JL: JSON Manipulation Language

Sam Shelley, Yongqiang Tan, Robert Wallace

August 16, 2013

Contents

1 Introduction

JSON is an interchange standard used to transmit structured data between a server and a web application in a human-readable format. JSON parsers have been built for nearly every language and allow data to be translated into the each language's native data structures. The data format has become incredibly popular and is used by many of the world's most popular websites for access to their data, both externally and internally. Unfortunately though, there does not exist a language specifically focused on quickly accessing and manipulating structured JSON objects. Many other languages require creating complex class definitions, or abstractions to facilitate JSON use.

As JSON has become the predominant format for transmitting data structures and arrays on many major APIs, the usefulness of a simple domain specific language drove the development of this project. A common workflow for API data interaction involves a series of steps and procedures sometimes spanning several scripting languages. As a first step, there is an initial call to the API in which JSON formatted objects are returned. Then the JSON objects string format is parsed into several objects. Finally, various manipulations and aggregations are performed depending on the task at hand.

JL (JSON Language) is a general purpose language that facilitates programming at each of the above steps, with particular emphasis on enabling navigation and item specification within a JSON objects structure. While maintaining the original JSON objects hierarchy, the language allows the hashmap, and list structures inherit in JSON objects to be easily accessed and manipulated through simple statements or functions.

2 Language Tutorial

This section gives a brief overview of our language. More examples are given in the test and demo sections.

2.1 Two Simple Examples

A JL program consists of statements and function definitions. A statement can be an *assignment*, an *if* statement, a *while* statement, a *foreach* statement, a *function call*, or a *block* of statements. A function definition begins with a keyword *fun*, a optional list of arguments, and is followed by a statement.

JL supports string, int, float, string, boolean, list, and hashtable data types. The elements of the list can be of any type. Here is an example that shows some features of the language as a general purpose language.

```
fun fib(n){
    if n < 2 then
        return n;
    else
        return fib(n-1)+fib(n-2);
}
```

```
print("fib(5) is "+fib(5));
```

JL has a built-in funciton *parse* that parses a JSON string into a JL expression. JL's built-in function *gets* grabs content from internet. JL supports "selector" inspired by XPath. A common idiom is to grab JSON object online, parse it, and select the information the user is interested in. Here is a program that does this.

```
a = parse(gets("http://localhost:3000/twittersample.json"));
b = 400;
fun atLeastXFollowers(obj) {
    return obj[:"followers_count"] > b;
}
result = a[::"name">,atLeastXFollowers];
acc = [];
foreach x : result do {
      acc = acc + [
                  {
                      "name": x[:"name"],
                      "location":x[:"location"],
                      "img":x[:"profile_image_url"]
                  }
                ];
}
print(acc);
```

2.2 Compiling and Running

The compiler can be created with the make command:

```
make jl
```

This will compile and assemble the compiler into an executable object.

To then compile a JL program, the jl executable compiles stdin. Therefore it can be run by redirecting source code like:

i.e. './jl <Program.jl'

The jl executable has four options:

-i to run the program in the interpreter

-s to run a semantic check of code

-b to output the bytecode

-c (the default) to compile the program into bytecode and then execute it in the bytecode interpreter

3 Language Manual

3.1 Lexical Conventions

3.1.1 Line Terminators

Expressions are terminated with a semi-colon,“;”

3.1.2 Comments

Comments begin with the characters /* and end with the characters */ and may span multiple lines. Comments do not nest inside one another.

3.1.3 Whitespace

Spaces, tabs, line terminators, and comments are all considered whitespace, and are ignored, except for tokenization and within strings.

3.1.4 Identifiers

An identifier is defined as sequence of letter and numbers of any length, starting with a letter.

3.2 Data Types

Data types can either be atomic or compound. Types are dynamically assigned during runtime, but combinations between types will be cast automatically, and throw a casting error if impermissible.

3.2.1 Atomic

The atomic data types are:

int, boolean, float, string, null

3.2.2 Compound

The compound data types are:

list, hashmap

3.3 Constants

3.3.1 Numeric Int

A numeric int constant is defined as one or more digits. It can be optionally preceded by a negative sign to delineate a negative integer.

3.3.2 Numeric Floats

A numeric float constant is defined as one or more digits, followed by a decimal point, followed by one or more digits.

3.3.3 String Literals

A string literal constant is defined as anything inside of a pair of double quotation marks.

3.3.4 Booleans

Booleans are delineated by either a $true$ or $false$ token.

3.3.5 Lists

Lists are styled after JSON lists and can be constructed using: [$value1$,$value2$, ...]. Values can be any of the built in data types.

3.3.6 Hashmaps

Hashmaps are styled after JSON object syntax and can be constructed using: {$key1$:$value1$, $key2$:$value2$,...}. Keys types are restricted to strings. Values can be any of the built in data types.

3.3.7 Undefined/Null

A null type also exists and is delineated by *null*.

3.4 Keywords

The following identifiers are reserved as keywords, and may not be used:

as	break	continue	do
else	foreach	fun	gets
if	parse	print	return
true	false	then	while

3.5 Operators

3.5.1 Mathematical

The following mathematical operators are supported:

+, -, *, /

+ is also overloaded for string, list, and hashmap concatenations.

3.5.2 Boolean

The following boolean operators are allowed:

>, <, >=, <=, ==, !=

3.5.3 Logical

The following logical operators are allowed:

&&, ||

3.6 Separators

The following characters are separators:

{ } [ ] ( ) : :: ; ,

3.7 Scope Rules

There are two scopes: global and local scopes. A variable is defined when it is first used as an lvalue (the id in $id = expr$; or $foreach\ id\ :\ expr\ do\ stmt$) and it is not in the local or global scopes. A unbound lvalue inside a function definition creates a local variable. A formal argument is also a local variable and it shadows any global variable of the same name. The life time of local variables ends when the function returns.

3.8 Selectors

JL uses selector syntax inspired by XPath to permit quick access to to the value associated with a given key in objects or arrays. Selectors are delineated using [ ] at the end of hashmaps/lists.

3.8.1 Root level

The simplest selector is identified by *object*[:*key*]. This selector is an alias of *object*[:*key*1] which will look only in the root level of the the object return the value associated with the provided *key*.

3.8.2 Any position

Using a double colon *object*[::*key*] descends to the bottom of an object finding all values with the listed identifier, returning a list of matching values.

3.8.3 Chaining

Selectors can be chained together like *object*[:*key*1:*key*2]. This example finds values associated with *key*2 in an object that is associated with *key*1.

3.8.4 Arrays

Arrays are treated as objects keyed by positive monotonically increasing integers and are accessed in the same way as objects: *array*[:int]. Arrays can also be chained if they are nested, *array*[:int:int]. They can also be accessed when chaining selectors with objects, *object*[:*key*:int].

3.8.5 Parents

If the value returned by a key is an object or a list, its containing compound constant is returned. Select parents by including > at the end of the selector like *object*[:*key* >] (otherwise nothing is returned)

3.8.6 Restrictions

Restrictions are functions which take a single parameter. They can be listed after the selector chain and optional postfix. *object*[:*key*, *restriction*, ...]. Whenever a selector finds a matching *key* in the object or array, it call each of the restriction functions passing in the parent as the parameter. This enables additional selectors to be run in context. The discovered value will only be included/returned if all of the restrictions evaluate to *true*.

```
fun restrictionFunction(obj) {
    return obj[:key2] == "Foo";
}
$object[$::$key, restrictionFunction]$
```

This program selects keys matching *key*, so long as *key*2 is a sibling that has the value "Foo".

3.8.7 Return Types

Selectors using any double colons will return a list of results or an empty list if none are found. Selectors using only single colons will return either a single typed constant, or *null* if no value is matched.

3.9 Function Definitions

Functions are defined as:

fun $identifier$ (*parameter-list*) *statement*

where

parameter-list :

ϵ
| $identifier$
| $identifier$, *parameter-list*

3.10 Expressions

The following describes the expressions, in decreasing order of priority.

3.10.1 Constant

An int, boolean, float, string, list, hash map or null constant is an expression. The value of the expressions is the constant.

3.10.2 logical-expression

A logical expression is one or more boolean expression concatenated by && (AND) or ||(OR). A boolean expression is of the form expression OP expression, where OP can be >, <, >=, <=, ==, or of the form ! expression.

3.10.3 selector-expression

The selector expression can be applied to hashmaps or lists to access elements within those items.

SelectorExpression → ((: | ::)string) + (>)?(, $identifier$)∗

3.10.4 compound[selector-expression]

The compound must be a hashmap or list, and the selector-expression must be a valid selector expression. An error will be thrown if the enclosed expression is not valid.

3.10.5 identifier(expression-list-opt)

The identifier must be the name of a function. A functional call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function.

3.10.6 expression * expression

The * indicates multiplication. Both expression must be of type float. The result has type float.

3.10.7 expression / expression

The / indicates division. Both expression must be of type float. The result has type float.

3.10.8 expression + expression

The + indicates addition. Both expression must be of type float. The result has type float.

3.10.9 expression - expression

The - indicates subtraction. Both expression must be of type float. The result has type float.

3.10.10 Variable

A variable is an expression. The value of the expression is the value of the variable.

$Variable \rightarrow expression$

3.10.11 (expression)

An expression enclosed in parentheses is an expression. Parentheses can be used to indicate precedence.

3.10.12 lvalue

The only lvalue is an identifier. As selectors can't be identifiers, values contained within hashmaps and lists are immutable.

$lvalue \rightarrow identifier$

3.11 Statements

3.11.1 Expression Statement

The most simple *statement* is created by adding a semicolon to the end of an expression.

statement → *expression*; |{*statement-list*}

Usually expression statements are assignments or function calls.

3.11.2 Compound Statement

Several statements can be used as a compound statement where one is expected. A compound statement takes the form: *statement-list*

where

statement-list:

statement
| *statement statement-list*

3.11.3 Conditional Statement

The two forms of conditional statements are:

if (*expression*) then *statement*

if (*expression*) then *statement* else *statement*

The else ambiguity is resolved by connecting an else with the last encountered else less if.

3.11.4 Foreach Statement

The foreach statement has the form:

foreach $identifier$: *expression* do *statement*

3.11.5 While Statement

The while statement has the form:

while *expression* do *statement*

3.11.6 Return Statement

A function returns to its caller by means of the return statement, which has the form:

return *expression*;

A function may have multiple return statements. They do not need to return the same type.

