CS 331 – Assignment 7
Due: 11:59PM, Thursday, April 13

Download your starter code from D2L in the file `a7.zip`. Unzipping it will create a directory called `a7` containing the following files:

- `interpreter.html`: This file is the web page that makes up the GUI of the interpreter. You may NOT modify this file for this assignment.
- `grammar.html`: This file is a web page (linked off of the previous page) that displays the grammar for SLang 1, not including the additions that you must complete for this assignment. You may NOT modify this file for this assignment.
- `scripts`: This is a directory that contains all of the JavaScript files that make up the interpreter, namely:
  - `init.js`: This file contains some helper code that you do not need to worry about for this assignment. You may NOT modify this file for this assignment.
  - `samples.js`: This file contains a collection of test programs that you can use to test your modified interpreter. You may NOT modify this file for this assignment.
  - `grammar.jison`: This Jison file contains the BNF grammar for SLang 1 that we discussed in class.
  - `absyn.js`: This file contains the implementation of the abstract syntax for SLang 1 that we discussed in class.
  - `env.js`: This file contains the implementation of the environment for SLang 1 that we discussed in class.
  - `interpreter.js`: This file contains the implementation of the interpreter for SLang 1 that we discussed in class.
  - `fp.js`: This file implements the `fp` module that we used in the section of the course on functional programming. You may find it convenient to use its map function when you build a mapping capability into the interpreter in the last problem.

  In this assignment, you only are allowed to modify absyn.js, grammar.jison, env.js, and interpreter.js. Those are the only four files you will be handing in.

Assignment overview

For each problem below, start with a working version of the interpreter (either the code handout or your modified version containing the answers to the previous problems) and add the features described in that problem. You will save a lot of time and effort if you design your solution prior to coding. Assuming that you fully understand the current code in each file of the code base, think about the changes that you need to make in each file. Design the code that each change implies. Then implement these changes using your favorite text editor. Finally, test your code. For this last step, you will be working in a browser. Chrome has an excellent debugger, so that would be my recommendation. For testing your interpreter, simply open the file `interpreter.html` in the browser, type your sample program in the topmost text box, and then click the button between the two text boxes. If your interpreter works as expected, its output will be shown in the bottom-most text box. You can also click on the "Show Test Programs" button to display a few test cases provided for your convenience. Clicking the "Interpret" button to the left of a test case will load its program into the topmost text box, will automatically invoke the interpreter on it, and will display the output of the interpreter in the bottom-most text box.

If your interpreter does not work as expected, you should use the browser's developer tools. They are under `View > Developer > Developer Tools` in Chrome. You can use the console to check for any syntax errors that occurred while loading the page or if you use print statements (i.e., `console.log(___)`) to debug your code. If needed, use the debugger (under the `Sources` tab in Chrome) to set breakpoints and trace the execution of the interpreter step by step.

One extra tool you can use to debug your JavaScript code is `jshint`, which is installed on the Linux lab machines. Typing `jshint <filename>.js` will typically output a bunch of warnings or errors that will help you find typos and other bugs in your code. For more info, see `http://jshint.com`.

Finally, when you think that you are done and ready to submit, you can click on the "Run Test Suite" button to run all provided test programs. Their output will be sent to the console. You must pass all of these tests (as well as others that I'll use during grading) to get full credit. Note that three of the programs in the provided test suite intentionally generate a run-time error.

Problem statements

1. Complete the suite of arithmetic functions in SLang 1 by adding subtraction (`-`), quotient (`/`), remainder (`%`), and unary minus (`~`) as primitive operations. When completed, the following SLang program:

   ```
   (fn (n,p,q) => /(-(+(~(n), 20), p) , %(q, 3)) 10 2 11)
   ```

   will return the value:

   ```
   ["Num",4]
   ```

   Note that you should use prefix notation for each of the added functions `-, / %, ~`. To complete this problem, you must perform the following modifications:

absyn.js Create two sets of functions (with suffix 1 and 2, respectively) to replace the existing set of functions pertaining to `prim_app_exp`. This change implies that we will now be implementing, within the SLang grammar, a check on the number of operands for each primitive operator. More specifically, the AST nodes for these two types of primitive applications will change since, instead of representing the operands as an array in a single node, the unary-application nodes will have one child for their only operand while the binary-application nodes will have two children representing their two operands. These two types of nodes will NOT use an array containing the operands.

grammar.jison Modify the grammar to reflect the new concrete syntax for primitive operators as well as the constraint on the number of their operands.

interpreter.js Modify the `applyPrimitive` function (but NOT its signature in the API) to accommodate the new primitive operators you implement here and in the next problem. Furthermore, modify the `evalExp` function to interpret the `Prim1AppExp` and `Prim2AppExp` expressions as two separate cases.

wherever necessary Add subtraction (" - "), division (" / "), the remainder operation (" % "), and the unary negation operator (" ~ ") to SLang 1. Note that the last operator is not a JavaScript operator. Instead, you will implement it using JavaScript's (unary) minus sign.

2. Add Booleans as a new type of values in our language. Then, add less-than (<), greater-than (>), and equal-to (==, that is, two equal signs) as binary primitive operators that are applied to two integer arguments. Furthermore, add `!` as a unary primitive operator that is applied to a Boolean value. These Boolean operators return a new kind of denoted value, namely `["Bool",true]` and `["Bool",false]`, where `true` and `false` are the JavaScript constants, so that Boolean expressions can be used with the conditional expression described in the next problem. However, the Boolean constants `true` and `false` should NOT be added to the concrete syntax of SLang. Sample programs are as follows:

```
==(1, %(/(100 , 4) , 3) )

!( >(/(11, 4), -(30, %(25, 13))) )
```

The output for each one of these two programs is: `["Bool",true]`.

