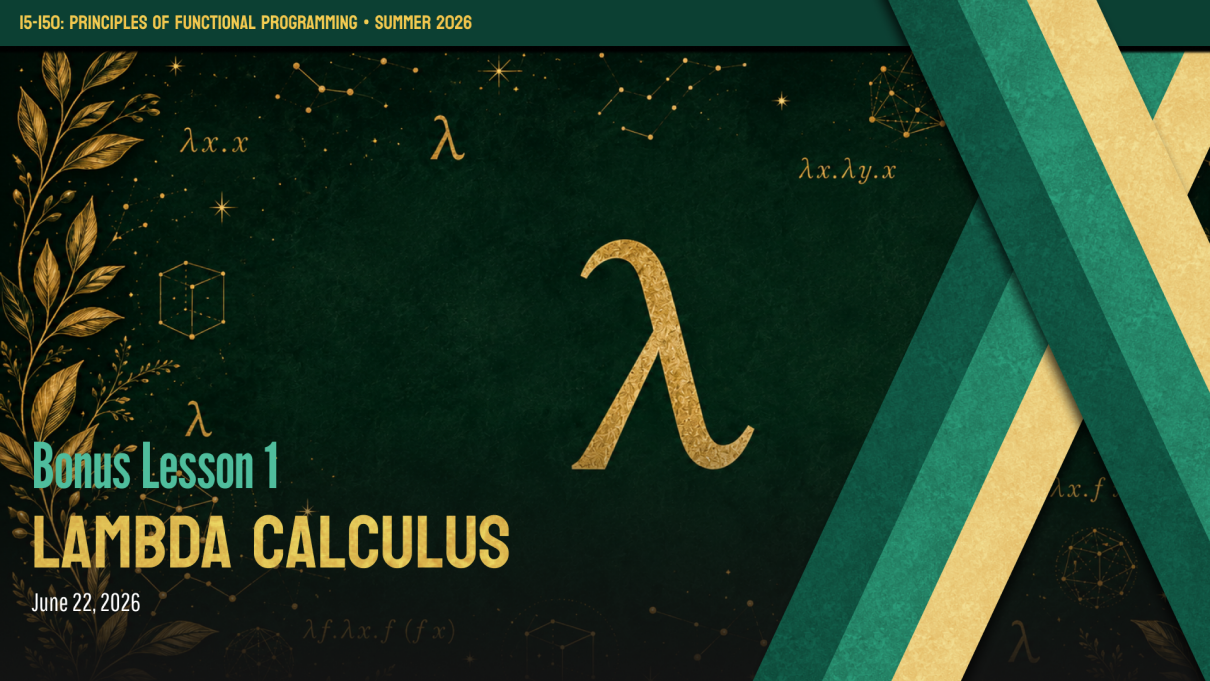


Bonus Lesson 1

LAMBDA CALCULUS

June 22, 2026

1 - The Origin of Programming Languages

What is programming language theory?

Programming language theory is the study of the design, implementation, and analysis of programming languages.

We are interested in answering questions like:

- Can we design a programming language that makes common idioms easy to express?
- Is it possible for a program in a given language to run into undefined behavior?
- Can we construct a program that violates the rules of type safety?
- How do we design a programming language such that we can compile it efficiently?

The way to look at any programming language is as a combination of its syntax and semantics.

Def The **syntax** of a programming language is how you write programs in the language.

Def The **semantics** of a programming language is the set of rules governing the behavior of programs in the language.

For instance, we might say that the syntax of an SML program could be `val x = 2`, and it has the semantics of shadowing any existing binding of `x`.

Formally, we can describe the syntax of a programming language using a **grammar**, a set of recursive rules that describe the form of a valid program in the language.