3.12 Examples

3.12.1 Convert an array to a keyed map

This is a function which takes a json object consisting of a list of objects containing name keys, and creates a new object which makes the names keys to access the objects. It assumes that names are unique.

```
fun makeNameKey(jsonObject) {
  result = {};
  for i : jsonObject do {
    result = result + {objectsWithName[: i:name]: objectsWithName};
  }
  return result;
}
```

input

```
[
  {
    name: "John",
    from: "New York"
   },
  {
    name:"Bob",
    from:"California"
   }
]
```

output

```
{
  "John":{
    name: "John",
    from: "New York"
  },
  "Sue": {
    name:"Bob",
    from:"California"
  }
}
```

3.12.2 Combine differently structured json

In this example, one data set is stored by seller, and another is stored by product. Our language allows the quick comparison of the two concepts with the following program, which returns a list of the pricing data.

```
fun restrictToGizmo(obj) {
        return obj[:name]=="Gizmo"
}
fun comparePricingData() {
   pricingData1 = gets(dataSet1)[::price];
   pricingData2 = gets(dataSet2)[::price, restrictToGizmo];
   return pricingData1 + pricingData 2;
}
```

dataSet1

```
[
  {
   "product_id": 1232131,
   "product_name": "Gizmo",
    "sellers": [
      {
        "name": "Discount Shop",
        "price": 1.5
      },
      {
        "name": "Luxury Good",
        "price": 2.75
      }
    ]
   }
]
```

dataSet2

```
[
  {"seller_id": 4324,
    "name": "Tech Deals",
    "products": [
      {
        "name": "Gizmo",
        "price": 1.4
      },
      {
        "name": "Widget",
        "price": 1.75
      }
    ]
  }
]
```

output

```
[1.5, 2.75, 1.4]
```

4 Project Plan

4.1 Development Process

Discussion about the project occurred mainly through email communication, as it was the most convenient way to discuss ideas asynchronously. The group would also hold meetings in class during the break time, to further discuss certain issues or clarify questions and to divide up the project work. Emails were then used to keep the rest of the group involved on progress, and to ask for assistance with any issues that may have come up.

4.2 Project Timeline

The following dates were set as deadlines for the major project milestones:

7-15-13	Project Proposal
7-24-13	Language Reference Manual
7-28-13	Parser Completed
8-4-13	Interpreter Completed
8-11-13	Compiler Completed
8-16-13	Presentation and Completed Report

4.3 Team Roles

Yongqiang lead the team in Language Design and Implementation. He and Sam headed up coding the compiler. Rob helped out with coding as needed and created the Test Suite and Report.

4.4 Software Development Environment

The project was developed using OCaml 4.00.1 on UNIX-based systems. Version control was done with Git using the GitHub website.

4.5 Project Log

The following are the actual dates of project milestones:

7-8-13	Group Assembled
7-15-13	Proposal Completed
7-17-13	Proposal Feedback
7-22-13	Grammar Finalized
7-24-13	LRM Completed
7-29-13	LRM Feedback
7-31-13	AST First Draft
8-5-13	Interpreter First Draft
8-6-13	Compiler First Draft
8-12-13	Compiler Completed
8-16-13	Project Completed

5 Architectural Design

5.1 Components and Interfaces

The following five sections are the major components to the JL Compiler. Following them in order provides a path from user typed code all the way to the compiled bytecode.

5.1.1 Lexical Analyzer

The lexical analyzer accepts a stream of characters and converts them into a token stream. The tokens are defined in scanner.mll, which makes up the set of acceptable words in the language. The stream of tokens is then passed into the Parser.

5.1.2 Parser

The Parser makes sure that the stream of tokens received is consistent with the grammar rules defined in parser.mly and that the order the tokens are combined are correct for the language syntax. While making sure the grammar is followed, the parser also creates an Abstract Syntax Tree with the help of structures defined in ast.ml.

5.1.3 Semantic Analyzer

The Semantic Analyzer makes sure that the syntax of the program is meaningful and that the program that conforms to the syntax also makes sense. This includes assignment of variables to values, operations done on correct types, and the scope of variable usage. This is done in the file sast.ml

5.1.4 Translator and Executor

Once the semantic analysis has finished successfully, the AST structure can be translated to our custom bytecode operations and run through our custom bytecode executor. This is done in the files compile.ml and execute.ml

5.1.5 JL

The file jl.ml is provided for an easy way to run the compiler, using the command line interface. Various command line options detailed in “Compiling and Running” allow the user an easy way to either run their program in the interpreter, output the generated bytecode for their program, or to generate and run their compiled bytecode.

5.2 Language Design

JL is a general purpose, simple, procedural, platform neutral, dynamic language with built-in support for JSON manipulation.

General JL is designed to be a general purpose language that suits a broad range of applications. It supports 6 commonly used data types: int, float, boolean, string, list, and hash table, and it supports sequential, conditional, and loop flow controls, function calls and recursions.

Simple JL is a simple language. Variables are dynamic, no variable declaration is needed. Variables are either local or global. The only lvalue is an identifier (variable). An assignment statement *id* = *expr*; creates a variable if *id* is not a local variable/formal argument nor a global variable. A variable is garbage collected when no longer used (thanks to OCaml’s memory management system).

Procedural A JL program consists of statements and function definitions. The statements are executed line by line; thus no main function is needed.

Platform neutral JL is designed to be platform neutral. Currently the only part that is platform dependent is the built-in function *gets*, which only supports Unix-like platforms.

Dynamic Variables are dynamic typed. Variables can change type, and often their types can only be determined at run time.

JSON support A JSON string can be parsed to be a *jsonexpression*, which is a subset of JL *expression*. A special ”selector” syntax is designed to filter out the user interested information.

6 Test Plan

6.1 Test Program One - Twitter JSON

'Twitter JSON' JL Source Code

```
a = parse(gets("http://localhost:3000/twittersample.json"));
fun atLeastXFollowers(obj) {
    return obj[:"followers_count"] > 400;
}
result = a[::"name">,atLeastXFollowers];
acc = [];
fun doThings(result) {
    foreach x : result do {
        acc = acc + [{"name": x[:"name"], "location":x[:"location
           "]}];
    }
}
doThings(result);
print(acc);
```

'Twitter JSON' Bytecode

```
3 global variables
0 Jsr 2
1 Hlt
2 Ent 0
3 Lit http://localhost:3000/twittersample.json
4 Jsr -2
5 Jsr -3
6 Str 2
7 Drp
8 Rsn 56
9 Lis 1
10 Lit >
11 Lit "name"
12 DCo
13 Lod 2
14 Slc 1
15 Str 1
16 Drp
17 Lis 0
18 Str 0
19 Drp
20 Lod 1
21 Jsr 28
22 Drp
23 Lod 0
24 Jsr -1
```

```
25 Drp
26 Lit 0
27 Rts 0
28 Ent 1
29 Lfp -2
30 Bra 23
31 Sfp 1
32 Lod 0
33 Lis 0
34 Lit
35 Lit "name"
36 Co
37 Lfp 1
38 Slc 1
39 Lit "name"
40 Lis 0
41 Lit
42 Lit "location"
43 Co
44 Lfp 1
45 Slc 1
46 Lit "location"
47 Has 2
48 Lis 1
49 Add
50 Str 0
51 Drp
52 Drp
53 Nit -22
54 Lit 0
55 Rts 1
56 Ent 0
57 Lis 0
58 Lit
59 Lit "followers_count"
60 Co
61 Lfp -2
62 Slc 1
63 Lit 400
64 Gt
65 Rts 1
66 Lit 0
67 Rts 1
```

6.2 Test Program Two - Fibonacci Series

'Fibonacci' JL Source Code

```
fun fib(n) {
        if n < 2 then
                return n;
        else
                return fib(n-1) + fib(n-2);
}

print("fib(5) is " + fib(5));
```

'Fibonacci' Bytecode

```
0 global variables
0 Jsr 2
1 Hlt
2 Ent 0
3 Lit fib(5) is
4 Lit 5
5 Jsr 11
6 Add
7 Jsr -1
8 Drp
9 Lit 0
10 Rts 0
11 Ent 0
12 Lfp -2
13 Lit 2
14 Lt
15 Beq 4
16 Lfp -2
17 Rts 1
18 Bra 11
19 Lfp -2
20 Lit 1
21 Sub
22 Jsr 11
23 Lfp -2
24 Lit 2
25 Sub
26 Jsr 11
27 Add
28 Rts 1
29 Lit 0
30 Rts 1
```

6.3 Test Suite Details

The tests in the test suite were created using the language reference manual as a guide. The tests were designed to fall into two categories: 'test to pass' and 'test to fail'. 'Test to pass' tests were designed to test valid inputs and ensure that the resulting output is as expected. 'Test to fail' tests were designed to test invalid inputs and ensure that no invalid inputs will be compiled. Testing was automated using a shell script.

