Grammar of Graphics in HTML5

Source code, Live Demo
Curran Kelleher
4/29/2013

Introduction

If I've learned anything in this Design of Programming Languages class, it is that in Computer Science, the context you are working in (the language, the tools, the abstractions) determine *what makes sense*, and also determine *how you think* about the problem at hand. While implementing a Lambda Calculus interpreter in Haskell, I had new insights about how programs can be structured as functional compositions of many tiny programs, and how this structuring style allows programs to scale in complexity while remaining mentally manageable. After porting the Lambda Calculus to CoffeeScript, I decided to use a similar approach to tackle a problem I've been struggling with: how can the Grammar of Graphics best be implemented in HTML5?

Leland Wilkinson's book "The Grammar of Graphics" details a language he invented for declaratively specifying information visualizations. Here are some typical visualizations expressed using his language (p. 585), which he calls "Graphics Production Language":

- Scatterplot: `ELEMENT: point(position(d*r))`
- Line Chart: `ELEMENT: line(position(d*r))`
- Bar Chart: `ELEMENT: interval(position(d*r))`
- Pie Chart:
  ```
  COORD: polar.theta(dim(1))
  ELEMENT: interval.stack(position(summary.proportion(r)), color(c))
  ```
- Histogram: `ELEMENT: interval(position(summary.count(bin.rect(y))))`

Though this language is effective and useful for producing visualizations, its implementation is proprietary. If this language were implemented in an open source library using Web technologies, it could be used by anyone to visualize data on any Web page, and would be open to extension and evolution by visualization researchers around the world.

Haskell-like Pattern Matching in CoffeeScript via Lambda Calculus

Writing a Lambda Calculus evaluator in Haskell was a trip. For a class assignment, we were given a parser for the Lambda Calculus written in Haskell (using the Parsec library), and an example function that traversed the AST (abstract syntax tree) to reproduce the input expression string, `showLx`:

```
showLx (Lambda vv tt) = "(&" ++ [vv] ++ "." ++ showLx tt ++ ")"
showLx (Name vv)      = [vv]
showLx (Apply aa bb@(Apply _ _)) = showLx aa ++ "(" ++ showLx bb ++ ")"
showLx (Apply aa bb)  = showLx aa ++ showLx bb
showLx (Number nn)    = show nn
showLx (Symbol cc)    = [cc]
showLx (Error ee)     = ee
```

The `showLx` function in Haskell, which generates a Lambda Calculus expression from an AST.

The `showLx` function is a prime example of how Haskell's pattern matching can be used to define a recursive function that walks an abstract syntax tree. I had never seen functions defined in this way before coming in contact with Haskell, and was struck by the logic of the organization. Each line is a concise thought. After grokking `showLx` and implementing the Lambda Calculus in Haskell with functions following a similar pattern, I thought to myself "Is the JavaScript platform up to the task?"

JavaScript is often criticized for being ugly and weird, but it is arguably *the* language of the future. With the maturation of HTML5 and Node.js, the JavaScript platform is blooming with fresh libraries and new approaches to "App" development - Browser-based Apps, Mobile Apps, and Apps that run in Unix terminals. There is a good open source JavaScript library for just about anything you could imagine. CoffeeScript, a concise language that translates directly into JavaScript, overcomes many of the "cleanliness" limitations of the JavaScript language, allowing developers to write compact and expressive code for the JavaScript platform.

```coffeescript
# Functions
square = (x) -> x * x
# Conditionals that return values
factorial = (x) ->
  if x < 1
    1
  else
    x * (factorial x - 1)
# Objects literals
person =
  firstName: "Joe"
  lastName: "Schmoe"
# Object access, "truthiness" of existence
if person['firstName']
  console.log person.firstName # prints "Joe"
# Object destructuring, string interpolation
sayHello = ({firstName, lastName}) ->
  "Hi, I'm #{firstName} #{lastName}"
# Array comprehensions
squares = for num in list
  square num
# Could also be on one line
# square = (square num for num in list)
```

```javascript
var factorial, num, person, sayHello, square, squares;
square = function(x) { return x * x; };
factorial = function(x) {
  if (x < 1) {return 1;}
  else {return x * (factorial(x - 1));}
};
person = {firstName: "Joe",lastName: "Schmoe"};
if (person['firstName']) {
  console.log(person.firstName);
}
sayHello = function(_arg) {
  var firstName, lastName;
  firstName = _arg.firstName;
  lastName = _arg.lastName;
  return "Hi, I'm "+firstName+" "+lastName;
};
squares = (function() {
  var _i, _len, _results; _results = [];
  for (_i=0,_len=list.length;_i<_len;_i++) {
    num = list[_i];_results.push(square(num));
  }
  return _results;
})();
```

A CoffeeScript Rosetta Stone. CoffeeScript on the left, generated JavaScript on the right.

