Tool in your browser: from Tool to JavaScript

Computer Language Processing '14 Final Report

Ogier Bouvier Tristan Overney
EPFL
{ogier.bouvier}{tristan.overney}@epfl.ch

1. Introduction

Describe in a few words what you did in the first part of the compiler project (the non-optional labs), and briefly say what problem you want to solve with your extension.

This section should convince us that you have a clear picture of the general architecture of your compiler and that you understand how your extension fits in it.

2. Examples

2.1 IO Functions

Tool's greatest limitation is its lack of interaction with the user. JavaScript, on the contrary, has some already implemented functions for all those purposes. All these IO functions are managed through the use of new built-in IO object in order to reduce the number of keywords required.

2.1.1 Input management

Name	Role
readDouble	Allows the user to enter a Double
readString	Allows the user to enter String
readInteger	Allows the user to enter an Integer

Table 1. IO methods

All these methods are meant to be used as expressions and can be combined as any expressions as shown below.

```
var b : Int;
var c : Double;

b = IO.readInteger();
c = b + IO.readDouble();
```

2.1.2 Output management

Running inside of a browser allows for two kinds of output. We can either show a popup or write on the webpage. Both these methods are supported as shown in Table 2. For debugging purposes, it is also possible to log message to the Javascript console.

Name	Role
log	Logs a message in the browser's console
writeLine	Writes a line on the webpage
showPopup	Creates a popup in the browser

Table 2. Output methods

```
println(''Hello''); //Compatibility
IO.showPopup(10 + ''Some string'');
IO.writeLine(''<strong>Some HTML!</strong>'');
```

2.2 New Types

Another great limitation of Tool is that it only supports very basic types. Our extension adds support for generic arrays and floating point numbers to Tool.

```
var c : Double;
c = 11.2;
IO.showPopup(''Here's a Double: '' + c);
```

2.2.1 Double

In order to facilitate manipulations of basic types Integer are considered to be a subtype of Double. Indeed you can view an Integer as a Double with a zero decimal part. It is thus possible to add an Integer with a Double and the result will be a Double.

```
var c : Integer;
var d : Double;
```

```
c = 10;
d = 11.2;

println(''This is ''c + d); // This is 21.2
```

2.2.2 Array of any Types

Generic arrays are also part of the extension. It is now possible to instantiate arrays of any types, including Object types.

```
class Hello {
  def method() : Int = {
    return 10;
  }
}

def someMethod() : Integer = {
  var array : Hello[];
  array = new Hello[5];
  array[2] = new Hello();
  return array[2].method();
}
```

2.2.3 Array of more than one dimension

If you want to see a real world use of a two dimension matrix, you can check the file *programs/Sudokupdated.tool* which is an updated version of the given *Sudoku.tool* making use of the multi-dimensional arrays.

```
var d : Double[][]; //Matrix of Double
var i : Integer;

d = new Double[3];
i = 0;
while(i < 3) {
  d[i] = new Double[3];
}
println(d[0][2]); //d is a 3x3 matrix of Doubles
```

2.3 Parametrized constructor

The support for constructors has also been added to the Tool language. However currently only one constructor per class is allowed. The constructor must be declared directly after the fields of the class and before any method declaration. In order to retain compatibility with older Tool programs a class declaring no constructor is assumed to have a default constructor which takes no arguments.

The following code shows how to declare a constructor:

```
class A {
  var field1 : Int;
  var field2 : Double;

  this(param1 : Int, param2 : Double) = {
    field1 = param1;
    field2 = param2;
  }
}

var a : A;
a = new A(10, 11.2);

class B {
  //B automatically has a default constructor
}

var b : B;
b = new B(); //Using the default constructor
```

3. Implementation

This section will summarize all the implementation details we had to implement after a quick look to the required theoretical background.

3.1 Theoretical Background

While not using specific theoretical concepts none of us had ever done any programming in JavaScript. The first task was then to learn about the specifications of this language in order to be able to identify the "translation" we'd have to make. [**?**] gave us a general picture about the ins-and-outs of JavaScript. And [**?**] was filled with details about the prototypes in JavaScript which we had to use as there is no classes in that delightful language.