Comment-1-Test - Commands inside comments do not get executed.

```
/* print(1); */
```

Comment-2-Test - Comments may span multiple lines

```
/* print(1);
print(2);
print(3); */
```

Comment-3-Fail - Comment missing terminating comment characters

```
/* print(1);
```

Conditional-1-Test - The 'then' statement is executed

```
if (1) then
        print("True");
```

Conditional-2-Test - The 'then' statement is not executed

```
if (0) then
        print("True");
```

Conditional-3-Test - The 'else' statement is executed

```
if (0) then
        print("True");
else
        print("False");
```

Float-1-Test - Valid float combinations

```
a = 1.2;
b = 3.45;
c = 6.789;
```

Float-2-Fail - Invalid float with no digit preceding the decimal point

```
a = .2;
```

Float-3-Fail - Invalid float with no digits succeeding the decimal point

```
a = 2.;
```

Foreach-1-Test - Iterating through a list variable

```
a = [1, 2, 3, 4, 5];

foreach x : a do {
        print(x);
}
```

Foreach-2-Test - Iterating through a doubly nested list

```
a = [[1, 2], [3, 4], [5, 6]];

foreach x : a do {
        foreach y : x do {
                print(y);
        }
}
```

Foreach-3-Fail - Attempting to iterate through an integer constant

```
foreach x : 5 do {
        print(x);
}
```

Foreach-4-Test - Iterating directly through a list

```
foreach var : [1, 2, 3] do {
        print(var);
}
```

Function-1-Test - Function with no parameters

```
fun test() {
        print("In function");
}

test();
```

Function-2-Test - Function with single parameter

```
fun test(a) {
        print(a);
}

test(1);
```

Function-3-Test - Function with multiple parameters

```
fun test(a, b, c) {
        print(a);
        print(b);
        print(c);
}

test(1, 2, 3);
```

Function-4-Test - Function return value is correct

```
fun test(a) {
        return a + 2;
}

print(test(2));
```

Gets-1-Test - GETS an URL

```
print(gets("http://maps.googleapis.com/maps/api/geocode/json?
   address=1600+Amphitheatre+Parkway,+Mountain+View,+CA"));
```

Gets-2-Fail - GETS an integer

```
print(gets(1));
```

Hashmap-1-Test - Empty hashmap

```
a = {};
print(a);
```

Hashmap-2-Test - Hashmap containing a single key/value pair

```
a = {"name":"John"};
print(a);
```

Hashmap-3-Test - Hashmap containing multiple items

```
a = {
"something":1,
"somethingelse":"test"
};
print(a);
```

Hashmap-4-Test - Adding two hashmaps

```
a = {"test1":"a"};
b = {"test2":"b"};
print(a+b);
```

Hashmap-5-Fail - Adding a list and a hashmap

```
a = {"test1":"a", "test2":"b"};
b = [1, 2, 3, 4];
print(a+b);
```

Hashmap-6-Fail - Subtracting a list and a hashmap

```
a = {"test1":"a", "test2":"b"};
b = [1, 2, 3, 4];
print(a-b);
```

Identifier-1-Test - Various valid identifier combinations

```
a = 1;
my_variable = 2;
CamelCaseVariable = 3;
```

Identifier-2-Fail - Invalid identifier starting with a number

```
1numberVariable = 3;
```

Identifier-3-Fail - Invalid identifier containing invalid characters

```
dash-variable = 4;
```

Int-1-Test - Valid integer combinations

```
a = 0;
b = 1;
c = 10;
d = 23045;
```

Int-2-Test - Double negative integer

```
a = 2;
b = --2;
print(a+b);
```

Int-3-Test - Addition and substraction with negative integers

```
a=2;
b=2-1;
c=2+-1;
d=1 + 2 - 2;
print(a);
print(b);
print(c);
print(d);
print(a + -(b + - -c) - d);
```

Int-4-Test - Multiplication and division with negative integers

```
a = 2 - 2*2;
b = 2 - 4/5;
print(a);
print(b);
```

List-1-Test - Empty list

```
a = [];
print(a);
```

List-2-Test - Single item list

```
a = [1];
print(a);
```

List-3-Test - Multi item list

```
a = [1, 2, 3, 4];
print(a);
```

List-4-Test - List containing a list

```
a = [1, 2, [3, 4, 5], 6];
print(a);
```

List-5-Test - Adding two lists

```
a = [1, 2, 3];
b = [3, 4, 5];
print(a + b);
```

List-6-Test - Subtracting two lists

```
a = [1, 2, 3];
b = [3, 4, 5];
print(a - b);
```

Math-1-Test - Various valid mathematical expression

```
print(1+2);
print(3-4);
print(5*6);
print(9/3);
```

Math-2-Fail - Unbalanced mathematical expression

```
a = 1 + 2 - ;
```

Math-3-Test - Combined mathematical expressions

```
print(1 + 2 - 3 * 4);
```

Paren-1-Test - Single parentheses around an expression

```
a = (1);
```

Paren-2-Test - Parentheses force precedence

```
print((1+2)*3);
```

Paren-3-Fail - Unbalanced parentheses

```
a = (1;
```

Paren-4-Test - Nested parentheses

```
print((1+(2)));
```

Parse-1-Test - Parsing a URL via GETS

```
print(parse(gets("http://maps.googleapis.com/maps/api/geocode/json
    ?address=1600+Amphitheatre+Parkway,+Mountain+View,+CA")));
```

Parse-2-Test - Parsing a self constructed list

```
print(parse("[1,2,3,4,5]"));
```

s Select-1-Test - Select all that match with a restriction

```
a = [{"test":2, "bad": true}, {"test":3, "bad":false}];

fun dummy (obj) {
    return obj[:"bad"] == true;
}

print(a[::"test">, dummy]);
```

Scope-1-Test - Global level scope reassignment

```
a = 1;
b = a;
print(b);
```

Scope-2-Test - Local level scope reassignment

```
fun test() {
        a = 1;
        b = a;
        print(a);
}

test();
```

Scope-3-Test - Local scope using global scope for reassignment

```
fun test() {
        b = a;
        print(b);
}

a = 1;
test();
```

Scope-4-Fail - Undeclared global variable

```
a = 1;
print(b);
```

Scope-5-Fail - Undeclared local variable

```
fun test() {
        print(b);
}

test();
```

Select-2-Test - Selecting with descending one level from the root in a list

```
a = [1,2,3];
print(a[:"1"]);
```

Select-3-Test - Selecting with descending one level from the root in a hashmap

```
a = {"test":2};
print(a[:"test"]);
```

Select-4-Test - Chaining multiple selectors together

```
a = {"test":{"test":2}};
print(a[:"test":"test"]);
```

Select-5-Test - Selecting a specific key value

```
a = [{"test":2},{"other":3}];
print(a[::"test"]);
```

Select-6-Test - Selecting a position in a list

```
a = [1,2,3];
print(a[::"0"]);
```

Select-7-Test - Selecting all that match

```
a = {"test":[2,1,3],"test2":[2,3,2]};
print(a[::"1"]);
```

Select-8-Test - Selecting a null value

```
a = [1,2,3];
print(a[:"3"]);
```

Select-9-Test - Selecting an empty set of elements

```
a = [1,2,3];
print(a[::"3"]);
```

Statement-1-Test - Single valid statement

```
a = 2;
```

Statement-2-Fail - Statement missing semicolon at the end

```
a = 2
```

String-1-Test - Valid string combinations

```
a = "a";
b = "ab";
c = "abc";
d = "testing";
e = "123abc";
```

String-2-Fail - String containing double quotation marks

```
a = "test"ing";
```

String-3-Test - String containing escaped double quotation mark

```
print("abc\"def");
```

While-1-Test - Simple While loop

```
a = 1;
while (a < 5) do {
        print(a);
        a = a + 1;
}
```

While-2-Test - Nested While loops

```
a = 1;
b = 10;
while (a < 5) do {
        while (b > 5) do {
                print(b);
                b = b - 1;
        }
        print(a);
        a = a + 1;
}
```

7 Lessons Learned

7.1 Sam Shelley

- Write some interesting examples in the language first. This is immensely helpful in guiding language design
- Design a language which solves a problem that you are very familiar. Solving a personal pain point will make the language more useful.
- OCaml and its lexing and parsing extensions are incredibly powerful for compiler building.

7.2 Yongqiang Tan

- Start early. Roll out a working version as soon as possible. This working version may have very limited features, but to make a *helloworld* program run is the first step to success. We skipped the semantic checking at first, and the checking was added after we had a working version.
- Do the interpreter before the bytecode executor. Doing the interpreter is a good exercise to practice ocaml, and when adding a new feature to the language, it is easier to test it with an interpreter.
- Ocaml rocks! Ocaml type system rocks! The syntax is simple, and it is easy to use once you know how to use it. Java or other similar languages are much more verbose, and treat everything as an object. To have it do some work, we have to add a lot of "syntactic noise". Ocaml allows us to focus on the problem we are solving rather than the cumbersome of the language we use.
- Teamwork is essential. Divide the problem to small pieces and keep everybody synchronized. Version control is key for teamwork to work.

7.3 Rob Wallace

- Accept the fact you have to work with OCAML
- Try your best to wrap your head around functional programming ASAP if you're not used to it
- Divide work up based on your strengths

8 Appendix

8.1 ast.ml

```
let removeQuotes str = String.sub str 1 ((String.length str) - 2);;

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater |
  Geq

type select = Colon | DoubleColon


type expr =
    Literal of int
  | Str of string
  | Float of float
  | Boolean of bool
  | Id of string
  | Select of expr * (select * expr) list * string * string list (*
     should be a boolean function *)
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Lis of expr list
  | Hash of (string * expr) list
  | Undef
  (*| Noexpr*)
and restriction =
  Restriction of expr * op * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | Foreach of expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    fname : string;
    formals : string list;
    locals : string list; (*for compiler*)
    body : stmt list;
  }

type gstmt = (*general statement*)
    Stmt of stmt
  | Func of func_decl

type program = gstmt list  (*string list * func_decl list*)

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Str(s) -> s
  | Float(f) -> string_of_float f
  | Boolean(b) -> string_of_bool b
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " "
        ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  (*| Noexpr -> ""*)
  | Select(id, chain, postfix, restrictions) ->
      string_of_expr id ^ "[" ^ String.concat "" (List.map (fun (t, id)
         -> if t=Colon then ":" ^ string_of_expr id else "::" ^
         string_of_expr id) chain) ^ postfix ^ ", " ^ String.concat ", "
          restrictions ^ "]" (* fix me *)
  | Lis(exprs_opt) -> "[" ^ String.concat "," (List.map string_of_expr
     exprs_opt) ^ "]"
  | Hash(kvpairs_list) -> let string_of_kvpair = fun (k, v) -> k ^ ":" ^
      string_of_expr v in "{" ^ String.concat "," (List.map
     string_of_kvpair kvpairs_list)  ^ "}"

and string_of_restriction = function
  Restriction(lhs,o,rhs) -> string_of_expr lhs ^ " " ^
    (match o with
      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
    | Equal -> "==" | Neq -> "!="
    | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
  string_of_expr rhs

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | Foreach(e1, e2, s) ->
      "foreach " ^ string_of_expr e1  ^ " : " ^ string_of_expr e2 ^ " do
         \n" ^
         string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt
     s

(*let string_of_vdecl id = "int " ^ id ^ ";\n"*) (*no need for
   declarations, variables are declared when first used*)

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  (* String.concat "" (List.map string_of_vdecl fdecl.locals) ^ *)
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"
```

```
let string_of_gstmt = function
    Stmt(s) -> string_of_stmt s
  | Func(f) -> string_of_fdecl f

let string_of_program gstmt = (*replace string_of_vdecl with gstmt*)
  String.concat "" (List.map string_of_gstmt gstmt)
  (*^ "\n" ^ String.concat "\n" (List.map string_of_fdecl funcs) *)
```

