Five Languages Symposium: Rascal

Introduction

Rascal is programming language for source code analysis and transformation. The primary application areas are (legacy) system renovation, reverse engineering and reengineering, and the implementation of domain specific languages (DSLs). DSLs are languages tailored to a specific application domain. Examples include SQL, Excel, Make, LaTeX, VHDL, etc. Today we are going to use Rascal to implement a small language for statemachines. The example is derived from Martin Fowler's book on Domain Specific Languages. The relevant chapter is published online:

http://www.informit.com/articles/article.aspx?p=1592379

State machines are useful for describing state dependent behaviour, for instance to control machines or workflow engines. DSLs are a particular form of model-driven development (MDE), where software is specified using high-level models, from which then the implementation is generated.

In the tutorial we will explore the Rascal language and environment by implementing the following facets of the DSL:

- A context-free grammar to describe the syntax of state machines
- An algebraic data type (ADT) for describing state machine abstract syntax trees (ASTs)
- A consistency checker for state machines. This component, for instance, highlights use of undefined states or events, marks duplicate states, commands or events and detects unreachable states.
- Reset events (see the link above) are syntactic sugar: they can be *desugared* into an equivalent statemachine that does not use them.
- A graphic visualization of a state machine as a graph.
- A code generator that produces a Java code consuming and producing tokens. (This is called a Model-to-Text transformation)
- A transformation that takes two state machines and produces a new state machine that runs the two original machines in parallel. (This is called a Model-to-Model transformation). The resulting state machine can be input to the original code generator and visualizer.
- A simple evaluator to simulate a state machine. This can be connected to the visualization to interactively step through a state machine.

- A simple source-to-source transformation to implement a rename refactoring for states, events and/or commands.
- Provide domain-specific IDE features for state machines: context-menus to invoke the code generator, outline views, folding, error marking.

Some of these assignments are more complicated than others. This is not a problem: we will see how far we get!

To get up and running, download the following zipfile:

> http://www.cwi.nl/~storm/5lang-rascal.zip

Unzip it into a dedicated directory. The Zip file contains pre-built Eclipse workspace including a Rascal project for state machines. The project contains some setup code and example state machines.

Warming Up

Syntax Definition for State Machines

Rascal has builtin support for context-free grammars which can be used to define the syntax of programming languages. A syntax rule consists of the following parts:

```
syntax NonTerminal = label_1: Element_1i ... Element_1m1
                   | ...
                   | label_n: Element ... Element_nmn
```

This defines a rule named "NonTerminal" with n alternatives. Each alternative has a label and a sequence of Elements. The elements of an alternative define the syntax to be recognized by this rule. An element can be one of the following symbols:

- "a literal"
- ANonTerminal
- a regular symbol: X? for optional, X* for zero-or-more, X+ for one-or-more, and the separated list operators: {X "sep"}*, and {X "sep"}+. X can be any symbol, but typically will be a non-terminal

To define lexical rules (e.g., for identifiers) character classes are used (similar to regular expressions):

- char class [a-z]: recognize a character between a and z. Or: [\t\n\r\ ]: recognize a whitespace character
- ![a-z]: recognize any non-lowercase-alphabetic character
- ?, *, and + can be used on character classes as well

To create a grammar for state machines, create a new Rascal module, and add rules to recognizes the syntax invented by Martin Fowler (page 3 of the article cited above).

To get up and running add the following definitions for layout (whitespace and comments) and identifiers:

```
syntax Id = lex [a-zA-Z][a-zA-Z0-9_]* - Reserved # [a-zA-Z0-9_];
syntax Reserved = "events" | "end" | "resetEvents" | "state" | "actions" ;
syntax LAYOUT = lex whitespace: [\t-\n\r\ ]  | lex Comment ;
layout LAYOUTLIST = LAYOUT* # [\t-\n\r\ ] # "/*" ;
syntax Comment = lex @category="Comment"  "/*" CommentChar* "*/" ;
syntax CommentChar = lex ![*] | lex Asterisk ;
syntax Asterisk = lex [*] # [/] ;
```

Quiz: what is the meaning of the # and - constructs?

Don't forget to add a start syntax rule. This will instruct the parser what to recognize when a file is parsed in an editor. For instance, like this:

```
start syntax Controller = controller: Events ResetEvents? Commands? State+ states;
```

After you've created the syntax module. Create a module Plugin.rsc in the src directory of the project. Add the following lines to make the syntax available to the environment:

Open up a Rascal console, enter "import Plugin;", then "main();". If all is well, you can now open ".ctl" files (see the input directory of the project) and get syntax highlighting.

```
module Plugin

import <you syntax module>;
import util::IDE;
import ParseTree;

public void main() {
  registerLanguage("Controller", "ctl:, <Your start symbol>(str input, loc org) {
     return parse(#<Your start symbol>, input, org);
  });
}
```