Abed Tony BenBrahim
ba2305@columbia.edu

JTemplate

COMS W4115 Final Report
Professor Stephen Edwards
Summer 2009

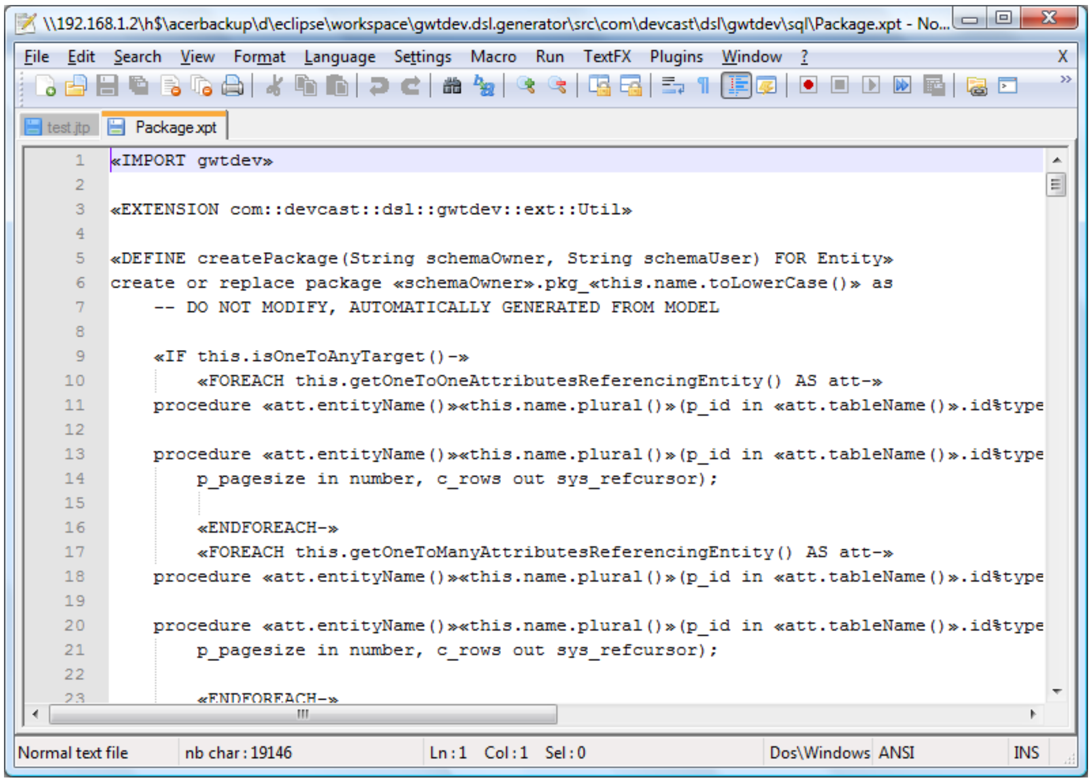
Abed Tony BenBrahim
ba2305@columbia.edu

JTemplate: Another Approach to Document Template Markup Languages

Introduction

In domains such as web page development and model driven software development (MDSD), the automated generation of documents is most often accomplished by the use of processing instructions written in a markup language, embedded directly into a document template, to conditionally or iteratively generate sections of the document or programmatically insert content. Mainstream web development technologies such as Java Server Pages, PHP, Ruby on Rails and Microsoft's ASP(X) all employ this method of embedding code inside a document template to generate dynamic content. In the same fashion, popular MDSD frameworks such as openArchitectureWare (openArchitectureWare 2009) make extensive use of templates to transform application models to source code.

The use of templates with embedded processing instructions greatly accelerates the initial development of document generation applications. However this methodology imposes severe penalties in terms of future maintainability. As the ratio of markup instructions to template content increases (Figure 1), it becomes increasingly difficult to locate and correct defects. Often, the template becomes a "write once" document, with each modification becoming prohibitively more expensive and error-prone to undertake.


\\192.168.1.2\h$\acerbackup\d\eclipse\workspace\gwtdev.dsl.generator\src\com\devcast\dsl\gwtdev\sql\Package.xpt - No...
File Edit Search View Format Language Settings Macro Run TextFX Plugins Window ?
test.jtp
Package.xpt
Normal text file
nb char : 19146
Ln : 1 Col : 1 Sel : 0
Dos\Windows ANSI
INS

```
«IMPORT gwtdev»

«EXTENSION com::devcast::dsl::gwtdev::ext::Util»

«DEFINE createPackage(String schemaOwner, String schemaUser) FOR Entity»
create or replace package «schemaOwner».pkg_«this.name.toLowerCase()» as
    -- DO NOT MODIFY, AUTOMATICALLY GENERATED FROM MODEL

    «IF this.isOneToAnyTarget()-»
        «FOREACH this.getOneToOneAttributesReferencingEntity() AS att-»
    procedure «att.entityName()»«this.name.plural()»(p_id in «att.tableName()».id%type

    procedure «att.entityName()»«this.name.plural()»(p_id in «att.tableName()».id%type
        p_pagesize in number, c_rows out sys_refcursor);

        «ENDFOREACH-»
        «FOREACH this.getOneToManyAttributesReferencingEntity() AS att-»
    procedure «att.entityName()»«this.name.plural()»(p_id in «att.tableName()».id%type

    procedure «att.entityName()»«this.name.plural()»(p_id in «att.tableName()».id%type
        p_pagesize in number, c_rows out sys_refcursor);

        «ENDFOREACH-»
```

Figure 1- A view of an openArchitectureWare template to generate an Oracle PL/SQL package

Several approaches have emerged to achieve the holy grail of separating markup instructions from content. The Apache Wicket web framework (Apache Wicket 2009) takes advantage of the tree structure of HTML to insert content into specified nodes, an approach that works well for

XML like and other hierarchical structures but presents severe challenges for less structured content such as a source code. Terrence Parr's StringTemplate library (Parr 2004) makes use of minimal template markup and an output grammar to generate dynamic content. While this is a promising approach, the use of output grammars and complex recursive constructs has hindered the widespread adoption of this technique. The JTemplate language proposed in this document explores another method of providing clear separation of markup instructions from content.

The JTemplate Language

Defining Templates

In the JTemplate language, templates are first-class constructs. The following code snippet shows a template for a Java bean, with each line of the template annotated with an optional label (an integer or an identifier) and a start of line marker.

```
template javaBean{
1      #package mypackage;
       #
2      #import java.util.Date;
       #
3      #public class MyClass{
       #
4      #     private Type field;
4      #
5      #     public MyClass(){
       #     }
       #
8      #     public MyClass(fieldList){
9      #            this.field=field;
8      #     }
       #
6      #     public Type getField(){
6      #            return this.field;
6      #     }
6      #
7      #     public void setField(Type field){
7      #            this.field=field;
7      #     }
7      #
       #}
}
```

The labels serve to delineate individual lines or blocks of code that can be manipulated by processing instructions. Likewise, processing instructions are first class constructs in the JTemplate language, as shown in the following code snippet:

```
instructions for javaBean(appOptions, entity){
1 once: mypackage = endWith('.',appOptions.basePackage) + 'model';
2 when (hasDateField(entity));
3 once: MyClass = toFirstUpper(entity.name);
4 foreach (field in entity.fields): Type=javaType(field), field=field.name;
5 once: MyClass = toFirstUpper(entity.name);
6 foreach (field in entity.fields):
       Type=javaType(field), field=field.name, Field=toUpper(field.name);
7 foreach (field in entity.fields):
      Type=javaType(field), field=field.name, Field=toUpper(field.name);
8 once : MyClass = toFirstUpper(entity.name), fieldList=getFieldList(entity);
9 foreach (field in entity.fields): field=field.name;
}
```

A processing instruction consists of a line or block label matching a label in the corresponding template, an expression specifying how the instruction should be processed (once, conditionally or iteratively) and an optional list of text replacements. Additionally, processing instructions accept arguments in the same way as a function declaration.

The JTemplate Language

The remainder of the JTemplate language consists of a subset of ECMAScript 5 (ECMA International April 2009) operating in strict mode. JTemplate is an interpreted, dynamically typed language. First-class types consist of integers, strings, doubles, Booleans, arrays, maps and functions. All variables must be declared and initialized before being referenced. Control structures include if/else, while, foreach and switch. The following incomplete code snippet shows how the JavaBean template described above might be generated:

```
var model={
  entities: [
    {name: 'customer', fields: [
       {name: 'lastName', type: 'char',  maxLength: 50},
       {name: 'firstName', type: 'char', maxLength: 50}
    ],
    references: [
      {entity: 'address', cardinality: 'one-to-many'}
    ]
  },
    {name: 'address', fields: [
      {name: 'address1', type:'char', maxLength: 100},
      {name: 'address2', type:'char', maxLength: 100}
     ]
    }
  ]
};

var appOptions={basePackage: 'edu.columbia.w4115'};

var hasDateField=function(entity){
  foreach (field in entity.fields){
    if (field.type=='date'){
      return true;
    }
  }
  return false;
};

var main=function(){
  var path='gen/'+replace(appOptions.basePackage,',','/')+'/model/';
  mkDirs(path);
  foreach(entity in model.entities){
    var text=javaBean(appOptions, entity);
    writeFile(path+toFirstUpper(entity.name)+'.java', text, true);
  }
};
```

Development Environment

Project web site: http://code.google.com/p/ojtemplate/

Source code repository: Anon SVN http://ojtemplate.googlecode.com/svn/trunk/jtemplate/

IDE: Eclipse/OcaIDE

Build System: ocamlbuild

Works Cited

Apache Wicket. http://wicket.apache.org/.

ECMA International. *ECMAScript Language Specification (Final Draft Standard ECMA-262).* Geneva: ECMA International, April 2009.

openArchitectureWare. http://www.openarchitectureware.org/.

Parr, Terrence. "Enforcing strict model-view separation in template engines." *Proceedings of the 13th international conference on World Wide Web.* New York: ACM, 2004.

Abed Tony BenBrahim
ba2305@columbia.edu

Introduction to JTemplate

Table of Contents

Introduction to Jtemplate

Introduction

Jtemplate's syntax emulates the syntax of C like languages like Java, C# or Javascript. This tutorial assumes a familiarity with one of those languages. More details are available in the Language Reference Manual.

Where indicated, tutorial files are included in the `tutorial` subdirectory of your distribution. To run a jtemplate program, simple invoke `jtemplate` on the command line, followed by the script file name and optionally script arguments.

Jtemplate is dynamic

Jtemplate is a dynamic scripting language. This means values have types, not variables. Variables assume the type of the value they are initialized with. Variables are *initialized* with the `let` or `var` keyword:

```
let a=1;            // integer initialization
var b=1.2;          // floating point
let c=”Hello”;      // string
var d=‘Hello’;      // another string. Strings can be delimited with “ or ‘A
let e=true;         // a boolean
let f=void;         // void, or the absence of a value;
```

A variable can be *assigned* a new value after initialization by omitting the `let` keyword:

```
let a=1;     // initialization
a=a+1;       // assignment
```

Jtemplate has collection types

Jtemplate has two collection types, arrays and maps. Arrays and maps can contain any other type of values, including collection types.

```
let values = [1,2.1,‘abc’, true]; // an array
let info={John: 42, Mary: 36};   // a map

// an array containing two maps
let people1=[{name: ‘John’,age: 42}, {name: Mary, age: 35}];

// a map containing two arrays
let people2={names: [‘John’,‘Mary’], ages: [42,35]};
```

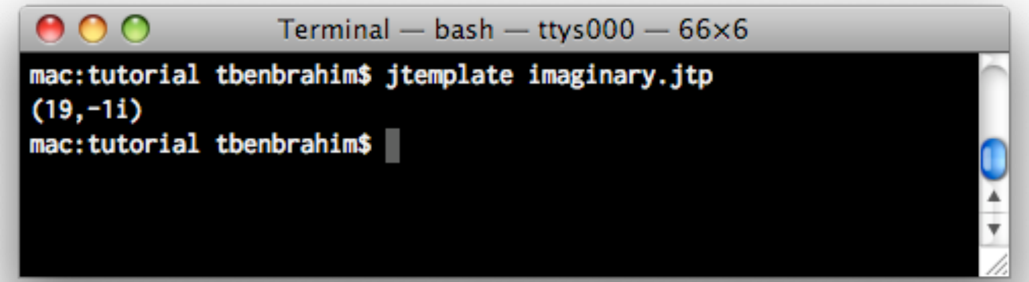
Array elements are accessed using an integer index, and maps are accessed using the key name.

```
let a=values[2];           // the third element of array values
let b=info.John;           // the value of the John key in info
let b=info[‘John’];        // another way to get the value of John key in info

let age=people1[0].age;    // John’s age in the people1 array
let age=people1[0][‘age’];// another way to get John’s age
let age=people2.ages[0];         //John’s age in the people2 array
let age=people2[‘ages’][0];      // another way
```

Functions are first class values

Functions in Jtemplate are first class values. This means they can be assigned to variables, stored in maps and arrays, and even passed as arguments to functions.

```
// a function definition
let add=function(x,y){
      return x+y;
};

// an array with two functions
let ops=[ function(x,y){ return x+y;}, function(x,y){ return x-y;} ];

//invoking the first array function
let sum=ops[0](2,4);

//sample included in tutorial/imaginary.jtp
// a map with three functions
let Imaginary={
      add: function(i1,i2){
            return {real: i1.real+i2.real, i: i1.i+i2.i};
      },
      subtract: function(i1,i2){
            return {real: i1.real-i2.real, i: i1.i-i2.i};
      },
      print: toString(i1){
            return '(' + i1.real + ',' + i1.i + 'i)';
      }
};

//invoking two of the map functions above.
let a={real: 12, i: 1};
let b={real: 7, i:-2};
println(Imaginary.toString(Imaginary.add(a,b)));
```



```
mac:tutorial tbenbrahim$ jtemplate imaginary.jtp
(19,-1i)
mac:tutorial tbenbrahim$
```

Jtemplate has control flow instructions

Jtemplate has three control flow instructions, `for`, `while` and `foreach`. The `foreach` command is used to iterate through an array's values or a map's keys.

```
// A for loop
for(i=0;i<10;++i){
	println('Hello');
}

// The equivalent while loop
let i=0;
while(i<10){
	print('Hello');
	i++;
}

//foreach example
let arr=[1,2,3,4];
foreach(el in arr){
	print(el,' ');
}
println();
```

The if/else statement allows statements to be conditionally executed based on the result of a conditional test.

```
// an if/else statement
if (a<10){
	println('small');
}else{
	println('large');
}

// an if statement
if (a<0){
	a=-a;
}
```

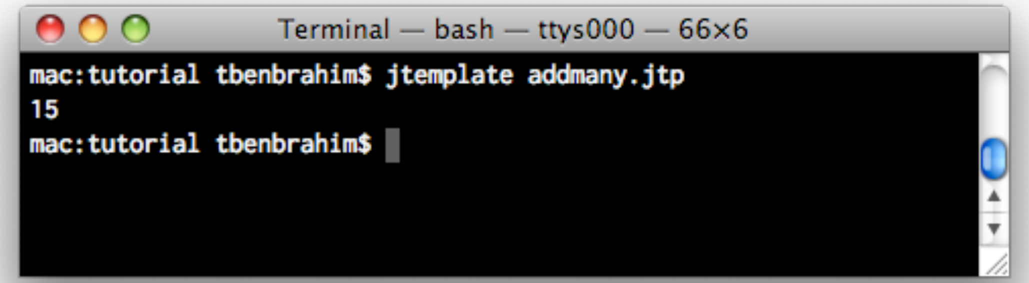
The switch statement allows statements to be conditionally executed based on a match between the switch statement and a case expression.

```
switch( i % 2){
	case 0:
		return 'even';
	default:
		return 'odd';
}
```

Jtemplate supports varag functions

Jtemplate supports function with a variable number of arguments. One such function is the built in `println` function, which accepts any number of arguments. You can also create your own by adding ... to the *last* argument. This argument will be passed in as an array.

```
//sample included in tutorial/addmany.jtp
let addMany=function(args...){
	let result=0;
	foreach(arg in args){
		result=result+arg;
	}
	return result;
};

println(addMany(1,2,3,4,5));
```



```
mac:tutorial tbenbrahim$ jtemplate addmany.jtp
15
mac:tutorial tbenbrahim$
```

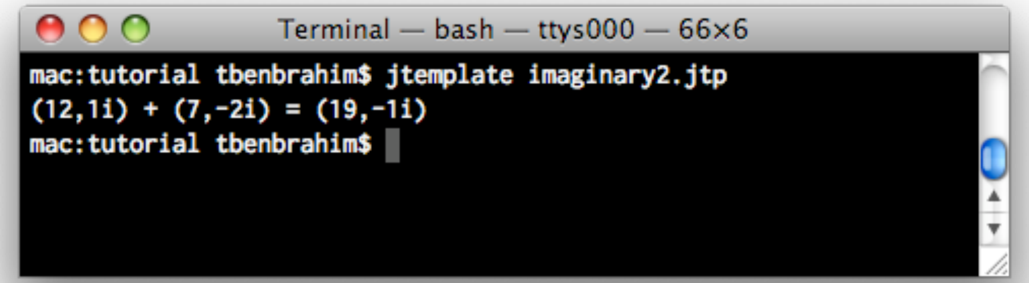
Jtemplate supports flexible dispatch of map functions

We already saw map functions in the Imaginary map. It turns out Jtemplate has a much more flexible map function dispatch mechanism. When a function `bar` is invoked on a map `foo`, it looks for the correct function to run in the following order:

```
foo.bar();
foo.prototype.bar();
foo.prototype.prototype.bar();
Map.bar();
```

Additionally when a map member function is called, the member (the part before the function call) is assigned to a special variable `this` that can be accessed by the function. We have already seen the first case. The second and third case allow for classes of objects to be created.

```
//sample included in imaginary2.jtp
let Imaginary={
  prototype: {
    create: function (r,i){
      return {real: r, i: i, prototype: Imaginary};
    },
    add: function(i2){
      return {real: this.real+i2.real, i: this.i+i2.i, prototype: Imaginary};
    },
    subtract: function(i2){
      return {real: this.real-i2.real, i: this.i-i2.i, prototype: Imaginary};
    },
    toString: function(){
      return '(' + this.real + ','+ this.i +'i)';
    }
  }
};

let a=Imaginary.create(12,1);
let b=Imaginary.create(7,-2);
println(a.toString(),' + ',b.toString(),' = ',a.add(b).toString());
```



```
mac:tutorial tbenbrahim$ jtemplate imaginary2.jtp
(12,1i) + (7,-2i) = (19,-1i)
mac:tutorial tbenbrahim$
```

Note that the functions are now defined in the `prototype` map of the `Imaginary` map. `Imaginary` is our class. Note also that every imaginary object created or returned from a function has its prototype set to `Imaginary`. This is the way to declare that the map is an instance of class `Imaginary`. When we call add on member `a`, the function that is called is `a.prototype.prototype.add`, with `a` assigned to `this` and `b` assigned to the argument `i2`.

This mechanism can even be used to support inheritance, as detailed in the Language reference manual. This powerful mechanism is also used in the runtime library to implement functions on the basic types. There are built in map prototypes for all the basic types, such as String, Integer, Float, Boolean, Map, Array and Void. When function `bar` is invoked on string `foo`, for example, the function that will be run is `String.prototype.bar`, with `foo` passed in as `this`.

Jtemplate has a runtime library

Jtemplate has a runtime library to manipulate the basic types, as well as for common functionality such as reading or writing files. A complete list of the library functions can be found in the Language Reference Manual.

Some functions are static. They are involved on the type map itself. An example is `Integer.random`.

```
// initializes rand with a random number with 0 and 100 exclusive
let rand=Integer.random(100);
```

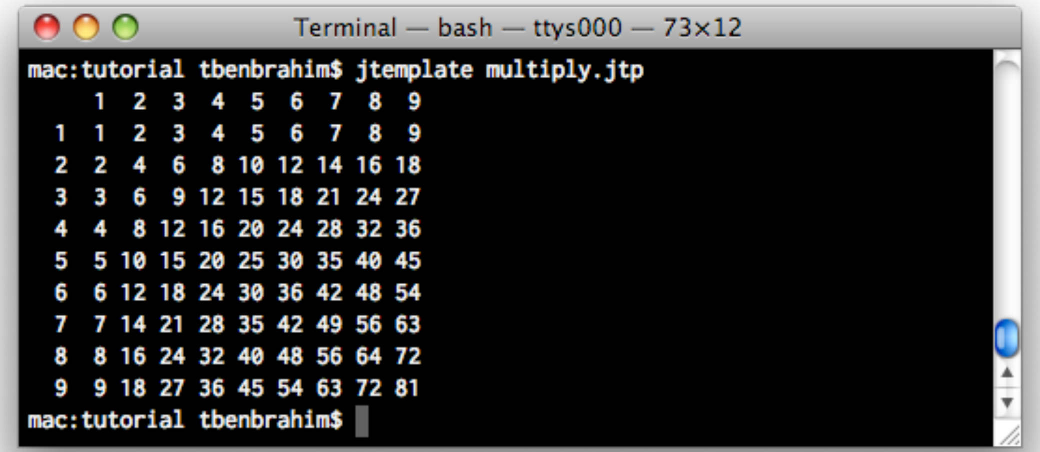
Other functions operate directly on the value.

```
let arr=[1.23, 12, 'abc'];
let len=arr.length(); //sets len to the length of the array

let s='This is a test';
let sl=s.length(); //sets sl to the length of the string
```

Jtemplate is extensible

The runtime library can be extended with user defined functions. For example, while the runtime library contains many String functions, it does not contain functions to pad a string to a certain length. However, we can easily add that functionality, by extending the `String.prototype` with function `lpad`, which pads a string to the left to a certain length using the specified character. We then extend `Integer` to add an `lpad` function.

```
//sample included in multiply.jtp

//left pads a string to length len with character char
let String.prototype.lpad=function(len,char){
	while(this.length()<len)
		this=char+this;
	return this;
};

//left pads an integer to length len with character char
let Integer.prototype={};
let Integer.prototype.lpad=function(len,char){
	return (this+'').lpad(len,char);
};


for (let row=0;row<10;++row){
	switch(row){
		case 0:
			print('   ');
			for (let col=1;col<10;++col)
				print(col.lpad(3,' '));
			break;
		default:
			print((row+'').lpad(3,' '));
			for (let col=1;col<10;++col)
				print((col*row).lpad(3,' '));
			break;
	}
	println();
}
```



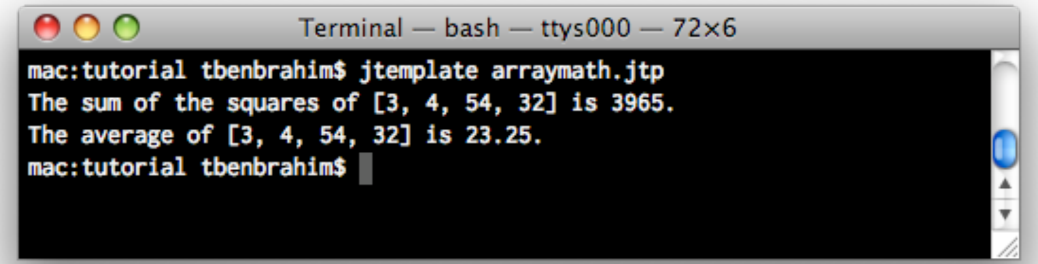
```
mac:tutorial tbenbrahim$ jtemplate multiply.jtp
     1  2  3  4  5  6  7  8  9
  1  1  2  3  4  5  6  7  8  9
  2  2  4  6  8 10 12 14 16 18
  3  3  6  9 12 15 18 21 24 27
  4  4  8 12 16 20 24 28 32 36
  5  5 10 15 20 25 30 35 40 45
  6  6 12 18 24 30 36 42 48 54
  7  7 14 21 28 35 42 49 56 63
  8  8 16 24 32 40 48 56 64 72
  9  9 18 27 36 45 54 63 72 81
mac:tutorial tbenbrahim$ 
```

Jtemplate supports higher order functions

Higher order functions are functions that accept functions as arguments or return a function as a result. The `Array` prototype has built in functions `length`, `push`, and `pop`. We will extend it to include a function `map`, that takes an array and returns an array with a function applied to each element, and a function `reduce`, that takes an array and an initial value and returns the cumulative result of applying a function to each element and the intermediate result. We then use these to calculate the sum of the squares and the average of an array.

```
//sample included in arraymath.jtp

let Array.prototype.map=function(f){
        let result=[];
        foreach(el in this)
                result.push(f(el));
        return result;
};

let Array.prototype.reduce=function(f, acc){
        let result=acc;
        foreach(el in this)
                result=f(el,result);
        return result;
};

let arr=[3,4,54,32];
let square=function(x){return x*x;};
let sum=function(x,y){return x+y;};

let ss=arr.map(square).reduce(sum,0);

println("The sum of the squares of ", arr, " is ", ss, '.');
```

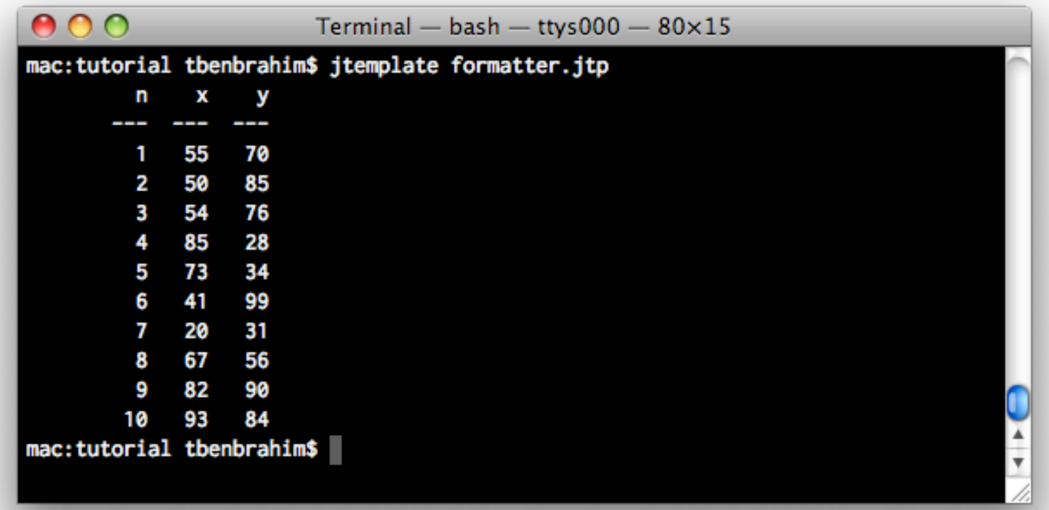
```
let avg=arr.reduce(sum,0)/(1.0*arr.length());

println("The average of ", arr, " is ", avg, '.');
```



```
mac:tutorial tbenbrahim$ jtemplate arraymath.jtp
The sum of the squares of [3, 4, 54, 32] is 3965.
The average of [3, 4, 54, 32] is 23.25.
mac:tutorial tbenbrahim$
```

A function can also return a function. In the following example, we define a function `String.formatter` that accepts an array of column sizes and returns a function that will format its input to the size of the columns. We then use this function create a formatter that is used to print 10 random points in cartesian space.

```
//sample included in tutorial/formatter.jtp

//left pads a string to length len with character char
let String.prototype.lpad=function(len,char){
	while(this.length()<len)
		this=char+this;
	return this;
};

let String.formatter=function(columns){
	return function(args...){
		let result='';
		for (let i=0;i<args.length();++i){
			result+=(args[i]+'').lpad(columns[i],' ');
		}
		return result;
	};
};

let format=String.formatter([10,5,5]);
println(format('n','x','y'));
println(format('---','---','---'));
for(let i=0;i<10;++i)
	println(format(i+1,Integer.random(100),Integer.random(100)));
```



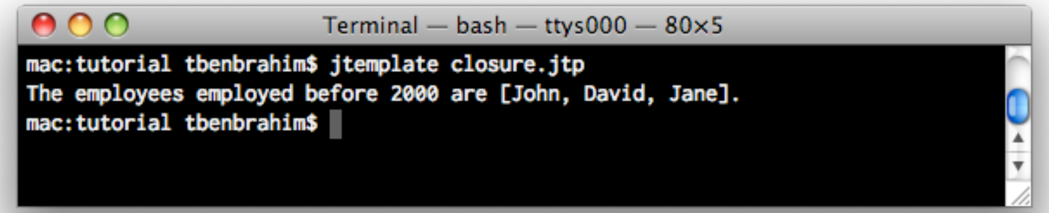
```
mac:tutorial tbenbrahim$ jtemplate formatter.jtp
         n    x    y
       ---  ---  ---
         1   55   70
         2   50   85
         3   54   76
         4   85   28
         5   73   34
         6   41   99
         7   20   31
         8   67   56
         9   82   90
        10   93   84
mac:tutorial tbenbrahim$ 
```

Jtemplate support function closures

We already saw an example of a function closure in the previous example. The function returned by `String.formatter` remembers the value of the `columns` array.

Here is another example. Here the closure is in the function returned by the `employedBefore` function, which remembers the value of the `year` variable after returning.

```
//sample included in closures.jtp

let Array.prototype.map=function(f){
	let result=[];
	foreach(el in this)
		result.push(f(el));
	return result;
};

let Array.prototype.filter=function(f){
	let result=[];
	foreach(el in this)
		if (f(el))
			result.push(el);
	return result;
};

let employees=[
	{name: 'John', startYear: 1998},
	{name: 'Mary', startYear: 2008},
	{name: 'David', startYear: 1999},
	{name: 'Pedro', startYear: 2002},
	{name: 'Al', startYear: 2002},
	{name: 'Jane', startYear: 1997}
];

let employedBefore=function(year){
	return function(employee){
		return employee.startYear<year;
	};
};

let year=2000;

println('The employees employed before ', year, ' are ',
	employees.filter(employedBefore(2000))
	.map(function(emp){return emp.name;}), '.');
```



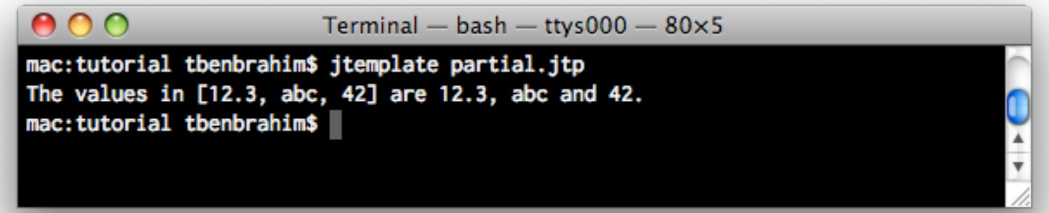
```
mac:tutorial tbenbrahim$ jtemplate closure.jtp
The employees employed before 2000 are [John, David, Jane].
mac:tutorial tbenbrahim$
```

Jtemplate supports partial function application

Partial function application allows a new function to be created from another function by partially applying some of the parameters and leaving the others free. In this example, we define a function `toDelimited` that formats an array with a delimiter separating each element, except for the last, which is separated by the value in the `last` parameter. We then define a function `prettyPrint`, derived from the partial application of `toDelimited` with ',' and ' and ', leaving the last argument free.

```
//sample include in partial.jtp

let toDelimited=function(delimiter,last, array){
	let result='';
	let lastIndex=arr.length()-1;
	for(let i=0;i<=lastIndex;++i){
		switch(i){
		case 0:
			result+=array[i];
			break;
		case lastIndex:
			result+=last+array[i];
			break;
		default:
			result+=delimiter+array[i];
			break;
		}
	}
```

```
        return result;
};

let prettyPrint=toDelimited(', ',' and ', @array);

let arr=[12.3, 'abc', 42];
println('The values in ',arr,' are ',prettyPrint(arr), '.');
```



```
mac:tutorial tbenbrahim$ jtemplate partial.jtp
The values in [12.3, abc, 42] are 12.3, abc and 42.
mac:tutorial tbenbrahim$
```

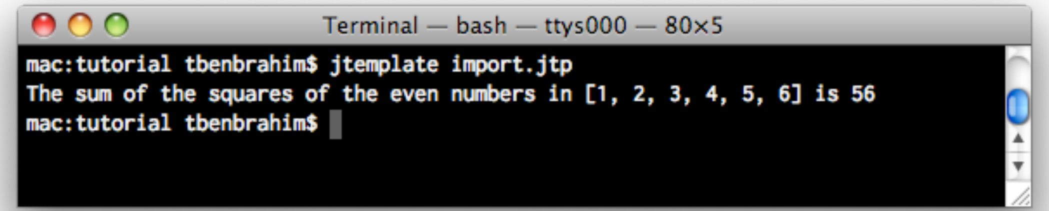
Jtemplate encourages modularity

Jtemplate supports importing declarations from external script files. For example, we have seen a few useful `Array` prototype functions, and we should include these in a separate file that we can import in our programs when we need them. For example, here is file `array.jtp`:

```
//sample included in array.jtp

let Array.prototype.map=function(f){
        let result=[];
        foreach(el in this)
                result.push(f(el));
        return result;
};

let Array.prototype.reduce=function(f, acc){
        let result=acc;
        foreach(el in this)
                result=f(el,result);
        return result;
};

let Array.prototype.filter=function(f){
        let result=[];
        foreach(el in this)
                if (f(el))
                        result.push(el);
        return result;
};
```

Now we can use the functions declared in `array.jtp` in new programs by importing the file.

```
//sample included in tutorial/import.jtp

import 'array.jtp';

let array=[1,2,3,4,5,6];

println('The sum of the squares of the even numbers in ', array, ' is ',
        array.filter(function(n){return n%2==0;})
             .map(function(n){return n*n;})
             .reduce(function(n,a){return n+a;},0)
);
```

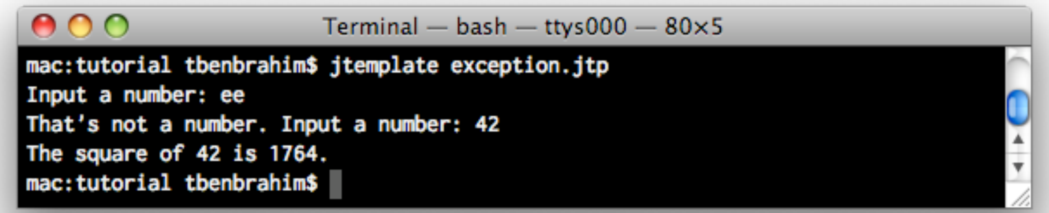


```
mac:tutorial tbenbrahim$ jtemplate import.jtp
The sum of the squares of the even numbers in [1, 2, 3, 4, 5, 6] is 56
mac:tutorial tbenbrahim$
```

Jtemplate supports structured exception handling

You can write functions that `throw` exceptions. You can also `try` an operation that may throw an exception, and `catch` the exception to perform recovery.

```
//sample included in exception.jtp
let inputNumber=function(){
        let result=readln().parseInt();
        if (result==void)
                throw("Not a number");
        return result;
};

while(true){
        try{
                print("Input a number: ");
                let n1=inputNumber();
                println("The square of ",n1," is ",n1*n1,".");
                break;
        }catch(e){
                print("That's not a number. ");
        }
}
```

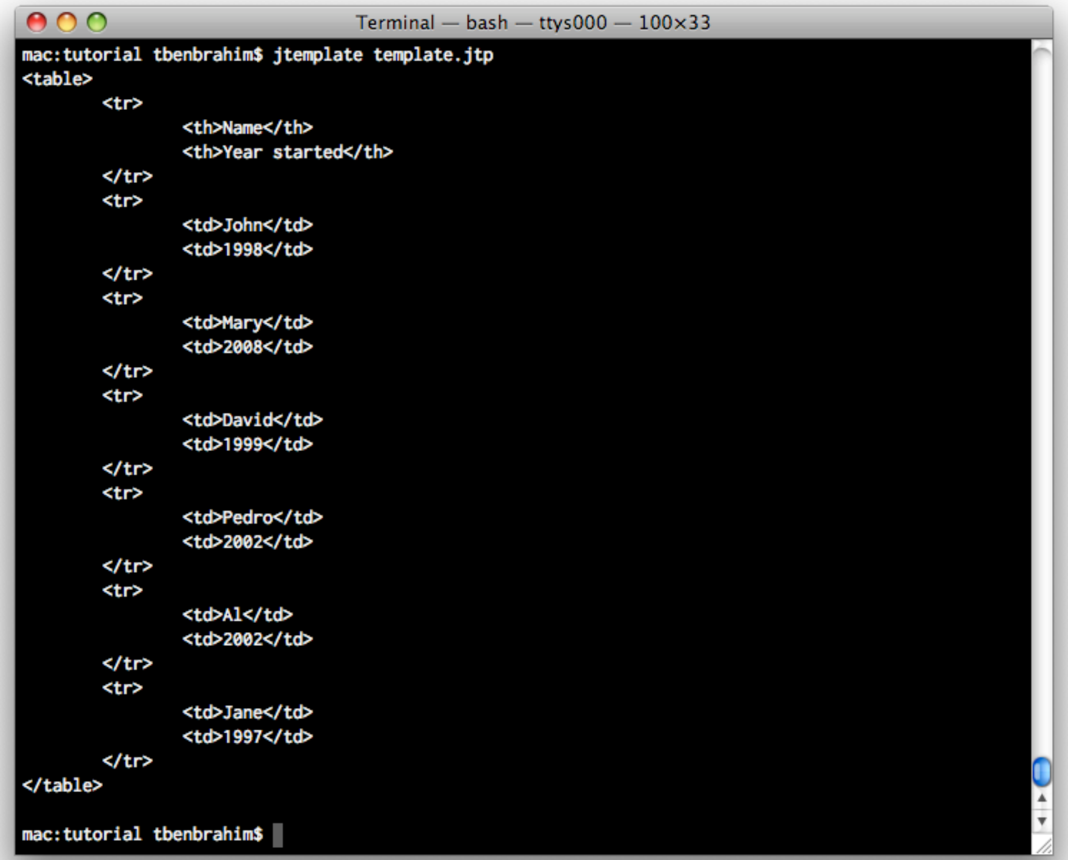


```
mac:tutorial tbenbrahim$ jtemplate exception.jtp
Input a number: ee
That's not a number. Input a number: 42
The square of 42 is 1764.
mac:tutorial tbenbrahim$
```

Jtemplate does templates

Template statements allow the generation of strings from a template definition and associated instructions.

```
//sample included in tutorial/template.jtp
template htmlTable{
             #<table>
             #      <tr>
header       #             <th>columnLabel</th>
             #      </tr>
row    #      <tr>
cell   #             <td>cellData</td>
row    #      </tr>
             #</table>
}

instructions for htmlTable(dataMap){
       header foreach(label in dataMap.labels): columnLabel=label;
       row foreach(dataArray in dataMap.data): ;
       cell foreach(element in dataArray): cellData=element;
}

let employees=[
       ['John', 1998],
       ['Mary', 2008],
       ['David', 1999],
       ['Pedro', 2002],
       ['Al', 2002],
       ['Jane', 1997]
];

println(htmlTable({labels: ['Name','Year started'], data: employees}));
```



```
mac:tutorial tbenbrahim$ jtemplate template.jtp
<table>
        <tr>
                <th>Name</th>
                <th>Year started</th>
        </tr>
        <tr>
                <td>John</td>
                <td>1998</td>
        </tr>
        <tr>
                <td>Mary</td>
                <td>2008</td>
        </tr>
        <tr>
                <td>David</td>
                <td>1999</td>
        </tr>
        <tr>
                <td>Pedro</td>
                <td>2002</td>
        </tr>
        <tr>
                <td>Al</td>
                <td>2002</td>
        </tr>
        <tr>
                <td>Jane</td>
                <td>1997</td>
        </tr>
</table>

mac:tutorial tbenbrahim$
```

A final example

This final example shows how to use Jtemplate functionality to build a rich model, then generate code from that model. It is by no means complete but meant to give you a taste of what is possible. Using the same technique, you could generate other artifacts such as SQL DDL, stored procedure and data access objects.

```
//sample included in tutorial/model.jtp

let Property={
  CARDINALITY_NONE: 0,
  CARDINALITY_ONE_TO_ONE: 1,
  CARDINALITY_ONE_TO_MANY: 2,
  CARDINALITY_MANY_TO_ONE: 3,
  CARDINALITY_MANY_TO_MANY: 4,

  stringToCardinality: function(s){
    switch(s){
    case '1-1': return 1;
    case '1-*': return 2;
    case '*-1': return 3;
    case '*-*': return 4;
    default: throw("Unknown cardinality "+s);
    }
  },

  prototype: {
    type: function(type){
      this.type=type;
      return this;
    },
    maxLength: function(len){
      this.maxLength=len;
      return this;
    },
    references: function(entity, cardinality){
      this.cardinality=Property.stringToCardinality(cardinality);
      if (entity.prototype!=Entity){
        throw(entity+' is not an entity');
      }
      let this.references=entity;
      return this;
    },
    required: function(){
      this.required=true;
      return this;
    },
    javaType: function(){
      if(this.references==void){
        switch(this.type){
          case 'string': return 'String';
          case 'integer': return 'int';
          case 'float': return 'double';
          case 'date': return 'Date';
          default: return 'XXX';
        }
      }else{
        switch(this.cardinality){
          case 0: throw("cardinality not specified for property "+this.name);
          case 1:
          case 3:
             return this.references.name.toFirstUpper();
          case 2:
          case 4:
            return "List<"+this.references.name.toFirstUpper()+">";
        }
      }
    }
  }
};

let Entity={
  prototype: {
    add: function(name){
      let prop= {
        prototype: Property,
        entity: this,
        name: name,
        type: 'string',
        maxLength: 100,
        references: void,
        cardinality: Property.CARDINALITY_NONE,
        required: false
      };
      this.properties.push(prop);
      return prop;
    }
  }
};

let Model={
  prototype: {
    create: function(applicationName){
      return {
        prototype: Model,
        name: applicationName,
        entities: []
      };
    },
    add: function(name){
      let entity={
        prototype: Entity,
        model: this,
        name: name,
        properties: []
      };
      this.entities.push(entity);
      return entity;
    }
  }
};

template javaBean{
  package     #package packageName;
          #
  class     #public class className{
          #
  fields      # private type propName;
          #
  constructor   # public className(){
  constructor   # }
          #
  getter      # public type getPropName(){
  getter      #   return propName();
  getter      # }
  getter      #
  setter      # public void setPropName(type propName){
  setter      #   this.propName=propName;
  setter      # }
  setter      #
  class     #}
}

instructions for javaBean(package, entity){
  package once: packageName=package;
  class once: className=entity.name.toFirstUpper();
  fields foreach(prop in entity.properties): type=prop.javaType(),
propName=prop.name;
  constructor once: className=entity.name.toFirstUpper();
  getter foreach(prop in entity.properties):
    type=prop.javaType(),PropName=prop.name.toFirstUpper(),
propName=prop.name;
  setter foreach(prop in entity.properties):
    type=prop.javaType(),PropName=prop.name.toFirstUpper(),
propName=prop.name;
}

let model=Model.create('Order application');

let customer=model.add('customer');
customer.add('lastName').type('string').maxLength(50).required();
customer.add('firstName').type('string').maxLength(50).required();

let item=model.add('item');
item.add('name').type('string').maxLength(100).required();
item.add('price').type('float').required();
item.add('quantity').type('integer').required();

let order=model.add('order');
order.add('orderDate').type('date').required();
order.add('customer').references(customer,'*-1').required();
order.add('items').references(item,'*-*').required();

let package='edu.columbia.w4115.demo';
foreach(entity in model.entities)
  println(javaBean(package,entity));
```

When run, it generates the following Java code to the screen (of course you would want to write to a file in a real application):

```
package edu.columbia.w4115.demo;

public class Customer{

        private String lastName;
        private String firstName;

        public Customer(){
        }

        public String getLastName(){
                return lastName();
        }
```

```
	public String getFirstName(){
		return firstName();
	}

	public void setLastName(String lastName){
		this.lastName=lastName;
	}

	public void setFirstName(String firstName){
		this.firstName=firstName;
	}

}

package edu.columbia.w4115.demo;

public class Item{

	private String name;
	private double price;
	private int quantity;

	public Item(){
	}

	public String getName(){
		return name();
	}

	public double getPrice(){
		return price();
	}

	public int getQuantity(){
		return quantity();
	}

	public void setName(String name){
		this.name=name;
	}

	public void setPrice(double price){
		this.price=price;
	}

	public void setQuantity(int quantity){
		this.quantity=quantity;
	}

}

package edu.columbia.w4115.demo;
```

```
public class Order{

	private Date orderDate;
	private Customer customer;
	private List<Item> items;

	public Order(){
	}

	public Date getOrderDate(){
		return orderDate();
	}

	public Customer getCustomer(){
		return customer();
	}

	public List<Item> getItems(){
		return items();
	}

	public void setOrderDate(Date orderDate){
		this.orderDate=orderDate;
	}

	public void setCustomer(Customer customer){
		this.customer=customer;
	}

	public void setItems(List<Item> items){
		this.items=items;
	}

}
```

Abed Tony BenBrahim
ba2305@columbia.edu

JTemplate

Language Reference Manual

Table of Contents

1 Introduction

Jtemplate is a dynamically typed language meant to facilitate the generation of text from template definitions. Jtemplate's support for prototypal inheritance, functions as first class values and a basic library permits the development of robust applications. While Jtemplate bears a strong resemblance to ECMAScript, there are number of significant differences that should be noted. Stronger type checking (for example, addition of a function and an integer, valid in ECMAScript, is not valid in Jtemplate) , mandatory declaration of variables before they are used, the different implementation of prototypal inheritance and singular implementation of varargs and partial function application, as well as improvements in scope visibility make Jtemplate quite distinct from ECMAScript.

2 Lexical Convention

2.1 Character set and whitespace

Programs in Jtemplate are written using the ASCII character set. Whitespace characters serve to separate language elements, except within strings and comments, and consist of spaces, tabs, carriage returns and newlines.

2.2 Comments

Multiline comments begin with the first character sequence /* and end with the first character sequence */ that is encountered.

Single line comments begin with the character sequence // and end at the end of the line

Example:

```
/*
 * This is a multiline comment
 */
let i=1; // this is single line comment
```

2.3 Identifiers

Identifiers begin with an uppercase or lowercase letter, an underscore (_) or a dollar sign ($) symbol. Following the first character, identifiers may optionally contain any number of uppercase or lowercase letters, digits 0 through 9, underscores or dollar signs. The following reserved keywords may *not* be used as identifiers:

```
break case catch continue default else false finally for foreach
function import in instructions let NaN once return use switch
throw true try template var Void when while
```

Identifiers in Jtemplate are case sensitive. The identifiers foo, Foo and FOO represent three different identifiers

2.4 Values

A value in Jtemplate assumes one of the following types: integer, float, string, Boolean, function, array, map, NaN or Void.

2.4.1 Integer

Integers are composed one or more digits, to form a whole number. A single optional minus (-) sign may precede the integer to negate its value. Integers may be in the range of -1073741824 to 1073741823 inclusive

2.4.2 Floating Point Number

Jtemplate supports IEEE-754 like double precision floating point numbers. Floating point numbers consist of:

- An optional minus sign (-) that negates the value
- A significand consisting of either or the sequence of both of :
    - An integer
    - A decimal point (.) followed by an integer, representing the fractional part
- An optional exponent part consisting of the character e, followed by an optional + or – sign, followed by an integer

Either the exponent or fractional part of the significand (or both) must be specified to form a valid floating point number. The following are examples of valid floating point numbers:

```
0.123 1.23 148.23e-32 1.e+12 1e+12
```

Jtemplate also defines three constants, Float.infinity, Float.negativeInfinity and Float.Nan that may be result of operations on floats.

2.4.3 String

Strings represent a sequence of ASCII characters. Strings start with either a single quote or double quote delimiter, and are terminated by the first non escaped matching delimiter. Strings may span several lines, and any newline spanned becomes part of the string. Non printable characters or string delimiters may be embedded in a string by using the following escape sequences:

`\b`	backspace
`\n`	Newline
`\r`	carriage return
`\t`	Tab
`\’`	single quote
`\”`	double quote
`\\`	Backslash

The following are examples of valid strings

```
'a string'
'a multiline
string'
"another string"
```

2.4.4 Boolean

Boolean values represent the logical values true and false. Boolean values consist of the two values `true` and `false`.

2.4.5 Function

A function represents a group of statements that can be invoked with an optional list of arguments and returns a value upon completion. Functions are defined with the `function` keyword, followed by a parenthesized list of zero or more identifiers, followed by a statement block:

`function (`*arglist*`)` *statement_block*

arglist is a comma separated list of zero or more identifiers.

statement_block begins with an opening brace (`{`), ends with a closing brace (`}`), and contains zero or more statements. Statements are fully described in section 4.1.

Example:

`function(){`*statements**`}`	a function with no arguments
`function(x,y){`*statements**`}`	a function with two arguments, x and y

The last identifier may optionally be followed by three periods, to indicate that the function accepts any number of values for the last argument, all of which will be placed in an array with the same name as the identifier.

Example:

`function println(items...){`*statement**`}` a function with any number of arguments (including none). When this function is invoked, any parameters will be passed in array `items`.

`function printat(x,y,strings...){`statement*`}` a function with at least two arguments x and y, and optionally any number of arguments that will be placed in an array named `strings`. For example, if this function is invoked with five arguments, the first will be bound to `x`, the second to `y`, the third, fourth and fifth will be added to an array bound to the identifier `strings`.

Functions may also be nested. If an inner function refers to an outer function's variable, a closure is created, that is the value of outer variable as it exists when the inner function is declared is remembered.

2.4.6 Array

An array represents an ordered list of values. Array values begin a left bracket (`[`), contain a comma delimited list of zero or more expressions (described in section 3), and end with a right bracket (`]`).

Example:

`[1,2,3]`	an array with 3 integer values
`[]`	an empty array
`[1,1.2,'abc',function(x,y){return x+y;}]`	an array with an integer, float, string and function value

2.4.7 Map

A map represents a container where a collection of values can each be associated with a key. Map values begin with an opening brace (`{`), contain a list of zero or more properties, and end with a closing brace (`}`). A property consist of an identifier, followed by a colon (`:`), followed by a value.

Example:

{}	An empty map
`{x:10,y:120,name:'test'}`	A map where `x` is associated to the integer 10, `y` to the integer 120 and `name` to the string 'test'
`{add:function(x,y){return x+y;}, subtract:function(x,y){return x-y;},x:10}`	A map with two functions, one mapped to the key `add` and one mapped to the key `subtract`, and an integer 10 mapped to the key `x`

Note than unlike other values which can be used anywhere where an expression can be used, the value for an empty map (`{}`) can only be used to the right of an assignment or declaration statement (Section 3.5), as function call arguments (Section 3.8), as property values in maps and as array elements.

2.4.8 NaN

`NaN` is the type of values that indicate that a value is not a number. There is a single eponymous value for this type, `NaN`. An example use of NaN is in a function that converts a float to an integer, to indicate that the passed in value could not be converted to a float. NaN is also used in integer operations to indicate that the result is invalid, such as when dividing by zero.

