CS442
Module 2: Untyped λ-calculus

University of Waterloo

Winter 2023

> "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."
>
> — *The Revised^5 Report on the Algorithmic Language Scheme [1]*

When studying programming languages, the most important item a programming language theorist will be working with is an underlying model. Such models are usually defined in mathematical logic for a few reasons:

- A mathematical logic provides a succinct and precise representation of the core mechanics, hence we do not need to worry about particular differences and similarities while reasoning about a class of programming languages.
- Using a mathematical logic gives us a framework to prove certain properties about a programming language.

A good model should be as simple as possible, yet powerful enough that we can use it to model a large class of programming languages. For functional languages, such a model does exist; it is known as the λ-calculus (Lambda calculus).

Aside: This is why many functional languages incorporate a λ in their logo.

Our goal in this chapter is to define the λ-calculus and demonstrate its utility in expressing different entities you should already be very familiar with while working with programming languages, e.g. booleans, lists, and natural numbers. λ-calculus itself is actually quite simple, and it uses the power of abstraction to represent all these features. We will look at the semantics of λ-calculus informally in this module. The following module will revisit concepts in this chapter and introduce formal semantics. The module after that will discuss adding types to λ-calculus.

What we will show is that even though λ-calculus has a paucity of concepts, it can nonetheless express all interesting computations. This fact gives programming language designers a baseline understanding for when features of their language are computationally powerful.

1 Introduction and Concepts

This short section will prepare you for the concepts that are to come, because the λ-calculus can seem extremely foreign. Basically, think of this as warning and mental preparation!

The λ-calculus is a mathematical language. In mathematical languages, we generally think of expressions as *equivalent* if they have the same value. For instance, in arithmetic, $2+2$ is equivalent to 4; you can substitute $2+2$ for 4, or vice-versa, and always result in the same value (so long as you're careful about precedence). However, in the λ-calculus and other computational languages, there's a further wrinkle: computation. Since the λ-calculus will be used to represent computation, equivalences can be complex. Thus, we tend not to think of equivalence, but

of computation, with precise "from" and "to" states. In a mathematical sense, $2+2$ is equivalent to 4, but also, if you're actually performing the computation, you would replace $2+2$ with 4, but never replace 4 with $2+2$.

In the λ-calculus, instead of traditional mathematical operators (like $+$), we only have functions and function calls. More precisely, we have *abstractions* and *applications*. Abstractions are written with a λ, the name of the parameter, a dot (.), then the body. Applications work by substitution, so if we "call" the "function" $\lambda x.\, x+2$ with the argument 2, we get $2+2$, by substituting x for 2. However, we have no $+$ or 2 in the λ-calculus; you'll see in the rest of this module how lacking such basics doesn't limit what we can do.

2 Definitions

The syntax of the λ-calculus is as follows, presented in Backus Normal Form (BNF):

$$\begin{aligned}
\langle Expr\rangle &::= \langle Var\rangle \mid \langle Abs\rangle \mid \langle App\rangle \mid (\langle Expr\rangle)\\
\langle Var\rangle &::= a \mid b \mid c \mid \cdots\\
\langle Abs\rangle &::= \lambda\, \langle Var\rangle\, .\, \langle Expr\rangle\\
\langle App\rangle &::= \langle Expr\rangle\, \langle Expr\rangle
\end{aligned}$$

The four rules define the four elements of the syntax of λ-calculus: *expressions*, *variables*, *abstractions*, and *application*.

- An expression—more precisely, a λ-*expression* or λ-*term*—is either a variable, an abstraction, an application, or an expression surrounded by parentheses;
- A variable is generally a single letter, although we might occasionally use longer identifier names for clarity;
- An abstraction is indicated by the leading character λ (Greek lower case letter lambda), and has two parts: the *variable* and an expression (the *body*), separated by a dot (.);
- An application is simply a concatenation of two expressions. The first is called the *rator* and the second is called the *rand*[1].

To bridge these concepts with terms you may be more familiar with, "abstractions" are essentially functions, and "applications" are essentially function calls. But, don't take this equivalence too far: the behavior of abstractions and applications may not match your expectations if you assume they behave exactly as in a programming language you're familiar with.

Since applications are simply expressions concatenated together, we need precedence and associativity rules to understand how to read them. λ-terms are parsed as follows if without parentheses to indicate precedence:

- Abstractions extend as far to the right as possible. For example, $\lambda x.\, xy$ is parsed as $\lambda x.\, (xy)$ and not as $(\lambda x.\, x)y$;
- Applications are left-to-right associative. For example, abc is $(ab)c$, and not $a(bc)$.

Example 1. Here are a few more examples to illustrate the precedence of λ-expressions:

λ-term	Equivalent λ-term, with least parenthesis necessary
$\lambda x.\, ((xy)\lambda z.\, z)$	$\lambda x.\, xy\lambda z.\, z$
$((\lambda x.\, x)y)$	$(\lambda x.\, x)y$
$((xw)(zy))$	$xw(zy)$
$((xy)\lambda x.\, z)$	$xy\lambda x.\, z$

[1] Short for "operator" and "operand".

Exercise 1. Verify that the meaning of the expression will change when parentheses are removed for the terms in the right column.

We've now described the *syntax* of the λ-calculus, but syntax alone doesn't tell us an expression's *meaning*. We will now discuss the meaning of λ-expressions; more specifically, how to "compute" in the λ-calculus. Intuitively, we can see that a λ-expression consists of functions and calls, but more precisely:

- An abstraction $\lambda x.\, E$ denotes the function that takes an argument x and returns the expression E.
- An application MN denotes the function M applied to the argument N.

Note that all abstractions have exactly one argument. We'll see soon that this does not limit the expressibility of the λ-calculus.

The only "type" in the λ-calculus is a function. So, all expressions are understood to be functions, and thus expressions like xy are always legal; in this case, the expression denotes the application of the function x to the argument y.

> **Aside:** Note that we generally don't give the functions defined in lambda calculus a name. That's why many languages use the term "lambda" or "lambda functions" to refer to anonymous (nameless) functions.

For the following sections, we will be talking about the *operational semantics* of λ-calculus in an informal way. The formal introduction of operational semantics will be seen in the next module.

3 Free and Bound Variables

First we will start by discussing the simplest entity: variables. To be specific, we shall determine where variables obtain their meaning, and whether two occurrences of the same name refer to the same variable.

Consider the identity function: the simplest function, that just returns its only parameter. In the λ-calculus, it is denoted as $\lambda x.\, x$. The x inside the body of the abstraction must refer to the same x in the variable position (the argument) of the abstraction. In formal terms, the latter is a *binding occurrence* of the former, and x is a *bound variable*. An occurrence of a variable that is not involved in a binding occurrence is called *free*.

Here are a few more examples to help build your intuition:

Example 2. In the λ-expression $\lambda x.\, x(\lambda z.\, x)y$, both occurrences of x are bound to the abstraction having variable x, and y is free.

Example 3. In the λ-expression $(\lambda x.\, x)(\lambda z.\, x)y$, the first occurrence is x is bound to the abstraction $(\lambda x.\, x)$. The second x and y are free variables.

Example 4. In the λ-expression abc, all variables are free.

Informally, we can see the set of bound variables for an expression E contains all variables which appear inside an abstractions that define them. This informal definition is fine for our understanding, but we will also define this property formally, as a formal definition can be used as a basis for proofs. Formal definitions tend to leverage the structural and recursive nature of the syntax; specifically, such definition will structurally and recursively define the property on every kind of expression. In this case, we need to have one definition each for abstraction, application and variable. Now let's formally define the notions of free and bound:

Definition 1. The set of *bound variables of an expression* E, denoted by $BV[E]$, is defined as follows:

$$\begin{aligned} BV[x] &= \varnothing \\ BV[\lambda x.\, L] &= BV[L] \cup \{x\} \\ BV[MN] &= BV[M] \cup BV[N] \end{aligned}$$

Variable x is *bound in expression* E if $x \in BV[E]$.

We can then define the set of *free variables* in a similar way:

Definition 2. The set of *free variables of an expression* E, denoted by $FV[E]$, is defined as follows:

$$\begin{aligned} FV[x] &= \{x\} \\ FV[\lambda x.\, L] &= FV[L] \setminus \{x\} \\ FV[MN] &= FV[M] \cup FV[N] \end{aligned}$$

Variable x is *free in expression* E if $x \in FV[E]$. An expression E is *closed* if $FV[E] = \varnothing$; that is, an expression is closed if it has no free variables. A closed expression is called a *combinator*.

Note that it is possible for a variable to be both free and bound. However, each *occurrence* of a variable in an expression is either free or bound, but not both. This is why our definition of "closed" depends on FV instead of BV.

Example 5. In the expression $x\lambda x.\, x$, x is both free and bound. the first occurrence of x is free, and the second one is bound.

Example 6. Figure 1 is a λ-expression in which arrows are drawn from each bound variable occurrence to its binding occurrence. Variable occurrences with no corresponding arrow are free.

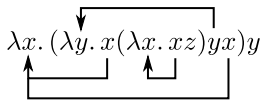


Figure 1: Binding occurrences.

Exercise 2. Provide a modified definition of bound variables, so one can track which expression a bound variable is bound to. Note: a variable can be bound to multiple expressions!

Bound variables get their meaning from the binding occurrences; on the other hand, free variables do not have a meaning within an expression. For free variables to be meaningful, we would have to rely on an external definition, and of course, if we included that external definition as part of the expression, then the variable would now be bound. Thus, an expression being a combinator means that the computation can proceed without any additional information.

Example 7. For analogy, consider the following C function:

```
int f(int x){
    return x + y;
}
```

In the function `f`, `x` is bound to the only parameter and `y` is free. While `f` is valid in C, the computation can only proceed if `y` is defined externally.

