《编译原理》实验使用说明

项目地址：https://github.com/bajdcc/jMiniLang
参考博客：http://www.cppblog.com/vczh/
项目编码：*UTF-8*
获取途径：*git clone https://github.com/bajdcc/jMiniLang.git*
联系方式：mailto:bajdcc@foxmail.com

目录

底层模块

正则表达式

该工具实现了以下功能：

- 解析正则表达式
- 字符区间合并
- 生成表达式树
- 生成 NFA
- NFA 消除 Epsilon 边
- NFA 确定化转 DFA
- DFA 最小化（部分优化）
- 生成状态转移矩阵

从而完成字符串的匹配工作。

类：util.lexer.regex.Regex

正则表达式分析样例：util.lexer.test.TestRegex.java

DEBUG 输出内容：该工具实现过程中的大部分步骤带有输出。

参考与拓展阅读：http://www.cppblog.com/vczh/archive/2008/05/22/50763.html

词法分析工具

该工具可以识别：

- 空白字符
- 注释
- 宏
- 字符串
- 字符
- 标识符
- 关键字
- 操作符
- 整数
- 实数

从而完成单词的匹配工作。

类：util.lexer.Lexer

词法分析样例：util.lexer.test.TestLexer.java

DEBUG 输出内容：该工具实现过程中的大部分步骤带有输出。

参考与拓展阅读：http://www.cppblog.com/vczh/archive/2008/05/22/50763.html

LL1 文法

输入参数：
- 文法要匹配的目标字符串
- 终结符声明和定义
- 非终结符声明和定义
- 设置 Epsilon 名称
- 产生式
- 初始符号

产生式格式：
- 单词（即 C 语言中变量名命名）表示非终结符
- @开头的单词为终结符
- @Epsilon 名称表示 Epsilon
- “|”表示产生式并列

类：LL1.grammar.Grammar

LL1 分析样例：LL1.grammar.test.TestGrammar.java

算符优先文法

输入参数和产生式格式同 LL1 文法。

添加了归约处理方案，即语义动作。

当从栈中找到最左素短语 A，应立即归约，此时，A 中包括终结符 T 和非终结符 U。

设定模式：以 1 代表 T，代 0 代表 U。

例：对括号进行归约，形如(exp)，则两边的括号属于 T，当中的 exp 属于 U，该模式为 101。

“模式”是判定当前最左素短语采用哪个语义动作进行归约的依据。

类：OP.grammar.Grammar

OP 分析样例：OP.grammar.test.TestGrammar.java

基于 LALR1 的简易解释器

语法分析

输入参数：
- 定义终结符
- 定义非终结符
- 定义语义错误处理动作
- 定义归约动作（语义动作）
- 定义终结符匹配动作（终结符通过）

初始化：

- 分析产生式
- 生成语义动作指令

产生式格式：

- 禁止产生式产生空串
- 单词（即 C 语言中变量名命名）表示非终结符
- @开头的单词为终结符
- 非终结符或终结符后跟“[”+数字+“]”为归约时的数据索引（归约需要取数据）
- 非终结符或终结符后跟“{”+数字+“}”为定义语义错误处理器（这条路走不通就报错）
- 非终结符或终结符后跟“#”+数字+“#”为单词通过动作（将匹配该单词前的动作）
- “|”表示产生式并列
- “[”和“]”包括的内容为可选

类：LALR1.grammar.Grammar

LALR1 语法分析样例：LALR1.grammar.test.TestGrammar.java

语义分析

输入参数：

- 语法分析的所有输入
- 语法分析产生的语义动作指令

分析阶段：

- 执行语义动作指令
- 进行归约
- 进行错误处理
- 进行终结符通过处理（终结符通过前做的动作）
- 非终结符或终结符后跟“{”+数字+“}”为定义语义错误处理器（这条路走不通就报错）
- 非终结符或终结符后跟“#”+数字+“#”为单词通过过滤动作（将匹配该单词前的动作）
- “|”表示产生式并列
- “[”和“]”包括的内容为可选

类：LALR1.semantic.Semantic

LALR1 语义分析样例：LALR1.semantic.TestSemantic.java

生成语法树

表达式种类：

- 赋值表达式
- 一元表达式
- 二元表达式
- 三元表达式
- 函数调用表达式
- 值表达式

语句种类：

- 块语句
- 表达式语句

- 循环语句 for
- 迭代语句 foreach
- 条件语句 if else
- 导入导出语句 import export
- 返回语句 return yield

其他：

- 块
- 函数

包：LALR1.grammar.tree

指令集

设计为基于栈的指令集。
代码带有注释，部分指令为满足新的语言特性（如协程指令 Yield 等）所添加。
参照：LALR1.grammar.runtime.RuntimeInst

生成代码

遍历语法树生成代码以及其他数据，包括：

- 数据段（基本数据类型）
- 代码段（虚拟机指令）
- 附加调试段（用于导入导出、扩展）

参照：LALR1.grammar.runtime.RuntimeCodePage

构建虚拟机

虚拟机组成：

- 指令页
- 导入表（import）
- 协程栈
- 运行时栈

参照：LALR1.grammar.runtime.RuntimeMachine
运行时栈组成：

- 数据栈
- 调用栈

参照：LALR1.grammar.runtime.RuntimeStack

语法

- 初始化语句

var 变量名 = 表达式;

- 赋值语句

let 变量名 = 表达式;

- 函数定义

var foo = func ~() -> 表达式;

var foo = func bar() { 语句 };

- 协程定义：

var foo = yield ~() -> 表达式;

var foo = yield bar() { 语句 或 yield 返回 };

- 函数调用

call 函数名([参数,参数…]);

- For 循环

for (var i = 0; i < 100; i++) { 语句 }

- Foreach 循环

foreach (var i : call 协程名([参数,参数…])) { 语句 }

- If 语句

if (表达式) {} else {}

if (表达式) {} else if (表达式) {}

- 导入导出

import “代码页名”;

export “函数名”;

语法设计参照语言：

- C# -> yield break, yield return, foreach
- Javascript -> var, var a = func() {}
- VB -> call
- Matlab -> ~表示匿名
- Java -> for (var i : collection) {} 中的冒号, import
- Lisp -> Lambda 函数
- Bash -> export
- Haskell -> let

高级语法说明：

- 所有以“g_”开头的函数调用和取值都是**外部调用**，为 export 声明或代码页扩展
- 传递**函数指针**可以直接传递函数名（本地）或传递函数名字符串（外部）
- 支持**递归**，但未做优化

- 支持**协程**（详情请自行搜索），即单线程切换
- 支持 **Lambda 函数**调用
- 实现**函数闭包**，即动态返回 Lambda 函数

参照样例：LALR1.interpret.test.TestInterpret.java

类型系统

本工程为解释器，实现类型为弱类型，采用 Java 自带的运行时类型检查。
由基本数据类型赋值是，为值传递；变量赋值时，为引用传递。
强制引用传递可用 g_new 函数；判断为空采用 g_is_null 函数；空值为 g_null。

文档

导出函数或扩展函数可书写文档，参见基本库。
动态获取文档可用 g_doc 函数，参数为函数名。

扩展

扩展分为代码扩展和扩展回调。
代码扩展：以代码形式进行扩展，表示为基本库、数据结构、算法。
扩展回调：分为值回调以及方法回调，两者都为函数回调。
值回调仅返回值；函数回调可返回值，并执行外部过程。
参照包：LALR1.interpret.module