2.4.9 Void

Void is the type of values that indicate the absence of a value. There is a single eponymous value for this type, `void`. Void is the value returned from functions that do not explicitly return a value, and may also be used in other contexts to indicate the absence of a value.

3 Expressions

Values can be combined with operators and other expressions to form expressions.

3.1 Values

Values are expressions, and may be used interchangeably with expressions where expressions are indicated in the rest of this manual, noting the exception for empty maps described in section 2.8.

3.2 Variables and 'left hand side' expressions

Left hand expressions are locations where values can be stored. Left hand expressions can be variables identified by an identifier, members of maps (Section 3.8) or array members (Section 3.7). The following are examples of left hand side expressions:

`foo`	A variable named `foo`
`foo.x`	member x of map `foo`
`bar[10]`	The eleventh element of array `bar`

3.3 Arithmetic expressions

3.3.1 Binary arithmetic expressions

Arithmetic expressions operate on two expressions with an operator to create a new value. The format of an operation is *expression operator expression* where operator is one of

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo

The resulting value of evaluating an arithmetic expression depends on the type of the expressions and the operator, as shown in the table below (without regard to the ordering of the expressions):

Value 1 type	Value 2 type	Operator	Result
integer	Integer	`+ - * / %`	integer result of operation
float	float	`+ - * /`	float result of operation

float	integer		
string	any type	`+` only	concatenation of first value to second value, with non string values converted to strings

Any combination of value types and operators not listed above, such as adding two Booleans, results in a runtime error.

3.3.2 Unary arithmetic expressions

The minus (−) operator, when prefixing an expression, serves to negate the expression it precedes. The minus operator can only be applied to expressions that evaluate to integer or float values.

Example: `-1` `-a`

3.4 Comparison expressions

3.4.1 Binary comparison expressions

Binary comparison expressions compare two expressions with a comparison operator, and evaluate to a Boolean value indicating whether the comparison is true or false. The format of a comparison expression is:

expression operator expression

where *operator* is one of:

<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal

Allowable comparison expression types, operator and their result are as follows

Value 1 type	Value 2 type	Operator	Result
Integer	Integer	Any	Comparison of integer values
Float	Float	Any	Comparison of float values
Float	Integer		
String	Float	Any	comparison of first value to second value, with non string values converted to strings
	Integer		
	String		
Both types are Booleans, maps, arrays, functions, NaN or void		`==` and `!=` only	comparison of first value to second value, observing the semantics of equality described below.

Different types not listed above	== and != only	== always returns false != always returns true

There are special semantics of equality for maps, arrays and functions:

- Arrays a and b are equal if a and b have the same number of elements, and if for each index i in 0<=i<length, a[i]=b[i]
- Maps a and b are equal if a and b have the same keys, and if for each key k, a[k]=b[k]
- Functions a and b are equal if a and b have the same formal arguments, with the same name at the same position in the argument list, and have the same list of statements.

3.4.2 Ternary comparison expressions

Ternary expressions consist of an expression which when evaluated yields a Boolean value, causing one of two expressions to be evaluated:

Boolean-expression ? *expression_if_true* **:** *expression_if_false*

Example

```
a<10 ? 'small' : 'large' //evaluates to the string 'small' if a<10
                         // or 'large' if a>=10
```

3.5 Logical expressions

Logical expressions operate on two Boolean expressions with an operator, or on a single expression with a unary operator, to produce a single Boolean value. The format of a logical expression is:

expression and *expression*	true if both expressions evaluate to true, false otherwise
expression or *expression*	true if either expression evaluates to true, false otherwise
not *expression*	true if *expression* evaluates to false, false otherwise

Attempting to apply logical operators to expressions that are not of Boolean type results in a runtime error.

3.6 Declaration and assignment expressions

3.6.1 Declarations

Declarations bind a value *and its type* to a left hand side expression. Declarations consist of the keyword let (or its synonym var, a tribute to JTemplate's ECMAScript legacy), followed by a left hand side exception, followed by the equals sign, followed by an expression:

let lhs_expression = value
or
var lhs_expression = value

Example:

```
let x=1;        // binds identifier x to 1
let a=[1,2,3]; //binds identifier a to the array value [1,2,3]
let a[2]='abc'; // rebinds the third element of a to string 'abc'
```

Declaration expressions evaluate to the value of the expression on the right hand side of the equals sign, so that one can chain declarations.

Example:

```
let x=let y=let z=0;  // declare x y and z as integers
                      // initialized to 0
```

3.6.2 Assignment

Assignment changes the value of a previous declared left hand side expression. Once a left hand side expression has been declared, it is bound to a type and can only be assigned a value of the same type, unless it is re-declared with another value of a different type.

Example:

```
let a=[1,2,3]; //binds a to the array value [1,2,3]
a[2]=10 + a[2]; // assigns 12 to the third element of a
```

Attempting to assign a value to a left hand side expression that has not been previously declared, or attempting to assign a value of a type different than the type of the left hand side expression, results in runtime error.

Assignment expressions evaluate to the value of the expression on the right hand side of the equals sign, so that one can chain assignments.

Example:

```
x=y=z=0;  //set x y and z to 0
```

3.6.3 Combined arithmetic operation and assignment expressions

Combined arithmetic operation and assignment expressions perform an operation then assign a result to the left hand side expressions. These expressions exist in both binary and unary form.

3.6.3.1 Binary form

The format of a binary combined arithmetic operation and assignment expression is

lhs_expression operator = expression

where *lhs_expression* is valid left hand side expression. The *lhs_expression* and the *expression* are evaluated with the *operator* using the same semantics described in section 3.3, with the result assigned to the *lhs_exception*. The expression evaluates to the result of the assignment.

Example:

```
a+=2  // equivalent to a=a+2
```

3.6.3.2 *Unary form*

Unary combined arithmetic operation and assignment expressions increment or decrement a left hand side expression, assigning the result to the left hand side expression. These expressions exist in both postfix and prefix forms, which affect the value evaluated by the expression:

Syntax	Operation	Evaluates to
`++ lhs_expression`	Increment `lhs_expression`	`lhs_expression` after it has been incremented
`lhs_expression ++`	Increment `lhs_expression`	`lhs_expression` before it has been incremented
`-- lhs_expression`	Decrement `lhs_expression`	`lhs_expression` after it has been decremented
`lhs_expression --`	Decrement `lhs_expression`	`lhs_expression` before it has been incremented

Example:

```
let a=0
let b = ++a    // both a and b are 1
let b = a++    // a is 2, b is 1, a's value before it was incremented
a++            // a is 3
++a            // a is 4
```

Note that the prefix form is more efficient and should be used in favor of the postfix form when there is no semantic difference, as in the last two examples above.

3.7 Index expressions

Index expressions are used to access a member of an array or a map value, or an expression that evaluates to an array or a map. Index expressions begin with an expression, followed by an opening bracket (`[`), contain an expression and end with a closing bracket (`]`):

expression `[`*expression*`]`

When applied to a map expression, the index can evaluate to a string or integer. When applied to an array, the index must evaluate to an integer. In both cases, the index must exist in the map or array when the index expression is used on the right hand side of an expression or when used on the left side of an assignment.

Example:

```
let arr=[1,2,'abc']; //define array arr
let b=arr[0];        // declare b and initialize
                     // with the first element of arr
let c=1;
b=arr[c];                //assign the second element of arr to b

let m={a:124,point:{x:10,y: 120}};  // define map m
let p=m['a'];                       // initialize p with member a of m
let name='x';
let q=m['point'][name];             // initialize q with member x of
                                    // map point in map m
```

3.8 Member expressions

Member expressions are used to access a member of a map value, or an expression that evaluates to a map value . Member expressions begin with an expression, followed by a period (`.`) followed by an identifier

expression.expression

Example:

```
let m={a:124,point:{x:10,y: 120}};  // define map m
let p=m.a;                          // initialize p with member a of m
let q=m.point.x;                    // initialize q with member x of
                                    // map point in map m
```

3.9 Function Calls

3.9.1 Function Invocation

Function calls cause the statements in a previously defined function value to be executed, first initializing the function value's formal parameters with the arguments passed in to the function call. The format of a function call is an expression that evaluates to a function value, followed by an opening parenthesis, a comma separated list of zero or more expressions, followed by a closing parenthesis:

expression (*expression_list*)

Example:

```
let sum=function(a,b){return a+b; }; // declare a function
let a=sum(1,2);                   // assign 1 to a, 2 to b then
                                  // evaluate the statements in sum
println('Hello',' ','World');     //  calls function assigned to println
                                  //   defined as function(items...){}
function(a,b){return a*b;}(10,20)// calling a function definition
```

Calling a function with fewer or more arguments than are required results in a runtime error.

Function calls evaluate to the value returned by a `return` statement (Section 4.3.4.3), or to `Void` if a `return` statement is not executed before leaving the function body.

3.9.2 Partial application

Partial application, also known as currying, allows a new function to be defined from an existing function, binding some of the curried function's parameters to existing values and leaving others unbound. Partial application in JTemplate is invoked by calling the function to be curried, and indicating which parameters should remain unbound by preceding them with an at symbol (`@`). The result is a new function with the unbound parameters as the new parameters. For example, consider the function declaration of `sum` below:

```
let sum=function(a,b){return a+b; };
```

We create a new function `increment` from `sum` by leaving one the variables unbound and binding the other one to 1. We can then invoke increment with one parameter.

```
let increment = sum(@value,1);
let b=increment(10);
```

Partial application can also operate on varagrs. Consider the function println defined as

```
let println=function(values...){ library_code };
```

We create a new function print2 that assigns the first element of values to the string `'>'`

```
let print2=println('>',@values...);
```

The print2 function has the signature `function(values...)` and when invoked, the first element of values will contain `'>'` and the succeeding elements will contain any additional parameters passed in the values `vararg`. As a result, any line that is printed by the `print2` function will be prefixed with the '>' character.

3.10 Grouping

Expressions can be grouped with parentheses to control the order of evaluation.

Example:

```
(2+3)*10     // evaluates to 50
2+(3*10)     // evaluates to 32
```

3.11 Operator precedence

When expressing are not grouped as shown in section 3.10, Jtemplate uses operator precedence rules to determine the order in which operations are performed. For example, in the expression `2+3*10`, the interpreter needs a rule to decide whether to evaluate the multiplication or addition first.

Expressions are evaluated using the operator precedence rules listed below, with the highest precedence listed at the top:

Expression	Operators	Section
Member expressions, index expressions	`.` `[]`	3.7, 3.8
Function calls	`()` `()[]` `().`	3.9
Postfix expression	`++` `--`	3.6.3.2
Prefix expression	`++` `--`	3.6.3.2
Negation	`-`	3.3.2
Logical Not	`!`	3.5
Multiplication/Division/Modulo	`*` `/` `%`	3.3.1
Addition/Subtraction	`+` `-`	3.3.1
Comparison	`<` `<=` `>` `>=` `==` `!=`	3.4.1
Logical operator	`&&` `||`	3.5
Ternary comparison	`?:`	3.4.2
Assignment/Declaration	`=`	3.6
Arithmetic assignment	`*=` `/=` `%=`	3.6.3
Arithmetic assignment	`+=` `-=`	3.6.3

When operators with the same precedence are encountered, they are evaluated from left to right, except for negation, logical not , arithmetic assignment, assignment and declaration which evaluates from right to left.

Example:

```
println(7-2-2);//associates to the left, equivalent to (7-2)-2
let a=let b=1; //associates to the right, equivalent to let a=(let b=1)
println(a-=b-=2); //associates to the right, equivalent to a-=(b-=2)
```

4 Statements

A program is composed of zero or more statements, all of which are described in this section.

4.1 Statement Blocks

A statement block is a group of statements with an inner scope (Scope is discussed in section 5). A statement block starts with an opening brace (`{`) , contain zero or more statements, and end with a closing brace(`}`).

Example

```
{                          // a statement block
      let a=1;
      let b=2;
}
{}                         // an empty statement block
```

4.2 Expressions

Expressions can be used as statements, when they are suffixed with a semicolon:

expression ;

Example:

```
let a=1;        //this expression assigns 1 to a
1+2;            //this expression has no side effect
;               //an empty expression, again with no side effect
```

Any expression can be used as a statement, except for the empty map (`{}`), as noted in section 2.3.7. Expressions which do not cause a value to be assigned or evaluate a value that is used by a subsequent statement, such as second and third expression in the example above have no side effect and serve no purpose.

4.3 Iteration statements

An iteration statement causes a statement or statement block to be executed repeatedly until a condition has been met.

4.3.1 `for` loops

For loops consist of an optional initializer expression, an optional conditional expression, and an optional counting expression, arranged as follows:

`for (` *initializer_expression?* `;` *conditional_expression?*`;` *counting_expression?* `)`
    statement_or_statement_block

Execution of a `for` statement proceeds as follows:

1. The initializer expression is evaluated, if is present.
2. The conditional expression is evaluated if it is present. If it is not present, the missing expression is substituted with `true`. If the expression evaluates to false, the for loop statement is completed and execution proceeds to the next statement.
3. If the expression evaluates to true, the statement or statement block is executed one time.
4. The counting expression is evaluated, if it is present
5. Execution proceeds to step 2.

This control flow can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
for(var i=0; i<10; ++i){
      println('Hello');
      println('World!');
}
```

4.3.2 while loops

While statements consist of a conditional expression and a statement or statement block arranged as follows:

`while` (*conditional_expression*)
        statement_or_statement_block

Execution of a `while` statement proceeds as follows:

1. The conditional expression is evaluated. If it evaluates to false, the `while` loop statement is completed and execution proceeds to the next statement.
2. If the expression evaluates to true, the statement or statement block is executed one time.
3. Execution proceeds to step 1.

This control flow can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
let i=0;
while(i<10){
      println('Hello');
      println('World!');
      ++i;
}
```

4.3.3 foreach loops

`foreach` statements execute a statement or statement block once for each element in an expression that evaluates to a map or an array. The syntax of a `foreach` statement is:

`foreach` (*element* `in` *collection*) *statement_or_statement_block*

where element is a left hand side expression, collection is an expression that evaluates to a map or array type.

If the collection expression evaluates to an array, the element expression is assigned with successive elements of the array, starting with the first element and proceeding in sequence, before each execution of the associated statement or statement block. Execution of the statement ends after execution of the statement or statement block for the last element of the array.

If the collection expression evaluates to a map, the element expression is assigned with successive property values of the map, presented in no particular order, before each execution of the associated statement or statement block. Execution of the statement ends after execution of the statement or statement block for the last value in the map.

Attempting to execute a `foreach` statement with an expression that cannot be assigned or with a type that is not a collection results in a runtime error. The control flow for a `foreach` statement can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
let a=[1,'abc',2];
foreach(el in a) println(el);
let b={a:1,b:'xyz'};
foreach (el in b) println(el);
```

Output:

```
1
abc
2
xyz
1
```

4.3.4 Altering loop statements control flow

The iteration in the loop statements described earlier in this section can be interrupted or modified by the following instructions.

4.3.4.1 *break statement*

The `break` statement causes the loop statement to terminate immediately. Execution resumes at the statement following the loop statement.

Example: This “inifinite” loop exists after 10 iterations.

```
let i=0;
for (;;){
	++i;
	if (i==10) break;
}
```

Using a `break` statement outside of a loop statement or `switch` statement (Section 4.4.2) results in a runtime error.

4.3.4.2 *continue statement*

The continue statement causes execution to be skipped for all statements in a statement block following the invocation of continue, and causes the loop statement to proceed to the next iteration.

Example: Only even numbers are printed

```
for(var i=0;i<10;++i){
	if (i%2==1) continue;
	println(i);
}
```

Using a `continue` statement outside of a loop statement results in a runtime error.

4.3.4.3 `return` statements

The `return` statement, used exclusively inside a function definition's function body, causes the execution of a function call to stop and immediately evaluate to the expression specified by the return statement. The syntax of a return statement is

`return` *expression?* ;

where the expression is optional. If the expression is not specified, `return` returns `Void`.

Example:

```
let sum=function(x,y){return x+y;}

let foo=function(){
	let x=10;
	for (var i=0;i<10;++i){
		x+=i;
		if (x>10) return i;
	}
}
```

Invoking `return` outside of a function body results in a runtime error.

4.4 Conditional statements

A conditional statement conditionally executes a statement or statement block based on the evaluation of an expression.

4.4.1 `if` statement

The `if` statement executes one of two statements or statement blocks based on the evaluation of a conditional expression. An `if` statement starts with the `if` keyword followed by a parenthesized conditional expression, followed by a statement or statement block, and optionally followed by the `else` keyword and another statement or statement block:

`if (`*cond-expression*`)` *statement_or statement_block*

`if (`*cond-expression*`)` *statement_or statement_block* `else` *statement_or statement_block*

An `if` statement's conditional expression is evaluated, and if it evaluates to true, the succeeding statement or statement block is executed. If it evaluates to false, and an `else` clause is present, the statement or statement block following the else keyword is executed. If the expression does not evaluate to a Boolean expression, a runtime error occurs.

Example: prints `a is equal to b`

```
let a=let b=10;
if (a>b)
        println('a is greater than b');
else
        if (a<b)
                println('a is smaller than b');
        else
                println('a is equal to b');
```

4.4.2 switch statement

The switch statement executes statements following a case label containing an expression that is equal to an expression being compared. A `switch` statement consists of the `switch` keyword, followed by a parenthesized expression, followed by a statement block. The statement block consists of the statements described in this section, and special `case` statements, consisting of the `case` keyword followed by an expression followed by a colon, or the `default` keyword followed by a colon:

```
switch(expression){
        case case_expression:
                statements
        case case_expression:
                statements
        default:
                statements
}
```

When a `switch` statement is executed, the expression is first evaluated. Then the statement block is executed, by looking for the first `case` statement where the evaluation of the associated `case` expression is equal to the `switch` value, using the rules of equality defined in section 3.4.1. The special `default` case statement, if encountered, matches any value. If a matching case statement is encountered, all statements following the case statement are executed, until the end of the statement block (any case statement encountered is not evaluated), or until a `break` or `return` instruction interrupts the flow of control.

Example:

```
let arith=function(x,op,y){
        switch(op){
        case '+':
                println(x,op,y,' is ',x+y);
                break;
        case '*':
                println(x,op,y,' is ',x*y);
                break;
        default:
                println('Only addition and multiplication are supported');
        }
};
```

Note that using a case statement outside of a switch statement results in a parsing error.

4.5 exception handling & recovery statements

Exception handling and recovery statements provide a structured way to deal with errors.

4.5.1 throw statement

The `throw` statement interrupts the normal flow of control by raising an error. Following the execution of a `throw` statement, execution of the program terminates, or if the `throw` statement is executed inside a `try/catch` block (Section 4.5.2), execution proceeds to the catch clause of the `try/catch` block. A `throw` statement consists of the `throw` keyword, followed by an expression, followed by a semicolon:

`throw` *expression*`;`

Example:

```
let safeDivide=function (x,y){
      if (y==0)
            throw 'Division by 0';
      return x/y;
};
```

4.5.2 try/catch block

A `try/catch` block allows a statement to catch exceptions thrown by a `throw` statement, or internally by the Jtemplate interpreter. A `try/catch` block starts with the `try` keyword, followed by a statement block (the try block), followed by the `catch` keyword, followed by a parenthesized identifier, followed by another statement block (the catch block):

```
try{
      statements
}catch(identifier){
      statements
}
```

A `try/catch` block executes by executing the statements in the try block. If an error does not occur, the catch block is never executed. If an exception is thrown, the exception is assigned to the catch block identifier and the statements in the catch block are executed.

Example:

Without a `try/catch` block, executing the following statements result in a runtime error, because a Boolean cannot be added to a float. This causes the program to terminate.

```
let safeAdd=function(x,y){
      return x+y;
};
println('the result is ',safeAdd(true,1.2));
```

With a `try/catch` block, the error is intercepted inside the function and an alternate value (`Void`) is returned:

```
let safeAdd=function(x,y){
	try{
		return x+y;
	}catch(e){
		return Void;
	}
};
println('the result is ',safeAdd(true,1.2));
```

4.5.3 try/finally block

A `try/finally` block ensures that a block of code always execute, even in the presence of an exception that would normally immediately terminate program execution. A `try/finally` block starts with the `try` keyword, followed by a statement block (the try block), followed by the `finally` keyword, followed by another statement block (the finally block):

```
try{
	statements
}finally{
	statements
}
```

When a `try/finally` block is encountered, all statements in the try block are executed. Then the statements in the finally block execute, even if the execution of the try block caused an exception to be thrown. If an exception was thrown by the try block, it is handled after the finally block has executed.

Example:

```
try{
	println('entering the try block');
	let a=1.2+true;
}finally{
	println('entering the finally block');
}
```

Output:

```
entering the try block
entering the finally block
At line 5 in file lrm_samples.jtp: uncaught exception
Interpreter.Interpreter.EIncompatibleTypes("float", "boolean")
```

Note that even though an exception is thrown in the try block, the code in the finally block executes before execution terminates. `try/finally` blocks are frequently nested inside of `try/catch` blocks, to ensure that a specific action is performed prior to handling the error.

Example:

```
try{
	try{
		println('entering the try block');
		let a=1.2+true;
	}finally{
		println('entering the finally block');
	}
}catch(e){
	println('Something went wrong: ',e);
}
```

Output:

```
entering the try block
entering the finally block
Something went wrong:
Interpreter.Interpreter.EIncompatibleTypes("float", "boolean")
```

4.6 Importing definitions

As a program grows larger, if may be useful to separate the program's code into separate modules. It may also be useful to develop modules of functionality than can be reused in other projects. The `import` keyword enables this, by allowing declarations to be imported from another program file. The `import` statement consists of the `import` keyword, followed by a string specifying the file's location, followed by a semicolon:

`import '`*path/to/file*`';`

The path that is specified can either be relative or absolute. When an `import` statement is encountered, the path is normalized to an absolute path. Jtemplate then determines if the file has previously been imported, and if it has already been loaded, execution of the statement terminates and execution proceeds to the next statement. If the file has not previously been imported, the file is loaded and executed, executing *only* declaration expression statements and import statements, ignoring all other statement types.

Since imported files are only loaded one time, circular references are allowed. For example, foo.jtp can import bar.jtp, and bar.jtp can import foo.jtp, since foo.jtp will not be imported again.

Example:

myfile.jtp contains:

```
let multiplicationSign=function(a,b){
      if (a==0 || b==0)
            return 0;
      else
            if ((a>0 && b>0) || (a<0 && b<0))
                  return 1;
            else if ((a>0 && b<0)||(a<0 && b>0))
                  return -1;
};
```

```
println('This statement will not be executed when the file is
imported');
```

sample.jtp contains:

```
import 'myfile.jtp';

println('The sign is ',multiplicationSign(-19,-20));
```

myfile.jtp declares the function `multiplicationSign`. Any file that imports myfile.jtp can use the function as if it had been included in the same file.

4.7 Template statements

Template statements allow the generation of strings from a template definition and associated instructions.

4.7.1 template statement

A template defines a labeled block of text than can later be manipulated by processing instructions (Section 4.7.2). A template statement consists of the `template` keyword, followed by an identifier specifying the template name, followed by an opening bracket (`{`) , then zero or more line specifications, and closed by a closing bracket (`}`). A line specification consists of an optional identifier or integer that serves as a label for the processing instructions, followed by a hash sign (#) indicating the start of line, followed by text. A line specification ends when the end of line is reached:

```
template template_name {
      line_specification*
}
```

where line_specification is *label?* #text

Example:

```
template htmlTable{
           #<table>
           #<tr>
header     #<th>columnLabel</th>
           #</tr>
row        #<tr>
cell       #<td>cellData</td>
row        #</tr>
           #</table>
}
```

Note that nesting can be defined by repeating the labels. In the example above, the line labeled `cell` is nested between two lines labeled `row`. Specifying an illegal nesting structure, such as A B A B, where B is nested inside A, and A is nested inside B, results in a runtime error.

4.7.2 instructions statement

4.7.2.1 Instruction definition

An instruction statement defines how a template definition is turned into a string. An instruction statement starts with the `instructions` keyword, followed by the `for` keyword, followed by an identifier referencing the name of a template definition, followed by a parenthesized list of arguments using the same convention as for a function definition, followed by an opening brace (`{`), a list of zero or more processing instructions terminated with a semicolon, followed by a closing brace (`}`):

`instructions for` *template_name* `(`*arglist*`) {`
processing_instructions`;`
`}`

A processing instruction consists of a label matching a label in the template definition, followed by a processing condition, followed by a colon (**:**), followed by a comma separated list of one more replacement specification.

label processing_condition **:** *replacement_specifications*

A processing condition takes one of the following forms:

Processing Condition	Action
`always`	always perform the replacements specified in the replacement specifications
`when (`*condition_expression*`)`	Perform the replacements specified in the replacement specification only if the *condition_expression* evaluates to true
`foreach(`*element* `in` *collection*`)`	Perform the replacements specified in the replacement specification, looping through each element of the map or array *collection*.
`foreach(`*element* `in` *collection*`)` `when (`*condition_expression*`)`	Perform the replacements specified in the replacement specification, looping through each element of the map or array *collection*, only if the *condition_expression* evaluates to true for the given iteration of the loop

Finally a replacement condition consists of an identifier followed by an equals sign (=) followed by an expression. The identifier is treated as a string. The expression is evaluated, then any text in the labeled template line matching the identifier is string is replaced with the value of the expression.

Example: an `instructions` statement for the template defined in 4.7.1

```
instructions for htmlTable(dataMap){
header foreach(label in dataMap.labels): columnLabel=label;
row foreach(dataArray in dataMap.data): ;
cell foreach(element in dataArray): cellData=element;
}
```

Note that any variable defined in a replacement condition is also available to nested definitions. In the example above, `dataArray` is introduced in the `row` definition, then used in the nested `cell` definition.

4.7.2.2 Invoking template instructions

Template instructions are invoked in the same manner as a function call, using the same semantics. For example, if `people` is defined as follows

```
let people={labels: ['Name','Age'], data: [['John',42], ['Mary',38]] };
```

invoking the instructions for `htmlTable` with `people` as an argument

```
let text=htmlTable(people);
```

results in the `text` variable being assigned the string

```
<table>
<tr>
<th>Name</th>
```

```
<th>Age</th>
</tr>
<tr>
<td>John</td>
<td>42</td>
</tr>
<tr>
<td>Mary</td>
<td>38</td>
</tr>
</table>
```

4.7.3 Replacement Methodology

It is useful to think of the replacements as occurring in parallel rather than serially. An example will better serve to illustrate this point.

Consider the input string '`foo bar`' given in template definition, and the replacement specification '`foo=x, bar=y`' given in a template instruction. If x has the value 'bar' and y has the value 'foo', performing the replacement serially would yield the string 'bar bar' after the first replacement and 'foo foo' after the second replacement. When the replacement is performed 'in parallel', the first foo is replaced with bar by the first replacement and bar is replace with foo by the second replacement, yielding the final string 'bar foo'.

To accomplish this, the offset of all replacements for each replacement condition in calculated before any replacements are performed. If any overlapping regions are detected, a runtime error occurs. As each replacement occurs in order, the offsets of the subsequent replacements are adjusted based on whether characters where added or replaced by the previous replacement.

Example: For the input string 'foo bar foo bar', and the replacement 'foo=x, bar=y', the following offsets are calculated, then sorted:

- Replace starting at 0 of length 3 with value of x
- Replace starting at 4 of length 3 with value of y
- Replace starting at 8 of length 3 with value of x
- Replace starting at 12 of length 3 with value of y

If x evaluates to 'a', after the first replacement, the subsequent offsets will be adjusted by 2 positions leftward, since the input string is now 'a bar foo bar'

- Replace starting at 2 of length 3 with value of y
- Replace starting at 6 of length 3 with value of x
- Replace starting at 10 of length 3 with value of y

5 Scope

Scope defines which variables are visible to the statement being executed. Scope is hierarchical, with declarations in an inner scope not visible to statements in an outer scope. Variables declared in an inner scope disappear as soon as the inner scope is exited.

5.1 Program level scope

When a program starts, all declarations occur in the top level scope. Any function declared in a scope can see functions declared in the same scope, even if the definition of the second function occurs after the definition of the first definition.

Example:

```
let odd=function(n){
      return n==0? false: even(n-1);
};

let even=function(n){
      return n==0? true: odd(n-1);
};
```

In the mutual recursion example above, the `odd` function body accesses the `even` function, even though it is declared after the `odd` function.

5.2 Statement block scope

A statement block, whether alone, or following a statement such as if or foreach, starts a new scope. Variables declared in the statement block are not visible to statements outside the statement block. Variables declared in a statement block with the same name as a variable declared outside the statement block is a separate variable, even if the name is the same.

Example:

```
let a=1;
print(a,' ');
{
      print(a,' ');
      let a=2;
      print(a,' ');
}
println(a,' ');
```

outputs 1 1 2 1

Note that the declaration of variable `a` inside the statement block does not affect the originally defined variable `a`, which maintains its value of 1.

5.3 Function scope

JTemplate is a lexically scoped language. As such, any variable reference inside a function statement that does reference an argument name or a variable declared inside the function body resolves to a variable in the same scope as the function definition.

Example:

```
let printX=function(){println(x);};

let x=0;
{
	let x=1;
	printX();
}
```

outputs 0, because the `printX` function sees the variable `x` defined in the same scope, not the value of `x` in the inner scope when `printX` was invoked.

6 Object Oriented Constructs

Jtemplate supports calling methods defined in map, in a manner reminiscent of an object oriented language. For example, the following map contains a function `test`, which can be invoked using a member expression:

```
let x={a:1, test: function(){println('hello world');}};
x.test();
```

The member notation has been extended to support calling member functions for non map types, and implementing flexible function dispatch for map types. For example, the runtime library exposes many operations on string types that are invoked as if they were a member function of the string itself. For example, to calculate the length of a string `myString`, the following statements would be invoked:

```
let myString='hello world';
let len=myString.length();
```

Note that a member function is invoked on a string.

6.1 Prototypes

6.1.1 Semantics for non map types

When a member function is invoked on an expression, the type of the expression is first determined. If the expression is not a map, the function is looked up in the type's prototype, which is simply a map containing functions for that type. Each type has a prototype, with zero or more functions, as shown below:

Type	Prototype
Integer	Integer.prototype
Float	Float.prototype
String	String.prototype
Boolean	Boolean.prototype
Array	Array.prototype
Map	Map.prototype
Void	Void.prototype
NaN	NaN.prototype
Function	Function.prototype

If the function is found in the prototype, it is invoked, passing the caller as an argument named `this`. Note that `this` is not part of a function's formal arguments. A type's prototype can be extended at run time, providing methods that can be invoked for all expressions of that type.

Example 1: extending the array type with a join method that concatenates all the elements as a string, and in this example, outputs `12abc1.2`.

```
let Array.prototype.join=function(){
      let result='';
      foreach(el in this) result+=el;
      return result;
};

let a=[1,2,'abc',1.2];
println(a.join());
```

Example 2: extending the array type with a map function, which takes a function as an argument and returns a new array with the function applied to each element of the array, and a clone function, which uses the map function to return a copy of the array:

```
let Array.prototype.map=function(f){
      let newArray=[];
      foreach(element in this)
            newArray.push(f(element));
      return newArray;
};
var Array.prototype.clone=function(){
      return this.map(function(x){return x;});
};
```

6.1.2 Semantics for map types

Since the map type has the ability to store member functions, or declare its own member map named `prototype`, resolving member functions for maps is more complex, and proceeds in the following order:

1. Try to find the definition for the function in the map. If it is found, it is invoked as a normal function call, in particular, `this` is assigned to Void. (`map.func()`, essentially a static invocation)
2. If the map has a map member named `prototype`, try to find the function definition in the map and invoke it, passing the caller in a variable named `this`. (`map.prototype.func()`)
3. If the map has a map member named `prototype`, and it contains a map with a prototype member, try to find the function definition in the map and invoke it, passing the caller in a variable named `this`. (`map.prototype.prototype.func()`)
4. Lookup the function is the `Map.prototype` map. If it is found, invoke it, passing the caller in a variable named `this`. (`Map.prototype.func()`)

Example:

```
let m={
         foo:         function(){print('hello');},
         prototype: {bar: function(){this.foo();println(' again');}}
      };
m.foo();println(); //using case 1
m.bar();           //using case 2
let a=m.keys();    //using case 4
```

The interesting use cases comes with case 3 above, which allows for a single level of inheritance.

Example:

```
let Foo={prototype: {print: function(){println('your value is
',this.value); }}};
let m={value:10, prototype: Foo};
m.print();                //using case 3
```

Here the declaration of `Foo` creates a new type that exposes function `print`. Any map with a prototype assigned to `Foo` can invoke functions of Foo's prototype as if it were a member function of the map.

A more involved example: The built in library exposes a Date map with a single(static) function, now(), which returns a map mimicking a tm structure, and with its prototype set to Date. This allows Date to be extended to add a toString() member function.

```
var Date.days =
['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday'
];

var Date.prototype={};

var Date.prototype.toString=function(){
    var offset='';
    if (this.gmtOffset>0){
        offset='+'+this.gmtOffset;
    }else{
        offset=this.gmtOffset+'';
    }
    return Date.days[this.dayOfWeek]+'
'+this.month+'/'+this.dayOfMonth+'/'+
        this.year+'
'+this.hour+':'+(this.minute+'').padLeft(2,'0')+':'+
        (this.second+'').padLeft(2,'0')+' (GMT'+ offset+')';
};
```

where `padLeft` is defined as

```
let String.prototype.padLeft=function(len,pad){
    var result=this+''; //cast to string
    while (result.length()<len){
        result=pad+result;
    }
    return result;
};
```

The expression `Date.now().toString()` evaluates to a string such as 'Sunday 6/21/2009 10:27:14 (GMT-5)'

6.2 Supporting multiple levels of inheritance

As noted in the previous section, prototypal inheritance only supports one level of inheritance. However, adding support for multiple levels can easily be accomplished. We begin by defining an object as a map and its prototype map and add a prototype function extend. Object will serve as the base class for all objects:

```
let Object={
      prototype: {
            extend: function(){
                  let obj={prototype:{}};
                  foreach(key in this.prototype.keys())
                        let obj.prototype[key]=this.prototype[key];
                  return obj;
            }
      }
};
```

As an example we then create a class Foo that extends from object, and introduces a new method foo():

```
let Foo=Object.extend();
let Foo.prototype.foo=function(){
      println('foo!');
};
```

We then define a new class Bar that extends Foo, and a new class Fun that extends Bar overriding foo in both classes and introducing a new method bar in the Bar class. We implement the calling of the inherited method by using the library function `Function.prototype.apply`, which lets us call an arbitrary map member method while passing an arbitrary `this` object

```
let Bar=Foo.extend();
let Bar.prototype.foo=function(){
      print('bar ');
      Foo.prototype.foo.apply(this);
};

let Bar.prototype.bar=function(){
      println('bar!');
};

let Fun=Bar.extend();
let Fun.prototype.foo=function(){
      print('fun ');
      Bar.prototype.foo.apply(this);
};
```

Finally, we create objects of each type, and invoke their methods. The method of constructing new objects is a little contrived, a matter that will be dealt with in the next section:

```
let foo={prototype: Foo};
let bar={prototype: Bar};
let fun={prototype: Fun};
print('foo.foo(): ');foo.foo();
print('bar.bar(): ');bar.bar();
print('bar.foo(): ');bar.foo();
print('fun.bar(): ');fun.bar();
print('fun.foo(): ');fun.foo();
```

which outputs:

```
foo.foo(): foo!
bar.bar(): bar!
bar.foo(): bar foo!
fun.bar(): bar!
fun.foo(): fun bar foo!
```

6.3 Implementing constructors

Jtemplate does not natively support a `new` constructor. However, the Object definition above can trivially be extended to provided a new() method:

```
let Object={
      prototype: {
            extend: function(){
                  let obj={prototype:{}};
                  foreach(key in this.prototype.keys())
                        let obj.prototype[key]=this.prototype[key];
                  return obj;
            },
            new: function(){
                  return {prototype: this };
            }
      }
};
```

Now the foo and bar objects in the previous example can be constructed using this new method

```
let fun=Fun.new();
fun.bar();
fun.foo();
```

which outputs:

```
bar!
fun bar foo!
```

In the same way that methods were overridden, constructors can be overridden. Overriding constructors lets us add parameters to the constructor and as importantly, define fields for our object.

Example:

```
let Point=Object.extend();
let Point.prototype.new=function(x,y){
      let point= Object.prototype.new.apply(this);
      let point.x=x;
      let point.y=y;
      return point;
};
let Point.prototype.print=function(){
      println('x: ',this.x,', y: ',this.y);
};

let p=Point.new(42,10);
print('p.print(): ');p.print();

let ThreeDPoint=Point.extend();
let ThreeDPoint.prototype.new=function(x,y,z){
      let point= Point.prototype.new.apply(this,x,y);
      let point.z=z;
      return point;
};
let ThreeDPoint.prototype.print=function(){
      print('z: ',this.z,', ');
      Point.prototype.print.apply(this);
};

let p3=ThreeDPoint.new(5,42,10);
print('p3.print(): ');p3.print();
```

Output:

```
p.print(): x: 42, y: 10
p3.print(): z: 10, x: 5, y: 42
```

Note how the ThreeDPoint class inherited members x and y, by calling the overridden Point constructor in its constructor and passing x and y.

7 Built in Library

7.1 Built in variables

7.1.1 Command line arguments

When a program is run, the program name and any arguments after the program name are placed in an array named `args`. `args`[0] will contain the program name, `args`[1] the first argument if present and so on.

7.1.2 Environment variables

When a program is run, the environment variable names and values are placed in a map named `env`, with environment variable names as keys and environment variable values as key values.

7.2 System Library

The system library contains functions to deal with Jtemplate native types, as well as functions to deal with the operating system environment.

Signature	Description
`Array.prototype.push(value)`	Adds `value` to the end of the caller array, returns `Void`
`Array.prototype.pop()`	Removes the last element from the caller array and returns the element that was removed
`Array.prototype.length()`	Returns the length of the caller array as an integer
`Map.prototype.remove(key)`	Removes `key` and its associated value from the caller map, returns `Void`.
`Map.prototype.contains(key)`	Returns true if `key` exists in the caller map, false otherwise.
`Map.prototype.keys()`	Returns an array with the caller's keys
`Integer.random(upperBound)`	Returns a pseudo random number between 0 and `upperBound-1` inclusive
`Float.prototype.round()`	Returns an integer with the float caller rounded up if the fractional part >0.5, rounded down otherwise
`Date.now()`	Returns a map representing today's date, with the following keys and values: `gmtOffset`: offset from GMT (integer) `second`: number of seconds in the time 0-59 (integer) `minute`: number of minutes in the time 0-59 (integer) `hour`: number of hours in the time 0-23 (integer) `dayofMonth`: number of day in the month 1-31 (integer) `month`: number of month in year 1-12 (integer) `year`: today's year (integer) `dayOfWeek`: today's day index relative to the week, starting at 0 for Sunday 0-6 (integer) `dayOfYear`: today's day index relative to the start of the year, starting at 0 for the first day of the year 0-366

	(integer) `dst`: true if daylight savings time is in effect, false otherwise (Boolean)
`Function.prototype.apply(this,args...)`	Calls caller map member function, passing `this` as parameter this, and passing any additional arguments specified in `args`
`typeof(value)`	Returns a string representing the type 's value, one of `string, integer, boolean, float, function, map, array, NaN, Void`
`System.command(command)`	Executes the external command specified by `command`, waits for execution to complete and returns an integer representing the exit code of the command.
`exit(exitcode)`	Causes the program to exit with exit code `exitcode`, which must be in the range -127 to 127 inclusive.
`Debug.dumpSymbolTable(incl)`	Dumps the symbol table to stdout, including library functions if `incl` is true
`Debug.dumpStackTrace()`	Dumps the current stack trace to stdout

7.3 String Library

The String library contains functions to perform string manipulation.

Signature	Description
`String.prototype.toUppercase()`	Returns a new string with every character in the string caller uppercased
`String.prototype.toLowercase()`	Returns a new string with every character in the string caller lowercased
`String.prototype.toFirstUpper()`	Returns a new string with the string caller's first letter uppercased
`String.prototype.toFirstLower()`	Returns a new string with the string caller's first letter lowercased
`String.prototype.length()`	Returns the length of the string caller as an integer
`String.prototype.charAt(index)`	Returns a new string with the character at the string caller's position indicated by `index`, with 0 indicating the first character. Throws a runtime error if the index is less than 0 or greater or equal to the string's length
`String.prototype.indexOf(substr)`	Returns an index representing the leftmost position of substring `substr` in the string caller, or -1 if the substring is not found.
`String.prototype.substr(st,len)`	Returns the substring in the string caller starting at position `st` of length `len`. Throw
`String.prototype.startsWith(substr)`	Returns true if the string caller starts with the substring `substr`, false otherwise
`String.prototype.endsWith(substr)`	Returns true if the string caller ends with the substring `substr`, false otherwise
`String.prototype.replaceAll(`	Returns a new string with every occurrence of

`substring, replacement)`	`substring` in the string caller replaced with `replacement`.
`String.prototype.split(sep)`	Returns an array containing the substrings in the string caller that are delimited by `sep`
`String.prototype.parseInt()`	Returns an integer parsed from the string caller, or Void if the string does not represent a valid integer
`String.prototype.parseFloat()`	Returns a float parsed from the string caller, or Void if the string does not represent a valid float

7.4 I/O Library

The I/O Library contains functions to deal with input and output to the console and file system, as well as functions to manipulate the file system.

Signature	Description
`print(value...)`	Prints the arguments to stdout
`println(value...)`	Prints the arguments to stdout, followed by a newline after the last argument
`readln()`	Returns a string with a line read from stdin
`File.openForWriting(handle, filename)`	Opens file `filename` for writing and associates the file handle with string `handle`.
`File.openForReading(handle, filename`	Opens file `filename` for reading and associates the file handle with string `handle`.
`File.close(handle)`	Closes a previously opened file, using the string `handle` associated with the file handle.
`File.write(handle,value...)`	Writes the `value` arguments (automatically cast to a string) to a file associated with the string `handle`
`File.writeln(handle,value...)`	Writes the `value` arguments (automatically cast to a string) to a file associated with the string `handle`, then writes a newline after the last argument is written.
`File.readln(handle)`	Returns a string read from a file associated with the string `handle`, previously opened for reading
`File.eof(handle)`	Returns true if the file previously opened for reading associated with the string `handle` has reached end of file.
`File.exists(filename)`	Returns true if `filename` exists, false otherwise
`File.delete(filename)`	Deletes `filename`, returns true if the file was successfully deleted, false otherwise
`File.rename(oldname,newname)`	Renames the file named `oldname` to name `newname`, returns true if the file was renamed, false otherwise.
`Directory.exists(dirname)`	Returns true if the directory named `dirname` exists, false otherwise
`Directory.delete(dirname)`	Deletes the directory named `dirname`, returns true if the directory was successfully deleted, false

	otherwise
`Directory.list(dirname)`	Returns an array containing every file and directory contained in the directory `dirname`
`Directory.create(dirname)`	Creates directory `dirname`, returns true if the directory was successfully created, false otherwise

Example:

This program takes two arguments, a source text file and a destination text file, and copies the source text file to the destination text file.

```
if (args.length() != 3) {
      println('Usage: ', args[0], ' source_file destination_file');
      exit(-1);
}
if (!File.exists(args[1])) {
      println('File ', args[1], ' does not exist.');
      exit(-1);
}
File.openForReading('in', args[1]);
try {
      File.openForWriting('out', args[2]);
      try {
            let lines = 0;
            while (!File.eof('in')) {
                  let s = File.readln('in');
                  File.writeln('out', s);
                  ++lines;
            }
            println(lines, ' lines copied');
      } finally {
            File.close('out');
      }
} finally {
      File.close('in');
}
```

Abed Tony BenBrahim
ba2305@columbia.edu

JTemplate Development Process and Architectural Guide

Table of Contents

1 Development Process

1.1 Language Design Process

“Language design is a very personal activity and each person brings to a language the classes of problems that they'd like to solve, and the manner in which they'd like them to be solved.” Alfred Aho[1]

The Jtemplate language seeks to be first and foremost an approachable tool for generating code from models. Having had a fair amount of experience with a fairly popular and robust code generation tool, Open Architecture Ware (OAW), I sought to avoid the perceived failings of the product that make it inaccessible to all but the top 1% of corporate developers. The corporate development landscape is one where boilerplate code is plentiful, and would be well served by using code generation as a tool to increase productivity, by generating up to 75% of the code and leaving the developers free to focus on the domain specific aspects of the software. But to use OAW, a developer first has to develop a DSL using *XText*, using a variant of an *Antlr* grammar, add constraints to the model using a purely functional language called *Check*, define the template using a templating engine called *Xpand* and define the substitutions using another purely functional language called *Xtend*. While OAW is a brilliant solution, it remains vastly underutilized, since most corporate developers, if they were ever exposed to functional language in their education (and most were not or have no recollection of ever being exposed), face too great a learning curve to make the product worth using.

The choice of using a variant of JavaScript for the language seemed like a natural one. After all, the syntax would already familiar to the great majority of corporate developers who program in Java, C#, C and C++, as well to most web developers. Despite its inauspicious start, when it was used solely as a scripting language to coordinate the interaction of applets with a web page or perform simple client side validation, JavaScript has evolved over the years to become a robust language, supporting the standard constructs of declarative and object oriented programming as well as functional constructs such as function closures and higher order functions. It is becoming much more common for JavaScript to be used to develop rich web applications with tens of thousands of lines of code, or even to be used as the target of compilation, as is the case with the Google Web Toolkit or Lazlo frameworks

Nevertheless, JavaScript still suffers from major flaws that make development in the language error prone. First and foremost is the permissiveness with which the language is interpreted. For example, JavaScript will interpret the addition of a function definition and an integer without so much as raising a warning, or will compare an empty array to an empty map and report that the empty array is smaller!. Jtemplate seeks to prevent this by imposing much stricter constraints on allowable operations. Since JavaScript is a dynamic language, where types are assigned to values and not to variables, and since

[1] http://www.computerworld.com.au/article/216844/-z_programming_languages_awk?pp=3

JavaScript allows variables to appear in code without having been previously been initialized, it is impossible to provide much more than cursory type checking. Jtemplate attempts to correct this by requiring all variables to be declared and initialized before use, with the goal of implementing type checking.

While JavaScript supports functions with a variable number of arguments, it does not do so in an explicit manner. Simply calling a function with more parameters than the number of declared formal parameters makes all extra parameters *varargs*. Moreover, it is not an error in JavaScript to call a function with fewer parameters than the number of formal arguments. Jtemplate eliminates this source of potential error by requiring all function invocations to be made with the same number of parameters as the number of formal arguments, and by providing an explicit mechanism to define *vararg* arguments. Jtemplate also provides for defining new functions by using partial function application.

Finally, Jtemplate provides a mechanism for generating text from a template definition and a set of processing instructions. Experience with a variety of templating solutions led to two important goals. First and foremost, there had to be a clear separation of the template text from any processing instructions, since prior experience has clearly shown that interspersing template text and markup instructions quickly leads to unmaintainable code. Second, templates and processing instructions should be an integral part of the language, not tucked away in a separate file. Jtemplate achieves both of these goals, to allow for experimentation as to whether this method of defining templates shows any promise in actual practice.

The delivered Jtemplate implementation meets most of its design goals. Where deficiencies exist, they are noted in the lessons learned section.

1.2 Project Schedule

The initial project plan was devised so as to complete as much passing functionality as possible in the first half of course, leaving the second half of the course open to experimentation and potentially risky refactoring, with the risk of failure mitigated by the ability to return at any time to the code developed in the first half of the course.

Development proceeded in two phases, over a period of forty days. In the first phase, running from June 3 to July 4, a basic interpreter was developed. It was slow, did not have a runtime AST, but passed a suite of over one hundred tests and ran some fairly sophisticated examples. Only the template functionality and a couple of minor function points were missing from this version. In the second phase, running from July 26 to August 4, a complete refactoring of variable storage was undertaken, leading to a dramatic five fold increase in performance. Further optimizations were undertaken, such as function inlining, constant folding, some modest variable liveness analysis and for loop optimizations, which led to a further nearly two fold increase in performance. Missing features were implemented and additional tests were also added to increase code coverage.

As can be seen from the table below, the first phase of development proceeded for the most part on schedule. The second part started late, probably due to the confidence that with Ocaml, making the necessary changes would not take as long as initially anticipated.

Week Ending	Actual	Milestone
May 30, 2009	June 3, 2009	Setup environment, preliminary lexer, parser and AST, Project Proposal
June 6, 2009	Jun 6, 2009	Finalize parser and AST, symbol table implementation, Interpreter for basic constructs (no templates)
June 13, 2009	June 6, 2009	Testing framework, Language Reference Manual
June 20, 2009	Jun 22, 2009	Language Reference Manual
June 27, 2009	Jun 13, 2009	Built in Library Implementation
July 4, 2009	July 26, 2009	Interpreter for template instructions
July 11, 2009	July 26, 2009	Semantic analysis (detect error in template labels, unused variable warnings, etc...)
July 18, 2009	N/A	Open
July 25, 2009	N/A	Open
August 1, 2009	N/A	Final Report
August 8, 2009	August 9, 2009	Final Report

Original Project Plan, with actual dates added

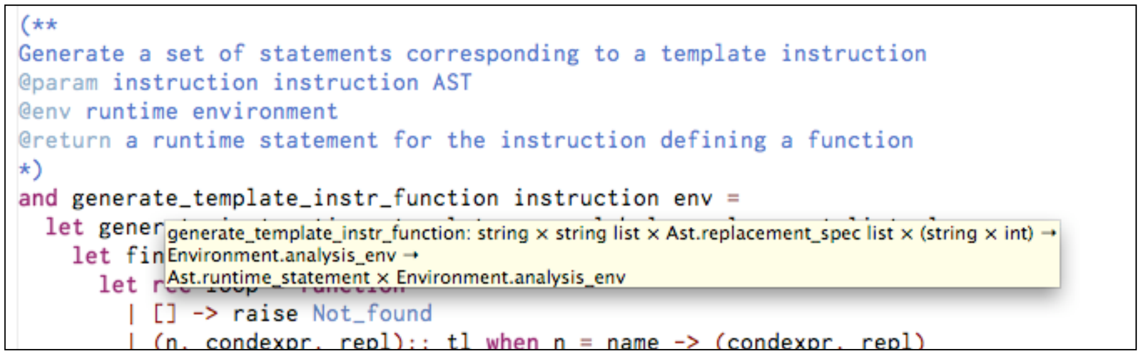
The following table lists major milestones in the development of the JTemplate interpreter, with information extracted from the SVN commit logs.

Revision	Date	Milestone
3	Jun 3 2009	Initial lexer and parser
10	Jun 6 2009	Symbol table completed
14	Jun 6 2009	OCaml Unit testing framework done
17	Jun 7 2009	Type checking of assignments
33	Jun 8 2009	Implementation of function calls
37	Jun 13 2009	Built in library completed
46	Jun 13 2009	Added varargs
63	Jun 13 2009	Added line numbers and filename to AST
64	Jun 14 2009	Added support for imports
66	Jun 14 2009	Added error reporting with stack traces
75	Jun 14 2009	Partial function application implemented
80	Jun 18 2009	Refactored AST and parser to correctly support member expressions.
89	Jun 20 2009	Object oriented examples added

106	Jun 26 2009	Makefile for code coverage using bisect
118	Jul 4 2009	END OF PHASE 1 DEVELOPMENT
121	Jul 26 2009	Replaced symbol table with more efficient storage, implemented runtime AST
124	Jul 26 2009	Implemented template functionality
126	Jul 30 2009	Constant folding, replacement of non-reassigned variable references with constant value
131	Aug 1 2009	Added support for closures
137	Aug 2 2009	Function inlining
141	Aug 3 2009	Optimize For loops
145	Aug 4 2009	Command line arguments processing, final version

1.3 Development Tools

All development was performed using OcaIDE on Eclipse 3.4. OcaIDE proved to be an ideal tool to develop in an initially unfamiliar language. Background camlp4 processing identified syntax errors such as missing parentheses without the need for compilation. More importantly, the ability to hover over a symbol and see its definition, as demonstrated below, was invaluable to quickly resolve problems due to mismatched arguments in function invocations. Finally, OcaIDE's code formatting functionality automatically gave the code a consistent look.