4 Substitution and Reduction

Computation in the λ-calculus is based on the notion of *reduction*. An expression is reduced until no further reductions are possible, or some other stopping condition is reached. We will discuss these stopping conditions in detail in the next section. The primary reduction mechanism in the λ-calculus is known as the β-reduction (beta reduction) and is the subject of this section.

First, we identify which expression can be reduced.

Definition 3. An expression of the form $(\lambda x.\, M)N$ is known as a ($\beta-$)redex.

Note that redex is short for *red*ucible *ex*pressions, and the plural of redex is *redices*.

Consider the redex $(\lambda x.\, M)N$ in the definition. Following our usual analogy, $\lambda x.\, M$ is a function with parameter x and argument N. The expectation here is that this evaluates to M with every occurrence of x substituted for N. In fact, since there could be another abstraction in M which defines x again, we only replace each occurrence of x which is bound to outermost x in $\lambda x.\, M$. This is the same as the free occurrences of x inside M.

Note that this substitution style is not how most programming languages are evaluated. Instead, languages have stacks or stores which contain the values of variables, and a variable's value is looked up when the variable is encountered. Substitution is sufficient, and clearer, for the λ-calculus, but we'll see when and why we need to actually store variables in later modules.

Since this process involves substitution, we need to have a formal notation for substitution: Let $M[N/x]$ be the *substitution* of N for all free occurrences of x in M. That is, $M[N/x]$ is the expression M, with all occurrences of the variable x replaced with the expression N. Using this notation, we define β-reduction as follows:

Definition 4. (β-reduction) Let M and N be λ-expressions, x a variable. The relation $\to_\beta$ (*β-reduction*) is defined by the rule

$$(\lambda x.\, M)N \to_\beta M[N/x]$$

Further, if $C[(\lambda x.\, M)N]$ denotes an expression C in which $(\lambda x.\, M)N$ appears as a subterm, then

$$C[(\lambda x.\, M)N] \to_\beta C[M[N/x]]$$

We can describe these processes as "$(\lambda x.\, M)N$ β-reduces to $M[N/x]$" and "in C, $(\lambda x.\, M)N$ β-reduces to $M[N/x]$", respectively.

There are a few things that are worth noting in this definition:

- The notation $C[M]$ refers to *a specific occurrence* of subterm M in C, not to all occurrences of the subterm. Thus, if $E \to_\beta E'$, then $C[E] \to_\beta C[E']$ means the reduction of a single occurrences of E in C, even if E happens to appear multiple times.
- This definition does not specify *which* redex to take for reduction; any valid redex inside an expression can be chosen to be reduced. We will discuss later how we choose which redex to reduce.

We will also introduce the following notations:

- $\to_\beta^n$ denotes the application of exactly n steps of β-reduction;
- $\to_\beta^*$ denotes the application of 0 or more steps of β-reduction;
- $\to_\beta^+$ denotes the application of 1 or more steps of β-reduction;
- $\leftarrow_\beta$ denotes β-expansion: $A \leftarrow_\beta B$ if and only if $B \to_\beta A$;
- $=_\beta$ denotes β-equivalence: $A =_\beta B$ if and only if A can be converted to B by some (possibly empty) sequence of applications of $\to_\beta$ and $\leftarrow_\beta$.

Repeated application of β reduction ($\to_\beta^n$, $\to_\beta^*$, $\to_\beta^+$) is, in essence, our computation. The concepts of β-expansion and β-equivalence will give us a starting place to discuss whether two computations are equivalent, even if they're not equivalently written.

Now, we need to take a step back and look at substitution. The intuition here is to just replace every free occurrence of a variable, say x, with an expression, say T, in an expression E. Remember that every expression is a variable, an abstraction, or an application, the latter two of which can contain subexpressions. So, we need to describe the process of substitution for each of these cases:

- If E is a variable and the variable is x, replace the variable;
- If E is a variable and the variable is not x, E does not change;
- If E is an application, perform substitution on the rator and rand;
- If E is an abstraction and the variable for it is x, E does not change since occurrences of x must not be free;
- If E is an abstraction and the variable for it is not x, perform the substitution on the body of the abstraction.

As usual, we want a formal definition:

Definition 5. (Substitution, provisional) Let E and T be λ-expression and x be a variable, Denote $E[T/x]$ the substitution of T for x in E, defined below:

$$\begin{aligned} x[T/x] &= T \\ y[T/x] &= y \text{ (if } y \neq x) \\ (MN)[T/x] &= M[T/x]N[T/x] \\ (\lambda x.\, M)[T/x] &= \lambda x.\, M \\ (\lambda y.\, M)[T/x] &= \lambda y.\, M[T/x] \end{aligned}$$

Here are a few examples of β-reduction and substitution in action:

Example 8.

$$\begin{aligned} (\lambda x.\, x)a &\to_\beta x[a/x] \\ &= a \end{aligned}$$

Example 9.

$$\begin{aligned} (\lambda x.\, \lambda y.\, x)ab &\to_\beta (\lambda y.\, x)[a/x]b \\ &= (\lambda y.\, x[a/x])b \\ &= (\lambda y.\, a)b \\ &\to_\beta a[b/y] \\ &= a \end{aligned}$$

Example 10.

$$\begin{aligned} (\lambda x.\, \lambda y.\, y)ab &\to_\beta (\lambda y.\, y)[a/x]b \\ &= (\lambda y.\, y[a/x])b \\ &= (\lambda y.\, y)b \\ &\to_\beta y[b/y] \\ &= b \end{aligned}$$

Earlier, we said that defining abstractions so they only take one parameter will not hinder expressibility. The last two examples illustrate this point: $\lambda x.\, \lambda y.\, x$ is an abstraction which can be used like a function which takes two expressions as arguments and produces the first one; similarly, $\lambda x.\, \lambda y.\, y$ is an abstraction that takes two expressions as arguments and produces the second one. The difference between those two functions is that $(\lambda x.\, \lambda y.\, x)a$ reduces to $\lambda y.\, a$, which produces the first argument passed to it when any second argument is passed, while the second function produces an identity function $(\lambda y.\, y)$, regardless of what the argument for parameter x is. This style of reduction-by-substitution allows us to build multi-parameter functions as just a special case of single-parameter functions.

> **Aside:** You might have heard the term *currying*, named after Haskell Curry, which is the process that converts a function taking multiple arguments into nested one-parameter functions which return functions accepting the remaining arguments. In λ-calculus, this is the most natural style of passing multiple arguments.

Also earlier, we said that the identity function is $(\lambda x.\, x)$, and now it is $(\lambda y.\, y)$. Similar to functions, if we replace the identifier in the variable part of an abstraction and their bounded occurrences with another identifier, the behavior of the abstraction should be the same. We express this observation with a principle known as α-conversion (alpha conversion):

Definition 6. (α-conversion) Let E be a λ-expression. We define the relation $=_\alpha$ (α-equivalence) by the rule:

$$\lambda x.\, E =_\alpha \lambda y.\, E[y/x]$$

given that $y \notin FV[E]$. If $C[M]$ is an expression C containing M as a subterm, and $M =_\alpha N$, then $C[M] =_\alpha C[N]$. Further, $=_\alpha$ is an equivalence relation. Finally, α-conversion is the replacement of a term with an α-equivalent term.

Applying α-conversions to an expression should not change its behavior; therefore, α-equivalent expressions should have the same meaning... or do they? Consider the following pairs of reductions:

$$\begin{aligned}(\lambda x.\,\lambda y.\,x)ab &\to_\beta (\lambda y.\,x)[a/x]b \\ &= (\lambda y.\,x[a/x])b \\ &= (\lambda y.\,a)b \\ &\to_\beta a[b/y] \\ &= a\end{aligned}$$

$$\begin{aligned}(\lambda x.\,\lambda y.\,x)yz &\to_\beta (\lambda y.\,x)[y/x]z \\ &= (\lambda y.\,\underline{x[y/x]})z \\ &= (\lambda y.\,\underline{y})z \\ &\to_\beta y[z/y] \\ &= z\end{aligned}$$

As noted earlier, the expectation was $\lambda x.\,\lambda y.\,x$ to behave like a function that given two arguments will produce the first one, yet for the second example we have seen the opposite behavior! The difference between the two examples is:

- On the left, a is a free variable and remains free after the first β-reduction and substitution.
- On the right, the first argument is y (and yes, this is intentionally chosen), which is a free variable. After the first β-reduction and substitution, y becomes bound to the abstraction $\lambda y.\,y$.

Of course, if we were to replace $\lambda x.\,\lambda y.\,x$ with some α-equivalent expression such as $\lambda c.\,\lambda d.\,c$, then the two reductions would behaving correctly again.... but what about $(\lambda c.\,\lambda d.\,c)dc$?

What actually happened is that the binding occurrence of x in the right example is changing. Before reduction, x is bound to the x in the outer abstraction. However, after the reduction against y, the binding occurrence for x, which is now a y, changed, so that this x is now bound to the inner abstraction. We call this behavior *dynamic binding*. More specifically, dynamic binding refers to systems where its binding occurrence could change in the middle of a reduction. While dynamic binding is useful in some cases, it is undesirable for now. We want *static binding* instead: binding occurrences should never change throughout the computation. Thus, a change in the definition of substitution is required:

Definition 7. (Substitution, corrected) Let E and T be λ-expression and x be a variable, Denote $E[T/x]$ the substitution of T for x in E, defined below:

$$\begin{aligned}x[T/x] &= T \\ y[T/x] &= y \text{ (if } y \neq x) \\ (MN)[T/x] &= M[T/x]N[T/x] \\ (\lambda x.\,M)[T/x] &= \lambda x.\,M \\ (\lambda y.\,M)[T/x] &= \lambda y.\,M[T/x] \text{ (if } y \neq x, y \notin FV[T]) \\ (\lambda y.\,M)[T/x] &= \lambda z.\,M[z/y][T/x] \text{ (if } y \neq x, y \in FV[T];\ z \text{ is a ``new'' variable)}\end{aligned}$$

This definition is mostly the same as the previous one, except for one part: instead of letting the abstraction capture the variable, we rename the variable in the abstraction and the bounded occurrences beforehand to some name that was never used before. Of course, a new variable would never have free occurrences in T above, so the capturing behavior will never occur.

