅 CON 填写。

3.2.4.2. initVal1 一维数组初值

仅 CON1 填写。

3.2.4.3. initVal2 二维数组初值

仅 CON2 填写。

3.2.5. SymbolVar 变量符号

3.2.6. SymbolFunc 函数符号

3.2.6.1. symbols 函数形参

3.3. SymbolTable 符号表类

保存符号的容器。
`HashMap<String, Symbol>` 当识别到标识符的时候，如果是在定义阶段，则先检查符号表中有没有这个符号，如果有，则是重定义的错误，如果没有则将其添加到符号表中；
如果在使用阶段(LVal / UnaryExpFunc)，也要检查符号表中有没有这个符号，如果有则正常；如果没有则说明使用了未定义的符号，应该报错。

新建符号表：
MainFuncDef、StmtFor、Block、CompUnit中新建。

3.4. STStack 符号表栈类

保存符号表的容器
`ArrayList<SymbolTable>` 栈顶即为当前域的符号表。
每进入一个域，就压入一个新的符号表，并保证此后的新符号存在这个符号表中。
退出一个域，就将这个符号表弹出。

函数只能定义在全局

一个变量如果在声明过程中发生了错误，那么他将被视作未定义，不加入符号表。
除了重定义以外，无论函数定义是否成功，都加入符号表，也就是说在调用时不会出现未定义。
一个成功定义了参数个数和类型的函数，我们不因为它的形参在声明时重定义而影响它的正常调用；而如果函数没能成功定义参数个数和类型（具体表现为 ')' 缺失 ），那么它也会正常加入符号表，只不过我们无法判断它的实参的任何问题。

如果一个变量没有成功声明，将被视作未定义，不加入符号表。它的类型相关问题也无从谈起。

如果函数没能成功定义参数个数和类型（具体表现为 ')' 缺失 ），那么其参数个数、类型和返回值可能出现的问题均不做考虑。

如果函数成功定义了参数个数和类型，那么其参数个数、类型和返回值在出现问题的时候各自报错即可。