PEG.js is a parser generator for JavaScript. Porting the Lambda Calculus parser from Haskell to the parser DSL for PEG.js was a straightforward process. The parser builds an abstract syntax tree comprised of objects each having a String "type" property denoting its type. This AST organization affords the creation of a `byType` function that dispatches a function call based on value of the "type" property of the first argument. The `byType` function takes as input an object `fns` whose keys (which are Strings) correspond to values of the "type" property to match, and whose values are functions to be executed when value of the "type" property is matched.

```
byType = (fns) ->
  (tree) ->
    fn = fns[tree.type]
    if fn
      fn.apply null, arguments
    else
      throw Error "no match for type ‘#{tree.type}’"
```

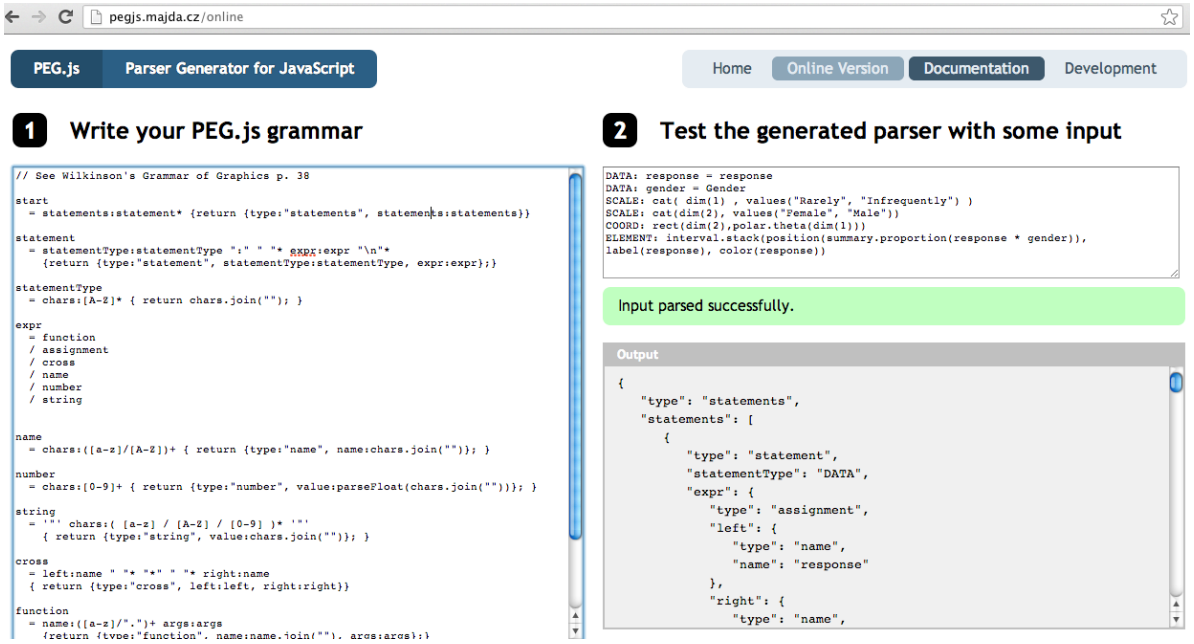
Using the `byType` function, the `showLx` function can be ported to CoffeeScript as follows, emulating Haskell's pattern matching:

```
show = byType
  lambda: (lambda) ->
    arg = lambda.arg.name
    body = show lambda.body
    "(&#{arg}.#{body})"
  apply: (apply) ->
    if apply.b.type == 'apply'
      (show apply.a) + '(' + (show apply.b) + ')'
    else
      (show apply.a) + (show apply.b)
  name: (name) -> name.name
  number: (number) -> number.value
```

The case of 'apply' illustrates the superiority of Haskell's pattern matching, but this is pretty close! The `byType` utility was used to port the rest of the Haskell Lambda Evaluator to CoffeeScript to great effect. Now the basic plumbing is in place to attack the Grammar of Graphics problem.

