The λ-Calculus

CAS CS 320: Concepts of Programming Languages

October 22, 2026

We shift our attention to a particular PL called the ***λ-calculus***, the canonical function PL. This will give us an opportunity to think about how higher-order functions and substitution work formally. The simplest (though not entirely correct) way to describe the λ-calculus is that it's the fragment of Python consisting of variables, function application, and single-argument anonymous functions—it's not an accident that Python uses the syntax `lambda x: e` for anonymous functions.

The λ-calculus was introduced in the 30s by Alonzo Church, and, as with many things, he introduced it in a setting fairly different from that of its ultimate use. Church was attempting to provide a *foundations of mathematics*, i.e., a basic system in which all mathematics can be carried out with complete formality. This project ultimately failed; two mathematicians Stephen Kleene and J.B. Rosser went on to show that Church's system is *inconsistent*. That is, working in it, you can prove *false* statements, which is problematic if you're doing math (you don't want to prove something only to find its not true). Church went on to extract a useful fragment of his system, which is how we get the untyped and simply-typed λ-calculus.[1]

We can think of the λ-calculus as a theory of functions as *procedures* as opposed to *input-output mappings*. In most math classes, we think of functions as a set of pairs, e.g., the integer "add 1" function is the set

$$\{\ldots, (-1, 0), (0, 1), (1, 2), (2, 3), \ldots\}$$

In other words, we don't care how a function "gets" the output, just which output corresponds to which input. But in other settings (e.g., when we program) we do care, and we instead specify a function by what it "does" to the input. In mathematical notation, we might write something like

$$x \mapsto x + 1$$

[1] There are many details of this story I'm eliding, it's worth looking into if you're interested in the history of 20th century mathematics. There's certainly something to be said here about the nature of research; fail at one thing and ultimately develop something fundamental. Fun fact: Alonzo Church is my great-great-grand-advisor (Stuart Kurtz → Carl Jockusch Jr. → Hartley Rogers Jr. → Alonzo Church) and he's Professor Das's great-grand-advisor (Frank Pfenning → Peter Bruce Andrews → Alonzo Church) making me Professor Das's second academic-cousin once removed. And since Alan Turing was advised by Alonzo Church, he's my great-grand-academic-uncle (and Professor Das's grand-academic-uncle).

```
<expr> ::= fun <var> -> <expr>
         | <expr> <expr>
         | <var>
```

Figure 1: Abstract syntax of the λ-calculus

expression	`<expr>`	e
variable	`<var>`	x
anonymous function	`fun <var> -> <expr>`	$\lambda x.e$
application	`<expr> <expr>`	ee

Figure 2: Relationship between the two abstract syntaxes for the λ-calculus

That is, to x, "do" the procedure of adding 1 to it. Below we'll replace "$\mapsto$" with "." and put a "λ" in front (i.e., $\lambda x.(x+1)$) but modulo syntax, this is our new object of study.

The λ-calculus essentially marks the beginning of functional PL theory; it's the study functional procedures as *syntactic* objects with computational (i.e., *operational semantic*) behavior. And if we strip away the bells and whistles of modern PLs, leaving only the ability to define (anonymous) functions, along with a few requisite features (variables to refer to their parameters, function application to make use of them) we're left with the λ-calculus. This might seem restrictive, but it turns out that the λ-calculus is ***Turing-complete***, i.e., we can "do" any kind of computation we want in the λ-calculus (albeit not all that ergonomically (more on that later)).

The syntax of the λ-calculus. In the style of BNF grammars we've seen thus far, the abstract syntax[2] of the λ-calculus is presented in Figure 1. It's as we've said; all we've got is anonymous functions, function application, and variables. This BNF meta-syntax is a bit tedious, so we instead use a more compact mathematical notation for syntax of the λ-calculus:

$$e \ ::= \ \lambda x.e \mid ee \mid x$$

The relationship between the two abstract syntaxes is given explicitly in Figure 2. Like in OCaml, we take anonymous functions to have *lower* precedence than function application, and we take function application to be left-associative. We also use parentheses to disambiguate when necessary (again, meta-syntactically).

