Hack Your Language with Rascal

- Tijs van der Storm: [@tvdstorm](https://twitter.com/tvdstorm), storm@cwi.nl
- Jouke Stoel [@jstoel](https://twitter.com/jstoel), jouke.stoel@cwi.nl

Preliminaries

The Github repo we'll use during this hands-on workshop is:

- https://github.com/cwi-swat/hack-your-javascript

Instructions on how to setup Eclipse and Rascal can be found in the repository in `doc/joc-prerequisites.pdf`.

Interactive documentation on Rascal can be found online at http://tutor.rascal-mpl.org. This can also be started from within Eclipse from the Rascal menu under "Show Tutor".

Check out the `src/demo` directory to see examples of simple and more advanced language extensions.

Desugaring in Rascal

We're going write "desugarings", which are source-to-source transformations that compile/transpile/rewrite Javascript language extensions ("syntactic sugar") to the base Javascript language (ECMAScript 5). The project mentioned above contains the basic desugaring infrastructure. The only thing you have to do is to extensions of the main `desugar` function. The framework will call all of them that are in the project.

The `desugar` function is extended by writing a case for the extension you want to desugar using concrete syntax matching. Let's dissect an example to see what that means:

```
Statement desugar((Statement)`debug <Expression s>;`) {
  return (Statement)`if (DEBUG_FLAG) console.log(<Expression s>);`;
}
```

This Rascal function definition *matches* on the `debug` statement using concrete syntax. This means that the pattern is written using the language you are actually defining (in this case Javascript + `debug`). The pattern in this case is:

```
(Statement)`debug <Expression s>;`
```

The first part in parentheses indicates the type of the values this pattern matches, – in this case values of type `Statement`. The second part, enclosed in backticks (‘) is the actual pattern: first the keyword `debug` and then some `String`. The string part (enclosed in < and >) represents a typed *hole*, which will match anything of type `Expression`. If the match is successful the meta variable `s` is bound to the matched sub-value.

The pattern used in the `return` statement is not used for matching, but for *construction*. In this case, the bound value of the meta variable `s` is inserted into the argument of `console.log`.

Some notes:

- `desugar` cases need to return the type they are consuming. For instance, if you desugar a `Statement` the return type should be `Statement`.
- For simple desugarings as the one above, there is a short-hand notation:

```
Statement desugar((Statement)`debug <Expression s>;`)
   = (Statement)`if (DEBUG_FLAG) console.log(<Expression s>);`;
```

- Concrete syntax matching works "modulo layout". This means that the patterns will match source terms regardless of the whitespace and/or comments used in either the pattern or the source term. For instance, the desugaring above will match statements like `debug /* a comment */ "debug!"`, or statements with newlines or extra spacing in them.
- As of yet, holes used in construction patterns (e.g., the returned value of the `debug` desugaring) only admit interpolation of variables. If you want to put in complex expressions, first make a variable and assign the complex expression to it.
- Patterns should comply to the grammar. If you make a mistake, you'll get a parse error in your Rascal program!
- If you need to use literal `<`, `>` or ` in patterns, escape them using `\` (backslash).
- If you need multiple lines start every line except the first with `'` (single quote). For instance, as follows:

```
Statement desugar((Statement)`debug <Expression s>;`)
  = (Statement)`if (DEBUG_FLAG)
               '  console.log(<Expression s>);`;
```

Notes on the Javascript grammar

Grammars define first-class data types in Rascal. A production rule `syntax S = p_1 | ... | p_n` introduces a (non-terminal) type `S` with data (syntax) constructors `p_1`...`p_n`. You can check out the Javascript grammar in `src/javascript/Syntax.rsc`.

The Javascript grammar used in the project closely follows the ECMAScript 5 syntax, except that the `;` is required after statement expressions, and there is no comma-expression.

While writing your desugarings, keep in mind that:

- Expressions are captured by the `Expression` type
- Statements are captured by the `Statement` type
- String literals are captured by the `String` type
- Identifiers (variables, field names, etc.) are captured by the `Id` type.

For most assignments we provide the necessary syntax extensions up front, so that you can focus on the source-to-source transformations.

Executing desugarings

There are two things required to execute a desugaring. After you made the change to the desugaring you should:

1. Click on the Language Refresh button (it's the button that has a black 'recycle' icon on it)
2. Save the corresponding SJS file.

Desugarings are automatically executed when an SJS file is saved. There is one catch though; the SJS files are not marked 'dirty' automatically when changing a desugaring and saving an unchanged SJS file won't have any effect. The solution is to simply add and delete an space (or something similar) to the SJS file and then save it.

After a save of the SJS file the corresponding JS and HTML file should be updated. To check the result of your changes open up the JS file to see the generated Javascript. The HTML file shows the SJS content, the desugared JS content and executes the desugared JS source. Note: you might have to reload the HTML.

