CSCI 3155: Lab Assignment 2

Spring 2013: Due Sunday, February 17, 2013 6:00 pm

The purpose of this lab is to get practice with reading and writing grammars and to build our first interpreter.

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 unfamilar to you. Finally, make sure that your file compiles and runs (using Scala 2.9.2 or Scala 2.10.0). 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, submit to your Github repository the files named as follows:

- `Lab2.md` with your answers to the written questions. Parse trees and judgements may be submitted as images or other appropriate format.
- `Lab2.scala` with your answers to the coding exercises

See the README.md in the Lab2 folder for more details

Getting Started. The code template is in the Lab2 folder. Be sure to write your name, your partner, and collaborators in the header comment block.

1. **Feedback**. Complete the survey by filling out SURVEY.md after completing this assignment. Any non-empty answer will receive full credit.
2. **Grammars: Synthetic Examples**.

(a) Consider the following grammar:

$$\begin{array}{rcl} A & ::= & A \,\&\, A \mid V \\ V & ::= & \texttt{a} \mid \texttt{b} \end{array}$$

Recall that a grammar defines inductively a set of syntactic objects (i.e., a language). We can also use judgments to define a langauge.

For this exercise, rewrite this grammar using the following two judgment forms:

$A \in$ **AObjects**	meaning	Syntactic object A is in the set **AObjects**.
$V \in$ **VObjects**	meaning	Syntactic object V is in the set **VObjects**.

(b) Show that the grammar in the previous part is ambiguous.

(c) Describe the language defined by the following grammar:

$$\begin{array}{rcl} S & ::= & A \mid B \mid C \\ A & ::= & \texttt{a}\, A \mid \texttt{a} \\ B & ::= & \texttt{b}\, B \mid \varepsilon \\ C & ::= & \texttt{c}\, C \mid \texttt{c} \end{array}$$

(from Sebesta, Chapter 3)

(d) Consider the following grammar:

$$\begin{array}{rcl} S & ::= & A\, \texttt{a}\, B\, \texttt{b} \\ A & ::= & A\, \texttt{b} \mid \texttt{b} \\ B & ::= & \texttt{a}\, B \mid \texttt{a} \end{array}$$

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving **derivations**.

1. `baab`
2. `bbbab`
3. `bbaaaaa`
4. `bbaab`

(from Sebesta, Chapter 3)

(e) Consider the following grammar:

$$\begin{array}{rcl} S & ::= & \texttt{a}\, S\, \texttt{c}\, B \mid A \mid \texttt{b} \\ A & ::= & \texttt{c}\, A \mid \texttt{c} \\ B & ::= & \texttt{d} \mid A \end{array}$$

Which of the following sentences are in the language generated by this grammar? For the sentences that are described by this grammar, demonstrate that they are by giving **parse trees**.

1. `abcd`

2. `acccbd`
3. `acccbcc`
4. `acd`
5. `accc`

(from Sebesta, Chapter 3)

3. **Grammars: Understanding a Language**.

For this part, consider an hypothetical language called MYSTERY. We will explore part of the syntax of MYSTERY here, and consider alternative ways to handle particular terms in the grammer.

(a) Consider the following two grammars for *Expr*. Note that the first grammar is part of *Expr*'s grammar in MYSTERY. In both grammars, *Operator* and *Operand* are the same as in MYSTERY, though you do not need to know their productions for this question.

$$\begin{array}{rcl} \textit{Expr} & ::= & \textit{Operand} \mid \textit{Expr Operator Operand} \end{array}$$

$$\begin{array}{rcl} \textit{Expr} & ::= & \textit{Operand ExprSuffix} \\ \textit{ExprSuffix} & ::= & \textit{Operator Operand ExprSuffix} \mid \varepsilon \end{array}$$

i. Intuitively describe the expressions generated by the two grammars.
ii. Do these grammars generate the same or different expressions? Explain.