[2]Reminder: Abstract syntax is distinct from concrete syntax in that it's just a way of describing ASTs, and doesn't necessarily correspond identically with what's written in a program, e.g., we don't deal with parentheses, precedence, or associativity in abstract syntax—these become meta-syntactic concerns. In particular, the grammars we present are technically ambiguous.

$$\frac{e_1 \longrightarrow e_2}{e_1 e \longrightarrow e_2 e}\ \textsc{appLeft} \qquad \frac{e_1 \longrightarrow e_2}{e e_1 \longrightarrow e e_2}\ \textsc{appRight} \qquad \frac{e_1 \longrightarrow e_2}{\lambda x.e_1 \longrightarrow \lambda x.e_2}\ \textsc{fun}$$

$$\frac{}{(\lambda x.e_1)e_2 \longrightarrow [e_2/x]e_1}\ \textsc{beta}$$

Figure 3: Non-deterministic full small-step semantics for the λ-calculus

Exercise 1. *Write the following expressions in the compact mathematical notation.*

1. `fun x -> x x`
2. `(fun x -> x) (fun x -> x) x`
3. `(fun x -> fun y -> y x) (fun x -> x) (fun z -> z z)`
4. `fun x -> x x y (y z w) x (fun x -> x y z)`

Exercise 2. *Rewrite each of the following expressions with all necessary parentheses (i.e., assuming no precedence or associativity).*

1. $\lambda x.xxx$
2. $\lambda x.\lambda y.yz(\lambda x.x)$
3. $xy(zw)(\lambda x.\lambda x.x(\lambda x.x))$

In OCaml, we can represent this abstract syntax as a simple ADT.

```
type var = string
type expr = Fun of var * expr | App of expr * expr | Var of var
```

The (operational) semantics of the λ-calculus. We begin with the strongest (nondeterministic) small-step semantics (Figure 3) which *a posteriori* takes a value to be any normal form, i.e., any irreducible expression. It's possible to express values syntactically:

$$v \ ::= \ \lambda x.v \mid xv \ldots v$$

In other words, a value is function whose body is a value, or a stack of zero or more (left-associative) applications starting with a variable.

Exercise 3. ***(Challenge)*** *Prove this, i.e., prove that an expression is a normal form w.r.t. the rules in Figure 3 if and only if it is of the form given by the above grammar.*

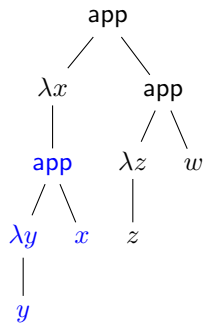


Figure 4: Abstact syntax tree for $(\lambda x.(\lambda y.y)x)((\lambda z.z)w)$

The first two rules (appLeft) and (appRight) say that we can evaluate the left and right sub-expression of an application, respectively. The third rule (fun) says that we can evaluate the body of a function. The last rule (beta)—also written β—is the most important. It says that if we have an anonymous function applied to an argument, we *substitute* e_2 *for* x *in* e_1. We'll say more about substitution in a moment, but for now we should use our intuitions, i.e., roughly speaking this means "replace all the x variables in e_1 with e_2." All together, we call the relation "$\longrightarrow$" as defined in Figure 3 ***β-reduction***, and we'll occasionally write it as "$\longrightarrow_\beta$".

