Introduction to the Lambda Calculus, Part 2

In our second reading on the lambda calculus, we finish up discussing syntax and move on to the semantics of the language.

Ambiguity

In part 1, we looked at an expression `λx.xx` and asked what its *derivation tree* should look like. Should it look like the tree in Figure 15 or the tree in Figure 16?

I told you that, because *abstraction is right-associative*, the correct interpretation is the tree shown in Figure 15.

But what if you wanted to encode the second tree as a lambda program? Unless you write your program as separate pieces, such as

```
a is λx.x
```

and

```
b is x
```

such that the whole program is

```
ab
```

then there does not appear to be a way to encode the second tree using the syntax we are given. Adding parentheses to the language solves this problem.

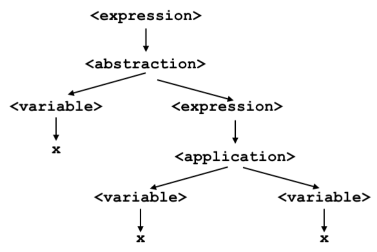


Figure 15: Correct derivation of the expression `λx.xx`.

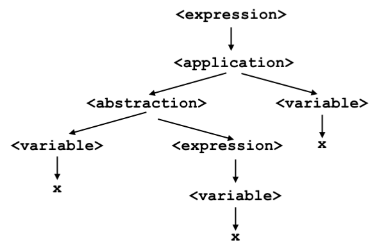


Figure 16: Incorrect derivation of the expression `λx.xx`.

Parentheses

Parentheses remove ambiguity. Let's start by introducing a simple axiom into our system:

$$[\![\texttt{(<expression>)}]\!] \equiv \texttt{<expression>}$$

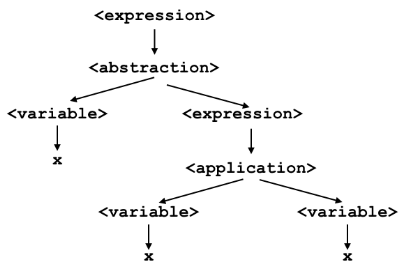
This axiom says that "the meaning of" (the part inside the $[\![\ ]\!]$ symbols) any expression enclosed in parentheses "is identical to" ($\equiv$) that same expression without parentheses. With this rule, you can feel free to surround an expression with parens. When an expression would not be made ambiguous by doing so, you can also remove them.

Let's augment our grammar so that we don't run into any more ambiguity traps.

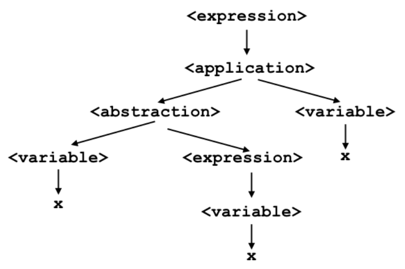
```
<expression>  ::= <variable>
                |  <abstraction>
                |  <application>
                |  <parens>


<variable>    ::= x
<abstraction> ::= λ<variable>.<expression>
<application> ::= <expression><expression>
<parens>      ::= (<expression>)
```

Now we can avoid the problem with `λx.xx`. If we write `λx.xx`, the interpretation, because abstraction is right associative, is:



and if we write `(λx.x)x` then the interpretation is



It is conventional, among people who use the lambda calculus, to drop parenthesis whenever an interpretation is unambiguous. Googling for lambda expressions will often turn up scads of examples that omit parens. When you are aware of this fact, many of these examples will be easier to interpret.

Other extensions

Additional variables—in other words, not limiting ourselves to just the variable `x`—makes the grammar easier to work with. Here is our grammar augmented with additional variables:

```
<expression>  ::= <variable>
               |  <abstraction>
               |  <application>
               |  <parens>

<variable>    ::= α ∈ { a ... z }
<abstraction> ::= λ<variable>.<expression>
<application> ::= <expression><expression>
<parens>      ::= (<expression>)
```

where, hopefully, it is clear that $\alpha \in$ `{ a ... z }` denotes *any letter*. E.g., any letter could be `x`, or `y`, or `b`, or `g`, etc.

