

Compiler Code 编译器设计文档

班级：212113

姓名：田乐

学号：21371362

参考编译器介绍

~~我tm纯手搓的，我参考谁去??~~

~~除了我还有什么傻逼会用JAVA来开发编译器??~~

编译器总体设计

文件组织

编译器的文件结构设计如下

- `Compiler` 类 编译器运行的入口，其中含有唯一的 `main` 方法
- `Input` 包 输入源代码相关
 - `InputSourceCode` 类 读入源代码
- `Output` 包 文件输出相关
 - `OutputIntoFile` 类 输出到 `output.txt` 或者 `error.txt`
- `Lexical` 包 词法分析相关
 - `LexicalAnalysis` 类 词法分析器
 - `ReserveWord` 类 保留字识别和处理
- `Syntactic` 包 语法分析相关
 - `SyntacticAnalysis` 类 语法分析器
 - `SyntacticComponents` 包 其中包含了文法中各个非终结符号的对应的处理的类
- `SymbolTable` 包 符号表相关

- `Result` 包 结果相关
 - `Error` 包 错误处理
 - `HandleError` 类 错误处理类
 - `AnalysisErrorType` 枚举 各个错误的枚举
 - `AnalysisResult` 枚举 统一返回值
- `Other` 包 其他
 - `ParamResult` 类 将多的返回值放置在参数中

在参数中的返回值

这里面有一个有点特殊的设计，也就是 `ParamResult` 类。其作用类似于C语言中的指针。众所周知，C语言中的指针可以在实际上实现一个函数有多个返回值。这对于复杂的大型系统的开发是大有好处的，虽然这个几千行的编译器也不算什么复杂的大型系统。

出于类似的目的，我设计了 `ParamResult` 类，通过其中的 `getValue` 和 `setValue` 方法，即可设置其 `value`，继而在实际上实现函数有多个返回值的功能

在后续开发中，这一设计被证明是极其富有远见的，彰显了开发者的智慧

我自夸一句没人反对吧嘿嘿嘿

统一方法返回值

众所周知，大一统是一种美

在实现了上面的参数中的返回值后，便可以实现另一件事情：**统一各个方法的返回值的意义及其表示。**

这件事情主要是由 `AnalysisResult` 枚举实现的，具体如下

```
public enum AnalysisResult {  
    SUCCESS,  
    FAIL,  
    END,  
}
```

总体结构

编译器运行的总体结构是这样的

1. 从 `Compiler` 类中的 `main` 方法开始
2. 调用 `InputSourceCode` 类中的 `readSourceCode` 方法，读入源代码
3. 通过 `SyntacticAnalysis.getInstance().run(false)`；执行语法分析。并在语法分析各个非终结符的时候调用 `LexicalAnalysis.getInstance().next()` 方法完成同步的词法分析

词法分析设计

词法分析主要在 `Lexical` 包中，包括如下文件

- `LexicalAnalysis.java`
- `ReserveWord.java`

其中 `LexicalAnalysis.java` 是词法分析器的实现类，采用单例模式设计，其中的方法是 `next` 方法，用以读取下一个单词，此外还有 `peek` 方法用以偷看一个单词和 `peekMany` 方法，用以偷看多个单词。

`ReserveWord.java` 是保留字相关类，其主要的方法是 `getReserveWord(String word)`，用以一个单词对应的保留字的类别码（如果不是的话返回 `NOT_RESERVE_WORD`）

语法分析设计

语法分析主要在 `Syntactic` 包中，包括如下文件

- `SyntacticAnalysis.java`
- `SyntacticComponents`
 - `ComponentValueType.java`
 - `SyntacticComponent.java`
 - 各个非终结符的处理类

其中 `SyntacticAnalysis.java` 是词法分析器的实现类，采用单例模式设计，其中的方法是 `run(boolean whetherOutput)` 方法，用以运行语法分析器（其中的 `whetherOutput` 代表是否输出语法分析结果）。这个方法后面调用文法的起始符号 `CompUnit` 对应的处理的类 `CompUnit.java` 中的 `analyze` 方法，从而启动语法分析。

`SyntacticComponent.java` 是所有非终结符的处理类的父类，在其中定义了所用非终结符处理中都要用到的一些东西，比如词法分析器实例，符号表实例和当前非终结符求出来的值的类型，还有 `analyze` 方法。