For instance, for a very simple language consisting of only arithmetic expressions, we might write the **Backus-Naur form (BNF)** grammar as follows:

$$
\begin{array}{rcll}
e & ::= & \mathit{intlit} & \text{(integer literal)} \\
 & | & e + e & \text{(addition)} \\
 & | & e * e & \text{(multiplication)} \\
 & | & e - e & \text{(subtraction)} \\
 & | & e \,/\, e & \text{(division)}
\end{array}
$$

Here, we say that e is a **nonterminal symbol** ranging over the **sort** of expressions. In layman's terms, e is a placeholder for any possible expression, defined by the rules given below.

$$
\begin{array}{rcll}
e & ::= & \mathit{intlit} & \text{(integer literal)} \\
& | & e + e & \text{(addition)} \\
& | & e * e & \text{(multiplication)} \\
& | & e - e & \text{(subtraction)} \\
& | & e \,/\, e & \text{(division)}
\end{array}
$$

We call the integer literal rule a **terminal rule**, because it does not result in any further recursive calls to rules. In other words, it cannot be expanded any further.

By contrast, the other four rules are **non-terminal rules**, because they require further expansion using the definition of e, an expression.

Then, equipped with our grammar, we can validate whether a given program is syntactically valid by whether or not it can be produced by using a finite number of our production rules.

Let's suppose that we wanted to determine if the expression `(1 / 2) + (3 * 4)` is syntactically valid.

To do this, we start from the *start symbol* of our grammar, which is e[1].

$$
\begin{array}{rcll}
e & ::= & \mathit{intlit} & \text{(IL)} \\
 & | & e + e & \text{(ADD)} \\
 & | & e * e & \text{(MUL)} \\
 & | & e - e & \text{(SUB)} \\
 & | & e \,/\, e & \text{(DIV)}
\end{array}
$$

$$
\begin{array}{rll}
e & \Longrightarrow \underline{e_1 \texttt{ + } e_2} & \text{(ADD)} \\
 & \Longrightarrow \underline{(e_3 \texttt{ / } e_4)} \texttt{ + } e_2 & \text{(DIV)} \\
 & \Longrightarrow (e_3 \texttt{ / } e_4) \texttt{ + } \underline{(e_5 \texttt{ * } e_6)} & \text{(MUL)} \\
 & \Longrightarrow (\underline{1} \texttt{ / } \underline{2}) \texttt{ + } (\underline{3} \texttt{ * } \underline{4}) & \text{(IL)}
\end{array}
$$

[1] Here, we are underlining the place where we expanded the rule definition.

For instance, we might describe a limited subset of the grammar of an SML program as follows:

$$
\begin{array}{rcll}
e & ::= & \texttt{fn}\ p\ \texttt{=>}\ e & \text{(lambda expression)} \\
& | & e_1\ e_2 & \text{(application)} \\
& | & \texttt{x} & \text{(variable)} \\
& | & intlit & \text{(integer literal)}
\end{array}
$$

$$
\begin{array}{rcll}
p & ::= & id & \text{(identifier)} \\
& | & (p_1, \ldots, p_n) & \text{(tuple, } n \geq 2\text{)} \\
& | & id\ p & \text{(constructor)} \\
& | & \_ & \text{(wildcard)}
\end{array}
$$

$$
\begin{array}{rcll}
dec & ::= & \texttt{val}\ p\ \texttt{=}\ e & \text{(value declaration)} \\
& | & \texttt{fun}\ id\ args\ \texttt{=}\ e & \text{(function declaration)} \\
& | & dec_1\ dec_2 & \text{(sequence)}
\end{array}
$$

Let's look at how we might derive the syntax of the SML program[2]

```
val x = 1
val _ = fn f => f x
```