```
(**
Generate a set of statements corresponding to a template instruction
@param instruction instruction AST
@env runtime environment
@return a runtime statement for the instruction defining a function
*)
and generate_template_instr_function instruction env =
  let gener
    let fin
      let r
        | [] -> raise Not_found
```

```
generate_template_instr_function: string × string list × Ast.replacement_spec list × (string × int) →
Environment.analysis_env →
Ast.runtime_statement × Environment.analysis_env
```

Ocamlbuild was used for the build environment, obviating the need to modify a make file every time a module's dependencies were modified. Google Code's Subversion repository was used for version control[2], with the Subclipse plug-in providing seamless integration into Eclipse. An internally developed Ocaml unit testing framework was used in the first phase of development, and a unit testing framework developed in the Jtemplate scripting language was used in the second phase of development. Bisect 1.0 was used to provide code coverage during the executions of the test suites. Additionally, the Jtemplate interpreter is able to print out a script's AST and symbol table when invoked with the -parsetree command line option, providing useful information while debugging the interpreter. While not used extensively, the Shark profiler (OS X 's alternative to gprof) was used near the end of development to identify two hotspots, one

[2] http://ojtemplate.googlecode.com/svn/

dealing with the generation of stack trace information and the other with floating point handling, both of which once remediated, resulted in a significant increase in performance. Finally, ocamldoc was used to generated module documentation for the project. As a whole, this development environment provided all the needed features to quickly resolve any problem that arose, leaving most of the development time for feature development rather than debugging.

1.4 Testing

Testing focused on correctness, and in the second phase, performance was added as a consideration.

1.4.1 Correctness testing

Early in the development process, before there was an interpreter to run test scripts, the need arose to test the various components that were being developed. A basic Ocaml unit testing framework was developed to meet this need. The tests were arranged in test suites, represented by an array of unit tests, with each unit test a function taking no parameters and returning a boolean. The following fragment shows the start of a test suite and two of the more than hundred and twenty unit tests that were developed before the interpreter was running.

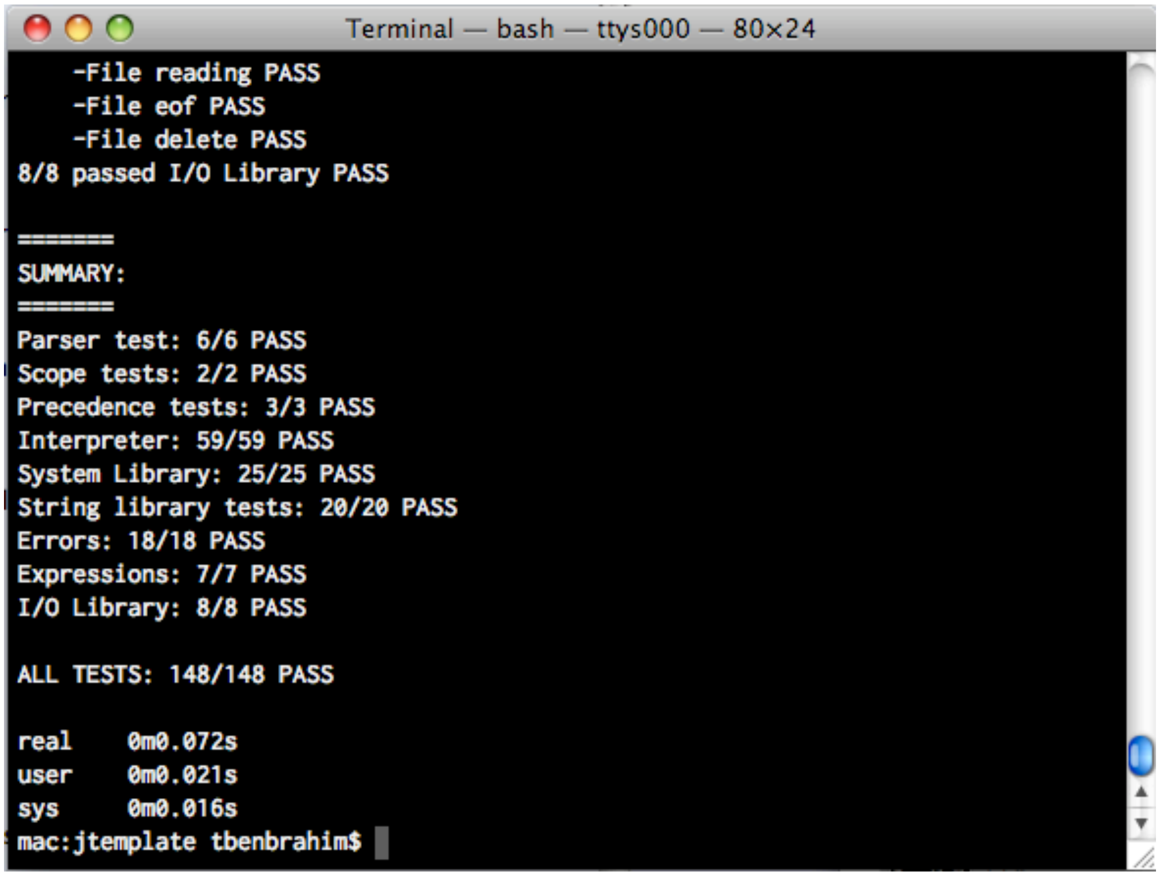
```
let test_suite = ("Interpreter",[
  ("integer arithmetic?", fun() ->
    let v1 = Value(IntegerValue(7)) in
    let v2 = Value(IntegerValue(2)) in
    let s = SymbolTable.initialize() in
    Interpreter.evaluate_expression (BinaryOp(v1, Plus, v2)) s = IntegerValue(9) &&
    Interpreter.evaluate_expression (BinaryOp(v1, Minus, v2)) s = IntegerValue(5) &&
    Interpreter.evaluate_expression (BinaryOp(v1, Times, v2)) s = IntegerValue(14)  &&
    Interpreter.evaluate_expression (BinaryOp(v1, Divide, v2)) s = IntegerValue(3) &&
    Interpreter.evaluate_expression (BinaryOp(v1, Modulo, v2)) s = IntegerValue(1));
  ("integer division/modulo by 0 should return NaN", fun() ->
    let v1 = Value(IntegerValue(7)) in
    let v2 = Value(IntegerValue(0)) in
    let s = SymbolTable.initialize() in
    Interpreter.evaluate_expression (BinaryOp(v1, Divide, v2)) s = NaN &&
    Interpreter.evaluate_expression (BinaryOp(v1, Modulo, v2)) s = NaN
);
...
```

As more functionality was developed and as the interpreter became mature enough to run Jtemplate scripts, writing unit tests that defined the AST to test by hand became too tedious, and this approach was dropped. A unit testing framework was developed in Jtemplate. Using the same basic structure as the Ocaml test framework, tests were grouped into test suites comprising many unit tests. The exception handling features of Jtemplate facilitated testing conditions that should generate errors. The following code fragment shows two of the Jtemplate unit tests in the Interpreter test suites. In total, more than 140 unit tests were developed, and are included in appendix C.

```
var testSuite={
    description: 'Interpreter',
    tests:
```

```
    [
        {
            description: 'Post increment test',
            test: function(){
                 var a=1;
                 var b=a++;
                 return b==1 && a==2;
            }
        },{
            description: 'Post decrement test',
            test: function(){
                 var a=1;
                 var b=a--;
                 return b==1 && a==0;
            }
        }
...
```

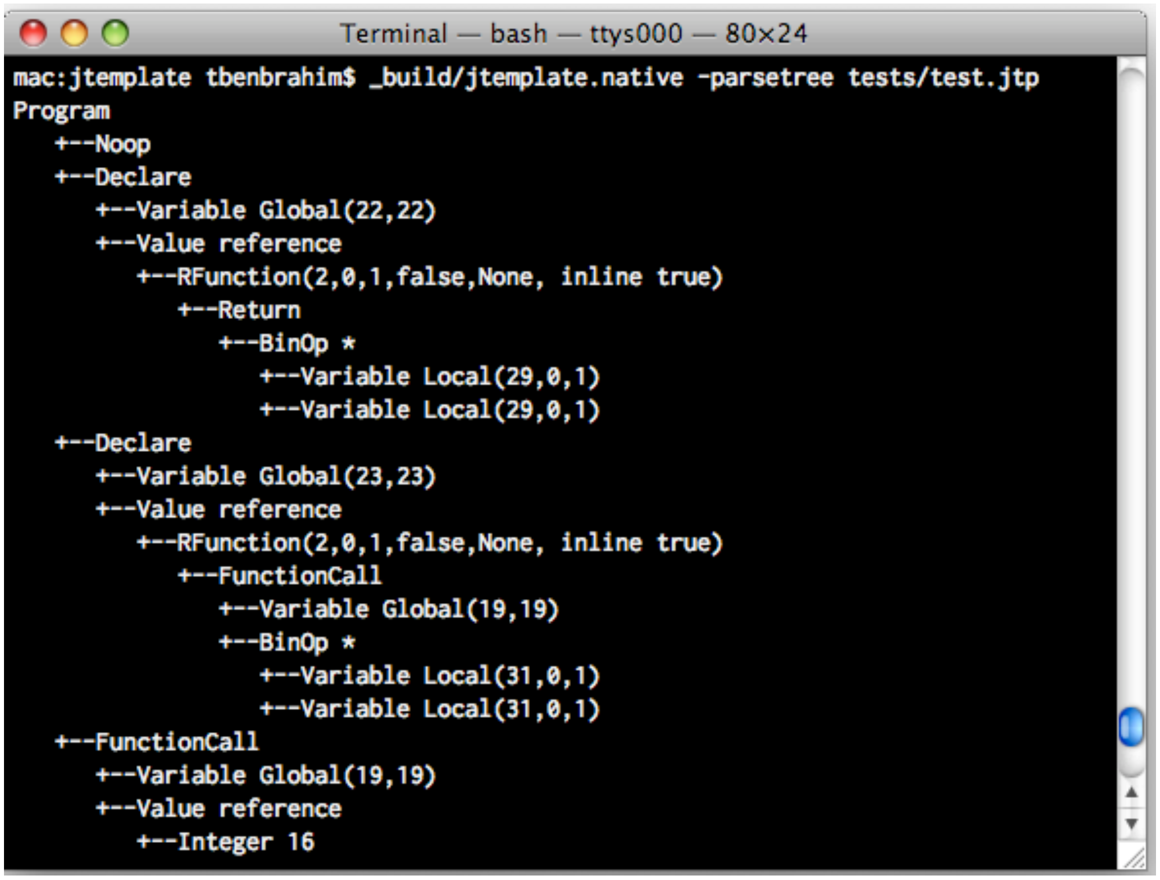
Running all the test suites produces a result for each unit test, a summary for each test suite and an overall total. Running all tests takes less than 100 ms, so there was no impediment to running the test suites after every change, a practice that helped uncover errors right after an erroneous change was introduced, at the time when a defect is easiest to correct.



```
    -File reading PASS
    -File eof PASS
    -File delete PASS
8/8 passed I/O Library PASS

========
SUMMARY:
========
Parser test: 6/6 PASS
Scope tests: 2/2 PASS
Precedence tests: 3/3 PASS
Interpreter: 59/59 PASS
System Library: 25/25 PASS
String library tests: 20/20 PASS
Errors: 18/18 PASS
Expressions: 7/7 PASS
I/O Library: 8/8 PASS

ALL TESTS: 148/148 PASS

real    0m0.072s
user    0m0.021s
sys     0m0.016s
mac:jtemplate tbenbrahim$
```

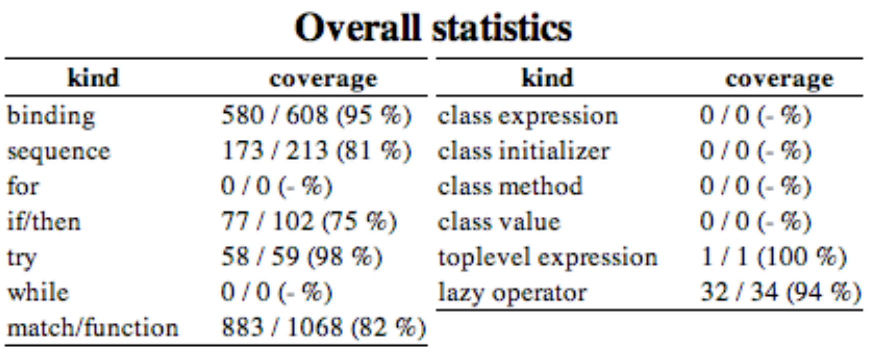
Tests were selected in one of three ways. First, tests were developed as each function point was implemented, usually one test for correctness and one or more tests to ensure that error conditions are detected. For example, to test popping a value from an array, there is a test for correct operation, and two tests for incorrect operation where the interpreter is expected to raise an exception: one test for popping a value from an empty array and one test for popping a a value from a variable that is not an array. Second, when in the course of running unit tests or sample scripts, a bug was uncovered, a test was written, usually before the bug was fixed. Finally code coverage information was generated with bisect and analyzed to detect sections of code needing more testing. At the time of writing, coverage was achieved for 86% for the code, as summarized in the tables on the following page.

In addition to automated testing, it was also sometimes necessary to perform visual inspection of the AST, especially in the second development phase when the runtime AST was being transformed and optimized. To facilitate inspection of the AST, a command line switch (`-parsetree`) was added to the interpreter, to pretty print the runtime AST at the point just prior to interpretation, as well as to display a symbol table showing all declared variables. This feature was essential to ensure that the AST transformations provided the desired result.



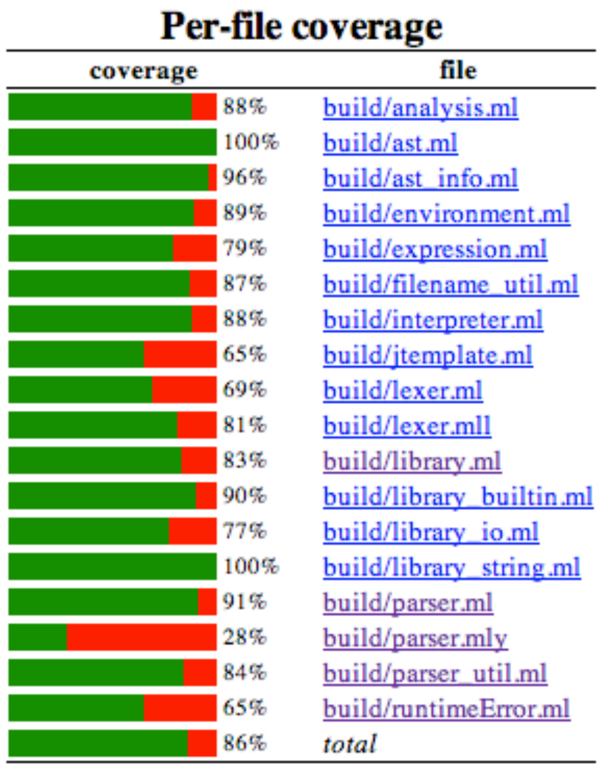
```
mac:jtemplate tbenbrahim$ _build/jtemplate.native -parsetree tests/test.jtp
Program
   +--Noop
   +--Declare
      +--Variable Global(22,22)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--Return
               +--BinOp *
                  +--Variable Local(29,0,1)
                  +--Variable Local(29,0,1)
   +--Declare
      +--Variable Global(23,23)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--FunctionCall
               +--Variable Global(19,19)
               +--BinOp *
                  +--Variable Local(31,0,1)
                  +--Variable Local(31,0,1)
   +--FunctionCall
      +--Variable Global(19,19)
      +--Value reference
         +--Integer 16
```

Running the Jtemplate interpreter with the -parsetree switch



Overall statistics

kind	coverage	kind	coverage
binding	580 / 608 (95 %)	class expression	0 / 0 (- %)
sequence	173 / 213 (81 %)	class initializer	0 / 0 (- %)
for	0 / 0 (- %)	class method	0 / 0 (- %)
if/then	77 / 102 (75 %)	class value	0 / 0 (- %)
try	58 / 59 (98 %)	toplevel expression	1 / 1 (100 %)
while	0 / 0 (- %)	lazy operator	32 / 34 (94 %)
match/function	883 / 1068 (82 %)		

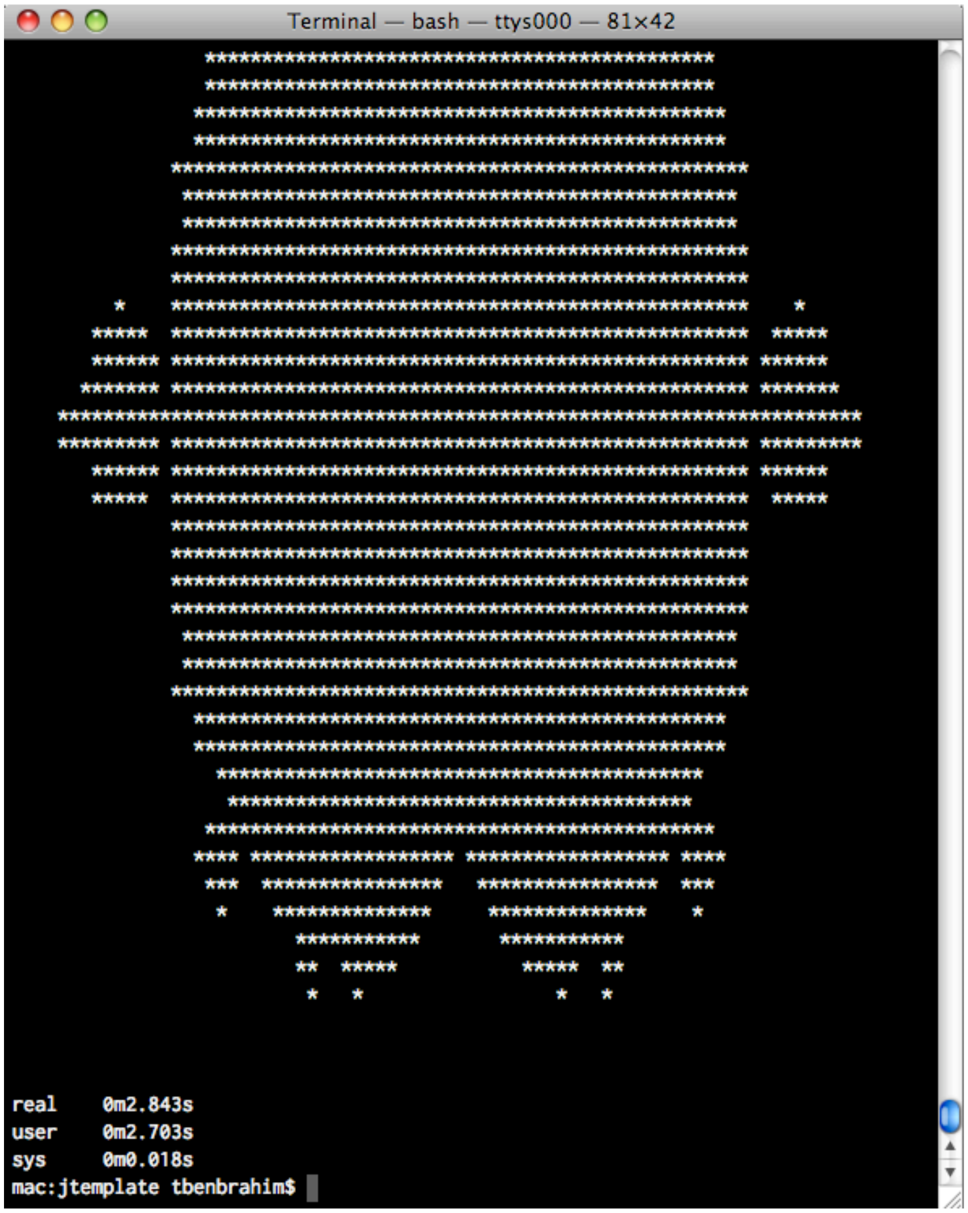


Per-file coverage

coverage	file
88%	build/analysis.ml
100%	build/ast.ml
96%	build/ast_info.ml
89%	build/environment.ml
79%	build/expression.ml
87%	build/filename_util.ml
88%	build/interpreter.ml
65%	build/jtemplate.ml
69%	build/lexer.ml
81%	build/lexer.mll
83%	build/library.ml
90%	build/library_builtin.ml
77%	build/library_io.ml
100%	build/library_string.ml
91%	build/parser.ml
28%	build/parser.mly
84%	build/parser_util.ml
65%	build/runtimeError.ml
86%	*total*

1.4.2 Performance Testing

While performance was secondary to correctness, it nonetheless became an important concern in the second phase of development. A set of three benchmarks was used to gauge performance. The Fibonacci benchmark involved calculating the 32nd Fibonacci number using a the naive recursive algorithm. The second involved calculating the Greatest Common Denominator of two large numbers ten thousand times in a for loop. The last benchmark used a Mandelbrot set generator ported from code at http://www.timestretch.com/FractalBenchmark.html. While these three benchmarks are far from being representative of all programs that can be run with Jtemplate, they nonetheless provided a good starting point for optimization in the limited time available, and allowed variable retrieval and storage, function invocation, integer and floating point arithmetics and iterative loops to be extensively optimized.



```
                 *********************************************
                 *********************************************
                ***********************************************
                ***********************************************
              ***************************************************
               *************************************************
               *************************************************
              ***************************************************
              ***************************************************
         *    ***************************************************    *
       *****  ***************************************************  *****
       ****** *************************************************** ******
      ******* *************************************************** *******
    ***********************************************************************
    ********* *************************************************** *********
       ****** *************************************************** ******
       *****  ***************************************************  *****
              ***************************************************
              ***************************************************
              ***************************************************
              ***************************************************
               *************************************************
               *************************************************
              ***************************************************
                ***********************************************
                ***********************************************
                  *******************************************
                   *****************************************
                 *********************************************
                **** ****************** ****************** ****
                 ***  ****************   ****************  ***
                  *    **************     **************    *
                         ***********       ***********
                         **  *****           *****  **
                          *   *                 *   *



real	0m2.843s
user	0m2.703s
sys	0m0.018s
mac:jtemplate tbenbrahim$
```

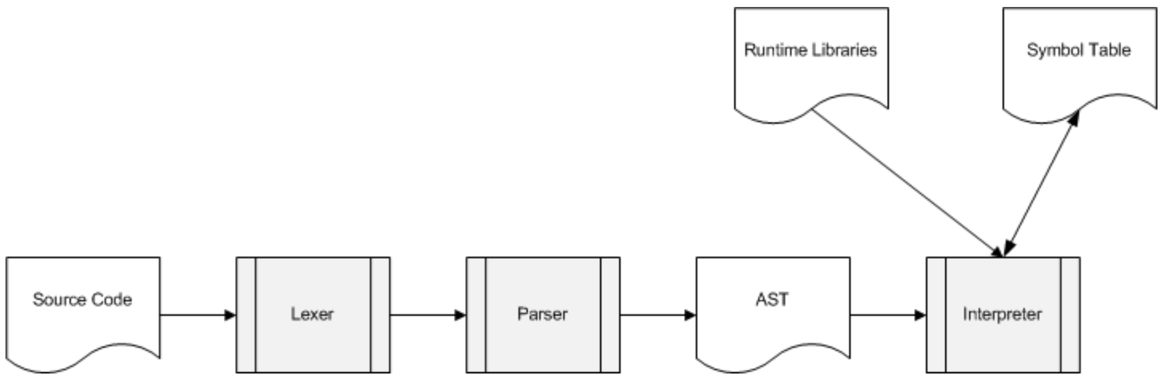
The Mandelbrot benchmark

Initially, Jtemplate performance was benchmarked against PHP 5.2.8, and later, as PHP performance was surpassed, against Ruby 1.8.6, Python 2.5.1 and the Rhino Mozilla JavaScript engine. All benchmark source code is available in Appendix D and results are discussed in detail in the subsequent sections

2 Architectural Guide

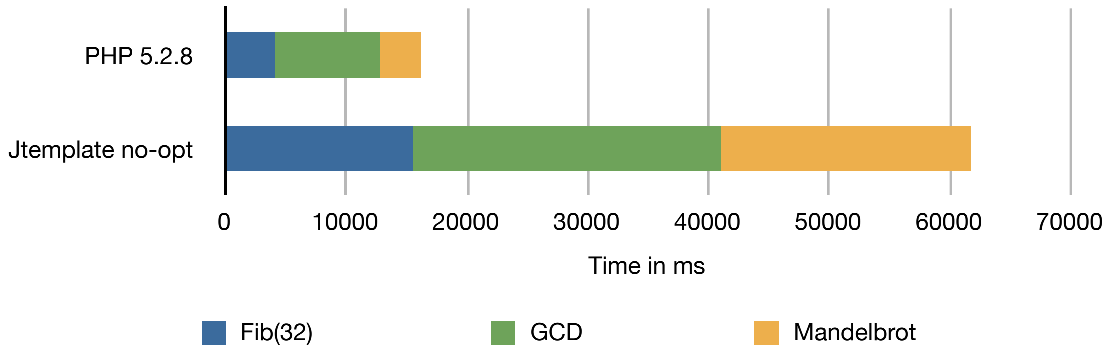
2.1 A false start

The initial design of the interpreter was rather simple, directly interpreting the abstract syntax tree generated by the parser without any pre-processing. All errors were detected at run time, and the symbol table was dynamically built as the program executed.



Block diagram of the initial implementation

This simple design had many flaws, chief among them the dismal performance experienced while benchmarking.



Implementation	Fib(32)	GCD	Mandelbrot	Total
PHP 5.2.8	4100	8700	3328	16128
Jtemplate no-opt	15500	25500	20750	61750

Performance of the initial implementation compared to PHP 5.2.8

Additionally, an oversight in the symbol table implementation caused all function calls made from within a function to be executed with normative rather than applicative order, due to the lack of dedicated storage for local variables.

To achieve better performance, the design was refactored to allow for correct and more efficient storage. In doing so, a runtime abstract syntax tree had to be generated, allowing for further optimizations to be made on the runtime AST before execution.

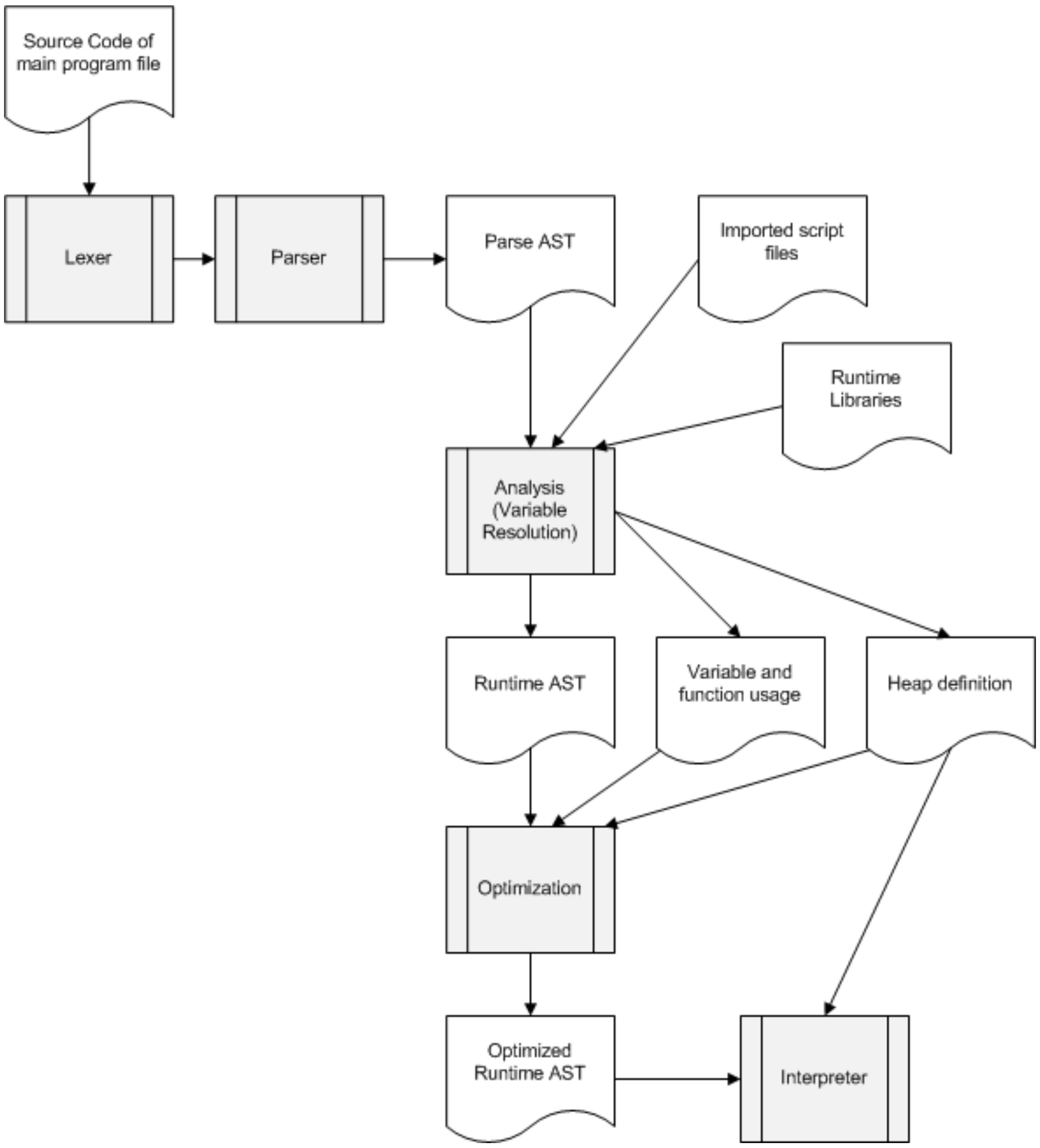
2.2 An improved design

The goals for the improved design were as follows:

- Resolve all variables before execution and allocate storage for globals before execution of a Jtemplate program.
- Speed up function invocation by predetermining the size of the stack frame needed by each procedure, and resolving every function parameter and local variable to an absolute position on the function stack frame.
- When errors such as undefined variable errors are found, continue processing and report all errors at the end of program analysis, rather than stopping processing on the first error encountered.
- Warn the developer before program execution of any unused variable, function argument and function definition, as these often indicate a potential logic mistake.
- Optimize the runtime AST. The initial plan was to remove assignments to variables that were never modified, and replace references to such variables with their constant value, and to inline some functions based on some yet undefined criteria.
- Experiment with as many other optimizations as time allowed and keep the ones that proved beneficial, as measured by running a set of benchmarks.

In achieving these objectives, a balance had to be maintained between over analyzing the abstract syntax tree thus delaying the actual program execution, and performing optimizations that would most likely benefit long running programs.

The following sections explore each step of the optimized runtime AST generation, that in the aggregate resulted in an almost nine fold increase in performance, surpassing PHP and Ruby in the performance of the selected benchmarks.



Block diagram of the final implementation

2.3 Storage allocation

The abstract syntax tree generated by the parsing phase uses names to identify variables. Since there is a need to differentiate between identically named but distinct variables in global and local scopes, each variable is assigned a unique id when it is first declared in a scope. Global variables are also assigned a slot in a global heap, an array used to hold the values of all global variables. Before analysis begins, the runtime libraries are processed and library objects are assigned a unique id and a slot in the global heap.

Example:

```
let a="test";
{
	let a="test 2";
	for (var a=1;a<10;a++)
		println(a);
}
println(a);
```

In the above program, the variable is declared in three different scopes, in the global scope at line 1, in a nested scope at line 3 and in a for loop at line 4. Each variable gets a unique id and storage allocation in the global heap:

19	20	21	22	23
println	readln	a (line 1)	a (line 3)	a (line 4)

and the AST gets rewritten to this equivalent form:

```
let Global(21,21)="test";
{
	let Global(22,22)="test 2";
	for (var Global(23,23)=1; Global(23,23) <10; Global(23,23) ++)
		Global(19,19)(Global(23,23));
}
Global(19,19)(Global(21,21));
```

The situation for local variables is bit more complex, since we cannot preallocate storage. A stack is needed, especially when calling functions recursively. The strategy used was to define a stack frame layout for each function definition, and assign each local variable a location on the stack frame. The layout of a stack frame for a function with *n* arguments is as follows:

- slot 0 is reserved for the `this` object pointer
- slots 1 to *n* are used for the function arguments. If the last argument is a vararg, it will be stored at position *n* as an array of values.
- slots n+1 to the end of the stack frame are used for any local variables declared within the body of the function definition.

For example, the following code fragment:

```
let add=function(x,y){
	let sum=x+y;
	return sum;
};
```

yields the following stack frame definition:

0	1	2	3
this	x	y	sum

Since Jtemplate supports nested functions, and allows references from a nested function to its parents' variables, we must also account for the stack frame depth. Consider the following code fragment:

```
let addTwo=function(x,y){
      let addOne=function(x){
            return x+y;
      };
      return addOne(x);
};
```

When the addOne function is invoked, it must have access to both its stack frame and to its parent function's stack frame, since the variable y referenced in the addOne function.

0	1	2	3	
this	x	y	addOne	addTwo stack frame (depth 0)
this	x			addOne stack frame (depth 1)

For local variables, the stack frame depth is stored in addition to the unique id and position within the stack frame. The AST generated for the code fragment above is equivalent to:

```
let Global(21,21)=function(Local(24,0,1),Local(25,0,2)){
      let Local(26,0,3)=function(Local(28,1,1){
            return Local(28,1,1)+Local(25,0,2);
      };
      return Local(26,0,3)(Local(24,0,1);
};
```

While recording the stack depth is useful for differentiating variables in different scope, this is not sufficient at run time, since there can be no guarantee that the parent stack frame will exist at run time. Consider the following example

```
let add=function(x){
      return function(y){
            return x+y;
      };
};

println(add(1)(2));
```

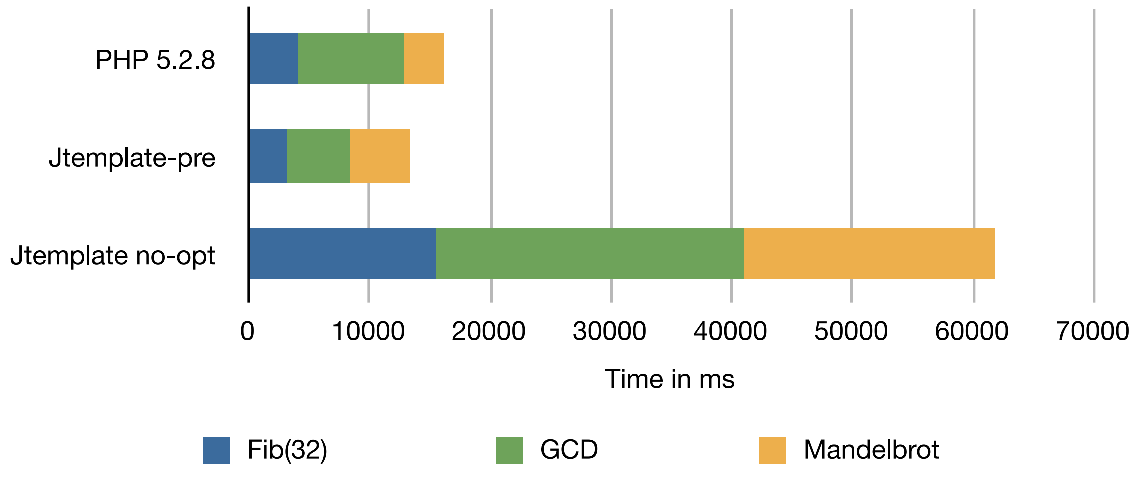
In this example, `add` is a function that returns a function. When the returned function is invoked with the value 2, the first stack frame, containing the value of x, no longer exists. In order to handle this situation, it is necessary to implement closures over a function's free variables.

When a function definition is analyzed, a distinction is made between bound variables (variables of the same stack frame depth as the function's stack frame) and free variables (variables declared in a parent stack frame). All free variables locations are recorded in the map associated with the function value. At run time, when the function is declared, the values of all free variables are looked up and bound to the closure variables. Every time the function is invoked, the values that were bound when the function was declared are used instead of the reference to stack frame location. For example, the AST for the example given above is equivalent to:

```
let Global(21,21)=function(Local(23,0,1)){
        return function(Local(25,1,1){
                return Local(23,0,1)+Local(25,1,1);
        };
};

Global(19,19)(Global(21,21)(1)(2));
```

The variable highlighted in red is a free variable. When the function is analyzed, the variable location is added to a map of closure variables to values. When the script is interpreted and the function is declared, the location and value of this variable replaces the previously added location in the closure variables map. Whenever the function is invoked, any reference to a local variable is looked up first in the closure variables map, and only if it does not exist is it looked up in the function's stack frame.

Replacing variable names with absolute locations, avoiding possibly recursive variable lookup in a symbol table map, had a very significant effect on performance, as can be seen from the following benchmark results:



Implementation	Fib(32)	GCD	Mandelbrot	Total
PHP 5.2.8	4100	8700	3328	16128
Jtemplate-pre	3200	5160	4936	13296
Jtemplate no-opt	15500	25500	20750	61750

2.4 Template handling

During the analysis phase, the template definition and template instruction AST generated by the parsing phase are turned in a single function call per template. First, thorough checking is performed to ensure that any nesting defined in the template specification is valid. A function is then generated for the entire template, followed by nested functions for each label in the template. For example, the following template specification and instructions:

```
template htmlTable{
             #<table>
             #<tr>
header       #<th>columnLabel</th>
             #</tr>
row          #<tr>
cell         #<td>cellData</td>
row          #</tr>
             #</table>
}

instructions for htmlTable(dataMap){
      header foreach(label in dataMap.labels): columnLabel=label;
      row foreach(dataArray in dataMap.data): ;
      cell foreach(element in dataArray): cellData=element;
}
```

result in the following code being generated and added to the runtime AST:

```
let htmlTable= function(dataMap){
      let result='';
      result+='<table>';
      result+='<tr>';
      foreach(label in dataMap.labels)
            result+=header(label);
      let header=function(label){
            let result='';
            result+='<th>columnLabel</th>'.mreplace(['columnLabel'],[label]);
            return result;
      };
      result+='</tr>';
      foreach(dataArray in dataMap.data)
            result+=row(dataArray);
      let row=function(dataArray){
            let result='';
            result+='<tr>';
            foreach(element in dataArray){
                  result+=cell(element);
                  let cell=function(element){
                        let result='';
                        result+='<td>cellData</td>'.mreplace(
                          ['cellData'],[element]);
                        return result;
                  };
```

```
				}
				result+='</tr>';
				return result;
		};
		result+='</table>';
		return result;
}
```

2.5 Variable Usage Analysis

During the first pass of the analysis phase, the usage of every variable is tracked to determine if a variable is read after being declared or written after being declared and initialized. If a variable is neither read nor written after being declared, it is reported to the user as an unused variable before the program is executed:

For example, the following program:

```
let foo=function(x,y,z){
		return x+y;
};
let a=1;
```

generates the following warnings:

```
WARNING: At line 1 in m.jtp: Variable foo is never read.
WARNING: At line 4 in m.jtp: Variable a is never read.
WARNING: At line 1 in m.jtp: Variable z is never read.
```

Variables that are read but never written, with the exception of function definitions, are replaced with their values in the runtime AST. For example, in the following program, the variable `side` is never written after being declared:

```
let side=4;

let square=function(x){
		return x*x;
};

let printArea=function(x){
		println(square(x));
};

printArea(side);
```

In the second pass of the analysis phase, the assignment to the variable side is eliminated, and every reference to the variable side is replaced with its value, yielding to an AST equivalent to:

```
let square=function(x){
		return x*x;
};
```

```
let printArea=function(x){
	println(square(x));
};

printArea(4);
```

2.6 Constant folding

In the first and second pass of analysis, any binary or comparison operation on two constant values is replaced by its evaluated value. In the following code fragment:

```
let a=4;
let b=a*a;
```

the variable a is replaced in the second line by its constant value as described in section 2.5, then constant folding is performed on the multiplication of two constant values to yield:

```
let b=16;
```

2.7 Function inlining

In the first pass of analysis, every function is examined to determine if it qualifies for inlining using these conservative rules:

- The function body must be one line, consisting of either a return statement or an expression statement, usually a function call.
- The arguments must not contain a vararg.
- The function must not contain closure variables.

If a function can be inlined, the expression contained within its body is recorded and used in the second pass to replace any function invocation with the expression, performing a one to one mapping between the function's arguments and the local variables of the original expression.

In the following example, both the square and printArea function can be inlined:

```
let side=4;

let square=function(x){
	return x*x;
};

let printArea=function(x){
	println(square(x));
};

printArea(side);
```

Dealing with the `square` function first, the `printArea` function is converted to:

```
let printArea=function(x){
	println(x*x);
};
```

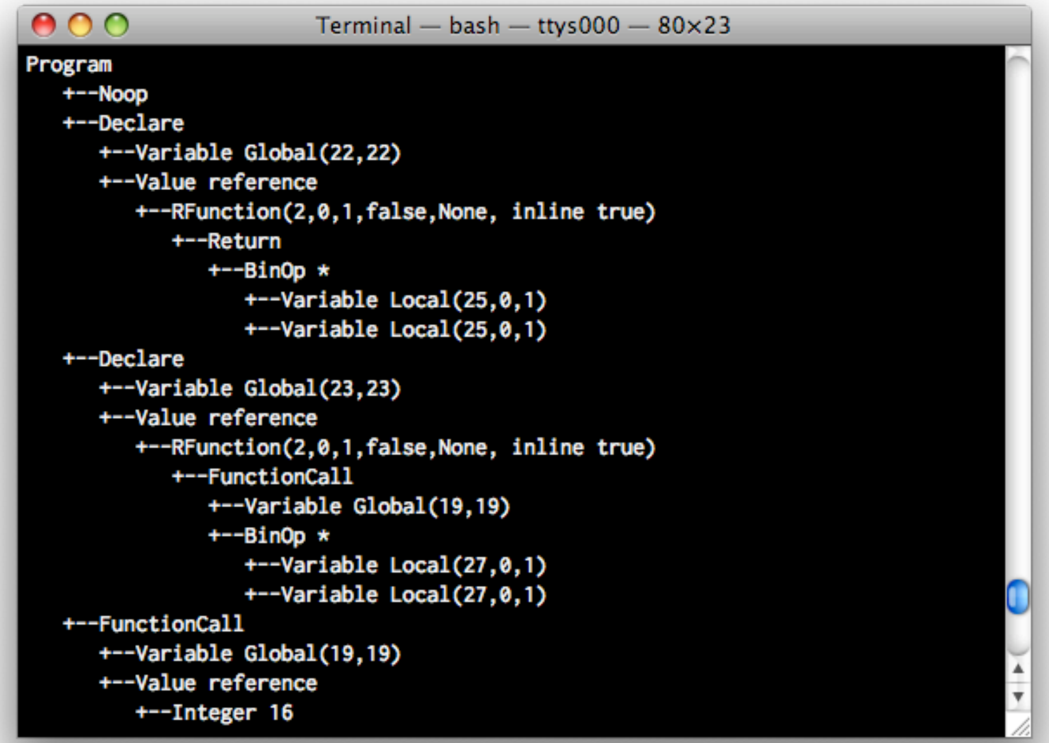
The `printArea` function is then inlined, so the call to `printArea` is converted to:

```
println(side*side);
```

After constant value substitution and constant folding, the program above simply becomes:

```
println(16);
```

The following image shows the generated AST for the program given above:



```
Program
   +--Noop
   +--Declare
      +--Variable Global(22,22)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--Return
               +--BinOp *
                  +--Variable Local(25,0,1)
                  +--Variable Local(25,0,1)
   +--Declare
      +--Variable Global(23,23)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--FunctionCall
               +--Variable Global(19,19)
               +--BinOp *
                  +--Variable Local(27,0,1)
                  +--Variable Local(27,0,1)
   +--FunctionCall
      +--Variable Global(19,19)
      +--Value reference
         +--Integer 16
```

Function inlining also had a beneficial effect on the performance of recursive functions, although care had to be taken to only inline those once, rather than let inlining proceed recursively until the program ran out of resources. Consider the following recursive function to calculate the greatest common denominator of two numbers:

```
let gcd=function(x,y){
	return x<y ? gcd(x,y-x) : x>y ? gcd(x-y,x) : x;
};
```

After one application of inlining, the AST is equivalent to:

```
let gcd=function(x,y){
	return x<y ? (x<y-x ? gcd(x,(y-x)-x) : x>y-x ? gcd(x-(y-x),x) : x) :
			x>y ? (x-y<y ? gcd(x-y,y-(x-y)) : x-y>y ? gcd((x-y)-y,
			x-y) : x-y) : x;
};
```

This form cuts the number of recursive calls in half, greatly improving performance.

2.8 Partial function application

The partial function application feature of Jtemplate is syntactic sugar for a function returning a function. For example, given the function

```
let add=function(x,y){
	return x+y;
};
```

a new function `incr` can be defined as

```
let incr=add(1,@y);
```

where @y indicates an unbound variable y. In the runtime AST, this expression is converted to the function

```
let incr=function(y){
	return add(1,y);
};
```

In the case where the bound argument is a variable, as in the statements:

```
let x=1;
let incr=add(x,@y);
```

a function closure will only be created if the bound argument is a local variable. If the bound variable is a global variable, incr will evaluate the sum of whatever x happens to be at the time and the argument y. This is different from:

```
let x=1;
let incr=function(x){
	return function(y){
		return add(x,y);
	}
}(x);
```

in which the bound variable is coerced into a local, always causing a function closure, a behavior which in retrospect may have been more desirable.

2.9 Optimizing for loops for the common case

Many for loops are counting loops. Rather than interpreting the loop condition and the post loop expression, a small performance gain can be achieved by evaluating those expressions natively. To qualify for optimization, a for loop must be of the form