8.2 bytecode.ml

```
type bstmt =
    Lit of Types.t     (* Push a literal *)
  | Drp                (* Discard a value *)
  | Bin of Ast.op      (* Perform arithmetic on top of stack *)
  | Lod of int         (* Fetch global variable *)
  | Str of int         (* Store global variable *)
  | Lfp of int         (* Load frame pointer relative *)
  | Sfp of int         (* Store frame pointer relative *)
  | Jsr of int         (* Call function by absolute address *)
  | Ent of int         (* Push FP, FP -> SP, SP += i *)
  | Rts of int         (* Restore FP, SP, consume formals, push result *)
  | Beq of int         (* Branch relative if top-of-stack is zero *)
  | Bne of int         (* Branch relative if top-of-stack is non-zero *)
  | Bra of int         (* Branch relative *)
  | Lis of int         (* List construction *)
  | Has of int         (* Hash table construction *)
  | Rsn of int         (* Call restriction function by absolute address
    *)
  | Sel of Ast.select (* Selector chain literal *)
  | Slc of int         (* Selector construction *)
  | Nit of int         (* Next iteration *)
  | Hlt                (* Terminate *)

type prog = {
    num_globals : int;    (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ Types.string_of_typed_expr i
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Lis(i) -> "Lis " ^ string_of_int i
  | Has(i) -> "Has " ^ string_of_int i
  | Rsn(i) -> "Rsn " ^ string_of_int i
  | Sel(Ast.Colon) -> "Co"
  | Sel(Ast.DoubleColon) -> "DCo"
  | Slc(i) -> "Slc " ^ string_of_int i
  | Nit(i) -> "Nit " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

8.3 compile.ml

```
open Bytecode

module NameMap = Map.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

module StringMap = Map.Make(String)

(*
  implement the scope rule
  an assignment at toplevel creates a global variable if it is not
      defined yet.
  an assignment at a fun_decl, however, creates a local variable when
      the left
  hand side is neither a global nor formal let variable.
  a formal variable shadows a global variable of the same name
*)
let rec env_expr env formals toplevel = function
      Ast.Assign(var, expr) -> let globals, locals, funcs = env in
                               if List.mem var formals || List.mem var
                                  globals then
                                   env
                               else if toplevel then
                                   var :: globals, locals, funcs
                               else
                                   globals, var :: locals, funcs
    | _                     -> env

and env_stmt env formals toplevel = function
      Ast.Expr(e)        -> let env = env_expr env formals toplevel e in
         env
    | Ast.Block(stmts)   -> List.fold_left (fun env s -> env_stmt env
       formals toplevel s) env stmts
    | Ast.If(e,s1,s2)    -> let env = env_expr env formals toplevel e in
                            let env = env_stmt env formals toplevel s1 in
                            let env = env_stmt env formals toplevel s2 in
                            env
    | Ast.While(e,s)     -> let env = env_expr env formals toplevel e in
                            let env = env_stmt env formals toplevel s in
                               env
    | Ast.Foreach(e1,e2,s) -> let globals, locals, funcs = env in
                               let env = (match e1 with
                                   Ast.Id id -> if toplevel then id::
                                      globals, locals, funcs
                                             else globals, id ::
                                                locals, funcs
                                 | _ -> raise (Failure "e1 in foreach has
                                    to be an identifier"))
                               in env_stmt env formals toplevel s
    | _                  -> env

and env_gstmt env = function
      Ast.Stmt(s)      -> let env = env_stmt env [] true s in env
    | Ast.Func(fdecl) -> let env = env_fdecl env fdecl.Ast.formals fdecl
        in env

and env_fdecl env formals fdecl =
    let globals, localvars, funcs = List.fold_left (fun env s ->
       env_stmt env formals false s) env fdecl.Ast.body in
    let funcs = NameMap.add fdecl.Ast.fname {fdecl with Ast.locals=
       localvars} funcs in
    globals, [], funcs
```

```
(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* "Address" for global variables
       *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a
   *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(** Translate a program in AST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate gstmts =
  let (globals, _, funcs) = List.fold_left env_gstmt ([], [], NameMap.
     empty) gstmts in
  let functions = NameMap.fold (fun k v a -> v :: a) funcs [] in

(*let translate (globals, functions) =*)

  (* Allocate "addresses" for each global variable *)
  let global_indexes = string_map_pairs StringMap.empty (enum 1 0
     globals) in

  (* Assign indexes to function names; built-in "print" is special *)
  let built_in_functions = StringMap.add "print" (-1) StringMap.empty in
  let built_in_functions = StringMap.add "gets" (-2) built_in_functions
     in
  let built_in_functions = StringMap.add "parse" (-3) built_in_functions
      in
  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> f.Ast.fname) functions)) in

  let rec expr env = function
      Ast.Literal i -> [Lit (Types.Int i)]
    | Ast.Float f   -> [Lit (Types.Float f)]
    | Ast.Str s     -> [Lit (Types.Str s)]
    | Ast.Boolean b -> [Lit (Types.Boolean b)]
    | Ast.Undef     -> [Lit (Types.Undef)]
    | Ast.Lis l     -> List.concat (List.map (expr env) l) @ [Lis (List.
      length l)]
    | Ast.Hash h    -> List.concat (List.map (fun (k,v) ->(expr env v) @
       [Lit (Types.Str k)]) h) @ [Has (List.length h)]
    | Ast.Id s ->
       (try [Lfp (StringMap.find s env.local_index)]
       with Not_found -> try [Lod (StringMap.find s env.global_index)]
       with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Ast.Binop (e1, op, e2) -> expr env e1 @ expr env e2 @ [Bin op]
    | Ast.Assign (s, e) -> expr env e @
       (try [Sfp (StringMap.find s env.local_index)]
       with Not_found -> try [Str (StringMap.find s env.global_index)]
       with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Ast.Select(id, chaining, postfix, restrictions) ->
      let rsn = (List.fold_left (fun acc fname -> [Rsn
                                    (try StringMap.find fname env.
                                       function_index
                                     with Not_found -> raise (Failure ("
                                        undefined function " ^ fname)))
                                  ]@acc) [] (List.rev restrictions)) in rsn @
                                    [Lis (List.length restrictions)] @
       [Lit (Types.Str postfix)] @
       List.concat (List.map (fun (k,v) ->(expr env v) @ [Sel k]) (List
          .rev chaining)) @
       (expr env id) @ [Slc (List.length chaining)]
    | Ast.Call (fname, actuals) -> (try
       (List.concat (List.map (expr env) (List.rev actuals))) @
       [Jsr (StringMap.find fname env.function_index) ]
       with Not_found -> raise (Failure ("undefined function " ^ fname)
          ))
    (*| Ast.Noexpr -> []*)

  in let rec stmt env num_formals = function
      Ast.Block sl     ->  List.concat (List.map (stmt env num_formals)
         sl)
    | Ast.Expr e       -> expr env e @ [Drp]
    | Ast.Return e     -> expr env e @ [Rts num_formals]
    | Ast.If (p, t, f) -> let t' = stmt env num_formals t and f' = stmt
      env num_formals f in
       expr env p @ [Beq(2 + List.length t')] @
       t' @ [Bra(1 + List.length f')] @ f'
    (*| Ast.For (e1, e2, e3, b) ->
       stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))])) *)
    | Ast.While (e, b) ->
       let b' = stmt env num_formals b and e' = expr env e in
       [Bra (1+ List.length b')] @ b' @ e' @
       [Bne (-(List.length b' + List.length e'))]
    | Ast.Foreach(e1, e2, b) ->
       let id = (match e1 with Ast.Id id -> id | _ -> raise (Failure "
          Foreach expects an indentifier")) in
             let a' = (try [Sfp (StringMap.find id env.local_index)]
                with Not_found -> try [Str (StringMap.find id env.
                   global_index)]
                with Not_found -> raise (Failure ("undeclared variable
                   " ^ id))) in
       let b' = a' @ (stmt env num_formals b) @ [Drp] and e' = expr env
           e2 in
       e' @ [Bra (1+ List.length b')] @ b' @
       [Nit (-(List.length b'))]


  in
  (* Translate a function in AST form into a list of bytecode statements
      *)
  let translate env fdecl =
    (* Bookkeeping: FP offsets for locals and arguments *)
    let num_formals = List.length fdecl.Ast.formals
    and num_locals = List.length fdecl.Ast.locals
    and local_offsets = enum 1 1 fdecl.Ast.locals
    and formal_offsets = enum (-1) (-2) fdecl.Ast.formals in
    let env = { env with local_index = string_map_pairs
      StringMap.empty (local_offsets @ formal_offsets) }


    in [Ent num_locals] @      (* Entry: allocate space for locals *)
      stmt env num_formals (Ast.Block fdecl.Ast.body) @  (* Body *)
      [Lit (Types.Int 0); Rts num_formals]   (* Default = return 0 *)

  in let env = { function_index = function_indexes;
     global_index = global_indexes;
     local_index = StringMap.empty } in

  let translate_toplevel =
     (Ent 0) :: stmt env 0 (Ast.Block (List.rev (List.fold_left (fun s e
       -> (match e with
        Ast.Stmt(st) -> st::s
       | _ -> s)) [] gstmts)))
    @ [Lit (Types.Int 0); Rts 0]
  in

  (* Code executed to start the program: Jsr main; halt *)
  let entry_function = translate_toplevel(*try
    [Jsr (StringMap.find "main" function_indexes); Hlt]
  with Not_found -> raise (Failure ("no \"main\" function"))*)
```

```
  in

  (* Compile the functions *)
  let func_bodies = entry_function :: List.map (translate env) functions
      in

  (* Calculate function entry points by adding their lengths *)
  let (fun_offset_list , _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],2) func_bodies
          in
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
        indexes in Jsr statements with PC values *)
    text = Array.of_list ((Jsr 2) :: Hlt :: List.map (function
        Jsr i when i > 0 -> Jsr func_offset.(i)
      | Rsn i when i > 0 -> Rsn func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

