CSCI 3155: Lab Assignment 3

Fall 2012: Due Saturday, October 6, 2012

The purpose of this lab is to grapple with how dynamic scoping arises and how to formally specifying semantics. Concretely, we will extend JAVA SCRIPTY with recursive functions and implement two interpreters. The first will be a big-step interpreter that is an extension of Lab 2 but implements dynamic scoping “by accident.” The second will be a small-step interpreter that exposes evaluation order and implements static scoping by substitution.

Like last time, find a partner. You will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment.

You are welcome to talk about these questions in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus. Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expression computation currently unfamiliar to you. Finally, make sure that your file compiles and runs (using Scala 2.9.2). A program that does not compile will not be graded.

Submission Instructions. First, submit your .scala file to the auto-testing system to get instant feedback on your code. You may submit as many times as you want, but please submit at least once before the deadline. We will only look at your last submission. A link to the auto-testing system is available on the moodle.

Then, upload to the moodle exactly two files named as follows:

- Lab3_YourIdentiKey.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- Lab3_YourIdentiKey.scala with your answers to the coding exercises

Replace YourIdentiKey with your IdentiKey. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting Started. Download the code template Lab3.scala from the assignment page. Be sure to also write your name, your partner, and collaborators there.

1. Feedback. Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.
expressions 
\[
e := x | n | b | \text{undefined} | \text{uop} e_1 | e_1 \text{ bop } e_2 | e_1 ? e_2 : e_3 \\
| \text{const } x = e_1 ; e_2 | \text{ jsy.print}(e_1)
\]
values 
\[
v := n | b | \text{undefined} | \text{str} | \text{function } p(x) e_1 | \text{typeerror}
\]
unary operators 
\[
uop ::= - | !
\]
binary operators 
\[
bop ::= , | + | - | * | / | === | !== | < | <= | > | >= | && | ||
\]
variables 
\[
x
\]
numbers (doubles) 
\[
n
\]
booleans 
\[
b ::= \text{true} | \text{false}
\]
strings 
\[
str
\]
function names 
\[
p ::= x | \varepsilon
\]
value environments 
\[
E ::= \cdot | E[x \mapsto v]
\]

Figure 1: Abstract Syntax of JAVASCRIPTY

2. **JavaScripty Interpreter: Tag Testing, Recursive Functions, and Dynamic Scoping**

We now have the formal tools to specify exactly how a JAVASCRIPTY program should behave. Unless otherwise specified, we will continue to try to match JavaScript semantics as implemented by Node.js/Google’s V8 JavaScript Engine. Thus, it is still useful to write little test JavaScript programs and run it through Node.js to see how the test should behave. Finding bugs in the JAVASCRIPTY specification with respect to JavaScript is certainly deserving of extra credit.

In this lab, we extend JAVASCRIPTY with strings and recursive functions. This language is very similar to the LETREC language in Section 3.4 of Friedman and Wand.

For this question, we try to implement functions as an extension of Lab 2 in the most straightforward way. What we will discover is that we have made a historical mistake and ended up with a form of dynamic scoping.

The syntax of JAVASCRIPTY for this lab is given in Figure 1. Note that the grammar specifies the abstract syntax using notation borrowed from the concrete syntax. The new constructs are highlighted. We have string literals str, function expressions function \( p(x) e_1 \), and function calls \( e_1(e_2) \).

In a function expression, the function name \( p \) can either be an identifier or empty. The identifier is used for recursion. For simplicity, all functions are one argument functions. Since functions are first-class values, we can get multi-argument functions via currying.

We have also added a “marker” typeerror to the expression and value languages. These are not part of the source language but arise during evaluation. Its purpose will be explained further below.

As before, the concrete syntax accepted by the parser is slightly less flexible than the abstract syntax in order to match the syntactic structure of JavaScript. For function expressions, the body is surrounded by curly braces (i.e., { }) and consists of a statement \( s_1 \) for const bindings followed by a return with an expression \( e_1 \).
An abstract syntax tree representation is provided for you in `ast.scala`. We also provide a parser and main driver for testing. The correspondence between the concrete syntax and the abstract syntax representation is shown in Figure 3.

A big-step operational semantics of JAVASCRIPTY is given in Figure 4. This figure may be one of the first times that you are reading a formal semantics of a programming language. It may seem daunting at first, but it will be become easier with practice. This lab is such an opportunity to practice.

A formal semantics enables us to describe the semantics of a programming language clearly and concisely. The initial barrier is getting used to the meta-language of judgment forms and inference rules. However, once you cross that barrier, you will see that we are telling you exactly how to implement the interpreter—it will almost feel like cheating!