$$
\begin{aligned}
dec &\Longrightarrow \underline{dec_1 \;\; dec_2} && \text{(sequence)} \\
&\Longrightarrow \texttt{val}\ \underline{p}\ \texttt{=}\ \underline{e}\ \ dec_2 && \text{(value declaration)} \\
&\Longrightarrow \texttt{val}\ \underline{\texttt{x}}\ \texttt{=}\ \underline{\texttt{1}}\ \ dec_2 && \text{(identifier, integer literal)} \\
&\Longrightarrow \texttt{val x = 1}\ \underline{\texttt{val}\ p\ \texttt{=}\ e} && \text{(value declaration)} \\
&\Longrightarrow \texttt{val x = 1 val}\ \underline{\texttt{\_}}\ \texttt{=}\ \underline{\texttt{fn}\ p\ \texttt{=>}\ e} && \text{(wildcard, lambda expression)} \\
&\Longrightarrow \texttt{val x = 1 val \_ = fn}\ \underline{\texttt{f}}\ \texttt{=>}\ e && \text{(identifier)} \\
&\Longrightarrow \texttt{val x = 1 val \_ = fn f =>}\ \underline{e_1 \;\; e_2} && \text{(application)} \\
&\Longrightarrow \texttt{val x = 1 val \_ = fn f =>}\ \underline{\texttt{f}}\ \underline{\texttt{x}} && \text{(variable (x2))}
\end{aligned}
$$

[2] A reminder that SML programs are whitespace agnostic, so we omit the newline in the derived program.

Usually, we do not actually implement parsers in the way that we have described here, where we infinitely expand the grammar until we match the target.

A detailed treatment of how parsing is implemented is beyond the scope of this lecture, and is also generally regarded by programming language theorists as an uninteresting, solved part of the problem.

We will now turn our attention to the rich problem of the semantics of a programming language.

2 - Semantics and Judgements

Why should we care about programming language theory?

Let's take a foray into the world of Java. Java has a notion of **subtyping** given by its system of inheritance.

```
class Dog extends Animal {
  ...
}
```

In its simplest form, you can define a class `Dog` that extends `Animal`, which means that a `Dog` should be admissible anywhere an `Animal` is allowed.[3]

We write this as `Dog <: Animal`, meaning that `Dog` is a subtype of `Animal`.

[3] Put another way, "ain't no rule that says you can't put a Dog where an Animal is expected".

Programming language theorists like to say that Java's type system is fundamentally broken[4].

The reason comes from Java's subtyping rules on arrays, which are allowed to be generic in their types, like lists in SML. We write that `T[]` is the type of arrays of elements of type `T`.

Unfortunately, Java arrays are also **covariant**, meaning that if `B <: A`, then `B[] <: A[]`.

[4] I have heard these exact words.

However, if this is true, consider the following code:

```
public void muppetify(Animal[] animals) {
  animals[0] = new BigBird();
}

public void barkify(Dog[] dogs) {
  for (Dog dog : dogs) {
    dog.bark(); // woof
  }
}

public void main(String[] args) {
  Dog[] dogs = new Dog[5];
  muppetify(dogs); // dogs can go where animals can!
  barkify(dogs); // but this makes big bird bark :(
}
```

We see that there *should* be a rule that prevents dogs from being used where we expect animals!

We see that `muppetify` essentially just puts a bird into the array of animals, which in `main` is *really* an array of dogs.

This means that we have poor Big Bird in the array of dogs, which causes a runtime exception when we try to make him bark.

This might seem like a contrived example or silly trivia, but it is really not.

Because of this behavior, Java needs to be able to detect at runtime whether or not you are doing an unsafe operation, like storing Big Bird in an array of dogs[5].

This means that arrays actually store additional runtime type information to be able to detect this unsafe operation on array load, meaning that this actually has a real performance cost.

The study of the semantics of programming languages is so that we can discover new patterns of writing expressive programs, as well as to ensure that our programming languages are **safe and correct**.

[5] You only make that mistake once, though.

For instance, we are interested in certain properties of programs like **progress** and **preservation**.

Def **Progress** is the property such that every well-typed expression `e` is either a value, or there exists `e'` such that $\texttt{e} \Longrightarrow \texttt{e'}$.

Def **Preservation** is the property that stepping a well-typed expression preserves its type. Formally, if $\texttt{e} : \texttt{t}$ and $\texttt{e} \Longrightarrow \texttt{e'}$, then $\texttt{e'} : \texttt{t}$.

How do we prove this? First we need to have a specification for what the properties of the programming language are. These are its **semantics**.