8.4 execute.ml

```
open Ast
open Bytecode

exception SelectReturnException of bool

(* Stack layout just after "Ent":

                <-- SP
   Local n
   ...
   Local 0
   Saved FP     <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let boolean = function
    Types.Boolean(true) -> true
  | _                   -> false

let int_of_t = function
    Types.Int i -> i
  | _           -> raise (Failure "Invalid frame pointer")
```

```
let execute_prog prog =
  let stack = Array.make 1024 Types.Undef
  and globals = Array.make prog.num_globals Types.Undef in
  let rec listcons n top s =
    if n == 0 then s
          else listcons (n-1) (top-1) (stack.(top) :: s) in

  let rec hashcons n top s =
        if n == 0 then s
    else hashcons (n-1) (top-2) ((Types.string_of_typed_expr stack.(top)
        , stack.(top-1)) :: s) in
  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)
  | Drp    -> exec fp (sp-1) (pc+1)
  | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
      stack.(sp-2) <- (let boolean = function
          Types.Boolean(true) -> Types.Int 1
        | _                   -> Types.Int 0 in
            match op with
        Add     -> Types.addition op1 op2
      | Sub     -> Types.subtraction op1 op2
      | Mult    -> Types.multiplication op1 op2
      | Div     -> Types.division op1 op2
      | Equal   -> boolean (Types.typed_eq op1 op2)
      | Neq     -> boolean (Types.typed_neq op1 op2)
      | Less    -> boolean (Types.typed_less op1 op2)
      | Leq     -> boolean (Types.typed_leq op1 op2)
      | Greater -> boolean (Types.typed_greater op1 op2)
      | Geq     -> boolean (Types.typed_geq op1 op2) );
      exec fp (sp-1) (pc+1)
  | Lod i   -> stack.(sp)   <- globals.(i)   ; exec fp (sp+1) (pc+1)
  | Str i   -> globals.(i)  <- stack.(sp-1)  ; exec fp sp     (pc+1)
  | Lfp i   -> stack.(sp)   <- stack.(fp+i)  ; exec fp (sp+1) (pc+1)
  | Sfp i   -> stack.(fp+i) <- stack.(sp-1)  ; exec fp sp     (pc+1)
  | Jsr(-1) -> print_endline (Types.string_of_typed_expr stack.(sp-1)) ;
      exec fp sp (pc+1)
  | Jsr(-2) -> stack.(sp-1) <- Util.gets stack.(sp-1) ; exec fp sp (pc
     +1)
  | Jsr(-3) -> stack.(sp-1) <- Util.parse stack.(sp-1) ; exec fp sp (pc
     +1)
  | Jsr i   -> stack.(sp)   <- Types.Int (pc + 1)      ; exec fp (sp+1) i
  | Ent i   -> stack.(sp)   <- Types.Int fp            ; exec sp (sp+i+1)
      (pc+1)
  | Rts i   -> let new_fp = int_of_t stack.(fp) and new_pc = int_of_t
     stack.(fp-1) in
                if new_pc < 0 then raise (SelectReturnException(Types.
                   toBool(stack.(sp-1))))
```

```
              else stack.(fp-i-1) <- stack.(sp-1); exec new_fp (fp-i)
                new_pc
| Beq i    -> exec fp (sp-1) (pc + if boolean (Types.typed_eq stack.(sp
  -1) (Types.Int 0)) then i else 1)
| Bne i    -> exec fp (sp-1) (pc + if boolean (Types.typed_neq stack.(
  sp-1) (Types.Int 0)) then i else 1)
| Bra i    -> exec fp sp (pc+i)
| Lis i    -> stack.(sp-i) <- Types.Lis (listcons i (sp-1) []);
              exec fp (sp-i+1) (pc+1)
| Has i    -> stack.(sp-2*i) <- Types.Hash (hashcons i (sp-1) []);
              exec fp (sp-2*i+1) (pc+1)
| Nit i    -> (match stack.(sp-1) with
                 Types.Lis l -> (match l with
                                   []     -> exec fp (sp-1) (pc
                                     +1)
                                 | hd::tl -> stack.(sp) <- hd;
                                    stack.(sp-1) <- Types.Lis tl;
                                    exec fp (sp+1) (pc+i))
               | _           -> raise (Failure "foreach expects a
                 list"))
| Rsn i    -> stack.(sp) <- Types.Int(i) ; exec fp (sp+1) (pc+1)
| Sel s    -> stack.(sp) <- (match s with Ast.DoubleColon -> Types.Int
  (1) | _ -> Types.Int(0)); exec fp (sp+1) (pc+1)
| Slc i    -> let obj = stack.(sp-1) in
              let chaining = (hashcons i (sp-2) []) in
              let post = stack.(sp-2-2*i) in
              let restrictions = (match stack.(sp-2-2*i-1) with Types.
                Lis(l) -> l | _ as t -> raise (Failure ("Invalid
                restriction list error" ^ (Types.string_of_type t))))
                in
              let rec select obj chain post restrict parent =
                (match chain with
                  | (chain_type,chain_typed)::chain_new ->
                    let chain_value = match chain_typed with Types.Str(
                      o) -> o | _ as expr -> Types.
                      string_of_typed_expr expr in
                    (match chain_type with
                        "0" -> (match obj with
                              Types.Lis(o) ->
                                (try select (List.nth o (int_of_string (
                                  Ast.removeQuotes chain_value)))
                                  chain_new post restrict obj with
                                  Failure exp -> [])
                            | Types.Hash(o) -> let (_,next) = (List.find
                                (fun v -> chain_value = fst v) o) in
                                             select next chain_new post
                                                restrict obj
                    | _ -> [])
                | "1" -> (match obj with
                    Types.Lis(o) -> let res, _ = List.
                        fold_left
                                        (fun (acc, idx) v ->
                                          let nacc =
                                            select v (if (Ast.
                                                removeQuotes
                                                chain_value) = (
                                                string_of_int idx)
                                                 then chain_new
                                                else chain) post
                                                restrict obj
                                          in (nacc::acc, idx+1))
                                              ([], 0) o
                                      in List.concat (List.rev
                                          res)
                  | Types.Hash(o) -> let res = List.fold_left
                                         (fun acc (key, v) ->
                                           let nacc =
                                             select v (if
                                                 chain_value = key
                                                  then chain_new
                                                 else chain) post
                                                 restrict obj
                                           in (nacc::acc)) [] o
                                       in List.concat (List.rev
                                           res)
                  | _ -> [])
              | _ -> raise (Failure ("invalid selector")))
          | _ -> let check_restrictions = List.fold_left (fun
              isvalid f ->
                    try
                      stack.(sp) <- parent; stack.(sp+1) <-
                          Types.Int(-1); exec fp (sp+2) (
                          int_of_string (Types.
                          string_of_typed_expr f)); true
                    with SelectReturnException(v) -> (v &&
                        isvalid)) true restrictions in
                  if check_restrictions then (match post with
                      ">" -> [parent]
                    | _ -> [obj]) else [])
  in let out = select obj chaining (Types.
      string_of_typed_expr post) restrictions Types.Undef in
  let returnlist = (List.fold_left (fun isdouble (
      chain_type, _) -> (match chain_type with
          "1" -> true
```

```
                                  | _ -> isdouble || false)) false chaining) in
                          let result = (match returnlist  with
                                  false -> (match out with hd::tl -> hd | _ -> Types
                                      .Undef)
                                | true  -> Types.Lis(out)) in stack.(sp-(2*i+2+1))
                                   <- result; exec fp (sp-(2*i+2+1)+1) (pc+1)
  | Hlt      -> ()

  in exec 0 0 0
```