Better Haskell-like Pattern Matching in CoffeeScript via GPL

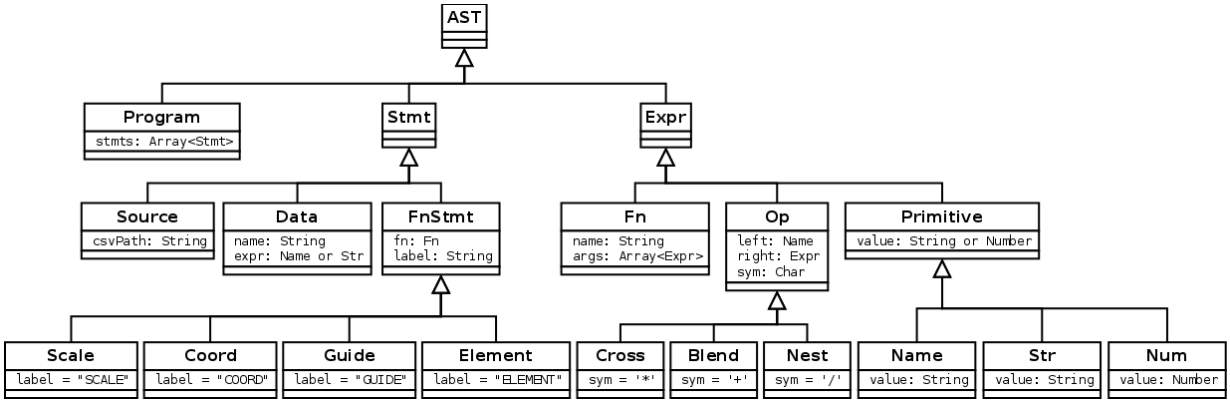
The first step toward implementing the Grammar of Graphics was to build a parser using PEG.js for Wilkinson's Graphics Production Language (GPL).



The GPL parser grammar with example input and output (in the PEG.js interactive tool).

JavaScript is an object oriented language with prototypal inheritance. Since prototypal inheritance is more powerful than class-based inheritance, class-based inheritance can be implemented (or "simulated") using prototypal inheritance. CoffeeScript has syntactic sugar for class-based inheritance, which has a well-thought-out implementation that sets up useful data structures such as the class hierarchy and named constructor references on instances. Using the data structures resulting from CoffeeScript's syntactic sugar for classes, a type matching function can be implemented that matches against classes:

```
matchClass = (fns) ->
  (obj) ->
    fn = fns[obj.constructor.name]
    if fn
      fn.apply this, arguments
    else
      throw Error "no match for type #{obj.constructor.name}."
```



Abstract syntax tree (AST) types used by our Graphics Production Language (GPL) parser.

After modifying the GPL parser to use classes for AST node types, the `show` function for a GPL program using `matchClass`, object destructuring, and string interpolation looks like this:

```
show = matchClass
  Program: ({stmts}) -> (_.map stmts, show).join '\n'
  Data: ({name, expr}) -> "DATA: #{name} = #{show expr}"
  Scale: ({fn}) -> "SCALE: #{show fn}"
  Coord: ({fn}) -> "COORD: #{show fn}"
  Guide: ({fn}) -> "GUIDE: #{show fn}"
  Element: ({fn}) -> "ELEMENT: #{show fn}"
  Name: ({value}) -> value
  Num: ({value}) -> value
  Str: ({value}) -> '"'+value+'"'
  Fn: ({name, args}) -> "#{name}(#{(_.map args, show).join ', '})"
  Cross: ({left, right}) -> "#{show left}*#{show right}"
  Blend: ({left, right}) -> "#{show left}+#{show right}"
  Nest: ({left, right}) -> "#{show left}/#{show right}"
```