Exercise 4. *Give a derivation for the judgment* $(\lambda x.xx)(\lambda y.y) \longrightarrow (\lambda y.y)(\lambda y.y)$ *using the rules in Figure 3.*

Exercise 5. *Determine a value* v *such that* $(\lambda x.xx)(\lambda y.y) \longrightarrow^\star v$ *and provide a derivation (using the rules in Figure 3).*

Exercise 6. *Demonstrate that the semantics in Figure 3 is non-deterministic (i.e., determine an expression that can be single-step evaluated in multiple ways).*

Exercise 7. *Before reading on, write down what you think would be a reasonable formal definition of substitution.*

Redexes. We can give an alternative, more combinatorial view of the above semantic rules via the notion of redexes, i.e., sub-expressions to which the β-rule can be applied. Informally, a sub-expression is a (well-formed) "part" of an expression. Since we're working with abstract syntax *trees*, it's easy to formally define this: a ***sub-expression*** of e is a (full) subtree of the AST represented by e. Figure 4 has the AST for $(\lambda x.(\lambda y.y)x)((\lambda z.z)w)$ and colors in blue the sub-expression $(\lambda y.y)x$.[3] A ***β-redex*** is a sub-expression of the form $(\lambda x.e_1)e_2$. For example, the sub-expression $(\lambda y.y)x$ colored blue in Figure 4 is a redex. The above rules in sum say that we can replace any redex in e with the result

[3]This presentation of ASTs differs slightly from that of the parse trees we've presented earlier, but not drastically so.

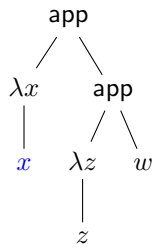


Figure 5: Abstact syntax tree for $(\lambda x.x)((\lambda z.z)w)$

of applying the β-rule to it. This primarily emphasizes that reduction is a local operation; we can think of any single β-reduction as:

$$(\cdots((\lambda x.e_1)e_2)\cdots) \longrightarrow (\cdots([e_2/x]e_1)\cdots)$$

where the only part of the expression that changes is the redex. Figure 5 has the AST of $(\lambda x.x)((\lambda z.z)w)$, the result of reducing the redex $(\lambda y.y)x$ in $(\lambda x.(\lambda y.y)x)((\lambda z.z)w)$. Note that the only part of the tree that changes is the subtree colored in blue.

Exercise 8. *For each of the following expressions, draw its AST and highlight the sub-expressions that are redexes.*

1. $(\lambda x.x)(\lambda y.y)z$
2. $(\lambda x.(\lambda y.y)x)((\lambda z.z)w)$
3. $y(\lambda x.(\lambda z.z)x)((\lambda x.\lambda y.xy)(\lambda z.z)w)$

Exercise 9. *Give a derivation of the following judgment*

$$(\lambda x.(\lambda y.x)(\lambda x.xx))(\lambda x.yx) \longrightarrow (\lambda x.x)(\lambda x.yx)$$

and highlight in that redex that is reduced.

Substitution and α-equivalence. It's finally time to formally define substitution. This is surprisingly complicated, so we'll take it in steps. Recall that we use the notation $[e_2/x]e_1$ for the result of substituting e_2 for x in e_1.

We should already have a sense that this isn't "obvious." For example, we can't just "replace all the occurences of x with e_2." Then

$$[\lambda y.y/x](\lambda x.x) = \lambda(\lambda y.y).(\lambda y.y) \qquad \text{(incorrect)}$$

which isn't a valid λ-expression (here I've colored the substitution variable and the replaced variables in blue). We call the "λx" in $\lambda x.e$ the ***binder***; the above indicates that substitution should not replace variables appearing in binders. But even if we don't replace binder variables, we still can't replace x naively:

$$[\lambda y.y/x](\lambda x.x) = \lambda x.\lambda y.y \qquad \text{(incorrect)}$$

Allowing this would mean substituting into an identity function can make it into *not* an identity function (in this case, a constant function), which should at least seem fishy. To state formally what's "fishy" here, we need to fix some basic concepts.