In Figure 4, we define the judgment form \( E \vdash e \Downarrow v \), which says informally, “In value environment \( E \), expression \( e \) evaluates to value \( v \).” This relation has three parameters: \( E \), \( e \), and \( v \). You can think of the other parts of the judgment as just punctuation. This judgment form corresponds directly to the `eval` function that we are asked to implement (not a coincidence). It similarly has three parts:

```scala
def eval(env: Env, e: Expr): Expr
```

It takes as input a value environment \( env (E) \) and an expression \( e (e) \) returns a value \( v \).

It is very informative to compare your Scala code for `env` with the inference rules that define \( E \vdash e \Downarrow v \). One thing you should observe is that all of the rules are implemented, except `EvalPlusString1`, `EvalPlusString2`, and `EvalCall`. In essence, implementing those rules is your task for this question.

In Lab 2, all expressions could be evaluated to something (because of conversions). With functions, we encounter one of the very few run-time errors in JavaScript: trying to call something that is not a function. In JavaScript and in JAVASCRIPTY, calling a non-function raises a run-time error. In the formal semantics, we model this with evaluating to the “marker” `typeerror`.

Such a run-time error is known a dynamic type error. Languages are called *dynamically*
Figure 4: Big-step operational semantics of JAVASCRIPTY (with dynamic scoping).
toNumber(n) \def n
\text{toNumber(true)} \def 1
\text{toNumber(false)} \def 0
\text{toNumber(undefined)} \def 0
\text{toNumber(str)} \text{parse str \text{NaN}}

toBoolean(n) \def \text{false} \text{if } n = 0 \text{ or } n = \text{NaN}
toBoolean(true) \def \text{true} \text{otherwise}
toBoolean(false) \def b
\text{toBoolean(undefined)} \def \text{false} \text{if } \text{str = ""}
\text{toBoolean(str)} \def \text{true} \text{otherwise}
\text{toBoolean(function } p(x) e_1) \def \text{true}

\text{toString(n)} \def \text{string of } n
\text{toString(true)} \def \text{"true"}
\text{toString(false)} \def \text{"false"}
\text{toString(undefined)} \def \text{"undefined"}
\text{toString(str)} \def \text{str}
\text{toString(function } p(x) e_1) \def \text{"function"}

\text{Figure 5: Conversion functions. We do not specify explicitly the parsing or string conversion of numbers. The conversion of a function to a string deviates slightly from JavaScript where the source code of the function is returned.}

\begin{align*}
\text{EvalTypeErrorEquality}_1 \\
E \vdash e_1 \downarrow v_1 & \quad v_1 = \text{function } p_1(x_1) e_1 \quad bop \in \{===,!=\} \\
& \quad E \vdash e_1 \ bop \ e_2 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalTypeErrorEquality}_2 \\
E \vdash e_2 \downarrow v_2 & \quad v_2 = \text{function } p_1(x_1) e_1 \quad bop \in \{===,!=\} \\
& \quad E \vdash e_1 \ bop \ e_2 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalTypeErrorCall} \\
E \vdash v_1 \neq \text{function } p(x) e_1 & \quad v_1(e_2) \longrightarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateUnary} \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash uop(e_1) \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateBinary}_1 \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash e_1 \ bop \ e_2 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateBinary}_2 \\
E \vdash e_2 \downarrow \text{typeerror} & \quad E \vdash e_1 \ bop \ e_2 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagatePrint} \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash \text{jsy.print}(e_1) \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateIf} \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash e_1 \ ? \ e_2 : e_3 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateConst} \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash \text{const } x = e_1 ; e_2 \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateCall}_1 \\
E \vdash e_1 \downarrow \text{typeerror} & \quad E \vdash e_1(e_2) \downarrow \text{typeerror}
\end{align*}

\begin{align*}
\text{EvalPropagateCall}_2 \\
E \vdash e_2 \downarrow \text{typeerror} & \quad E \vdash e_1(e_2) \downarrow \text{typeerror}
\end{align*}

\text{Figure 6: Big-step operational semantics of JavaScript: Dynamic type error rules.}
When they allow all syntactically valid programs to run and check for type errors during execution.

In our Scala implementation, we will not clutter our Expr type with a typeerror marker. Instead, we will use a Scala exception DynamicTypeError:

```scala
class DynamicTypeError(e: Expr) extends Exception
```
to signal this case. In other words, when your interpreter discovers a dynamic type error, it should throw this exception using the following Scala code:

```scala
throw new DynamicTypeError(e)
```
The argument should be the input expression to eval where the type error was detected. Another advantage of using a Scala exception for typeerror is that the marker does not need to be propagated explicitly (as in the inference rules).