NB: The JS and HTML files in the `sjs/` directory can always be deleted since they get regenerated on save.

NB2: For successful generation of the SJS files should be one level below the `sjs` directory; i.e., in `sjs/series1` or `sjs/your_own_dir`.

Exercises

Every exercise has a corresponding test that shows you what the desugared JS variant should look like. To run the test you should:

1. Start a Rascal Console by right-clicking on a Rascal file in the editor (i.e. on the opened `Series1.rsc` file) and selecting the `Run as -> Rascal Application` from the pop-up menu.
2. By entering the `:test` command in the console and hitting enter all the tests in scope will be run.

A successful test will light up green, a failed one will get a red squiggly line under it. Hovering over the failed test in your editor will show you the reason of failure.

Next to the test every exercise has a corresponding `SJS` file. These can be found in the `sjs/` directory. The name of the SJS file corresponds with the exercise you are working on. For instance, the SJS file needed for exercise 1 is `sjs/series1/ex1_atField.sjs`.

In the exercises we use *upper-case* identifiers in snippets to indicate *meta-variables*. Lower-case identifiers either represent keywords (e.g. `unless`) or object-language identifiers (e.g. `this`, `console`).

Series 1: basic desugaring

1 At Fields In Ruby instance variables (fields) are prefixed with an `@`-sign. Write a desugaring that transforms `@X` (where `X` can be an `Id`) to `this.X`.

Remember: in the above description upper-case identifiers are *meta-variables*. For instance `@X` 'captures' the concrete SJS code `@myVar` or `@x` or `@anotherVariable`, etc. See the corresponding test for an example.

2 Twitter Search In this exercise we will implement a very simple Twitter DSL which lets you search for tweets **@somebody** or **#someHashTag**. In SJS Twitter searches are first-class citizens. You can write `@(Expression)` or `@(Expression1, Expression2)` to search for tweets to certain persons or `#(Expression)` or `#(Expression1, Expression2)` to search for tweets containing certain hashtags.

This 'DSL' desugars to a simple JS library. `@("somebody")` desugars to `searchAt("somebody")`, or using meta variables, `@({Expression ","} *)` desugars to `searchAt({Expression ","}*)`. Similar; `#("someHastag")` desugars to `searchHash("someHashtag")`, or again using meta variables, `#({Expression ","}*)` debuggers to `searchHash({Expression ","}*)`. `searchAt(..)` and `searchHash(..)` are JS functions that are already defined in our small Twitter JS client.

With the `{Expression ","}*` syntax you capture zero or more `Expressions` which are separated by a `,`.

Since searching Twitter is an asynchronous action the JS methods `searchAt()` and `searchHash()` return *Promises*. This makes it easy to display the result once the search finishes. You don't need to implement any of this, this is already done. Take a look corresponding SJS file (`sjs/series1/ex2_twitter.sjs`). Once your desugarring is correct this SJS will be transformed to JS that can be executed. Opening the generated HTML file should show you the result of live Twitter searches!

3 Dont statement The `dont` statement can be seen as a code comment. It just means, don't execute this statemenet.

Write a desugaring for the "dont" statement with syntax `dont Statement`. It should desugar to code where the argument statement is eliminated. For instance, `dont S` would rewrite to the empty statement `;`.

4 Todo statement Comments are often used to mark todo items in code. But why not use an explicit statement that nags your by writing the todo item to the `console`? In this case the desugaring transforms `todo X;` (where X represents a `String`) to `console.log("TODO: " + X);`.

5 Unless statement Some languages include a statement for negated conditional. For instance, Ruby has `unless`. In this assignment we're adding such a statement to Javascript. The syntax is `unless "(" Cond ")" Body` (where `Cond` is an `Expression` and `Body` a `Statement`), and it should rewrite to `if (!(Cond)) Body`.

Quiz: why are the extra parentheses around `cond` needed?

6 Repeat-until Write a desugaring for `repeat Body until "(" Cond ")"` which transforms to a `do-while` loop.

7 Assert statement Assert statements are used to document your assumptions. If an assertion fails you get an exception listing showing the expression that failed and (optionally) a textual message. The `assert` we're defining here has the following syntax: `assert Expression: Message;` (where `Message` is a `String`). It should be translated to code that throws an exception if the expression evaluates to a falsy value. For instance, `assert E: S` desugars to: `if (!(E)) throw new Error("Assertion failed: " + msg);`

Use Rascal string interpolation (using < and >) to *unparse* the argument expression into a Rascal string, and then parse it as a Javascript string literal (`String`) as follows: `msg = parse(#String, "\"<e>\"")` (assuming `e` is the expression). Now you can use `msg` in the constructed pattern.