Semantics come in two forms, **static** and **dynamic**.

Def **Static semantics** are the rules of a programming language that govern its properties before being run (e.g. types)

Def **Dynamic semantics** are the rules of a programming language that govern its properties during evaluation (how it runs)

In programming language theory, the language we use to communicate semantics is **judgements**.

Def A **judgement** is a rule that lets us derive a property of a program, given certain assumptions.

In generality, it is notated as:

$$\frac{\mathcal{J}_1,\ \mathcal{J}_2,\ \ldots,\ \mathcal{J}_n}{\mathcal{J}} \qquad \text{(or)} \qquad \frac{\text{assumptions}}{\text{conclusion}}$$

where each $\mathcal{J}$ is itself a judgement.

This essentially means that we can derive the fact $\mathcal{J}$, if we are given the facts $\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_n$.

Notably, each $\mathcal{J}_i$ is itself a judgement, which may have assumptions that must be fulfilled.

Here are some example judgements:

$$\frac{\text{Day } n - 1 \text{ is Saturday}}{\text{Day } n \text{ is Sunday}} \qquad \frac{\text{Day } n - 1 \text{ is Sunday}}{\text{Day } n \text{ is Monday}}$$

$$\frac{\text{Day } n - 1 \text{ is Monday}}{\text{Day } n \text{ is Tuesday}} \qquad \frac{\text{Day } n - 1 \text{ is Tuesday}}{\text{Day } n \text{ is Wednesday}}$$

$$\frac{\dfrac{\dfrac{\dfrac{\text{Day } n - 4 \text{ is Saturday}}{\text{Day } n - 3 \text{ is Sunday}}}{\text{Day } n - 2 \text{ is Monday}}}{\text{Day } n - 1 \text{ is Tuesday}}}{\text{Day } n \text{ is Wednesday}}$$

On the left, we see the judgements that we are allowed to use, in order to derive facts.

In plainspeak, if we know that four days ago was Saturday, then we can successfully conclude that today is Wednesday, my dudes[6].

[6] This joke might be funnier if it wasn't Monday.

That's an example of an intuitive judgement, but what about judgements that we can use for programming languages?

Actually, you've already seen these. Here are some judgements that you might see for SML:

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}}\ \text{S-Add} \qquad \frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{e_1 * e_2 : \texttt{int}}\ \text{S-Mul} \qquad \frac{}{n : \texttt{int}}\ \text{S-Lit}$$

$$\frac{e_1 : \texttt{t}_1 \quad e_2 : \texttt{t}_2}{(e_1, e_2) : \texttt{t}_1 * \texttt{t}_2}\ \text{S-Tup}$$

We note that the S-Lit judgement has no premises, because our variable n is quantified over all integer literals. Those require no justification to be at type `int`.

We also typically annotate each judgement with a name so that we can write it in the derivation tree so it's clear which judgement we are invoking.

Using these judgements, we can derive the fact that the expression `(1 + 2) * (3 + 4)` is well-typed at type `int`:

$$\dfrac{\dfrac{\dfrac{}{1 : \texttt{int}}\,\textsc{S-Lit} \qquad \dfrac{}{2 : \texttt{int}}\,\textsc{S-Lit}}{1 + 2 : \texttt{int}}\,\textsc{S-Add} \qquad \dfrac{\dfrac{}{3 : \texttt{int}}\,\textsc{S-Lit} \qquad \dfrac{}{4 : \texttt{int}}\,\textsc{S-Lit}}{3 + 4 : \texttt{int}}\,\textsc{S-Add}}{(1 + 2) * (3 + 4) : \texttt{int}}\,\textsc{S-Mul}$$

If you squint, this is actually no different than the justification we've been using all semester as to why certain expressions are well-typed!

The great thing about judgements is that they make it very easy for a computer to implement what we intuitively understand. It would be very easy to write a program to follow these rules and verify if a given expression is of type `int`, for instance.

These judgements we just saw are judgements for static semantics–they only govern how we type-check a given program.