```
for (n=value;n compop value; n = n op incr | n++)
```

where n is any variable `compop` is any comparison operator, `op` is any arithmetic operator and `incr` is an integer value. When such loops are found, the standard loop statement is replaced with an optimized loop statement where the comparison and incrementing are performed in native code, and the result of the increment is assigned to the loop variables.

2.10 Library importing

The first step of the analysis step is to import the various runtime libraries functions and objects into the analysis environment, so that calls to library functions can be resolved. Library functions and objects are added to the global heap, just as global variable declared in a program would be. At the end of the analysis phase, prior to execution the library functions are transferred from the analysis heap into the runtime heap.

As of this writing, every program has the following twenty objects defined in its global heap:

Index	Name	Description
0	`args`	an array that holds any script command line arguments
1	`String`	a map containing functions to work with strings
2	`Integer`	a map containing functions to work with integers
3	`Float`	a map containing functions to work with floats
4	`Boolean`	a map containing functions to work with booleans
5	`Function`	a map containing functions to work with functions
6	`Void`	a map containing functions to work with void values
7	`NaN`	a map containing functions to work with NaN values
8	`Array`	a map containing functions to work with arrays
9	`Map`	a map containing functions to work with maps
10	`void`	the void value
11	`Date`	a map containing functions to work with dates
12	`Debug`	a map containing debugging functions
13	`System`	a map containing system functions
14	`exit`	the exit function, causing a program to terminate
15	`typeof`	the typeof function, returning the type of a variable

Index	Name	Description
16	`File`	a map containing functions to work with files
17	`Directory`	a map containing functions to work with directories
18	`print`	the print function
19	`println`	the println function
20	`readln`	the readln function

As can be seen from the table above, most library functions are exposed through maps corresponding to a namespace, rather than as a top level functions, but in either case, the manner in which library functions are defined is similar.

A library function is defined by:

- an array of names, either a single name, such as `[“println”]`, or the namespace qualified name, such as `[“String”,”prototype”,”length”]`. If a single name is given, the function is added to the global heap. If a namespace is given, the function is added to the corresponding map, which is later added to the heap.
- the number of arguments.
- whether the function has a variable number of arguments.
- an Ocaml function that takes a runtime environment and returns nothing. If a value must be returned, this is accomplished through the control flow exception mechanism, as if the value had been returned from a native Jtemplate function.

Library functions are organized into libraries (a list of library functions), each of which is imported into the analysis environment.

2.11 Jtemplate imports

Jtemplate supports importing external script files into the main script and other imported script files. The import statements are processed during the analysis phase, by parsing an imported file and adding its declarations into the AST at the point where the import occurred, and by recursively processing any import statements found in the imported file. Dependency loops are avoided by only processing a given import file once.

2.12 Program Execution

Profiling revealed that two areas needed special attention to improve performance.

The first concern was the overhead created by stack trace generation during recursive calls. To eliminate the overhead, stack traces were limited to the first entry into a recursive function.

The second concern dealt with the slowness with which floating point arithmetic expressions were evaluated. The original Language Reference Manual stated tin section 2.4.2 that:

Jtemplate differs from IEEE-754 in its treatment of NaN, infinity and –inifinity, which are all converted to the non float value NaN when they occur.

This turned out to be a bad idea, as implementing this required three comparisons after every floating point arithmetic expression, nearly doubling the execution time. Instead, float constants were defined (`Float.infinity`, `Float.negativeInfinity` and `Float.NaN`) so that the user could compare the result to one of these values, and the three comparisons were eliminated.

3 Performance

Performance was gauged using three benchmarks:

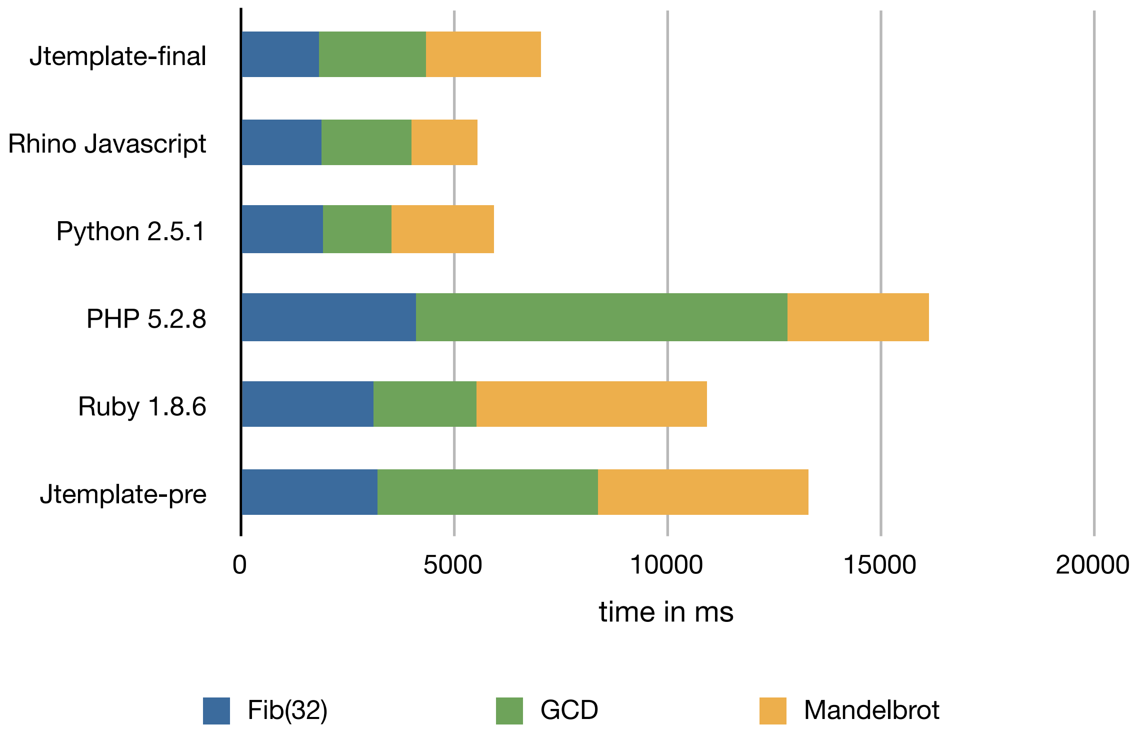
- The Fibonacci benchmark involved calculating the 32nd Fibonacci number using a the naive recursive algorithm.
- The GCD benchmark involved calculating the Greatest Common Denominator of two large numbers ten thousand times in a for loop.
- The Mandelbrot set generator benchmark was a late addition used to provide a more realistic mix of instructions.

While in no way representative of the full range of programs to be run on the Jtemplate interpreter, these benchmarks nonetheless focused the development effort on useful optimizations.

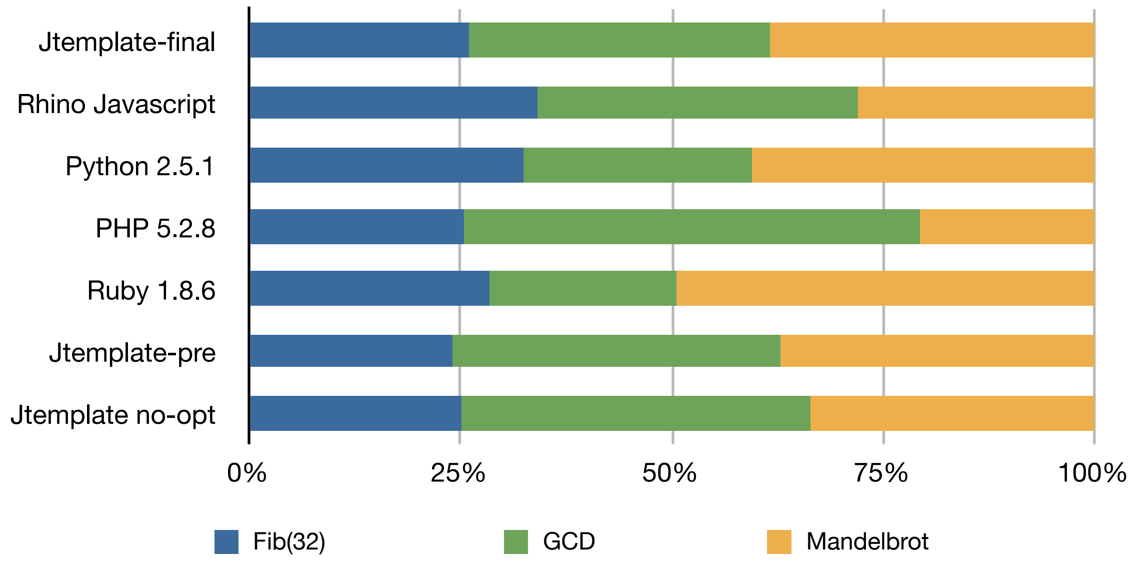
Final benchmark results are shown below. Three versions of Jtemplate are shown:

- Jtemplate no-opt: the initial version of Jtemplate with no runtime AST.
- Jtemplate-pre: the initial version of Jtemplate with only the more efficient storage mechanism implemented.
- Jtemplate final: the final version of Jtemplate.

Implementation	Fib(32)	GCD	Mandelbrot	Total
Jtemplate-final	1830	2500	2700	7030
Rhino Javascript	1890	2100	1550	5540
Python 2.5.1	1925	1600	2400	5925
PHP 5.2.8	4100	8700	3328	16128
Ruby 1.8.6	3100	2420	5391	10911
Jtemplate-pre	3200	5160	4936	13296
Jtemplate no-opt	15500	25500	20750	61750



A 1.9 fold increase in performance was noted in the overall benchmarks between Jtemplate-pre and Jtemplate final, an advantage that carried over to the more general Mandelbrot benchmark, with a 1.8 fold increase. It is also interesting to note than in going from the original interpreter's 61.8 seconds to the final interpreter's 7 seconds, the relative time spent in each benchmark stayed surprisingly consistent.



Both the GCD and Fibonacci benchmarks were designed to have roughly 7 million function calls and the same number of arithmetic expression evaluations and comparison operations. The GCD benchmark does however have to iterate 10,000 times in a for loop, and may have to perform one more comparison, while the Fibonacci benchmark is purely recursive. It would be interesting to explore why both Ruby and Python performed so much better in relative and absolute terms against the Javascript and Jtemplate implementation of GCD, and relative to their implementation of the Fibonacci benchmark.

4 Lessons Learned

4.1 Define “done”

Any number of intermediate versions of the interpreter could have been considered done. However, this project suffered from scope creep as the optimizations to be performed on the AST were not clearly defined before development started. This resulted in delaying the start of the manual until the last possible minute, because I was busy adding another neat feature to the interpreter.

Clearly, I was more interested in semantic analysis and experimenting with dynamic language optimizations than I was in the stated purpose of the language,that is to experiment with an alternate method of working with text templates. This is reflected by the fact that the functionality did not appear in the code until revision 124 (of 153), one week before the completion of the project, and that the samples for templating functionality are on the whole fairly weak.

Nevertheless, the Jtemplate interpreter is more than done for my purposes (the midterm version would have been adequate), and I plan to test out a model driven code generation application in the language fairly soon, as soon as I write a model parser so that models do not have to be defined in the Jtemplate language.

4.2 Technical Lessons Learned

Since this is my first interpreter, the opportunity for mistakes and lessons learned was abundant. As development progressed, mistakes were uncovered that due to time constraints were not remedied.

One such mistake is believing that by requiring all variables to be declared and initialized before use, types could be inferred. While this is true to some extent, the way in which the storage allocation is currently implemented does not allow for this. Currently, if two variables with the same name have the same scope, they are allocated to the same storage location even if they have different types. A more correct implementation would have been to allocate a new storage location for each declaration found, or even to use SSA (which I was not familiar with until late in the process).

Even with this correction, maps are an integral part of the language and the current specification of the language makes it impossible to infer the type of a map member. Since the member expression can be anything as long as it evaluates to a string or can be cast to a string (anything can be cast to a string in Jtemplate), it is impossible to reliably determine what map member is being assigned, For example, x.1, x[1], x[“1”], x[“a”.length()] and x[a-b] where are a-b=1 at runtime all refer to the same member “1” of map x. A more robust implementation would have defined two types of map, a structured map where members are declared using the dot notation, and a dynamic map where keys are accessed using brackets. It would have then been possible to implement type checking against structured maps. Additionally, the runtime library and other code could have required that functions be defined in structured maps to obtain the “dynamic” dispatch functionality, opening up the possibility of constructing a virtual table for every map at analysis time, or at the very least memoizing the correct function dispatch at run time, avoiding the current state of checking for the existence of up to four functions when invoking a map member function.

Even with those changes, functions present a challenge of their own. For example, `function(x,y){return x+y;};` defines a function that takes any two types and returns a string, an integer or a floating point number. It is not clear how to avoid this without explicitly declaring the argument and return types, or resorting to Ocmal’s type specific operators and conversion functions, something not very palatable to users of most mainstream languages.

Another mistake was letting function closures slip into the language, despite my initial intent to avoid them. This was caused by failing to realize that allowing nested functions to refer to outer variables may require closures, especially when those values are returned. A more reasonable model would have been to allow function invocations from an outer level to an inner level, but not the reverse. In addition, the implementation of partial function application, meant explicitly to avoid function closures, is such that it requires closures. This is because partially applied are processed in the AST during the analysis phase, when not all values are known and some of those values may come local variables in a parent stack frame, rather than at run time, when all values are known. This is how partial function application was originally implemented in the first version of the Jtemplate interpreter, and function closures were not needed.

5 Suggestions for Future Project Teams

5.1 Testing

As a professional Java developer, I am not a big fan of unit testing and test driven development. In the enterprise development that comprises the majority of my work, the time spent writing unit tests is a drain on productivity rather than a productivity enhancer. In the last year, I have written a total of maybe four unit tests, a rather sad sign that my development work is too mundane to merit unit testing.

However, writing an interpreter is not like writing yet another database driven web site. There are complex interactions between the components where one seemingly trivial change can break a lot of previously developed features. For this project, I initially wrote over 120 tests in Ocaml, before the interpreter even ran, and almost 150 unit tests in the Jtemplate language once the compiler was running.

You should want to write unit tests because they will save you time in your development, not as an afterthought because tests are required for the project. It is very convenient in all stages of development to be able to add a feature to the code and to be able to run the tests to ensure nothing was broken. Even more importantly, the best time to find out that a "minor tweak" to the code broke 76 of 148 tests is right after the code was changed, not the next day or even the next hour when you get around to testing. While developing, I kept a terminal window open so that I could regularly run the unit tests after every change. In a typical hour of development, the tests were run five or six times, which was not a burden since all the tests completed in well under a second. For this to be successful, it is important that you can launch all tests with one command, and that you can see the aggregate result at the end without having to scroll through pages of test results.

A mistake made is often a mistake repeated. It is not uncommon, especially on projects that span several months, to fix a bug only to have it reappear some time later. While this is unavoidable, it can at least be detected very early if a test is written for every bug that is found. Every time I found a bug, usually by running sample programs, I wrote a unit test for the condition before fixing the bug. I knew I had fixed the bug when all the tests passed.

Finally, be thorough in the tests you write. For example, writing a test for normal array access is not sufficient. A test is also needed to test array bounds error, both under and over, and invalid indices, such as indexing an array with a string.

5.2 Ocaml

Ocaml may seem like a drag to your productivity. After all you know Java, and things would be so much faster with Java. As someone who writes tens of thousands line of Java every year, I found this to be totally untrue, for two reasons. First, you would have to revert to C to gain the performance of Ocaml. Java, while reasonably fast, cannot keep up with Ocaml in term of performance. It is easy to convince yourself of that by benchmarking a Fibonacci function in both languages. Compile the Ocaml code to native code with -ccopt -O3. It wont be close.

Second, there is a lot of "ceremony" in Java that requires one to type seemingly unnecessary code. For example, why do you have to type `ArrayList<String> myList=new ArrayList<String>()`, when a simple `let myList=new ArrayList<String>()` would suffice. Second there are constructs in Ocaml that have no equivalent in Java. Take the following Ocaml statement, extracted from the function inlining code in Jtemplate:

```
let inline_expr = match (stmt_list, closure_vars) with
  | (RReturn(RValue(RFunctionValue(_,_,_,_,_,Some h,_)),_)::[],_) -> None
  | (RReturn(expr, _)::[], None)
  | (RExpressionStatement(expr, _)::[], None) -> Some expr
  | ([], None) -> Some (RValue(RVoid))
  | _ -> None
```

Here is some equivalent Java code for just the <u>first</u> match, one line of code in Ocaml:

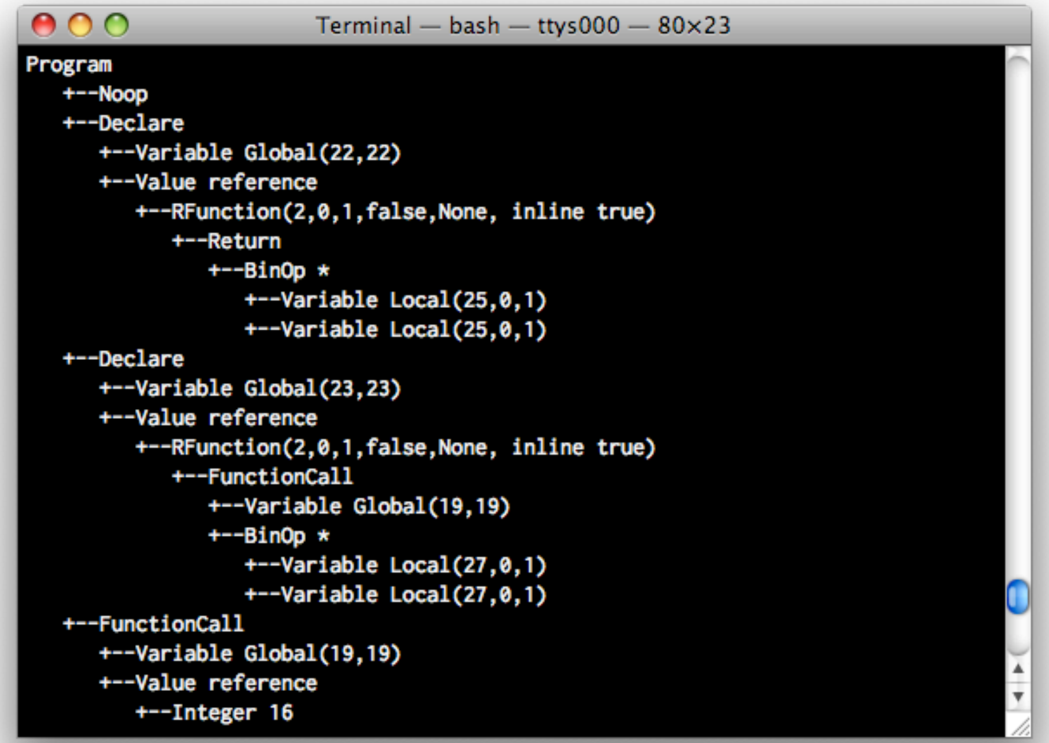
```
if (stmtList.size()==1){
  Statement first=stmtList.get(0);
  if (first instanceof ReturnStatement){
    Expression expr=((ReturnStatement)first).getReturnExpression();
      if (expr instanceof ValueExpression){
        ValueExpression expr=(ValueExpression)expr;
        if (expr.getValue() instanceof FunctionValue){
          FunctionValue func=(FunctionValue)(expr.getValue());
          if (func.getClosureVars()!=null)
            return null;
        }
      }
    }
}
```

I know I would rather type `(RReturn(RValue (RFunctionValue(_,_,_,_,_,Some h,_)),_)::[],_) -> None` rather than the code above and there are still three more cases and default case to write! Not to mention that Statement, ReturnStatement, Expression, ValueExpression and FunctionValue in the Java code above are classes that have to be written, rather the one line of OCaml code required to declare each construct. More importantly, I can still clearly tell that I am matching a statement list with a single statement consisting of a return of a function that contains some closure variables, ten days after the code was written. I am not sure what I am looking at in the Java code above five minutes after I have written it. This is not to say that Java is not an appropriate language, I use it every day, but that Ocaml is so much better suited to the task of interpreter development.

In fact, I am quite confident that using Java, I would never have included all the features in the Jtemplate interpreter, as I would have spent most of time typing and fixing unmaintainable Java code.

5.3 Tooling

In developing the Jtemplate interpreter, the most valuable tool besides unit tests was the ability to pretty print the AST and dump the symbol table.

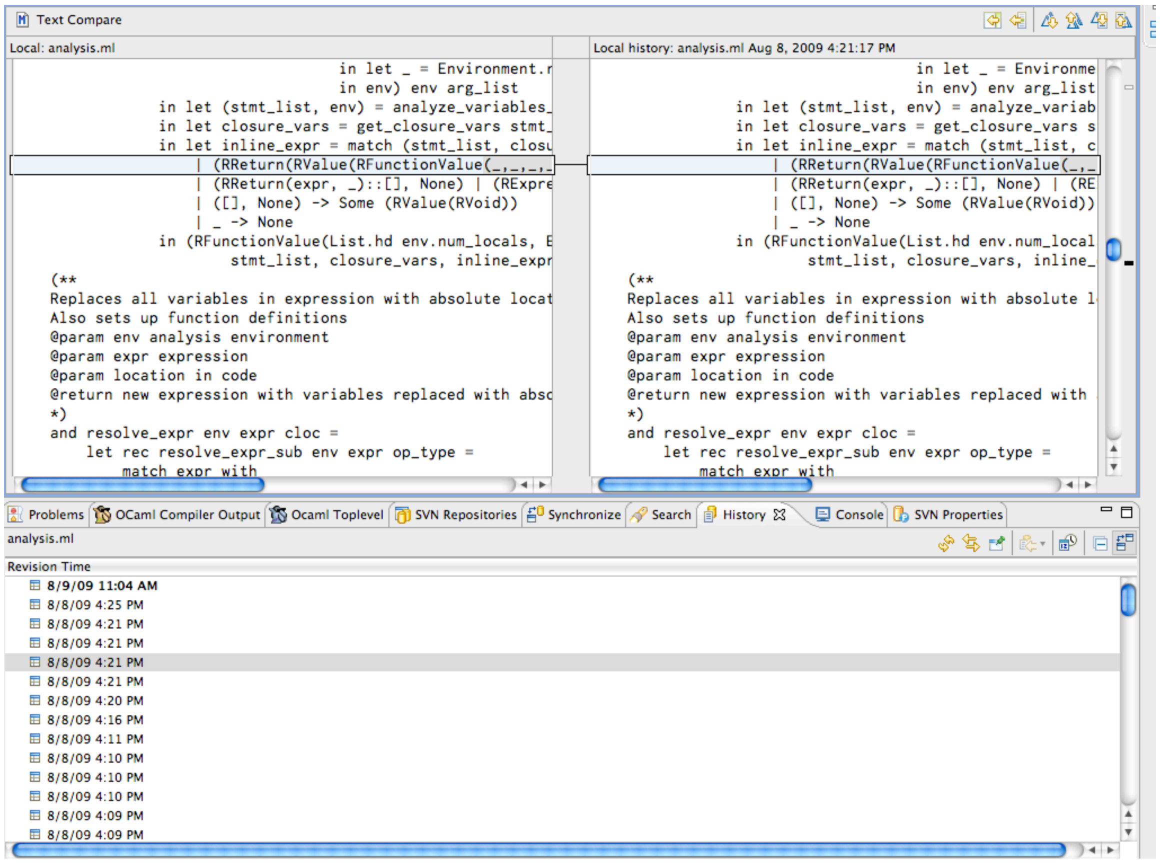


```
Program
   +--Noop
   +--Declare
      +--Variable Global(22,22)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--Return
               +--BinOp *
                  +--Variable Local(25,0,1)
                  +--Variable Local(25,0,1)
   +--Declare
      +--Variable Global(23,23)
      +--Value reference
         +--RFunction(2,0,1,false,None, inline true)
            +--FunctionCall
               +--Variable Global(19,19)
               +--BinOp *
                  +--Variable Local(27,0,1)
                  +--Variable Local(27,0,1)
   +--FunctionCall
      +--Variable Global(19,19)
      +--Value reference
         +--Integer 16
```

This code was developed right after completion of the parser, to ensure that the generated AST was equivalent to the input code. While developing the interpreter, it served as a guide to the structure of the AST, and while transforming the parse AST into a runtime AST and optimizing the runtime AST, to ensure that the transformations that were applied are correct.

Consider using ocamlbuild instead of a makefile, especially early when the dependencies between modules are changing frequently.

Commit often. It gives you a point to return to in case things break beyond all hope of repair. Eclipse has a nice feature of an internal CVS repository that commits your code every time you save. This gives you the ability to compare against code as it existed minutes ago and even revert to any previously saved version.



Local: analysis.ml

```
                              in let _ = Environment.r
                              in env) env arg_list
                in let (stmt_list, env) = analyze_variables_
                in let closure_vars = get_closure_vars stmt_
                in let inline_expr = match (stmt_list, closu
                    | (RReturn(RValue(RFunctionValue(_,_,_,_
                    | (RReturn(expr, _)::[], None) | (RExpre
                    | ([], None) -> Some (RValue(RVoid))
                    | _ -> None
                in (RFunctionValue(List.hd env.num_locals, E
                         stmt_list, closure_vars, inline_expr
    (**
    Replaces all variables in expression with absolute locat
    Also sets up function definitions
    @param env analysis environment
    @param expr expression
    @param location in code
    @return new expression with variables replaced with abso
    *)
    and resolve_expr env expr cloc =
        let rec resolve_expr_sub env expr op_type =
            match expr with
```

Local history: analysis.ml Aug 8, 2009 4:21:17 PM

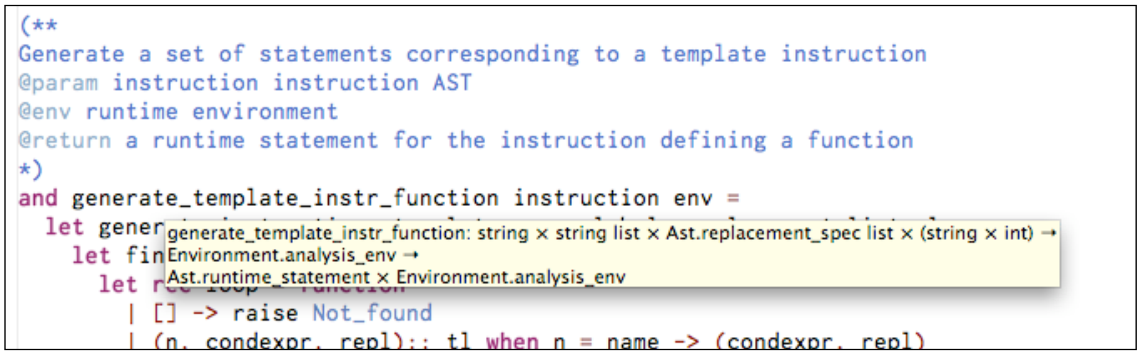
```
                              in let _ = Environme
                              in env) env arg_list
                in let (stmt_list, env) = analyze_variab
                in let closure_vars = get_closure_vars s
                in let inline_expr = match (stmt_list, c
                    | (RReturn(RValue(RFunctionValue(_,_
                    | (RReturn(expr, _)::[], None) | (RE
                    | ([], None) -> Some (RValue(RVoid))
                    | _ -> None
                in (RFunctionValue(List.hd env.num_local
                         stmt_list, closure_vars, inline_
    (**
    Replaces all variables in expression with absolute l
    Also sets up function definitions
    @param env analysis environment
    @param expr expression
    @param location in code
    @return new expression with variables replaced with
    *)
    and resolve_expr env expr cloc =
        let rec resolve_expr_sub env expr op_type =
            match expr with
```

analysis.ml

Revision Time
8/9/09 11:04 AM
8/8/09 4:25 PM
8/8/09 4:21 PM
8/8/09 4:21 PM
8/8/09 4:21 PM
8/8/09 4:21 PM
8/8/09 4:20 PM
8/8/09 4:16 PM
8/8/09 4:11 PM
8/8/09 4:10 PM
8/8/09 4:10 PM
8/8/09 4:10 PM
8/8/09 4:09 PM
8/8/09 4:09 PM

Eclipse's local history feature

Finally, I found the ocaide Eclipse plugin a real time saver, for its ability to highlight syntax errors and the ability to determine the type of any symbol by simply hovering above the symbol.



```
(**
Generate a set of statements corresponding to a template instruction
@param instruction instruction AST
@env runtime environment
@return a runtime statement for the instruction defining a function
*)
and generate_template_instr_function instruction env =
  let gener
    let fin
      let r
        | [] -> raise Not_found
        | (n, condexpr, repl):: tl when n = name -> (condexpr, repl)
```

```
generate_template_instr_function: string × string list × Ast.replacement_spec list × (string × int) →
Environment.analysis_env →
Ast.runtime_statement × Environment.analysis_env
```

5.4 Start early

There is never enough time. I found it very helpful to have a running interpreter by midterm, as it took some of the pressure off. Too much pressure off, as it turns out, as I find myself scrambling to complete the project report on the day before it is due.

I also had difficulty determining the end of the project, as I did not clearly define the objectives. A number of "done" versions were produced between July 26 and August 3rd, as there was always one more neat optimization to implement. In the end, I had to accept good enough as done, as I would have needed another three weekends of development to implement all the desired optimizations. This is especially unusual for me, as I am used to working in an agile Scrum environment, where “done” is defined before the first line of code is written. So make sure to define “done”, especially since the final product is on the critical path to finalizing the documentation and samples. This is especially true in a Summer semester, as the end of the term approaches very quickly.

Developing the project individually rather than in a team was a tremendous advantage as it eliminated the coordination overhead. But if you are working in a team, start especially early and plan that much more thoroughly.

Appendix A – Module Documentation

Module Analysis

```
module Analysis: sig .. end
```

Create an optimized AST from the parsing phase AST

Pass 1:

- resolve all variable references, including closure variables
- process imports and add declarations to AST
- builds runtime AST
- convert template definitions / instructions
- evaluate operations on constants and replace with value in AST
- determine if variables are initialized with a constant value (rather than an expression), if a variable is written after being declared and if it is ever read after being declared
- determine if a function is inlineable

Pass 2: The second pass replaces all non function variables whose value have not been modified with a constant value, and evaluates operations on constants , eliminates assignment statements on constant values when the variable is not reassigned and not written, inline functions

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val check_errors : Environment.analysis_env -> unit
```

Prints all errors in an analysis environment and raises FatalExit if there are errors

Raises `FatalExit` if there are errors in the environment

Returns unit

`env` : analysis environment

```
val check_warnings : Environment.analysis_env ->
Environment.analysis_env
```

Generates additional warnings about unused variables, then prints all warnings

Returns analysis environment with newly added warnings

`env` : analysis environment

```
val print_name_info : Environment.analysis_env -> unit
```

Prints information about names found during analysis

Returns unit

`env` : analysis environment

FIRST PASS

- resolve all variable references, including closure variables
- process imports and add declarations to AST
- builds runtime AST
- convert template definitions / instructions
- evaluate operations on constants and replace with value in AST
- determine if variables are initialized with a constant value (rather than an expression)
- determine if a variable is written after being declared and if it is ever read after being declared
- determine if a function is inlineable

`exception TemplateError of string`
internal exception to signal an error in template processing.

```
val check_template_nesting : Ast.template_spec list ->
       (string, Environment.label_pos) Hashtbl.t * (string * int)
list
```

Checks for invalid nesting in a template specification
Returns an list of tuples containing the label and line offset where conflicts where found *
`template_spec` : the template spec to check

```
val generate_template_instr_function : string * string list *
Ast.replacement_spec list * (string * int) ->
       Environment.analysis_env -> Ast.runtime_statement *
Environment.analysis_env
```

Generate a set of statements corresponding to a template instruction
Returns a runtime statement for the instruction defining a function
`instruction` : instruction AST
`env` : runtime environment

```
val filter_imported_ast : Ast.statement list -> Ast.statement
list
```

Filters an ast, returning only a list of declaration and import statement
Returns a statement list containing only declarations and imports
`stmts` : the statement list to process

```
val analyze_variables : Environment.analysis_env ->
       Ast.statement -> Ast.runtime_statement *
Environment.analysis_env
```

find declarations and resolve references to variables. Since declarations are visible in the entire scope in which they are defined, and not just after they are declared, a breadth first search is necessary before recursively processing children statements
Returns an ast where all variables have been resolved to an absolute location, either on a stack or in the global heap and an environment containing information about all variables
`env` : an analysis environment
`ast` : the intermediate ast

SECOND PASS

- replace all constant declarations with Noop
- replace all constant variables with their value
- replace all constant expressions with the computed value
- replace all calls to inlineable functions with an expression

```
val inline_expr_replace : int ->
       int ->
       Ast.runtime_expression list ->
       Ast.runtime_expression -> Ast.runtime_expression
```

replaces an expression from an inlined function with the corresponding values from a function call expression list
Returns the inline expression with the arguments replacing the former local args
`depth` : the stack depth, for sanity checking
`numargs` : the number of arguments
`expr` : the inline expression

```
val replace_constant : Environment.analysis_env ->
```

```
       int list -> Ast.runtime_expression ->
Ast.runtime_expression
```

Replace non modified variables with their declared value
Returns an expression with constant variables replaced by their value
`env` : analysis environment
`inline_uids` : list of inlined functions to avoid recursively inlining recursive inlinable functions

```
val pass2 : Environment.analysis_env -> Ast.runtime_statement ->
Ast.runtime_statement
```

Looks for expressions where constants can be substituted
`env` : analysis environment

```
val analyze : Ast.statement -> Ast.runtime_statement *
Environment.analysis_env
```

Analyzes an AST, generates a runtime AST
Returns a tuple of the runtime AST and analysis environment
`ast` : a parsing AST

Module Ast

```
module Ast: sig .. end
```

Definition of the parser generated AST and the runtime AST
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
type operator =
| Plus
| Minus
| Times
| Divide
| Modulo
| And
| Or
```

binary operation operators

```
type comparator =
| LessThan
| LessThanEqual
| Equal
| GreaterThanEqual
| GreaterThan
| NotEqual
```

binary comparaison operators

```
type variable_location =
| GlobalVar of int * int
| LocalVar of int * int * int
```
location for a variable in the runtime AST for globals, unique id * an index into the global variables array for locals, unique id * an index into the current stackframe * an index into the stack
```
type replacement = string * expression
```
string replacement specification in a template instruction
```
type replacement_list = replacement list
```
list of replacements for a template instructions

```
type conditional_spec =
| Once
| When of expression
| Loop of string * expression
| CondLoop of expression * string * expression
```
conditional replacement criteria for a template instruction
```
type replacement_spec = string * conditional_spec *
replacement_list
```
a single instruction in a set of template instructions
```
type template_spec = string option * string
```
definition for a line in a template definition

```
type map_subtype =
| MapSubtype
| ArraySubtype
```
type of map variable, either a dictionary or an array

```
type variable_value =
| IntegerValue of int
| FloatValue of float
| StringValue of string
| BooleanValue of bool
| FunctionValue of string list * statement list
| MapValue of (string, variable_value) Hashtbl.t * map_subtype
| Void
```
variable values used in parsing AST

```
type runtime_variable_value =
| RIntegerValue of int
| RFloatValue of float
| RStringValue of string
| RBooleanValue of bool
| RFunctionValue of int * int * int * bool * runtime_statement
  list
  * (int * int, runtime_variable_value) Hashtbl.t option
  * runtime_expression option
| RLibraryFunction of lib_function_def
| RMapValue of (string, runtime_variable_value) Hashtbl.t *
  map_subtype
| RVoid
| RUndefined
```

variable values used in runtime AST

```
type runtime_env = {
    heap : (int *                (* heap, arary of tuple of uid      *)
    runtime_variable_va              and value
    lue) array;
    stackframes :                (* array of stackframes             *)
    runtime_variable_va
    lue array array;
    mutable closure_var          (* map of closure variables         *)
    s : (int * int,
    runtime_variable_va
    lue) Hashtbl.t
    option;
    gnames : string              (* array of global names,           *)
    array;                           indexed by uid
    mutable current_lin          (* file and line currently          *)
    e : string * int;                interpreted
    callstack : (string          (* callstack                        *)
    * int) Stack.t;
    mutable skip_callst          (* to indicate whether the call     *)
    ack_pop : bool;                  stack entry was skipped in a
                                     recursive call
}
```

The runtime environment. consists of a heap for globals and an array of stackframes to support nested functions

```
type lib_function_def = {
    name : string list;          (* namespace and name of            *)
                                     function
    args : string list;          (* list of arguments                *)
    num_args : int;              (* number of arguments              *)
    vararg : bool;               (* flag indicating whether the      *)
                                     last argument is vararg
    code : runtime_env    (* function call to invoked the    *)
    -> unit;                 function
}
```

Definition for a library function

```
type expression =
| Id of string
| VarArg of string
| BinaryOp of expression * operator * expression
| CompOp of expression * comparator * expression
| Not of expression
| FunctionCall of expression * expression list
| MapExpr of (string * expression) list
| ArrayExpr of expression list
| Value of variable_value
| UnboundVar of string
| Assignment of expression * expression
| Declaration of expression * expression
| MemberExpr of expression * expression
| PostFixSum of expression * int
| TernaryCond of expression * expression * expression
```

expressions used in parsing AST

```
type runtime_expression =
| RVariable of variable_location
| RVarArg of variable_location
| RBinaryOp of runtime_expression * operator * runtime_expression
| RCompOp of runtime_expression * comparator * runtime_expression
| RNot of runtime_expression
| RFunctionCall of runtime_expression * runtime_expression list
| RMapExpr of (string * runtime_expression) list
| RArrayExpr of runtime_expression list
| RValue of runtime_variable_value
| RAssignment of runtime_expression * runtime_expression
| RDeclaration of runtime_expression * runtime_expression
| RMemberExpr of runtime_expression * runtime_expression
| RPostFixSum of runtime_expression * int
| RTernaryCond of runtime_expression * runtime_expression *
  runtime_expression
```

expressions used in runtime AST

```
type statement =
| ForEach of string * expression * statement * (string * int)
| For of expression * expression * expression * statement
  * (string * int)
| ExpressionStatement of expression * (string * int)
| Break of (string * int)
| Continue of (string * int)
| Noop
| Return of expression * (string * int)
| If of expression * statement * statement * (string * int)
| TemplateDef of string * template_spec list * (string * int)
| Instructions of string * string list * replacement_spec list *
  (string * int)
| StatementBlock of statement list
| Program of statement list
| Import of string * (string * int)
| Switch of expression * statement list * (string * int)
| Case of expression option * (string * int)
| TryCatch of statement * string * statement * (string * int)
| TryFinally of statement * statement * (string * int)
| Throw of expression * (string * int)
```

statements used in parsing AST

```
type runtime_statement =
| RForEach of variable_location * runtime_expression *
  runtime_statement
  * (string * int)
| RFor of runtime_expression * runtime_expression *
  runtime_expression
  * runtime_statement * (string * int)
| RExpressionStatement of runtime_expression * (string * int)
| RBreak of (string * int)
| RContinue of (string * int)
| RNoop
| RReturn of runtime_expression * (string * int)
| RIf of runtime_expression * runtime_statement *
  runtime_statement
  * (string * int)
| RStatementBlock of runtime_statement list
| RProgram of runtime_statement list
| RSwitch of runtime_expression * runtime_statement list *
  (string * int)
| RCase of runtime_expression option * (string * int)
| RTryCatch of runtime_statement * variable_location *
  runtime_statement
  * (string * int)
| RTryFinally of runtime_statement * runtime_statement * (string
  * int)
| RThrow of runtime_expression * (string * int)
| RFastIterator of variable_location * int * int * int *
  runtime_statement
  * (string * int)
```

statements used in runtime AST

`val is_vararg : string -> bool`

determines if a variable is a varag

Returns true if the variable is a vararg, false otherwise

`varname` : the variable name

`val vararg_formalname : string -> string`

retuns the name for a vararg

`exception CFReturn of runtime_variable_value`

control flow exception for return instruction

`exception CFBreak`

control flow exception for break instruction

`exception CFContinue`

control flow exception for continue instruction

`exception CFUserException of runtime_variable_value * string`

exception generated by interpreted throw exception

Module Ast_info

`module Ast_info: sig .. end`

Pretty prints the runtime AST

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

`val statement_description : Ast.runtime_statement -> string`

Returns a pretty printed representation of the runtime AST

Returns a string with the pretty printed AST

`statement` : the top level statement (program)

`val print_ast : Ast.runtime_statement -> unit`

Pretty prints the representation of the runtime AST

Returns unit

`statement` : the top level statement (program)

Module Environment

`module Environment: sig .. end`

Operations on AST analysis and runtime environments.
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
module StringMap: Map.Make(String)
type var_info = int * int
```

Variable information, tuple of index into scope and unique id

```
type rec_varmap = {
    variable_map : var_info StringMap.t;  (* map of variable name to variable info *)
    parent : rec_varmap option;           (* parent scope variable map, or None if top level scope *)
}
```

represents variables map in a global or local scope, and reference to parent scope

```
type var_prop = {
    written_after_declared : bool;        (* is the variable assigned after it is declared *)
    read_after_declared : bool;           (* is the variable read after declared *)
    declaration_loc : string * int;       (* tuple of file where variable is declared and line number *)
}
```

Properties of variable locations

```
type label_pos = int * int * int * int
```

position of a label with within a template spec, tuple of start begin, start end, end begin, end ending

```
type template_spec_def = Ast.template_spec list * (string,
label_pos) Hashtbl.t *
        (string * int)
```

definition of a template specidifcation, used during validity checking. tuple of sepecfication list and map of labels to label position

```
type analysis_env = {
    globals : rec_varmap;                 (* map of global variables *)
    num_globals : int;                    (* number of globals *)
    locals : rec_varmap list;             (* recursive list of stack frames *)
    num_locals : int list;                (* number of locals in current stack frame *)
    sdepth : int;                         (* current stack depth *)
    max_depth : int;       (* maximum stack depth          *)
                              encountered
    errors : string        (* list of errors found during  *)
    list;                     analysis
    warnings : string      (* list of warning generated    *)
    list;                     during analysis
    unique_id : int;       (* counter for next unique id   *)
    names : string         (* list of names encountered    *)
    list;
    varprops : (int,       (* properties of variables      *)
    var_prop)
    Hashtbl.t;
    imported : string      (* list of files already imported *)
    list;
    templates :            (* map of template names to     *)
    (string,                  template definitions
    template_spec_def)
    Hashtbl.t;
    constants : (int,      (* map of variables unique id to *)
    Ast.runtime_variabl       declared value
    e_value) Hashtbl.t;
}
```

The analysis environment

```
val new_analysis_environment : unit -> analysis_env
```

returns a newly initialized analysis environment

Returns analysis_env

```
val set_constant_value : analysis_env -> int ->
Ast.runtime_variable_value -> unit
```

sets the declaration value for a variable

Returns unit

`env` : analysis environment

`uid` : unique id of variable

`value` : runtime value of variable

```
val get_constant_value : analysis_env -> int ->
Ast.runtime_variable_value
```

gets the constant value for a variable

Returns runtime value of variable

`env` : analysis environment

`uid` : unique id of variable

```
val is_constant : analysis_env -> int -> bool
```

returns whether the variable is a constant

Returns true if the variable is a constant

`env` : analysis environment

`uid` : unique id of variable

```
val declare_variable : analysis_env ->
       StringMap.key -> analysis_env * int
```

declare a variable if it does not exist or create a new entry and return new index
Returns a tuple of the modified environment and uid
`env` : analysis environment
`name` : name of variable to declare

```
val declare_variable_and_value : analysis_env ->
       StringMap.key ->
       Ast.runtime_variable_value -> analysis_env
```

declare a variable if it does not exist or create a new entry and return new index, then sets constant value
Returns the modified environment
`name` : name of variable to declare
`env` : analysis environment
`value` : the value to initialize the variable with

```
exception Variable_not_found of string
```

internal exception used during analysis

```
val resolve_variable : StringMap.key ->
       analysis_env -> Ast.variable_location
```

Find variable in analysis scope
Raises `Variable_not_found` when the variable is not found
Returns location
`name` : the variable name
`env` : the analysis environment

```
val uid_from_loc : Ast.variable_location -> int
```

returns uid from location
Returns the unique id of the variable

```
val resolve_variable_value : StringMap.key ->
       analysis_env ->
       Ast.runtime_variable_value * Ast.variable_location
```

Find variable and value in analysis scope
Raises `Variable_not_found` when the variable is not found
Returns tuple of value and location
`name` : the variable name
`env` : the analysis environment

```
val new_analysis_scope : analysis_env -> analysis_env
```

Setups a new scope within the same global or local scope
Returns a new analysis environment setup for the new scope
`env` : analysis environment

```
val pop_scope : analysis_env -> analysis_env
```

Pops the analysis scope
Returns a new environment with the last scope popped
`env` : analysis environment

```
val new_analysis_stackframe : analysis_env -> analysis_env
```

Create a new stackframe
Returns a new analysis environment with a new stackframe
`env` : analysis environment

```
val get_depth : analysis_env -> int
```

Returns the depth of the current stack frame
Returns the depth of the current stack frame, 0 indexed
`env` : analysis environment

```
val add_error : analysis_env ->
       string * int -> string -> analysis_env
```
Add an error to the analysis environemnt
Returns an analysis environment with the error added
`env` : the analysis environment
`codeloc` : a filename, line number tuple
`message` : the error message

```
val add_warning : analysis_env ->
       string * int -> string -> analysis_env
```
Add a warning to the analysis environemnt
Returns an analysis environment with the warning added
`env` : the analysis environment
`codeloc` : a filename, line number tuple
`message` : the warning message

```
val has_errors : analysis_env -> bool
```
Returns true if there are errors in the environment
Returns true if there are errors, false otherwise
`env` : the analysis environment

```
val add_import : analysis_env -> string -> analysis_env
```
adds an import to the list of imports
Returns the modified environment
`env` : analysis environment
`filename` : the filename

```
type var_op_type =
| ReadOp                   (* variable is read                  *)
| WriteOp                  (* variable is written               *)
| DeclareOp of             (* variable is declared              *)
  (string * int)
| DeclareWriteOp of        (* variable is declared and          *)
  (string * int)              written, used for function
                              args
```
type of operation performed on variable

```
val record_usage : analysis_env ->
       Ast.variable_location -> var_op_type -> unit
```
Records a variables property
Returns unit
`env` : analysis environment
`loc` : variable location

```
val add_template : analysis_env ->
       string ->
       Ast.template_spec list ->
       (string, label_pos) Hashtbl.t ->
       string * int -> analysis_env
```
Adds a template to the environment
Returns a new environment *
`name` : template name
`spec_list` : list of line specifications

`labels` : label positions

```
val has_import : analysis_env -> string -> bool
```

checks if a file has already been imported

Returns true if already imported, false otherwise

`env` : analysis environment

`filename` : the filename to check

```
val get_value : Ast.runtime_env -> Ast.variable_location ->
Ast.runtime_variable_value
```

Retrieves a value at a location

Returns the value at the selected location

`env` : a runtime environment

```
val set_value : Ast.runtime_env ->
       Ast.runtime_variable_value ->
       Ast.variable_location -> Ast.runtime_variable_value
```

Sets a value at a location

Returns the value that was set

`env` : a runtime environment

`value` : the value to set

```
val get_loc_name : Ast.runtime_env -> Ast.variable_location ->
string
```

Returns the name of a location

Returns the name of the variable at location loc

`env` : the runtime environment

Module Expression

```
module Expression: sig .. end
```

Evaluation of binary operations and comparaison of values Various helper functions for expression evaluation

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val list_of_array : Ast.runtime_variable_value ->
Ast.runtime_variable_value list
```

Converts a MapValue array to a list of values

Returns a list of values

`arr` : array

```
val string_of_value : Ast.runtime_variable_value -> string
```

Converts a value to a string

Returns a string representing the value

```
type valuetype =
| IntegerType
| FloatType
| BooleanType
| StringType
| FunctionType
| LibraryCallType
| MapType
| ArrayType
| VoidType
| NaNType
| UndefinedType
```

enumeration of a value's possible types

```
val value_type : Ast.runtime_variable_value -> valuetype
```

Returns a value's type
Returns the value's type

```
val string_of_value_type : Ast.runtime_variable_value -> string
```

returns a string name for a value's type
Returns string name for the value's type

```
type cast_type =
| IntegerCast of int * int
| FloatCast of float * float
| StringCast of string * string
| BoolCast of bool * bool
```

type to hold the result of casting two values to the same type

```
val cast_to_integer : Ast.runtime_variable_value -> int
```

cast a value to an integer
Raises `EInvalidCast` if the value cannot be cast
Returns an integer representation of the value
`value` : the runtime value

```
val cast_to_float : Ast.runtime_variable_value -> float
```

cast a value to a float
Raises `EInvalidCast` if the value cannot be cast
Returns an float representation of the value
`value` : the runtime value

```
val evaluate_op : Ast.runtime_variable_value ->
       Ast.runtime_variable_value -> Ast.operator ->
Ast.runtime_variable_value
```

Evaluate the operation
Returns the value that results from the operation
`value1` : the first value
`value2` : the second value
`operator` : the operator

```
val compare : Ast.runtime_variable_value ->
       Ast.comparator -> Ast.runtime_variable_value ->
Ast.runtime_variable_value
```

Implements comparaison of two values, according to the following semantics:
-Integer Integer Any Comparison of integer values -Float Float Any Comparison of float values -Float Integer Any Comparison of float values -String any type Float comparison of first value to second value, -Integer with non string values converted to strings -Both types are Booleans,== and != comparison of first value to second value -maps, arrays, functions,== and != comparison

of first value to second value -NaN or void == and != comparison of first value to second value - Different types == always returns false != always returns true
Returns a boolean value type
`v1` : the first value to compare
`op` : the comparaison operator
`v2` : the second value to compare

```
val opname : Ast.comparator -> string
val hashtbl_equal : (string, Ast.runtime_variable_value)
Hashtbl.t ->
       (string, Ast.runtime_variable_value) Hashtbl.t -> bool
val mismatched_compare : Ast.runtime_variable_value ->
       Ast.comparator -> Ast.runtime_variable_value ->
Ast.runtime_variable_value
val make_stackframe : int ->
       int ->
       bool ->
       Ast.runtime_variable_value list ->
       Ast.runtime_variable_value -> Ast.runtime_variable_value
array
```

Makes a stack frame from the supplied value list
Returns a stack frame (an array of values)
`size` : size of stack frame
`vararg` : true if the last argument is a vararg, false otherwise
`value_list` : list of values to add to the stack frame
`this` : the value of this

```
val array_of_value_list : Ast.runtime_variable_value list ->
Ast.runtime_variable_value
```

Creates an Array from a list of values
Returns a MapValue with the array
`value_list` : a list of values

Module Filename_util

```
module Filename_util: sig .. end
```

Filename utilities.
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val resolve_filename : string -> string -> string
```

converts a relative filename and path into an absolute filename
Returns absolute path of file
`dir` : relative of absolute path
`filename` : the filename to process

Module Interpreter

```
module Interpreter: sig .. end
```

The Jtemplate interpreter
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val interpret : Ast.runtime_env -> Ast.runtime_statement -> unit
```
Interpret a runtime AST
Returns unit
`env` : a runtime environment
```
val interpret_stmts : Ast.runtime_env -> Ast.runtime_statement
list -> unit
```
Interprets a list of statements
`env` : runtime environments
```
val evaluate : Ast.runtime_env ->
       Ast.runtime_expression -> Ast.runtime_env *
Ast.runtime_variable_value
```
Evaluates an expression
Returns a value
`env` : runtime environment
```
val resolve_func_this : Ast.runtime_env ->
       Ast.runtime_expression ->
       Ast.runtime_variable_value * Ast.runtime_variable_value
```
Resolves a function call by an expression into a function and a this object
Returns a tuple of the this object and the function
`env` : runtime environment
`fexpr` : the expression to analyze
```
val run_function : Ast.runtime_env ->
       Ast.runtime_variable_value list ->
       Ast.runtime_variable_value ->
       Ast.runtime_variable_value -> Ast.runtime_env *
Ast.runtime_variable_value
```
Runs a function
Returns a tuple of the environemt and return value
`env` : runtime environment
`value_list` : list of values to pass as arguments
`this` : this pointer
`func` : function
```
val evaluate_memb_expr_index : Ast.runtime_env ->
Ast.runtime_expression -> string * bool
```
Determines the value and type of expression for the last member of a member expression
Returns a tuple with the index of the expression and a boolean indicating whether it is an integer
`env` : the runtime environment
`index` : the expression to evaluate
```
val get_member_expr_map : Ast.runtime_env ->
       Ast.runtime_expression ->
       Ast.runtime_expression ->
       (string, Ast.runtime_variable_value) Hashtbl.t * string
```
Returns the hashmap that corresponds to the member expression
Returns the hashmap that corresponds to the member expression
`env` : the runtime environment
`expr` : the member expression (without the last member)

`index` : the index (the last member of the member expression)

```
val evaluate_expr_list : Ast.runtime_env ->
       Ast.runtime_expression list -> Ast.runtime_variable_value
list
```

Evaluates a list of expressions

`env` : the runtime environment

`expr_list` : an expression list

Module Jtemplate

```
module Jtemplate: sig .. end
```

Jtemplate initialization and launching of program

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

Module Library

```
module Library: sig .. end
```

Registration of libraries

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val register_for_analysis : Environment.analysis_env ->
Environment.analysis_env
```

Registers all library functions and params in an analysis environment

Returns a modified environment with all library functions registered

`env` : analysis environment

```
val register_for_runtime : Environment.analysis_env ->
Ast.runtime_env -> unit
```

Registers library functions into a runtime environment

Returns unit

`env` : analysis environment from which definitions will be transferred

`renv` : runtime environment into which definitions will be transferred

Module Library_builtin

```
module Library_builtin: sig .. end
```

Built in library implementation

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val initialize : Environment.analysis_env ->
       Ast.lib_function_def list * Environment.analysis_env
```

Entry point for library initialization

Returns a list of exported functions

Module Library_io

```
module Library_io: sig .. end
```
I / O library implementation
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
exception EIOPassthrough of string
```
internal error raised to indicate a inconsistent usage of a handle

```
type channelType =
| OutChannel of Pervasives.out_channel
| InChannel of Pervasives.in_channel * (string * bool)
```
type of I/O channel
```
val initialize : Environment.analysis_env ->
       Ast.lib_function_def list * Environment.analysis_env
```
Entry point for library initialization
Returns a list of exported functions

Module Library_string

```
module Library_string: sig .. end
```
String library implementation
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
val get_this : Ast.runtime_env -> string
```
Returns the value of this as a string
Raises `InternalError` is this is not a string
Returns the string value of this
`env` : the runtime environment
```
val indexOf : string -> string -> int
```
Returns the positing of a substring within a string
Returns the position of the substring in the string, or - 1 if not found
`str` : the string to search
`substr` : the substring to find
```
val initialize : 'a -> Ast.lib_function_def list * 'a
```
Entry point for library initialization
Returns a list of exported functions

Module Parser

