

CSCI 3155: Lab Assignment 2

Fall 2022

Checkpoint due Friday, September 9, 2022

Assignment due Friday, September 16, 2022

Learning Goals. The primary learning goals of this lab are to understand the following:

- how grammars are used to specify the syntax of programming languages;
- the distinction between concrete and abstract syntax;
- the basics of inductive definitions via judgments, judgment forms, and inference rules; and
- variable binding and variable environments.

PL Ideas Syntax (grammars). Inductive definitions (judgments, judgment forms, and inference rules). Semantics (via detective work).

FP Skills Recursion over abstract syntax. Maps (environments for variable binding).

General Guidelines. The instructor will randomly assign partners for the lab assignment, and you will get a different partner for every lab assignment. You will work on this assignment closely with your partner. However, note that **each student needs to submit** on Canvas and are individually responsible for completing the assignment so that you can do well in your interview.

You are welcome to talk about these questions beyond your teams. However, we ask that you code in pairs. See the collaboration policy for details, including the following:

*Bottom line, feel free to use resources that are available to you as long as the use is **reasonable** and you **cite** them in your submission. However, copying answers directly or indirectly from solution manuals, web pages, or your peers is certainly unreasonable.*

Also, recall the evaluation guideline from the course syllabus.

Both your ideas and also the clarity with which they are expressed matter—both in your English prose and your code!

We will consider the following criteria in our grading:

- How well does your submission answer the questions? *For example, a common mistake is to give an example when a question asks for an explanation. An example may be useful in your explanation, but it should not take the place of the explanation.*
- How clear is your submission? *If we cannot understand what you are trying to say, then we cannot give you points for it. Try reading your answer aloud to yourself or a friend; this technique is often a great way to identify holes in your reasoning. For code, not every program that “works” deserves full credit. We must be able to read and understand your intent. Make sure you state any pre-conditions or invariants for your functions (either in comments, as assertions, or as require clauses as appropriate).*

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 via `sbt test`. A program that does not compile will *not* be graded—no interview will be conducted.

Submission Instructions. We are using GitHub for assignment distribution and auto-testing. You need to have a GitHub identity and must have your full name in your GitHub profile so that we can associate you with your submissions.

You will be editing and submitting the the following files:

- `src/main/scala/jsy/student/Lab2.scala` with your solution to the coding exercises;
- `lab2-writeup.pdf` or `lab2-writeup.md` for a pdf or a Markdown document that should be pushed to the root directory of your repository with your response to the written questions (scanned, clearly legible handwritten write-ups are acceptable). **You will not get credit for write-ups in any other file format.**

You are also likely to edit

- `src/test/scala/jsy/student/Lab2Spec.scala` with your additional tests;
- `src/main/scala/jsy/student/Lab2.worksheet.sc` for any scratch work; and
- `src/main/scala/jsy/student/Lab2.worksheet.js` for any JavaScript experimentation.

You can also add JAVASCRIPTY tests in

- `src/test/resources/lab2/` as a pair of a `*.jsy` and a `*.ans` for a JAVASCRIPTY expression and the expected value of that expression, respectively.

Following good git practice, please make commits in small bits corresponding to completing small conceptual parts and push often so that your progress is evident. We expect that you have some familiarity with git from prior courses. If not, please discuss with your classmates and the course staff (e.g., via Piazza).

At any point, you may push your updated files to your GitHub repository for auto-testing. You need to push to your GitHub repository for the auto-testing part of your score, as well as to continue to the interview.

Sign-up for an interview slot for an evaluator. To fairly accommodate everyone, the interview times are strict and **will not be rescheduled**. Missing an interview slot means missing the interview evaluation component of your lab score. Please take advantage of your interview time to maximize the feedback that you are able to receive. Arrive at your interview ready to show your implementation and your written responses. Implementations that do not compile and run will not be evaluated.

Finally, upload the zip file of your repo to Canvas by clicking the Code button and then Download ZIP. The generated file should be named *your-lab-repo-name-main.zip*.

Getting Started. First, form a team of two and pick a team name. For our bookkeeping, please prefix your team name with the lab number and your CU IdentiKeys (i.e., your team should look something like L2_your-identikey1_your-identikey2_anatomists).

You must work in teams of two, and you will form teams in lab section. If you cannot connect with your partner, then please contact the course staff (via Piazza).

Then, log into Canvas and follow the GitHub Classroom link for setting up your Lab 2 repository with your team name. The first person will create the team, and the second person will select the team name from the existing team names. If you need to move teams after you have already created or joined a repository, you will need to contact the Course Manager (via Piazza) or work with a course staff member to move you manually.

