**编译原理第二次次程序实验报告**

**实验目的**

在第一个实验（以一个字符串的形式读入一个文件内容，然后输出符号对应的类型和种别码）的基础上，分析输入的串的语法结构，判断这个串的语法是否正确。

**完成的模块和模块流程**

运行环境：

运行在JDK 10上 使用了Kotlin编写的代码，调试环境是Jet Brains Intelli J

1. 完成内容

（1）从一个文件中读入一串代码进行分析，然后通过先前已经规定的种别码和字符的对应表进行选择，最后输出放入到一个返回的数组中。

（2）将这个数组中的每一个token放入函数中进行解析，如果是错误的语法串，则抛出异常 结束程序。如果是正确的代码的话则输出 正确，如果是不符合语法的token串的话则指出在哪一个index有错误。

1. 实验流程
2. 编写parse函数打开输入流，读取对应路径下面的文件中的字符代码，通过Java split函数进行简单分段，分段的过程中，对于tab，换行符号还有空格都要分割开，最后再顺序扫描进行处理，最后将token和种别码分别放到list 中输出。

（2）将上一步parse函数最后的结果读取，根据教材中提供的结构编写函数接受输入。有些语法需要尝试输入，如果输入错误的话，退回之前的下标，尝试别的输入。

**关键技术和难点**

1. 语法分析程序如何制定一个框架能够接受正确的输入

在递归下降语法分析器中需要确定一个框架，才可以比较正确对token串进行处理。

实验中采取的框架是书上的框架。对于每一个公式都写出一个函数，这些函数还有公式之间相互调用，公式自己也可能会递归调用自己。

但是使用框架的时候需要注意调用顺序，比如语句中需要先检查while 然后是if 和 赋值语句。所以会先调用while if 语句块相关的判断函数。在发现while 和 if不正确的时候需要退回到之前的下标，而且在这个时候，有错误的语句也不应该抛出异常，这样会结束程序向后查找。

1. 递归调用的程序遇到异常的时候如何跳出并且返回
2. 如何处理一次或者多次的表达式，如何返回正确的下标

递归调用的程序只有两个种类

项 = <因子>{\*<因子>|/<因子>}

表达式 = <项>{+<因子>|-<因子>}

使用递归调用的目的是为解决语法中的一次和多次输入的问题。这个时候可以先向后多读取一个值但是不改变下标，如果是相关的运算符的话可以直接递归调用。

或者对一个表达式使用逆波兰式计算这个表达式能否产生正确的结果，否则则不是正确的表达式。

1. 如何正确的结束程序，不抛出错误的异常

我的做法是在每一个函数中增加一个判断的代码，当index对应的值是函数的结束符号即end的时候，抛出一个完成的异常，可以就算是在递归栈中的函数也可以马上结束

**编程调试程序的难点**

1. 递归调用程序不容易调试：

因为是一个递归调用的函数，栈顶的下一个值可以通过intelliJ 中的查看frame 来找到下一个值。并且需要注意编写递归程序的跳出condition，否则JVM会抛出StackOverflow异常

2.下标控制问题

下标是一个全局变量，需要控制去“尝试”输入的代码在遇到错误的过程中需要返回，并且剪掉多输入的文字

1. 有时候会漏掉一些case，需要反复修正才可以

核心代码：

import java.lang.Exception

//简单的词法分析程序

lateinit var parseResult :ArrayList<Pair<String,Int>>

var index = 0

//程序块

fun program():Boolean{

val begin = parseResult[index++].first

if(begin != "begin")

throw AnalyzerException("$index 需要使用begin 语句开头")

val main = parseResult[index++].first

if(main!="main")

throw AnalyzerException("$index 需要使用main语句作为函数体")

return blockNode()

}

//语句串分析

fun sentenceString():Boolean{

//重复一次到多次！

val firstSentence = sentence()

//检查后面的是不是分号

val back = parseResult[index++].first

return if(firstSentence && back == ";") {

sentenceString()

}else{

throw AnalyzerException("表达式没有以分号结尾")

}

}

//语句分析

fun sentence():Boolean{

//有可能是一个没有while if语句的串

//因为是或者关系 需要加一个mute 不让报错停止程序

//通过try-catch 方式！

//扫描一下现在的start是什么

val start = parseResult[index]

return when(start.first){

"while"->{

index++

val item = parseResult[index]

if(item.first == "end")

throw AnalyzerFinisheException("扫描完成,正常结束")

whileNode()

