Written Assignment 1

CS 538, Spring 2019

January 27, 2020

1 Calculator language: Syntax (10)

In the next two exercises, you will develop a small language for a basic calculator. For simplicity, you do not need to model parentheses and precedence. Of course, if we were to design a parser for this language, we would have to carefully sort out how to parse programs like "$5 \times 3 + 2$".

Let's start by specifying the *syntax* of the language, i.e., the structure of valid programs. *Integer constants* are given by a non-empty string of digits $0 - 9$, possibly beginning with $-$ for negative numbers. *Boolean expressions* include: (1) the constants "*tt*" and "*ff*" for true and false, (2) the "and"/"or" of two Boolean expressions, written "$\cdot \,\&\&\, \cdot$" and "$\cdot \,||\, \cdot$" respectively, (3) equality ($==$) and lesser-than ($<$) between two arithmetic expressions. *Arithmetic expressions* include (1) integer constants, (2) addition and multiplication of two arithmetic expressions, written "$\cdot + \cdot$" and "$\cdot \times \cdot$" respectively, and (3) if-then-else, written "*if* $\cdot$ *then* $\cdot$ *else* $\cdot$". The guard expression can be any Boolean expression, while the body expressions should both be arithmetic expressions.

Translate this English description of the language into a grammar. Your grammar should have three classes of non-terminals: *int* for integer constants, *ae* for arithmetic expressions, and *be* for boolean expressions. You do not need to stick exactly to EBNF, so long as it is clear which symbols in your grammar represent literal characters, and which symbols in your grammar represent expressions.

2 Calculator language: Operational semantics (10)

Now, let's specify the behavior of programs in this language. We will build a *small-step* semantics like we saw in class. We first specify the values, i.e., the programs that are done executing/do not step. We take the Boolean constants tt, ff and integer constants (e.g., 42, 0, -125) as values.

1. Give a small-step operational semantics for our calculator language. For if-then-else, the program should first evaluate the guard to a constant before evaluating the corresponding body expression; the other body expression should not be evaluated. (This behavior is also known as *short-circuiting*, or a *lazy-if*.)

2. Give an alternative operational semantics for if-then-else that evaluates both bodies before evaluating the guard; call this semantics an *eager-if*. What are some reasons to prefer *lazy-if* over *eager-if*?

3 Lambda calculus (5)

These exercises will give you some practice with the core functional language, also known as the *lambda calculus*. Throughout, assume that the language is *call by value*: when evaluating a function application $(\lambda x.\ e_1)\ e_2$, we first evaluate e_2 to a value before substituting for x in e_1. We'll usually leave out parentheses. For instance, $(\lambda x.\ \lambda y.\ e)\ a\ b$ is short for $((\lambda x.\ \lambda y.\ e)\ a)\ b$

Evaluate each of the following programs until it reaches a value, or returns to the original program. Show each step in the evaluation.

1. $(\lambda f.\ \lambda x.\ f\ x)\ (\lambda x.\ x+1)\ 5$
2. $(\lambda f.\ \lambda x.\ \lambda y.\ f\ y\ x)\ (\lambda x.\ \lambda y.\ x-y)\ 5\ 3$
3. $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

   This program is also known as the Ω *combinator*; it is a building block that can be used to model recursive (i.e., looping) programs.

4 Recursion (5)

Use the program construct $\mathit{fix}\, f.\ e$ we introduced to write a lambda calculus function that takes in a natural number n and returns the n-th Fibonacci number $\mathit{fib}(n)$, defined by $\mathit{fib}(0) = 1, \mathit{fib}(1) = 1$ and $\mathit{fib}(n) = \mathit{fib}(n-1) + \mathit{fib}(n-2)$ for $n \geq 2$. Test out your program by applying it to 3; show the steps in your reduction.