We will demonstrate our new, now correct, substitution, by recomputing the β-reduction of $(\lambda x.\,\lambda y.\,x)yz$ above:

Example 11.

$$\begin{aligned}(\lambda x.\,\lambda y.\,x)yz &\to_\beta (\lambda y.\,x)[y/x]z \\ &= (\lambda a.\,x[a/y][y/x])z \\ &= (\lambda a.\,x[y/x])z \\ &= (\lambda a.\,y)z \\ &\to_\beta y[z/a] \\ &= y\end{aligned}$$

5 Reduction and Normal Forms

We will now take a closer look at how computation proceeds in the λ-calculus. Essentially, computation in the λ-calculus is a series of reductions. But, we have not yet decided what to reduce and what not to reduce. Let's start with an obvious option: reduce everything. If the expression contains a β-redex, we β-reduce it, and repeat the process until no β-redex is found in the expression.

Definition 8. (β-normal form) A λ-expression with no β-redex is in β-normal form (β-NF).

However, this process may not terminate. It is possible for reduction of a β-redex to produce a β-redex ad infinitum. Thus, not all terms have a β-normal form:

Example 12.

$$(\lambda x.\,xx)(\lambda x.\,xx) \to_\beta xx[(\lambda x.\,xx)/x] = (\lambda x.\,xx)(\lambda x.\,xx)$$

This expression has no β-normal form since reduction by taking the only β-redex yields the original expression.

For that reason, it is worthwhile to consider other kinds of reduction rules, or reduce to other normal forms. For example, recall that sometimes you might be tempted to write this in Racket:

```
(map (lambda (x) (f x)) lst)
```

While it is always recommended to do this instead:

```
(map f lst)
```

Here is an intuition: if two functions accept the same set of values S as argument and produce the same value when supplied the same argument, then these two functions are equal. Clearly, `(lambda (x) (f x))` and `f` are the same function by this intuition, since the expression `(equal? ((lambda (x) (f x)) a) (f a))` would be true for any given valid input `a`. Put in λ-calculus, $(\lambda x.\,fx)y = fy$, since all we're doing is passing through the y as x, and thus, $\lambda x.\,fx = f$.

> **Aside:** This intuition is called *function extensionality.*

We will formalize this intuition in λ-calculus by defining another kind of reduction, called η-reduction (eta reduction):

Definition 9. (η-reduction) η-reduction is denoted by the following rule:

$$\lambda x.\,Mx \to_\eta M \text{ (if } x \notin FV[M])$$

If $C[M]$ denotes an expression in which M occurs as a subterm, and $M \to_\eta M'$ then $C[M] \to_\eta C[M']$.

Analogously to the definition of β-expansion and β-conversion, we can define an η-expansion relation, $\leftarrow_\eta$ and η-conversion relation, $=_\eta$. Note that η-redices are not reduced during β-reduction, and β-redices are not reduced during η-reduction. A reduction in which both reductions may occur is called $\beta\eta$-reduction.

Our goal is to reach some stopping condition, and we wish to define that stopping condition syntactically. These conditions are, in general, referred as *normal forms*. The normal form which we will spend most of the time talking about is β-normal form; however, other normal forms exists as well. We can speak, for example, of η-normal form and of $\beta\eta$-normal form.

Exercise 3. Formally define η-normal form and $\beta\eta$-normal form.

Among the various alternative definitions of normal form that have been studied, the most important for our purpose in known as *weak normal form* (WNF), defined below:

Definition 10. An expression E is in *weak normal form* (WNF) if every β-redex in E lies within the body of some abstraction.

The intuition behind WNF is that, in real programming languages, computation does not occur inside a function until it is called. Hence, we do not consider redices that occur inside an abstraction as candidates for reduction until the abstraction itself has been supplied with an argument and reduced, at which point the redices inside become exposed.

Example 13. The term $\lambda x.\,(\lambda y.\,y)(\lambda z.\,(\lambda w.\,w)z)$ is in WNF but not in β-normal form. The term contains two β-redices: $(\lambda y.\,y)(\lambda z.\,(\lambda w.\,w)z)$ and $(\lambda w.\,w)z$, but the former lies within the " λx " abstraction, and the latter lies within the " λz " abstraction.

Every term in β-normal form is also in WNF, as WNF is a strictly weaker criteria.

Whenever we speak of "normal form" without qualification, we are referring to β-normal form.

6 Order of Evaluation

6.1 The Church-Rosser Theorem

So far, we have not specified *which* redex to choose at any given time. For example, we have two choices of β-redex to reduce in the expression $(\lambda y.\,y)((\lambda x.\,x)b)$.

$$\begin{aligned}(\lambda y.\,y)((\lambda x.\,x)b) &\to_\beta (\lambda y.\,y)(x[b/x]) \\ &= (\lambda y.\,y)b \\ &\to_\beta y[b/y] \\ &= b\end{aligned}$$

$$\begin{aligned}(\lambda y.\,y)((\lambda x.\,x)b) &\to_\beta y[(\lambda x.\,x)b/y] \\ &= (\lambda x.\,x)b \\ &\to_\beta x[b/x] \\ &= b\end{aligned}$$

In this case, the two reductions taking different paths have reduced to the same expression. Is this true for all of λ-expressions which have a normal form? Luckily, the answer is yes! This theorem was proved by Alonzo Church and J.Barkley Rosser [2].

Theorem 1. (Church-Rosser, 1936) For λ-expression E_1, E_2, and E_3, if $E_1 \to_\beta^* E_2$ and $E_1 \to_\beta^* E_3$, then there exists an expression E_4 such that $E_2 \to_\beta^* E_4$ and $E_3 \to_\beta^* E_4$ (up to α-equivalence).

The proof is provided in a separate document, but you don't have to be familiar with it:

Aside: Proof of Church-Rosser Theorem (https://student.cs.uwaterloo.ca/~cs442/W23/extras/c-r-thm-proof.pdf)

The Church-Rosser Theorem is one of the fundamental theoretical results about λ-calculus. As illustrated in Figure 2, it guarantees that when faced with a choice of redex, it is always possible to arrive at the same final expression regardless of your choice. More precisely, if an expression E_1 can be reduced by zero or more reduction steps to either expression E_2 and E_3, then there exists some other expression E_4 to which both E_2 and E_3 can be reduced.

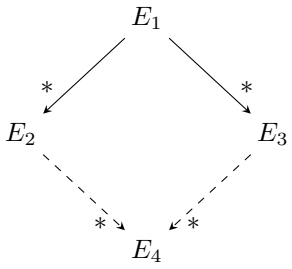


Figure 2: The Church-Rosser Theorem. Solid Arrow indicate the hypotheses, and dashed arrows indicate the reductions whose existence the theorem guarantees.

This idea implies that there is a unique normal form for any expression. The theorem says that E_2 and E_3 can be reduced to some other expression, and normal forms are irreducible by definition. One important thing to note is that the Church-Rosser Theorem does not guarantee the *existence* of β-normal form; the theorem just indicates that *if* the reduction terminates, it will reach a unique normal form. We summarize this idea in the following corollary:

Corollary 1. A λ-expression can reduce to at most one β-normal form (up to α-equivalence).

Proof. Let E_1 be an expression that reduces to normal forms E_2 and E_3. By the Church-Rosser Theorem, there is an expression E_4 such that $E_2 \to_\beta^* E_4$ and $E_3 \to_\beta^* E_4$ (up to α-equivalence). However, E_2 and E_3 are both normal forms, hence irreducible. Therefore, the only possible reduction to E_4 from E_2 and E_3 must take zero steps, so we have $E_2 =_\alpha E_3 =_\alpha E_4$. □

Although the Church-Rosser Theorem guarantees a unique β-normal form, a λ-expression may have several different instances of other kinds of normal forms.

Exercise 4. Prove or disprove the following statement:

For λ-expression E_1, E_2, and E_3, if $E_1 \to_\beta E_2$ and $E_1 \to_\beta E_3$, then there exists an expression E_4 such that $E_2 \to_\beta E_4$ and $E_3 \to_\beta E_4$ (up to α-equivalence).

Note the absence of asterisk as superscript for all $\to_\beta$ operators.

6.2 Reduction Strategies

Although the Church-Rosser Theorem guarantees that no matter how we choose redices to reduce, we can never reach an expression from which we can't reach the unique β-normal form given that one exists. However, most real-world programming languages have much more clearly defined policies regarding the order in which computation proceeds, because order of evaluation can affect the meaning of programs in most languages. We are going to see that even in λ-calculus, the policy for choice of redex can matter when dealing with expressions that could have possibly infinite reduction paths. We will examine two such policies, known as *reduction strategies*, for the λ-calculus. A reduction strategy is a policy which, given a λ-term, decides which redex to reduce next. We will see that even in the presence of the Church-Rosser Theorem, the two strategies have quite different properties.

We first consider the reduction strategy you commonly see in "mainstream" programming languages called *Applicative Order Reduction*(AOR): we will always choose the leftmost, innermost redex at a step. A redex is *innermost* if it contains no other redices.

Example 14.

$$\begin{aligned} & (\lambda x.\, fx)((\lambda y.\, gy)z) \\ \to_\beta \; & (\lambda x.\, fx)((gy)[z/y]) \\ = \; & (\lambda x.\, fx)(gz) \\ \to_\beta \; & (fx)[gz/x] \\ = \; & f(gz) \end{aligned}$$

In this example we can see why AOR is similar to the programming languages you have seen[2]: the argument to an abstraction is reduced to normal form before it is substituted. For this reason, AOR is sometimes dubbed as *call-by-value*, and demostrates a semantic property called *eager evaluation*.