Finally, to make the lambda calculus more immediately useful, let's add one more rule, `<arithmetic>`. Note that this rule is not strictly required, because although it may not be apparent to you, arithmetic can be encoded using all of the pieces we've already discussed. But learning those encodings can be difficult and they are not required to understand the lambda calculus, so we will put them off until later.

An example of arithmetic of the kind we're talking about might be an expression like `1 + 1`. The expression `1 + 1` is in *infix form*, because the operator (`+`) is in between the operands (`1` and `1`). You are accustomed to infix notation because of years of practice, but from a computational standpoint, it is a little bit of a hassle to work with. Instead, we will write all arithmetic in *prefix form*, which is easier to manipulate programmatically. In prefix form,

- The entire expression is enclosed within parentheses.
- The operator is the first term written inside the parentheses.
- All subsequent terms written after the operator, but still within the parentheses, are operands.

Formally,

```
<arithmetic> ::= (<op> <expression> ... <expression>)
```

For example, `1 + 1` is written as `(+ 1 1)`. Another example is `c ÷ z`, which is written `(÷ c z)`.

What is `<op>`? A simple formulation might include just `+`, `-`, $\times$, and $\div$. I will stick to `+` and `-` in this class.

One nice thing about prefix form, which is why we're using it here, is that the order of operations is very clear. For example, with a little practice, you should have no difficulty computing `(+ 3 (* 5 4))`. Recall that the equivalent expression in "normal" arithmetic might be written as `3 + 5 * 4`, which is ambiguous if you don't remember your precedence and associativity rules from algebra class.

Here's an updated grammar.

```
<expression>   ::= <value>
                |  <abstraction>
                |  <application>
                |  <parens>
                |  <arithmetic>

<value>        ::= v ∈ ℕ
                |  <variable>
<variable>     ::= α ∈ { a … z }
<abstraction>  ::= λ<variable>.<expression>
<application>  ::= <expression><expression>
<parens>       ::= (<expression>)
<arithmetic>   ::= (<op> <expression> ... <expression>)
<op>           ::= o ∈ { +, - }
```

Finally, notice that I added one more component, called `<value>`. What is `<value>`? In this language, `<value>` is either a number or an `<variable>`.

You can see why I don't introduce these complications all up front: the grammar is starting to look a little hairy. Still, if you remember that there are essentially three parts to the lambda calculus, *variables*, *abstractions*, and *applications*, you will be fine. To recap, we added:

1. Parentheses to remove ambiguity.
2. Extra variables.
3. Simple arithmetic in prefix form.

Abstract syntax

One of the tasks newcomers to the lambda calculus most struggle with is identifying parts of an expression. Before we talk about what the lambda calculus means, let's expand on our parsing skills. Given a grammar and and expression, by now you can probably give me a derivation tree for that expression, or you can tell me that the expression is not a valid sentence in the language defined by the grammar. However, although derivation trees are useful, they are also cumbersome. They focus more on *how an expression is derived* and less on *what a sentence means*. Although the two are clearly related in some ways, sometimes we really just want to understand an expression's meaning.

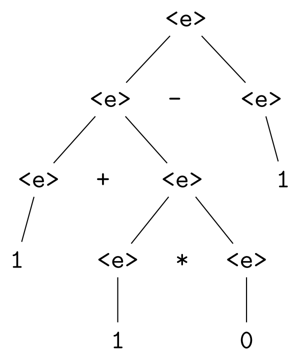
When we want to understand the meaning of a sentence, we usually turn to an alternative kind of parse tree called an *abstract syntax tree* or AST. Abstract syntax makes the meaning of an expression quite clear. The term *abstract* means that we no longer care about all the details of a language's syntax; instead, we focus on the most important content.

Typically, the content we care about is *data* and *operations*. In an AST, interior nodes are operations and leaf nodes are data.