Variables can either be bound or free in an expression, depending on how they appear. Informally, a variable x is bound in e if it appears in the scope of a binder of the form "λx" within e, and a variable is free if this is not the case. We can define the set of **free variables** of an expression e, written $\mathsf{FV}(e)$, inductively by:

$$\begin{aligned}\mathsf{FV}(\lambda x.e) &= \mathsf{FV}(e) \setminus \{x\} \\ \mathsf{FV}(e_1 e_2) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\ \mathsf{FV}(x) &= \{x\}\end{aligned}$$

The important part of this definition is how it behaves on anonymous functions: it *removes* x from the from free variable set, exactly because the "λx" binds x. As a reminder, inductive definitions are convenient because they're easy to translate into recursive (OCaml) implementations.

```
module VarSet = Set.Make (String)

let rec fv e =
  match e with
  | Fun (x, e) -> VarSet.remove x (fv e)
  | App (e1, e2) -> VarSet.union (fv e1) (fv e2)
  | Var x -> VarSet.singleton x
```

The set of **bound variables** of an expression e, written $\mathsf{BV}(e)$, is defined similarly:

$$\begin{aligned}\mathsf{BV}(\lambda x.e) &= \mathsf{BV}(e) \cup \{x\} \\ \mathsf{BV}(e_1 e_2) &= \mathsf{BV}(e_1) \cup \mathsf{BV}(e_2) \\ \mathsf{BV}(x) &= \varnothing\end{aligned}$$

but this only gives the set of *names* used as bound variables. Every subexpression that is a variable is either free or bound, but a given variable name can appear both bound and free, e.g., x appears free and bound in $x(\lambda x.x)$. Likewise, a name can be used as a bound variable more than once, and in particular two instances of a bound variable named x may be bound by different binders, e.g., $(\lambda x.x)(\lambda x.x)$.

Exercise 10. *Determine* $\mathsf{FV}(e)$ *and* $\mathsf{BV}(e)$ *for each of the expressions* e*. Additionally, draw an arrow from each bound variable to the binder that binds it.*

1. $\lambda x.xy$

2. $(\lambda x.xy)(\lambda y.x)$

3. $\lambda x.\lambda v.(\lambda y.yx)(\lambda y.wy)(\lambda w.w)$

Exercise 11. *Write the OCaml implementation of* `bv` *for bound variables.*

Two expression are ***α-equivalent*** if they're the same up to renaming of bound variables. We write "$e_1 =_\alpha e_2$" for "e_1 and e_2 are α-equivalent." For example, $\lambda z.wz(\lambda q.rs) =_\alpha \lambda a.wa(\lambda b.rs)$ because they differ only in the choice of binder variable names (and their free variables are the same). We choose to keep this concept informal, but it's possible define α-equivalence as an inference system.

Exercise 12. ***(Challenge)*** *Write an inference system over judgments of the form $e_1 =_\alpha e_2$ that formally defines α-equivalence.*

We use α-equivalence to state the ***the principle of name irrelevance***, which informally says that it shouldn't matter what names I gives to variables in my program; the computational behavior of a program should be unaffected by consistently renaming variables. For example, the following two functions in OCaml behave exactly the same way.

```
let foo x =
  let y = x + 1 in
  y * y

let bar z =
  let w = z + 1 in
  w * w
```

The principle of name irrelevances says that two α-equivalent expressions are computationally indistinguishable. The takeaway: *we only care about expressions up to α-equivalence.* Moving forward, we'll write "=" for "$=_\alpha$" when comparing λ-expressions.