}

"if"->{

index++

val item = parseResult[index]

if(item.first == "end")

throw AnalyzerFinisheException("扫描完成,正常结束")

ifNode()

}

//其他情况就是assign

else->{

//todo buggy

index++

//检查完这个符号之后要++

val item = parseResult[index++]

if(item.first == "end")

throw AnalyzerFinisheException("扫描完成,正常结束")

assignNode()

}

}

}

//语句块分析

fun blockNode():Boolean{

val left = parseResult[index++].first

val sentenceString = sentenceString()

if(!sentenceString)

return false

val right = parseResult[index++].first

if(left != "{" || right != "}")

throw AnalyzerException("$index 缺少结束符号{ 或者 }")

return true

}

//while node

fun whileNode():Boolean{

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val condition = conditionNode()

val blockNode = blockNode()

return true

}

//todo

//赋值语句

fun assignNode():Boolean{

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val rightValue = expressionNode()

//最后应该以分号结尾

return rightValue

}

//if 语句分析

fun ifNode():Boolean{

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val condition = conditionNode()

val block = blockNode()

return false

}

//condition node语句分析

//并且会分析condition 左右两侧的括号

fun conditionNode():Boolean{

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val leftColumn = parseResult[index++].first

if(leftColumn !="(")

throw AnalyzerException("$index 缺少左括号 (")

val leftValue = expressionNode()

val comparator = when(parseResult[index].first){

">=" -> true

"<=" -> true

"==" -> true

else -> false

}

if(!comparator)

throw AnalyzerException("$index 缺少比较符号")

index++

expressionNode()

val rightColum = parseResult[index].first

index ++

if(rightColum!=")")

throw AnalyzerException("$index 缺少右括号")

return true

}

//表达式分析

/\*\*\*

\* return true表示扫描完成一个或者多个项

\*/

fun expressionNode():Boolean{

//如果第一符号不是项

//这个时候将所有的<因子>部分消耗完成 return true表示<？> +|- <?>中？表示的是一个项

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val first = itemNode()

if(first){

val temp = index

//不一定需要这样的消耗

val append = { pair: Pair<String, Int> ->

if (pair.first == "+" || pair.first == "-") {

index++

}

}

val item = parseResult[index]

//消耗

append(item)

if(index > temp){

itemNode()

}

}

return first

}

//项分析

/\*\*\*

\* 返回true表示满足一个或者多个因子 （多个因子相乘或者相除）

\*/

fun itemNode():Boolean{

//项的组成：<因子>{\* <因子> | / 因子}

//如果index指向的部分是一个因子

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val first = factorNode()

if(first) {

//如果是真的 说明可能还要继续分析

//如果可以继续分析

val temp = index

//超前扫描 扫面后面一个运算符号

//附加结构消耗完成\*<因子> | /<因子>的部分

val append = { pair: Pair<String, Int>->

if (pair.first == "\*" || pair.first == "/") {

index++

}

}

val item = parseResult[index]

//扫描后面一个运算符

append(item)

if(index > temp){

//后面可能是一个项

factorNode()

}

//else 在else的情况下index 不会增加

}

return first

}

//最底层

//因子分析

/\*\*\*

\* return true表示这个是一个数字或者是标示符

\* 或者是一个表达式 在表达式中进行判断

\*/

fun factorNode( ):Boolean{

if(parseResult[index].first == "end") throw AnalyzerFinisheException("分析完成 正常结束")

val id = parseResult[index].second

return if(id == 11 || id == 10 ) {

index ++

true

}else if ( parseResult[index].first == "==" || parseResult[index].first == ">=" || parseResult[index].first

== "<="){

throw AnalyzerException("表达式语法错误")

} else{

expressionNode()

}

}

class AnalyzerException(msg:String):Exception(msg)

class AnalyzerFinisheException(msg:String):Exception(msg)

fun main(args:Array<String>) {

val source = file2Code("/home/kolibreath/githubProject/complier/src/TestCase.txt")

parseResult = parse(sourceString = source)

try {

println(program())

} catch (e: AnalyzerFinisheException) {

println(e.message)

} catch (e: AnalyzerException) {

println(e.message)

}

}