3.2 Implementations Details

3.2.1 Tool Classes and inheritance

As already mentioned, JavaScript does not have Classes as of, instead it has an element called Object which you can define a prototype of and then define a function associated to a prototype. Heritage its self can be simulated with the method *Object.create(prototype)* which will take the *prototype* as a base for our child class.The following code snippet represent a simple heritage in Tool:

```
class Slot {
        var value: Int;

        this(val : Int) = {
```

```
        value = val;
    }

    def getVal() : Int = {
        return value;
    }

    def isInit() : Bool = {
        return false;
    }

    def setVal(val : Int) : Slot = {
        value = val;
        return this;
    }
}

class InitSlot extends Slot{

    this(val : Int) = {
        value = val;
    }

    def isInit() : Bool = {
        return true;
    }
}
```

And the following snippet is the Javascript code generated from the previous Tool code:

```
function Slot(val){
    this.value = val;
}
Slot.prototype.getVal = function() {
    return this.value;
}
Slot.prototype.isInit = function() {
    return false;
}
Slot.prototype.setVal = function(val) {
    this.value = val;
    return this;
}

InitSlot.prototype = Object.create(Slot.prototype);
function InitSlot(val){
    this.value = val;
}
InitSlot.prototype.isInit = function() {
    return true;
}
```

3.2.2 Imitating the environnement of a shell

3.2.3 Input management

JavaScript has a built in *prompt()* method but this does carry Type information so as our Tool is strongly typed, we have created three input reading methods (as you've seen in Table 1.) which are all translated to a *prompt()* call but we use the functions *parseInt()* or *parseFloat()* on the result of the method call to ensure we have either integer or double and thus respect tool's strong typing.

3.2.4 Double in Tool and Integer in JavaScript

Integer are now defined as a subtype of double so it is possible to do the arithmetic operations on an integer and a double without any trouble thanks to the weak typing of JavaScript.

But this introduce a new problem, as there is only a number type (IEEE 754 64-bits standard) there is no integer in absolute. This is problematic, for example when you want to do an integer by integer division in JavaScript the result will not be an integer so we had to make a special case for that kind of division and call *Math.floor()* over the division.

3.2.5 Array of any Types (including arrays of arrays)

To be able to handle multiple types of array we had to build a similar architecture for the arrays than the one existing for the TObject. We then have a type named TGenericArray which ensure that we have indeed an array and the type TArray which has a field innerType describing the nature of the array. This way, we are able to have arrays of any types and the sub-typing is done by comparing the types of the innerTypes. Lexing and parsing had to be changed in order to support any number of brackets (i.e. *superArray[][][][][]*) and to support method call on array reading (i.e. *superArray[2].myMethod()*).

3.2.6 Parametrized constructor

The syntax for constructor declaration was designed with the primary goal of keeping the Tool grammar LL(1). Initially, the support for multiple constructors was planned but Javascript's handling of constructor does not allow for constructor overloading.

3.2.7 Legacy support

As the number of developper using our modified syntax of Tool is really small, we were obligated to have

legacy support for the original Tool syntax in order to have more that five programs to test! That's why we kept the lexing of println() but we read it as a token WriteLine to avoid code duplication.

4. Current Limitation

The inherent design of JavaScript prototypes make it impossible to have a field and a function sharing the same name. But it is not something our current implementation is either checking or correcting it (by appending something to the function name if there is already a field having the same name in the Tool class).

As there is no pretty and foolproof ways to solve that issue in a reasonable amount of time we chose not to fix it and, as such, acknowledge it as a feature.

5. Possible Extensions

Generating "web pages" from Tool is quite awesome but for now our web pages imitate the console environnement in the webpage, which is a bit nasty in 2014.

A possible extension would be to generate more complex and web 2.0 web pages. Such goal could be implemented by adding the support for imports in Tool and creating framework to generate specific part of web pages through Tool. But aside of the import support and possible addition of new IO methods, this is not much of a compiler question rather than just writing a Tool library to generate HTML and/or CSS.

Another extension would be to add any other cool feature to our modified Tool syntax (e.g: Scala-like pattern matching) and do the changes needed to all our compiler components (lexer, parser, name analyser and type checker) and to our Tool-to-JS translator, the printerJS.

References