Note in rule E\[VAL\]E\[QUALITY\], we disallow equality and disequality checks (i.e., `==` and `!=`) on function values. If either argument to a equality or disequality check is a function value, then we consider this a dynamic type error. This choice is a slight departure from JavaScript.

(a) First, write some JavaScript programs and execute them as JavaScript programs. This step will inform how you will implement your interpreter and will serve as tests for your interpreter.

Give one test case that behaves differently under dynamic scoping versus static scoping (and does not crash). Explain the test case and how they behave differently in your write-up.

(b) Then, implement

```scala
def eval(env: Env, e: Expr): Expr
```
that evaluates a JavaScript expression e in a value environment env to a value according to the evaluation judgment \( E \vdash e \downarrow v \).

The following helper functions for converting values to numbers, booleans, and strings are provided:

```scala
def toNumber(v: Expr): Double
def toBoolean(v: Expr): Boolean
def toString(v: Expr): String
```
We suggest the following step-by-step process:

1. Extend your Lab 2 implementation with strings. The only string operation that we support is string concatenation via the `e\[1\] + e\[2\]` expression. Thus, the only change you need to make is for the `e\[1\] + e\[2\]` expression case.

2. Get your implementation working with non-recursive functions. On function calls, you need to extend the environment for the formal parameter but not for the function itself.

3. Modify your implementation to support recursive functions.
3. **JavaScript Interpreter: Substitution and Evaluation Order**

In this question, we will do two things. First, we will remove environments and instead use a language semantics based on substitution. This change will “fix” the scoping issue, and we will end up with static, lexical scoping.

As an aside, substitution is not the only way to “fix” the scoping issue. Another way is to represent function values as *closures*, which is a pair of the function with the environment when it is defined. Substitution is a fairly simple way to get lexical scoping, but in practice, it is often not used because it is not the most efficient implementation.

The second thing that we do is move to implementing a small-step interpreter. A small-step interpreter makes explicit the evaluation order of expressions. These two changes are orthogonal, that is, one could implement a big-step interpreter using substitution or a small-step interpreter using environments.

(a) Implement

```python
    def substitute(e: Expr, v: Expr, x: String): Expr
```

that substitutes value v for all *free* occurrences of variable x in expression e. We advise defining substitute by recursion on e. The cases to be careful about are ConstDecl and Function because these are the variable binding constructs. In particular, substitute on expression

```javascript
    a; (const a = 4; a)
```

with value 3 for variable "a" should return

```javascript
    3; (const a = 4; a)
```

not

```javascript
    3; (const a = 4; 3)
```

This function is a helper for the step function, but you might want to implement all of the cases of step that do not require substitute first.

(b) Implement

```python
    def step(e: Expr): Expr
```

that performs one-step of evaluation by rewriting the input expression e into a “one-step reduced” expression. This one-step reduction should be implemented according to the judgment form $e \rightarrow e'$ defined in Figures 7, 8, and 9. We write $e[v/x]$ for substituting value v for all free occurrences of the variable x in expression e (i.e., a call to substitute).

(c) Explain whether the evaluation order is deterministic.

It is informative to compare the small-step semantics used in this question and the big-step semantics from the previous one. In particular, for all programs where dynamic scoping is not an issue, your interpreters in this question and the previous should behave the same.

We have provided the functions evaluate and iterateStep that evaluate “top-level” expressions to a value using your interpreter implementations.
\[ e \rightarrow e' \]

**DO\_NEG**
\[ n' = -\text{toNumber}(v) \]
\[ -v \rightarrow n' \]

**DO\_NOT**
\[ b' = \neg\text{toBoolean}(v) \]
\[ !v \rightarrow b' \]

**DO\_SEQ**
\[ v_1, e_2 \rightarrow e_2 \]

**DO\_PLUS\_NUMBER**
\[ n' = \text{toNumber}(v_1) + \text{toNumber}(v_2) \]
\[ v_1 + v_2 \rightarrow n' \]

**DO\_PLUS\_STRING\_1**
\[ str' = \text{toString}(v_1) + \text{toString}(v_2) \]
\[ str_1 + v_2 \rightarrow str' \]

**DO\_PLUS\_STRING\_2**
\[ v_1 + str_2 \rightarrow str' \]

**DO\_ARITH**
\[ n' = \text{toNumber}(v_1) \ bop \text{toNumber}(v_2) \]
\[ v_1 \ bop \ v_2 \rightarrow n' \]

\[ bop \in \{ -, \ast, / \} \]

**DO\_INEQUALITY**
\[ b' = \text{toNumber}(v_1) \ bop \text{toNumber}(v_2) \]
\[ bop \in \{ <, \leq, >, \geq \} \]

\[ v_1 \ bop \ v_2 \rightarrow b' \]