A function on expressions is said to ***preserve α-equivalence*** if α-equivalent expressions remain α-equivalent after transformation. In symbols, a function f preserves α-equivalence if $e_1 =_\alpha e_2$ implies $f(e_1) =_\alpha f(e_2)$. The "fishyness" above can be expressed formally as saying that the naive definition of substitution doesn't preserve α-equivalence, e.g., $\lambda x.x =_\alpha \lambda z.z$ but

$$[\lambda y.y/x](\lambda x.x) = \lambda x.\lambda y.y \neq_\alpha \lambda z.z = [\lambda y.y/x](\lambda z.z) \qquad \text{(incorrect)}$$

According to the above discussion, since substitution is used in evaluation, this also breaks the principle of name irrelevance. The fix: bound variables, because they can be named whatever we want, *should not be changed by substitution.* If a variable is bound, it's because it'll be "filled in" when the anonymous function with its corresponding binder is applied. This fix ensures that we don't change the computational behavior of an expression by virtue of a bound

$$[e/x](\lambda y.e_1) = \begin{cases} \lambda y.e_1 & x = y \\ \lambda y.[e/x]e_1 & x \neq y \end{cases}$$
$$[e/x](e_1 e_2) = ([e/x]e_1)([e/x]e_2)$$
$$[e/x]y = \begin{cases} e & x = y \\ y & x \neq y \end{cases}$$

Figure 6: Substitution (not capture-avoiding)

variable being incidentally named the same thing as the free variable at which we're substituting.

This brings us to our first "real" definition of substitution, which appears in Figure 6. The anonymous functions part of the definition ensures that we only substitute in the body of the function if the variable is not the same as the one in the binder. In the above example, we have

$$[\lambda y.y/x](\lambda x.x) = \lambda x.x =_\alpha \lambda z.z = [\lambda y.y/x](\lambda z.z)$$

since the variable being substituted is the same as that of the binder (the "blocking" binder variable is colored red).

Exercise 13. *Perform each of the following substitutions according to the definition in Figure 6.*

1. $[\lambda z.z/x](xyx)$
2. $[\lambda z.zz/x](\lambda y.x(\lambda x.xy))$
3. $[\lambda y.y/x]((\lambda x.xy)(\lambda y.x))$

It's not immediately obvious, but it turns out that the definition of substitution given in Figure 6 *still doesn't preserve α-equivalence.*

Exercise 14. *Before reading the next line, try to think of a counterexample.*

Consider the following counterexample:

$$[y/x](\lambda y.x) = \lambda y.y \neq_\alpha \lambda z.y = [y/x](\lambda z.x)$$

This definition fails to handle the situation in which *the expression being subed* has free variables; our first error was that we accidentally treated bound variables as free, whereas here we're treating free variables as bound. This phenomenon is called ***variable capture*** and it's one of the reasons why substitution is difficult to define.

Capture-avoiding substitution. The fix isn't particularly elegant: in the case of potential variable capture, we just *change the binder.* This amounts to

$$[e/x](\lambda y.e_1) = \begin{cases} \lambda y.e_1 & x = y \\ \lambda y.[e/x]e_1 & x \neq y, y \notin \mathsf{FV}(e) \\ \lambda z.[e/x][z/y]e_1 & x \neq y, y \in \mathsf{FV}(e), z \text{ is fresh} \end{cases}$$

$$[e/x](e_1 e_2) = ([e/x]e_1)([e/x]e_2)$$

$$[e/x]y = \begin{cases} e & x = y \\ y & x \neq y \end{cases}$$

Figure 7: Capture-avoiding substitution

choosing a new name for the binder variable and replacing the corresponding bound variables in the body of the function (see Figure 7). The condition that "z is fresh" in the definition means that z is a variable that doesn't appear already bound or free anywhere in the expression being sub-ed into, or the expression being sub-ed in (we don't want accidently choose a new binder variable that captures a *different* free variable).