(b) Consider the following two grammars for *Operand*. The first grammar is part of *Operand*'s grammar in MYSTERY. In both grammars, *Expr* is the same as in MYSTERY. Again, you do not need to look at the definition of *Expr* to answer this question.

$$\begin{array}{rcl} \textit{Operand} & ::= & \textit{Number} \mid \textit{Id} \mid \textit{Operand}[\textit{Expr}] \end{array}$$

$$\begin{array}{rcl} \textit{Operand} & ::= & \textit{Number OperandSuffix} \mid \textit{Id OperandSuffix} \\ \textit{OperandSuffix} & ::= & [\textit{Expr}]\,\textit{OperandSuffix} \mid \varepsilon \end{array}$$

Note that we are using BNF (not EBNF), the square brackets ([...]) are terminals (used for array references).

i. Intuitively describe the expressions generated by the two grammars.
ii. Do these grammars generate the same or different expressions? Explain.

(c) A portion of the syntax for MYSTERY is given below.

$$\begin{array}{rcl} \textit{Expr} & ::= & \textit{Operand} \mid \textit{Expr Operator Operand} \\ \textit{Operand} & ::= & \textit{Number} \mid \textit{Id} \mid \textit{Operand}[\textit{Expr}] \\ \textit{Operator} & ::= & + \mid > \mid \texttt{AND} \end{array}$$

Looking at the syntax of MYSTERY determine if the + operator is left or right associative. Also, determine if + has higher, lower, or the same precedence as >. Explain the reasoning behind your answer.

(d) Write a Scala expression to determine if $-$ has higher precedence than $<<$ or vice versa. Make sure that you are checking for precedence in your expression and not for left or right associativity. Use parentheses to indicate the possible abstract syntax trees, and then show the evaluation of the possible expressions. Finally, explain how you arrived at the relative precedence of $-$ and $<<$ based on the output that you saw in the Scala interpreter.

(e) Give a BNF grammar for floating point numbers that are made up of a fraction (e.g., 5.6 or 3.123 or -2.5) followed by an optional exponent (e.g., E10 or E-10). The exponent, if it exists, is the letter 'E' followed by an integer. For example, the following are floating point numbers: 3.5E3, 3.123E30, -2.5E2, -2.5E-2, and 3.5. The following are not examples of floating point numbers: 3.E3, E3, and 3.0E4.5.

More precisely, our floating point numbers must have a decimal point, do not have leading zeros, can have any number of trailing zeros, non-zero exponents (if it exists), must have non-zero fraction to have an exponent, and cannot have a '-' in front of a zero number. The exponent cannot have leading zeros.

For this exercise, let us assume that the tokens are characters in the following alphabet Σ:

$$\Sigma \stackrel{\text{def}}{=} \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{E}, \text{-}, .\}$$

Your grammar should be completely defined (i.e., it should not count on a non-terminal that it does not itself define).

4. **JavaScript Interpreter: Numbers, Booleans, Variable Binding, and Conversions**.

JavaScript is a complex language and thus difficult to build an interpreter for it all at once. In this course, we will make some simplifications. We consider subsets of JavaScript and incrementally examine more and more complex subsets during the course of the semester. For clarity, let us call the language that we implement in this course JavaScripty. We may choose to omit complex behavior in JavaScript, but we want any programs that we admit in JavaScripty to behave in the same way as in JavaScript.

In actuality, there is not one language called JavaScript but a set of closely related languages that may have slightly different semantics. In deciding how a JavaScripty program should behave, we will consult a reference implementation that we fix to be Node.js 0.8.8, which uses Google's V8 JavaScript Engine. Thus, we will often need to write little test JavaScript programs and run it through Node.js to see how the test should behave.

One aspect that makes the JavaScript specification complex is the presence of implicit conversions (e.g., string values may be implicitly converted to numeric values depending on the context in which values are used). In this exercise, we will explore some of this complexity by implementing an evaluator with conversions for the subset with numbers, booleans, and variable binding. JavaScript has a distinguished **undefined** value that we will also consider. This version of JavaScripty is much like the LET language in Section 3.2 of Friedman and Wand.

