# 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.

### Getting to know Rascal

To get to know Rascal a little bit (Rascal is a BIG language!), let's implement FizzBuzz.

- Create a new Rascal module in the `src` directory of the project via `File/New...`. You may call it "FizzBuzz".
- Define a function `myFizzbuzz()` returning `void`. You may want to look here for example implementations.
- Right-click on the editor and select `Start Console`. You should see a Rascal prompt `rascal>` in the console area of Eclipse.
- Import your newly created module `import FizzBuzz;`.
- Type in: `myFizzbuzz()` and see the result.

*Optional*: implement FizzBuzz using a different implementation strategy for instance, like listed on the Rascal tutor, and try to understand the code.

### 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 + `let`). 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 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 `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

SJS file

Save Hover doc See output in js file Open html file where see output + input + execution.

**NB**: the files should be one level below the `sjs` directory; i.e., in `sjs/somedir`.

How to enable ECMAScript 6 in Eclipse Browser

## Exercises

Below 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 Ruby-style instance variables** 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`.

**2 Pairs** Javascript has structured literals for objects and arrays, but not for pairs (tuples). Write a transformation that desugars pairs written in between < and > to object literals with fields `_1` and `_2`. IOW: `<E1, E2>` desugars to `{_1: E1, _2: E2}`.

**3 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);`.

*Optional*: write the syntax for a "dont" statement (similar to `todo`) 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 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?

*Optional*: write a desugaring for `repeat Body until "(" Cond ")"` which transforms to a `do-while` loop.

**5 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.

**6 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)?

**7 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)`.

**8 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: E) S`, where `X` is an identifier, `E` an expression and `S` a statement. The `foreach` statement should be desugared to something like:

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

Note the use of `let`; this is actually an ECMAScript 6 extension we need here. `let` works like `var`, except that the variables are block-scoped, instead of function scoped.

*Quiz*: why can't we use the IIFE here, but instead have to rely on ECMAScript 6 `let` to introduce a variable?

*Quiz*: why do we need to assign the expression `E` to a local variable (`array`) first?

**9 Fat arrow functions** ECMACScript 6 introduced fat arrow functions. Arrow functions are a short-hand 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.

**Bonus: 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! For an elaborate example, check out the state machine language in the `src/demo` directory.