Here is another example with AOR:

Example 15.

$$\begin{aligned} & (\lambda x.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \\ \to_\beta \; & (\lambda x.\, y)((xx)[(\lambda x.\, xx)/x]) \\ = \; & (\lambda x.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \end{aligned}$$

This example does, in fact, have a normal form: y. Yet we cannot reach it using the AOR strategy. The primary goal for reductions is to reach the normal form, but AOR fails to do so in this particular case.

Let us consider another reduction strategy, known as *Normal Order Reduction*(NOR). This time, we always choose the leftmost, outermost redex at each step. A redex is *outermost* if it is not contained in any other redex. Here is $(\lambda x.\, fx)((\lambda y.\, gy)z)$ reduced in NOR:

Example 16.

$$\begin{aligned} & (\lambda x.\, fx)((\lambda y.\, gy)z) \\ \to_\beta \; & (fx)[(\lambda y.\, gy)z/x] \\ = \; & f((\lambda y.\, gy)z) \\ \to_\beta \; & f((gy)[z/y]) \\ = \; & f(gz) \end{aligned}$$

One of the properties of NOR is that arguments to a function are not evaluated until they are needed. Above, at each reduction step, the formal parameter is replaced verbatim with the argument—that is, without simplifying the argument further before substitution. NOR is sometimes called *call-by-name*, and demonstrates a semantic property called *lazy evaluation*, which we will explore later.

Under NOR, $(\lambda x.\, y)((\lambda x.\, xx)(\lambda x.\, xx))$ reduces to the normal form correctly:

Example 17.

$$\begin{aligned} & (\lambda x.\, y)((\lambda x.\, xx)(\lambda x.\, xx)) \\ \to_\beta \; & y[(\lambda x.\, xx)(\lambda x.\, xx)/x] \\ = \; & y \end{aligned}$$

As demonstrated, while NOR reduces this expression immediately to y, AOR immediately gets caught in an infinite reduction of the argument, and makes no progress.

[2] Actually, these languages do not completely follow AOR. You'll see how when we discuss conditionals.

In general, our goal in reducing an expression is to reach the normal form, but we've seen at least one example in which one obvious reduction strategy fails to do so. One might therefore ask whether there is always a reduction strategy that will reach a normal form. Luckily, there is. NOR always reaches the normal form, if one exists:

Theorem 2. (Standardization, 1958) If an expression has a normal form, then Normal Order Reduction is guaranteed to reach it.

The proof of the Standardization Theorem was due to Curry and Feys [3]. We do not present it here.

Thus, even though the Church-Rosser Theorem guarantees that no choice of redex puts the β-normal form (if it exists) out of reach, our choice of reduction strategy is nonetheless important.

Aside: In a purely functional setting, the Church-Rosser Theorem guarantees that the reduction will lead to the unique normal form. However, in any programming languages with state, such as mutable variables or fields of objects or I/O, a difference in the order of evaluation could cause completely different behaviors.

A notorious example in C and C++ is the order of evaluation for arguments of functions and operators. Consider the following C++ code snippet:

```
int f(){ cout << 'f'; return 1; }
int g(){ cout << 'g'; return 2; }
int add(int x, int y){ return x + y; }

int main(){
    int i = add(f(),g());
    cout << endl;
    return i;
}
```

What will be the order of the characters printed? The answer is "depends on the implementation", since all possible permutations of the output are allowed according the C++ standard. The C++ standard does not specify the order of evaluation for arguments of functions and operators. Hence, the choice of which argument to evaluate first is completely up to the particular compiler. For example, if a particular compiler chooses to always evaluate the rightmost argument first, the output would be `gf`.

7 Programming With λ-Calculus

So far, we've introduced λ-calculus as a model of computation. However, we haven't discussed how expressive it is compared to the programming languages you use. Alonzo Church, the inventor of λ-calculus, intended to use λ-calculus to provide a foundation for all of mathematics, but it was shown to be inconsistent for this purpose by Kleene and Rosser in 1935. Nonetheless, Church and Turing had proved that their models of computation, the λ-calculus and the Turing Machine, are equivalent in terms of expressive power. While computation in the Turing Machine is rather cumbersome to manipulate and analyze due to the stateful nature of it, programs in λ-calculus are stateless, making them much easier to perform formal analysis on.

But, it's not obvious in λ-calculus alone whether such analysis would be useful. To prove that λ-calculus is useful, we need to show that it is expressive enough to represent the kinds of computation we might want in real languages.

In this section, we are going to discuss how to imitate real-world programming using λ-calculus, by showing how to implement different (functional) programming constructs. More specifically, we are going to introduce

- How to represent and manipulate values you typically see in programming languages, such as booleans, natural numbers and lists;
- how to implement recursive functions; and
- how to add primitives and associated reduction rules.

Before we start our discussion, it would be nice to have some kind of shorthand notation for "the λ-calculus representation of x" where x is some entity you would see in modern programming languages. We are going to use the double square brackets ($[\![,]\!]$) to denote this correspondence. For example, we show the λ-calculus representation for the identity function (`id` for short) here:

Example 18. The shorthand for "the λ-calculus representation for the identity function is $\lambda x.\, x$" is:

$$[\![\texttt{id}]\!] := \lambda x.\, x$$

Note that this shorthand notation is not a part or extension of the language of λ-calculus; an expression only becomes a λ-expression after we expend all the shorthand notations into their corresponding λ-expression.

It is sometimes also useful to denote some arbitrary expressions; we will use upper case letters for them.

7.1 Booleans and Conditionals

Before we introduce the primitive for Booleans, we should note that Booleans are often used in conditional expressions. Thus, it would be helpful if we picture how to represent conditional expressions before we designate what `true` and `false` are. The simplest conditional expression would look like this:

$$[\![\texttt{if } B \texttt{ then } T \texttt{ else } F]\!]$$

where B is the Boolean value, T and F are the expressions to take if B is true or false, respectively. As every expression is a function in λ-calculus, we can imagine it's a function that takes three arguments: the boolean value and the two expressions. Since the boolean values are also functions, we can let those values be a selector which given two values as arguments, and produce one of them.

$$[\![\texttt{if}]\!] := (\lambda b.\, \lambda t.\, \lambda f.\, btf)$$

Alternatively, with the arguments applied:

$$[\![\texttt{if } B \texttt{ then } T \texttt{ else } F]\!] := (\lambda b.\, \lambda t.\, \lambda f.\, btf)\ [\![B]\!]\ [\![T]\!]\ [\![F]\!]$$

Then it follows that our $[\![\texttt{true}]\!]$ and $[\![\texttt{false}]\!]$ would be, respectively, functions that given two values as arguments, produce the first or the second value.

$$[\![\texttt{true}]\!] := \lambda x.\, \lambda y.\, x$$

$$[\![\texttt{false}]\!] := \lambda x.\, \lambda y.\, y$$

These should look familiar!

To simplify it a bit further, we can just apply β-reduction three times and obtain a representation which is just a concatenation of B, T and F:

$$[\![\texttt{if } B \texttt{ then } T \texttt{ else } F]\!] := [\![B]\!]\ [\![T]\!]\ [\![F]\!]$$

We are going to use this concatenation-based representation to cut down our number of reductions needed.

Modelling conditionals in this way might seem strange, but we can show that it works by tracing the reduction with B being `true` or `false`.

$$\begin{aligned}
[\![\texttt{if true then } T \texttt{ else } F]\!] &:= [\![\texttt{true}]\!]\ [\![T]\!]\ [\![F]\!] \\
&= (\lambda x.\, \lambda y.\, x)\ [\![T]\!]\ [\![F]\!] \\
&\to_\beta (\lambda y.\ [\![T]\!]\ )\ [\![F]\!] \\
&\to_\beta [\![T]\!]
\end{aligned}$$

$$
\begin{aligned}
[\![\texttt{if false then } T \texttt{ else } F]\!] &:= [\![\texttt{false}]\!]\ [\![T]\!]\ [\![F]\!] \\
&= (\lambda x.\, \lambda y.\, y)\ [\![T]\!]\ [\![F]\!] \\
&\to_\beta (\lambda y.\, y)\ [\![F]\!] \\
&\to_\beta [\![F]\!]
\end{aligned}
$$

Using the construction of boolean values and primitives, we can then build boolean operators as follows:

$$
\begin{aligned}
[\![\texttt{and } p\ q]\!] &= [\![\texttt{if } p \texttt{ then } q \texttt{ else false}]\!] \\
[\![\texttt{and}]\!] &= \lambda p.\, \lambda q.\, pq(\lambda x.\, \lambda y.\, y)
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{or } p\ q]\!] &= [\![\texttt{if } p \texttt{ then true else } q]\!] \\
[\![\texttt{or}]\!] &= \lambda p.\, \lambda q.\, p(\lambda x.\, \lambda y.\, x)q
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{not } p]\!] &= [\![\texttt{if } p \texttt{ then false else true}]\!] \\
[\![\texttt{not}]\!] &= \lambda p.\, p(\lambda x.\, \lambda y.\, y)(\lambda x.\, \lambda y.\, x)
\end{aligned}
$$

We will show that our $[\![\texttt{not}]\!]$ is correct by showing that $[\![\texttt{not true}]\!]$ and $[\![\texttt{not false}]\!]$ behave correctly. Ideally, we want them to evaluate to $[\![\texttt{false}]\!]$ and $[\![\texttt{true}]\!]$, respectively.