This is pretty clean but has redundancy. Note the duplicated code with small variations for (`Scale`, `Coord`, `Guide`, `Element`), (`Name`, `Num`), and (`Cross`, `Blend`, `Nest`). We can do better by making the matching utility polymorphic: if the class of the object doesn't match, walk up its class hierarchy until a match is found. For this we can take advantage of the data structure that is set up by CoffeeScript's syntactic sugar for classes, namely `constructor.name` and `constructor.__super__`. Here's what the polymorphic `match` function looks like:

```
match = (fns) ->
  (obj) ->
    constructor = obj.constructor
    fn = fns[constructor.name]
    while !fn and constructor.__super__
      constructor = constructor.__super__.constructor
      fn = fns[constructor.name]
    if fn
      fn.apply this, arguments
    else
      throw Error "no match for type #{constructor.name}."
```

This polymorphic `match` function can simplify our `show` implementation. The types can be treated as more generalized types as follows:

- (`Scale`, `Coord`, `Guide`, `Element`) → `FnStmt`
- (`Name`, `Num`) → `Primitive`
- (`Cross`, `Blend`, `Nest`) → `Op`

Here is the simplified `show` that takes advantage of the class hierarchy:

```
show = match
  Program: ({stmts}) -> (_.map stmts, show).join '\n'
  Data: ({name, expr}) -> "DATA: #{name} = #{show expr}"
  FnStmt : ({label, fn}) -> "#{label}: #{show fn}"
  Primitive: ({value}) -> value
  Str: ({value}) -> '"'+value+'"'
  Fn: ({name, args}) -> "#{name}(#{(_.map args, show).join ', '})"
  Op: ({left, right, sym}) -> "#{show left}#{sym}#{show right}"
```

An Aside on Development Principles

YAGNI - You Ain't Gonna Need It

Even if you think you'll need a certain feature, generality, or optimization, don't put it in until you need to use it. This keeps code as simple as possible, and ensures that each feature, generality, or optimization added gets tested and used right away.

KISS - Keep It Simple, Stupid

Keep everything as simple, clear, and mentally manageable as possible. For example, when a source file grows to over a couple hundred lines of code, split it up into separate self-contained

modules. Functional programming utilities should feel like fundamental operations, so I like to extract Underscore methods to module-local variables to simplify statements like `(_.map foo, bar)` to statements like `(map foo, bar)`.

Types Matter

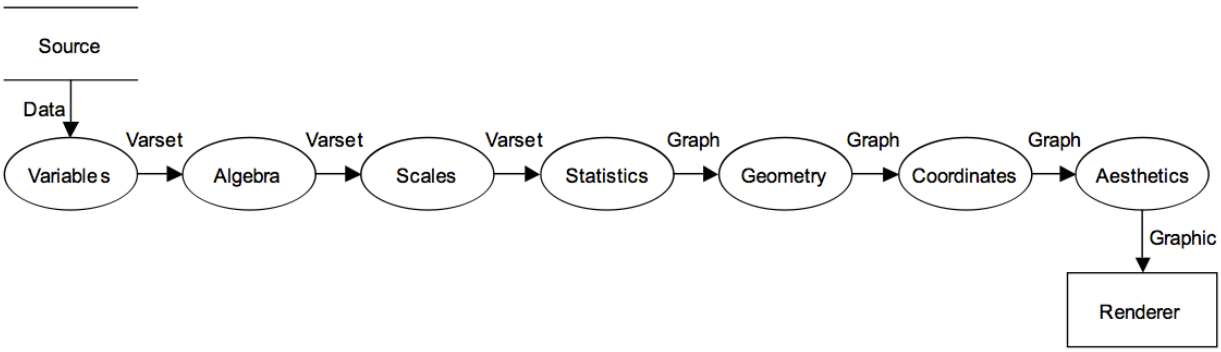
One thing I love about Java is that it is strongly typed. JavaScript (and thus CoffeeScript) is not strongly typed. This does not mean, however, that it is impossible to express and enforce type constraints in the language. While constructing the AST classes, I found myself desiring to express type constraints on the properties of each class. Here is a type checking utility function that allows one to write code that enforces type constraints at runtime (and communicates useful type information to future readers of the code). It supports JavaScript's Number and String primitive types, as well as types created by CoffeeScript classes. Note that it walks up the class hierarchy using the data structures set up by CoffeeScript's syntactic sugar for classes, enabling any type in the hierarchy to be matched.

```
type = (object, expectedType) ->
  error = false
  if expectedType == Number
    error = (typeof object != 'number')
  else if expectedType == String
    error = (typeof object != 'string')
  else
    c = object.constructor
    error = (c != expectedType)
    # Walk up the class hierarchy
    while error and c?.__super__
      c = c.__super__.constructor
      error = (c != expectedType)
  if error
    throw Error 'Type Error'
```