As an example, let's revisit the simple calculator grammar shown in one of our previous readings.

```
<e> ::= 0 | 1 | (<e> + <e>) | (<e> - <e>) | (<e> * <e>)
```

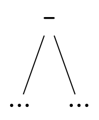
Suppose we have the expression `((1 + (1 * 0)) - 1)`. It has the following derivation:



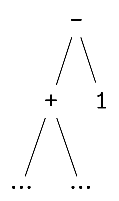
Instead of filling our tree with `<e>`, let's define our tree purely in terms of our three operations, `+`, `-`, and `*`, and our two number literals, `0` and `1`. If you need a little more precision than this, imagine we're using the following `type` definition from ML.

```
type Expr =
  | Zero
  | One
  | Addition of Expr * Expr
  | Subtraction of Expr * Expr
  | Multiplication of Expr * Expr
```

The top level operation in our derivation is subtraction, so our new tree will start like this:



The expression on the left of the subtraction is addition, and the expression on the right of the subtraction is `1`.



The expression on the left of the addition is `1`, and the expression on the right of the addition is multiplication.

```
        -
       / \
      +   1
     / \
    1   *
       / \
     ...  ...
```

The expression on the left of the multiplication is `1`, and the expression on the right of the multiplication is `0`.

```
        -
       / \
      +   1
     / \
    1   *
       / \
      1   0
```

Observe that the meaning of this expression is much clearer. As noted before, all of the data is at the leaves and all of the operations are in the interior.

A big reason why we like ASTs is that they suggest a natural procedure for evaluating an expression: any operator whose operands are data can be evaluated. See if you can evaluate the tree above from the "bottom up." [14] Since you already know how to perform arithmetic, I hope you'll see that ASTs provide additional clarity about what to do with an expression. As you shall see, we will rely on ASTs to aid our understanding of the meaning of lambda calculus expressions.

[14] Hint: the only operation whose operands are data is `*`. After evaluating `*`, redraw the tree, replacing the `*` subtree with the result.

Semantics of the Lambda Calculus

Semantics is the study of *meaning*. In the context of programming languages, what we usually care about is how an expression can be converted into a sequence of mechanical steps that can be performed by a machine.

So how do we interpret the meaning of a lambda expression?

Unfortunately, the meaning of the word "interpret" is unclear. There are two meanings for the word "interpret":

1. To understand.

2. To evaluate.

In programming languages, and more generally in computer science, we tend to favor the latter definition of the word "interpret," i.e., to evaluate. Why? Because evaluation is a process that we can carry out on a machine.[15] In a well-designed programming language, there is no room for ambiguity. Precision is what sets programming languages apart from natural languages. It is why we can precisely describe programs and run them billions of times without error. But it also means, because they are so different from natural languages, that they are usually harder to learn.

[15] Maybe someday our study of artificial intelligence will allow computers to "understand" things, but we're not there yet. Part of the difficulty may be that we don't really know what "understanding" itself means.

Evaluation via term rewriting

The lambda calculus is an example of a *term rewriting system*. You've seen a term rewriting system before. The system of mathematics you learned in high school—algebra—is a term rewriting system. The big idea behind term rewriting systems is that, by following simple substitution rules, you can "solve" the system. In algebra, we mix term rewriting (substitution) and arithmetic (e.g., `1 + 2`) to solve for the value of a variable.

The amazing and surprising fact about the lambda calculus is that term rewriting is sufficient to *compute anything*. Yes, really, all you need to do is follow a few simple rewriting rules, just like in algebra, and you don't even need the arithmetical part.

That said, if you intend to compute "interesting" things—the kinds of programs we normally write—the language is inconvenient to work with. Although it is not as primitive as a Turing machine, it's pretty darned primitive. Alonzo Church's contribution was to show that the functions we most care about—the ones that are computable in principle—can all be written as lambda expressions. In this class, I prefer that you not have to work at such a primitive level, and so our version of the lambda calculus has "first class" support for arithmetic (`<arithmetic>`). Without it, writing useful programs requires a lot of work, but this one simple extension makes the language useful without sacrificing much of its simplicity.