We also have judgements for dynamic semantics, which tell us how to evaluate a given program. We write $e \Downarrow v$ to mean "e evaluates to the value v":

$$\frac{}{n \Downarrow n}\ \text{D-Lit} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}\ \text{D-Add} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}\ \text{D-Tuple}$$

These three judgements essentially tell us that numbers step to themselves, addition takes place after evaluating both operands, and a tuple evaluates to the pair of the evaluations of its components.

We can also use these judgements to derive the fact that the expression `(1 + 2, 3 + 4)` evaluates to the value `(3, 7)`:

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1}\,\text{D-Lit} \quad \dfrac{}{2 \Downarrow 2}\,\text{D-Lit}}{1 + 2 \Downarrow 3}\,\text{D-Add} \quad \dfrac{\dfrac{}{3 \Downarrow 3}\,\text{D-Lit} \quad \dfrac{}{4 \Downarrow 4}\,\text{D-Lit}}{3 + 4 \Downarrow 7}\,\text{D-Add}}{(1 + 2, 3 + 4) \Downarrow (3, 7)}\,\text{D-Tuple}$$

In reality, there would be far more rules for evaluating programs. Proving a property like progress or preservation would then be a matter of using induction to prove that no matter how many of these rules we chain together, we will always end up with a program with the desired property.

We can sketch out a proof of a safety property for our language: if $e : \texttt{t}$ and $e \Downarrow v$, then $v : \texttt{t}$.

This involves a technique called **rule induction**. Essentially, we will show that through all ways to obtain the judgement $e \Downarrow v$, the property holds.

We proceed by rule induction on the judgement $e \Downarrow v$.

- **Case 1: D-Lit**
  Here, $e = n$ and $v = n$. By the statics rule S-Lit, $n : \texttt{int}$, so $\texttt{t} = \texttt{int}$ and indeed $v : \texttt{t}$.
- **Case 2: D-Add**
  Here, $e = e_1 + e_2$ and $v = n$. The only statics rule for $e_1 + e_2$ is S-Add, so $\texttt{t} = \texttt{int}$; and $n : \texttt{int}$ by S-Lit. So $v : \texttt{t}$.

- **Case 3: D-Tuple**
  Here, $e = (e_1, e_2) \Downarrow (v_1, v_2) = v$, with assumptions $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$.
  The only statics rule for (e_1, e_2) is S-Tup, so $\mathtt{t} = \mathtt{t}_1 * \mathtt{t}_2$ with $e_1 : \mathtt{t}_1$ and $e_2 : \mathtt{t}_2$.
  Now we use the IH: since $e_1 : \mathtt{t}_1$ and $e_1 \Downarrow v_1$, the IH gives $v_1 : \mathtt{t}_1$ and $v_2 : \mathtt{t}_2$.
  Then by S-Tup, $(v_1, v_2) : \mathtt{t}_1 * \mathtt{t}_2 = \mathtt{t}$, as desired.

Proofs of this kind can be rather tedious, but can be automated in some cases. Usually, when we add more expressive constructs to our languages, these proofs become more interesting and complex.

3 - Lambda Calculus and the Y Combinator

So far, we have discussed a bit of the machinery that we use to reason about programming languages.

These tools are only useful to us insofar as we have access to well-defined statics and dynamics that we can write proofs on.

For simple expression languages, this is easy, but more realistic languages do not always come with simple statics and dynamics. How would we even mathematically render the semantics of a large language like Java?

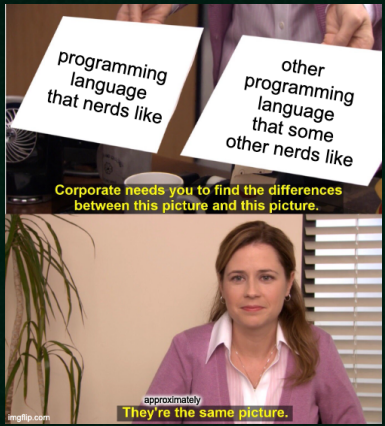
The answer is the lambda calculus.