**DO\_EQUALITY**
\[ v_1 \neq \text{function} \ p_1(x_1) \ e_1 \]
\[ v_2 \neq \text{function} \ p_1(x_2) \ e_2 \]
\[ b' = (v_1 \ bop \ v_2) \]
\[ bop \in \{ ===, !== \} \]

\[ v_1 \ bop \ v_2 \rightarrow b' \]

**DO\_AND\_TRUE**
\[ \text{true} = \text{toBoolean}(v_1) \]
\[ v_1 \ &\& e_2 \rightarrow e_2 \]

**DO\_AND\_FALSE**
\[ \text{false} = \text{toBoolean}(v_1) \]
\[ v_1 \ &\& e_2 \rightarrow \text{false} \]

**DO\_OR\_TRUE**
\[ \text{true} = \text{toBoolean}(v_1) \]
\[ v_1 \ || \ e_2 \rightarrow \text{true} \]

**DO\_OR\_FALSE**
\[ \text{false} = \text{toBoolean}(v_1) \]
\[ v_1 \ || \ e_2 \rightarrow e_2 \]

**DO\_PRINT**
\[ v_1 \text{ printed} \]
\[ \text{jsy.print}(v_1) \rightarrow \text{undefined} \]

**DO\_CONST**
\[ \text{const} \ x = v_1 ; e_2 \rightarrow e_2[v_1/x] \]

**DO\_CALL**
\[ v_1 = \text{function} (x) \ e_1 \]
\[ v_1(v_2) \rightarrow e_1[v_2/x] \]

**DO\_CALL\_REC**
\[ v_1 = \text{function} x_1(x_2) \ e_1 \]
\[ v_1(v_2) \rightarrow e_1[v_1/x_1][v_2/x_2] \]

Figure 7: Small-step operational semantics of JAVASCRIPT: Do rules.
Figure 8: Small-step operational semantics of JavaScript: Search rules.

Figure 9: Small-step operational semantics of JavaScript: Dynamic type error rules.
4. Evaluation Order.

(a) **Mystery.** Calls are amongst the most complicated expression constructs in programming languages. There are many, many different ways of supporting calls and different languages often differ in their support for calls. In this question, we will consider only the operand evaluation order issues with calls; later in the semester we will consider other issues.

Let’s consider a call schematically \( F(A_1, \ldots, A_n) \). \( F \) is the procedure being called. While we are used to calling a procedure directly (e.g., `print()`), most languages, including Mystery, allow complicated expressions for \( F \). For example, if \( a \) is an array of procedures, one can do \( a[2](5) \) in Mystery to call the procedure at \( a[2] \) with argument 5. Mystery also allows a procedure to return a procedure (i.e., has higher-order functions like Scala and JavaScript). Thus, if a procedure \( f \) returns a procedure, then one could write \( f() (x) \). This calls the procedure returned by \( f \) with the argument \( x \). Needless to say, most languages allow one to pass arbitrarily complicated arguments (i.e., \( A_i \)).

i. Discover whether Mystery evaluates the procedure being called before it evaluates its arguments. For example, with \( f() (x) \) does Mystery call the function \( f \) first or does it evaluate \( x \) first (i.e., looks up the value of \( x \)). You may execute up to 4 PRINT statements when you use the submission link below.

ii. Discover whether or not Mystery evaluates arguments to a call from left-to-right. You may execute up to 4 PRINT statements.

Use the following link to submit programs:

http://csci3155.cs.colorado.edu/pl-detective/hw/pldeval.htm

You are welcome to use the “playground” page

http://csci3155.cs.colorado.edu/pl-detective/hw/pldwelcome.htm

as much as you like before using the problem submission link. In your write-up, explain how you came to a conclusion. You should provide evidence that supports your argument. There will be a 5% deduction for this question for each additional executed PRINT.

(b) **JavaScriptry.** Consider the small-step operational semantics for JavaScriptry shown in Figures 7, 8, and 9. What is the evaluation order for \( e_1 + e_2 \)? Explain.

5. Short-Circuit Evaluation. In this question, we will discuss some issues with short-circuit evaluation.

(a) **Concept.** Give an example that illustrates the usefulness of short-circuit evaluation. Explain your example.

(b) **Mystery.** Discover and indicate whether or not Mystery uses short-circuit evaluation of boolean operators (i.e., `AND`). Use this link to submit programs:

http://csci3155.cs.colorado.edu/pl-detective/hw/pldshortcircuit.htm

You should provide evidence that supports your argument. Your program should not execute PRINT more than three times. There will be a 5% deduction for this question for each additional executed PRINT.
(c) **JAVASCRIPTY.** Consider the small-step operational semantics for JAVASCRIPTY shown in Figures 7, 8, and 9. Does \( e_1 && e_2 \) short circuit? Explain.