There are essentially two rewriting rules in the lambda calculus. To understand them, I need to introduce few things first: equivalence and bound/free variables.

Equivalence

In the lambda calculus, we say that two expressions are *equivalent* when they are *lexically identical*. In other words, when they have exactly the same string of characters. For example, the expressions `λx.x` and `λy.y` mean the same thing, but they are not equivalent because they *literally* have different letters in them. `λx.x` and `λx.x` *are* equivalent, however, because they are exactly the same.

We often talk a little informally about equivalence and you might hear someone say that `λx.x` and `λy.y` "are equivalent." What they mean is that, after rewriting, the two expressions can be made equivalent.

Bound and free variables

Depending on where a variable appears in a lambda expression, it is either *bound* or *free*.

What is a bound variable? A bound variable is a variable named in a lambda abstraction. For example, in the expression

```
λx.<expression>
```

where `<expression>` denotes some expression (that I don't care about at the moment), and where `x` is the bound variable. How do we know that `x` is bound? Well, that's precisely what a lambda abstraction *means*. You can read the expression `λx.<expression>` literally as saying "the variable `x` is bound in expression `<expression>`."

`x` is a variable, and when `x` is used inside of `<expression>`, its value takes on whatever value is given when the lambda abstraction is evaluated. Lambda abstractions are the precise mathematical meaning of what we're talking about when we *define a function* in a conventional programming language. To give you an analogy in a language with which you might be more familiar (Python), you can informally think of the above expression to mean almost exactly the same thing as the following program

```
def foo(x):
    <expression>
```

except that in the lambda calculus, functions don't have names (i.e., no `foo`).

Now it should make sense to you when I say that `x` is a bound variable. When we call `foo`, for example, `foo(3)`, we know that wherever we see `x` in the function body, we should substitute in the value `3`. Let's say we have the Python program:

```
plus_one(x):
    return x + 1
```

Then the expression

```
plus_one(3)
```

means

```
3 + 1
```

Until we actually *call* the function, `x` could pretty much be anything; it's just a parameter. Its value is tied to the calling of the function.

Back to the lambda calculus. In the following expression,

```
λx.x
```

all instances of `x` refer to the same value. How do we know? Because the lambda abstraction tells us that the value of any `x` within its scope (remember: abstraction is right-associative) is *bound* to the value of the parameter `x`.

What about the following expression?

```
λx.y
```

The lambda abstraction tells us that `x` is bound but it says nothing about `y`. In fact, we can't really make any assumptions about `y`. Therefore, we call `y` a *free* variable.

To understand a lambda expression, you must always determine whether every variable is bound or free. Be careful! Consider the following expression:

```
λx.λy.xy
```

Both `x` and `y` are bound. Why? Let's rewrite using parens. Let's start with the rightmost lambda. We know, because of right-associativity, that everything to the right of the last period must belong to the rightmost lambda. So,

```
λx.λy.(xy)
```

We also know that everything to the right of the leftmost period must belong to the first lambda.

```
λx.(λy.(xy))
```

Is the rightmost `y` bound or free? It is bound because it is within the scope of the `λy.___` abstraction. What about the rightmost `x`? It is *also* bound, because it is within the scope of the `λx.___` abstraction. If you don't believe me, just look at the outermost parens:

```
λx.(... x ...)
```

Here's a more complicated example with a free variable:

```
(λx.x)y
```

Can you spot which one is free? [16]

One more:

[16] It's `y`.

```
λx.λx.xx
```

Which variables are bound?

In this case, all the xes you see are bound. However, there are *two* x variables. Let's put in some parens to make the expression easier to understand.

```
λx.(λx.(xx))
```

So both x values are within the scope of both lambda abstractions. To which abstraction is x bound? In the lambda calculus, the *last abstraction wins*. Therefore, it is *as if* λx.(λx.(xx)) were written

```
λy.(λx.(xx))
```

and to give you an intuitive sense of this, this is more or less equivalent to the following (perfectly valid but slightly unusual) Python program,

```
def yfunc(y):
    def xfunc(x):
        return x(x)
    return xfunc
```

but, of course, without the function names.