Note that, implementation-wise, there's a degree of freedom here in how we determine the new binder variable z. It's convenient to use a bit of OCaml mutability[4] to generate these fresh variables.

```
let gensym =
  let i = ref 0 in
  fun () -> i := !i + 1; "$" ^ string_of_int i
```

We prepend a dollar sign to fresh names so they don't clash with user-defined names.[5]

Exercise 15. *Perform each of the following substitutions according to the definition given in Figure 7. You're free to choose whatever fresh variables you'd like.*

1. $[yz/x](\lambda w.xw)$
2. $[y/x](\lambda y.xy)$
3. $[xy/z](\lambda x.z(\lambda z.z)(\lambda w.zx))$

Exercise 16. *Demonstrate that if the sub-ed in expression e has no free variables (i.e., it's **closed**), then the definitions of substitution in Figures 6 and 7 are equivalent.*

[4] We're lifting the veil here. Yes, OCaml has mutability. You're not allowed to use it other in this instance.

[5] Most PLs, including the ones we implement, don't allow dollar signs in variable names. But you can use any mechanism to avoid clashing with user-defined names.

$$\frac{e_1 \longrightarrow e_2}{e_1 e \longrightarrow e_2 e}\ \textsc{appLeft} \qquad \frac{e_1 \longrightarrow e_2}{e e_1 \longrightarrow e e_2}\ \textsc{appRight}$$

$$\frac{}{(\lambda x.e_1)e_2 \longrightarrow [e_2/x]e_1}\ \textsc{beta}$$

$$v \ ::= \ \lambda x.e$$

Figure 8: Non-deterministic small-step semantics for the λ-calculus with no evaluation under binders

Exercise 17. ***(Challenge)*** *Prove that the definition of capture-avoiding substitution (Figure 7) preserve α-equivalence. This depends on the formal definition of α-equivalence.*

Evaluation under binders. The reason we don't consider the definition of non-capture-avoiding substitution (Figure 6) "incorrect" is that it's sufficient in most practical settings, and avoids the implementation issue of determining fresh variables. In practice, we often have no need to reduce the body of a function. This is partly why when you evaluate a function expression, Utop reports `<fun>`; it's a function value, and we don't care as users how it's represented until it gets applied. We also generally don't deal with expressions with free variables. This is why when you evaluate the expression `x`, Utop reports `Error: Unbound variable x`.

As is hinted in the above exercise, if the sub-ed in expression has no free variables, then we can use non-capture avoiding substitution. And if we remove the (fun) rule (reproduced in Figure 8 for convenience) then every application of the β-rule in a β-reduction of a ***closed*** expression (i.e., an expression with no free variables) has this property.

Exercise 18. *Prove this. Hint: Use the fact that all derivations in this small-step semantics are "unary trees" with the β-rule as an axiom.*

We update our values to include only functions (in practice, free variables are "unknowns") and to allow the bodies of function values to be any expression (because we can't evaluate them). Likewise, this is case where we make an implicity assumption about our initial expressions we care about, i.e., that they're closed.

Call-by-value and call-by-name evaluation strategies. As noted previously, it's useful to have a *deterministic* small-step semantics so that we've fixed the computational behavior (i.e., the evaluation strategy) of our PL (making it generally more predictable and more clearly implementable). In the case of simple PLs like arithmetic, the choice of evaluation strategy is mostly for convenience. In the λ-calculus, evaluation strategies have potentially drastic differences. Note: In what follows we do *not* evaluate under binders.

$$\frac{e_1 \longrightarrow e_2}{e_1 e \longrightarrow e_2 e}\ \textsc{appLeft} \qquad \frac{e_1 \longrightarrow e_2}{(\lambda x.e)e_1 \longrightarrow (\lambda x.e)e_2}\ \textsc{appRight}$$

$$\frac{}{(\lambda x.e_1)(\lambda y.e_2) \longrightarrow [\lambda y.e_2/x]e_1}\ \textsc{beta}$$

Figure 9: Call-by-value small-step semantics for the λ-calculus

$$\frac{e_1 \longrightarrow e_2}{e_1 e \longrightarrow e_2 e}\ \textsc{appLeft} \qquad \frac{}{(\lambda x.e_1)e_2 \longrightarrow [e_2/x]e_1}\ \textsc{beta}$$