8.5 interpreter.ml

```
(*open Ast*)
open Jscanner

module NameMap = Map.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

exception ReturnException of Types.t * Types.t NameMap.t

module StringMap = Map.Make(String)

let debug_map m =
    let print_key k v = print_endline k in NameMap.iter print_key m

let rec exec toplevel funcs env = function
       Ast.Block(stmts) -> List.fold_left (exec toplevel funcs) env stmts
     | Ast.Expr(e) -> let _, env, funcs = eval env funcs toplevel e in
        env
     | Ast.If(e, s1, s2) ->
         let v, env, funcs = eval env funcs toplevel e in
         exec toplevel funcs env (if Types.toBool(v) then s1 else s2)
     | Ast.While(e, s) ->
         let rec loop env =
             let v, env, funcs = eval env funcs toplevel e in
                 if Types.toBool(v) then loop (exec toplevel funcs env s)
                     else env
         in loop env
     | Ast.Foreach(e1, e2, s) ->
       (match e1 with
         Ast.Id(id) ->
           let v, env, funcs = eval env funcs toplevel e2 in
           let rec loop env = function
               hd :: tl -> let locals , globals = env in
                           let env = if toplevel then (locals , NameMap.
                              add id hd globals) else (NameMap.add id hd
                              locals , globals) in
                            let env = exec toplevel funcs env s in
                            loop env tl
                | _          -> env
              in (match v with
                Types.Lis l -> loop env l
                    | _ -> raise (Failure "foreach expects a list"))
          | _           -> raise (Failure "foreach expects an identifier"))

      | Ast.Return(e) -> if toplevel then raise (Failure "return cannot be
          used at top level") else
          let v, (locals , globals), funcs = eval env funcs toplevel e in
          raise (ReturnException(v, globals))

and

call fdecl actuals globals funcs =
      (* Execute a statement and return an updated environment *)
      let toplevel = false
      in
          (* Enter the function: bind actual values to formal arguments *)
          let locals =
              try List.fold_left2
                  (fun locals formal actual -> NameMap.add formal actual
                      locals)
                  NameMap.empty fdecl.Ast.formals actuals
              with Invalid_argument(_) ->
                  raise (Failure ("wrong number of arguments passed to " ^
                      fdecl.Ast.fname))
          in
              (* Execute each statement in sequence, return updated global
                  symbol table *)
              snd (List.fold_left (exec toplevel funcs) (locals , globals)
                  fdecl.Ast.body)

and

eval env funcs toplevel = function
        Ast.Literal(i) -> Types.Int(i), env, funcs
      | Ast.Float(f)    -> Types.Float(f), env, funcs
      | Ast.Str(s)      -> Types.Str(s), env, funcs (*Types.Str(
          removeQuotes s), env*)
      | Ast.Boolean(b) -> Types.Boolean(b), env, funcs
      | Ast.Undef       -> Types.Undef, env, funcs
      | Ast.Id(var)     ->
          let locals , globals = env in
          if NameMap.mem var locals then
          (NameMap.find var locals), env, funcs
      else if NameMap.mem var globals then
          (NameMap.find var globals), env, funcs
      else
          raise (Failure ("undeclared identifier " ^ var))
  | Ast.Binop(e1, op, e2) ->
      let v1, env, funcs = eval env funcs toplevel e1 in
      let v2, env, funcs = eval env funcs toplevel e2 in
  (match op with
      Ast.Add     -> Types.addition v1 v2
    | Ast.Sub     -> Types.subtraction v1 v2
    | Ast.Mult    -> Types.multiplication v1 v2
    | Ast.Div     -> Types.division v1 v2
    | Ast.Equal   -> Types.typed_eq v1 v2
    | Ast.Neq     -> Types.typed_neq v1 v2
    | Ast.Less    -> Types.typed_less v1 v2
    | Ast.Leq     -> Types.typed_leq v1 v2
    | Ast.Greater -> Types.typed_greater v1 v2
    | Ast.Geq     -> Types.typed_geq v1 v2), env, funcs
    | Ast.Assign(var, expr) -> let v, (locals, globals), funcs = eval
       env funcs toplevel expr in
      if NameMap.mem var locals then
          v, (NameMap.add var v locals, globals), funcs
      else if NameMap.mem var globals then
          v, (locals, NameMap.add var v globals), funcs
      else if toplevel then
          v, (locals, NameMap.add var v globals), funcs
else
          v, (NameMap.add var v locals, globals), funcs
          (*raise (Failure ("undeclared identifier " ^ var))*)
  | Ast.Lis(list) -> let (res, env, funcs) =
     (List.fold_left (fun (acc, env, funcs) x ->
       let out, new_env, new_funcs = eval env funcs toplevel x in
       (out::acc, new_env, new_funcs)) ([], env, funcs) list) in
         Types.Lis(List.rev res), env, funcs
  | Ast.Hash(hash) -> let (res, env, funcs) =
      (List.fold_left (fun (map, env, funcs) (x,y) ->
        let outx = x (*removeQuotes x*) in
        let outy, env, funcs = eval env funcs toplevel y in
        ((outx,outy)::(List.filter ((fun new_key (m,n) -> new_key <> m
           ) outx) map), env, funcs)) ([], env, funcs) hash)
    in Types.Hash(List.rev res), env, funcs
  | Ast.Select(id, chaining, postfix, restrictions) ->
    let rec select obj chain post restrict parent env funcs =
      match chain with
        | (chain_type,chain_expr)::chain_new ->
          let chain_typed,_,_ = eval env funcs toplevel chain_expr in
let chain_value = match chain_typed with Types.Str(o) -> o |
    _ as expr -> Types.string_of_typed_expr expr in
(match chain_type with
    Ast.Colon -> (match obj with
        Types.Lis(o) ->
          (try select (List.nth o (int_of_string (Ast.
             removeQuotes chain_value))) chain_new post
             restrict obj env funcs with Failure exp -> [],
             env, funcs)
      | Types.Hash(o) -> let (_,next) = (List.find (fun v ->
          chain_value = fst v) o) in
        select next chain_new post restrict obj env funcs
      | _ -> [], env, funcs)
  | Ast.DoubleColon -> (match obj with
        Types.Lis(o) -> let res, env, funcs, _ = List.
           fold_left
                    (fun (acc, env, funcs, idx) v ->
                      let nacc, nenv, nfuncs =
                        select v (if (Ast.removeQuotes
                           chain_value) = (string_of_int
                            idx) then chain_new else
                           chain) post restrict obj env
                           funcs
                      in (nacc::acc, env, funcs, idx+1))
                          ([], env, funcs, 0) o
                  in List.concat (List.rev res), env,
                     funcs
      | Types.Hash(o) -> let res, env, funcs = List.
         fold_left
                    (fun (acc, env, funcs) (key, v) ->
                      let nacc, nenv, nfuncs =
                        select v (if chain_value = key
                           then chain_new else chain)
                           post restrict obj env funcs
                      in (nacc::acc, env, funcs)) ([],
                         env, funcs) o
                  in List.concat (List.rev res), env,
                     funcs
      | _ -> [], env, funcs))
  | _ -> let check_restrictions, env, funcs = List.fold_left
      (fun (isvalid, env, funcs) f ->
    let fdecl =
      try NameMap.find f funcs
      with Not_found -> raise (Failure ("undefined function
         " ^ f))
    in let (locals, globals) = env in
       try
```

```
                    let globals = call fdecl [parent] globals funcs
                    in (isvalid , (locals , globals), funcs)
                  with ReturnException(v, globals) -> ((Types.toBool(v)
                     && isvalid), (locals ,globals), funcs)) (true,env,
                     funcs) restrictions in
                     if check_restrictions then  (match post with
                        ">" -> [parent], env, funcs
                      | _ -> [obj], env, funcs) else [], env, funcs
          in let expr, env, funcs = eval env funcs toplevel id in
             let out, env, funcs = select expr chaining postfix
                restrictions Types.Undef env funcs in
             let returnlist = (List.fold_left (fun isdouble (
                chain_type , _) ->
               (match chain_type with
                  Ast.DoubleColon -> true
                | _ -> isdouble || false)) false chaining) in
             (match returnlist  with
                 false -> (match out with hd::tl -> hd | _ -> Types.
                    Undef), env, funcs
           | true  -> Types.Lis(out), env, funcs)
| Ast.Call("print", [expr]) -> let res, env, funcs = eval env funcs
   toplevel expr in print_endline (Types.string_of_typed_expr res);
    Types.Int(0), env, funcs
| Ast.Call("gets", [expr]) -> let res, env, funcs = eval env funcs
   toplevel expr in
                                     let res = Util.gets res in
                              res, env, funcs
| Ast.Call("parse", [expr])-> let res, env, funcs = eval env funcs
   toplevel expr in
                              let res = Util.parse res in res, env,
                                 funcs
| Ast.Call(f, actuals) ->
    (*print_endline ("call function " ^ f);*)
    let fdecl =
        try NameMap.find f funcs
        with Not_found -> raise (Failure ("undefined function " ^ f)
           )
    in (*let fdecl*)
    let actuals, env = List.fold_left
             (fun (actuals, env) actual ->
                 let v, env, funcs = eval env funcs toplevel actual
                    in v :: actuals, env)
             ([], env) (List.rev actuals)
        in (*let actuals, env*)
             (*List.map (fun a -> print_endline (Types.
                string_of_typed_expr a)) actuals;*)
             let (locals, globals) = env in
```

```
                try
                    let globals = call fdecl actuals globals funcs
                    in Types.Int(0), (locals, globals), funcs
                with ReturnException(v, globals) -> v, (locals, globals)
                    , funcs

let rec run_gstmt env funcs = function
      Ast.Stmt(s) -> let env = exec true funcs env s in env, funcs
(* (match s with
           Ast.Expr(e)      -> let res, env, funcs = eval env funcs true
                e in env, funcs
         | Ast.Block(stmts) -> List.fold_left (fun (env, funcs) e ->
              run_gstmt env funcs (Ast.Stmt e)) (env, funcs) stmts

         | _                -> raise (Failure "unsupported statement")
           (*Ast.If...*)
           (*Ast.While...*)
    ) *)
    | Ast.Func(fdecl) ->
        let funcs = NameMap.add fdecl.Ast.fname fdecl funcs in
        env, funcs

let run program =
    let env = (NameMap.empty, NameMap.empty) in
    let funcs = NameMap.empty in
    List.fold_left (fun (e, f) stmt -> run_gstmt e f stmt) (env, funcs)
        program

(*
let _ =
    let lexbuf = Lexing.from_channel stdin in
    let program = Parser.program Scanner.token lexbuf in
    let env = (NameMap.empty, NameMap.empty) in
    let funcs = NameMap.empty in
    List.fold_left (fun (e, f) stmt -> run e f stmt) (env, funcs) (List.
        rev program)
*)
```

8.6 jl.ml

```
type action = Ast | Sast | Interpret | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-i", Interpret);
                              ("-b", Bytecode);
                              ("-s", Sast);
                              ("-c", Compile) ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = List.rev (Parser.program Scanner.token lexbuf) in
  match action with
    Ast        -> let listing = Ast.string_of_program program
                  in print_string listing
  | Sast       -> ignore(Sast.check program)
  | Interpret  -> ignore(Sast.check program); ignore (Interpreter.run
     program)
        | Bytecode   -> ignore(Sast.check program);
                         let listing =
                        Bytecode.string_of_prog (Compile.translate program
                           )
                  in print_endline listing
  | Compile    -> Execute.execute_prog (ignore(Sast.check program);
     Compile.translate program)
```

8.7 jparser.mly

```
%{ open Ast %}

%token LBRACE RBRACE COMMA LBRACKET RBRACKET
%token COLON
%token TRUE FALSE
%token NULL
%token <int> LITERAL
%token <float> FLOAT
%token <bool> BOOLEAN
%token <string> STR
%token EOF

%start expr
%type <Ast.expr> expr

%%

expr:
    LITERAL             { Literal($1) }
  | STR                 { Str($1) }
  | FLOAT               { Float($1) }
  | BOOLEAN             { Boolean($1) }
  | NULL                { Undef }
  | LBRACKET exprs_opt RBRACKET { Lis($2) }
  | LBRACE kvpairs_opt RBRACE { Hash($2)  }

exprs_opt:
    /* nothing */ { [] }
  | exprs_list { List.rev $1 }

exprs_list:
    expr                   { [$1] }
  | exprs_list COMMA expr  { $3 :: $1 }

kvpairs_opt:
    /* nothing */ { [] }
  | kvpairs_list  { List.rev $1 }

kvpairs_list:
    STR COLON expr                        { [($1, $3)] }
  | kvpairs_list COMMA STR COLON expr { ($3, $5) :: $1 }
```

8.8 jscanner.mll

```
{ open Jparser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| ['\"']([^'\r' '\n' '\"']|"\\\"")*['\"'] as sc      { STR(sc) }
| '{'       { LBRACE }
| '}'       { RBRACE }
| ":"       { COLON }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ','       { COMMA }
| "null"    { NULL }
| ['0'-'9']+['.']['0'-'9']+ as lxm { FLOAT(float_of_string lxm) }
| "true"|"false" as bool { BOOLEAN(bool_of_string bool) }
| ['-']?['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
(*| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }*)
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char))
  }
```

8.9 parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET RBRACKET
%token COLON DOUBLECOLON
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE WHILE FUNCTION FOREACH THEN CONTINUE BREAK DO /*
  TO AS ELSEIF FOR */
%token TRUE FALSE
%token NULL
```