Reductions

The rules used to rewrite expressions in the lambda calculus are called *reduction rules*. Reductions are the heart of what it means *to evaluate*, or more colloquially, "to execute," a lambda expression.

α Reduction

The first rewriting rule is called *alpha reduction*. Alpha reduction is a rule introduced to deal with ambiguity surrounding the use of variables. Specifically, alpha reduction relies on the property that, in the lambda calculus, *the given name* of a bound variable largely does not matter.

Let's look at a concrete example. Remember the identity function?

```
def identity(x):
    return x
```

It returns whatever we give it. Here's a Python interpreter session:

```
$ python
Python 2.7.15 (default, Aug 22 2018, 16:36:18)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def identity(x):
...     return x
...
>>> identity(3)
3
```

Hopefully no surprises there, right? What if, after we explored the above definition of `identity`, I asked you what the following function meant:

```
def identity2(z):
    return z
```

I suspect that you might be a little annoyed. Why? Because obviously `identity1` and `identity2` are the same function.

Remember, though, that in the lambda calculus, we have a strict notion of equivalence. Two expressions are equivalent if and only if they are exactly the same string. So even though `identity` in the lambda calculus is:

```
λx.x
```

and `identity2` in the lambda calculus is:

```
λy.y
```

and we can sort of squint and see that they're the same, that's not enough. We have to *prove* it.

α Equivalence

The notation `[a/b]<expression>` means "replace the variable `b` with variable `a` in `<expression>`." To perform alpha reduction, we rely on the following property of the lambda calculus, which is called "alpha equivalence":

$$[\![\lambda x_1.e]\!] =_\alpha [\![\lambda x_2.[x_2/x_1]e]\!]$$

where x_1 and x_2 are variables, and e is an expression.

This property says that the meaning of the expression of the form $\lambda x_1.e$ is the same when you replace it with an expression of the form $\lambda x_2.[x_2/x_1]e$. The two expressions are "alpha equivalent" (that's what $=_\alpha$ means). Since $[x_2/x_1]$ is not a valid lambda-calculus expression, you must continue replacement of x_1 with x_2 wherever you find it in *e*. You continue doing this until you can proceed no further with substitution.

Here's a proof that the two expressions are the same.

`λx.x`	given
`[y/x] λx.x`	alpha reduce x with y
`λy.[y/x]x`	step 1: rename outer x and continue to inner expression
`λy.y`	step 2: rename inner x

Therefore, `λx.x` $=_\alpha$ `λy.y`.

I mentioned before that sometimes you can "proceed no further with substitution". Substitution can be "blocked" by nested lambdas. Recall the expression from before:

```
λx.λx.xx
```

You might be wondering: is this expression equivalent to `λy.λx.yy`? This is a perfect time to do alpha reduction. Let's replace `x` with `y`.

`λx.λx.xx`	given
`[y/x] λx.λx.xx`	alpha reduce x with y
`λy.[y/x] λx.xx`	step 1: rename outer x and continue to inner expression
`λy.λx.xx`	done (substitution blocked by λx)

We cannot rename the `x` inside the inner expression because it is a *different* `x` than the outer `x`. The inner lambda "redefines" `x`. Therefore `λx.λx.xx` is *not* equivalent to `λy.λx.yy`, at least not using alpha reduction.

When two expressions can be made equivalent using alpha reduction, we call them *alpha equivalent*.

β Reduction

There is one other kind of reduction called *beta reduction*. Beta reduction is the beating heart of the lambda calculus because it is, essentially, what it means to *call a function*. We refer to calling a lambda function *application*.

The simplest possible example uses the identity function:

```
λx.x
```

Let's "call" this function with a value. Say, `y`.

```
(λx.x)y
```

Recall that we use parentheses to make this expression unambiguous.