Example 19. Reductions of $[\![\texttt{not true}]\!]$ and $[\![\texttt{not false}]\!]$.

$$
\begin{aligned}
[\![\texttt{not true}]\!] &= [\![\texttt{not}]\!]\ [\![\texttt{true}]\!] \\
&= (\lambda b.\, (b\ [\![\texttt{false}]\!]\ [\![\texttt{true}]\!]))\ [\![\texttt{true}]\!] \\
&\to_\beta [\![\texttt{true}]\!]\ [\![\texttt{false}]\!]\ [\![\texttt{true}]\!] \\
&= (\lambda x.\, \lambda y.\, x)(\lambda x.\, \lambda y.\, y)\ [\![\texttt{true}]\!] \\
&=_\alpha (\lambda x.\, \lambda y.\, x)(\lambda x.\, \lambda z.\, z)\ [\![\texttt{true}]\!] \\
&\to_\beta (\lambda y.\, (\lambda x.\, \lambda z.\, z))\ [\![\texttt{true}]\!] \\
&\to_\beta (\lambda x.\, \lambda z.\, z) \\
&= [\![\texttt{false}]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{not false}]\!] &= [\![\texttt{not}]\!]\ [\![\texttt{false}]\!] \\
&= (\lambda b.\, (b\ [\![\texttt{false}]\!]\ [\![\texttt{true}]\!]))\ [\![\texttt{false}]\!] \\
&\to_\beta [\![\texttt{false}]\!]\ [\![\texttt{false}]\!]\ [\![\texttt{true}]\!] \\
&= (\lambda x.\, \lambda y.\, y)(\lambda x.\, \lambda y.\, y)\ [\![\texttt{true}]\!] \\
&\to_\beta (\lambda y.\, y)\ [\![\texttt{true}]\!] \\
&\to_\beta [\![\texttt{true}]\!]
\end{aligned}
$$

Exercise 5. Show that $[\![\texttt{and}]\!]$ and $[\![\texttt{or}]\!]$ work as expected.

Here is a question: what is the difference between the `if` you see here and the one you see in real-life programming languages? Consider the AOR strategy. Our $[\![\texttt{if true then } T \texttt{ else } F]\!]$ will actually reduce both the true and false branches before returning to the one we want. Similarly, our `and` and `or` constructions don't perform short-circuit evaluation either. While this isn't what a mainstream programming language would do, the Church-Rosser theorem provides us with the guarantee that as long as there are no infinite reductions on either branch, we will be able to reach the desired expression. Considering the absence of side effects in λ-calculus, the only downside of evaluating both branches would be the decrease in efficiency. We will discuss implementing short-circuit evaluation later.

Aside: The fact that booleans are represented by their behavior should also look familiar if you've read through the Smalltalk module. In a way, Smalltalk's booleans are a lot like λ-calculus booleans!

Video 2.1 (https://student.cs.uwaterloo.ca/~cs442/W23/videos/2.1/): Lambda calculus booleans

7.2 Pairs and Lists

Programs generally require storage facilities in order to compute their results. As you saw in your first-year Racket course (if you're a Waterloo undergrad), the simplest expandable storage facility is the list. However, we will start by talking about the fundamental data structure, *pair*, which is just a combination of two data values. Like in Racket, by nesting pairs within each other, we can create lists with arbitrary length, as well as trees. We'll refer to the two elements of a pair as its *head* and its *tail*[3], respectively.

The intuition here is that we're trying to select either `head` or `tail`, given a pair. Hence, a pair is basically a function that has the `head` and `tail` stored in it. In order to access the individual elements, the pair must accept some function, which is called a selector, as a parameter. The selectors will be functions that, given two parameters, produce the former (i.e. the head) or the latter (i.e. the tail). Hey, doesn't that sound familiar? That's what our `true` and `false` are doing!

$$\begin{aligned}[\![\langle h, t\rangle]\!] &:= \lambda s.\ [\![\texttt{if}\ \ s\ \ \texttt{then}\ \ h\ \ \texttt{else}\ \ t]\!] \\ &= \lambda s.\ sht\end{aligned}$$

Therefore, the function `head` and `tail`, which actually extract the values from a list, should pass `true` or `false` into the list given as parameter.

$$\begin{aligned}[\![\texttt{head}]\!] &:= \lambda l.\ l\ [\![\texttt{true}]\!] \ = \lambda l.\ l \lambda x.\ \lambda y.\ x \\ [\![\texttt{tail}]\!] &:= \lambda l.\ l\ [\![\texttt{false}]\!] \ = \lambda l.\ l \lambda x.\ \lambda y.\ y\end{aligned}$$

We then implement a function `cons` (from *constructor*) that takes two arguments (the head and the tail) and returns the pair containing them:

$$\begin{aligned}[\![\texttt{cons}]\!] &:= \lambda h.\ \lambda t.\ [\![\langle h, t\rangle]\!] \\ &= \lambda h.\ \lambda t.\ \lambda s.\ sht\end{aligned}$$

[3]These are also referred as `car` and `cdr`, which originates from the implementation of Lisp on the IBM 704. `car` stands for "Contents of Address part of the Register", and `cdr` (pronounced "could-er"), stands for "Contents of the Decrement part of Register", referring to particular register properties of the IBM 704.

We will show that the pairs exhibit the correct behaviour, namely:

$$[\![\texttt{head (cons } A\ B\texttt{)}]\!] = [\![A]\!] \text{ and } [\![\texttt{tail (cons } A\ B\texttt{)}]\!] = [\![B]\!]$$

$$\begin{aligned}
& [\![\texttt{head (cons } A\ B\texttt{)}]\!] \\
= \ & [\![\texttt{head}]\!]\ (\ [\![\texttt{cons}]\!]\ [\![A]\!]\ [\![B]\!]\ ) \\
= \ & [\![\texttt{head}]\!]\ ((\lambda h.\, \lambda t.\, \lambda s.\, sht)\ [\![A]\!]\ [\![B]\!]\ ) \\
\to_\beta^2 \ & [\![\texttt{head}]\!]\ (\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ ) \\
= \ & (\lambda l.\, l\lambda x.\, \lambda y.\, x)(\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ ) \\
\to_\beta \ & (\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ )\lambda x.\, \lambda y.\, x \\
\to_\beta \ & (\lambda x.\, \lambda y.\, x)\ [\![A]\!]\ [\![B]\!] \\
\to_\beta^2 \ & [\![A]\!]
\end{aligned}$$

$$\begin{aligned}
& [\![\texttt{tail (cons } A\ B\texttt{)}]\!] \\
= \ & [\![\texttt{tail}]\!]\ (\ [\![\texttt{cons}]\!]\ [\![A]\!]\ [\![B]\!]\ ) \\
= \ & [\![\texttt{tail}]\!]\ ((\lambda h.\, \lambda t.\, \lambda s.\, sht)\ [\![A]\!]\ [\![B]\!]\ ) \\
\to_\beta^2 \ & [\![\texttt{tail}]\!]\ (\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ ) \\
= \ & (\lambda l.\, l\lambda x.\, \lambda y.\, y)(\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ ) \\
\to_\beta \ & (\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ )\lambda x.\, \lambda y.\, y \\
\to_\beta \ & (\lambda x.\, \lambda y.\, y)\ [\![A]\!]\ [\![B]\!] \\
\to_\beta^2 \ & [\![B]\!]
\end{aligned}$$

Finally, we need a way to denote the empty list ($[\![\texttt{nil}]\!]$). In fact, we're going to work with what we have first: how can we tell that a list is not empty? Well, the selectors used in a pair can choose to produce other expressions than the head or the tail; in this case, the selector will produce $[\![\texttt{false}]\!]$ when given the head and the tail as parameters:

$$[\![\texttt{null?}]\!] := \lambda l.\, l\lambda h.\, \lambda t.\ [\![\texttt{false}]\!]$$

This guarantees when given a non-empty list (i.e. a pair), it will always produce `false` (we are going to show that soon). Then it follows that we can just let $[\![\texttt{nil}]\!]$ be something that makes $[\![\texttt{null?}]\!]$ return `true`. As a reminder, a pair is a function that accepts a selector and passes the two parts of the pair into the selector, in the hope that the selector returns one of the parts. In this case, the empty list is just going to ignore the selector (i.e., not use it) and just produce true.

$$[\![\texttt{nil}]\!] := \lambda s.\ [\![\texttt{true}]\!]$$

Again, as verification, we wish our constructions to behave correctly:

$$[\![\texttt{null? nil}]\!] = [\![\texttt{true}]\!] \text{ and } [\![\texttt{null? (cons } A\ B\texttt{)}]\!] = [\![\texttt{false}]\!]$$

$$\begin{aligned}
& [\![\texttt{null? nil}]\!] \\
= \ & \lambda l.\, l\lambda h.\, \lambda t.\ [\![\texttt{false}]\!]\ (\lambda s.\ [\![\texttt{true}]\!]\ ) \\
\to_\beta \ & (\lambda s.\ [\![\texttt{true}]\!]\ )\lambda h.\, \lambda t.\ [\![\texttt{false}]\!] \\
\to_\beta \ & [\![\texttt{true}]\!]
\end{aligned}$$

$$\begin{aligned}
& [\![\texttt{null? (cons A B)}]\!] \\
= \ & [\![\texttt{null?}]\!]\ (\ [\![\texttt{cons}]\!]\ [\![A]\!]\ [\![B]\!]\ ) \\
\to_\beta^2 \ & [\![\texttt{null?}]\!]\ \lambda s.\, s\ [\![A]\!]\ [\![B]\!] \\
= \ & (\lambda l.\, l\lambda h.\, \lambda t.\ [\![\texttt{false}]\!]\ )\lambda s.\, s\ [\![A]\!]\ [\![B]\!] \\
\to_\beta \ & (\lambda s.\, s\ [\![A]\!]\ [\![B]\!]\ )\lambda h.\, \lambda t.\ [\![\texttt{false}]\!] \\
\to_\beta \ & (\lambda h.\, \lambda t.\ [\![\texttt{false}]\!]\ )\ [\![A]\!]\ [\![B]\!] \\
\to_\beta^2 \ & [\![\texttt{false}]\!]
\end{aligned}$$