This type function allows us to, for example, define an AST node class with "strongly typed" properties as follows:

```
class Source extends Stmt
  constructor: (@csvPath) ->
    type @csvPath, String
```

The Grammar of Graphics Pipeline



Leland Wilkinson's original pipeline diagram (Grammar of Graphics p.24).

The Grammar of Graphics Pipeline is a process that transforms data into graphics. The process is governed by GPL statements. As Wilkinson notes, there are many alternative computational strategies that can be used to implement the pipeline in different ways: procedural processes, functional processes, an object oriented system, or a data flow network. No matter how it is implemented, the order of the phases is fixed. Since our context affords clean functional algorithms for AST traversal and transformation, it makes sense to implement the pipeline using functional processes that allow the data to "bubble through" the AST.

The self-contained functional processes of our implementation are as follows:
source → variables → algebra → scales → statistics → (geometry → coordinates → aesthetics)

The **source** phase of our implementation extracts `SOURCE` statements from the AST and loads the specified CSV files. Each column name is made available to `DATA` statements for creating variables.

The **variables** phase of our implementation extracts `DATA` expressions from the AST and creates new variable name bindings based on the assignments therein. The named variables are made available use in the **algebra** phase.

The **algebra** phase of our implementation evaluates graphics algebra expressions, replacing top-level occurences of `Op` in the AST with references to memory-resident data tables encapsulated in a `Relation` abstraction (a table with rows and named columns). These relations contain the result of evaluating the previously present `Op` tree against the variable names made available in the **variables** phase.

Next the **scales** phase extracts the scales defined in `SCALE` statements and uses them to normalize the relations present in the AST. After this phase, the relations in the AST contain normalized values between 0 and 1.

The **geometry → coordinates → aesthetics** part of the pipeline is organized in a computationally different way. First the `COORD` statements are extracted from the AST and stored for later use. Next for each `ELEMENT` statement in the AST, two functions that take as input a key and a Mark and return as output a modified Mark are generated:

- `geometry` evaluates the `position` portion of the element statement
- `aesthetics` evaluates all the aesthetics other than `position`, such as `size` and `color`.

With these functions in hand, the **geometry → coordinates → aesthetics** part of the pipeline can be roughly described by the following pseudocode:

```
coordinates = extractCoordinates(ast)
renderFns = for each ELEMENT statement
  generate (geometry, aesthetics) functions
for each (geometry, aesthetics) in renderFns
  for each key of the evaluated graphics algebra expression
    mark = default mark
    mark = geometry(key)
    mark = applyCoordinates(mark, coordinates)
    mark = aesthetics(mark)
    render(mark, canvas)
```

The `mark` abstraction encapsulates Jaque Bertin's notion of a visual mark that is a primitive component of a visualization. A "default mark" is generated that has default characteristics (such as size and color), then passed through a series of transformations that execute the **geometry → coordinates → aesthetics** part of the pipeline for a single key. Executing this pipeline for each key in the relation resulting from evaluating the graphics algebra expression yields a rendered visualization.

Conclusion

Two new "language features" have been introduced for CoffeeScript: `match` for emulating Haskell's pattern matching, and `type` for expressing and enforcing type constraints. Use of a `match` variant that matches against an object property value was shown to be useful for walking an abstract syntax tree (AST) for the Lambda Calculus. A parser for a variant of Leland Wilkinson's Graphics Production Language (GPL) was introduced. The advantages of a polymorphic variant of `match` were showcased in a `show` function that generates a GPL expression string from an AST. An evaluator for our GPL variant that uses CSV files and HTML5 Canvas was introduced that implements the full Grammar of Graphics pipeline, showcasing the use of `match` for AST transformation functions.