LAPS Development Manual

Geoffrey Casper

2020

**Introduction**

LAPS stands for LAnguage Processor & Synthesizer. It can be referred to as the LAPS system, the LAPS library, or, simply LAPS. The purpose of this software is to be an accessible, student-friendly alternative to a standard compiler compiler, such as PLCC or YACC. To avoid redundancy, this document assumes the reader knows what regular expression are, as well as, context-free grammars.

**Getting Started with LAPS**

LAPS takes advantage of annotations, reflection, and class loading in Java to enable developers to write their own Java classes which can be interpreted as a language specification. To define a language, there needs to be a set of tokens (lexemes) defined using regular expressions for lexical analysis, a context-free grammar in terms of those tokens for syntactic analysis, and an executable portion to process the abstract syntax tree (AST) during semantic analysis.

**First Steps**

The first thing that needs to take place is annotating a Java class with @GrammarRule to tell the LAPS system that the class is indeed intended to contain a definition for the context-free grammar. Once that is done, public constructors will now represent acceptable sequences of tokens and other grammar rules. In addition, anything annotated with @Token LAPS will attempt to interpret as an uncompiled Java regular expression pattern. These token types can be accessed from your constructor grammar rule definitions by accepting a String type with the name of the field or method which defined them.

See Calculator.java for a simple example.

See the Annotations section for more information on how tokens types are interpreted by LAPS.

**Goals**

**Clear and Well-Documented**

This software is free and open source, so we encourage contributions and constructive criticism. To aid this mentality, clear and well-documented code is a must. This includes the use of meaningful function and variable names, JavaDoc compliant comments, and easy to follow code.

**Modular**

To allow this software to potentially serve more than its original purpose, modularity is a necessity. This means files should be meaningfully separated with minimal dependencies. An added benefit of this goal is that additional features are much less likely to break current features.

**Debugging and IDE Support**

Most compiler compilers don’t provide debugging support for their user’s source code making errors much harder to find and fix. This is because these compilers use customized, or even new, languages that have a small following. So, it’s a requirement to use a common language as a front-end for language developers.

**Design**

LAPS is organized into packages which can be used independently of one and other. Below are the explanations for each file and sub-package in the edu.rit.gec8773.laps package.

**Annotations**

This package contains all the annotations used during reflection of all the classes associated with a LAPS language specification.

@GrammarRule:

Types annotated with this give confirmation to the parser that it is intended to be a grammatical rule in the defined language. This should only be applied on classes in the current version.

@Token(skip=false):

This can be applied on both fields and methods in any @GrammarRule type, which are ***public*** and ***static***, to mark them as definitions for token types in the language specification. The names of the token types are the case-insensitive names of the fields and/or methods.

Fields are parsed as generic objects. Methods, on the other hand, are executed and the return generic objects. Once these objects are collected, the objects’ toString() methods are invoked one time before the language’s runtime and stored as the regular expression for tokens.

This annotation has a field called skip. Set this equal to true to mark the token type as text to be removed from the language’s input source.

@RunBeforeFirstInit:

This can only be applied on ***public*** and ***static*** methods with no parameters and a void return type in @GrammarRule types (classes). Methods found with this annotation will be invoked once **before** the **first** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

Example use cases:

* Initialize the global environment for a language.
* Print out an introduction for a language.
* Initialize any variable a language might need.

@RunBeforeEachInit:

This can only be applied on ***public*** and ***static*** methods with no parameters and a void return type in @GrammarRule types (classes). Methods found with this annotation will be invoked once **before** **each** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

Example use cases:

* Update a variable, i.e. number of instances created.
* Check if a variable should be reset or cleared like a List.

@RunAfterEachInit:

This can only be applied on ***public*** **instance** methods with no parameters and a void return type in @GrammarRule types (classes). Methods found with this annotation will be invoked once **after** **each** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

Example use cases:

* Reorganize the current state of the abstract syntax tree.
* Interpret a language.
* Evaluate an expression.
* Confirm the abstract syntax tree is what you expect it to be.

@Priority(value=0):

This can be applied to constructors in @GrammarRule types. The point of this is to distinguish between the ambiguity of which grammatical rule to attempt to parse first. So, all constructors take advantage of this priority system, even when not explicitly annotated. The way it works is the smaller (more negative) the value, the earlier the parser attempts to parse the corresponding constructor’s grammatical rule. If not annotated with this, default value is 0.