Similar to what you have seen in Racket, a nested construction would allow us to construct lists of arbitrary length. Here is an example:

Example 20. Construction of a list `(list` a b c`)`:

$$\begin{aligned}
[\![\texttt{(list } a \ b \ c\texttt{)}]\!] &= [\![\texttt{(cons } a \ \texttt{(cons } b \ \texttt{(cons } c \ \texttt{nil)))}]\!] \\
&= [\![\texttt{cons}]\!]\, a(\, [\![\texttt{cons}]\!]\, b(\, [\![\texttt{cons}]\!]\, c\, [\![\texttt{nil}]\!]\,)) \\
&= (\lambda h.\, \lambda t.\, \lambda s.\, sht) a((\lambda h.\, \lambda t.\, \lambda s.\, sht) b((\lambda h.\, \lambda t.\, \lambda s.\, sht) c\, [\![\texttt{nil}]\!]\,)) \\
&\to_\beta^6 \lambda s.\, sa\lambda s.\, sb\lambda s.\, sc\, [\![\texttt{nil}]\!] \\
&= \lambda s.\, sa\lambda s.\, sb\lambda s.\, sc\lambda x.\, \lambda x.\, \lambda y.\, x
\end{aligned}$$

Video 2.2 (https://student.cs.uwaterloo.ca/~cs442/W23/videos/2.2/): Lambda calculus pairs and lists

Exercise 6. Define the function $[\![\texttt{second}]\!]$ [4], which gets the second element from the list.

7.3 Numbers

After introducing lists, it's easy to represent numbers: just represent them using lists! An empty list would be 0, a list of one element would be 1, etc.

Exercise 7. Define the following entities using this idiom: $[\![\texttt{0}]\!]$, $[\![\texttt{1}]\!]$, $[\![\texttt{pred}]\!]$, $[\![\texttt{succ}]\!]$, $[\![\texttt{isZero?}]\!]$, where $[\![\texttt{pred } n]\!]$ and $[\![\texttt{succ } n]\!]$ are functions that produce the number that's one less or one more than n respectively. Verify that your solution works by showing that

$$[\![\texttt{pred (succ } n\texttt{)}]\!] = [\![n]\!] \text{ (if } n \neq 0\text{)}$$

However, there is a cleverer solution introduced by Alonzo Church called *Church numerals*. In Church's representation of numbers, $[\![n]\!]$ is defined as a function that takes a function f and a value x[5], and produces the result of f applied n times to its argument x. More specifically:

$$\begin{aligned}
[\![\texttt{0}]\!] &= \lambda f.\, \lambda x.\, x \\
[\![\texttt{1}]\!] &= \lambda f.\, \lambda x.\, fx \\
[\![\texttt{2}]\!] &= \lambda f.\, \lambda x.\, f(fx) \\
[\![\texttt{3}]\!] &= \lambda f.\, \lambda x.\, f(f(fx))
\end{aligned}$$

We will consider how to perform basic arithmetic with the Church numerals (except division, which is rather complicated). First, let's consider addition. Given two Church numerals m and n, we want to have

$$[\![m+n]\!] = \lambda f.\, \lambda x.\, \underbrace{f \cdots\cdots\cdots f}_{m+n \text{ occurrences of } f}\ x$$

A way to get to this is to apply f n times on x, and then apply f m times to the result. Our addition function will accept two arguments, m and n:

$$[\![\texttt{+}]\!] := \lambda m.\, \lambda n.\, \cdots$$

Then, it must produce a number, which is a function that takes some f and some x:

$$[\![\texttt{+}]\!] := \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, \cdots$$

[4] Also called conveniently `cadr`, which is `car` of the `cdr`.
[5] Technically speaking, they are both functions. Everything in λ-calculus is a function.

Applying f n times on x would just be nfx, since n itself is a Church numeral that takes f and x:

$$[\![\texttt{+}]\!] := \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, \cdots (nfx)$$

At last, apply the result of nfx to f by m times:

$$[\![\texttt{+}]\!] := \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, mf(nfx)$$

To demonstrate that it works, we will show $2 + 3 = 5$. Starting at this point, we will highlight some parts in a few reduction steps to clearly illustrate what reduction to take.

$$\begin{aligned}
[\![2+3]\!] &= [\![\texttt{+}]\!]\ [\![2]\!]\ [\![3]\!] \\
&= (\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, mf(nfx))\ [\![2]\!]\ [\![3]\!] \\
&\to^2_\beta \lambda f.\, \lambda x.\ [\![2]\!]\ f(\ [\![3]\!]\ fx) \\
&= \lambda f.\, \lambda x.\, \underbrace{(\lambda f.\, \lambda x.\, f(fx))}_{\text{function}}\ \underbrace{f}_{\text{arg 1}}\ \underbrace{(\ [\![3]\!]\ fx)}_{\text{arg 2}} \\
&\to^2_\beta \lambda f.\, \lambda x.\, f(f(\ [\![3]\!]\ fx)) \\
&= \lambda f.\, \lambda x.\, f(f(\underbrace{(\lambda f.\, \lambda x.\, f(f(fx)))}_{\text{function}}\ \underbrace{f}_{\text{arg 1}}\ \underbrace{x}_{\text{arg 2}})) \\
&\to^2_\beta \lambda f.\, \lambda x.\, f(f(f(f(fx)))) \\
&= [\![5]\!]
\end{aligned}$$

A special case of addition is the $[\![\texttt{succ}]\!]$ function, which produces the number that's one more than a given number (the successor):

$$[\![\texttt{succ n}]\!] := [\![n+1]\!] = \lambda n.\, \lambda f.\, \lambda x.\, nf(fx)$$

Video 2.3 (https://student.cs.uwaterloo.ca/~cs442/W23/videos/2.3/): Lambda calculus numbers and addition

Addition was pretty simple, but subtraction is harder. While it's easy to take away a number in real-world mathematics, it's impossible to just "take away" a function application from an expression. There is no way for us to "undo" a function application in λ-calculus. An alternative arises from the observation that `succ` is easy. We can instead use a pair $\langle a, b\rangle$ to track a number a and its successor. For each given pair $\langle a, b\rangle$, we can create a new pair $\langle a+1, a\rangle$. Start with $\langle 0, 0\rangle$ and applying this procedure n times, we would obtain $\langle n, n-1\rangle$. Taking the tail of the resulting pair gives us the predecessor of n. Then, we "simply" need to take the predecessor the correct number of times.

Let's get started. A function $[\![\texttt{pred}]\!]$ should take one argument, the number n, and the value produced should be a number as well. That is, the value produced should be in the form of $\lambda f.\, \lambda x.$:

$$[\![\texttt{pred}]\!] := \lambda n.\, \lambda f.\, \lambda x.\, \cdots$$

We need to get the tail of the complicated computation stated above. The result from that computation gives us a number which takes two parameters, so we need to also pass the parameter of our newly made number into there:

$$[\![\texttt{pred}]\!] := \lambda n.\, \lambda f.\, \lambda x.\, (\ [\![\texttt{tail}]\!]\ \cdots) fx$$

We can η-reduce now and get rid of f and x.

$$[\![\texttt{pred}]\!] := \lambda n.\, (\ [\![\texttt{tail}]\!]\ \cdots)$$

The computation itself would repeat n times. But, repeating n times is just the number n in Church numerals:

$$[\![\texttt{pred}]\!] := \lambda n.\,(\,[\![\texttt{tail}]\!]\,(n(\cdots)))$$

Then the computation should take a pair, and the starting pair is $\langle 0, 0\rangle$:

$$[\![\texttt{pred}]\!] := \lambda n.\,(\,[\![\texttt{tail}]\!]\,(n(\lambda p.\cdots)\,[\![\langle 0,0\rangle]\!]\,)) = \lambda n.\,(\,[\![\texttt{tail}]\!]\,(n(\lambda p.\cdots)(\,[\![\texttt{cons}]\!]\;[\![\texttt{0}]\!]\;[\![\texttt{0}]\!]\,)))$$

Then fill in the computation itself:

$$[\![\texttt{pred}]\!] := \lambda n.\,(\,[\![\texttt{tail}]\!]\,(n(\lambda p.\;[\![\texttt{cons}]\!]\,(\,[\![\texttt{succ}]\!]\;[\![\texttt{head}\;\;p]\!]\,)(\,[\![\texttt{head}\;\;p]\!]\,))(\,[\![\texttt{cons}]\!]\;[\![\texttt{0}]\!]\;[\![\texttt{0}]\!]\,)))$$

To keep our explanation brief, we will just stop here.

Exercise 8. Verify that our definition of $[\![\texttt{pred}]\!]$ is correct by working through an example of your own.