expressions	$e ::= x \mid n \mid b \mid \textbf{undefined} \mid uop\, e_1 \mid e_1\ bop\ e_2 \mid e_1\ ?\ e_2 : e_3$
	$\mid \textbf{const}\ x = e_1;\ e_2 \mid \textbf{jsy}.\textbf{print}(e_1)$
values	$v ::= n \mid b \mid \textbf{undefined}$
unary operators	$uop ::= \texttt{-} \mid \texttt{!}$
binary operators	$bop ::= \texttt{,} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{\|\|}$
variables	x
numbers (doubles)	n
booleans	$b ::= \textbf{true} \mid \textbf{false}$

Figure 1: Abstract Syntax of JavaScripty

statements	$s ::= \textbf{const}\ x = e \mid e \mid \{\ s_1\ \} \mid ; \mid s_1\ s_2$
expressions	$e ::= \cdots \mid \textbf{const}\ x = e_1;\ e_2 \mid (e_1)$

Figure 2: Concrete Syntax of JavaScripty

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.

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.

The concrete syntax accepted by the parser is slightly less flexible than the abstract syntax in order to match the syntactic structure of JavaScript. In particular, all **const** bindings must be at the top-level. For example,

```
1 + (const x = 2; x)
```

is not allowed. The reason is that JavaScript layers a language of *statements* on top of its language of *expressions*, and the **const** binding is considered a statement. A program is a statement s as given in Figure 2. A statement is either a **const** binding, an expression, a grouping of statements (i.e., $\{\ s_1\ \}$), an empty statement (i.e., ;), or a statement sequence (i.e., $s_1\ s_2$). Expressions are as in Figure 1 except **const** binding expressions are removed, and we have a way to parenthesize expressions.

To make the project simpler, we also deviate slightly with respect to scope. Whereas JavaScript considers all **const** bindings to be in same scope, our JavaScripty bindings each introduce their own scope. In particular, for the binding $\textbf{const}\ x = e_1;\ e_2$, the scope of variable x is the expression e_2.

Statement sequencing and expression sequencing are right associative. All other binary operator expressions are left associative. Precedence of the operators follow JavaScript.

The semantics are defined by the corresponding JavaScript program. The one exception is that we have a JavaScripty system function **jsy.print** for printing out values to the console and returns **undefined**. Its implementation is provided for you.

(a) First, write some JavaScripty 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.

(b) Then, implement

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

that evaluates a JavaScripty expression `e` in a value environment `env` to a value. A value is one of a number n, a boolean b, or **undefined**.

It will be useful to first implement two helper functions for converting values to numbers and booleans.

```
def toNumber(v: Expr): Double
def toBoolean(v: Expr): Boolean
```

```
sealed abstract class Expr
case class Var(x: String) extends Expr
  Var($x$)  $x$
case class ConstDecl(x: String, e1: Expr, e2: Expr) extends Expr
  ConstDecl($x$,$e_1$,$e_2$)  const $x = e_1; e_2$
case class N(n: Double) extends Expr
  N($n$)  $n$
case class B(b: Boolean) extends Expr
  B($b$)  $b$
case object Undefined extends Expr
  Undefined  undefined
case class Unary(uop: Uop, e1: Expr) extends Expr
  Unary($uop$,$e_1$)  $uop\, e_1$
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  Binary($bop$, $e_1$)  $e_1\, bop\, e_2$
sealed abstract class Uop
case object Neg extends Uop
  Neg  $-$
case object Not extends Uop
  Not  !
sealed abstract class Bop
case object Plus extends Bop
  Plus  $+$
case object Minus extends Bop
  Minus  $-$
case object Times extends Bop
  Times  $*$
case object Div extends Bop
  Div  /
case object Eq extends Bop
  Eq  ===
case object Ne extends Bop
  Ne  !==
case object Lt extends Bop
  Lt  <
case object Le extends Bop
  Le  <=
case object Gt extends Bop
  Gt  >
case object Ge extends Bop
  Ge  >=
case object And extends Bop
  And  &&
case object Or extends Bop
  Or  ||
case object Seq extends Bop
  Seq  ,
case class If(e1: Expr, e2: Expr, e3: Expr) extends Expr
  If($e_1$, $e_2$, $e_3$)  $e_1$ ? $e_2$ : $e_3$
case class Print(e1: Expr) extends Expr
  Print($e_1$)  jsy.print($e_1$)
```

Figure 3: Representing in Scala the abstract syntax of JavaScripty. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.