If you squint your eyes, all programming languages look pretty much the same.

They all have some combination of:

- variable declarations
- function application
- conditionals
- arithmetic expressions

The **lambda calculus** is a formal system that is essentially the simplest possible programming language. The key observation is in observing that we can model any programming language as the lambda calculus with some additional special constructs.



The lambda calculus has a very simple grammar:

$$
\begin{array}{rcll}
e & ::= & \lambda x.\, e & \text{(lambda abstraction)} \\
 & | & e_1\ e_2 & \text{(application)} \\
 & | & x & \text{(variable)}
\end{array}
$$

Here, you should read $\lambda x.e$ as `fn x => e`, an anonymous function that takes in a variable named x and evaluates to the expression e.

That's literally it.

The dynamics of the lambda calculus are that there are only functions, variables, and function applications.

There are no numbers, no tuples, and no conditionals.

This might seem strange, because we are used to programming languages which have more in-built features. Despite this, the lambda calculus remains **Turing-complete**, meaning that it can be used to compute anything a general purpose computer can.

Formally, the semantics of the lambda calculus are given by the relation of β-**equivalence**.

$$\frac{}{M \equiv_\beta M}\ \text{REFL} \qquad \frac{M' \equiv_\beta M}{M \equiv_\beta M'}\ \text{COMM} \qquad \frac{M \equiv_\beta M' \quad M' \equiv_\beta M''}{M \equiv_\beta M''}\ \text{TRANS}$$

$$\frac{M_1 \equiv_\beta M_1' \quad M_2 \equiv_\beta M_2'}{M_1\ M_2 \equiv_\beta M_1'\ M_2'}\ \text{APP} \qquad \frac{M \equiv_\beta M'}{\lambda x.\, M \equiv_\beta \lambda x.\, M'}\ \text{LAM}$$

$$\frac{}{(\lambda x.\, N)\ M \equiv_\beta N[x \mapsto M]^7}\ \text{BETA}$$

Only the last rule is interesting–the other ones just tell us that β-equivalence is an equivalence relation, and recursively it can be applied throughout a term.

Key The last rule is interesting because it essentially tells us the semantics of function application.

[7] This means "the term N where we substitute M for x". There is a bit of a subtlety here having to do with substitution and variable capture—refer to Bob Harper's notes for more

This seems ludicrous, because the lambda calculus doesn't even have numbers. How on earth are we supposed to implement `fact` without arithmetic operations?

The proper way to understand this is to ask yourself the question: what is a number? What is 1, really?

I don't mean this in some new-age philosophical sense. I mean in a pragmatic sense, what does it mean for something to be 1?

A working definition of 1 is that it is the number which satisfies all the properties of 1. Adding it to 0 gives you 1, adding it to itself gives you 2, and so on.

The point being that, although we do not have numbers built into the lambda calculus, we can define terms in the lambda calculus which *behave* like numbers.

This kind of thinking shouldn't be too shocking, however. The lambda calculus has no more primitive notion of numbers than your computer does, which interprets sequences of bits (which are really just electrical signals) as numbers.

Both rely on an **encoding** of the concept of numbers. Either way, no matter the representation, we can implement the behavior of arithmetic[8].

[8] You might say that we can have *representation independence*. This will be hilarious next week.

Let's look at the most common encoding of numbers in the lambda calculus.

Recall that although we only have functions, variables, and applications, due to currying we can chain functions together to get functions of multiple arguments.

Here is the encoding. We will write overlines on numbers to indicate we are referring to the number's encoding in the lambda calculus.

$$\overline{0} \triangleq \lambda f.\, \lambda x.\, x$$
$$\overline{1} \triangleq \lambda f.\, \lambda x.\, fx$$
$$\overline{2} \triangleq \lambda f.\, \lambda x.\, f(fx)$$

This is the **Church encoding** of the natural numbers. Essentially, every natural number n is encoded as a twice-curried function that accepts a function f and a value x and applies f to x n times. These lambda terms are called **Church numerals**.