Series 2: introducing bindings

Names are an important language feature. They allow us to create abstractions, store intermediate values for reuse, and create sharing in values. In this series, we'll define language extensions that require the introduction of name bindings.

1 Swap A swap statement allows you to swap the value of two variables in a single statement. Its syntax is `swap X, Y;`, where `X` and `Y` represent variables. Write a transormation which transforms this to:

```
(function() { var tmp = X; X = Y; Y = tmp; })();
```

Check that our desugaring framework avoids inadvertent name capture by trying swapping a variable `tmp` with another variable.

Quiz: why do we need the Immediately Invoked Function Expression (IIFE)?

2 Test Write a desugaring for a `test` statement, similar to the `assert` desugaring above. In this case the syntax could be: `test Expression should be Expression;`. Note that `should` and `be` are two keywords! Instead of throwing an exception it evaluates both expressions, tests if they are equal, and prints out a message with expected and actual value if the test failed.

Tip: pass the two expressions as parameters to an IFFE, like so `(function (actual, expected) { ... })(E1, E2)`.

3 Foreach Javascript has a `for (x in array) ...` statement, but its semantics are not always wat you expect. In this assignment, we'll add an explicit `foreach` construct that works intuitively on arrays. The syntax is `foreach (X in E) S`, where `X` is an identifier, `E` an expression and `S` a statement. The `foreach` statement should be desugared to something like:

```
(function(array){ for (var i = 0; i < array.length; i++) { var X = array[i]; S } })(E);
```

Quiz: why do we need to bind the expression `E` to the parameter (`array`) first?

Quiz: The use of the IFFE to create a scope for `i`, `X` and `array` is, in fact, wrong: not all `foreach`-loops will desugar to correct code. Can you think of the reason? Hint: it has to do with non-local control flow.

4 Arrow functions ECMACScript 6 introduced fat arrow functions. Arrow functions are a shorthand notation for anonymous functions (closures). In their simplest form the syntax is `X => E`, where `X` is an identifier and `E` some expression.

At first sight the desugaring seems to be simple, just map it to `function(X) { return E; }`. Unfortunately, functions in Javascript introduce a new context for the `this` variable but arrow functions are explicitly designed not to do this; instead the value of `this` in the body expression of an arrow function is lexically determined. IOW: it is inherited from its lexically enclosing function.

A solution is to introduce a special variable (e.g., `_this`) in the scope of the arrow function and replace all uses of `this` by `_this` in its body. This gives us something like:

```
(function (_this) { return function (X) { return E'; }; })(this)
```

In this snippet, the variable `E'` is the original `E` with all occurrences of `this` replaced by `_this`. You can use the provided helper function `replaceThis` to realize this.

Tip: see what happens if you desugar the arrow function `_this => _this + this.x`.

Optional: Add an extension for the second kind of arrow functions which accept any number of arguments enclosed in parentheses (`{Id ","}*`), and where the part after the arrow is a list of statements (`Statement*`) enclosed in curly braces.

5 Array comprehensions Comprehensions are convenient short-hand notation for building collections like lists, sets, or, in Javascript, arrays. They often have the following syntax:

```
syntax Expression = "[" Expression "|" {Generator ","}+ "]"
```

Generators come in two forms:

- Conditions: `Expression`
- Enumerators: `Id "in" Expression`

Write a desugaring that transforms comprehensions to an IFFE which contains a local accumulator array. A condition generator maps to an `if`-statement, and enumerators map to `for`-loops. The sequence of generators leads to nested `if`- and `for`-statements. Only at the innermost position is an element added to the accumulator array.

An example:

```
[ x | x in array, x % 2 === 0]
===>
(function() {
  var result = [];
  {
     var coll = array;
     for (var i = 0; i < coll.length; i++) {
       var x = coll[i];
       if (x % 2) {
         result.push(x);
       }
     }
  }
  return result;
})()
```

Tip: iterate through the generators in reverse using the `reverse` function of the Rascal standard library module `List`. Convert the syntactic sequence of generators (`{Generator ","}+`) to a list as follows: `[ g | g <- gens ]`. Now it's easy to start with the innermost statement, and wrap it successively with the `if`- and `for`-statements.

Optional: Rascal has builtin notation for reducers. For instance, to sum a list of integers, you can write:

```
( 0 | it + x | x <- [1..10] )
```

Use this construct to desugar comprehensions. Think about what the “zero” element should be.

Domain-Specific Languages

The above language extensions involved small additions to the Javascript language. Language extensions, however, do not have to be limited to this small scope. In fact, it is very well possible to embed complete Domain-Specific Languages on top of the host language! Check out the state machine language in the `src/demo` directory for an example.