At a high level. This expression has two parts: *left* and *right*. The left side is `λx.x`. The right side is `y`. How do we know? The parens tell us.

What is the *one* grammar rule in the lambda calculus that lets us interpret an expression with a left part and a right part? It's this rule:

```
<application> ::= <expression><expression>
```

Colloquially, we call the left part the *function* and the right part the *argument*. Why? Because those two parts work just like the function and argument parts you might see in a conventional programming language.

```
(λx.x)y
```

works just like

```
def identity(x):
    return x

identity(y)
```

because we expect to get `y` back when we call the `identity` function with `y` as an argument[17].

[17] The exact Python equivalent to `(λx.x)y` is actually `(lambda x: x)(y)`. Python will complain that `y` is not defined if you do not define it somewhere; the lambda calculus is less strict because it doesn't really care if `y` is free. Python is an *eager* language, which essentially means that variables can never be free.

β Equivalence

Beta reduction is a substitution rule that achieves the same effect as calling a function. We again use the substitution operation, `[a/b] c` which means "substitute variable `b` with expression `a` in the expression `c`", but in the case of beta reduction, substitution *eliminates* both the function (the lambda abstraction) and its argument.

We rely on the following property, which is called "beta equivalence":

$$[\![(\lambda x_1.e)x_2]\!] =_\beta [\![[x_2/x_1]e]\!]$$

where x_1 is a variable, and x_2 and e are expressions.

This property says that an expression of the form $(\lambda x_1.e)x_2$ has the same meaning as an expression of the form $[x_2/x_1]e$. The two expressions are "beta equivalent" (that's what $=_\beta$ means). Since $[x_2/x_1]$ means "substitute x_2 for x_1 in e" and is not a valid lambda calculus expression, you must continue replacement until you can proceed no further. As with alpha reduction, redefinition of a variable inside a lambda "blocks" substitution.

Example:

`(λx.x)y`	given
`([y/x] x)`	β-reduce x with y; step 1: eliminate abstraction and argument
`(y)`	step 2: replace x with y
`y`	eliminate parens (because `<expression> = (<expression>)`)

Let's look at another example.

`(λx.xx)z`	given
`([z/x] xx)`	β-reduce x with z; step 1: eliminate abstraction and argument
`(zz)`	step 2: replace x with z
`zz`	eliminate parens

In the fourth step, we beta reduce inside the application `xx` because `[z/x]<expression><expression>` means `([z/x]<expression>)([z/x]<expression>)`.

Reduction order

In the lambda calculus, the order of reductions does not matter.[18] You are already familiar with this idea. Suppose I ask you to evaluate the polynomial $2x^2 + y/3$, where $x = 1$ and $y = 3$. Does it matter which variable you substitute first? Clearly the answer is no. We could first substitute x to obtain $2 + y/3$ and then y, yielding $2 + 1$. Or we could substitute y to obtain $2x^2 + 1$ and then x, also yielding $2 + 1$. The result is the same. A term rewriting system whose substitution order does not matter is *confluent*.