```
module Parser: sig .. end
```
Jtemplate parser expression parsing adapted from ECMA-262 http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

```
type token =
| ID of string
| INT of int
| STRING of string
| REAL of float
| BOOLEAN of bool
| TEXT of string
| COMPOP of Ast.comparator
| IMPORT of bool
| FOREACH
| WHILE
| IF
| FOR
| ELSE
| TEMPLATE
| INSTRUCTIONS
| FUNCTION
| CONTINUE
| BREAK
| RETURN
| IN
| ONCE
| WHEN
| VAR
| EOF
| LBRACE
| RBRACE
| LPAREN
| RPAREN
| LBRACKET
| RBRACKET
| COMMA
| SEMICOLON
| COLON
| DOTDOTDOT
| DOT
| EQUALS
| NOT
| QUESTION
| PLUS
| MINUS
| TIMES
| DIVIDE
| MODULO
| AND
| OR
| VOID
| SWITCH
| CASE
| DEFAULT
| PLUSEQUALS
| MINUSEQUALS
| TIMESEQUALS
| DIVEQUALS
| MODEQUALS
| PLUSPLUS
| MINUSMINUS
| AT
| TRY
| CATCH
| THROW
| FINALLY
| PROTOTYPE
| OUTOFRANGENUMBER
```

Jtemplate parser expression parsing adapted from ECMA-262 http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

```
val parse_error : string -> unit
val get_env : unit -> string * int
val resolve_import : string * 'a * (string * 'b) -> string
val extract_stmt_list : Ast.statement -> Ast.statement list
```

Module Parser_util

```
module Parser_util: sig .. end
```

Routines to parse a file

Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

`val parse : Pervasives.in_channel -> string -> Ast.statement`
Parse a channel
Returns the parse AST
`name` : the name to use in error reporting
`val parse_filename : string -> Ast.statement`
Parse a filename
Returns the parse AST
`filename` : to parse

Module RuntimeError

`module RuntimeError: sig .. end`
This module defines runtime errors that are reported to the user
Author(s): Tony BenBrahim < tony.benbrahim at gmail.com >

`exception InternalError of string`
this error represents an unexpected condition, caused by a programming error in the interpreter implementation
`exception LibraryError of string`
this error is a generic error thrown by library routines to indicate an error condition, such as the incorrect type of a passed in argument
`exception LexerException of string * int * int`
this error is caused by an abnormal error caused by the lexer, such as an unterminated string *
`exception FatalExit`
marker exception to note that the program should exit, error has already been reported during analysis
`exception EIncompatibleTypes of string * string`
indicates that an assignment was attempted on two incompatible types
`exception EInvalidCast of string * string`
indicates that the value is not of the expected type
`exception EInvalidOperation of string * string`
indicates that an invalid operation was attempted on the specified types
`exception EInvalidComparaison of string * string * string`
indicates that an invalid comparaison was attempted on the given types
`exception ELeftSideIsNotAMap of string * string`
indicates that a member expression is not applied to a map
`exception ELeftSideIsNotAMap of string * string`
indicates that a member expression is not applied to a map
`exception ELeftSideCannotBeAssigned`
indicates an attempt at an assignment to something that is a not a variable or map
`exception EInvalidMember of string * string`
indicates that the map member did not evaluate to a string or integer
`exception EUndefinedMapMember of string`
indicates that a reference was made to a map member that does not exist
`exception EInvalidArrayIndex of string * string`
indicates a non integer array index

`exception EArrayIndexOutOfBounds of string`
indicates an out of bounds index
`exception ETypeMismatchInAssignment of string * string * string`
indicates that the type in the assignment does not match the declare type
`exception EMismatchedFunctionArgs of int * int`
indicates that an incorrect number of arguments were passed to a function
`exception ENotAFunction`
indicates an attempt to apply a function to a non function
`exception ENotACollectionType of string * string`
indicates applying for each on a non collection type
`exception EDefaultCaseShouldBeLast`
indicates that the default case should be last
`exception ParseException of string`
indicates a parsing error
`val string_of_error : exn -> string`
Returns an error message for an exception
Returns error message
`ex` : exception
`val display_error : exn -> string * int -> unit`
Displays an error to stdout
`err` : exception
`cloc` : tuple of file, line where error occured

Appendix B – Source Code

analysis.ml

```
(**
Create an optimized AST from the parsing phase AST

Pass 1:
- resolve all variable references, including closure variables
- process imports and add declarations to AST
- builds runtime AST
- convert template definitions / instructions
- evaluate operations on constants and replace with value in AST
- determine if variables are initialized with a constant value (rather than an expression),
if a variable is written after being declared and if it is ever read after being declared
- determine if a function is inlineable

Pass 2:
The second pass replaces all non function variables whose value have not been modified
with a constant value, and evaluates operations on constants , eliminates assignment
statements on constant values when the variable is not reassigned and not written,
inlines functions


@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)

(* This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*)


open Ast
open Environment
open Parser_util
open RuntimeError

(**
Prints all errors in an analysis environment and raises FatalExit if there are errors
@param env analysis environment
@return unit
@raise FatalExit if there are errors in the environment
*)
let check_errors env =
  List.fold_left (fun _ error -> print_string ("ERROR: "^error^"\n")) () (List.rev env.errors);
  if List.length env.errors > 0 then raise RuntimeError.FatalExit
  else ()

(**
Generates additional warnings about unused variables, then prints all warnings
@param env analysis environment
@return analysis environment with newly added warnings
```

```
*)
let check_warnings env =
  (* check for unused templates *)
  let env = Hashtbl.fold (fun name spec env ->
            let (_, _, cloc) = spec
            in Environment.add_warning env cloc ("Template definition '" ^ name ^ "' is never
used.")
      ) env.templates env
  (* check variables usage *)
  in let rec loop_names env max names = function
    | n when n = max -> env
    | n ->
        let env =
          try
            let varprop = Hashtbl.find env.varprops n
            in let (_, line) = varprop.declaration_loc
            in match line with
            | 0 -> env
            | _ ->
                let env = if varprop.read_after_declared = false && line!= 0 then
                    add_warning env varprop.declaration_loc ("Variable "^(List.hd names)^" is
never read.")
                  else
                    env
                in env
          with Not_found ->
              env
        in loop_names env max (List.tl names) (n + 1)
  in let env = loop_names env (List.length env.names) env.names 0
  (* print warnings *)
  in List.fold_left (fun _ warning -> print_string ("WARNING: "^warning^"\n")) () (List.rev
env.warnings);
  env

(**
Prints information about names found during analysis
@param env analysis environment
@return unit
*)
let print_name_info env =
  let _ = List.fold_left (fun ind name ->
            print_int ind;
            let (read, written, analyzed, (file, line)) =
              try
                let vp = Hashtbl.find env.varprops ind
                in (vp.read_after_declared, vp.written_after_declared, true, vp.declaration_loc)
              with Not_found -> (false, false, false, ("", 0))
            in
            print_string (" "^name^" read="^(string_of_bool read)^" written="^(string_of_bool
written)
                ^ " analyzed="^(string_of_bool analyzed)^" ("
                ^( Filename.basename file)^","^(string_of_int line)^")\n") ;
            ind + 1) 0 env.names
  in ()

(**
FIRST PASS
- resolve all variable references, including closure variables
- process imports and add declarations to AST
- builds runtime AST
- convert template definitions / instructions
- evaluate operations on constants and replace with value in AST
- determine if variables are initialized with a constant value (rather than an expression)
- determine if a variable is written after being declared and if it is ever read after being
declared
- determine if a function is inlineable
*)

(**internal exception to signal an error in template processing. *)
exception TemplateError of string

(**
Checks for invalid nesting in a template specification
@param template_spec the template spec to check
@return an list of tuples containing the label and line offset where conflicts where found
**)
let rec check_template_nesting template_spec =
  let labels = Hashtbl.create 10
  (* get start and end for each label *)
  in let (_, errors) = List.fold_left(fun acc spec ->
            let (line, errors) = acc
            in let (name_opt, _) = spec
            in match name_opt with
            | None -> (line + 1, errors)
            | Some label ->
                try
                  let (start, end_start, ending, end_ending) = Hashtbl.find labels label
                  in let (new_pos, errors) =
                    if line = end_start + 1 then ((start, line, line, line), errors)
                    else if line = end_ending + 1 then ((start, end_start, ending, line),
errors)
                    else if ending = end_start then ((start, end_start, line, line), errors)
                    else ((start, end_start, ending, end_ending), (label, line):: errors)
                  in Hashtbl.replace labels label new_pos;
                  (line + 1, errors)
                with Not_found ->
                    Hashtbl.add labels label (line, line, line, line);
                    (line + 1, errors))
      (0,[]) template_spec
  (* find overlapping labels *)
  in let errors = Hashtbl.fold ( fun label pos list ->
            let (start1, _, _, ending1) = pos
            in Hashtbl.fold(fun label pos list ->
                    let (start2, _, _, ending2) = pos
                    in if start2 > start1 && start2 < ending1 && ending2 > ending1 then (label,
ending2):: list
                    else list
              ) labels list
      ) labels errors
  in (labels, errors)

(**
Generate a set of statements corresponding to a template instruction
@param instruction instruction AST
@param env runtime environment
@return a runtime statement for the instruction defining a function
*)
and generate_template_instr_function instruction env =
  let generate_instructions template_specs labels replacement_list cloc args =
    let find_replacement name =
      let rec loop = function
        | [] -> raise Not_found
        | (n, condexpr, repl):: tl when n = name -> (condexpr, repl)
        | hd:: tl -> loop tl
      in loop replacement_list
    in let make_repl_vars replacements =
      let rec loop substrings expressions = function
        | [] -> (ArrayExpr(List.rev substrings), ArrayExpr(List.rev expressions))
        | hd:: tl ->
            let (string, expr) = hd
            in loop (Value(StringValue(string)):: substrings) (expr:: expressions) tl
      in loop [] [] replacements
    in let array = Array.of_list template_specs
    in  let rec loop name result index endindex args target_label substrings_var repl_var =
      if index = endindex then
        let result = (Return(Id("result"), cloc)):: result
        in ExpressionStatement(Declaration(Id(name), Value(FunctionValue(args, List.rev
result))), cloc)
      else
        let (label_opt, line) = array.(index)
        in match label_opt with
        | None ->
            loop name (ExpressionStatement(Assignment(Id("result"), BinaryOp(Id("result"), Plus,
Value(StringValue(line)))), cloc):: result) (index + 1) endindex args target_label
substrings_var repl_var
        | Some label ->
            if label = target_label then
              let call = FunctionCall(MemberExpr(Value(StringValue(line)),
Value(StringValue("mreplace"))),[substrings_var; repl_var])
              in loop name (ExpressionStatement(Assignment(Id("result"), BinaryOp(Id("result"),
Plus, call)), cloc):: result) (index + 1) endindex args target_label substrings_var repl_var
            else
              try
                let (condexpr, replacements) = find_replacement label
                in let (substrings, replexprs) = make_repl_vars replacements
                in let (start, _, _, ending) = Hashtbl.find labels label
                in let arg_array = match condexpr with
                  | Once | When(_) -> []
                  | Loop(iname, _) | CondLoop(_, iname, _) -> [iname]
                in let earg_array = match condexpr with
                  | Once | When(_) -> []
                  | Loop(iname, _) | CondLoop(_, iname, _) -> [Id(iname)]
                in let stmt1 = loop label [ExpressionStatement(Declaration(Id("result"),
Value(StringValue(""))), cloc)]
                    start (ending + 1) arg_array label substrings replexprs
                in let call = ExpressionStatement(Assignment(Id("result"),
BinaryOp(Id("result"), Plus, FunctionCall(Id(label), earg_array))), cloc)
                in let stmt2 = match condexpr with
                  | Once -> call
                  | When(cexpr) -> If(cexpr, call, Noop, cloc)
                  | Loop(iname, iexpr) -> ForEach(iname, iexpr, call, cloc)
                  | CondLoop(cexpr, iname, iexpr) -> If(cexpr, ForEach(iname, iexpr, call,
cloc), Noop, cloc)
                in loop name (stmt2:: stmt1:: result) (ending + 1) endindex args target_label
substrings_var repl_var
              with Not_found ->
                  raise(TemplateError ("could not find instruction for label '"^label^"'"))
    in let (name, args, specs, cloc) = instruction
    in loop name [ExpressionStatement(Declaration(Id("result"), Value(StringValue(""))), cloc)]
      0 (List.length template_specs) args "" (ArrayExpr([])) (ArrayExpr([]))
  in let (name, args, replacements, cloc) = instruction
  in  try
    let (template_specs, labels, _) = Hashtbl.find env.templates name
    in let (rstmt, env) =
      try
        let stmt = generate_instructions template_specs labels replacements cloc args
        in analyze_variables env stmt
      with TemplateError message ->
          (RNoop, Environment.add_error env cloc message)
    in Hashtbl.remove env.templates name; (rstmt, env)
  with
  | Not_found -> (RNoop, env)
  | Variable_not_found(name) -> (RNoop, env)
(**
Filters an ast, returning only a list of declaration and import statement
@param stmts the statement list to process
@return a statement list containing only declarations and imports
*)
and filter_imported_ast stmts =
  let rec loop result = function
    | [] -> List.rev result
    | stmt:: tl ->
        (match stmt with
          | ExpressionStatement(Declaration(_, _), _) | Import (_, _) ->
              loop (stmt:: result) tl
          | _ -> loop result tl
        ) in
  loop [] stmts

(** find declarations and resolve references to variables. Since declarations are
visible in the entire scope in which they are defined, and not just after they are
declared, a breadth first search is necessary before recursively processing children
statements
@param env an analysis environment
@param ast the intermediate ast
@return an ast where all variables have been resolved to an absolute location, either
on a stack or in the global heap and an environment containing information about all
variables
*)
and analyze_variables env ast =
  (**
  recursively searches an expression for declarations
  @param env the analysis environment
  @param expr the expression to search
  @param cloc the location in code of the expression
  @return an environment with any declared name added
  *)
  let rec find_decl_in_expr env expr cloc =
    match expr with
    | Declaration(expr1, expr2) ->
        let env = (match expr1 with
            | Id(name) -> let (env, uid) = Environment.declare_variable env name in env
            | MemberExpr(_, _) -> env
            | _ -> Environment.add_error env cloc "Left side cannot be assigned to")
        in find_decl_in_expr (find_decl_in_expr env expr1 cloc) expr2 cloc
    | Not(expr) | PostFixSum (expr, _) ->
```

```
        find_decl_in_expr env expr cloc
    | Assignment(expr1, expr2) | BinaryOp(expr1, _, expr2) | CompOp(expr1, _, expr2)
    | MemberExpr(expr1, expr2) ->
        find_decl_in_expr (find_decl_in_expr env expr1 cloc) expr2 cloc
    | TernaryCond(expr1, expr2, expr3) ->
        find_decl_in_expr(find_decl_in_expr (find_decl_in_expr env expr1 cloc) expr2 cloc) expr3
cloc
    | FunctionCall(expr, expr_list) ->
        List.fold_left (fun env expr ->
                find_decl_in_expr env expr cloc) (find_decl_in_expr env expr cloc) expr_list
    | MapExpr(proplist) ->
        List.fold_left (fun env prop -> let (_, expr) = prop in find_decl_in_expr env expr cloc)
env proplist
    | ArrayExpr(expr_list) ->
        List.fold_left (fun env expr -> find_decl_in_expr env expr cloc) env expr_list
    | Value(_) | UnboundVar(_) | Id(_) | VarArg(_) -> env
  (** finds all the declarations in the statement and registers them in the environment
  @param env analysis environment
  @param stmt the statement to be analyzed
  @return an analysis environment where all names found in assigments are registered
  *)
  and find_declarations_in_stmt env stmt =
    match stmt with
    | ExpressionStatement(expr, cloc) | Throw(expr, cloc) | Switch(expr, _, cloc) | Case(Some
expr, cloc)
    | If(expr, _, _, cloc) | Return(expr, cloc) | ForEach(_, expr, _, cloc) ->
        find_decl_in_expr env expr cloc
    | TryFinally(_, _, _) | TryCatch(_, _, _, _) | StatementBlock(_) | Case(None, _) |
Continue(_)
    | Break(_) | Noop | Program(_) | Import(_) | For(_, _, _, _, _) -> env
    | Instructions(name, _, _, _) -> let (env, _) = Environment.declare_variable env name in env
    | TemplateDef(name, spec_list , cloc) ->
        let (labels, errors) = check_template_nesting spec_list
        in match errors with
        | [] -> Environment.add_template env name spec_list labels cloc
        | error_list ->
            List.fold_left (fun env label_offset ->
                    let (label, offset) = label_offset
                    in let (file, line) = cloc
                    in Environment.add_error env (file, line + offset + 1) ("Invalid nesting of
labels for label '"^label^"'")
              ) env error_list
  (**
  Find all all variables in statement list whose stack depth is lower than that given
  @param stmt_list a list of statements to be searched recursively
  @param stack_depth the stack depth for the given statement
  @return Some map of unique id to Void of all variables found or None if none where found
  *)
  and get_closure_vars stmt_list stack_depth =
    let add loc = function
      | None -> let t = Hashtbl.create 2 in Hashtbl.replace t loc RUndefined; Some t
      | Some t -> Hashtbl.replace t loc RUndefined; Some t
    in
    let rec find_in_expr result = function
      | RVariable(LocalVar(_, sdepth, ind)) | RVarArg(LocalVar(_, sdepth, ind)) ->
          if sdepth = stack_depth then result else add (sdepth, ind) result
      | RValue(_) | RVariable(GlobalVar(_, _)) | RVarArg(GlobalVar(_, _)) -> result
      | RPostFixSum(e, _) | RNot(e) -> find_in_expr result e
      | RBinaryOp(e1, _, e2) | RCompOp(e1, _, e2) | RAssignment(e1, e2) | RDeclaration(e1, e2)
      | RMemberExpr(e1, e2) ->
          find_in_expr (find_in_expr result e1) e2
      | RTernaryCond(e1, e2, e3) ->
          find_in_expr(find_in_expr (find_in_expr result e1) e2) e3
      | RArrayExpr(elist) -> List.fold_left (fun r e -> find_in_expr r e) result elist
      | RMapExpr(proplist) ->
          List.fold_left (fun r prop -> let (_, e) = prop
                  in find_in_expr r e) result proplist
      | RFunctionCall(e, elist) ->
          List.fold_left (fun r e -> find_in_expr r e)
            (find_in_expr result e) elist
    and process result = function
      | RNoop | RContinue(_) | RCase(None, _) | RBreak(_) | RFastIterator _ -> result
      | RStatementBlock(slist) -> loop result slist
      | RTryCatch(s1, _, s2, _) | RTryFinally(s1, s2, _) ->
          process (process result s1) s2
      | RReturn (e, _) | RThrow(e, _) | RCase(Some e, _) | RExpressionStatement(e, _) ->
          find_in_expr result e
      | RFor(e1, e2, e3, stmt, _) ->
          process (find_in_expr(find_in_expr (find_in_expr result e1) e2) e3) stmt
      | RIf(e, stmt1, stmt2, _) -> process(process(find_in_expr result e) stmt1)stmt2
      | RSwitch(e, stmts, _) -> loop (find_in_expr result e) stmts
      | RForEach(_, e, stmt, _) -> process (find_in_expr result e) stmt
      | RProgram(_) -> raise (RuntimeError.InternalError "unexpected statement in closure
processing")
    and loop result = function
      | [] -> result
      | stmt:: list -> loop (process result stmt) list
    in loop None stmt_list
  (**
  Convert a parsing value to a runtime value
  *)
  and convert_value env cloc = function
    | StringValue(v) -> (RStringValue(v), env)
    | IntegerValue(v) -> (RIntegerValue(v), env)
    | FloatValue(v) -> (RFloatValue(v), env)
    | BooleanValue(v) -> (RBooleanValue(v), env)
    | Void -> (RVoid, env)
    | MapValue(h, s) ->
        (RMapValue(Hashtbl.fold (fun k v h ->
                    let (repl, env) = convert_value env cloc v
                    in Hashtbl.replace h k repl ; h ) h (Hashtbl.create 10), s), env)
    | FunctionValue(arg_list, stmt_list) ->
        let rec analyze_vararg has_vararg namelist env = function
          | [] -> (has_vararg, List.rev namelist, env)
          | name::[] when is_vararg name -> analyze_vararg true ((vararg_formalname name)::
namelist) env []
          | name:: tl ->
              let env =
                (if is_vararg name then
                    Environment.add_error env cloc "vararg must be last argument"
                  else
                    env)
              in analyze_vararg has_vararg (name:: namelist) env tl
        in let (has_vararg, arg_list, env) = analyze_vararg false [] env arg_list
        in let env = Environment.new_analysis_stackframe env
        in let (env, _) = Environment.declare_variable env "this"
        in let _ = Environment.record_usage env (LocalVar(env.unique_id - 1, 0, 0)) (DeclareOp
cloc)
```

```
        in let _ = Environment.record_usage env (LocalVar(env.unique_id - 1, 0, 0)) WriteOp
        in let _ = Environment.record_usage env (LocalVar(env.unique_id - 1, 0, 0)) ReadOp
        in let env = List.fold_left
            (fun env name ->
                  let (env, _) = Environment.declare_variable env name
                  in let _ = Environment.record_usage env (LocalVar(env.unique_id - 1 , 0, 0))
(DeclareOp cloc)
                  in let _ = Environment.record_usage env (LocalVar(env.unique_id - 1, 0, 0))
WriteOp
                  in env) env arg_list
        in let (stmt_list, env) = analyze_variables_in_block env stmt_list
        in let closure_vars = get_closure_vars stmt_list (Environment.get_depth env)
        in let inline_expr = match (stmt_list, closure_vars) with
          | (RReturn(RValue(RFunctionValue(_,_,_,_,_,Some h,_)),_)::[],_) -> None
          | (RReturn(expr, _)::[], None) | (RExpressionStatement(expr, _)::[], None) -> Some
expr
          | ([], None) -> Some (RValue(RVoid))
          | _ -> None
        in (RFunctionValue(List.hd env.num_locals, Environment.get_depth env, List.length
arg_list, has_vararg,
            stmt_list, closure_vars, inline_expr), Environment.pop_scope env)
  (**
  Replaces all variables in expression with absolute locations
  Also sets up function definitions
  @param env analysis environment
  @param expr expression
  @param location in code
  @return new expression with variables replaced with absolute location
  *)
  and resolve_expr env expr cloc =
    let rec resolve_expr_sub env expr op_type =
      match expr with
      | Id(name) ->
          let loc = Environment.resolve_variable name env
          in let _ = record_usage env loc op_type
          in (RVariable(loc), env)
      | VarArg(name) ->
          let loc = Environment.resolve_variable name env
          in (RVarArg(loc), env)
      | BinaryOp(e1, op, e2) ->
          let (expr1, env) = resolve_expr_sub env e1 ReadOp
          in let (expr2, env) = resolve_expr_sub env e2 ReadOp
          in (match (expr1, expr2) with
            | (RValue(v1), RValue(v2)) ->
                (try
                  (RValue(Expression.evaluate_op v1 v2 op), env)
                with
                | Division_by_zero -> (RBinaryOp(expr1, op, expr2), Environment.add_error env
cloc "division by zero")
                | EInvalidOperation(_, t) ->
                    (RBinaryOp(expr1, op, expr2), Environment.add_error env cloc ("invalid
operation for "^t^" types"))
                | EIncompatibleTypes(t1, t2) ->
                    (RBinaryOp(expr1, op, expr2), Environment.add_error env cloc ("incompatible
types "^t1^" and "^t2))
                )
            | _ -> (RBinaryOp(expr1, op, expr2), env))
      | CompOp(e1, op, e2) ->
          let (expr1, env) = resolve_expr_sub env e1 ReadOp
          in let (expr2, env) = resolve_expr_sub env e2 ReadOp
          in (match (expr1, expr2) with
            | (RValue(v1), RValue(v2)) ->
                (try
                  (RValue(Expression.compare v1 op v2), env)
                with
                | EInvalidComparaison(_, _, _) ->
                    (RCompOp(expr1, op, expr2), Environment.add_error env cloc ("invalid
comparaison"))
                )
            | _ -> (RCompOp(expr1, op, expr2), env))
      | Not(e) ->
          let (expr, env) = resolve_expr_sub env e ReadOp
          in (match expr with
            | RValue(RBooleanValue(b)) -> (RValue(RBooleanValue(not b)), env)
            | _ -> (RNot(expr), env))
      | FunctionCall(e, el) ->
          let rec has_unbound_var = function
            | [] -> false
            | UnboundVar(_):: tl -> true
            | _:: tl -> has_unbound_var tl
          in if has_unbound_var el then
            let rec unbound_list result = function
              | [] -> List.rev result
              | UnboundVar(name):: tl -> unbound_list (name:: result) tl
              | _:: tl -> unbound_list result tl
            in let rec bound_list result = function
              | [] -> List.rev result
              | UnboundVar(name):: tl ->
                  let variable =
                    if is_vararg name then
                      VarArg(vararg_formalname name)
                    else
                      Id(name)
                  in bound_list (variable:: result) tl
              | expr:: tl -> bound_list (expr:: result) tl
            in resolve_expr_sub env (Value(FunctionValue(unbound_list [] el,
[Return(FunctionCall(e, bound_list [] el), cloc)]))) ReadOp
          else
            let (expr, env) = resolve_expr_sub env e ReadOp
            in let (expr_list, env) = List.fold_left(fun acc expr -> let (lst, env) = acc
                  in let (expr, env) = resolve_expr_sub env expr ReadOp in (expr:: lst,
env)) ([], env) el
            in (RFunctionCall(expr, List.rev expr_list), env)
      | MapExpr(prop_list) ->
          let (prop_list, env) = List.fold_left(fun acc prop -> let (lst, env) = acc
                in let (name, expr) = prop
                in let (expr, env) = resolve_expr_sub env expr ReadOp
                in ((name, expr):: lst, env)) ([], env) prop_list
          in (RMapExpr(List.rev prop_list), env)
      | ArrayExpr(el) ->
          let (expr_list, env) = List.fold_left(fun acc expr -> let (lst, env) = acc
                in let (expr, env) = resolve_expr_sub env expr ReadOp
                in (expr:: lst, env)) ([], env) el
          in (RArrayExpr(List.rev expr_list), env)
      | Value(v) -> let (repl, env) = convert_value env cloc v in (RValue(repl), env)
      | Assignment(e1, e2) ->
          let (expr1, env) = resolve_expr_sub env e1 WriteOp
          in let (expr2, env) = resolve_expr_sub env e2 ReadOp
          in (RAssignment(expr1, expr2), env)
      | Declaration(e1, e2) ->
          let (expr1, env) = resolve_expr_sub env e1 (DeclareOp cloc)
          in let (expr2, env) = resolve_expr_sub env e2 ReadOp
          in let _ = match (expr1, expr2) with
            | (RVariable(loc), RValue(value)) ->
                let uid = uid_from_loc loc
                in Environment.set_constant_value env uid value
            | _ -> ()
          in (RDeclaration(expr1, expr2), env)
      | TernaryCond(e1, e2, e3) ->
          let (e1, env) = resolve_expr_sub env e1 ReadOp
          in let (e2, env) = resolve_expr_sub env e2 ReadOp
          in let (e3, env) = resolve_expr_sub env e3 ReadOp
          in (RTernaryCond(e1, e2, e3), env)
      | MemberExpr(e1, e2) ->
          let (expr1, env) = match op_type with
            | DeclareOp(loc) -> resolve_expr_sub env e1 (DeclareWriteOp(loc))
            | _ -> resolve_expr_sub env e1 op_type
          in let (expr2, env) = resolve_expr_sub env e2 ReadOp
          in (RMemberExpr(expr1, expr2), env)
      | PostFixSum(e, inc) ->
          let (expr, env) = resolve_expr_sub env e WriteOp
          in (RPostFixSum(expr, inc), env)
      | UnboundVar(_) ->
          (RValue(RVoid), Environment.add_error env cloc "Unexpected unbound var")
    in
    try
      resolve_expr_sub env expr ReadOp
    with
    | Variable_not_found(name) -> (RValue(RVoid), Environment.add_error env cloc ("Undefined
variable '"^name^"'"))
  (**
  Replace variables in all statements in a list with an absolute location
  @param env analysis environment
  @param stmt_list the statement list to process
  @return a tuple consisting of a new statement list with all variables replaced with an
absolute location
  and an updated environment, possibly containing new errrors
  *)
  and replace_variables_in_block env stmt =
    let rec loop env stmt_list new_list =
      match stmt_list with
      | [] -> (List.rev new_list, env)
      | stmt:: tl ->
          let (stmt, env) = analyze_variables env stmt
          in loop env tl (stmt:: new_list)
    in loop env stmt []
  (**
  In the given statement list, replace all top level import statements
  @param env analysis enviroment
  @param stmt_list the statement list
  @return a tuple of the statement list with imports replaced by imported statements and
  a new environment, possibly containing new errors
  *)
  and process_imports env stmt_list =
    let rec loop result env = function
      | [] -> (List.rev result, env)
      | Import(filename, cloc):: tl ->
```

```
          (if Environment.has_import env filename then
              loop (Noop:: result) env tl
            else (
              let env = Environment.add_import env filename
              in let stmt = Parser_util.parse_filename filename
              in (match stmt with
                | Program(stmts) ->
                    let (stmts, env) = process_imports env (filter_imported_ast stmts)
                    in let result = List.fold_left (fun lst stmt -> stmt:: lst) result stmts
                    in loop result env tl
                | _ -> raise (RuntimeError.InternalError "Unexpected node from import"))))
      | stmt:: tl -> loop (stmt:: result) env tl
    in loop [] env stmt_list
  (**
  In the given statement list, find all top level declarations, then resolve all top
  level variables, then recurse into nested statement blocks
  @param env analysis environment
  @param stmt_list the statement list to process
  @return a tuple of modified statements and the modified analysis environment
  **)
  and analyze_variables_in_block env stmt_list =
    let (stmt_list, env) = process_imports env stmt_list
    in let env = List.fold_left(fun env stmt -> find_declarations_in_stmt env stmt) env
stmt_list
    in replace_variables_in_block env stmt_list
  and analyze_variables_in_stmt env stmt =
    let env = find_declarations_in_stmt env stmt
    in analyze_variables env stmt
  in
  match ast with
  | Program(stmt_list) ->
      let (ast, env) = analyze_variables_in_block env stmt_list
      in (RProgram(ast), env)
  | StatementBlock(stmt_list) ->
      let newenv = Environment.new_analysis_scope env
      in let (ast, newenv) = analyze_variables_in_block newenv stmt_list
      in (RStatementBlock(ast), Environment.pop_scope newenv)
  | Switch(expr, stmt_list, cloc) ->
      let newenv = Environment.new_analysis_scope env
      in let (expr, newenv) = resolve_expr newenv expr cloc
      in let (ast, newenv) = analyze_variables_in_block newenv stmt_list
      in (RSwitch(expr, ast, cloc), Environment.pop_scope newenv)
  | TryCatch(stmt1, name, stmt2, cloc) ->
      let newenv = Environment.new_analysis_scope env
      in let (stmt1, newenv) = analyze_variables_in_stmt newenv stmt1
      in let newenv = Environment.new_analysis_scope (Environment.pop_scope newenv)
      in let (newenv, _) = Environment.declare_variable newenv name
      in let (stmt2, newenv) = analyze_variables_in_stmt newenv stmt2
      in let loc = Environment.resolve_variable name newenv
      in (RTryCatch(stmt1, loc, stmt2, cloc), Environment.pop_scope newenv)
  | If(expr, stmt1, stmt2, cloc) ->
      let (expr, env) = resolve_expr env expr cloc
      in let newenv = Environment.new_analysis_scope env
      in let (stmt1, newenv) = analyze_variables_in_stmt newenv stmt1
      in let newenv = Environment.new_analysis_scope (Environment.pop_scope newenv)
      in let (stmt2, newenv) = analyze_variables_in_stmt newenv stmt2
      in (RIf(expr, stmt1, stmt2, cloc), Environment.pop_scope newenv)
  | TryFinally(stmt1, stmt2, cloc) ->
      let newenv = Environment.new_analysis_scope env
      in let (stmt1, newenv) = analyze_variables_in_stmt newenv stmt1
      in let newenv = Environment.new_analysis_scope (Environment.pop_scope newenv)
      in let (stmt2, newenv) = analyze_variables_in_stmt newenv stmt2
      in (RTryFinally(stmt1, stmt2, cloc), Environment.pop_scope newenv)
  | ForEach(name, expr, stmt, cloc) ->
      let (newenv, _) = Environment.declare_variable (Environment.new_analysis_scope env) name
      in let (expr, newenv) = resolve_expr newenv expr cloc
      in let (stmt, newenv) = analyze_variables_in_stmt newenv stmt
      in let loc = Environment.resolve_variable name newenv
      in (RForEach(loc, expr, stmt, cloc), Environment.pop_scope newenv)
  | For(expr1, expr2, expr3, stmt, cloc) ->
      let newenv = Environment.new_analysis_scope env
      in let newenv = find_decl_in_expr (find_decl_in_expr (find_decl_in_expr newenv expr1 cloc)
expr2 cloc) expr3 cloc
      in let (expr1, newenv) = resolve_expr newenv expr1 cloc
      in let (expr2, newenv) = resolve_expr newenv expr2 cloc
      in let (expr3, newenv) = resolve_expr newenv expr3 cloc
      in let (stmt, newenv) = analyze_variables_in_stmt newenv stmt
      in (RFor(expr1, expr2, expr3, stmt, cloc), Environment.pop_scope newenv)
  | Noop -> (RNoop, env)
  | ExpressionStatement(e, cloc) ->
      let (expr, env) = resolve_expr env e cloc
      in (RExpressionStatement(expr, cloc), env)
  | Return(e, cloc) ->
      let (e, env) = resolve_expr env e cloc
      in (RReturn(e, cloc), env)
  | Case(Some e, cloc) ->
      let (e, env) = resolve_expr env e cloc
      in (RCase(Some e, cloc), env)
  | Case(None, cloc) -> (RCase(None, cloc), env)
  | Throw(e, cloc) ->
      let (e, env) = resolve_expr env e cloc
      in (RThrow(e, cloc), env)
  | Break(cloc) -> (RBreak(cloc), env)
  | Continue(cloc) -> (RContinue(cloc), env)
  | Import(_, _) | TemplateDef(_, _, _) -> (RNoop, env)
  | Instructions(name, args, specs, cloc) ->
      generate_template_instr_function (name, args, specs, cloc) env

(**
SECOND PASS
- replace all constant declarations with Noop
- replace all constant variables with their value
- replace all constant expressions with the computed value
- replace all calls to inlineable functions with an expression
*)

(** replaces an expression from an inlined function with the corresponding
values from a function call expression list
@param depth the stack depth, for sanity checking
@param numargs the number of arguments
@param repl_exprs the expressions used in the call invocation
@param expr the inline expression
@return the inline expression with the arguments replacing the former local args
*)
let rec inline_expr_replace depth numargs repl_expr expr =
  let rec loop = function
    | RVariable(LocalVar(uid, d, ind)) when d = depth && ind <= numargs ->
        List.nth repl_expr (ind - 1)
    | RVariable(_) | RValue(_) | RVarArg(_) as e -> e
    | RBinaryOp(expr1, op, expr2) -> RBinaryOp(loop expr1, op, loop expr2)
    | RCompOp(e1, op, e2) -> RCompOp(loop e1, op, loop e2)
    | RNot(e) -> RNot(loop e)
    | RTernaryCond(e1, e2, e3) -> RTernaryCond(loop e1, loop e2, loop e3)
    | RDeclaration(e1, e2) -> RDeclaration(loop e1, loop e2)
    | RAssignment(e1, e2) -> RAssignment(loop e1, loop e2)
    | RMapExpr(props) -> RMapExpr(List.map (fun p -> let (name, e) = p in (name, loop e)) props)
    | RArrayExpr(elist) -> RArrayExpr(List.map(fun e -> loop e) elist)
    | RFunctionCall(e, elist) -> RFunctionCall(loop e, List.map(fun e -> loop e) elist)
    | RMemberExpr(e1, e2) -> RMemberExpr(loop e1, loop e2)
    | RPostFixSum(e, i) -> RPostFixSum(loop e, i)
  in loop expr
and
(**
Replace non modified variables with their declared value
@param env analysis environment
@param inline_uids list of inlined functions to avoid recursively inlining recursive inlinable
functions
@param expression expression to process
@return an expression with constant variables replaced by their value
*)
replace_constant env inline_uids = function
  | RVariable(loc) ->
      let uid = uid_from_loc loc
      in if Environment.is_constant env uid then
        try
          match (Environment.get_constant_value env uid) with
          | RFunctionValue(_) -> RVariable(loc)
          | value -> RValue(value)
        with Not_found -> RVariable(loc)
      else RVariable(loc)
  | RNot(expr) -> RNot(replace_constant env inline_uids expr)
  | RBinaryOp(expr1, op, expr2) ->
      let (e1, e2) = (replace_constant env inline_uids expr1, replace_constant env inline_uids
expr2)
      in (try match (e1, e2) with
        | (RValue(v1), RValue(v2)) -> RValue(Expression.evaluate_op v1 v2 op)
        | _ -> RBinaryOp(e1, op, e2)
      with _ -> RBinaryOp(e1, op, e2))
  | RCompOp(expr1, op, expr2) ->
      RCompOp(replace_constant env inline_uids expr1, op, replace_constant env inline_uids
expr2)
  | RValue(RFunctionValue(locals, depth, args, vararg, stmts, closvars, inline)) ->
      RValue(RFunctionValue(locals, depth, args, vararg, List.map(fun stmt -> pass2 env stmt)
stmts, closvars, inline))
  | RValue(_) | RPostFixSum(_) | RVarArg(_) as value -> value
  | RFunctionCall(expr, expr_list) ->
      let e = replace_constant env inline_uids expr
      in let e_list = List.map(fun e -> replace_constant env inline_uids e) expr_list
      in (match e with
        | RVariable(GlobalVar(uid, _))
        | RVariable(LocalVar(uid, _, _)) when is_constant env uid ->
            (match get_constant_value env uid with
              | RFunctionValue(_, depth, numargs, false, _, None, Some expr) ->
                  if List.exists (fun i -> i == uid) inline_uids then
                    RFunctionCall(e, e_list)
                  else
```

```
                    replace_constant env (uid:: inline_uids) (inline_expr_replace depth numargs
e_list expr)
              | _ -> RFunctionCall(e, e_list))
        | _ -> RFunctionCall(e, e_list))
  | RAssignment(expr1, expr2) -> RAssignment(expr1, replace_constant env inline_uids expr2)
  | RDeclaration(expr1, expr2) ->
      let expr2 = replace_constant env inline_uids expr2
      in (match (expr1, expr2) with
        | (RVariable(loc), RValue(value)) ->
            let uid = uid_from_loc loc
            in if is_constant env uid then
              match value with
              | RFunctionValue(_) -> RDeclaration(expr1, expr2)
              | _ -> (Environment.set_constant_value env uid value; RValue(RUndefined))
            else
              RDeclaration(expr1, expr2)
        | _ -> RDeclaration(expr1, expr2))
  | RMemberExpr(expr1, expr2) ->
      RMemberExpr(replace_constant env inline_uids expr1, replace_constant env inline_uids
expr2)
  | RArrayExpr(expr_list) ->
      RArrayExpr(List.map(fun e -> replace_constant env inline_uids e) expr_list)
  | RMapExpr(prop_list) ->
      RMapExpr(List.map (fun prop -> let (name, e) = prop in (name, replace_constant env
inline_uids e)) prop_list)
  | RTernaryCond(expr1, expr2, expr3) ->
      RTernaryCond(replace_constant env inline_uids expr1, replace_constant env inline_uids
expr2,
        replace_constant env inline_uids expr3)
(**
Looks for expressions where constants can be substituted
@param env analysis environment
@param stmt statement
*)
and pass2 env = function
  | RProgram(stmts) -> RProgram(List.map (fun stmt -> pass2 env stmt) stmts)
  | RStatementBlock(stmts) -> RStatementBlock(List.map (fun stmt -> pass2 env stmt) stmts)
  | RThrow(expr, cloc) -> RThrow(replace_constant env [] expr, cloc)
  | RCase(Some expr, cloc) -> RCase(Some (replace_constant env [] expr), cloc)
  | RReturn(expr, cloc) -> RReturn(replace_constant env [] expr, cloc)
  | RContinue(_) | RBreak(_) | RCase(None, _) | RNoop | RFastIterator _ as stmt -> stmt
  | RExpressionStatement(expr, cloc) ->
      (match replace_constant env [] expr with
        | RValue(RUndefined) -> RNoop
        | expr -> RExpressionStatement(expr, cloc))
  | RFor(expr1, expr2, expr3, stmt, cloc) ->
      let expr1 = replace_constant env [] expr1
      in let expr2 = replace_constant env [] expr2
      in let expr3 = replace_constant env [] expr3
      in let stmt = pass2 env stmt
      in (match (expr1, expr2, expr3) with
        | (RDeclaration(RVariable(vloc1), RValue(RIntegerValue(start))),
        RCompOp(RVariable(vloc2), LessThan, RValue(RIntegerValue(max))),
        RPostFixSum(RVariable(vloc3), inc)) when vloc1 = vloc2 && vloc1 = vloc3 ->
            RFastIterator(vloc1, start , max , inc, stmt , cloc)
        | (RDeclaration(RVariable(vloc1), RValue(RIntegerValue(start))),
        RCompOp(RVariable(vloc2), LessThan, RValue(RIntegerValue(max))),
        RAssignment(RVariable(vloc3), RBinaryOp(RVariable(vloc4), Plus,
RValue(RIntegerValue(inc)))))
```

```
        when vloc1 = vloc2 && vloc1 = vloc3 & vloc1 = vloc4 ->
            RFastIterator(vloc1, start , max , inc, stmt , cloc)
        | _ -> RFor(expr1, expr2 , expr3 , stmt , cloc))
  | RIf(expr, stmt1, stmt2, cloc) -> RIf(replace_constant env [] expr, pass2 env stmt1, pass2
env stmt2, cloc)
  | RTryFinally(stmt1, stmt2, cloc) -> RTryFinally(pass2 env stmt1, pass2 env stmt2, cloc)
  | RTryCatch(stmt1, v, stmt2, cloc) -> RTryCatch(pass2 env stmt1, v, pass2 env stmt2, cloc)
  | RSwitch(expr, stmts, cloc) -> RSwitch(replace_constant env [] expr,
        (List.map (fun stmt -> pass2 env stmt) stmts), cloc)
  | RForEach(v, expr, stmt, cloc) -> RForEach(v, replace_constant env [] expr, pass2 env stmt,
cloc)

(**
Analyzes an AST, generates a runtime AST
@param ast a parsing AST
@return a tuple of the runtime AST and analysis environment
*)
let analyze ast =
  let analyze_all env ast =
    let (rast, env) = analyze_variables env ast
    in let (rast, env) =
      (rast, {
          globals = env.globals;
          num_globals = env.num_globals;
          locals = env.locals;
          num_locals = env.num_locals;
          sdepth = env.sdepth;
          max_depth = env.max_depth;
          errors = env.errors;
          warnings = env.warnings;
          unique_id = env.unique_id;
          names = List.rev env.names;
          varprops = env.varprops;
          imported = env.imported;
          templates = env.templates;
          constants = env.constants;
        })
    in let (rast, env) = (pass2 env rast, env)
    in let _ = check_errors env
    in let env = check_warnings env
    in (rast, env)
  in let env = Environment.new_analysis_environment()
  in let env = Library.register_for_analysis env
  in analyze_all env ast
```

ast.ml

```
(**
Definition of the parser generated AST and the runtime AST

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details. s                                                     *)

(** binary operation operators *)
type operator = | Plus | Minus | Times | Divide | Modulo | And | Or

(** binary comparaison operators *)
type comparator =
  | LessThan
  | LessThanEqual
  | Equal
  | GreaterThanEqual
  | GreaterThan
  | NotEqual

(**
location for a variable in the runtime AST
for globals, unique id * an index into the global variables array
for locals, unique id * an index into the current stackframe * an index into the stack
*)
type variable_location =
  | GlobalVar of int * int | LocalVar of int * int * int

(** string replacement specification in a template instruction *)
type replacement =
  (string * expression)

(** list of replacements for a template instructions *)
and replacement_list =
  replacement list

(** conditional replacement criteria for a template instruction *)
and conditional_spec =
  | Once
  | When of expression
  | Loop of string * expression
  | CondLoop of expression * string * expression

(** a single instruction in a set of template instructions *)
and replacement_spec =
  (string * conditional_spec * replacement_list)

(** definition for a line in a template definition *)
and template_spec =
  ((string option) * string)

(** type of map variable, either a dictionary or an array *)
and map_subtype =
  | MapSubtype | ArraySubtype

(** variable values used in parsing AST *)
and variable_value =
  | IntegerValue of int
  | FloatValue of float
  | StringValue of string
  | BooleanValue of bool
  | FunctionValue of string list * statement list
  | MapValue of (string, variable_value) Hashtbl.t * map_subtype
  | Void
```

```
(** variable values used in runtime AST *)
and runtime_variable_value =
  | RIntegerValue of int
  | RFloatValue of float
  | RStringValue of string
  | RBooleanValue of bool
  | RFunctionValue of int * int * int * bool * runtime_statement list * (((int * int),
runtime_variable_value) Hashtbl.t) option * runtime_expression option
  | RLibraryFunction of lib_function_def
  | RMapValue of (string, runtime_variable_value) Hashtbl.t * map_subtype
  | RVoid
  | RUndefined

(**
The runtime environment.
consists of a heap for globals and an array of stackframes to support nested functions
*)
and runtime_env =
  { heap : (int * runtime_variable_value) array; (** heap, arary of tuple of uid and value *)
    stackframes : (runtime_variable_value array) array; (** array of stackframes *)
    mutable closure_vars :
    (((int * int), runtime_variable_value) Hashtbl.t) option; (** map of closure variables *)
    gnames : string array; (** array of global names, indexed by uid *)
    mutable current_line : (string * int);  (** file and line currently interpreted *)
    callstack : (string * int) Stack.t; (** callstack *)
    mutable skip_callstack_pop: bool; (** to indicate whether the call stack entry was skipped
in a recursive call *)
  }

(**
Definition for a library function
*)
and lib_function_def =
  { name : string list; (** namespace and name of function *)
    args : string list; (** list of arguments *)
    num_args : int; (** number of arguments *)
    vararg : bool; (** flag indicating whether the last argument is vararg *)
    code : runtime_env -> unit (** function call to invoked the function *)
  }

(** expressions used in parsing AST *)
and expression =
  | Id of string
  | VarArg of string
  | BinaryOp of expression * operator * expression
  | CompOp of expression * comparator * expression
  | Not of expression
  | FunctionCall of expression * expression list
  | MapExpr of (string * expression) list
  | ArrayExpr of expression list
  | Value of variable_value
  | UnboundVar of string
  | Assignment of expression * expression
  | Declaration of expression * expression
  | MemberExpr of expression * expression
  | PostFixSum of expression * int
  | TernaryCond of expression * expression * expression
```

```
(** expressions used in runtime AST *)
and runtime_expression =
  | RVariable of variable_location
  | RVarArg of variable_location
  | RBinaryOp of runtime_expression * operator * runtime_expression
  | RCompOp of runtime_expression * comparator * runtime_expression
  | RNot of runtime_expression
  | RFunctionCall of runtime_expression * runtime_expression list
  | RMapExpr of (string * runtime_expression) list
  | RArrayExpr of runtime_expression list
  | RValue of runtime_variable_value
  | RAssignment of runtime_expression * runtime_expression
  | RDeclaration of runtime_expression * runtime_expression
  | RMemberExpr of runtime_expression * runtime_expression
  | RPostFixSum of runtime_expression * int
  | RTernaryCond of runtime_expression * runtime_expression * runtime_expression

(** statements used in parsing AST *)
and statement =
  | ForEach of string * expression * statement * (string * int)
  | For of expression * expression * expression * statement * (string * int)
  | ExpressionStatement of expression * (string * int)
  | Break of (string * int)
  | Continue of (string * int)
  | Noop
  | Return of expression * (string * int)
  | If of expression * statement * statement * (string * int)
  | TemplateDef of string * template_spec list * (string * int)
  | Instructions of string * string list * replacement_spec list
  * (string * int)
  | StatementBlock of statement list
  | Program of statement list
  | Import of string * (string * int)
  | Switch of expression * statement list * (string * int)
  | Case of expression option * (string * int)
  | TryCatch of statement * string * statement * (string * int)
  | TryFinally of statement * statement * (string * int)
  | Throw of expression * (string * int)

(** statements used in runtime AST *)
and runtime_statement =
  | RForEach of variable_location * runtime_expression * runtime_statement  * (string * int)
  | RFor of runtime_expression * runtime_expression * runtime_expression  * runtime_statement *
(string * int)
  | RExpressionStatement of runtime_expression * (string * int)
  | RBreak of (string * int)
  | RContinue of (string * int)
  | RNoop
  | RReturn of runtime_expression * (string * int)
  | RIf of runtime_expression * runtime_statement * runtime_statement * (string * int)
  | RStatementBlock of runtime_statement list
  | RProgram of runtime_statement list
  | RSwitch of runtime_expression * runtime_statement list * (string * int)
  | RCase of runtime_expression option * (string * int)
  | RTryCatch of runtime_statement * variable_location * runtime_statement  * (string * int)
  | RTryFinally of runtime_statement * runtime_statement * (string * int)
  | RThrow of runtime_expression * (string * int)
  | RFastIterator of variable_location * int * int * int * runtime_statement * (string * int)
```

```
(**
determines if a variable is a varag
@param varname the variable name
@return true if the variable is a vararg, false otherwise
*)
let is_vararg varname = varname.[0] = '['

(** retuns the name for a vararg *)
let vararg_formalname varname =
  String.sub varname 1 ((String.length varname) - 1)

(** control flow exception for return instruction *)
exception CFReturn of runtime_variable_value

(** control flow exception for break instruction *)
exception CFBreak

(** control flow exception for continue instruction *)
exception CFContinue

(** exception generated by interpreted throw exception *)
exception CFUserException of runtime_variable_value * string
```

ast_info.ml

```
(**
Pretty prints the runtime AST

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)