Figure 10: Call-by-name small-step semantics for the λ-calculus

First, we should note that, unlike arithmetic, the λ-calculus (as given by the semantics in Figure 8) is *not* normalizing. Consider the following λ-expression.

$$(\lambda x.xx)(\lambda x.xx)$$

Exercise 19. *Show that* $(\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)(\lambda x.xx)$ *according to the rules in Figure 8. Then argue that the* λ*-calculus is not normalizing.*

We call this expression (for historical reasons) the ***Ω-combinator***, or just Ω. It's also worth noting that an expression with non-normalizing sub-expression like Ω may still have a normal form. So the choice of evaluation strategy could determine *whether or not the computation terminates.*

Exercise 20. *Determine a value* v *such that* $(\lambda x.\lambda y.y)\Omega(\lambda x.x) \longrightarrow v$ *(using the rules in Figure 8).*

Two of the most studies evaluation strategies are ***call-by-value (CBV)*** and ***call-by name (CBN)***, the difference being in how we deal with ***parameter passing***, i.e., in what state we give an argument to a function at application time. In the case of the λ-calculus, "passing a parameter" means substituting into the body of function.

- In CBV, we evaluate the argument before passing it to the function.
- In CBN, we pass the argument "as is."

This is expressed in small-step semantics in Figures 9 and 10, respectively.

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \textsc{fun} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}\ \textsc{app}$$

Figure 11: Call-by-value big-step semantics for the λ-calculus

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \textsc{fun} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}\ \textsc{app}$$

Figure 12: Call-by-name big-step semantics for the λ-calculus

Exercise 21. *Give traces for both the CBV and CBN evaluation of the following λ-expressions.*

1. $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$
2. $(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.z))$
3. $(\lambda x.\lambda y.xy)((\lambda z.z)(\lambda z.z))$

This distinction can also be naturally expressed in big-step semantics (Figure 11 for CBV and 12 for CBN). We write $e_1 \longrightarrow^{\text{CBV}} e_2$ and $e_1 \longrightarrow^{\text{CBN}} e_2$, and $e \Downarrow^{\text{CBV}} v$ and $e \Downarrow^{\text{CBN}} v$ to distinguish between these evaluation relations.

Exercise 22. *Repeat the previous exercise, but instead provide both CBV and CBN big-step derivations.*

Exercise 23. ***(Challenge)*** *Show that the CBV (resp. CBN) big-step semantics is adequate and sound w.r.t. the CBV (resp. CBN) small-step semantics.*

CBV vs. CBN. CBV has the benefit of being predictable; you know you'll be paying up-front for computing the values of the parameters of a function at application time. Likewise, if you end up using that parameter frequently, it has potentially great savings over CBN; every time you use a parameter in the body of a function you have to *recompute* its value under CBN (exactly because you didn't compute it up-front). The benefits are essentially dual in CBN; if you *don't* use a parameter in the body, or you conditionally use it and the condition seldomly succeeds, then you save the cost of computing the value of the parameter up-front.[6]

Exercise 24. *Let e^n denote the left-associative application of k copies of e, e.g.,*

$$I^5 = (\lambda x.x)^5 = (\lambda x.x)(\lambda x.x)(\lambda x.x)(\lambda x.x)(\lambda x.x)$$

Show that $I^n \longrightarrow I^{n-1}$ under both CBV and CBN for $n \geq 2$. Use this to show that $I^n \longrightarrow^\star I$ in $n-1$ steps for CBV and CBN (where $n \geq 2$).

[6] All that said, it's worth noting that it's possible to *simulate* CBN in CBV, but this is perhaps a conversation for another time.

Exercise 25. *Determine the number of steps under CBV and CBN required to evaluate the expression* $(\lambda y.\lambda x.x)I^n$*. Justify your answer.*

Exercise 26. *Determine the number of steps under CBV and CBN required to evaluate the expression* $(\lambda x.x^n)I^n$*. Justify your answer.*