```
/*%token PUTS GETS*/
%token <int> LITERAL
%token <float> FLOAT
%token <bool> BOOLEAN
%token <string> ID
%token <string> STR
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
/*%left EQ NEQ
%left LT GT LEQ GEQ*/
%nonassoc EQ NEQ
%nonassoc LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
    /* nothing */ { [] }
  | program gstmt { $2 :: $1 }

gstmt:
    stmt  { Stmt($1) }
  | fdecl { Func($1) }

fdecl:
   FUNCTION ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
      { { fname = $2;
              formals = $4;
              locals = [];
              body = List.rev $7 } }

formals_opt:
     /* nothing */ { [] }
   | formal_list   { List.rev $1 }

formal_list:
     ID                     { [$1] }
   | formal_list COMMA ID { $3 :: $1 }

stmt_list:
     SEMI /* nothing */  { [] }
   | stmt                { [$1] }
   | stmt_list stmt      { $2 :: $1  }
   | stmt_list SEMI      { $1 }

stmt:
     expr SEMI { Expr($1) }
   | RETURN expr SEMI { Return($2) }
   | LBRACE stmt_list RBRACE { Block(List.rev $2) }
   | IF expr THEN stmt %prec NOELSE { If($2, $4, Block([])) }
   | IF expr THEN stmt ELSE stmt    { If($2, $4, $6) }
   /*| FOR expr TO expr DO stmt
      { For($2, $4, $6) }*/
   | FOREACH ID COLON expr DO stmt  { Foreach(Id($2), $4, $6) }
   | WHILE expr DO stmt  { While($2, $4) }

selector_chain:
   selector_list { List.rev $1 }

selector_list:
   selector { [$1] }
   | selector_list selector { $2::$1 }

selector:
   selector_type STR { ($1, Str($2)) }
   | selector_type ID { ($1, Id($2)) }

selector_postfix_opt:
     /* nothing */ { "" }
   | GT { ">" }

selector_restriction_opt:
     /* nothing */ { [] }
   | COMMA selector_restriction_list { List.rev $2 }

selector_restriction_list:
     ID { [$1] }
   | selector_restriction_list COMMA ID { $3::$1 }

selector_type:
   COLON { Colon }
   | DOUBLECOLON { DoubleColon }

expr:
     LITERAL           { Literal($1) }
   | ID                { Id($1) }
   | STR               { Str(removeQuotes $1) }
   | FLOAT              { Float($1) }
   | BOOLEAN            { Boolean($1) }
   | expr PLUS   expr { Binop($1, Add,    $3) }
   | expr MINUS  expr { Binop($1, Sub,    $3) }
   | expr TIMES  expr { Binop($1, Mult,   $3) }
   | expr DIVIDE expr { Binop($1, Div,    $3) }
   | expr EQ     expr { Binop($1, Equal, $3) }
   | expr NEQ    expr { Binop($1, Neq,    $3) }
   | expr LT     expr { Binop($1, Less,   $3) }
   | expr LEQ    expr { Binop($1, Leq,    $3) }
   | expr GT     expr { Binop($1, Greater,  $3) }
   | expr GEQ    expr { Binop($1, Geq,    $3) }
   | MINUS expr         { Binop($2, Mult,  Literal(-1)) }
   | NULL               { Undef }
   | ID ASSIGN expr     { Assign($1, $3) }
   | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
   /*| GETS LPAREN actuals_opt RPAREN { Call("gets", $3) }*/
   | LPAREN expr RPAREN { $2 }
   | ID LBRACKET selector_chain selector_postfix_opt
      selector_restriction_opt RBRACKET { Select(Id($1), $3, $4, $5) }
   | LBRACKET exprs_opt RBRACKET { Lis($2) }
   | LBRACE kvpairs_opt RBRACE { Hash($2)  }

actuals_opt:
     /* nothing */ { [] }
   | actuals_list  { List.rev $1 }

actuals_list:
     expr                    { [$1] }
   | actuals_list COMMA expr { $3 :: $1 }

exprs_opt:
     /* nothing */ { [] }
   | exprs_list { List.rev $1 }

exprs_list:
     expr                    { [$1] }
   | exprs_list COMMA expr   { $3 :: $1 }

kvpairs_opt:
     /* nothing */ { [] }
   | kvpairs_list  { List.rev $1 }

kvpairs_list:
     STR COLON expr                        { [($1, $3)] }
   | kvpairs_list COMMA STR COLON expr { ($3, $5) :: $1 }
```

8.10 sast.ml

```
exception UninitializedVariable of string
exception FunctionNotDefined of string
exception MultipleDefinition of string
exception UnboundVariable of string

type expression = Int | Float | Str | Boolean | Lis | Hash | Any |
   Unbound

type symbol_table = {
  parent : symbol_table option;
  variables : (string * expression) list
}


(***from compile.ml**)
module NameMap = Map.Make(struct
    type t = string
    let compare x y = Pervasives.compare x y
end)

module StringMap = Map.Make(String)

(*
  implement the scope rule
  an assignment at toplevel creates a global variable if it is not
     defined yet.
  an assignment at a fun_decl, however, creates a local variable when
     the left
  hand side is neither a global nor formal let variable.
  a formal variable shadows a global variable of the same name
*)
let rec env_expr env formals toplevel = function
      Ast.Assign(var, expr) -> let globals, locals, funcs = env in
                               if List.mem var formals || List.mem var
                                  globals then
                                   env
                               else if toplevel then
                                   var :: globals, locals, funcs
                               else
                                   globals, var :: locals, funcs
    | _                      -> env

and env_stmt env formals toplevel = function
      Ast.Expr(e)        -> let env = env_expr env formals toplevel e in
         env
    | Ast.Block(stmts)   -> List.fold_left (fun env s -> env_stmt env
   formals toplevel s) env stmts
 | Ast.If(e,s1,s2)    -> let env = env_expr env formals toplevel e in
                         let env = env_stmt env formals toplevel s1 in
                         let env = env_stmt env formals toplevel s2 in
                         env
 | Ast.While(e,s)     -> let env = env_expr env formals toplevel e in
                         let env = env_stmt env formals toplevel s in
                            env
 | Ast.Foreach(e1,e2,s) -> let globals, locals, funcs = env in
                         let env = (match e1 with
                             Ast.Id id -> if toplevel then id::globals
                                , locals, funcs
                                                 else globals, id ::
                                                    locals, funcs





                                     in env_stmt env formals toplevel s
 | _                  -> env
```

```
and env_gstmt env = function
      Ast.Stmt(s)      -> let env = env_stmt env [] true s in env
    | Ast.Func(fdecl) -> let env = env_fdecl env fdecl.Ast.formals fdecl
        in env

and env_fdecl env formals fdecl =
    let _,_,funcs = env in
      if NameMap.mem fdecl.Ast.fname funcs then raise (
          MultipleDefinition fdecl.Ast.fname)
      else
        let globals, localvars, funcs = List.fold_left (fun env s ->
            env_stmt env formals false s) env fdecl.Ast.body in
        let funcs = NameMap.add fdecl.Ast.fname {fdecl with Ast.locals=
            localvars} funcs in
        globals, [], funcs
(**end compile.ml**)

let fun_exists f funcs =
        try
                ignore(List.find (fun e -> e.Ast.fname = f) funcs); true
        with Not_found -> raise (FunctionNotDefined f)

let rec find_variable (scope : symbol_table) name =
          try
            List.find (fun (s, _) -> s = name) scope.variables
          with Not_found ->
            match scope.parent with
              Some(parent) -> find_variable parent name
            | _ -> raise (UnboundVariable name)

(* TODO add a variable to symble table *)
let rec add_variable env (name, t) =
  let sym = fst env in
  let funcs = snd env in
  let sym = {parent = sym.parent; variables = (name, t) :: sym.variables
      } in
  (sym, funcs)

let rec expr ((sym, funcs) as env) toplevel = function
    Ast.Literal(i) -> env, Int
  | Ast.Float(f)   -> env, Float
  | Ast.Str(s)     -> env, Str
  | Ast.Boolean(b) -> env, Boolean
  | Ast.Lis(l)     -> env, Lis
  | Ast.Hash(h)    -> env, Hash
  | Ast.Id(var)    -> let _, t = find_variable sym var in env, t(*TODO
      perform lookup in env, raise an exception if var is uninitialized*)
    | Ast.Binop(e1, op, e2) -> let env, _ = expr env toplevel e1 in let
      env, _ = expr env toplevel e2 in env, Any (*TODO*)
    | Ast.Assign(var, e) -> let env, t = expr env toplevel e in let env =
      add_variable env (var, t) in env, Any (*TODO add var to env with
      the type of expr*)
        | Ast.Select(var,_,_,_) -> env, Any (*TODO var has to be a list,
          a hash, or an "any"*)
    | Ast.Call(f, actuals)  -> ignore(fun_exists f funcs); env, Any (*TODO
      return type of f?*)
    | Ast.Undef             -> env, Unbound (* TODO Shall we raise an
      exception *)