(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Ast

(**
Returns a pretty printed representation of the runtime AST
@param statement the top level statement (program)
@return a string with the pretty printed AST
*)
let statement_description statement =
  let rec statement_descriptionl level statement =
    let rec get_closure_vars = function
      | None -> "None"
      | Some t -> (Hashtbl.fold (fun n _ s -> let (d, i) = n in s^"Local("^(string_of_int
d)^","^(string_of_int i)^"),") t "[")^"]"

    and prefix = function
      | 0 -> ""
      | level -> (String.make (level * 3) ' ') ^ "+--"
    and location_name = function
      | GlobalVar(uid, ind) ->"Global("^(string_of_int uid)^","^(string_of_int ind)^")\n"
      | LocalVar(uid, d, ind) ->"Local("^(string_of_int uid)^","^(string_of_int
d)^","^(string_of_int ind)^")\n"

    and opname = function
      | Plus -> "+"
      | Minus -> "-"
      | Divide -> "/"
      | Times -> "*"
      | Modulo -> "%"
      | And -> "&&"
      | Or -> "||"

    and compopname = function
      | LessThan -> "<"
      | LessThanEqual -> ">"
      | Equal -> "=="
      | GreaterThan -> ">"
      | GreaterThanEqual -> ">="
      | NotEqual -> "!="

    and expr_descriptionl level = function
      | RBinaryOp (op1, op, op2) ->
          (prefix level) ^
          ("BinOp " ^
            ((opname op) ^
              ("\n" ^
                ((expr_descriptionl (level + 1) op1) ^
                  (expr_descriptionl (level + 1) op2)))))
      | RCompOp (op1, op, op2) ->
          (prefix level) ^
          ("CompOp " ^
            ((compopname op) ^
              ("\n" ^
                ((expr_descriptionl (level + 1) op1) ^
                  (expr_descriptionl (level + 1) op2)))))
      | RValue value ->
          (prefix level) ^ "Value reference\n" ^ value_descriptionl (level + 1) value
      | RMapExpr v ->
          (prefix level) ^ ("Map\n" ^ ((property_list (level + 1) v) ^ "\n"))
      | RArrayExpr v ->
          (prefix level) ^ ("Array\n" ^ (expression_list (level + 1) v))
      | RVariable(loc) ->
          (prefix level) ^"Variable " ^(location_name loc)
      | RVarArg(loc) ->
          (prefix level) ^ "VarArg" ^(location_name loc)
      | RNot(expr) ->
          (prefix level) ^ "Not\n"^(expr_descriptionl (level + 1) expr)
      | RDeclaration(expr1, expr2) ->
          (prefix level)^"Declare\n"^
          (expr_descriptionl (level + 1) expr1)^(expr_descriptionl (level + 1) expr2)
      | RAssignment(expr1, expr2) ->
          (prefix level)^"Assignment\n"^
          (expr_descriptionl (level + 1) expr1)^(expr_descriptionl (level + 1) expr2)
      | RPostFixSum(expr, inc) -> (prefix level)^"++("^(string_of_int inc)^")\n"^
          (expr_descriptionl (level + 1) expr)
      | RMemberExpr(expr1, expr2) ->
          (prefix level)^"Member\n"^
          (expr_descriptionl (level + 1) expr1)^(expr_descriptionl (level + 1) expr2)
        | RTernaryCond(expr1, expr2, expr3) ->
          (prefix level)^"Ternary condition\n"^
          (expr_descriptionl (level + 1) expr1)^(expr_descriptionl (level + 1) expr2)^
          (expr_descriptionl (level + 1) expr3)
        | RFunctionCall(expr, exprl) ->
          (prefix level)^"FunctionCall\n"^(expr_descriptionl (level + 1) expr)^
          (expression_list (level + 1) exprl)

    and value_descriptionl level = function
        | RIntegerValue v ->
          (prefix level) ^ ("Integer " ^ ((string_of_int v) ^ "\n"))
        | RFunctionValue (stacksize, depth, numargs, varargs, stmts, closure_vars, inline) ->
          (prefix level) ^
          "RFunction("^(string_of_int stacksize)^","^(string_of_int depth)^","^(string_of_int
numargs)
          ^","^(string_of_bool varargs)^","^(get_closure_vars closure_vars)^
          ", inline "^(match inline with | Some _ -> "true" | None -> "false")^
          ")\n"
          ^ (statement_list (level + 1) stmts)
        | RLibraryFunction(_) -> "" (* never in ast *)
        | RBooleanValue v ->
          (prefix level) ^ ("Boolean " ^ ((string_of_bool v) ^ "\n"))
        | RStringValue v -> (prefix level) ^ ("String " ^ (v ^ "\n"))
        | RFloatValue v ->
          (prefix level) ^ ("Float " ^ ((string_of_float v) ^ "\n"))
        | RMapValue( _, _) -> "" (* Not in AST *)
        | RVoid -> (prefix level) ^"void\n"
        | RUndefined -> (prefix level) ^"Undefined\n"

    and statement_list level stmt_list =
      List.fold_left (fun acc el -> acc ^ (statement_descriptionl level el))
        "" stmt_list

    and expression_list level expr_list =
      List.fold_left (fun acc el -> acc ^ (expr_descriptionl level el)) ""
        expr_list

    and property_list level prop_list =
      List.fold_left
        (fun acc el ->
              let (name, expr) = el
              in
              acc ^
              ((prefix level) ^
                ("Property " ^
                  (name ^ ("\n" ^ (expr_descriptionl (level + 1) expr)))))
        "" prop_list

    in match statement with
    | RStatementBlock(stmts) ->
        List.fold_left(fun pre stmt -> pre^(statement_descriptionl (level + 1) stmt))
          ((prefix level) ^"Statement block\n") stmts
    | RProgram(stmts) ->
        List.fold_left(fun pre stmt -> pre^(statement_descriptionl (level + 1) stmt))
          ((prefix level) ^"Program\n") stmts
    | RTryFinally(stmt1, stmt2, env) ->
        (prefix level)^"Try\n"^(statement_descriptionl (level + 1) stmt1)^
        (prefix level)^"finally\n"^(statement_descriptionl (level + 1) stmt2)
    | RTryCatch(stmt1, vloc, stmt2, env) ->
        (prefix level)^"Try\n"^(statement_descriptionl (level + 1) stmt1)^
        (prefix level)^"catch "^(location_name vloc)^(statement_descriptionl (level + 1) stmt2)
    | RSwitch(expr, stmts, env) ->
        List.fold_left(fun pre stmt -> pre^(statement_descriptionl (level + 1) stmt))
          ((prefix level)^"Switch\n"^(expr_descriptionl (level + 1) expr)) stmts
    | RForEach(vloc, expr, stmt, env) ->
        (prefix level)^"RForEach "^(location_name vloc)^"\n"^(expr_descriptionl (level + 1)
expr)^
        (statement_descriptionl (level + 1) stmt)
    | RIf (expr, iflist, elselist, env) ->
        (prefix level) ^
        ("If/Else\n" ^
          ((expr_descriptionl (level + 1) expr) ^
            ((statement_descriptionl (level + 1) iflist) ^
              (statement_descriptionl (level + 1) elselist))))
    | RReturn( expr, env) ->
        (prefix level) ^
        ("Return\n" ^ (expr_descriptionl (level + 1) expr))
    | RExpressionStatement (expr , env) -> expr_descriptionl level expr
    | RContinue(env) -> (prefix level) ^ "Continue\n"
    | RBreak(env) -> (prefix level) ^ "Break\n"
    | RFor (expr1, expr2, expr3, stmt_list, env) ->
        (prefix level) ^
        ("For\n" ^
          ((expr_descriptionl (level + 1) expr1) ^
            ((expr_descriptionl (level + 1) expr2) ^
              ((expr_descriptionl (level + 1) expr3) ^
                (statement_descriptionl (level + 1) stmt_list)))))
    | RFastIterator(vloc, start, max, incr, stmt, env) ->
        (prefix level)^ ("FastIterator "^" "^(string_of_int start)^" "
          ^(string_of_int max)^" "^(string_of_int incr)^" "
          ^(location_name vloc)^(statement_descriptionl (level + 1) stmt))
    | RNoop -> (prefix level) ^ "Noop\n"
    | RThrow(expr, env) -> (prefix level)^"Throw\n"^(expr_descriptionl (level + 1) expr)
    | RCase(Some expr, env) -> (prefix level)^"Case\n"^(expr_descriptionl (level + 1) expr)
    | RCase(None, env) -> (prefix level)^"DefaultCase\n"

  in statement_descriptionl 0 statement

(**
Pretty prints the representation of the runtime AST
@param statement the top level statement (program)
@return unit
*)
let print_ast statement = print_string (statement_description statement)
```

environment.ml

```
(**
Operations on AST analysis and runtime environments.

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)

(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Ast

module StringMap = Map.Make(String)

(** Variable information, tuple of index into scope and unique id *)
type var_info = (int * int )

(**
represents variables map in a global or local scope, and reference to parent scope
*)
type rec_varmap ={
  variable_map: var_info StringMap.t; (** map of variable name to variable info *)
  parent: rec_varmap option; (** parent scope variable map, or None if top level scope *)
}

(**
Properties of variable locations
*)
type var_prop ={
  written_after_declared: bool; (** is the variable assigned after it is declared *)
  read_after_declared: bool; (** is the variable read after declared *)
  declaration_loc: string * int; (** tuple of file where variable is declared and line number*)
}

(** position of a label with within a template spec, tuple of start begin, start end,
end begin, end ending *)
type label_pos = int * int * int * int

(** definition of a template specidifcation, used during validity checking.
tuple of sepecfication list and map of labels to label position *)
type template_spec_def = (template_spec list *  (string , label_pos ) Hashtbl.t * (string *
int))

(**
The analysis environment
*)
type analysis_env ={
  globals: rec_varmap; (** map of global variables *)
  num_globals: int; (** number of globals *)
  locals: rec_varmap list; (** recursive list of stack frames *)
  num_locals: int list; (** number of locals in current stack frame *)
  sdepth: int; (** current stack depth *)
  max_depth: int; (** maximum stack depth encountered *)
  errors: string list; (** list of errors found during analysis *)
  warnings: string list; (** list of warning generated during analysis *)
  unique_id: int; (** counter for next unique id *)
  names: string list; (** list of names encountered *)
  varprops: (int, var_prop) Hashtbl.t; (** properties of variables *)
  imported: string list; (** list of files already imported *)
  templates: (string, template_spec_def) Hashtbl.t; (** map of template names to template
definitions *)
  constants: (int, runtime_variable_value) Hashtbl.t; (** map of variables unique id to declared
value *)
}
```

```
(**
returns a newly initialized analysis environment
@return analysis_env
*)
let new_analysis_environment () =
  {
    globals ={ variable_map = StringMap.empty; parent = None };
    num_globals = 0;
    locals =[];
    num_locals = [];
    sdepth = 0;
    max_depth = 0;
    errors =[];
    warnings =[];
    unique_id = 0;
    varprops = Hashtbl.create 10;
    names =[];
    imported =[];
    templates = Hashtbl.create 1;
    constants = Hashtbl.create 10;
  }

(**
sets the declaration value for a variable
@param env analysis environment
@param uid unique id of variable
@param value runtime value of variable
@return unit
*)
let set_constant_value env uid value =
  Hashtbl.replace env.constants uid value

(**
gets the constant value for a variable
@param env analysis environment
@param uid unique id of variable
@return runtime value of variable
*)
let get_constant_value env uid =
  Hashtbl.find env.constants uid

(**
returns whether the variable is a constant
@param env analysis environment
@param uid unique id of variable
@return true if the variable is a constant
*)
let is_constant env uid =
  let varprop = Hashtbl.find env.varprops uid
  in let (_, line) = varprop.declaration_loc
  in not varprop.written_after_declared && line!= 0

(**
declare a variable if it does not exist or create a new entry and return new index
@param name name of variable to declare
@param env analysis environment
@return a tuple of the modified environment and uid
*)
```

```
let declare_variable env name =
  let find_or_declare varmaps nextind uid =
    try let (_, uid) = StringMap.find name varmaps in (varmaps, 0, uid)
    with Not_found -> (StringMap.add name (nextind, uid) varmaps, 1, uid)
  in
  match env.locals with
  | [] ->
      let ( map, num_added, uid) = find_or_declare env.globals.variable_map env.num_globals
env.unique_id
      in
      ({ globals ={ variable_map = map; parent = env.globals.parent };
          num_globals = env.num_globals + num_added;
          locals = env.locals;
          num_locals = env.num_locals;
          sdepth = env.sdepth;
          max_depth = env.max_depth;
          errors = env.errors;
          warnings = env.warnings;
          unique_id = env.unique_id + num_added;
          names = if num_added = 0 then env.names else name:: env.names;
          varprops = env.varprops;
          imported = env.imported;
          templates = env.templates;
          constants = env.constants;
        }, uid)
  | _ ->
      let (map, num_added, uid) = find_or_declare (List.hd env.locals).variable_map (List.hd
env.num_locals) env.unique_id
      in
      ({ globals = env.globals;
          num_globals = env.num_globals;
          locals ={ variable_map = map; parent = (List.hd env.locals).parent }:: List.tl
env.locals;
          num_locals = ((List.hd env.num_locals) + num_added):: List.tl env.num_locals;
          sdepth = env.sdepth;
          max_depth = env.max_depth;
          errors = env.errors;
          warnings = env.warnings;
          unique_id = env.unique_id + num_added;
          names = if num_added = 0 then env.names else name:: env.names;
          varprops = env.varprops;
          imported = env.imported;
          templates = env.templates;
          constants = env.constants;
        }, uid)

(**
declare a variable if it does not exist or create a new entry and return new index,
then sets constant value
@param name name of variable to declare
@param env analysis environment
@param value the value to initialize the variable with
@return the modified environment
*)
let declare_variable_and_value name env value =
  let (env, uid) = declare_variable name env
  in set_constant_value env uid value; env

(** internal exception used during analysis *)
```

```
exception Variable_not_found of string

(**
Find variable in analysis scope
@param name the variable name
@param env the analysis environment
@return location
@raise Variable_not_found when the variable is not found
*)
let resolve_variable name env =
  let rec find scopes =
    try
      let (ind, uid) = StringMap.find name scopes.variable_map
      in (uid, ind)
    with Not_found ->
        (match scopes.parent with
          | Some parent -> find parent
          | None -> raise Not_found)
  in
  let rec find_in_stackframes = function
    | [] -> raise Not_found
    | scope:: tl ->
        try
          let (uid, ind) = find scope
          in (LocalVar(uid, List.length tl, ind))
        with
        | Not_found -> find_in_stackframes tl
  in
  try
    match env.locals with
    | [] -> let (uid, ind) = find env.globals in (GlobalVar(uid, ind))
    | _ -> (try
          find_in_stackframes env.locals
        with Not_found ->
            let (uid, ind) = find env.globals in (GlobalVar(uid, ind)))
  with Not_found -> raise (Variable_not_found name)

(**
returns uid from location
@param loc the variable location
@return the unique id of the variable
*)
let uid_from_loc = function
  | GlobalVar(uid, _) -> uid
  | LocalVar(uid, _, _) -> uid

(**
Find variable and value in analysis scope
@param name the variable name
@param env the analysis environment
@return tuple of value and location
@raise Variable_not_found when the variable is not found
*)
let resolve_variable_value name env =
  let loc = resolve_variable name env
  in let uid = uid_from_loc loc
  in ( get_constant_value env uid, loc)

(**
Setups a new scope within the same global or local scope
@param env analysis environment
@return a new analysis environment setup for the new scope
*)
let new_analysis_scope env =
  match env.locals with
  | [] ->{
        globals = { variable_map = StringMap.empty; parent = Some env.globals };
        num_globals = env.num_globals;
        locals =[];
        num_locals = env.num_locals;
        sdepth = env.sdepth;
        max_depth = env.max_depth;
        errors = env.errors;
        warnings = env.warnings;
        unique_id = env.unique_id;
        names = env.names;
        varprops = env.varprops;
        imported = env.imported;
        templates = env.templates;
        constants = env.constants;
      }
  | hd:: tl -> {
        globals = env.globals;
        num_globals = env.num_globals;
        locals ={ variable_map = StringMap.empty; parent = Some hd }:: tl;
        num_locals = env.num_locals; (* on new scope, same number of locals *)
        sdepth = env.sdepth;
        max_depth = env.max_depth;
        errors = env.errors;
        warnings = env.warnings;
        unique_id = env.unique_id;
        names = env.names;
        varprops = env.varprops;
        imported = env.imported;
        templates = env.templates;
        constants = env.constants;
      }
(**
Pops the analysis scope
@param env analysis environment
@param old old analysis environment
@return a new environment with the last scope popped
*)
let pop_scope env =
  match env.locals with
  | [] ->
      (match env.globals.parent with
        | Some old_globals -> {
              globals = old_globals;
              num_globals = env.num_globals;
              locals = env.locals;
              num_locals = env.num_locals;
              sdepth = env.sdepth;
              max_depth = env.max_depth;
              errors = env.errors;
              warnings = env.warnings;
              unique_id = env.unique_id;
              names = env.names;
              varprops = env.varprops;
              imported = env.imported;
              templates = env.templates;
              constants = env.constants;
            }
        | None -> raise (RuntimeError.InternalError "popping a top level scope"))
  | local:: tl ->
      match local.parent with
      | Some old_parent -> {
            globals = env.globals;
            num_globals = env.num_globals;
            locals = old_parent:: tl;
            num_locals = env.num_locals; (* preserve, we are still in the same stack frame *)
            sdepth = env.sdepth;
            max_depth = env.max_depth;
            errors = env.errors;
            warnings = env.warnings;
            unique_id = env.unique_id;
            names = env.names;
            varprops = env.varprops;
            imported = env.imported;
            templates = env.templates;
            constants = env.constants;
          }
      | None -> {
            globals = env.globals;
            num_globals = env.num_globals;
            locals = tl;
            num_locals = List.tl env.num_locals; (* exiting stack frame, restore old number of
locals *)
            sdepth = env.sdepth - 1;
            max_depth = env.max_depth;
            errors = env.errors;
            warnings = env.warnings;
            unique_id = env.unique_id;
            names = env.names;
            varprops = env.varprops;
            imported = env.imported;
            templates = env.templates;
            constants = env.constants;
          }

(**
Create a new stackframe
@param env analysis environment
@return a new analysis environment with a new stackframe
*)
let new_analysis_stackframe env =
  {
    globals = env.globals;
    num_globals = env.num_globals;
    locals ={ variable_map = StringMap.empty; parent = None }:: env.locals;
    num_locals = 0:: env.num_locals; (* push new stackframe number of locals *)
    sdepth = env.sdepth + 1;
    max_depth = if env.sdepth + 1 > env.max_depth then env.sdepth + 1 else env.max_depth;
    errors = env.errors;
    warnings = env.warnings;
    unique_id = env.unique_id;
    names = env.names;
    varprops = env.varprops;
    imported = env.imported;
    templates = env.templates;
    constants = env.constants;
  }

(**
Returns the depth of the current stack frame
@param env analysis environment
@return the depth of the current stack frame, 0 indexed
*)
let get_depth env =
  (List.length env.locals) - 1

(**
Add an error to the analysis environemnt
@param env the analysis environment
@param codeloc a filename, line number tuple
@param message the error message
@return an analysis environment with the error added
*)
let add_error env codeloc message =
  let (filename, line_number) = codeloc
  in  {
    globals = env.globals;
    num_globals = env.num_globals;
    locals = env.locals;
    num_locals = env.num_locals;
    errors = ("At line "^(string_of_int line_number)^" in "^(Filename.basename filename)^":
"^message):: env.errors;
    sdepth = env.sdepth;
    max_depth = env.max_depth;
    warnings = env.warnings;
    unique_id = env.unique_id;
    names = env.names;
    varprops = env.varprops;
    imported = env.imported;
    templates = env.templates;
    constants = env.constants;
  }

(**
Add a warning to the analysis environemnt
@param env the analysis environment
@param codeloc a filename, line number tuple
@param message the warning message
@return an analysis environment with the warning added
*)
let add_warning env codeloc message =
  let (filename, line_number) = codeloc
  in  {
    globals = env.globals;
    num_globals = env.num_globals;
    locals = env.locals;
    num_locals = env.num_locals;
    sdepth = env.sdepth;
    max_depth = env.max_depth;
    errors = env.errors;
    warnings = ("At line "^(string_of_int line_number)^" in "^(Filename.basename filename)^":
"^message):: env.warnings;
    unique_id = env.unique_id;
    names = env.names;
    varprops = env.varprops;
    imported = env.imported;
    templates = env.templates;
    constants = env.constants;
  }

(**
Returns true if there are errors in the environment
@param env the analysis environment
@return true if there are errors, false otherwise
*)
let has_errors env =
  env.errors!=[]

(**
adds an import to the list of imports
@param env analysis environment
@param filename the filename
@return the modified environment
*)
let add_import env filename =
  {
    globals = env.globals;
    num_globals = env.num_globals;
    locals = env.locals;
    num_locals = env.num_locals;
    sdepth = env.sdepth;
    max_depth = env.max_depth;
    errors = env.errors;
    warnings = env.warnings;
    unique_id = env.unique_id;
    names = env.names;
    varprops = env.varprops;
    imported = filename:: env.imported;
    templates = env.templates;
    constants = env.constants;
  }

(**
type of operation performed on variable
*)
type var_op_type =
  | ReadOp  (** variable is read *)
  | WriteOp (** variable is written *)
  | DeclareOp of (string * int) (** variable is declared *)
  | DeclareWriteOp of (string * int) (** variable is declared and written, used for function
args *)

(**
Records a variables property
@param env analysis environment
@param loc variable location
@param operation the operation on the variable
@return unit
*)
```

```
let record_usage env loc op =
  let uid = match loc with
    | GlobalVar(uid, _) -> uid
    | LocalVar(uid, _, _) -> uid
  in let props =
    try Hashtbl.find env.varprops uid
    with Not_found ->
        { written_after_declared = false;
          read_after_declared = false;
          declaration_loc = ("", 0);
        }
  in let new_props =
    match op with
    | ReadOp ->
        { written_after_declared = props.written_after_declared;
          read_after_declared = true;
          declaration_loc = props.declaration_loc;
        }
    | WriteOp ->
        { written_after_declared = true;
          read_after_declared = props.read_after_declared;
          declaration_loc = props.declaration_loc;
        }
    | DeclareOp(loc) ->
        (match props.declaration_loc with
          | ("", 0) ->
              { written_after_declared = false;
                read_after_declared = props.read_after_declared;
                declaration_loc = loc
              }
          | _ ->
              { written_after_declared = true;
                read_after_declared = props.read_after_declared;
                declaration_loc = props.declaration_loc
              })
    | DeclareWriteOp(loc) ->
        match props.declaration_loc with
        | ("", 0) ->
            { written_after_declared = true;
              read_after_declared = true;
              declaration_loc = loc
            }
        | _ ->
            { written_after_declared = true;
              read_after_declared = true;
              declaration_loc = props.declaration_loc
            }
  in Hashtbl.replace env.varprops uid new_props

(**
Adds a template to the environment
@param runtime environment
@param name template name
@param spec_list list of line specifications
@param labels label positions
@return a new environment
**)
let add_template env name spec_list labels cloc =
  let env = if Hashtbl.mem env.templates name then
```

```
      add_warning env cloc ("Duplicate template definition '" ^ name ^ "'")
    else
      env
  in Hashtbl.replace env.templates name (spec_list, labels, cloc); env

(**
checks if a file has already been imported
@param env analysis environment
@param filename the filename to check
@return true if already imported, false otherwise
*)
let has_import env filename =
  let rec loop = function
    | [] -> false
    | s:: tl -> if s = filename then true else loop tl
  in loop env.imported

(**
Retrieves a value at a location
@param env a runtime environment
@param loc the location of the variable
@return the value at the selected location
*)
let get_value env = function
  | GlobalVar(uid, ind) -> let (_, value) = env.heap.(ind) in value
  | LocalVar(uid, depth, ind) ->
      match env.closure_vars with
      | None -> env.stackframes.(depth).(ind)
      | Some h ->
          try match Hashtbl.find h (depth, ind) with
            | RUndefined -> env.stackframes.(depth).(ind) (* needed for recursive function defs
*)
            | value -> value
          with Not_found -> env.stackframes.(depth).(ind)

(**
Sets a value at a location
@param env a runtime environment
@param value the value to set
@param loc the location of the variable
@return the value that was set
*)
let set_value env value = function
  | GlobalVar(uid, ind) -> env.heap.(ind) <- (uid, value); value
  | LocalVar(uid, depth, ind) -> env.stackframes.(depth).(ind) <- value; value

(**
Returns the name of a location
@param env the runtime environment
@param loc the location of the variable
@return the name of the variable at location loc
*)
let get_loc_name env = function
  | GlobalVar(uid, _) | LocalVar(uid, _, _) -> env.gnames.(uid)

(**
Evaluation of binary operations and comparaison of values
Various helper functions for expression evaluation
@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

*)

open Ast
open RuntimeError

(**
Converts a MapValue array to a list of values
@param arr array
@return a list of values
*)
let list_of_array arr =
  match arr with
  | RMapValue(h, ArraySubtype) -> (
        match Hashtbl.find h "length" with
        | RIntegerValue(0) ->[]
        | RIntegerValue(len) ->
            let rec loop lst ind =
              (let lst = (Hashtbl.find h (string_of_int ind)):: lst in
                if ind = 0 then lst else (loop lst (ind - 1)))
            in loop [] (len - 1)
        | _ -> raise (RuntimeError.InternalError "inconsistent array/length not found")
      )
  | _ -> raise (RuntimeError.InternalError "inconsistent array/not an array")

(**
Converts a value to a string
@param value the value to convert
@return a string representing the value
*)
let rec string_of_value = function
  | RStringValue(s) -> s
  | RIntegerValue(i) -> string_of_int i
  | RFloatValue(f) -> string_of_float f
  | RBooleanValue(b) -> string_of_bool b
  | RMapValue(t, ArraySubtype) as v ->
      let lst = list_of_array v
      in let rec loop s = function
        | [] -> s^"]"
        | v::[] -> loop (s^(string_of_value v)) []
        | v:: tl -> loop (s^(string_of_value v)^", ") tl
      in loop "[" lst
  | RMapValue(t, MapSubtype) ->
      (Hashtbl.fold (fun prop v s ->
            s^prop^": "^(string_of_value v)^";") t "{")^"}"
  | RFunctionValue(_, _, _, _, _, _, _) | RLibraryFunction(_) -> "function"
  | RVoid -> "void"
  | RUndefined -> "undefined"
```

```
(**
enumeration of a value's possible types
*)
type valuetype =
  | IntegerType
  | FloatType
  | BooleanType
  | StringType
  | FunctionType
  | LibraryCallType
  | MapType
  | ArrayType
  | VoidType
  | NaNType
  | UndefinedType

(**
Returns a value's type
@param a value
@return the value's type
*)
let value_type = function
  | RIntegerValue(_) -> IntegerType
  | RFloatValue(_) -> FloatType
  | RBooleanValue(_) -> BooleanType
  | RStringValue(_) -> StringType
  | RFunctionValue(_, _, _, _, _, _, _) -> FunctionType
  | RLibraryFunction(_) -> LibraryCallType
  | RMapValue(_, MapSubtype) -> MapType
  | RMapValue(_, ArraySubtype _) -> ArrayType
  | RVoid -> VoidType
  | RUndefined -> UndefinedType

(**
returns a string name for a value's type
@param value a value
@return string name for the value's type
*)
let string_of_value_type = function
  | RStringValue(s) -> "string"
  | RIntegerValue(i) -> "integer"
  | RFloatValue(f) -> "float"
  | RBooleanValue(b) -> "boolean"
  | RMapValue(_, ArraySubtype) ->"array"
  | RMapValue(_, MapSubtype) -> "map"
  | RFunctionValue(_, _, _, _, _, _, _) | RLibraryFunction(_) -> "function"
  | RVoid -> "void"
  | RUndefined -> "undefined"

(** type to hold the result of casting two values to the same type *)
type cast_type =
  | IntegerCast of int * int
  | FloatCast of float * float
  | StringCast of string * string
  | BoolCast of bool * bool

(** cast a value to an integer
@param value the runtime value
@return an integer representation of the value
@raise EInvalidCast if the value cannot be cast
*)
let cast_to_integer value =
  match value with
  | RIntegerValue(i) -> i
  | _ -> raise (EInvalidCast (string_of_value_type value,"integer"))

(** cast a value to a float
@param value the runtime value
@return an float representation of the value
@raise EInvalidCast if the value cannot be cast
*)
let cast_to_float value =
  match value with
  | RFloatValue(f) -> f
  | RIntegerValue(i) -> float_of_int i
  | _ -> raise (EInvalidCast (string_of_value_type value,"float"))

(**
Evaluate the operation
@param value1 the first value
@param value2 the second value
@param operator the operator
@return the value that results from the operation
*)
let evaluate_op value1 value2 operator =
  let string_of_operator = function
    | Plus -> "+"
    | Minus -> "-"
    | Times -> "*"
    | Divide -> "/"
    | Modulo -> "%"
    | Or -> "||"
    | And -> "&&"
  in let string_op s1 s2 =
    (match operator with
      | Plus -> RStringValue(s1 ^ s2)
      | _ -> raise (EInvalidOperation (string_of_operator operator,"string"))
    )
  in let float_op f1 f2 = (let f = (match operator with
          | Plus -> f1 +. f2
          | Minus -> f1 -. f2
          | Times -> f1 *. f2
          | Divide -> f1 /. f2
          | _ -> raise (EInvalidOperation (string_of_operator operator,"float"))) in
       RFloatValue(f)
    )
  in match (value1, value2) with
  | (RIntegerValue(i1), RIntegerValue(i2)) ->
      (match operator with
        | Plus -> RIntegerValue( i1 + i2 )
        | Minus -> RIntegerValue( i1 - i2)
        | Times -> RIntegerValue( i1 * i2)
        | Divide -> RIntegerValue( i1 / i2)
        | Modulo -> RIntegerValue( i1 mod i2)
        | _ -> raise (EInvalidOperation (string_of_operator operator,"integer"))
      )
  | (RBooleanValue(b1), RBooleanValue(b2)) ->
```

```
      (match operator with
        | And -> RBooleanValue(b1 && b2)
        | Or -> RBooleanValue(b1 || b2)
        | _ -> raise (EInvalidOperation (string_of_operator operator,"boolean"))
      )
  | (RFloatValue(f1), RFloatValue(f2)) -> float_op f1 f2
  | (RFloatValue(f1), RIntegerValue(i2)) -> float_op f1 (float_of_int i2)
  | (RIntegerValue(i1), RFloatValue(f2)) -> float_op (float_of_int i1) f2
  | (RStringValue(s1), RStringValue(s2)) -> string_op s1 s2
  | (RStringValue(s1), v2) -> string_op s1 (string_of_value v2)
  | (v1, RStringValue(s2)) -> string_op (string_of_value v1) s2
  | (value1, value2) -> raise (EIncompatibleTypes(string_of_value_type value1,
string_of_value_type value2))

(**
Implements comparaison of two values, according to the following semantics:

-Integer Integer Any Comparison of integer values
-Float Float Any Comparison of float values
-Float Integer Any Comparison of float values
-String any type Float comparison of first value to second value,
-Integer with non string values converted to strings
-Both types are Booleans,== and != comparison of first value to second value
-maps, arrays, functions,== and != comparison of first value to second value
-NaN or void == and != comparison of first value to second value
-Different types == always returns false != always returns true

@param v1 the first value to compare
@param op the comparaison operator
@param v2 the second value to compare
@return a boolean value type
*)
let rec compare v1 op v2 =
  match v1 with
  | RIntegerValue(i1) ->
      (match v2 with
        | RIntegerValue(i2) ->
            (match op with
              | Equal -> RBooleanValue(i1 = i2)
              | NotEqual -> RBooleanValue(i1 <> i2)
              | LessThan -> RBooleanValue(i1 < i2)
              | LessThanEqual -> RBooleanValue(i1 <= i2)
              | GreaterThan -> RBooleanValue(i1 > i2)
              | GreaterThanEqual -> RBooleanValue(i1 >= i2) )
        | RFloatValue(f2) -> compare (RFloatValue (float_of_int i1)) op v2
        | RStringValue(s2) -> compare (RStringValue (string_of_int i1)) op v2
        | _ -> mismatched_compare v1 op v2 )
  | RStringValue(s1) ->
      (match v2 with
        | RStringValue(s2) ->
            (match op with
              | Equal -> RBooleanValue(s1 = s2)
              | NotEqual -> RBooleanValue(s1 <> s2)
              | LessThan -> RBooleanValue(s1 < s2)
              | LessThanEqual -> RBooleanValue(s1 <= s2)
              | GreaterThan -> RBooleanValue(s1 > s2)
              | GreaterThanEqual -> RBooleanValue(s1 >= s2) )
        | RIntegerValue(i2) -> compare v1 op (RStringValue(string_of_int i2))
        | RFloatValue(f2) -> compare v1 op (RStringValue(string_of_float f2))
        | _ -> mismatched_compare v1 op v2 )
  | RBooleanValue(b1) ->
      (match v2 with
        | RBooleanValue(b2) -> (
              match op with
              | Equal -> RBooleanValue(b1 = b2)
              | NotEqual -> RBooleanValue(b1 <> b2)
              | _ -> raise (EInvalidComparaison(opname op, string_of_value_type v1,
string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RVoid ->
      (match v2 with
        | RVoid -> (
              match op with
              | Equal -> RBooleanValue(true)
              | NotEqual -> RBooleanValue(false)
              | _ -> raise (EInvalidComparaison(opname op, string_of_value_type v1,
string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RFloatValue(f1) ->
      (match v2 with
        | RFloatValue(f2) ->
            (match op with
              | Equal -> RBooleanValue(f1 = f2)
              | NotEqual -> RBooleanValue(f1 <> f2)
              | LessThan -> RBooleanValue(f1 < f2)
              | LessThanEqual -> RBooleanValue(f1 <= f2)
              | GreaterThan -> RBooleanValue(f1 > f2)
              | GreaterThanEqual -> RBooleanValue(f1 >= f2) )
        | RIntegerValue(i2) -> compare v1 op (RFloatValue (float_of_int i2))
        | RStringValue(s2) -> compare (RStringValue(string_of_float f1)) op v2
        | _ -> mismatched_compare v1 op v2 )
  | RMapValue(h1, ArraySubtype) ->
      (match v2 with
        | RMapValue(h2, ArraySubtype) -> (
              match op with
              | Equal -> RBooleanValue(hashtbl_equal h1 h2)
              | NotEqual -> RBooleanValue(not (hashtbl_equal h1 h2))
              | _ -> raise (EInvalidComparaison(opname op,
                        string_of_value_type v1,
                        string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RMapValue(h1, MapSubtype) ->
      (match v2 with
        | RMapValue(h2, MapSubtype) -> (
              match op with
              | Equal -> RBooleanValue(hashtbl_equal h1 h2)
              | NotEqual -> RBooleanValue(not (hashtbl_equal h1 h2))
              | _ -> raise (EInvalidComparaison(opname op,
                        string_of_value_type v1,
                        string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RFunctionValue(size1, depth1, len1, varargs1, stmts1, clos1, inline1) ->
      (match v2 with
        | RFunctionValue(size2, depth2, len2, varargs2, stmts2, clos2, inline2) -> (
              match op with
              | Equal -> RBooleanValue(size1 = size2 && stmts1 = stmts2)
              | NotEqual -> RBooleanValue(not (size1 = size2 && stmts1 = stmts2))
              | _ -> raise (EInvalidComparaison(opname op,
                        string_of_value_type v1,
                        string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RLibraryFunction(def1) ->
      (match v2 with
        | RLibraryFunction(def2) ->
            ( match op with
              | Equal -> RBooleanValue(def1 == def2)
              | NotEqual -> RBooleanValue(not (def1 == def2))
              | _ -> raise (EInvalidComparaison(opname op,
                        string_of_value_type v1,
                        string_of_value_type v2)) )
        | _ -> mismatched_compare v1 op v2 )
  | RUndefined -> raise (RuntimeError.InternalError "unexpected value in compare")
and opname = function
  | LessThan -> "<"
  | LessThanEqual -> "<="
  | Equal -> "=="
  | NotEqual ->"!="
  | GreaterThanEqual -> ">="
  | GreaterThan -> ">"
and hashtbl_equal h1 h2 =
  (Hashtbl.length h1) = (Hashtbl.length h2) &&
  try
    Hashtbl.fold (fun k v init -> init && (compare (Hashtbl.find h2 k) Equal v) =
RBooleanValue(true) ) h1 true
  with
  | Not_found -> false
and mismatched_compare v1 op v2 =
  match op with
  | Equal -> RBooleanValue(false)
  | NotEqual -> RBooleanValue(true)
  | _ -> raise (EInvalidComparaison(opname op, string_of_value_type v1, string_of_value_type
v2))
(**
Makes a stack frame from the supplied value list
@param size size of stack frame
@param vararg true if the last argument is a vararg, false otherwise
@param value_list list of values to add to the stack frame
@param this the value of this
@return a stack frame (an array of values)
*)
and make_stackframe size numargs vararg value_list this =
  let stackframe = Array.make (size + 1) RUndefined
  in let rec loop_single_values = function
    | (0, _, rest) -> rest
    | (num_left, ind, value:: rest) ->
        stackframe.(ind) <- value;
        loop_single_values (num_left - 1, ind + 1, rest)
    | (num_left, ind,[]) -> raise (EMismatchedFunctionArgs (numargs, List.length value_list))
  in let rest = loop_single_values ((if vararg then numargs - 1 else numargs), 1, value_list)
  in ((match (rest, vararg) with
      | (list, true) -> stackframe.(numargs) <- array_of_value_list(list)
      | ([] , false) -> ()
      | (_, false) ->
          raise (EMismatchedFunctionArgs (numargs, List.length value_list))));
  stackframe.(0) <- this;
  stackframe
(**
Creates an Array from a list of values
@param value_list a list of values
@return a MapValue with the array
*)
and array_of_value_list value_list =
  let rec loop = function
    | (_,[], h) -> h
    | (ind, value:: rest, h) ->
        Hashtbl.replace h (string_of_int ind) value;
        loop (ind + 1, rest, h)
  in let length = List.length value_list
  in let h = Hashtbl.create (length + 1)
  in Hashtbl.replace h "length" (RIntegerValue(length));
  RMapValue(loop (0, value_list, h), ArraySubtype)
```

filename_util.ml

```
(**
Filename utilities.
@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)

(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*)

(**
converts a relative filename and path into an absolute filename
@param dir relative of absolute path
@param filename the filename to process
@return absolute path of file
*)
let resolve_filename dir filename =
  let rec cleanup check ok =
    let right = Filename.basename check in
    let left = Filename.dirname check in
    if (right ="." && Filename.dirname left = left) then
      Filename.concat left ok
    else
      match right with
      | "." -> cleanup left ok
      | ".." -> cleanup (Filename.dirname left) ok
      | "" -> ok
      | _ -> cleanup left (if ok ="" then right else Filename.concat right ok)
  in
  if Filename.is_relative filename then
    cleanup (Filename.concat dir filename) ""
  else
    filename
```

interpreter.ml

```
(**
The Jtemplate interpreter

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*)

open Ast
open Expression
open RuntimeError

(**
Interpret a runtime AST
@param env a runtime environment
@param ast the runtime AST to interpret
@return unit
*)
let rec interpret env = function
  | RProgram(stmts) | RStatementBlock(stmts) ->
      List.iter (fun stmt -> interpret env stmt) stmts
  | RExpressionStatement(expr, cloc) ->
      env.current_line <- cloc;
      let _ = evaluate env expr in ()
  | RReturn (expr, cloc) ->
      env.current_line <- cloc;
      let (_, value) = evaluate env expr
      in raise (CFReturn value)
  | RFor (preloop, condexpr, postloop, stmt, cloc) ->
      env.current_line <- cloc;
      let _ = evaluate env preloop
      in let rec loop () =
        let (env, value) = evaluate env condexpr
        in match value with
        | RBooleanValue(false) -> ()
        | RBooleanValue(true) | RVoid ->
            (try
              interpret env stmt
            with
            | CFContinue -> ()
            );
            let _ = evaluate env postloop
            in loop ()
        | value ->
            raise (EInvalidCast(string_of_value value,"boolean"))
      in (try
        loop ()
      with
      | CFBreak -> ())
  | RFastIterator (vloc, start, max, inc, stmt, cloc) ->
      env.current_line <- cloc;
      let rec loopfast ind =
        let _ = Environment.set_value env (RIntegerValue(ind)) vloc
        in if ind < max then
          ((try
              interpret env stmt
            with
            | CFContinue -> ()
            );
            loopfast (ind + inc))
        else
          ()
      in (try
        loopfast start
      with
      | CFBreak -> ())
  | RForEach (vloc, expr, stmt, cloc) ->
      env.current_line <- cloc;
      let list =
        let (env, value) = evaluate env expr
        in match value with
        | RMapValue(h, MapSubtype) -> Hashtbl.fold (fun k v lst -> RStringValue(k):: lst) h []
        | RMapValue(_, ArraySubtype) as v -> list_of_array v
        | _ as v -> raise(ENotACollectionType("the second argument of forEach",
string_of_value_type v))
      in let rec loop = function
        | [] -> ()
        | hd:: tl ->
            let _ = Environment.set_value env hd vloc
            in (try interpret env stmt with CFContinue -> ());
            loop tl
      in
      (try loop list with CFBreak -> () )
  | RIf(condexpr, if_stmt, else_stmt, cloc) ->
      env.current_line <- cloc;
      let (env, value) = evaluate env condexpr
      in (match value with
        | RBooleanValue(true) -> interpret env if_stmt
        | RBooleanValue(false) -> interpret env else_stmt
        | value -> raise (EInvalidCast(string_of_value value,"boolean"))
      )
  | RSwitch(expr, stmtlist, cloc) ->
      env.current_line <- cloc;
      let rec find_cases caselist defaultfound = function
        | []-> caselist
        | RCase(Some expr, _):: tl ->
            if defaultfound then
              raise EDefaultCaseShouldBeLast
            else(
              let caselist = (Some expr, tl):: caselist in
              find_cases caselist false tl)
        | RCase(None, _):: tl ->
            if defaultfound then
              raise EDefaultCaseShouldBeLast
            else(
              let caselist = (None, tl):: caselist in
              find_cases caselist true tl)
        | _:: tl -> find_cases caselist defaultfound tl
      (* match a value with a case and return a statement list *)
      in let rec match_case expr1 = function
        | [] -> []
        | (Some expr2, stmts):: tl ->
            let (env, value) = evaluate env (RCompOp(expr1, Equal, expr2))
            in if value = RBooleanValue(true) then
              stmts
            else
              match_case expr1 tl
        | (None, stmts):: tl -> stmts
      in let caselist = List.rev (find_cases [] false stmtlist)
      in let (env, value) = evaluate env expr
      in let stmts = match_case (RValue(value)) caselist
      in (try
        List.iter (fun stmt -> interpret env stmt) stmts
      with
      | CFBreak -> ())
  | RTryCatch(stmt1, vloc, stmt2, cloc) ->
      env.current_line <- cloc;
      (try
        interpret env stmt1;
      with
      | CFUserException(e, _) ->
          let _ = Environment.set_value env e vloc
          in interpret env stmt2
      | CFBreak | CFContinue as exn -> raise exn
      | CFReturn(v) -> raise (CFReturn v)
      | exn ->
          let _ = Environment.set_value env (RStringValue(RuntimeError.string_of_error exn))
vloc
          in interpret env stmt2
      )
  | RTryFinally(stmt1, stmt2, cloc) ->
      env.current_line <- cloc;
      (try
        interpret env stmt1
      with
      | _ as e -> interpret env stmt2; raise e)
  | RContinue(cloc) ->
      env.current_line <- cloc;
      raise CFContinue
  | RBreak(cloc) ->
      env.current_line <- cloc;
      raise CFBreak;
  | RThrow(expr, cloc) ->
      env.current_line <- cloc;
      let (_, value) = evaluate env expr
      in raise (CFUserException(value, string_of_value value))
  | RNoop | RCase(_, _) -> ()

(**
Interprets a list of statements
@param env runtime environments
@param stmts list of statements
*)
and interpret_stmts env = function
```

```
  | [] -> ()
  | stmt:: tl ->
      interpret env stmt; interpret_stmts env tl
(**
Evaluates an expression
@param env runtime environment
@param expr expression to be evaluated
@return a value
*)
and evaluate env = function
  | RVariable(loc) -> (env, Environment.get_value env loc)
  | RValue(v) ->
      (match v with (* look for closure vars*)
        | RFunctionValue(framesize, depth, argslen, has_varargs, statements, Some closure_vars,
_) ->
            let closure_vals = Hashtbl.create 10
            in let _ = Hashtbl.fold(
                  fun k _ _ -> let (d, i) = k
                      in let (_, value) = evaluate env (RVariable(LocalVar(0, d, i)))
                      in Hashtbl.replace closure_vals (d, i) value) closure_vars ()
            in (env, RFunctionValue(framesize, depth, argslen, has_varargs, statements, Some
closure_vals, None))
        | _ -> (env, v))
  | RPostFixSum(expr, inc) ->
      let (env, v) = evaluate env expr
      in let _ = evaluate env (RAssignment(expr, RBinaryOp(RValue(v), Plus,
RValue(RIntegerValue(inc)))))
      in (env, v)
  | RArrayExpr(expr_list) ->
      let value_list = List.map (fun e -> let (_, v) = evaluate env e in v) expr_list
      in let len = (List.length value_list)
      in let t = Hashtbl.create len
      in let _ = List.fold_left (fun ind v -> Hashtbl.add t (string_of_int ind) v; ind + 1) 0
value_list
      in let _ = Hashtbl.add t "length" (RIntegerValue(len))
      in (env, RMapValue(t, ArraySubtype))
  | RMapExpr(prop_list) ->
      let t = Hashtbl.create (List.length prop_list)
      in let _ = List.iter(fun prop -> let (name, e) = prop in Hashtbl.add t name (let (_, v) =
evaluate env e in v)) prop_list
      in (env, RMapValue(t, MapSubtype))
  | RBinaryOp(e1, op, e2) ->
      let (env, v1) = evaluate env e1
      in let (env, v2) = evaluate env e2
      in (env, evaluate_op v1 v2 op)
  | RCompOp(e1, op, e2) ->
      let (env, v1) = evaluate env e1
      in let (env, v2) = evaluate env e2
      in (env, compare v1 op v2)
  | RTernaryCond(e1, e2, e3) ->
      let (env, value) = evaluate env e1
      in (match value with
        | RBooleanValue(true) -> evaluate env e2
        | RBooleanValue(false) -> evaluate env e3
        | v -> raise (EIncompatibleTypes ("boolean" , string_of_value_type v)))
  | RMemberExpr(left, index) ->
      let (env, left_map) = evaluate env left
      in let (key, is_int) = evaluate_memb_expr_index env index
      in (match left_map with
        | RMapValue(h, ArraySubtype) ->
            if (not is_int) then
              raise (EInvalidArrayIndex("string", key))
            else
              (try
                (env, Hashtbl.find h key)
              with Not_found ->
                  raise (EArrayIndexOutOfBounds key))
        | RMapValue(h, MapSubtype) ->
            (try
              (env, Hashtbl.find h key)
            with Not_found ->
                raise (EUndefinedMapMember key))
        | _ -> raise (ELeftSideIsNotAMap(string_of_value_type left_map, string_of_value
left_map)))
  | RNot(expr) ->
      let (env, v) = evaluate env expr in
      (match v with
        | RBooleanValue(b) -> (env, RBooleanValue(not b))
        | _ -> raise (EIncompatibleTypes(string_of_value_type v, "boolean"))
      )
  | RDeclaration(left, right) ->
      let (env, value) = evaluate env right in
      (match left with
        | RVariable(loc) -> (env, Environment.set_value env value loc)
        | RMemberExpr(expr, key) ->
            let (h, index) = get_member_expr_map env expr key
            in Hashtbl.replace h index value; (env, value)
        | _ -> raise ELeftSideCannotBeAssigned
      )
  | RAssignment(left, right) ->
      let (env, value) = evaluate env right in
      (match left with
        | RVariable(loc) ->
            let oldvalue = Environment.get_value env loc
            in (if value_type oldvalue = value_type value then
                (env, (Environment.set_value env value loc))
              else
                raise ( ETypeMismatchInAssignment(Environment.get_loc_name env loc,
string_of_value_type oldvalue, string_of_value_type value)))

        | RMemberExpr(expr, key) ->
            let (h, index) = get_member_expr_map env expr key
            in let oldvalue = Hashtbl.find h index
            in (if value_type oldvalue = value_type value then
                (Hashtbl.replace h index value; (env, value))
              else
                raise ( ETypeMismatchInAssignment(index, string_of_value_type oldvalue,
string_of_value_type value)))
        | _ -> raise ELeftSideCannotBeAssigned
      )
  | RFunctionCall(fexpr, args_expr) ->
      let (this, func) = resolve_func_this env fexpr
      in let value_list = (evaluate_expr_list env args_expr)
      in run_function env value_list this func
  | RVarArg(_) ->
      raise (RuntimeError.InternalError "unexpected expression in evaluate")
(**
Resolves a function call by an expression into a function and a this object
```