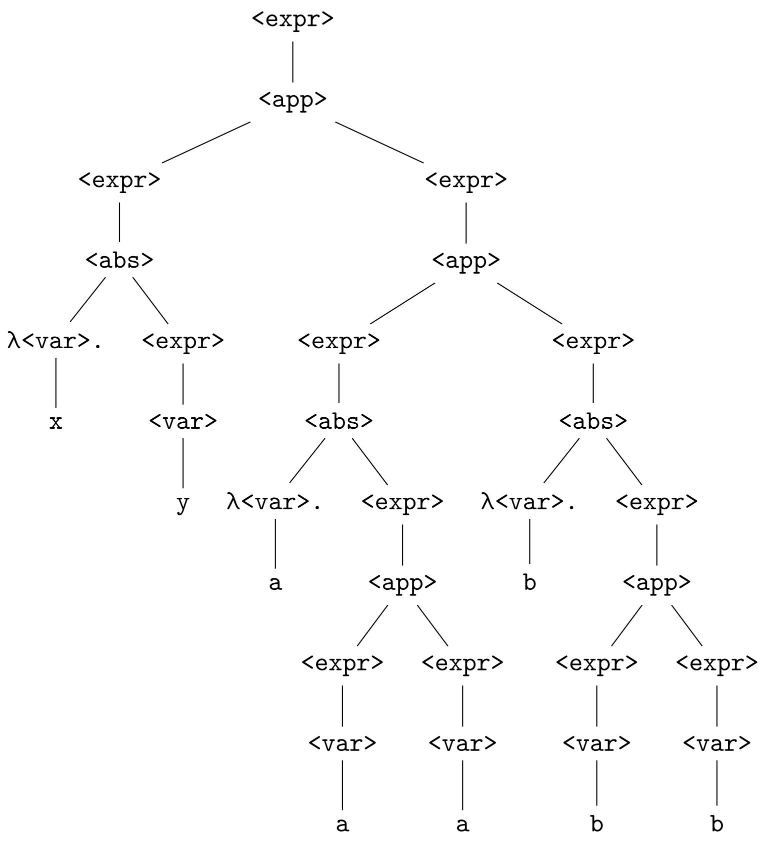
[18] This idea is called the Church-Rosser Theorem.

Likewise, the order of reductions does not matter in the lambda calculus. The lambda calculus is confluent. Let's look at an example of an expression where there is a choice about which reduction we can apply.

```
(λx.y)((λa.aa)(λb.bb))
```

It's probably hard for you to see where reductions can be applied in the above expression. Things are greatly clarified by drawing a lambda expression's abstract syntax tree. Let's start by producing a derivation tree, then converting it into an abstract syntax tree like we did before with our arithmetic expression.[19]

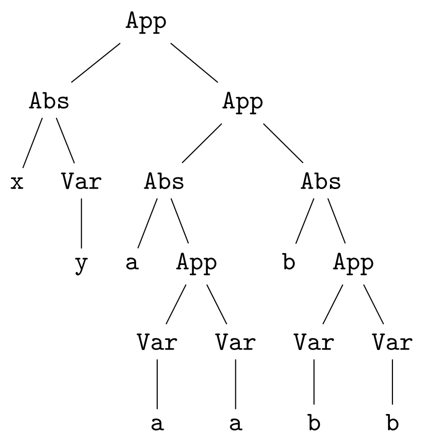
[19] You will eventually be able to produce ASTs directly from an expression without first having to draw a derivation tree.



That's a lot of drawing! Hopefully the AST is simpler and clearer. As before, we need to define our AST's operations and data. Suppose we use the following ML `type` definition for our tree, where a `char` is data and everything else is an operation of some kind.

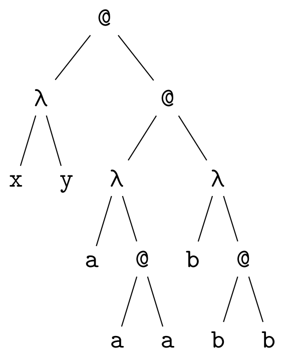
```
type Expr =
| Var of char
| Abs of char * Expr
| App of Expr * Expr
```

Applying the above definition to our expression, we obtain the following AST.



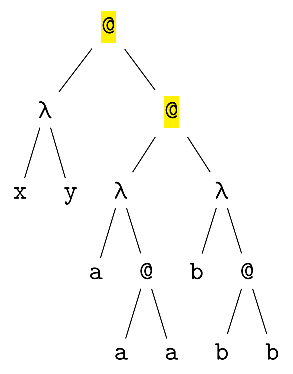
Much clearer, right? As you will see, I like to write lambda ASTs with a shorthand that makes them even easier to jot down. Everywhere we see `Var`, we simply replace it with its variable. Everywhere we see `Abs`, we write `λ`. Finally, everywhere we see `App`, we write `@`.[20]

[20] At-plication. Get it? No? OK, I think it's only amusing to me. Still, it's easier to write.