(*TODO the "intersection" of two enviroment*)
let merge env1 env2 =
  let sym1 = fst env1 in
  let sym2 = fst env2 in
  let v = List.filter (fun x -> List.mem x sym2.variables) sym1.
      variables in
  ({parent = sym1.parent; variables = v}, snd env1)

let rec stmt env toplevel = function
    Ast.Block(stmts) -> List.fold_left (fun e s -> stmt env toplevel s)
        env stmts
  | Ast.Expr e         -> let env, _ = expr env toplevel e in env
  | Ast.If(e, s1, s2) -> (*how to handle if statement*)
                               let env, _ = expr env toplevel e in
                       let env1 = stmt env toplevel s1 in
                       let env2 = stmt env toplevel s2 in
                       merge env1 env2
  | Ast.While(e, s)   -> let env1,_ = expr env toplevel e in
                       let env2 = stmt env1 toplevel s in
                       merge env1 env2
  | Ast.Return(e)     -> (*TODO if all return statements agree in type,
      put the return type to env*)
                       env
  | Ast.Foreach(e1, e2, s) -> let id = (match e1 with Ast.Id id -> id |
      _ -> raise (Failure "foreach expects an identifier")) in
                               let env, t = expr env toplevel e2 in
                               let env1 = add_variable env (id, Any) in
                               let env2 = stmt env1 toplevel s in
                               merge env1 env2;;

let func env toplevel f =
  let func = List.find (fun x -> x.Ast.fname = f.Ast.fname) (snd env) in
  let vars = List.fold_left (fun l v -> (v, Any) :: l) [] func.Ast.
      formals in
  let vars = List.fold_left (fun l v -> (v, Any) :: l) vars func.Ast.
     locals in
        let sym = {parent = Some((fst env)); variables = vars} in
  stmt (sym, (snd env)) toplevel (Ast.Block f.Ast.body)

let rec gstmt env = function
    Ast.Stmt s -> stmt env true s
  | Ast.Func f -> func env false f;;

let check program =
  let functions = NameMap.add "print" {Ast.fname="print"; Ast.formals
     =[]; Ast.locals=[]; Ast.body=[]} NameMap.empty in
  let functions = NameMap.add "gets" {Ast.fname="gets"; Ast.formals=[];
     Ast.locals=[]; Ast.body=[]} functions in
  let functions = NameMap.add "parse" {Ast.fname="parse"; Ast.formals
     =[]; Ast.locals=[]; Ast.body=[]} functions in
        let (globals, _, functions) = List.fold_left env_gstmt ([], [],
           functions) program in
  let funcs = NameMap.fold (fun k v a -> v :: a) functions [] in
  let sym = {parent = None; variables = List.fold_left (fun l v -> (v,
     Any) :: l) [] globals} in
        (*let funcs = List.fold_left (fun s e -> match e with
                   Ast.Stmt st -> s
    | Ast.Func f -> if List.mem f.Ast.fname s then raise (
       MultipleDefinition f.Ast.fname) else f.Ast.fname :: s) [] program
        in*)
        List.fold_left (fun env gs -> gstmt env gs) (sym, funcs) program
```

8.11 scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| ['\"']([^'\r' '\n' '\"']|"\\\"")*['\"'] as sc     { STR(sc) }
|  '('       { LPAREN }
|  ')'       { RPAREN }
|  '{'       { LBRACE }
|  '}'       { RBRACE }
|  ';'       { SEMI }
| ":"        { COLON }
| "::"       { DOUBLECOLON }
|  '['       { LBRACKET }
|  ']'       { RBRACKET }
|  ','       { COMMA }
|  '+'       { PLUS }
|  '-'       { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '='        { ASSIGN }
| "=="       { EQ }
| "!="       { NEQ }
| '<'        { LT }
| "<="       { LEQ }
| ">"        { GT }
| ">="       { GEQ }
| "null"     { NULL }
| "if"       { IF }
| "else"     { ELSE }
(*| "for"      { FOR } *)
| "foreach" { FOREACH }
| "while"   { WHILE }
| "return"  { RETURN }
| "break"      { BREAK }
| "continue" { CONTINUE }
| "then"       { THEN }
(*| "to"          { TO }
| "elseif"    { ELSEIF } *)
| "do"         { DO }
(*| "as"          { AS } *)
(*| "puts"        { PUTS }
| "gets"       { GETS }*)
| "fun"        { FUNCTION }
| ['-']?['0'-'9']+['.']['0'-'9']+ as lxm { FLOAT(float_of_string lxm) }
| "true"|"false" as bool { BOOLEAN(bool_of_string bool) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char))
   }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.12 types.ml

```
type t =
   Undef (* for compiler. when all local variables are calculated in
      advance, a variable is undefined before the decl*)
 | Int of int
 | Float of float
 | Str of string
 | Boolean of bool
 | Lis of t list
 | Hash of (string*t) list

let string_of_type = function
      Int(v) -> "Integer"
    | Float(v) -> "Float"
    | Str(v) -> "String"
    | Boolean(v) -> "Boolean"
    | Lis(v) -> "List"
    | Hash(v) -> "Map"
    | Undef -> "Undefined"

let addition a b =
    match a, b with
      Int(i1), Int(i2)     -> Int (i1 + i2)
    | Float(f1), Float(f2) -> Float (f1 +. f2)
    | Int(i1), Float(f2)   -> Float ((float_of_int i1) +. f2)
    | Float(f1), Int(i2)   -> Float (f1 +. (float_of_int i2))
    | Str(s1), Str(s2)     -> Str(s1 ^ s2)
    | Str(s1), Int(i2)     -> Str(s1^ (string_of_int i2))
    | Int(i1), Str(s2)     -> Str((string_of_int i1) ^ s2)
    | Str(s1), Float(f2)   -> Str(s1 ^ (string_of_float f2))
    | Float(f1), Str(s2)   -> Str((string_of_float f1) ^ s2)
    | Hash(f1), Hash(f2)   -> Hash(f1 @ f2) (*more work here to preserve
        the uniqness of the keys*)
    | Lis(l1), Lis(l2)     -> Lis(l1 @ l2)
    | _ -> raise(Failure("can't add " ^ string_of_type a ^ " and "  ^
       string_of_type b))


let subtraction a b =
    match a, b with
      Int(i1), Int(i2)     -> Int (i1 - i2)
    | Float(f1), Float(f2) -> Float (f1 -. f2)
    | Int(i1), Float(f2)   -> Float ((float_of_int i1) -. f2)
    | Float(f1), Int(i2)   -> Float (f1 -. (float_of_int i2))
    | Lis(l1), Lis(l2) -> Lis(List.filter (fun x -> not (List.mem x l2))
        l1)
    | _ -> raise(Failure("can't subtract " ^ string_of_type a ^ " and "
        ^ string_of_type b))

let multiplication a b =
    match a, b with
      Int(i1), Int(i2)     -> Int (i1 * i2)
    | Float(f1), Float(f2) -> Float (f1 *. f2)
    | Int(i1), Float(f2)   -> Float ((float_of_int i1) *. f2)
    | Float(f1), Int(i2)   -> Float (f1 *. (float_of_int i2))
    | _ -> raise(Failure("can't multiply " ^ string_of_type a ^ " and "
        ^ string_of_type b))

let division a b =
    match a, b with
      Int(i1), Int(i2)     -> Int (i1 / i2)
    | Float(f1), Float(f2) -> Float (f1 /. f2)
    | Int(i1), Float(f2)   -> Float ((float_of_int i1) /. f2)
    | Float(f1), Int(i2)   -> Float (f1 /. (float_of_int i2))
    | _ -> raise(Failure("can't divide " ^ string_of_type a ^ " and "  ^
        string_of_type b))


let toBool a = match a with
      Int(v)      -> v != 0
    | Float(v)    -> v != 0.0
    | Str(v)      -> v != ""
    | Boolean(v) -> v
    | Undef -> false
    | _ -> raise(Failure(string_of_type a ^ " can't is an invalid
       boolean expression"))

let typed_eq v1 v2 = match v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 = i2)
    | Float(f1), Float(f2) -> Boolean(f1 = f2)
    | Str(s1), Str(s2) -> Boolean(s1 = s2)
    | Boolean(b1), Boolean(b2) -> Boolean(b1 = b2)
    | _ -> Boolean(false)

let typed_neq v1 v2 = match v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 <> i2)
    | Float(f1), Float(f2) -> Boolean(f1 <> f2)
    | Str(s1), Str(s2) -> Boolean(s1 <> s2)
    | Boolean(b1), Boolean(b2) -> Boolean(b1 <> b2)
    | _ -> Boolean(true)

let typed_less v1 v2 =   match v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 < i2)
    | Float(f1), Float(f2) -> Boolean(f1 < f2)
    | _ -> Boolean(false)

let typed_leq v1 v2 = match  v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 <= i2)
    | Float(f1), Float(f2) -> Boolean(f1 <= f2)
    | _ -> Boolean(false)

let typed_greater v1 v2 = match v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 > i2)
    | Float(f1), Float(f2) -> Boolean(f1 > f2)
    | _ -> Boolean(false)

let typed_geq v1 v2 = match v1, v2 with
      Int(i1), Int(i2) -> Boolean(i1 >= i2)
    | Float(f1), Float(f2) -> Boolean(f1 >= f2)
    | _ -> Boolean(false)


let rec string_of_typed_expr = function
      Int(v)      -> string_of_int v
    | Float(v)    -> string_of_float v
    | Str(v)      -> v (*"\""^v^"\""*)
    | Boolean(v) -> string_of_bool v
    | Lis(v)      -> "[" ^ String.concat "," (List.map
       string_of_typed_expr v) ^ "]"
    | Hash(v)    -> let string_of_kvpair = fun (k, v) -> k ^ ":" ^
       string_of_typed_expr v in "{" ^ String.concat "," (List.map
       string_of_kvpair v)  ^ "}"
    | Undef      -> "null"
```

8.13 util.ml

```
(*utility functions*)

let rec t_of_jexpr = function
    Ast.Literal i    -> Types.Int i
  | Ast.Str s        -> Types.Str s
  | Ast.Float f      -> Types.Float f
  | Ast.Boolean b    -> Types.Boolean b
  | Ast.Lis l        -> Types.Lis (List.map t_of_jexpr l)
  | Ast.Hash h       -> Types.Hash (List.map (fun (s, e) -> (s,
     t_of_jexpr e)) h)
  | Ast.Undef        -> Types.Undef
  | _                -> raise (Failure "Expression is not valid json")

let gets = function
    Types.Str s ->
      let ic, oc = Unix.open_process ("curl -s " ^ s) in
      let buf = Buffer.create 16 in
      (try
        while true do
          Buffer.add_channel buf ic 1
        done
      with End_of_file -> ());
    let _ = Unix.close_process (ic, oc) in
    Types.Str (Buffer.contents buf)
  | _            -> raise (Failure "Parameter has to be a string")
```

```
let parse t = match t with
    Types.Str s -> let lexbuf = Lexing.from_string s in
                   let expr = Jparser.expr Jscanner.token lexbuf in
                      t_of_jexpr expr
  | _           -> raise (Failure "Parameter has to be a string")
```

8.14 TestScript.sh

```
#!/bin/bash

failedTests=0

echo "Running Tests"
for test in *.jl
do
        output=$(../jl $1 < "$test" 2>&1)
        correct=$(cat "$test".output)
        if [ "$output" != "$correct" ]
        then
            failedTests=$(($failedTests+1))
            echo "Failed $test. Expected '$correct' but got '$output'"
        fi
done

if [ $failedTests -gt 0 ]
then
    echo "Failed $failedTests test(s)"
else
    echo "All tests passed!"
fi
```