```
@param env runtime environment
@param fexpr the expression to analyze
@return a tuple of the this object and the function
*)
and resolve_func_this env fexpr =
  let find_prototype h =
    match Hashtbl.find h "prototype" with
    | RMapValue(h, MapSubtype) -> h
    | _ -> raise Not_found
  in let find_func h name =
    match Hashtbl.find h name with
    | RFunctionValue(_, _, _, _, _, _, _) | RLibraryFunction(_) as v -> v
    | _ -> raise Not_found
  in let rec find_map_func h = function
    | "prototype":: tl -> find_map_func (find_prototype h) tl
    | name:: tl -> find_func h name
    | [] -> raise (RuntimeError.InternalError "map function find")
  in match fexpr with
  | RMemberExpr(this_expr, funcname) ->
      let (env, this) = evaluate env this_expr
      in let f = match this with
        | RUndefined -> raise (RuntimeError.InternalError "unexpected undefined this in function
resolution")
        | RStringValue(_) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(1, 1)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RIntegerValue(_) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(2, 2)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RFloatValue(_) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(3, 3)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RBooleanValue(_) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(4, 4)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RFunctionValue(_, _, _, _, _, _, _) | RLibraryFunction(_) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(5, 5)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RVoid ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(6, 6)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RMapValue(_, ArraySubtype) ->
            let (env, v) = evaluate env (RMemberExpr(RMemberExpr(RVariable(GlobalVar(8, 8)),
RValue(RStringValue("prototype"))), funcname))
            in v
        | RMapValue(h, MapSubtype) ->
            let (env, value) = evaluate env funcname
            in let name = string_of_value value
            in try find_func h name
            with | Not_found ->
                try find_map_func h ["prototype"; name]
                with | Not_found ->
                    try find_map_func h ["prototype";"prototype"; name]
                    with | Not_found ->
                        try
                          let (env, value) = evaluate env
(RMemberExpr(RMemberExpr(RVariable(GlobalVar(9, 9)), RValue(RStringValue("prototype"))),
funcname))
                          in value
                        with EUndefinedMapMember _ -> raise (EUndefinedMapMember name)
      in (this, f)
  | _ ->
      let (env, v) = evaluate env fexpr
      in (RVoid, v)
(**
Runs a function
@param env runtime environment
@param value_list list of values to pass as arguments
@param this this pointer
@param func function
@return a tuple of the environemt and return value
*)
and run_function env value_list this func =
  match func with
  | RFunctionValue(framesize, depth, argslen, vararg, stmts, closure_vars, _) ->
      let old_frame = env.stackframes.(depth)
      in let _ = env.stackframes.(depth) <- make_stackframe framesize argslen vararg value_list
this
      in let old_closure_vars = env.closure_vars
      in let old_skip_callstack_pop = env.skip_callstack_pop
      in let _ = env.closure_vars <- closure_vars
      in (try
        (if Stack.is_empty env.callstack or Stack.top env.callstack!= env.current_line then
            (Stack.push env.current_line env.callstack;
              env.skip_callstack_pop <- false)
          else
            env.skip_callstack_pop <- true
        );
        interpret_stmts env stmts;
        (if env.skip_callstack_pop then ()
          else let _ = Stack.pop env.callstack in ());
        env.skip_callstack_pop <- old_skip_callstack_pop;
        env.stackframes.(depth) <- old_frame;
        env.closure_vars <- old_closure_vars;
        (env, RVoid)
      with
      | CFReturn value ->
          (if env.skip_callstack_pop then ()
            else let _ = Stack.pop env.callstack in ());
          env.skip_callstack_pop <- old_skip_callstack_pop;
          env.stackframes.(depth) <- old_frame;
          env.closure_vars <- old_closure_vars;
          (env, value)
      | ex ->
          (if env.skip_callstack_pop then ()
            else let _ = Stack.pop env.callstack in ());
          env.skip_callstack_pop <- old_skip_callstack_pop;
          env.stackframes.(depth) <- old_frame;
          env.closure_vars <- old_closure_vars;
          raise ex)
  | RLibraryFunction(def) ->
      let old_frame = env.stackframes.(0)
      in let old_skip_callstack_pop = env.skip_callstack_pop
      in env.stackframes.(0) <- make_stackframe def.num_args def.num_args def.vararg value_list
this;
      (try
        (if Stack.is_empty env.callstack or Stack.top env.callstack!= env.current_line then
            (Stack.push env.current_line env.callstack;
              env.skip_callstack_pop <- false)
          else
            env.skip_callstack_pop <- true
        );
        def.code env;
        (if env.skip_callstack_pop then ()
          else let _ = Stack.pop env.callstack in ());
        env.skip_callstack_pop <- old_skip_callstack_pop;
        env.stackframes.(0) <- old_frame;
        (env, RVoid)
      with
      | CFReturn value ->
          (if env.skip_callstack_pop then ()
            else let _ = Stack.pop env.callstack in ());
          env.skip_callstack_pop <- old_skip_callstack_pop;
          env.stackframes.(0) <- old_frame;
          (env, value)
      | ex ->
          (if env.skip_callstack_pop then ()
            else let _ = Stack.pop env.callstack in ());
          env.skip_callstack_pop <- old_skip_callstack_pop;
          env.stackframes.(0) <- old_frame;
          raise ex)
  | _ -> raise ENotAFunction

(**
Determines the value and type of expression for the last member of a member expression
@param env the runtime environment
@param index the expression to evaluate
@return a tuple with the index of the expression and a boolean indicating
whether it is an integer
*)
and evaluate_memb_expr_index env index =
  (match evaluate env index with
    | (_, RStringValue(s)) -> (s, false)
    | (_, RIntegerValue(i)) -> (string_of_int i, true)
    | (_, v) -> raise (EInvalidMember(string_of_value_type v, string_of_value v))
  )
(**
Returns the hashmap that corresponds to the member expression
@param env the runtime environment
@param expr the member expression (without the last member)
@param index the index (the last member of the member expression)
@return the hashmap that corresponds to the member expression
*)
and get_member_expr_map env expr index =
  let (env, left) = evaluate env expr
  in let (index, is_int) = evaluate_memb_expr_index env index
  in (match left with
    | RMapValue(h, ArraySubtype) ->
        (if not is_int then raise (EInvalidArrayIndex("string", index))
          else
            try
```

```
              let _ = Hashtbl.find h index in (h, index)
            with
            | Not_found -> raise (EArrayIndexOutOfBounds index)
        )
    | RMapValue(h, MapSubtype) -> (h, index)
    | _ -> raise (ELeftSideIsNotAMap(string_of_value_type left, string_of_value left))
  )
(**
Evaluates a list of expressions
@param env the runtime environment
@param expr_list an expression list
@param a list of the corresponding value for each expression
*)
and evaluate_expr_list env expr_list =
  let rec loop result = function
    | [] -> List.rev result
    | RVarArg(loc):: tl ->
        loop (List.concat [
              (let (env, v) = evaluate env (RVariable(loc))
                in match v with
                | RMapValue(_, ArraySubtype) as arr -> List.rev (list_of_array arr)
                | _ -> raise (RuntimeError.InternalError "expected array while expanding
args")); result]) tl
    | expr:: tl -> let (env, v) = evaluate env expr in loop (v:: result) tl
  in loop [] expr_list
```

jtemplate.ml

```
(**
Jtemplate initialization and launching of program

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Lexer
open Parser
open Lexing
open Ast
open Filename_util
open RuntimeError
open Environment
open Analysis
open Ast_info

(**
Program entry point
*)
let _ =
  (**
  Registers script arguments (arguments following the script file)
  @param args a list of string arguments
  @param env runtime environment
  @return unit
  *)
  let register_args args env =
    let h = Hashtbl.create 1
    in let rec loop = function
      | (n,[]) ->
          Hashtbl.add h "length" (RIntegerValue(n));
          Environment.set_value env (RMapValue(h, ArraySubtype)) (GlobalVar(0, 0))
      | (ind, arg:: tl) -> Hashtbl.add h (string_of_int ind) (RStringValue(arg)); loop (ind + 1,
tl)
    in loop (0, args);
  (* parse command line arguments *)
  in let show_parse_tree = ref false and args_list = ref [] and print_version = ref false
  in let _ = Arg.parse
      [("-parsetree", Arg.Unit (fun () -> show_parse_tree := true), "print the parse tree and
symbols before executing the program");
      ("-version", Arg.Unit (fun () -> print_version := true), "print the version and exit")
      ] (fun arg -> args_list:= (arg::!args_list)) ("jtemplate [-p] scriptfile [scriptargs...]"
^
        "scriptfile the script file to execute, read from stdin if missing\n"^"scriptargs...
optional script arguments, separated by a space")
  in let _ = (if !print_version then (print_string "Jtemplate 0.8\n"; exit(0)) else ())
  in let args = List.rev !args_list
  in let filename = match args with
    | [] -> "-"
    | name:: tl -> name
  (* generate parsing AST *)
  in let ast = try
      if filename ="-" then Parser_util.parse stdin "stdin"
      else (
        let resolved_filename = resolve_filename (Unix.getcwd()) filename
        in Unix.chdir (Filename.dirname resolved_filename);
        Parser_util.parse_filename (resolved_filename)
      )
    with ParseException(_) as ex ->
        RuntimeError.display_error ex ("", 0);
        exit(2)
  (* analyze AST and create optimized runtime AST *)
  in let (ast, env) =
    try Analysis.analyze ast
    with
    | RuntimeError.FatalExit(_) -> exit(2)
    | ex ->
        RuntimeError.display_error ex ("", 0);
        exit(2)
  (* show parse tree if dictated by command line switch *)
  in let _ = (if !show_parse_tree then
        (Ast_info.print_ast ast; print_name_info env)
      else
        ())
  in
  (* create a runtime environment *)
  let renv ={
    heap = Array.make env.num_globals (- 1, RUndefined);
    stackframes = Array.make (env.max_depth + 1) [||];
    closure_vars = None;
    gnames = Array.of_list env.names;
    current_line = ("", 0);
    callstack = Stack.create ();
    skip_callstack_pop = false;
  }
  (* copy library definitions to runtime environment *)
  in let _ = Library.register_for_runtime env renv
  in let _ = register_args args renv
  (* interpret runtime AST *)
  in try
    Interpreter.interpret renv ast
  with
  | RuntimeError.FatalExit _ -> exit(1)
  | ex ->
      RuntimeError.display_error ex renv.current_line;
      Stack.iter (fun loc -> let (file, line) = loc in
              print_string ("\tCalled from " ^ file ^ " line " ^ (string_of_int line)))
renv.callstack;
      exit(1)
```

library.ml

```
(**
Registration of libraries

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Ast
open Environment

(**
Registers all library functions and params in an analysis environment
@param env analysis environment
@return a modified environment with all library functions registered
*)
let register_for_analysis env =
  let env = Environment.declare_variable_and_value env "args" (RMapValue(Hashtbl.create 10,
ArraySubtype))
  in let env = Environment.declare_variable_and_value env "String" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "Integer" (RMapValue(Hashtbl.create
10, MapSubtype))
  in let h = Hashtbl.create 10
  in let _ = Hashtbl.add h "infinity" (RFloatValue(infinity))
  in let _ = Hashtbl.add h "nan" (RFloatValue(nan))
  in let _ = Hashtbl.add h "negativeInfinity" (RFloatValue(neg_infinity))
  in let env = Environment.declare_variable_and_value env "Float" (RMapValue(h, MapSubtype))
  in let env = Environment.declare_variable_and_value env "Boolean" (RMapValue(Hashtbl.create
10, MapSubtype))
  in let env = Environment.declare_variable_and_value env "Function" (RMapValue(Hashtbl.create
10, MapSubtype))
```

```
  in let env = Environment.declare_variable_and_value env "Void" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "Nan" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "Array" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "Map" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "void" RVoid
  in let rec define_map_lib_call h libdef = function
    | name::[] -> Hashtbl.replace h name libdef
    | name:: tl ->
      let h = (try
          match Hashtbl.find h name with
          | RMapValue(h, MapSubtype) -> h
          | _ -> raise (RuntimeError.InternalError "inconsistent library call definition")
        with Not_found ->
          let nh = Hashtbl.create 10
          in Hashtbl.replace h name (RMapValue(nh, MapSubtype)); nh)
      in define_map_lib_call h libdef tl
    | [] -> raise (RuntimeError.InternalError "invalid library call definition")
  in let rec loop env = function
    | [] -> env
    | def:: tl ->
      (match def.name with
        | name::[] ->
          let env = Environment.declare_variable_and_value env name (RLibraryFunction(def))
          in loop env tl
        | name:: name_rest ->
          let (h, env) = (try
              match resolve_variable_value name env with
              | (RMapValue(h, MapSubtype), loc) -> (h, env)
              | _ -> raise (RuntimeError.InternalError "inconsistent library call
definition")
            with
            | Variable_not_found(_) ->
              let h = Hashtbl.create 10
              in let env = Environment.declare_variable_and_value env name (RMapValue(h,
MapSubtype))
              in (h, env)
            )
          in define_map_lib_call h (RLibraryFunction(def)) name_rest;
          loop env tl
        | [] -> raise (RuntimeError.InternalError "invalid library call definition")
      )
  in let (exported, env) = Library_builtin.initialize env
  in let env = loop env exported
  in let (exported, env) = Library_string.initialize env
  in let env = loop env exported
  in let (exported, env) = Library_io.initialize env
  in loop env exported

(**
Registers library functions into a runtime environment
@param env analysis environment from which definitions will be transferred
@param renv runtime environment into which definitions will be transferred
@return unit
*)
let register_for_runtime env renv =
  let rec process rmap =
    StringMap.fold(fun k v _ ->
            let (ind, uid) = v
            in try
              let value = get_constant_value env uid
              in renv.heap.(ind) <- (uid, value)
            with Not_found -> ()
      ) rmap.variable_map ();
    match rmap.parent with
    | None -> ()
    | Some m -> process m
  in process env.globals
```

lib_builtin.ml

```
(**
Built in library implementation

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Environment
open Ast
open Expression
open RuntimeError
open Unix

(**
Entry point for library initialization
@return a list of exported functions
*)
let initialize env =
  let this_array_map env =
    match env.stackframes.(0).(0) with
    | RMapValue(h, ArraySubtype) -> h
    | _ -> raise (InternalError "expected array for this")
  in let this_map_map env =
    match env.stackframes.(0).(0) with
    | RMapValue(h, MapSubtype) -> h
    | _ -> raise (InternalError "expected map for this")
  in let this_float env =
    match env.stackframes.(0).(0) with
    | RFloatValue(f) -> f
    | _ -> raise (InternalError "expected float for this")
  in let _ = Random.self_init()
  in let date_map = Hashtbl.create 2
  in let date_proto_map = Hashtbl.create 1
  in let _ = Hashtbl.replace date_map "prototype" (RMapValue(date_proto_map, MapSubtype))
  in let env = Environment.declare_variable_and_value env "Date" (RMapValue(date_map,
MapSubtype))
```

```
  in let env = Environment.declare_variable_and_value env "Debug" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "System" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in ([
    {
      name = ["Array";"prototype";"push"];
      args = ["value"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let value = env.stackframes.(0).(1)
            in let array = this_array_map env
            in let len = string_of_value (Hashtbl.find array "length") in
            Hashtbl.replace array len value;
            Hashtbl.replace array "length" (RIntegerValue((int_of_string len) + 1))
    };
    {
      name =["Array";"prototype";"pop"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
            let hashtbl = this_array_map env
            in let len = int_of_string (string_of_value (Hashtbl.find hashtbl "length")) in
            if len = 0 then
              raise (LibraryError "Error while attempting to pop an empty array in Array.pop")
            else
              let result = Hashtbl.find hashtbl (string_of_int (len - 1)) in
              Hashtbl.remove hashtbl (string_of_int (len - 1));
              Hashtbl.replace hashtbl "length" (RIntegerValue(len - 1));
              raise (CFReturn result)
    };
    {
      name = ["Array";"prototype";"length"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
            let hashtbl = this_array_map env
            in try (match Hashtbl.find hashtbl "length" with
                | RIntegerValue(len) -> raise (CFReturn(RIntegerValue(len)))
                | _ -> raise (LibraryError "First parameter is not an array in call to
Array.length"))
            with Not_found ->
                raise (LibraryError "First parameter is not an array in call to Array.length")
    };
    {
      name =["Map";"prototype";"remove"];
      args =["key"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let hashtbl = this_map_map env
            in let key = string_of_value (env.stackframes.(0).(1))
            in let _ = Hashtbl.remove hashtbl key
            in ()
    };
    {
      name =["Map";"prototype";"contains"];
      args =["key"];
      num_args = 1;
      vararg = false;
      code = fun env ->
             let hashtbl = this_map_map env
             in let key = string_of_value (env.stackframes.(0).(1))
             in raise (CFReturn(RBooleanValue(Hashtbl.mem hashtbl key)))
    };
    {
      name =["Map";"prototype";"keys"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
             let hashtbl = this_map_map env
             in let result = Hashtbl.create 10
             in let (_, cnt) = Hashtbl.fold (fun k _ (h, cnt) -> (Hashtbl.add h (string_of_int
cnt) (RStringValue k); h, cnt + 1 )) hashtbl (result, 0)
             in Hashtbl.replace result "length" (RIntegerValue cnt );
             raise (CFReturn (RMapValue (result, ArraySubtype)))
    };
    {
      name =["Integer";"random"];
      args =["upperBound"];
      num_args = 1;
      vararg = false;
      code = fun env ->
             let upperBound =
               (try
                 cast_to_integer(env.stackframes.(0).(1))
               with
               | _ -> raise (LibraryError("upperBound must an integer in call to
Integer.random")))
             in
             raise (CFReturn(RIntegerValue(Random.int upperBound)))
    };
    {
      name =["Float";"prototype";"round"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
             let f = this_float env
             in raise (CFReturn (RIntegerValue(int_of_float(
                         let (frac, _) = modf f in (if frac >= 0.5 then ceil f else floor f)))))
    };
    {
      name =["Date";"now"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
             let t = (localtime (time())) in
             let gmt_offset = (localtime (time())).tm_hour - (gmtime (time())).tm_hour in
             let h = Hashtbl.create 10 in
             Hashtbl.add h "prototype" (RMapValue(date_proto_map, MapSubtype));
             Hashtbl.add h "second" (RIntegerValue(t.tm_sec));
             Hashtbl.add h "minute" (RIntegerValue(t.tm_min));
```

```
            Hashtbl.add h "hour" (RIntegerValue(t.tm_hour));
            Hashtbl.add h "dayOfMonth" (RIntegerValue(t.tm_mday));
            Hashtbl.add h "month" (RIntegerValue(t.tm_mon + 1));
            Hashtbl.add h "year" (RIntegerValue(1900 + t.tm_year));
            Hashtbl.add h "dayOfWeek" (RIntegerValue(t.tm_wday)); (* Sunday 0 *)
            Hashtbl.add h "dayOfYear" (RIntegerValue(t.tm_yday));
            Hashtbl.add h "dst" (RBooleanValue(t.tm_isdst));
            Hashtbl.add h "gmtOffset" (RIntegerValue(gmt_offset));
            raise(CFReturn(RMapValue(h, MapSubtype)))
    };
    {
      name =["System";"command"];
      args = ["command"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let command = string_of_value (env.stackframes.(0).(1)) in
            raise (CFReturn (RIntegerValue(Sys.command command)))
    };
    {
      name =["exit"];
      args =["exitcode"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            match env.stackframes.(0).(1) with
            | RIntegerValue(c) -> if c >= 0 && c <= 255 then exit c else
                  raise (LibraryError("exitcode must be an integer between 0 and 255 in call to
exit"))
            | _ -> raise (LibraryError("exitcode must be an integer in call to exit"))
    };
    {
      name =["Debug";"dumpSymbolTable"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
            let rec loop = function
              | 0 -> ()
              | n ->
                  let (uid, value) = env.heap.(n - 1)
                  in let _ = print_string (env.gnames.(uid)^" = " ^(string_of_value value) ^
"\n")
                  in loop (n - 1)
            in loop (Array.length env.heap)
    };
    {
      name =["Function";"prototype";"apply"];
      args =["args..."];
      num_args = 1;
      vararg = true;
      code = fun env ->
            let func = env.stackframes.(0).(0)
            in match func with
            | RFunctionValue(_, _, _, _, _, _, _) | RLibraryFunction(_) ->
                let args = list_of_array env.stackframes.(0).(1)
                in let this = List.hd args
                in let args = List.tl args
                in let (_, v) = Interpreter.run_function env args this func
                in raise (CFReturn v)
            | _ -> raise (LibraryError "expected a function in first parameter of call to
apply")
    };
    {
      name =["Debug";"dumpStackTrace"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
            Stack.iter (fun loc ->
                    let (file, line) = loc
                    in print_string ("Called from line "^(string_of_int line)^" in file "^
(Filename.basename file)^":\n"))
              env.callstack
    };
    {
      name = ["typeof"];
      args =["value"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let s = match env.stackframes.(0).(1) with
              | RStringValue(_) -> "string"
              | RIntegerValue(_) ->"integer"
              | RFloatValue(_) -> "float"
              | RBooleanValue(_) -> "boolean"
              | RFunctionValue(_) | RLibraryFunction(_) -> "function"
              | RMapValue(_, ArraySubtype) -> "array"
              | RMapValue(_, MapSubtype) ->"map"
              | RVoid ->"void"
              | RUndefined -> "undefined"
            in raise (CFReturn (RStringValue s))
    };
    ], env)
```

lib_io.ml

```
(**
I / O library implementation

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open Environment
open Ast
open Expression
open RuntimeError
open Unix

(** internal error raised to indicate a inconsistent usage of a handle *)
```

```
exception EIOPassthrough of string

(** type of I/O channel *)
type channelType = OutChannel of out_channel | InChannel of in_channel * (string * bool)

(**
Entry point for library initialization
@return a list of exported functions
*)
let initialize env =
  let env = Environment.declare_variable_and_value env "File" (RMapValue(Hashtbl.create 10,
MapSubtype))
  in let env = Environment.declare_variable_and_value env "Directory" (RMapValue(Hashtbl.create
10, MapSubtype))
  in let descriptors = Hashtbl.create 10 in
  let get_descriptor handle command =
    (try
      Hashtbl.find descriptors handle
    with
    | Not_found -> raise (EIOPassthrough("invalid handle for "^command)))
  in let try_read ch =
    (
      try
        (input_line ch, false)
      with
      | End_of_file -> ("", true)
    )
  in ([
    {
      name = ["print"];
      args = ["values..."];
      num_args = 1;
      vararg = true;
      code = fun env ->
            match env.stackframes.(0).(1) with
            | RMapValue(t, ArraySubtype) ->
                List.iter (fun value ->
                        print_string (string_of_value value)) (list_of_array (RMapValue(t,
ArraySubtype)))
            | _ -> raise (RuntimeError.LibraryError "expected vararg for first parameter of
print")
    };
    {
      name = ["println"];
      args = ["values..."];
      num_args = 1;
      vararg = true;
      code = fun env ->
            match env.stackframes.(0).(1) with
            | RMapValue(t, ArraySubtype) ->
                List.iter (fun value ->
                        print_string (string_of_value value)) (list_of_array (RMapValue(t,
ArraySubtype)));
                print_newline()
            | _ -> raise (RuntimeError.LibraryError "expected vararg for first parameter of
println")
    };
    {
      name =["readln"];
      args =[];
      num_args = 0;
      vararg = false;
      code = fun env ->
            raise (CFReturn(RStringValue(read_line())))
    };
    {
      name =["File";"openForWriting"];
      args =["handle"; "filename"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let filename = string_of_value (env.stackframes.(0).(2)) in
            try
              if Hashtbl.mem descriptors handle then
                raise (EIOPassthrough("handle "^handle^" is already opened in call to
openForWriting"))
              else
                let ch = open_out filename in
                Hashtbl.add descriptors handle (OutChannel(ch), filename)
            with
            | EIOPassthrough(msg) -> raise (LibraryError msg)
            | _ -> raise (LibraryError ("error opening file "^filename^" in
openFileForWriting"))
    };
    {
      name =["File";"openForReading"];
      args =["handle"; "filename"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let filename = string_of_value (env.stackframes.(0).(2)) in
            try
              if Hashtbl.mem descriptors handle then
                raise (EIOPassthrough("handle "^handle^" is already opened in call to
openForReading"))
              else
                let ch = open_in filename in
                Hashtbl.add descriptors handle (InChannel(ch, (try_read ch)), filename)
            with
            | EIOPassthrough(msg) -> raise (LibraryError msg)
            | _ -> raise (LibraryError ("error opening file "^filename^" in
openFileForReading"))
    };
    {
      name =["File";"close"];
      args =["handle"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let (c, _ ) = get_descriptor handle "closeFile" in
            try
              (match c with
                | OutChannel(ch) -> close_out ch
                | InChannel(ch, _) -> close_in ch);
              Hashtbl.remove descriptors handle
            with
            | EIOPassthrough(msg) -> raise (LibraryError msg)
            | Sys_error msg -> raise (LibraryError ("System error on closeFile:" ^ msg ))
    };
    {
      name =["File";"write"];
      args =["handle"; "values..."];
      num_args = 2;
      vararg = true;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let (c, filename) = get_descriptor handle "write" in
            match c with
            | OutChannel(ch) ->
                ( try
                  let _ = List.map (fun el -> output_string ch (string_of_value el))
                      (list_of_array (env.stackframes.(0).(2))) in ()
                with
                | EIOPassthrough(msg) -> raise (LibraryError msg)
                | _ -> raise (LibraryError ("error writing file "^filename^" in write")))
            | InChannel(ch, _) -> raise (LibraryError ("invalid handle in call to write. Handle
"^handle^" was opened for reading "^filename))
    };
    {
      name =["File";"writeln"];
      args =["handle"; "values..."];
      num_args = 2;
      vararg = true;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let (c, filename) = get_descriptor handle "writeln" in
            match c with
            | OutChannel(ch) ->
                (try
                  let _ = List.map (fun el -> output_string ch (string_of_value el))
                      (list_of_array (env.stackframes.(0).(2))) in
                  output_string ch ( "\n")
                with
                | EIOPassthrough(msg) -> raise (LibraryError msg)
                | _ -> raise (LibraryError ("error writing file "^filename^" in writeln")))
            | InChannel(ch, _) -> raise (LibraryError ("invalid handle in call to write. Handle
"^handle^" was opened for reading "^filename))
    };
    {
      name =["File";"readln"];
      args =["handle"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let (c, filename) = get_descriptor handle "readln" in
            match c with
            | OutChannel(ch) -> raise (EIOPassthrough ("invalid handle in call to readln. Handle
"^handle^" was opened for writing "^filename))
            | InChannel(ch, (_, true)) -> raise (EIOPassthrough ("End of file reached for handle
"^handle^" in call to readln"))
            | InChannel(ch, (data, false)) ->
                ( try
                  Hashtbl.replace descriptors handle (InChannel(ch, (try_read ch)), filename)
                with
                | EIOPassthrough(msg) -> raise (LibraryError msg)
                | _ -> raise (LibraryError ("error reading file "^filename^" in readln")));
                raise (CFReturn(RStringValue(data)))
    };
    {
      name =["File";"eof"];
      args =["handle"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let handle = string_of_value (env.stackframes.(0).(1)) in
            let (c, filename) = get_descriptor handle "eof" in
            match c with
            | OutChannel(ch) -> raise (EIOPassthrough("invalid handle in call to eof. Handle
"^handle^" was opened for writing "^filename))
            | InChannel(ch, (_, eof)) -> raise (CFReturn(RBooleanValue(eof)))
    };
    {
      name =["File";"exists"];
      args =["filename"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let filename = string_of_value (env.stackframes.(0).(1)) in
            raise (CFReturn(RBooleanValue (Sys.file_exists filename)))
    };
    {
      name =["File";"delete"];
      args =["name"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let name = string_of_value (env.stackframes.(0).(1)) in
            try
              unlink name;
              raise (CFReturn(RBooleanValue(true)))
            with
            | _ -> raise (CFReturn(RBooleanValue(false)))

    };
    {
      name =["File";"rename"];
      args =["fromname";"toname"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            let fromname = string_of_value (env.stackframes.(0).(1)) in
            let toname = string_of_value (env.stackframes.(0).(2)) in
            try
              rename fromname toname;
              raise (CFReturn(RBooleanValue(true)))
            with
            | _ -> raise (CFReturn(RBooleanValue(false)))
    };
    {
      name =["Directory";"create"];
      args =["name"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let name = string_of_value (env.stackframes.(0).(1)) in
            try
              mkdir name 0o777;
              raise (CFReturn(RBooleanValue(true)))
            with
            | CFReturn _ as e -> raise e
            | _ -> raise (CFReturn(RBooleanValue(false)))
    };
    {
      name =["Directory";"delete"];
      args =["name"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let name = string_of_value (env.stackframes.(0).(1)) in
            try
              rmdir name;
              raise (CFReturn(RBooleanValue(true)))
            with
            | CFReturn _ as e -> raise e
            | _ -> raise (CFReturn(RBooleanValue(false)))
    };
    {
      name =["Directory";"list"];
      args =["name"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let name = string_of_value (env.stackframes.(0).(1)) in
            let arr = (try
                let handle = opendir name in
                let h = Hashtbl.create 10
                in let rec loop cnt =
                  try
                    Hashtbl.add h (string_of_int cnt) (RStringValue(readdir handle));
                    loop (cnt + 1)
                  with
                  | End_of_file -> closedir handle; cnt
                  | _ -> closedir handle; raise (CFReturn(RVoid))
                in Hashtbl.add h "length" (RIntegerValue(loop 0));
                h
              with
              | _ -> raise (CFReturn(RVoid)))
            in raise (CFReturn(RMapValue(arr, ArraySubtype)));
    };
    {
      name =["Directory";"exists"];
      args =["name"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let name = string_of_value (env.stackframes.(0).(1)) in
            raise (CFReturn(RBooleanValue((try
                      Sys.is_directory name
                    with
                    | _ -> raise (CFReturn(RBooleanValue(false)))))))
    };
```

```
    ], env)
```

lib_string.ml

```
(**
String library implementation

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                        *)
open Environment
open Ast
open Expression
open RuntimeError

(**
Returns the value of this as a string
@param env the runtime environment
@return the string value of this
@raise InternalError is this is not a string
*)
let get_this env =
  match env.stackframes.(0).(0) with
  | RStringValue(s) -> s
  | v -> raise (RuntimeError.InternalError "mismatched this in call to String.prototype.length")

(**
Returns the positing of a substring within a string
@param str the string to search
@param substr the substring to find
@return the position of the substring in the string, or - 1 if not found
*)
let indexOf str substr =
  let ssl = String.length substr in
  let max = String.length str - ssl in
  let rec loop i =
    (if i > max then - 1
      else(
        if String.sub str i ssl = substr then i
        else loop (i + 1))
    )
  in loop 0

(**
Entry point for library initialization
@return a list of exported functions
*)
let initialize env =
  ([{
      name = ["String";"prototype";"length"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env -> raise (CFReturn (RIntegerValue(String.length (get_this env))))
    };
    {
      name =["String";"prototype";"toUppercase"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env -> raise (CFReturn (RStringValue(String.uppercase (get_this env))))
    };
    {
      name =["String";"prototype";"toLowercase"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env -> raise (CFReturn (RStringValue(String.lowercase (get_this env))))
    };
    {
      name =["String";"prototype";"toFirstUpper"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env -> raise (CFReturn (RStringValue(String.capitalize (get_this env))))
    };
    {
      name =["String";"prototype";"toFirstLower"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env -> raise (CFReturn (RStringValue(String.uncapitalize (get_this env))))
    };
    {
      name =["String";"prototype";"charAt"];
      args = ["index"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            match env.stackframes.(0).(1) with
              RIntegerValue(index) ->
                (try
                  raise (CFReturn (RStringValue (String.make 1 (String.get (get_this env)
index))))
                with
                | Invalid_argument _ -> raise (LibraryError ("invalid index "^(string_of_int
index)^" in call to charAt"))
            | _ -> raise (LibraryError "argument index in charAt should be an integer")
    };
    {
      name =["String";"prototype";"indexOf"];
      args = ["substring"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let substring = string_of_value (env.stackframes.(0).(1))
            in raise (CFReturn (RIntegerValue(indexOf (get_this env) substring)))
    };
    {
      name =["String";"prototype";"substr"];
      args = ["start";"length"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            match env.stackframes.(0).(1) with
            | RIntegerValue(start) ->
                (match env.stackframes.(0).(2) with
                  | RIntegerValue(length) -> raise (CFReturn (RStringValue(String.sub (get_this
env) start length)))
                  | _ -> raise (LibraryError "length in substr should be an integer"))
            | _ -> raise (LibraryError "start in substr should be an integer")
    };
    {
      name =["String";"prototype";"startsWith"];
      args = ["substring"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let substring = string_of_value (env.stackframes.(0).(1))
            in raise (CFReturn (RBooleanValue(substring = String.sub (get_this env) 0
(String.length substring))))
    };
    {
      name =["String";"prototype";"endsWith"];
      args = ["substring"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let ss = string_of_value (env.stackframes.(0).(1))
            in let s = get_this env
            in raise (CFReturn (RBooleanValue(ss = String.sub s (String.length s - String.length
ss) (String.length ss))))
    };
    {
      name =["String";"prototype";"replaceAll"];
      args = ["substring";"replacement"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            let substr = string_of_value (env.stackframes.(0).(1))
            in let repl = string_of_value (env.stackframes.(0).(2))
            in let s = get_this env
            in let ssl = String.length substr in
            let rec loop str =
              match indexOf str substr with
              | - 1 -> raise (CFReturn(RStringValue str))
              | i -> loop ((String.sub str 0 i)^ repl ^
                      (String.sub str (i + ssl)
                          (String.length str - ssl - i)))
            in loop s
    };
    {
      name =["String";"prototype";"split"];
      args = ["delim"];
      num_args = 1;
      vararg = false;
      code = fun env ->
            let str = get_this env
            in let substr = string_of_value (env.stackframes.(0).(1))
            in let result = Hashtbl.create 10
            in let rec loop s ind =
              match indexOf s substr with
              | - 1 -> Hashtbl.add result (string_of_int ind) (RStringValue(s));
                  Hashtbl.add result "length" (RIntegerValue(ind + 1));
                  raise (CFReturn (RMapValue(result, ArraySubtype)))
              | i -> let offset = i + String.length substr in
                  Hashtbl.add result (string_of_int ind) (RStringValue(String.sub s 0 i));
                  loop (String.sub s offset (String.length s - offset)) (ind + 1) in
            loop str 0
    };
    {
      name =["String";"prototype";"parseInt"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env ->
            raise (CFReturn( try RIntegerValue(int_of_string (get_this env)) with Failure _ ->
RVoid ))
    };
    {
      name =["String";"prototype";"parseFloat"];
      args = [];
      num_args = 0;
      vararg = false;
      code = fun env ->
            raise (CFReturn( try RFloatValue(float_of_string (get_this env)) with Failure _ ->
RVoid ))
    };
    {
      name =["String";"prototype";"mreplace"];
      args =["substrings";"values"];
      num_args = 2;
      vararg = false;
      code = fun env ->
            let rec make_array result = function
              | [] -> Array.of_list(List.rev result)
              | v:: tl -> make_array ((string_of_value v):: result) tl
            in let string = string_of_value env.stackframes.(0).(0)
            in let substrings = make_array [] ( list_of_array env.stackframes.(0).(1))
            in let values = make_array [] (list_of_array env.stackframes.(0).(2))
            in let len = Array.length substrings
            in if len!= Array.length values then
              raise (LibraryError "substrings and values arrays should have the same length in
String.prototype.mreplace")
            else
              let rec loop_replacements result ind =
                if ind = len then
                  result
                else
                  let substring = substrings.(ind)
                  in let len = String.length substring
                  in let rec loop_string string result offset =
                    match indexOf string substring with
                    | - 1 -> result
                    | pos ->
                        if pos + len!= String.length string then
                          loop_string (String.sub string (pos + len) ((String.length string) -
pos - len) ) (pos + offset:: result) (offset + pos + len)
                        else
                          loop_string "" (pos + offset:: result) (offset + pos + len)
                  in let positions = loop_string string [] 0
                  in loop_replacements ((values.(ind), len, positions) :: result) (ind + 1)
              in let replacements = loop_replacements [] 0
              in let replace str i ssl repl =
                (String.sub str 0 i)^ repl ^ (String.sub str (i + ssl) (String.length str - ssl
- i))
              in let rec loop_values result_string offset = function
                | [] -> result_string
                | repl:: tl ->
                    let (value, len, positions) = repl
                    in let delta = String.length value - len
                    in let rec loop_repl result_string offset = function
                      | [] -> (offset, result_string)
                      | pos:: tl ->
                          loop_repl (replace result_string pos len value) (offset + delta) tl
                    in let (offset, result_string) = loop_repl result_string offset positions
                    in loop_values result_string offset tl
              in let result = loop_values string 0 replacements
              in raise (CFReturn(RStringValue(result)))
    };
    ], env)
```

parser_util.ml

```
(**
Routines to parse a file

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(* This program is free software; you can redistribute it and / or modify  *)
(* it under the terms of the GNU General Public License as published by    *)
(* the Free Software Foundation; version 3 of the License. This program is *)
(* distributed in the hope that it will be useful, but WITHOUT ANY         *)
(* WARRANTY; without even the implied warranty of MERCHANTABILITY or       *)
(* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    *)
(* for more details.                                                       *)

open RuntimeError

(**
Parse a channel
@param chanel the channel to read the program from
@param name the name to use in error reporting
@return the parse AST
*)
let parse channel name =
  let get_ast lexbuf =
    let _ = Parsing.set_trace false in
    try Parser.program Lexer.main lexbuf
    with
    | RuntimeError.LexerException (msg, line, col) ->
        (print_string
            (msg ^
              (" at line " ^
                ((string_of_int line) ^
                  (" col " ^ ((string_of_int col) ^ "\n")))));
          exit 2)
  in let lexbuf = Lexing.from_channel channel in
  let pos = lexbuf.Lexing.lex_curr_p in
  let _ =
    lexbuf.Lexing.lex_curr_p <-
    { (pos) with Lexing.pos_lnum = 1; Lexing.pos_fname = name; }
  in get_ast lexbuf

(**
Parse a filename
@param filename to parse
@return the parse AST
*)
let parse_filename filename =
  let channel = open_in filename in
  let ast = parse channel filename in
  close_in channel;
  ast
```

runtimeError.ml

```
(**
This module defines runtime errors that are reported to the user

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*)

open Ast

(** this error represents an unexpected condition, caused by a programming
error in the interpreter implementation *)
exception InternalError of string

(** this error is a generic error thrown by library routines to indicate an
error condition, such as the incorrect type of a passed in argument *)
exception LibraryError of string

(** this error is caused by an abnormal error caused by the lexer, such as an
unterminated string **)
exception LexerException of string * int * int

(** marker exception to note that the program should exit, error has already
been reported during analysis *)
exception FatalExit

(** indicates that an assignment was attempted on two incompatible types*)
exception EIncompatibleTypes of string * string (* type1, type2 *)

(** indicates that the value is not of the expected type *)
```

```
exception EInvalidCast of string * string (* value type name, typename *)

(** indicates that an invalid operation was attempted on the specified types *)
exception EInvalidOperation of string * string  (* operator, typename *)

(** indicates that an invalid comparaison was attempted on the given types *)
exception EInvalidComparaison of string * string * string (* comparator, typename *)

(** indicates that a member expression is not applied to a map *)
exception ELeftSideIsNotAMap of string * string (* typename value *)

(** indicates that a member expression is not applied to a map *)
exception ELeftSideIsNotAMap of string * string (* typename value *)

(** indicates an attempt at an assignment to something that is a not a variable or map *)
exception ELeftSideCannotBeAssigned

(** indicates that the map member did not evaluate to a string or integer *)
exception EInvalidMember of string * string (* typename,value *)

(** indicates that a reference was made to a map member that does not exist *)
exception EUndefinedMapMember of string (* value *)

(** indicates a non integer array index *)
exception EInvalidArrayIndex of string * string (* type value *)

(** indicates an out of bounds index *)
exception EArrayIndexOutOfBounds of string (*index*)

(** indicates that the type in the assignment does not match the declare type *)
exception ETypeMismatchInAssignment of string * string * string (* name oldtype new type *)

(** indicates that an incorrect number of arguments were passed to a function *)
exception EMismatchedFunctionArgs of int * int (* expected actual *)

(** indicates an attempt to apply a function to a non function *)
exception ENotAFunction

(** indicates applying for each on a non collection type *)
exception ENotACollectionType of string * string (* message, value *)

(** indicates that the default case should be last *)
exception EDefaultCaseShouldBeLast

(** indicates a parsing error *)
exception ParseException of string

(**
Returns an error message for an exception
@param ex exception
@return error message
*)
let string_of_error ex =
  match ex with
  | InternalError msg -> "INT-00 internal error, interpreter is in inconsistent state: "^msg
  | LibraryError msg -> "LIB-00 library error: "^msg
  | LexerException (msg, line, col) -> "PRS-00 parsing error: " ^ msg ^ " at line " ^
      (string_of_int line) ^ " column " ^ (string_of_int col)
  | ParseException msg -> "PRS-01 parsing error: " ^ msg
  | EIncompatibleTypes(type1, type2) -> "EXP-00 incompatible types " ^ type1 ^ " and " ^ type2
  | EInvalidCast(type1, type2) ->"EXP-01 cannot cast a " ^ type1 ^ " to a " ^ type2
  | EInvalidOperation(operator, type1) -> "EXP-02 invalid operation " ^ operator ^ " for " ^
type1 ^ "s"
  | EInvalidComparaison(operator, type1, type2) -> "EXP-03 invalid comparaison " ^ operator ^ "
for " ^
      type1 ^ " and " ^ type2
  | ELeftSideIsNotAMap (typename, value) -> "EXP-04 left side of member expression is not a map
or array, but a " ^
      typename ^ " with value " ^ value
  | ELeftSideCannotBeAssigned -> "EXP-05 left side of assignment expression cannot be assigned"
  | EInvalidMember (typename, value) -> "EXP-06 member expression did not evaluate to a string
or integer, but to a " ^
      typename ^ " with value " ^ value
  | EUndefinedMapMember(name) -> "EXP-07 member expression " ^ name ^ " is undefined"
  | EInvalidArrayIndex(typename, value) -> "EXP-08 invalid array index of type " ^ typename ^ "
with value " ^ value
  | EArrayIndexOutOfBounds(value) -> "EXP-09 array index out of bounds: " ^ value
  | ETypeMismatchInAssignment(name, shouldbe, is) -> "EXP-10 type mismatch in assignment of " ^
name ^
      " declared as " ^ shouldbe ^ ", attempting to assign "^is
  | EMismatchedFunctionArgs(expected, actual) -> "EXP-11 wrong number of arguments in function
call, expected " ^
      (string_of_int expected) ^ ", got " ^ (string_of_int actual)
  | ENotAFunction -> "EXP-12 invalid function call on a non-function variable"
  | ENotACollectionType (msg, typename) -> "EXP-13 expected a collection type for " ^ msg ^ ",
but got a " ^
      typename
  | Division_by_zero -> "EXP-14 Division by zero"
  | EDefaultCaseShouldBeLast -> "STM-00 the default case in a switch statement should be the
last case"
  | CFReturn _ -> "STM-01 unexpected return statement outside of a function definition"
  | CFBreak -> "STM-02 unexpected break statement outside of a loop"
  | CFContinue -> "STM-03 unexpected continue statement outside of a loop"
  | CFUserException (_, value) -> "USR-00 unhandled user exception " ^ value
  | Parsing.Parse_error -> ""
  | e -> "uncaught exception "^(Printexc.to_string e)

(**
Displays an error to stdout
@param err exception
@param cloc tuple of file, line where error occured
*)
let display_error err cloc =
  let (file, line) = cloc
  in match string_of_error err with
  | "" -> ()
  | msg -> print_string ("\nAt line " ^ (string_of_int line) ^ " in file " ^ file ^ ":\n\t" ^
        msg ^ "\n" )
```

lexer.mll

```
{

(**
Jtemplate lexer

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
*)

open RuntimeError
open Parser
open Ast

(* from http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial.pdf , p.9 *)
let incr_linenum lexbuf =
    let pos = lexbuf.Lexing.lex_curr_p in
    lexbuf.Lexing.lex_curr_p <- { pos with
        Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
        Lexing.pos_bol = pos.Lexing.pos_cnum;
        }

let syntax_exception msg lexbuf=
    raise (LexerException (msg, lexbuf.Lexing.lex_curr_p.Lexing.pos_lnum,
                   lexbuf.Lexing.lex_curr_p.Lexing.pos_cnum -
lexbuf.Lexing.lex_curr_p.Lexing.pos_bol))

let map_id name=
    match name with
      "foreach" -> FOREACH
    | "in" -> IN
    | "while" -> WHILE
    | "function" -> FUNCTION
    | "if" -> IF
    | "else" -> ELSE
    | "template" -> TEMPLATE
    | "instructions" ->INSTRUCTIONS
    | "continue" -> CONTINUE
    | "break" -> BREAK
    | "return" -> RETURN
    | "for" -> FOR
    | "once" -> ONCE
    | "when" -> WHEN
    | "var" -> VAR
    | "let" -> VAR
    | "true" -> BOOLEAN(true)
    | "false" -> BOOLEAN(false)
    | "Void"  -> VOID
    | "import" -> IMPORT(false)
    | "use" -> IMPORT(true)
    | "switch" -> SWITCH
    | "case" -> CASE
    | "default" -> DEFAULT
    | "try" -> TRY
    | "catch" -> CATCH
    | "finally" -> FINALLY
    | "throw" -> THROW
    | _ ->  ID(name)
}

let digit = ['0'-'9']
let id = ['a'-'z' 'A'-'Z' '_' '$']['A'-'Z' 'a'-'z' '0'-'9' '_' '$' ]*
let whitespace = ['\r' '\t' ' ']
let text = '#'[^'\n']*
let float = (( (['0'-'9']+'.'['0'-'9']*) | (['0'-'9']*'.'['0'-'9']+) ) ('e'['+' '-']?['0'-'9']
+)? ) | (['0'-'9']+ ('e'['+' '-']?['0'-'9']+))

(** main lexer *)
rule main = parse
| whitespace { main lexbuf }
| text as token { TEXT(String.sub token 1 ((String.length token) - 1))}
| digit+ as token { try  INT( int_of_string token) with _ -> OUTOFRANGENUMBER }
| float as token { try REAL(float_of_string token) with _ -> OUTOFRANGENUMBER }
| id as token { (map_id token )}
| '\'' { single_quote_string "" lexbuf }
| '"' { double_quote_string "" lexbuf }
| "//" [^'\n']* { main lexbuf}
| "/*" {multiline_comment lexbuf}
| '\n' { incr_linenum lexbuf;main lexbuf }
| "&&" {AND}
| "||" {OR}
| "<" {COMPOP(LessThan)}
| ">" {COMPOP(GreaterThan)}
| "<=" {COMPOP(LessThanEqual)}
| ">=" {COMPOP(GreaterThanEqual)}
| "==" {COMPOP(Equal)}
| "!=" {COMPOP(NotEqual)}
| "..." {DOTDOTDOT}
| "+=" {PLUSEQUALS}
| "-=" {MINUSEQUALS}
| "*=" {TIMESEQUALS}
| "/=" {DIVEQUALS}
| "%=" {MODEQUALS}
| "++" {PLUSPLUS}
| "--" {MINUSMINUS}
| '=' {EQUALS}
| '.' {DOT}
| '{' {LBRACE}
| '}' {RBRACE}
| '(' {LPAREN}
| ')' {RPAREN}
| '[' {LBRACKET}
| ']' {RBRACKET}
| ',' {COMMA}
| ';' {SEMICOLON}
| ':' {COLON}
| '!' {NOT}
| '?' {QUESTION}
| '+' {PLUS}
| '-' {MINUS}
| '*' {TIMES}
| '/' {DIVIDE}
| '%' {MODULO}
| '@' {AT}
| _ as c { syntax_exception ("Invalid character "^String.make 1 c) lexbuf}
| eof { EOF }
(** Lexer for a single quoted string *)
and single_quote_string s = parse
| '\n'  { incr_linenum lexbuf; single_quote_string (s ^ "\n") lexbuf }
| '\''  { STRING(s) }
| '\\' { single_quote_string (s ^ (escape_char lexbuf)) lexbuf }
| [^ '\''] as c  { single_quote_string (s ^ String.make 1 c) lexbuf }
| eof  { syntax_exception "Unterminated string constant" lexbuf }
(** Lexer for a double quoted string *)
and double_quote_string s = parse
| '\n'  { incr_linenum lexbuf; double_quote_string (s ^ "\n") lexbuf }
| '"'  { STRING(s) }
| '\\' { double_quote_string (s ^ (escape_char lexbuf)) lexbuf }
| [^ '"'] as c  { double_quote_string (s ^ String.make 1 c) lexbuf }
| eof  { syntax_exception "Unterminated string constant" lexbuf }
(** Lexer for escape characters in strings *)
and escape_char =parse
| '\\' {"\\"}
| 'n'  {"\n"}
| 'r'  {"\r"}
| '\'' {"'"}
| '"' {"\""}
| 't'  {"\t"}
| 'b'  {"\b"}
(** Lexer for a multiline comment *)
and multiline_comment = parse
| '\n' { incr_linenum lexbuf; multiline_comment lexbuf }
| "*/" { main lexbuf }
| [^ '\n'] { multiline_comment lexbuf}
| eof { syntax_exception "Unterminated multiline comment" lexbuf }


%{
```

parser.mly

```
(**
Jtemplate parser
expression parsing adapted from ECMA-262
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

@author Tony BenBrahim < tony.benbrahim at gmail.com >
*)
(*
This program is free software; you can redistribute it and / or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; version 3 of the License.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.