3. Add a conditional expression to our language, with the following syntax:

```
...
<exp>     ::= ... | <cond_exp>
<cond_exp> ::= ?(exp, exp, exp)
...
```

Modify all necessary files. The first expression in the conditional should evaluate to a `Bool` value. If it is `true`, return the value of the second expression. If it is `false`, return the value of the third expression. If it is not a Boolean value, raise an exception (see exact error message below). Here are some sample programs and their output:

(a) `(fn (n,p,q) => ?(n, +(p, q), *(p, q)) <(6, 1) 2 3)`
Output: `["Num",6]`

(b) `(fn (n,p,q) => ?(n, +(p, q), *(p, q)) >(6, 1) 2 3)`
Output: `["Num",5]`

(c) `(fn (n,p,q) => ?(n, +(p, q), /(q, 0)) 1 2 3)`
Output: First, pops up an alert window with the following message:
`"Error: Interpreter error: The argument of getBoolValue is not a Bool value."`
Then returns the value: `No output [Runtime error]`

(d) `(fn (n,p,q) => ?(n, +(p, q), /(q, 0)) >(6, 1) 2 3)`
Output: `["Num",5]`

As shown in the last test above, you must short-circuit the evaluation of the conditional expression so that only the second or the third expression in the conditional is actually evaluated, but not both.

4. In this problem, you must add some support for lists in SLang 1. More precisely, you need only support flat lists of integers. Lists will have the concrete syntax illustrated by the following two examples: `[]` (the empty list) and `[1,2,3]`. As evidence that you have added lists as a denoted value in the environment, write a new **sumlist** primitive operator that returns the sum of all the elements in a list. Here are a few sample programs and their output:

(a) `[]` Output: `["List",[]]`

(b) `[1]` Output: `["List",[1]]`

(c) `[1,2,3,4,5]` Output: `["List",[1,2,3,4,5]]`

(d) `( fn (x) => sumlist(x) [1,2,3] )` Output: `["Num",6]`

To complete this problem, you must add a new `list_exp` expression to the abstract syntax and a new `List` value in the environment. Finally, in the interpreter, besides the handling of the new `list_exp` expression, the only changes needed are in the `applyPrimitive` function, in which you should call a function that you develop to compute the sum of the elements in a list. Inside the interpreter this list's value will just be a JavaScript array of numbers, so it should be easy to walk through it and accumulate the sum of the numbers in it.

5. In this problem, you must add to SLang 1 the `map` function as a primitive, binary operation with `map` as its symbol. The first operand of `map` must be an expression that evaluates to a function and its second operand must be an expression that evaluates to a list of integers. To keep things simple in this problem, we can assume that the first operand is always a function of one number that also returns a number. Here are a few sample programs and their output:

   (a) `map(fn(x) => add1(x), [ ])` Output: `["List",[]]`

   (b) `map(fn(x) => x, [1,2,3,5,6])` Output: `["List",[1,2,3,5,6]]`

   (c) `map(fn(x) => *(2, x), [1,2,3,5,6])` Output: `["List",[2,4,6,10,12]]`

   (d) `map(fn(x) => *(x, x), [1,2,3]` Output: `["List",[1,4,9]]`

   (e) `map((fn(x) => fn (y) => +(x, y) 10), [1,2,3,5,6])` Output: `["List",[11,12,13,15,16]]`

   (f) `(fn (f,list) => map(f, list) x [1,2,3])`
   Output:
   First, pops up an alert window with the message:
   ``Error: The first argument of 'map' has the wrong type.''
   Then returns the value:
   `No output [Runtime error]`

   (g) `map(fn(x) => x, 1)`
   Output:
   First, pops up an alert window with the message:
   ``Error: The second argument of 'map' has the wrong type.''
   Then returns the value:
   `No output [Runtime error]`

   After adding `map` to the grammar in the JISON file, you will only have to modify the interpreter file. The new case that you must add to the `applyPrimitive` function will resemble the existing cases. However, instead of using a built-in JavaScript operator, it will call a new function called `applyMap` that you must design and implement in the `interpreter.js` file. Here are some hints for the design and implementation of the `applyMap` function:

   - The only argument (say, `args`) of `applyMap` is an array containing a closure and a list of integers. This array is identical to the `args` argument of the `applyPrimitive` function.
   - Declare a local variable `f` and initialize it to the first element of `args`.
   - Declare a local variable `list` and initialize it to the integer list inside the second element of `args`.
   - Declare a local variable `mapFunction` and initialize it to a function of an integer that returns the result of applying `f` to one element of `list`. This is the step that will require the most forethought on your part.
   - Finally, return the list obtained by calling `fp.map` with `mapFunction` and `list` as arguments. Note that the return value of `applyMap` must be a denoted value of type `List`.

How to submit your work:

When you are done, put the four files *grammar.jison, absyn.js, env.js*, and *interpreter.js* in a zip file called *a7.zip*, and submit it to the dropbox I have set up for Assignment 7. The zip file should not contain a folder (directory) with these four files – just the four JavaScript files. Any submission that does not consist of one zip file with *exactly* four files (and no folders) in it will receive a 10-point penalty.