Subtraction is defined in the following way: given m and n, produce m applied to `pred` n times.

$$[\![\texttt{-}]\!] = \lambda m.\,\lambda n.\,n\,[\![\texttt{pred}]\!]\,m$$

Video 2.4 (https://student.cs.uwaterloo.ca/~cs442/W23/videos/2.4/): Lambda calculus predecessor and subtraction

Multiplication of m and n is defined as the n-fold repetition of f, m times:

$$[\![\texttt{*}]\!] = \lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,m(nf)x \to_\eta \lambda m.\,\lambda n.\,\lambda f.\,m(nf)$$

Exponentiation (m^n) is just the n-fold repetition of m itself:

$$[\![\texttt{\^{}}]\!] = \lambda m.\,\lambda n.\,\lambda f.\,\lambda x.\,nmfx \to_\eta^2 \lambda m.\,\lambda n.\,nm$$

7.4 Recursion

Suppose one wants to find length of a list in λ-calculus. Being an expert in recursion, they write this down:

$$\begin{aligned}[\![\texttt{len}]\!] &:= \lambda l.\;[\![\texttt{if (null?}\;\;l\texttt{) then 0 else (succ (len (tail }l\texttt{)))}]\!] \\ &= \lambda l.\,(\,[\![\texttt{null?}]\!]\,l)\,[\![0]\!]\,(\,[\![\texttt{succ}]\!]\,(\,[\![\texttt{len}]\!]\,(\,[\![\texttt{tail}]\!]\,l)))\end{aligned}$$

It looks great, except $[\![\texttt{len}]\!]$ is defined in terms of itself. There is no way for us to replace $[\![\texttt{len}]\!]$ if it's a self definition: remember, no functions in λ-calculus have a name, so we can't simply have $[\![\texttt{len}]\!]$ refer to the name $[\![\texttt{len}]\!]$, as that's actually just a shorthand for the expansion of $[\![\texttt{len}]\!]$. Hence, this definition is invalid. So what should we do now? We can try to solve the equation. We can "factor out" the $[\![\texttt{len}]\!]$ on the right hand side by β-expansion:

$$\begin{aligned}[\![\texttt{len}]\!] &= \lambda l.\,(\,[\![\texttt{null?}]\!]\,l)\,[\![0]\!]\,(\,[\![\texttt{succ}]\!]\,(\,[\![\texttt{len}]\!]\,(\,[\![\texttt{tail}]\!]\,l))) \\ &\leftarrow_\beta (\lambda f.\,\lambda l.\,(\,[\![\texttt{null?}]\!]\,l)\,[\![0]\!]\,(\,[\![\texttt{succ}]\!]\,(f(\,[\![\texttt{tail}]\!]\,l))))\;[\![\texttt{len}]\!]\end{aligned}$$

Now, we have the equation $[\![\texttt{len}]\!] = (\lambda f.\,\lambda l.\,(\,[\![\texttt{null?}]\!]\,l)\,[\![0]\!]\,(\,[\![\texttt{succ}]\!]\,(f(\,[\![\texttt{tail}]\!]\,l))))\,[\![\texttt{len}]\!]$. If we can solve this for $[\![\texttt{len}]\!]$, then we'll have a definition of $[\![\texttt{len}]\!]$ that does not include $[\![\texttt{len}]\!]$. Let

$$F = \lambda f.\,\lambda l.\,(\,[\![\texttt{null?}]\!]\,l)\,[\![0]\!]\,(\,[\![\texttt{succ}]\!]\,(f(\,[\![\texttt{tail}]\!]\,l)))$$

So our equation just becomes $[\![\texttt{len}]\!] = F\,[\![\texttt{len}]\!]$. To solve the equation, we will use the theory of *fixed points*.

In mathematics, a *fixed point* of a function is a value that, when passed to the function, yields itself. In other words, the fixed points of a function f are the values of x such that $f(x) = x$. For example, for $f(x) = x^2 - 6$, $x = 3$ is a fixed point.

Consider the following expression:

$$\begin{aligned} X = &\quad (\lambda x.\, f(xx))(\lambda x.\, f(xx)) \\ &\to_\beta f((\lambda x.\, f(xx))(\lambda x.\, f(xx))) \\ &= \quad fX \end{aligned}$$

Since $X = fX$, X is a fixed point of f, whatever f is. In this case, f is just a place holder for the function whose fixed point we wish to find. By setting f to the desired expression, we can find a fixed point for *any* λ-expression! We can construct a general fixed-point combinator[6] by modifying X to accept an argument for specifying f. What we get is the *Curry's Paradoxical Combinator* (or simply the *Y combinator*):

$$Y := \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

The critical property of the Y combinator is the following:

$$\begin{aligned} Yg = &\quad (\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)))g \\ &\to_\beta (\lambda x.\, g(xx))(\lambda x.\, g(xx)) \\ &\to_\beta g((\lambda x.\, g(xx))(\lambda x.\, g(xx))) \\ &\leftarrow_\beta g(Yg) \end{aligned}$$

So for any g, we have $Yg =_\beta g(Yg)$. Note that this sequence can be repeated to give $g(g(Yg)), g(g(g(Yg)))$ and so on. This sequence of repeated applications of g does appear to capture the general character of recursive computation. Y is in fact one of the broader class of combinators, defined below:

Definition 11. A *fixed-point combinator* is any combinator C such that

$$Cf =_\beta f(Cf)$$

for every f.

Let's go back to the list length example. Remember that we had arrived at $[\![\texttt{len}]\!] = F\,[\![\texttt{len}]\!]$, by factoring out $[\![\texttt{len}]\!]$. We observe that solving our defining equation for `len` is equivalent to finding a fixed point for the function F. According to our observation, the fixed point is just YF. Hence the following definition:

$$[\![\texttt{len}]\!] := YF = Y(\lambda f.\, \lambda l.\, (\, [\![\texttt{null?}]\!]\, l)\, [\![0]\!]\, (\, [\![\texttt{succ}]\!]\, (f(\, [\![\texttt{tail}]\!]\, l))))$$

Let us step through an example. This example is quite long, so bear with us:

$$\begin{aligned} &\quad [\![\texttt{len}]\!]\, (\, [\![\texttt{cons}]\!]\, a(\, [\![\texttt{cons}]\!]\, b\, [\![\texttt{nil}]\!]\,)) \\ &=_\beta [\![\texttt{len}]\!]\, (\lambda s.\, sa\lambda s.\, sb\, [\![\texttt{nil}]\!]\,) \\ &= YF(\lambda s.\, sa\lambda s.\, sb\, [\![\texttt{nil}]\!]\,) \\ &= (\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)))F(\lambda s.\, sa\lambda s.\, sb\, [\![\texttt{nil}]\!]\,) \qquad (1) \\ &\ldots \end{aligned}$$

Let's pause this example for a bit here and think about a very important question: should we use AOR or NOR as our reduction strategy in this case? Well, the AOR redex here is $(\lambda x.\, f(xx))(\lambda x.\, f(xx))$, which does not have

[6] Recall that a combinator is just a λ-term with no free variables.

a normal form; continuing to reduce it results in something that looks like $f(f(f(\cdots)))$. AOR always chooses the leftmost innermost redex, which means we'll fall into this infinite reduction trap while reducing in the step labeled (1). So, we have to use NOR here.

Let's go back to our reduction now:

$$
\begin{aligned}
&\cdots \\
&= \quad (\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)))F(\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ) \\
&\to_\beta\ (\lambda x.\,F(xx))(\lambda x.\,F(xx))(\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ) \\
&\to_\beta\ F((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ) \qquad (2) \\
&= \quad (\lambda f.\,\lambda l.\,(\ [\![\texttt{null?}]\!]\ l)\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (f(\ [\![\texttt{tail}]\!]\ l))))((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ) \\
&\to_\beta\ (\lambda l.\,(\ [\![\texttt{null?}]\!]\ l)\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ l))))(\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ) \\
&\to_\beta\ (\ [\![\texttt{null?}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ )))) \\
&\to_\beta^*\ [\![\texttt{false}]\!]\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ )))) \\
&\to_\beta^2\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))) \\
&\cdots
\end{aligned}
$$

So far, we have completed one recursive "call". Under NOR, we would substitute the whole expression after $[\![\texttt{succ}]\!]$ into the function `succ`, and this would lead us to the correct result. However, in order to illustrate the next recursive reduction, we will choose $(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))$ and $((\lambda x.\,F(xx))(\lambda x.\,F(xx)))$ as our next redices to reduce instead:

$$
\begin{aligned}
&\cdots \\
&\to_\beta^2\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sa\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))) \\
&\to_\beta^*\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\lambda s.\,sb\ [\![\texttt{nil}]\!]\ )) \\
&\to_\beta\ [\![\texttt{succ}]\!]\ (F((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\lambda s.\,sb\ [\![\texttt{nil}]\!]\ )) \\
&= \quad [\![\texttt{succ}]\!]\ ((\lambda f.\,\lambda l.\,(\ [\![\texttt{null?}]\!]\ l)\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (f(\ [\![\texttt{tail}]\!]\ l))))((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\lambda s.\,sb\ [\![\texttt{nil}]\!]\ )) \\
&\to_\beta^2\ [\![\texttt{succ}]\!]\ ((\ [\![\texttt{null?}]\!]\ (\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ (\lambda s.\,sb\ [\![\texttt{nil}]\!]\ ))))) \\
&\to_\beta^*\ [\![\texttt{succ}]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))\ [\![\texttt{nil}]\!]\ )) \\
&\to_\beta^*\ [\![\texttt{succ}]\!]\ (\ [\![\texttt{succ}]\!]\ ((\ [\![\texttt{null?}]\!]\ [\![\texttt{nil}]\!]\ )\ [\![0]\!]\ (\ [\![\texttt{succ}]\!]\ (((\lambda x.\,F(xx))(\lambda x.\,F(xx)))(\ [\![\texttt{tail}]\!]\ [\![\texttt{nil}]\!]\ )))) \qquad (3) \\
&\to_\beta^*\ [\![\texttt{succ}]\!]\ (\ [\![\texttt{succ}]\!]\ (\ [\![0]\!]\ )) \\
&\to_\beta^*\ [\![2]\!]
\end{aligned}
$$

We have now seen that the Y combinator indeed works under NOR. However, we sometimes do want AOR in λ-calculus, since it more closely resembles how many real-world programming languages are evaluated, so we need to consider how to make recursion work with something like AOR. Here are a few observations we can make from the above example, and how we may need to change AOR to compensate:

1. As step labeled (1) has shown, the fixed-point combinator must be the leftmost expression at some point in the reduction. AOR will always try to reduce the innermost redex in the leftmost expression first, resulting in infinite reduction. To still have the "eagerness" in our reduction, we need to modify AOR a bit. The new reduction strategy is called *Applicative Order Evaluation* (AOE), and it will choose the redex that is **leftmost, innermost, and not inside the body of an abstraction** as the one to reduce first. This can still reach WNF, since WNF does not require reduction of redices inside the body of abstractions, which is also true of real programming languages.
2. Even with AOE, the step labeled (2) will still have an infinite reduction, since $(\lambda x.\,F(xx))(\lambda x.\,F(xx))$ part in (2) will be reduced to $F(F(F(\cdots(F((\lambda x.\,F(xx))(\lambda x.\,F(xx))))\cdots))$ under AOE. Therefore, we need modify our fixed-point combinator to wrap the "repetition" part, (xx), in an abstraction[7] to prevent AOE from

[7] More precisely, perform η-expansion. Note that $Y' \to_\eta^2 Y$.

choosing it as next redex to reduce:

$$Y' = \lambda f.\,(\lambda x.\,f(\lambda y.\,xxy))(\lambda x.\,f(\lambda y.\,xxy))$$

3. We have mentioned that our $[\![\texttt{if}]\!]$ does not perform short-circuit evaluation, and it shouldn't be a problem as long as both branches don't have infinitely reducing expression. But, whoops, we do have an infinitely reducing expression in the `if` expression within the step labeled (3), and that branch is the one we don't intend to take! As a result, we do need to have a short-circuited version of $[\![\texttt{if}]\!]$. As you may have guessed, the same trick works again. Let's wrap both branches in their respective abstractions:

$$[\![\texttt{if } B \texttt{ then } T \texttt{ else } F]\!] := [\![B]\!]\,(\lambda x.\,[\![T]\!]\,)(\lambda x.\,[\![F]\!]\,)x$$

The short-circuiting works, as $[\![B]\!]\,(\lambda x.\,[\![T]\!]\,)(\lambda x.\,[\![F]\!]\,)$ reduces to either $(\lambda x.\,[\![T]\!]\,)$ or $(\lambda x.\,[\![F]\!]\,)$ before x is passed into either of those two functions under AOE.

With these three modifications, the reduction will work under a modified form of AOR, AOE.

8 deBruijn Notation

As our search for a proper definition of substitution has taught us, the names of variables can often get in the way of the true meaning of an expression. Also, as we have seen many times, a name can have different meanings. In the expression $\lambda y.\,a(\lambda y.\,b(\lambda y.\,cy)y)y$, each y does not have the same meaning, because there are three different bindings of y. Of course, we can α-convert all of the repeated occurrences of y to prevent accidental capture of variables. Here, we present an alternative solution to this problem presented by deBruijn in 1972 [5], as part as his implementation of Automath [4], a tool for automatically verifying mathematical proofs.

In the deBruijn notation, variables are replaced with integers. Thus the expression $\lambda x.\,\lambda y.\,x$ in the deBruijn notation is $\lambda.\,\lambda.\,2$. The integers indicate the number of function bodies that must be escaped to locate the binding for the variable. Note that a single binding can therefore be represented by several different integers, depending on how many further function bodies the reference is embedded in.

Example 21. The expression $\lambda x.\,(\lambda y.\,x)x$ has the deBruijn equivalent $\lambda.\,((\lambda.\,2)1)$, in which x is replaced by both 2 and 1.

Free variables can be handled in a number of ways, of which the simplest is to leave them unconverted. Importantly, in the deBruijn notation, unlike in our conventional notation, all α-equivalent expressions have exactly one representation.

The definition of β-reduction for the deBruijn notation is

$$(\lambda.\,N)M \to_\beta N[M/1]$$

We also need a new definition for substitution:

$$n[N/m] = \begin{cases} n & \text{if } n < m \\ n-1 & \textit{if } n > m \\ \text{rename}_{n,1}(N) & \textit{if } n = m \end{cases}$$
$$(M_1M_2)[N/m] = M_1[N/m]M_2[N/m]$$
$$(\lambda.\,M)[N/m] = \lambda.\,M[N/m+1]$$

where

$$\text{rename}_{m,i}(j) = \begin{cases} j & \text{if } j < i \\ j+m-1 & \text{if } j \geqslant i \end{cases}$$
$$\text{rename}_{m,i}(N_1N_2) = \text{rename}_{m,i}(N_1)\,\text{rename}_{m,i}(N_2)$$
$$\text{rename}_{m,i}(\lambda.\,N) = \lambda.\,\text{rename}_{m,i+1}(N)$$

The deBruijn notation is useful for avoiding the cost of α-conversion. Substitution and β-reduction in deBruijn notation are essentially operations on integers, which is often faster than operations on names. Furthermore, deBruijn notation exposes the concept of a "stack frame": the integers can be regarded as indices to a stack of values mapping such indices to the expression those bound variables refer to after a reduction.

However, deBruijn notation is considerably harder to read and understand, and often a program evaluating λ-calculus in deBruijn notation needs to provide facilities to convert one notation to another. Such conversions can be costly, offsetting the performance benefit deBruijn notation itself provides.

9 Implementation Strategies

Many programming languages are based on λ-calculus to some degree, so there is a long history of implementing λ-calculus concepts. Generally speaking, a λ-calculus expression itself can be stored as an abstract syntax tree, in which each node is labeled with its type (variables, abstractions, applications). For instance, the λ-calculus expression $\lambda b.\, \lambda t.\, \lambda f.\, btf$ (i.e., $[\![\texttt{if}]\!]$) could be represented as the following tree:

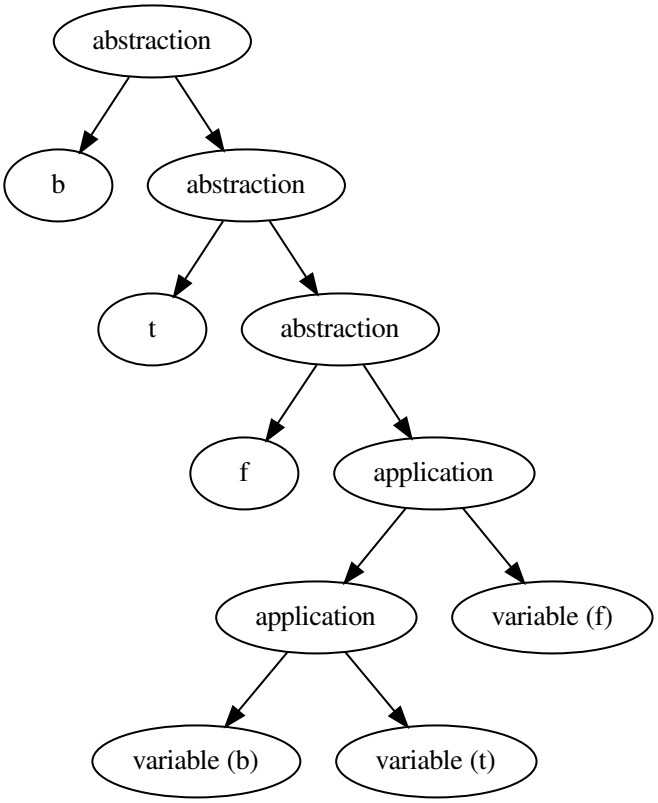
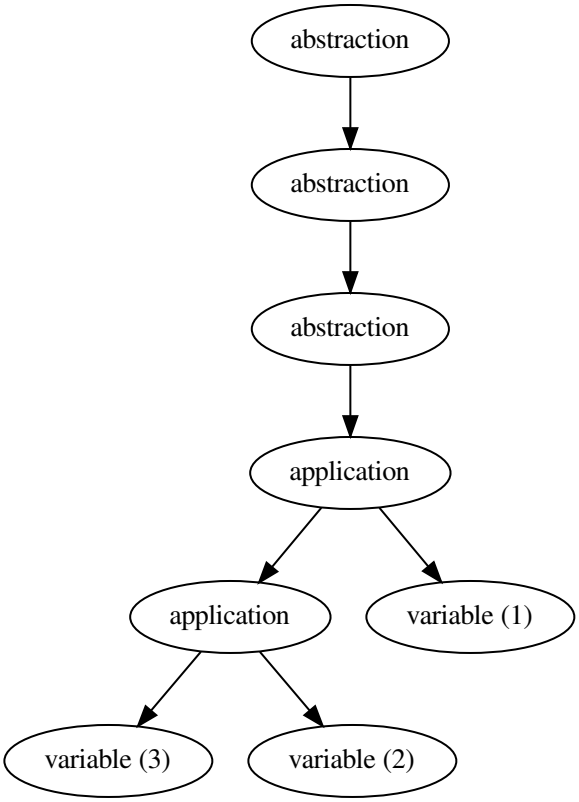


Figure 3: AST for $[\![\texttt{if}]\!]$

To evaluate it, you simply walk the tree, following the evaluation strategy of your choice (i.e., choose whether to enter an abstraction first), and replacing applications with their substitutions. To do this, you need to carry a list of bindings, so you know what to substitute. This can get complicated, since a subtree may reuse a variable name, and further complicated because you need to be able to generate a fresh name in order to avoid conflicts.

With deBruijn notation, you instead carry a *stack* of bindings, and a variable is simply an index into that stack. This makes evaluation simpler, but requires an extra step, and makes debugging more difficult. Modified to use deBruijn notation, our tree would instead look like this:



Of course, with deBruijn notation, it's also easier to verify that a λ-calculus interpreter is correct, since you don't need a final α-equivalence check on the final output.

10 Fin

In the next module, we will describe the behavior of λ-calculus expressions with formal rigor, by introducing *formal semantics*.

References

[1] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, NI Adams, Daniel P. Friedman, E Kohlbecker, GL Steele, David H Bartley, Robert Halstead, et al. Revised 5 report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[2] Alonzo Church and J Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[3] Haskell B. Curry and R. Feys. *Combinatory Logic*. Number Vol. I in Combinatory Logic. North-Holland Publishing Company, 1968.

[4] Nicolaas Govert De Bruijn. The mathematical language automath, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970.

[5] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

Rights