*)

open Ast

let parse_error s =
    let pos = Parsing.symbol_start_pos() in
    print_string ("in file " ^ (Filename.basename pos.Lexing.pos_fname) ^ ": "^ s^" at line ");
    print_int pos.Lexing.pos_lnum;
    print_string " at columns ";
    print_int (Parsing.symbol_start() - pos.Lexing.pos_bol);
    print_string("-");
    print_int (Parsing.symbol_end() - pos.Lexing.pos_bol);
    print_string "\n";
    flush stdout

let get_env ()=
    let pos=Parsing.symbol_start_pos() in
    (pos.Lexing.pos_fname,pos.Lexing.pos_lnum)


let resolve_import (filename, library, (inp_file, _))=
    Filename_util.resolve_filename (Filename.dirname inp_file) filename

let extract_stmt_list=function
    | StatementBlock(lst) -> lst
    | _ -> raise ( RuntimeError.InternalError "expected statement block" )
%}

%token<string> ID
%token <int> INT
%token <string> STRING
%token <float> REAL
%token <bool> BOOLEAN
%token <string> TEXT
%token <Ast.comparator> COMPOP
%token <bool> IMPORT

%token FOREACH WHILE IF  FOR ELSE TEMPLATE INSTRUCTIONS FUNCTION CONTINUE BREAK
%token RETURN IN ONCE WHEN VAR EOF LBRACE RBRACE LPAREN RPAREN LBRACKET RBRACKET
%token COMMA SEMICOLON COLON DOTDOTDOT DOT EQUALS NOT QUESTION PLUS MINUS TIMES
%token DIVIDE MODULO AND OR VOID SWITCH CASE DEFAULT PLUSEQUALS MINUSEQUALS
%token TIMESEQUALS DIVEQUALS MODEQUALS PLUSPLUS MINUSMINUS AT TRY CATCH THROW
%token FINALLY PROTOTYPE OUTOFRANGENUMBER


%start program
%type <Ast.statement> program

/* resolve shift/reduce conflict for ELSE */
```

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

/* resolve shift/reduce conflict for {} (map) and {} (empty block) */
%nonassoc MAP
%nonassoc EMPTYBLOCK

%right PLUSEQUALS MINUSEQUALS
%right TIMESEQUALS DIVEQUALS MODEQUALS
%right EQUALS
%right COLON
%right QUESTION
%left OR
%left AND
%left COMPOP
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right NOT
%right UMINUS
%right PREFIX_INCDEC
%left POSTFIX_INCDEC
%left ARR_INDEX

%%
program:
    | opt_statements EOF { Program($1) }
;
statements:
    | statement   { [$1] }
    | statement statements  { $1::$2 }
;
opt_statements:
    | statements  { $1 }
    | /*nothing*/ { [] }
;
statement_block:
    | LBRACE statements RBRACE  { StatementBlock($2) }
    | empty_statement_block     { $1 }
;
empty_statement_block:
    | LBRACE RBRACE  { StatementBlock([]) }
;
else_clause:
    | ELSE statement { $2 }
    | %prec LOWER_THAN_ELSE { Noop }
;
statement:
    | IF LPAREN expression RPAREN statement else_clause { If($3,$5,$6,get_env()) }
    | expression SEMICOLON { ExpressionStatement($1, get_env()) }
    | SEMICOLON  { Noop }
    | statement_block { $1 }
    | FOREACH LPAREN ID IN expression RPAREN statement { ForEach($3,$5,$7,get_env()) }
    | WHILE LPAREN expression RPAREN statement { For(Value(Void),$3,Value(Void),$5,get_env()) }
    | CONTINUE SEMICOLON                      { Continue(get_env())}
    | BREAK SEMICOLON                         { Break(get_env())}
    | RETURN opt_expression SEMICOLON         { Return($2,get_env()) }
    | IMPORT STRING SEMICOLON
{ Import(resolve_import($2,$1,get_env()),get_env()) }
    | TEMPLATE ID LBRACE template_specs RBRACE { TemplateDef($2, $4,get_env()) }
    | INSTRUCTIONS FOR ID LPAREN arglist RPAREN LBRACE instruction_specs RBRACE
                                                   { Instructions($3,$5,$8,get_env()) }
    | SWITCH LPAREN expression RPAREN LBRACE switch_statements RBRACE
                                                   { Switch($3,$6, get_env()) }
    | FOR LPAREN opt_expression SEMICOLON
                 opt_expression SEMICOLON
                 opt_expression RPAREN statement { For($3,$5,$7,$9,get_env()) }
    | TRY statement_block CATCH LPAREN ID RPAREN statement_block { TryCatch($2,$5,$7,
get_env()) }
    | TRY statement_block FINALLY statement_block { TryFinally($2,$4,get_env()) }
    | THROW expression SEMICOLON                   { Throw($2, get_env()) }
;
switch_statement:
    | CASE expression COLON                        { Case(Some $2,get_env()) }
    | DEFAULT COLON                                { Case(None,get_env()) }
    | statement                                    { $1 }
;
switch_statements:
    | switch_statement                             { [$1] }
    | switch_statement switch_statements           { $1::$2 }
;
opt_expression:
    | expression                                   { $1 }
    | empty_expression                             { Value(Void) }
;
empty_expression:
    | /*nothing */                                 { Value(Void) }
;
atom_expr:
    | INT                                          { Value(IntegerValue($1)) }
    | REAL                                         { Value(FloatValue($1)) }
    | STRING                                       { Value(StringValue($1)) }
    | BOOLEAN                                      { Value(BooleanValue($1)) }
    | VOID                                         { Value(Void) }
    | LBRACKET expr_list RBRACKET                  { ArrayExpr($2) }
    | LBRACE prop_list RBRACE                      { MapExpr($2) }
    | ID                                           { Id($1) }
    | LPAREN expression RPAREN                     { $2 }
;
member_expr:
    | atom_expr                                    {$1}
    | FUNCTION LPAREN arglist RPAREN statement_block
{ Value(FunctionValue($3,extract_stmt_list($5))) }
    | member_expr  LBRACKET expression RBRACKET    { MemberExpr($1,$3) }
    | member_expr DOT ID                           { MemberExpr($1,Value(StringValue($3))) }
;
call_expr:
    | member_expr LPAREN fexpr_list RPAREN         { FunctionCall($1,$3) }
    | call_expr LPAREN fexpr_list RPAREN           { FunctionCall($1,$3) }
    | call_expr LBRACKET expression RBRACKET       { MemberExpr($1,$3) }
    | call_expr DOT ID                             { MemberExpr($1,Value(StringValue($3))) }
;
lhs_expr:
    | member_expr                                  {$1}
    | call_expr                                    {$1}
;
unary_expr:
    | lhs_expr                                     { $1 }
    | %prec PREFIX_INCDEC PLUSPLUS lhs_expr
{ Assignment($2,BinaryOp($2,Plus,Value(IntegerValue(1)))) }
    | %prec PREFIX_INCDEC MINUSMINUS lhs_expr
{ Assignment($2,BinaryOp($2,Minus,Value(IntegerValue(1)))) }
    | %prec POSTFIX_INCDEC lhs_expr PLUSPLUS  { PostFixSum($1,1) }
    | %prec POSTFIX_INCDEC lhs_expr MINUSMINUS { PostFixSum($1,-1) }
;
op_expr:
    | unary_expr                         {$1}
    | op_expr PLUS op_expr             { BinaryOp($1,Plus,$3) }
    | op_expr MINUS op_expr            { BinaryOp($1,Minus,$3) }
    | op_expr TIMES op_expr            { BinaryOp($1,Times,$3) }
    | op_expr DIVIDE op_expr           { BinaryOp($1,Divide,$3) }
    | op_expr MODULO op_expr           { BinaryOp($1,Modulo,$3) }
    | op_expr COMPOP op_expr           { CompOp($1,$2,$3) }
    | NOT lhs_expr                     { Not($2) }
    | op_expr  AND op_expr             { BinaryOp($1,And,$3) }
    | op_expr OR op_expr               { BinaryOp($1,Or,$3) }
    | %prec UMINUS MINUS op_expr       { BinaryOp(Value(IntegerValue(0)),Minus,$2) }
;
cond_expr:
    | op_expr {$1}
    | expression QUESTION expression COLON expression
                                       { TernaryCond($1,$3,$5) }
;
expression:
    | cond_expr                          {$1}
    | lhs_expr EQUALS expression         { Assignment($1,$3) }
    | lhs_expr EQUALS empty_statement_block  { Assignment($1,MapExpr([])) }
    | VAR lhs_expr EQUALS expression     { Declaration($2,$4) }
    | VAR lhs_expr EQUALS empty_statement_block { Declaration($2,MapExpr([])) }
    | lhs_expr TIMESEQUALS expression    { Assignment($1,(BinaryOp($1,Times,$3))) }
    | lhs_expr MODEQUALS expression      { Assignment($1,(BinaryOp($1,Modulo,$3))) }
    | lhs_expr DIVEQUALS expression      { Assignment($1,(BinaryOp($1,Divide,$3))) }
    | lhs_expr PLUSEQUALS expression     { Assignment($1,(BinaryOp($1,Plus,$3))) }
    | lhs_expr MINUSEQUALS expression    { Assignment($1,(BinaryOp($1,Minus,$3))) }
;
arglist:
    | ID                                 { [$1] }
    | ID DOTDOTDOT                       { ["["^$1] }
    | ID COMMA arglist                   { $1::$3 }
    | /* nothing */                      { [] }
;
expr_list:
    | expression                         { [$1] }
    | empty_statement_block              { [MapExpr([])] }
    | expression COMMA expr_list         { $1::$3 }
    | /*nothing*/                        { [] }
;
fexpr:
    | expression                         { $1 }
    | empty_statement_block              { MapExpr([]) }
    | AT ID                              { UnboundVar($2) }
    | AT ID DOTDOTDOT                    { UnboundVar("["^$2) }
;
fexpr_list:
    | fexpr                              { [$1] }
    | fexpr COMMA fexpr_list             { $1::$3 }
    | /*nothing*/                        { [] }
```

```
;
property:
    | ID COLON expression                  { ($1,$3) }
    | ID COLON empty_statement_block       { ($1,MapExpr([])) }
;
prop_list:
    | property                             { [$1] }
    | property COMMA prop_list             { $1::$3 }
;
template_spec:
    | label TEXT                           { (Some $1,$2 ^ "\n") }
    | TEXT                                 { (None, $1 ^ "\n") }
;
template_specs:
    | template_spec                        { [$1] }
    | template_spec template_specs         { $1::$2 }
;
instruction_spec:
    | label repl_condition COLON replacement_list SEMICOLON { ($1,$2,$4) }
    | label repl_condition SEMICOLON       { ($1,$2,[]) }
;
repl_condition:
    | ONCE                                 { Once }
    | WHEN LPAREN expression RPAREN        { When($3) }
    | FOREACH LPAREN ID IN expression RPAREN  { Loop($3,$5) }
    | FOREACH LPAREN ID IN expression RPAREN WHEN LPAREN expression RPAREN
                                           { CondLoop($9,$3,$5) }
;
replacement:
    | ID EQUALS expression                 { ($1,$3) }
;
replacement_list:
    | /* nothing */                        { [] }
    | replacement                          { [$1] }
    | replacement COMMA replacement_list   { $1::$3 }
;
instruction_specs:
    | instruction_spec                     { [$1] }
    | instruction_spec instruction_specs   { $1::$2 }
;
label:
    | ID                                   { $1 }
    | INT                                  { string_of_int($1) }
;
%%
```

Appendix C – Tests

tests.jtp

```
var main=function(){
    var results=[];

    import 'suites/test_parser.jtp';
    let x=runSuite(testSuite);
    results.push(x);

    import './suites/test_scope.jtp';
    results.push(runSuite(testSuite));

    import '../tests/./suites/test_precedence.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_interpreter.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_lib_builtin.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_lib_string.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_errors.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_expressions.jtp';
    results.push(runSuite(testSuite));

    import 'suites/test_io.jtp';
    results.push(runSuite(testSuite));

    println();
    println('=======');
    println('SUMMARY:');
    println('=======');
    var total=var pass=0;
    foreach(result in results){
        total=total+result.total;
        pass=pass+result.pass;
        println(result.name,': ',result.pass,'/',result.total,'
',status_descr(result.pass,result.total));
    }
    println();
    println('ALL TESTS: ',pass,'/',total,' ',status_descr(pass,total));
};

var runSuite=function(suite){
    var numPass=0;
    var numTests=0;
    println();
    println('Running test suite: ', suite.description);
    foreach(test in suite.tests){
        numTests = numTests + 1;
        print('    -', test.description,' ');
        var result=false;
        try{
             result=test.test();
        }catch(e){
             println(' *exception ',e,' thrown');
             result=false;
        }
        if (typeof(result)!='boolean'){
             println('FAIL');
             println('         *test did not return a boolean value, but a 
',typeof(result));
        }else if (result){
            println('PASS');
            numPass=numPass+1;
        }else println('FAIL');
    }
    println(numPass,'/',numTests,' passed ',suite.description,' 
',status_descr(numPass,numTests));
    return {name: suite.description,pass: numPass,total: numTests};
};

var status_descr=function(x,y){
    if(x==y) return 'PASS';
    else return 'FAIL';
};

main();
```

suites/test_errors.jtp

```
var testSuite={
    description: 'Errors',
    tests:
    [
        {
            description: 'Incompatible types',
            test: function(){
             try{
                    let a=true+"a".length();
                    return false;
                }catch (e){
                    return e == 'EXP-00 incompatible types boolean and 
integer';
                }
            }
        },{
            description: 'Invalid cast',
            test: function(){
             try{
                    for (var i=0;i=i+1;i++);
                    return false;
                }catch (e){
                    return e == 'EXP-01 cannot cast a 1 to a boolean';
                }
            }
        },{
            description: 'Invalid operation',
            test: function(){
             try{
                    let a=true+('a'.length()==1);
                    return false;
                }catch (e){
                    return e == 'EXP-02 invalid operation + for booleans';
                }
            }
        },{
            description: 'Invalid comparaison',
            test: function(){
             try{
                    let a=true>('a'.length()==1);
                    return false;
                }catch (e){
                    return e == 'EXP-03 invalid comparaison > for boolean and
boolean';
                }
            }
        },{
            description: 'Left side is not a map',
            test: function(){
             try{
                    let a=10;
                    println(a.test);
                    return false;
                }catch (e){
                    return e == 'EXP-04 left side of member expression is not
a map or array, but a integer with value 10';
                }
            }
        },{
            description: 'Left side cannot be assigned',
            test: function(){
             try{
                    2=3;
                    return false;
                }catch (e){
                    return e == 'EXP-05 left side of assignment expression
cannot be assigned';
                }
            }
        },{
            description: 'Invalid member',
            test: function(){
             try{
                 let a={};
                 a[true]=1;
                 return false;
              }catch (e){
                 return e == 'EXP-06 member expression did not evaluate to
a string or integer, but to a boolean with value true';
              }
            }
        },{
            description: 'Invalid member',
            test: function(){
             try{
                 let a={};
                 a[true]=1;
                 return false;
              }catch (e){
                 return e == 'EXP-06 member expression did not evaluate to
a string or integer, but to a boolean with value true';
              }
            }
        },{
            description: 'Undefined member',
            test: function(){
             try{
                 let a={};
                 let c=a.b;
                 return false;
              }catch (e){
                 return e == 'EXP-07 member expression b is undefined';
              }
            }
        },{
            description: 'invalid index',
            test: function(){
             try{
                 let a=[1,2,3];
                 let c=a['a'];
                 return false;
              }catch (e){
                 return e == 'EXP-08 invalid array index of type string
with value a';
              }
            }
        },{
            description: 'invalid index',
            test: function(){
             try{
                 let a=[1,2,3];
                 let c=a[10];
                 return false;
              }catch (e){
                 return e == 'EXP-09 array index out of bounds: 10';
              }
            }
        },{
            description: 'type mismatch in assignment',
            test: function(){
             try{
                   let a=[1,2,3];
                   a=10;
                   return false;
                }catch (e){
                   return e == 'EXP-10 type mismatch in assignment of a
declared as array, attempting to assign integer';
                }
            }
        },{
            description: 'too few arguments in function call',
            test: function(){
             try{
                   let a=function(x,y){let b=1;let c=2;};
                   a(1);
                   return false;
                }catch (e){
                   return e == 'EXP-11 wrong number of arguments in function
call, expected 2, got 1';
                }
            }
        },{
            description: 'too many arguments in function call',
            test: function(){
             try{
                   let a=function(x,y){let b=1;let c=2;};
                   a(1,2,3);
                   return false;
                }catch (e){
                   return e == 'EXP-11 wrong number of arguments in function
call, expected 2, got 3';
                }
            }
        },{
            description: 'calling a non-function',
            test: function(){
             try{
                   let a=10;
                   a(1,2,3);
                   return false;
                }catch (e){
                   return e == 'EXP-12 invalid function call on a non-
function variable';
                }
            }
        },{
            description: 'not a collection in foreach',
            test: function(){
             try{
                   let a=10;
```

```
                    foreach(i in a);
                    return false;
                }catch (e){
                    return e == 'EXP-13 expected a collection type for the
second argument of forEach, but got a integer';
                }
            }
        },{
            description: 'default case should be last',
            test: function(){
             try{
                    let a=10;
                    switch(a){
                    default: break;
                    case 10: break;
                    }
                    return false;
                }catch (e){
                    return e == 'STM-00 the default case in a switch statement
should be the last case';
                }
            }
        },{
             description: 'division by zero',
             test: function(){
                    try{
                          println('a'.length()/0);
                          return false;
                    }catch(e){
                          return e=='EXP-14 Division by zero';
                    }
             }
        }
      ]
};
```

suites/test_expressions.jtp

```
var testSuite={
    description: 'Expressions',
    tests:
    [
        {
            description: 'floating point operations',
            test: function(){
             return 1.2+2.5==3.7 && 1.2*1.2==1.44 && 3.5-1.2==2.3 &&
0.5/.25==2
              && 1.2/0==Float.infinity && -1.2/0==Float.negativeInfinity;
            }
        },{
            description: 'floating point modulo throws exception',
            test: function(){
             try{
                    let a=('a'.length/2.1)%2.1;
                    return false;
             }catch(e){
                    return true;
             }
            }
        },{
            description: 'integer operations',
            test: function(){
             return 2+3==5 && 2-3==-1 && 2*3==6 && 45/4==11 && 45%4==1;
            }
        },{
            description: 'invalid integer operations',
            test: function(){
             try{
                    let a='a'.length() && 1;
                    return false;
             }catch(e){
                    return true;
             }
            }
        },{
            description: 'string operations',
            test: function(){
             return 'a'+'b'=='ab' && 'a'+1=='a1' && 1+'a'=='1a';
            }
        },{
             description: 'mixed floating point/integer operations',
             test: function(){
                           return 1+1.2==2.2 && 1.2+1==2.2;
             }
        },{
             description: 'not expression on non boolean throws exception',
             test: function(){
                    try{
                           let a=! 'a'.length();
                           return false;
                    }catch(e){
                           return true;
                    }
             }
        }
      ]
};
```

suites/test_interpreter.jtp

```
var testSuite={
    description: 'Interpreter',
    tests:
    [
        {
            description: 'Post increment test',
            test: function(){
                var a=1;
                var b=a++;
                return b==1 && a==2;
            }
        },{
            description: 'Post decrement test',
            test: function(){
                var a=1;
                var b=a--;
                return b==1 && a==0;
            }
        },{
            description: 'Pre increment test',
            test: function(){
                var a=1;
                var b=++a;
                return b==2 && a==2;
            }
        },{
            description: 'Pre decrement test',
            test: function(){
                var a=3;
                var b=--a;
                return b==2 && a==2;
            }
        },{
            description: 'op= tests',
            test: function(){
                var a=var b=var c=var d=var e=10;
                return (a+=1)==11 && (b-=1)==9 && (c*=2)==20 && (d/=5)==2 &&
(e%=3)==1 &&
                       a==11 && b==9 && c==20 && d==2 && e==1;
            }
        },{
            description: 'ternary ?: tests',
            test: function(){
                var a=10;
                return (a==10?true:false) && a!=10?false:true;
            }
        },{
            description: 'ternary expression with non boolean throws
exception',
            test: function(){
             try{
                   var a=10;
                   let b=('a'.length()?true:false) && a!=10?false:true;
                   return false;
               }catch(e){
                   return true;
               }
            }
        },{
            description: 'for loop test',
            test: function(){
```

```
            var a=0;
            for (var i=0;i<100;++i){
                ++a;
            }
            return a==100;
        }
    },{
        description: 'for loop test with continue',
        test: function(){
            var a=0;
            for (var i=0;i<100;++i){
                if (i%2==0) continue;
                ++a;
            }
            return a==50;
        }
    },{
        description: 'for loop test with break',
        test: function(){
            var a=0;
            for (var i=0;i<20;++i){
                if (i>=10) break;
                ++a;
            }
            return a==10;
        }
    },{
        description: 'while loop test',
        test: function(){
            var a=var i=0;
            while (i<100){
                ++a;
                ++i;
            }
            return a==100;
        }
    },{
        description: 'if test',
        test: function(){
         if ('a'.length()==1)
                return true;
         return false;
        }
    },{
        description: 'else test',
        test: function(){
         if ('a'.length()==2)
                return false;
         else
                return true;
        }
    },{
        description: 'if with non boolean throws exception',
        test: function(){
         try{
                if ('a'.length())
                       return true;
                return false;
         }catch(e){
                return true;
         }
        }
    },{
        description: 'while loop test with continue',
        test: function(){
                var a=var i=0;
            while (i<100){
                ++i;
                 if (i%2==0) continue;
                 ++a;
            }
            return a==50;
        }
    },{
        description: 'while loop test with break',
        test: function(){
            var a=var i=0;
            while (i<100){
                ++i;
                 ++a;
                 if (i>=10) break;
            }
            return a==10;
        }
    },{
        description: 'try finally with return in try block',
        test: function(){
                var a=0;
            var f=function(){
                try{
                       return 1;
                }finally{
                       a=2;
                }
            };
            return f()==1 && a==2;
        }
    },{
        description: 'try finally with return in finally block',
        test: function(){
                var a=0;
            var f=function(){
                try{
                       a=2;
                       return a;
                }finally{
                       return 3;
                }
```

```
		};
		return f()==3 && a==2;			}
	},{
	    description: 'try catch with user exception',
	    test: function(){
			try{
				throw 'test';
				return false;
			}catch(e){
				return e=='test';
			}
	    }
	},{
	     description: 'try catch with system exception',
	     test: function(){
			try{
				var a=[1,2];
				var b=a[10];
				return false;
			}catch(e){
				return true;
			}
	     }
	},{ //regression bug
	     description: 'try catch passes return through',
	     test: function(){
			return function(){
				try{
					return true;
				}catch(e){
					return false;
				}
			}();
	     }
	},{
	     description: 'try catch passes break through',
	     test: function(){
			for (var i=0;i<10;i++){
				try{
					break;
				}catch(e){
					return false;
				}
			}
			return true;
	     }
	},{
	     description: 'try catch passes continue through',
	     test: function(){
			for (var i=0;i<10;i++){
				try{
					continue;
				}catch(e){
					return false;
```

```
                        }
                }
                return true;
            }
        },{
            description: 'try finally catch, all steps reached',
            test: function(){
                var a=var b=var c=var d=0;
                try{
                        a=1;
                        try{
                                b=1;
                                var x=a[10];
                        }finally{
                                c=1;
                        }
                }catch(e){
                        d=1;
                }
                return a==1 && b==1 && c==1 && d==1;
            }
        },{
            description: 'try finally catch, catch after finally',
            test: function(){
                var a=0;
                try{
                        try{
                                var x=a[10];
                        }finally{
                                a=10;
                        }
                }catch(e){
                        a=1;
                }
                return a==1;
            }
        },{
            description: 'compare array for equality',
            test: function(){
                var a=[1,2,3,'4','5'];
                var b=[1,2,3,'4','5'];
                return a==b;
            }
        },{
            description: 'prototype construction/usage',
            test: function(){
                var Foo={prototype:{getBar: function(){return 'bar';}}};
                var x={prototype:Foo};
                return x.getBar()=='bar';
           }
        },{
            description: 'function chaining',
            test: function(){
                var a='1.343.34.2';
```

```
                        return a.replaceAll('.','').replaceAll('34','x')=='1x3x2';
                }
        },{
                description: 'static scoping',
                test: function(){
                        let x = 0;
                        let f=function() { return x; };
                        let g=function () { let x = 1; return f(); };
                        return g()==0;
                }
        },{
                description: 'compare integers',
                test: function(){
                        return 10==10 && 10!=6 && 10>6 && 10<12 && 10>=6 && 10<=12
&&
                                10>=10 && 10<=10;
                }
        },{
                description: 'compare floats',
                test: function(){
                        return 10.==10. && 10.!=6. && 10.>6. && 10.<12. && 10.>=6.
&& 10.<=12. &&
                                10.>=10. && 10.<=10.;
                }
        },{
                description: 'compare strings',
                test: function(){
                        return 'a'=='a' && 'a'!='b' && 'a'<'b' && 'a'<='b' &&
'a'<='a'
                                && 'b'>'a' && 'bb'>='ba' && 'bb'>='bb';
                }
        },{
                description: 'compare booleans',
                test: function(){
                        return true==true && false==false && true!=false;
                }
        },{
                description: 'compare Void',
                test: function(){
                        let a=Void;
                        let b=Void;
                        let c=1;
                        return a==b && b!=c && a!=c && !(a!=b);
                }
        },{
                description: 'compare functions',
                test: function(){
                        let a=function(a,b){return a+b;};
                        let b=[1,a];
                        let c=function(a,b){return a-b;};
                        return a==b[1] && b[1]!=c && a!=c && !(a!=b[1]);
                }
        },{
                description: 'compare library functions',
			test: function(){
				let a=String.prototype.length;
				let b=[1,a];
				let c=String.prototype.substr;
				return a==b[1] && b[1]!=c && a!=c && !(a!=b[1]);
			}
		},{
			description: 'compare maps',
			test: function(){
				let a={a:1,b:'2',c:3};
				let b={a:1,b:'2',c:3};
				let c={a:1,b:'2',c:4};
				let d={a:1,b:'2',x:3};
				return a==b && b!=c && a!=c && !(a!=b) && !(a==d);
			}
		},{
			description: 'compare arrays',
			test: function(){
				let a=[1,'a',[1,2,['a','b',{a:true,b:34.5}]]];
				let b=[1,'a',[1,2,['a','b',{a:true,b:34.5}]]];
				let c=[1,'a',[1,2,['a','b',{a:true,b:34.6}]]];
				return a==b && b!=c && a!=c && !(a!=b);
			}
		},{
			description: 'compare integer and float',
			test: function(){
				return 10==10. && 10!=6. && 10>6. && 10<12. && 10>=6. &&
10<=12. &&	10>=10. && 10<=10.
				 && 10.==10 && 10.!=6 && 10.>6 && 10.<12 && 10.>=6 &&
10.<=12 && 10.>=10 && 10.<=10;
			}
		},{
			description: 'compare integer and string',
			test: function(){
				return '10'==10 && '10'!=6 && '20'>10 && '10'<12 &&
'20'>=11 && '10'<=12 && '10'>=10 && '10'<=10
				&& 10=='10' && 10!='6' && 20>'16' && 10<'12' && 20>='16'
&& 10<='12' && 10>='10' && 10<='10';
			}
		},{
			description: 'compare float and string',
			test: function(){
				return '10.'==10. && '10.'!=6. && '20.'>10. && '10.'<12.
&& '20.'>=11. && '10.'<=12. && '10.'>=10. && '10.'<=10.
				&& 10.=='10.' && 10.!='6.' && 20.>'16.' && 10.<'12.' &&
20.>='16.' && 10.<='12.' && 10.>='10.' && 10.<='10.';
			}
		},{
			description: 'mismatched type compare',
			test: function(){
				return !(true==1) && !(true==1.) && true!=1. && !
(true=='1') && true!='1'
						&& 1!=true && 1.!=true && '1'!=true
```

```
                          && String.prototype.length!=true && function()
{return 1;}!=true
                          && {a:1}!=true && [1,2]!=true;
             }
        },{
             description: 'switch()',
             test: function(){
                   let a=10;
                   switch(a){
                   case 1: throw 'error';
                   case 10: a=2;break;
                   default: throw 'error';
                   }
                   return a==2;
             }
        },{
             description: 'switch() without match',
             test: function(){
                   let a=11;
                   switch(a){
                   case 1: throw 'error';
                   case 10: a=2;break;
                   case 12: throw 'error';
                   }
                   return a==11;
             }
        },{
             description: 'switch() default case',
             test: function(){
                   let a=11;
                   switch(a){
                   case 1:
                   case 10: throw 'error';
                   default: a=2;
                   }
                   return a==2;
             }
        },{
             description: 'switch() fallthrough',
             test: function(){
                   let a=10;
                   switch(a){
                   case 1:
                   case 10: a=3;
                   default: a=2;
                   }
                   return a==2;
             }
        },{
             description: 'switch() too many default throws exception',
             test: function(){
                   let a=10;
                   try{
                          switch(a){
```

```
                        case 1:
                        case 10: a=3;
                        default: a=2;
                        default: a=2;
                        }
                        return false;
                }catch(e){
                        return true;
                }
        }
    },{
        description: 'switch() default not last throws exception',
        test: function(){
                let a=10;
                try{
                        switch(a){
                        case 1:
                        case 10: a=3;
                        default: a=2;
                        case 20: a=2;
                        }
                        return false;
                }catch(e){
                        return true;
                }
        }
    },{
        description: 'foreach on array',
        test: function(){
                let sum=0;
                let arr=[1,2,3];
                foreach (n in arr)
                        sum+=n;
                return sum==6;
        }
    },{
        description: 'foreach on map',
        test: function(){
                let sum='';
                let map={a:1,b:2,c:3};
                foreach (n in map){
                        sum+=n;
                }
                return sum=='abc' || sum=='acb' || sum=='bac' ||
sum=='bca' || sum=='cab' || sum=='cba';
        }
    },{
        description: 'foreach with break',
        test: function(){
                let sum=0;
                let arr=[1,2,3];
                foreach (n in arr){
                        if(n==3)
                                break;
                        sum+=n;
                }
                return sum==3;
        }
},{
        description: 'foreach with continue',
        test: function(){
                let sum=0;
                let arr=[1,2,3];
                foreach (n in arr){
                        sum+=n;
                        continue;
                        sum+=10;
                }
                return sum==6;
        }
},{
        description: 'unbound vars/partial application',
        test: function(){
                let add=function(x,y){return x+y;};
                let add1=add(@v,1);
                let add2=add(2,@v);
                return add1(10)==11 && add2(23)==25;
        }
},{
        description: 'unbound vars/partial application with varargs',
        test: function(){
                let foo=function(x,a...){
                        return typeof(a)=='array' && x==1
                        && a[0]==2 && a[1]==3 && a[2]==4;
                };
                let bar=foo(1,2,@x...);
                return bar(3,4);
        }
},{
        description: 'vararg function',
        test: function(){
                let foo=function(x,a...){
                        return typeof(a)=='array' && x==1
                        && a[0]==2 && a[1]==3 && a[2]==4;
                };
                return foo(1,2,3,4);
        }
},{
        description: 'closures',
        test: function(){
                let f={};
                for (var i=0;i<10;++i){
                        let f[i]=function(n){
                                return function(){
                                        return n;
                                };
                        }(i);
                }
```

```
                        for (var i=0;i<10;++i){
                                if (f[i]()!=i)
                                        return false;
                        }
                        return true;
                }
        },{
                description: 'templates',
                test: function(){
                        template htmlTable{
                                                        #<table>
                                                        #<tr>
                                        header          #<th>columnLabel</th>
                                                        #</tr>
                                        row     #<tr>
                                        cell    #<td>cellData</td>
                                        row     #</tr>
                                                        #</table>
                                }
                                instructions for htmlTable(dataMap){
                                        header foreach(label in dataMap.labels):
columnLabel=label;
                                        row foreach(dataArray in dataMap.data): ;
                                        cell foreach(element in dataArray):
cellData=element;
                                }
                                let people={labels: ['Name','Age'], data: [['John',
42], ['Mary',38]] };
                                let res=htmlTable(people);
                                let ref='<table>\r\n<tr>\r\n<th>Name</th>\r
\n<th>Age</th>\r\n</tr>\r\n<tr>\r\n<td>John</td>\r\n'
                          + '<td>42</td>\r\n</tr>\r\n<tr>\r\n<td>Mary</td>\r
\n<td>38</td>\r\n</tr>\r\n</table>\r\n';
                        return ref==res;
                }
        },{
                description: 'bad template nesting',
                test: function(){
                                return 2==System.command('../_build/jtemplate.native
suites/bad_samples/bad_nesting1.jtp');
                }
        },{
                description: 'unused template',
                test: function(){
                                template htmlTable{
                                                        #<table>
                                                        #<tr>
                                        header          #<th>columnLabel</th>
                                                        #</tr>
                                        row     #<tr>
                                        cell    #<td>cellData</td>
                                        row     #</tr>
                                                        #</table>
                                }
                        return true;
                    }
        }
    ]
};
```

suites/test_io.jtp

```
var testSuite={
    description: 'I/O Library',
    tests:
    [
        {
            description: 'Create directory',
            test: function(){
             return Directory.create('testdir') &&
                    Directory.exists('testdir');
            }
        },       {
            description: 'Delete directory',
            test: function(){
             return Directory.delete('testdir') &&
                    !Directory.exists('testdir');
            }
        },{
             description: 'List directory',
             test: function(){
                    Directory.create('testdir2');
                    Directory.create('testdir2/subdir');
                    let a=Directory.list('testdir2');
                    let result=a.length()==3 && a[2]=='subdir';
                    Directory.delete('testdir2/subdir');
                    Directory.delete('testdir2');
                    return result;
             }
        },{
             description: 'Rename',
             test: function(){
                    Directory.create('testdir2');
                    File.rename('testdir2','testdir3');
                    let result=Directory.exists('testdir3');
                    Directory.delete('testdir3');
                    return result;
             }
        },{
             description: 'File creation',
             test: function(){
                    let handle='1';
                    File.openForWriting(handle,'testfile');
                    File.write(handle,'Hello ');
                    File.writeln(handle,'world!');
                    File.close(handle);
```

```
                        return File.exists('testfile');
                }
        },{
                description: 'File reading',
                test: function(){
                        let handle='1';
                        File.openForReading(handle,'testfile');
                        let s=File.readln(handle);
                        File.close(handle);
                        return s=='Hello world!';
                }
        },{
                description: 'File eof',
                test: function(){
                        let handle='1';
                        File.openForWriting(handle,'testfile');
                        for (var i=0;i<10;++i)
                                File.writeln(handle,'test');
                        File.close(handle);
                        File.openForReading(handle,'testfile');
                        let numlines=0;
                        while(!File.eof(handle)){
                                File.readln(handle);
                                ++numlines;
                        }
                        File.close(handle);
                        return numlines==10;
                }
        },{
                description: 'File delete',
                test: function(){
                        File.delete('testfile');
                        return !File.exists('testfile');
                }
        }
       ]
};
```

suites/test_lib_builtin.jtp

```
var testSuite={
        description: 'System Library',
        tests: [
                {
                description: 'System.command',
                test: function(){
                                var filename='testdir.txt';
                                File.delete(filename);
                        var result=
                                !File.exists(filename) &&
                                System.command('ls>'+filename)==0 &&
                                File.exists(filename);
                        File.delete('testdir.txt');
                        return result;
```

```
			}
		},{
			description: 'array push',
			test: function(){
				let a=[1,2,'3'];
				a.push('4');
				a.push(5);
				return a==[1,2,'3','4',5];
			}
		},{
			description: 'array push on non array throws exception',
			test: function(){
				try{
					let a=12;
					Array.prototype.push.apply(a,'4');
					return false;
				}catch(e){
					return true;
				}
			}
		},{
			description: 'array pop',
			test: function(){
				let arr=[1,2,'3','4',5];
				let a=arr.pop();
				let b=arr.pop();
				return arr==[1,2,'3'] && a==5 && b=='4';
			}
		},{
			description: 'array pop on empty array throws exception',
			test: function(){
				try{
					let arr=[];
					let a=arr.pop();
					let b=arr.pop();
					return false;
				}catch(e){
					return true;
				}
			}
		},{
			description: 'array pop on non array throws exception',
			test: function(){
				try{
					let arr=12;
					Array.prototype.pop.apply(arr);
					return false;
				}catch(e){
					return true;
				}
			}
		},{
			description: 'array length',
			test: function(){
```

```
                let arr1=[1,2,'3','4',5];
                let arr2=[];
                return arr1.length()==5 &&
                       arr2.length()==0 &&
                       [1,2,'3','4',5].length()==5;
            }
        },{
            description: 'array length on non array throws exception',
            test: function(){
                try{
                    let arr1=12;
                    Array.prototype.length.apply(arr1);
                    return false;
                }catch(e){
                    return true;
                }
            }
        },{
            description: 'integer parse',
            test: function(){
                return '15'.parseInt()==15 &&
                         '15.4'.parseInt()==Void &&
                       '1.5a'.parseInt()==Void;
            }
        },{
            description: 'float parse',
            test: function(){
                return '1.5'.parseFloat()==1.5 &&
                       '1.5a'.parseFloat()==Void;
            }
        },{
            description: 'float round',
            test: function(){
                return 12.34.round()==12 &&
                       12.64.round()==13;
            }
        },{
            description: 'float round on non float throws exception',
            test: function(){
                try{
                    Float.prototype.round.apply(1);
                    return false;
                }catch(e){
                    return true;
                }
            }
        },{
            description: 'map keys',
            test: function(){
                let m={a:1,b:'123',d:true};
                let a=m.keys();
                return a==['a','b','d'] || a==['a','d','b'] ||
                         a==['b','a','d'] || a==['b','d','a'] ||
                       a==['d','a','b'] || a==['d','b','a'];
            }
        },{
            description: 'map keys on non map throws exception',
            test: function(){
                try{
                    let m=[12,123,'a'];
                    Map.prototype.keys.apply(m);
                    return false;
                }catch(e){
                    return true;
                }
            }
        },{
            description: 'map contains',
            test: function(){
                let m={a:1,b:'123',d:true};
                let a=m.keys();
                return m.contains('a') && !m.contains('c');
            }
        },{
            description: 'map contains on non map throws exception',
            test: function(){
                try{
                    let m=[12,123,'a'];
                    Map.prototype.contains.apply(m,'a');
                    return false;
                }catch(e){
                    return true;
                }
            }
        },{
            description: 'map remove',
            test: function(){
                let m={a:1,b:'123',d:true};
                m.remove('a');
                return m=={b:'123',d:true};
            }
        },{
            description: 'map remove on non map throws exception',
            test: function(){
                try{
                    let m=[12,123,'a'];
                    Map.prototype.remove.apply(m);
                    return false;
                }catch(e){
                    return true;
                }
            }
        },{
            description: 'Date.now()',
            test: function(){
                let d=Date.now();
                foreach(f in
['second','minute','hour','dayOfMonth','month','year','dayOfWeek',
                                'dayOfYear','gmtOffset'])
                        if (!d.contains(f) || typeof(d[f])!='integer')
                          return false;
                    return d.contains('dst') && typeof(d.dst)=='boolean';
                }
        },{
                description: 'random()',
                test: function(){
                        let i=0;
                        let sum=0;
                        while(i<1000){
                                let r=Integer.random(101);
                                if (r!=50){
                                        sum+=r;
                                        ++i;
                                }
                        }
                        let avg=sum/i;
                        return avg>=46 && avg<=54;
                }
        },{
                description: 'typeof()',
                test: function(){
                        return typeof(1)=='integer' && typeof(1.2)=='float' &&
typeof('a')=='string' &&
                          typeof(true)=='boolean' && typeof(Void)=='void' &&
typeof(function(){})=='function' &&
                          typeof([])=='array' && typeof({})=='map' &&
                          typeof(String.prototype.length)=='function';
                }
        },{
                description: 'apply()',
                test: function(){
                        let foo=function(){return this*2;};
                        return String.prototype.length.apply('233')==3 &&
                                foo.apply(2)==4 && function(){return
this*2;}.apply(3)==6;
                }
        },{
                description: 'apply() on non function throws exception',
                test: function(){
                        try{
                                let a=1.2;
                                Function.prototype.apply(a,12);
                                return false;
                        }catch(e){
                                return true;
                        }
                }
        },{
                description: 'debug dump symbol table',
                test: function(){
                        Debug.dumpSymbolTable();
                        return true;
                }
            },{
                description: 'debug dump stack trace',
                test: function(){
                      Debug.dumpStackTrace();
                      return true;
                }
            }
        ]
};
```

suites/test_lib_string.jtp

```
var testSuite={
    description: 'String library tests',
    tests: [
            {
             description: 'toUppercase()',
             test: function(){
                   var a='abcDef'; return a.toUppercase()=='ABCDEF';
             }
            },{
             description: 'toLowercase()',
             test: function(){
                         var a='abcDef'; return a.toLowercase()=='abcdef';
             }
            },{
             description: 'toFirstUpper()',
             test: function(){
                         var a='abcDef'; return a.toFirstUpper()=='AbcDef';
             }
            },{
             description: 'toFirstLower()',
             test: function(){
                         var a='AbcDef'; return a.toFirstLower()=='abcDef';
             }
            },{
             description: 'length()',
             test: function(){
                         var a='abcDef'; var b='';
                         return a.length()==6 && b.length()==0;
             }
            },{
             description: 'charAt()',
             test: function(){
                         var a='AbcDef'; return a.charAt(3)=='D';
             }
            },{
             description: 'charAt() with out of range index throws library
error',
             test: function(){
                   try{
                         var a='AbcDef';
                         a.charAt(11);
```

```
                        return false;
                }catch (e){
                        return true;
                }
        }
       },{
        description: 'charAt() with non integer index throws library
error',
        test: function(){
                try{
                        var a='AbcDef';
                        a.charAt(true);
                        return false;
                }catch (e){
                        return true;
                }
        }
       },{
        description: 'indexOf()',
        test: function(){
                        var a='AbcDef';
                        return a.indexOf('X')==-1 &&
                        a.indexOf('cD')==2;
        }
       },{
        description: 'substr()',
        test: function(){
                        var a='AbcDef'; return a.substr(2,3)=='cDe';
        }
       },{
        description: 'substr() with invalid type start throws exception',
        test: function(){
                        try{
                                'AbcDef'.substr('a',3);
                                return false;
                        }catch(e){
                                return true;
                        }
        }
       },{
        description: 'substr() with invalid type length throws
exception',
        test: function(){
                        try{
                                'AbcDef'.substr(3,'a');
                                return false;
                        }catch(e){
                                return true;
                        }
        }
       },{
        description: 'substr() with out of range start/length throws
exception',
        test: function(){
```

```
                    try{
                          'AbcDef'.substr(10,3);
                          return false;
                    }catch(e){
                          return true;
                    }
              }
             },{
              description: 'startsWith()',
              test: function(){
                          var a='AbcDef';
                          return a.startsWith('Abc') && !a.startsWith('abc');
              }
             },{
              description: 'endsWith()',
              test: function(){
                          var a='AbcDef';
                          return a.endsWith('ef') && !a.endsWith('def');
              }
             },{
              description: 'replaceAll()',
              test: function(){
                          var a='1.343.34.2';
                          return a.replaceAll('.','')=='1343342';
              }
             },{
              description: 'split()',
              test: function(){
                          var a='1.343.34.2';
                          return a.split('.')==['1','343','34','2'];
              }
             },{
              description: 'mreplace()',
              test: function(){
                    return 'foo bar foo bar'.mreplace(['foo','bar'],
['bar','foo'])=='bar foo bar foo';
              }
             },{
              description: 'non-string this throws exception',
              test: function(){
                    try{
                          String.prototype.length.apply(123);
                          return false;
                    }catch(e){
                          return true;
                    }
              }
             },{
              description: 'mreplace() with different size arrays throws
exception',
              test: function(){
                    try{
                          'foo bar foo bar'.mreplace(['foo'],
['bar','foo'])=='bar foo bar foo';
```

```
                        return false;
                }catch(e){
                        return true;
                }
        }
      }
    ]};
```

suites/test_parser.jtp

```
/*
        regresstion tests for parser of some constructs that have failed with
new parser
 */

var testSuite = {
        description : 'Parser test',
        tests : [ {
                description : 'empty block {}',
                test : function() {
                        var a = function() {
                        };
                        for ( var i = 0; i < 10; ++i) {
                        }
                        while (false) {
                        }
                        if (true) {
                        } else {
                        }
                        return true;
                }
        }, {
                description : 'empty map {}',
                test : function() {
                        var a = {};
                        a={};
                        function(x,y){}(1,{});
                        var b={a:{}};
                        return true;
                }
        }, {
                description : 'empty statement ;',
                test : function() {
                        ;
                        ;
                        ;
                        ;
                        ;
                        for ( var i = 0; i < 10; ++i)
                                ;
                        while (false)
                                ;
                        if (true)
                                ;
```

```
                    else
                          ;
                    return true;
                }
        },{
                description: 'precendence of ?: expressions',
                test: function(){
                        let gcd = function(x, y) {
                                return x == y ? x : x < y ? gcd(x, y - x) : gcd(x -
y, x);
                                // will get casting error compare boolean to int if
broken
                        };
                        return gcd(24,6)==6;
                }
        },{
                description: '!(grouped expression)', //fixed in 114
                test: function(){
                        let a=!(true==false);
                        return true;
                }
        },{
                description: 'unterminated comment',
                test: function(){
                                return 2==System.command('../_build/jtemplate.native
suites/bad_samples/bad_unterminated_comment.jtp');
                }
         }

        ]
};
```

suites/test_precedence.jtp

```
var testSuite={
    description: 'Precedence tests',
    tests:
    [
        {
            description: 'Check that if then if then else evaluates
correctly',
            test: function(){
                var a=1;
                if (a==1)
                    if (a==3)
                        a=20;
                    else
                        a=10; // would not run if this else was associated
                                // with first if!
                return a==10;
            }
        },{
            description: 'Check that chained expr?expr:expr?expr:expr
evaluates correctly',
            test: function(){
                var b=2;
                var a=b==3?1:b==2?10:1;
                return a==10;
            }
        },{
             description: 'Operator precedence tests',
             test: function(){
                    return 10+2*30==70 && 10-5-5==0 && 2*30+10==70;
            }
        }
    ]
};
```

suites/test_scope.jtp

```
var testSuite = {
       description : 'Scope tests',
       tests : [ {
              description : 'redeclaration in nested scope',
              test : function() {
                     var a = 1;
                     {
                            var a = 2;
                     }
                     return a == 1;
              }
       }, {
              description : 'assignment in nested scope',
              test : function() {
                     var a = 1;
                     {
                            a = 2;
                     }
                     return a == 2;
              }
       }

       ]
};
```

suites/bad_samples/bad_nesting.jtp

```
/*
Unused template
*/
template htmlTable{
                     #<table>
                     #<tr>
       header        #<th>columnLabel</th>
                     #</tr>
       row    #<tr>
       cell   #<td>cellData</td>
       row    #</tr>
       cell   #</table>
}
```

suites/bad_samples/bad_unterminated_comment.jtp

```
/*
This is an unterminated comment

println('test');
```

Appendix D – Benchmark Source Code

GCD

Jtemplate

```
var gcd=function(x,y){
	return x<y ? gcd(x,y-x) : x>y ? gcd(x-y,x) : x;
};

var a=0;
for (var i=0;i<10000;i=i+1)
	a=gcd(28388383,100101);
println(a);
```

JavaScript

```
def gcd(a,b)
    return a<b ? gcd(a,b-a) : a > b ? gcd(a-b,b) : a
end

a=0
for i in 1..10000 do
    a=gcd(28388383,100101)
end

print(a)
```

Python

```
def gcd(a,b):
	if a<b :
			return gcd(a,b-a)
	else:
		if a > b:
			return gcd(a-b,b)
		else :
			return a

a=0

for i in range(1,10000):
	a=gcd(28388383,100101)

print(a)
```

Ruby

```
def gcd(a,b)
    return a<b ? gcd(a,b-a) : a > b ? gcd(a-b,b) : a
end

a=0
```

```
for i in 1..10000 do
    a=gcd(28388383,100101)
end

print(a)
```

PHP

```
<?php
function gcd($x,$y){
      return $x<$y ? gcd($x,$y-$x) :( $x>$y ? gcd($x-$y,$x) : $x);
}

$a=0;
for ($i=0;$i<10000;$i+=1){
      $a=gcd(28388383,100101);
}
print_r ($a);
?>
```

Fibonacci

Jtemplate

```
var fib = function(n) {
      return n >1?  fib(n - 1) + fib(n - 2):n;
};
println(fib(32));
```

JavaScript

```
var fib = function(n) {
      return n >1?  fib(n - 1) + fib(n - 2): n;
};
print(fib(32));
```

Python

```
def fib(n):
      if n > 1:
            return fib(n-1) + fib(n-2)
      else:
            return 1

print(fib(32))
```

Ruby

```
def fib(n)
 return n >1 ? fib(n-1) + fib(n-2) : n
end
```

```
print(fib(32))
```

PHP

```
<?php
function fib($n)
{
        return $n >1 ?  fib($n - 1) + fib($n - 2) : $n;
}

print_r (fib(32));
?>
```

Mandelbrot

source code from http://www.timestretch.com/FractalBenchmark.html

Jtemplate

```
let mandelbrot=function(x, y) {
        var cr = y - 0.5;
        var ci = x;
        var zi = 0.0;
        var zr = 0.0;
        var i = 0;
        var BAILOUT = 16;
        var MAX_ITERATIONS = 1000;

        while(true) {
                i++;
                var temp = zr * zi;
                var zr2 = zr * zr;
                var zi2 = zi * zi;
                zr = zr2 - zi2 + cr;
                zi = temp + temp + ci;
                if (zi2 + zr2 > BAILOUT) {
                        return i;
                }
                if (i > MAX_ITERATIONS) {
                        return 0;
                }
        }
};

let mandelbrot_run =function () {
        var output = '';

        for (var y = -39; y < 39; y++) {
                println(output);
                output = '';
                for (var x = -39; x < 39; x++) {
                        var i = mandelbrot(x/40.0, y/40.0);
                        if (i==0) {
                                output += '*';
                        }
                        else {
                                output += ' ';
                        }
                }
        }
        println(output);
        return false;
};

mandelbrot_run();
```

JavaScript

```
function mandelbrot(x, y) {
        var cr = y - 0.5;
        var ci = x;
        var zi = 0.0;
        var zr = 0.0;
        var i = 0;
        var BAILOUT = 16;
        var MAX_ITERATIONS = 1000;

        while(1) {
                i++;
                var temp = zr * zi;
                var zr2 = zr * zr;
                var zi2 = zi * zi;
                zr = zr2 - zi2 + cr;
                zi = temp + temp + ci;
                if (zi2 + zr2 > BAILOUT) {
                        return i;
                }
                if (i > MAX_ITERATIONS) {
                        return 0;
                }
        }
}

function mandelbrot_run() {
        var x; var y;
        output = "";

        for (y = -39; y < 39; y++) {
                print(output);
                output = "";
                for (x = -39; x < 39; x++) {
                        var i = mandelbrot(x/40.0, y/40.0);
                        if (i==0) {
                                output += "*";
                        }
                        else {
                                output += " ";
```

```
                    }
             }
      }
      print(output);
      return false;
}

mandelbrot_run();
```

Python

```
#!/usr/local/bin/python
# by Daniel Rosengren

import sys, time
stdout = sys.stdout

BAILOUT = 16
MAX_ITERATIONS = 1000

class Iterator:
  def __init__(self):
    print 'Rendering...'
    for y in range(-39, 39):
      stdout.write('\n')
      for x in range(-39, 39):
        i = self.mandelbrot(x/40.0, y/40.0)

        if i == 0:
          stdout.write('*')
        else:
          stdout.write(' ')

  def mandelbrot(self, x, y):
    cr = y - 0.5
    ci = x
    zi = 0.0
    zr = 0.0
    i = 0

    while True:
      i += 1
      temp = zr * zi
      zr2 = zr * zr
      zi2 = zi * zi
      zr = zr2 - zi2 + cr
      zi = temp + temp + ci

      if zi2 + zr2 > BAILOUT:
        return i
      if i > MAX_ITERATIONS:
        return 0

t = time.time()
Iterator()
print '\nPython Elapsed %.02f' % (time.time() - t)
```

Ruby

```
#!/usr/local/bin/ruby

BAILOUT = 16
MAX_ITERATIONS = 1000

class Mandelbrot

    def initialize
            puts "Rendering"
            for y in -39...39 do
                    puts
                    for x in -39...39 do
                            i = iterate(x/40.0,y/40.0)
                            if (i == 0)
                                    print "*"
                            else
                                    print " "
                            end
                    end
            end
    end

    def iterate(x,y)
            cr = y-0.5
            ci = x
            zi = 0.0
            zr = 0.0
            i = 0

            while(1)
                    i += 1
                    temp = zr * zi
                    zr2 = zr * zr
                    zi2 = zi * zi
                    zr = zr2 - zi2 + cr
                    zi = temp + temp + ci
                    return i if (zi2 + zr2 > BAILOUT)
                    return 0 if (i > MAX_ITERATIONS)
            end

    end
```

```
end

time = Time.now
Mandelbrot.new
puts
puts "Ruby Elapsed %f" % (Time.now - time)
```

PHP

```
<?php
define("BAILOUT",16);
define("MAX_ITERATIONS",1000);

class Mandelbrot
{
	function Mandelbrot()
	{
		$d1 = microtime(1);
		for ($y = -39; $y < 39; $y++) {
			echo("\n");
			for ($x = -39; $x < 39; $x++) {
				if ($this->iterate($x/40.0,$y/40.0) == 0)
					echo("*");
				else
					echo(" ");
			}
		}
		$d2 = microtime(1);
		$diff = $d2 - $d1;
		printf("\nPHP Elapsed %0.2f", $diff);
	}

	function iterate($x,$y)
	{
		$cr = $y-0.5;
		$ci = $x;
		$zi = 0.0;
		$zr = 0.0;
		$i = 0;
		while (true) {
			$i++;
			$temp = $zr * $zi;
			$zr2 = $zr * $zr;
			$zi2 = $zi * $zi;
			$zr = $zr2 - $zi2 + $cr;
			$zi = $temp + $temp + $ci;
			if ($zi2 + $zr2 > BAILOUT)
				return $i;
			if ($i > MAX_ITERATIONS)
				return 0;
		}
	}
}
```

```
$m = new Mandelbrot();
?>
```