所谓“当前非终结符求出来的值的类型”，主要指的是 `ComponentValueType.java` 中定义的各种类型，具体如下

```
public enum ComponentValueType {  
    INT, // int  
    ONE_DIMENSION_ARRAY, // 一维数组  
    TWO_DIMENSION_ARRAY, // 二维数组,  
    VOID, // 空  
    NO_MEANING // 无意义  
}
```

其默认值为 `NO_MEANING`，也就是无意义。

对于大多数的非终结符，不存在所谓的“值的类型”一说，这里主要考虑的是函数参数类型和 `Exp` 等非终结符的问题。这一种设计是为了能成功地报出“函数实参类型不匹配”而设计的。

各个非终结符的处理类继承自 `SyntacticComponent` 类，主要重载了 `AnalysisResult analyze(boolean whetherOutput)` 方法，具体设计参考文法对照即可，在此不在赘述。

符号表设计

符号表的总体设计如下

- **主表 + 子表** 的模式
 - 主表：符号名，种类 `category`（变量、常量、函数），类型 `type`（int, 数组），到子表的连接，作用域ID
 - 函数子表
 - 数组子表
- 作用域（分程序）栈

错误处理设计

报错

在运行的过程中，一旦发现错误，则调用 `HandleError.handleError()` 方法，进行报错。这个方法接受一个 `AnalysisErrorType` 枚举类型的参数，`AnalysisErrorType` 枚举中规定了如下的错误类型，`handleError` 方法则实现按照题目要求输出报错信息

```

public enum AnalysisErrorType {
    UNEXPECTED_ERROR, // 预料之外的其他错误
    ILLEGAL_SYMBOL, // 非法符号
    NAME_REPEAT, // 名字重定义
    ASSIGN_TYPE_NOT_MATCH, // 赋值类型不匹配
    IDENTIFIER_NOT_DEFINE, // 标识符未定义
    ASSIGN_TO_CONST, // 改变常量的值
    ASSIGN_TO_FUNCTION, // 给函数赋值
    ARRAY_INDEX_OUT_OF_BOUND, // 数组下标越界
    VALUE_NOT_INITIALIZE, // 值未初始化
    ARRAY_DIMENSION_NOT_NEAT, // 数组维度不整齐, {{1,2}, 3}这样的
    ARRAY_DIMENSION_BEYOND_TWO, // 数组维度超过二
    LACK_OF_RBRACK, // 缺少右中括号
    LACK_OF_RPARENT, // 缺少右小括号
    NOT_FUNCTION, // 不是函数
    FUNCTION_PARAMS_NUMBER_NOT_MATCH, // 函数参数个数不匹配
    FUNCTION_PARAMS_TYPE_NOT_MATCH, // 函数参数类型不匹配
    VOID_FUNCTION_WITH_RETURN, // 无返回值的函数存在不匹配的return语句
    INT_FUNCTION_WITHOUT_RETURN, // 有返回值的函数缺少return语句
    LACK_OF_SEMICN, // 缺少分号
    PRINTF_NOT_MATCH, // printf中格式字符与表达式个数不匹配
    UNEXPECTED_BREAK_OR_CONTINUE, // 在非循环块中使用break和continue语句
    FORMAT_STRING_WITH_ILLEGAL_CHAR, // 格式字符串中有非法符号
    FORMAT_CHAR_NUMBER_NOT_MATCH, // printf中格式字符与表达式个数不匹配
}

```

局部化处理

错误局部化出于这样的一种考虑，**一行中错误仅仅会有一种**，和**所有错误都不会出现恶性换行的情况**。那么我就可以在读取到某一行中的错误的时候，跳过这一行，直接进入下一行的处理。

涉及到的文法如下

```

ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
VarDecl → BType VarDef { ',' VarDef } ';'
Stmt → LVal '=' Exp ';'
      | [Exp] ';'
      | Block
      | 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
      | 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
      | 'break' ';'
      | 'continue' ';'
      | 'return' [Exp] ';'
      | LVal '=' 'getint' '(' ')' ';'
      | 'printf' '(' 'FormatString' {',' Exp} ')' ';'
Block → '{' { BlockItem } '}'

```

那么解决思路就是

出错的语法成分不断向上报错，直到报错到上面的四条之一，随后在上面的四条中实现直接进入下一行。

除此以外还有一个特殊的文法的错误处理

```
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
```

出于对**所有错误都不会出现恶意换行的情况**的信任，我们决定在 `Ident` 或者其他 `Block` 之前的部分出现错误的时候，在报错后直接进入 `Block` 块处理。