If you would like to look at the code before getting your own copy from GitHub Classroom, you may go to <https://github.com/csci3155/pppl-lab2>.

Checkpoint. The checkpoint is to encourage you to start the assignment early and it requires you to submit (i.e., push) your partial solution on GitHub a week before the assignment is due. You do not need to attempt everything a week early, but we want you to start working on it and make note in this handout any required questions in the checkpoint. This means that submitting the empty template that fails all tests is **not sufficient**. Failing to submit to the checkpoint will prevent you from proceeding to the interview.

1. **Feedback.** Complete the survey on linked on Canvas after completing this assignment. Any non-empty answer will receive full credit.

Written Section

2. Grammars: Synthetic Examples.

- (a) Describe the language defined by the following grammar:

$$\begin{aligned} S &::= ABA \\ A &::= a \mid aA \\ B &::= \varepsilon \mid bBc \mid BB \end{aligned}$$

(b) Consider the following grammar:

$$\begin{aligned} S &::= A a B b \\ A &::= A b \mid b \\ B &::= a B \mid a \end{aligned}$$

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)

(c) Consider the following grammar:

$$\begin{aligned} S &::= a S c B \mid A \mid b \\ A &::= c A \mid c \\ B &::= d \mid A \end{aligned}$$

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)

(d) Consider the following grammar:

$$A ::= a \mid b \mid A \oplus A$$

Show that this grammar is ambiguous.

(e) Let us ascribe a semantics to the syntactic objects A specified in the above grammar from part d. In particular, let us write

$$A \Downarrow n$$

for the judgment form that should mean A has a total n symbols where n is the meta-variable for natural numbers. Define this judgment form via a set of inference rules. You may rely upon arithmetic operators over natural numbers. Hint: There should be one inference rule for each production of the non-terminal A (called a syntax-directed judgment form).

3. Grammars: Understanding a Language.

- (a) Consider the following two grammars for expressions e . In both grammars, *operator* and *operand* are the same; you do not need to know their productions for this question.

$$e ::= \text{operand} \mid e \text{ operator operand}$$

$$\begin{aligned} e &::= \text{operand esuffix} \\ \text{esuffix} &::= \text{operator operand esuffix} \mid \epsilon \end{aligned}$$

- i. Intuitively describe the expressions generated by the two grammars.
 - ii. Do these grammars generate the same or different expressions? Explain.
- (b) 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.
- (c) 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 :

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

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

Programming Section

4. JavaScripty Interpreter: Booleans, Strings, Variable Binding, and Conversions.

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, strings, 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.

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. Also note that JAVASCRIPTY in this lab extends JAVASCRIPTY from the previous lab.

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.

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.

To make the project simpler, we also deviate slightly with respect to scope. Whereas JavaScript considers all **const** bindings to be in the same scope, our JAVASCRIPTY bindings each introduce their own scope. In particular, for the binding **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. We also have a system function **console.log** 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

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

Figure 1: Abstract syntax of JAVASCRIPTY

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

Figure 2: Concrete syntax of JAVASCRIPTY

```
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 , a string s , or **undefined**.

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

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

A value environment, a map from strings to JAVASCRIPTY values, is represented by a Scala Map[String,Expr]:

```
type Env = Map[String, Expr]
val empty: Env = Map()
def lookup(env: Env, x: String): Expr = env(x)
def extend(env: Env, x: String, v: Expr): Env = {
  require(isValue(v))
  env + (x -> v)
}
```

We provide the above the three functions to interface with the Scala standard library. You may use the Scala standard library directly if you wish, but we recommend that you just use these interfaces, as they are all that you need. The empty Scala value represents an empty value environment, the lookup function gets the value bound to the variable named by a given string, and the extend function extends a given environment with a new variable binding.

```

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, e1, e2)  const x = e1; e2
case class N(n: Double) extends Expr
  N(n)  n
case class B(b: Boolean) extends Expr
  B(b)  b
case class S(str: String) extends Expr
  S(str)  str
case object Undefined extends Expr
  Undefined  undefined
case class Unary(uop: Uop, e1: Expr) extends Expr
  Unary(uop, e1)  uope1
case class Binary(bop: Bop, e1: Expr, e2: Expr) extends Expr
  Binary(bop, e1)  e1 bope2
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(e1, e2, e3)  e1 ? e2 : e3
case class Print(e1: Expr) extends Expr
  Print(e1)  console.log(e1)

```

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.