Identifying reducible expressions

Reducible expressions, or *redexes* for short, are the parts of a lambda expression where we can apply β-reductions. To find a redex, convert the expression to an AST and look for an application whose left side is an abstraction. If you think about this for a moment, this is like saying "look for a function definition that is being called." Here's the same AST with all the reducible applications highlighted.



Normal form

When do we stop doing reductions? We stop when there are no beta reductions left to do. One easy way to see that an expression can no longer be reduced is to look for redexes using our AST-drawing procedure. If there are no redexes, the expression is what we call a *normal form*.

For example, the following expressions are already in normal form:

```
x
xx
λx.y
xz
```

However, the following are not:

```
(λx.x)(λx.x)
(λx.λx.z)y
y(λx.xx)(λx.xx)
```

Try reducing the above expressions yourself. [21]

[21] The third expression has no normal form.

Normal order

In the tree above, the "outermost leftmost" reduction applies the argument `((λa.aa)(λb.bb))` to the function `(λx.y)`.[22] Always following the outermost leftmost beta reduction, at every step, is what we call the *normal order reduction.*

[22] "Outer" means *up* the tree in our diagrams.

Let's reduce this expression using the normal order.

`(λx.y)((λa.aa)(λb.bb))`	given
`([((λa.aa)(λb.bb))/x] y)`	β reduce `((λa.aa)(λb.bb))` for `x`
`y`	substitute and done

Applicative order

In the tree above, the "innermost leftmost" reduction applies the argument `(λb.bb)` to the function `(λa.aa)`.[23] Always following the innermost leftmost beta reduction, at every step, is what we call the *applicative order reduction*.

[23] "Inner" means *down* the tree in our diagrams.

Let's reduce this expression using the applicative order.

`(λx.y)((λa.aa)(λb.bb))`	given
`(λx.y)(([(λb.bb)/a] aa))`	β reduce `(λb.bb)` for `a`
`(λx.y)((λb.bb)(λb.bb))`	substitute
`(λx.y)((λa.[a/b] bb)(λb.bb))`	α reduce `a` for `b`
`(λx.y)((λa.aa)(λb.bb))`	uh-oh... we're back to where we started
...	

In this case, the applicative order reduction does not terminate. If you were following along, though, you know that we proved that the expression has a normal form because we were able to reduce it using the normal order.

Confluent, but with a catch

If an expression has a normal form, reductions can be applied in any order. Except, as you saw above, that's not the entire story. When we say "reductions can be applied in any order," we mean that you can never derive an incorrect expression by your choice of β reductions. Nevertheless, an unwise choice may reproduce an expression you had already evaluated.

Fortunately, there is an easy way to avoid the pain: choose the normal order. If a normal form exists for expression e, then the normal order reduction **will** find it. By contrast, if a normal form exists for expression e, then the applicative order reduction **may** find it.

You might be wondering why we even care about applicative order. It turns out that applicative order is equivalent to the order employed by most ordinary programming languages like Java and C! We call this kind of program evaluation *eager evaluation*. Very few languages utilize the normal order because it is difficult to implement, however there are examples. Haskell, for example, uses the normal order utilizing a form of evaluation we call *lazy evaluation*.

Nontermination

You might be thinking "I'm so glad I read this far in the course packet. Now all of my lambda calculus reductions will terminate!" I have sad news for you. As with ordinary programming languages, it is possible to write lambda expressions whose reductions will never terminate, regardless of your choice of reduction order. Consider the expression `(λa.aa)(λb.bb)`. Since there is only one redex, the normal order and the applicative order are the same. Go ahead, give the reduction a try. I'll wait.[24]

[24] Forever.

When we say that an expression does not have a normal form, nontermination is what we mean. The expression `(λa.aa)(λb.bb)` does not have a normal form.

Remember, the lambda calculus was designed to capture all the essential parts of computation. If it is possible to write an infinitely-looping program like the following in an ordinary language,

```
while(true) {}
```

then we should not be surprised that we are also able to write nonterminating programs in the lambda calculus.