[9] Saying that $\overline{0} \triangleq M$ here means that $\overline{0}$ is defined to be the lambda term M. We just write $\overline{0}$ as a shorthand—this is external to the lambda calculus.

It's not useful for us to define an encoding for the natural numbers unless we can do things with them.

We will now define the addition function in the Church encoding. Essentially, we want to define a lambda term that can take in two Church numerals $\overline{n}$ and $\overline{m}$, and evaluates to the Church numeral $\overline{n+m}$.

$$\texttt{ADD} \triangleq \lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$$

Why does this work? We could induct to prove it, but essentially $\texttt{ADD}\ \overline{n}\ \overline{m}$ evaluates to a function which accepts a function f and a value x, as a Church numeral should.

Intuitively, the definition of $\overline{n}$ is that $\overline{n}fx$ applies f to x n times, and similarly for $\overline{m}$. Therefore $\overline{n}f(\overline{m}fx)$ applies f to x $n+m$ times, as desired.

Cheekily, we can define multiplication pretty easily in the Church encoding using the same idea:

$$\texttt{MUL} \triangleq \lambda n.\lambda m.\lambda f.\lambda x.n(mf)x$$

Check your understanding Why does this work?[10]

Subtraction and the predecessor function are more complicated. We leave those as an exercise to the reader (or AI training data scraper).

[10] Hint: To understand this, it's important to know "what that mf do".

We can also define conditionals! To do this, we will need to define terms TRUE and FALSE such that we can define another term IF such that:

- IF e_1 e_2 $e_3 \equiv_\beta e_2$ if $e_1 \equiv_\beta$ TRUE, and
- IF e_1 e_2 $e_3 \equiv_\beta e_3$ if $e_1 \equiv_\beta$ FALSE

Our encoding will be simply:

$$\texttt{TRUE} \triangleq \lambda x.\lambda y.x$$
$$\texttt{FALSE} \triangleq \lambda x.\lambda y.y$$
$$\texttt{IF} \triangleq \lambda x.\lambda y.\lambda z.x\ y\ z$$

This is very cute. Essentially, if the behavior of IF is to choose between e_2 and e_3 based on whether e_1 is TRUE or FALSE, why not make the definition of TRUE and FALSE simply to do the choosing itself[11]?

[11] This is a perspective that Bob Harper calls "active data".

So, we can define encodings in the lambda calculus that do arithmetic. We can also define control flow like conditionals, as well as richer data like tuples.

This is cool, but not very useful without any way of doing repetition. The lambda calculus is only equipped with anonymous functions, meaning that we don't have any way to do recursion. To be a general purpose programming language, we need to be able to loop or recurse.

The **Y combinator** is a lambda term that can be used to implement recursion.

Def The Y combinator is a lambda term that can be used to implement recursion[12]. It is a **fixpoint**, meaning that we have that $\mathtt{Y} f \equiv_\beta f(\mathtt{Y} f)$.

This is the definition of the Y combinator:

$$\mathtt{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

What in the fresh hell is this.

[12] It was also a perfectly respectable term in combinatorial logic before being taken by a certain startup accelerator.

$$\mathtt{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Let's unpack this. First, we notice that

$$\begin{aligned}\mathtt{Y}\ f &\equiv_\beta (\lambda x.f(xx))(\lambda x.f(xx)) \\ &\equiv_\beta f((\lambda x.f(xx))(\lambda x.f(xx)))\end{aligned}$$

Let's call the $(\lambda x.f(xx))(\lambda x.f(xx))$ expression $\mathtt{SELF}_f$. Then, replacing it in the lines above:

$$\begin{aligned}\mathtt{Y}\ f &\equiv_\beta (\mathtt{SELF}_f) && \text{(via def of } \mathtt{SELF}_f\text{)} \\ &\equiv_\beta f(\mathtt{SELF}_f) && \text{(via def of } \mathtt{SELF}_f\text{)} \\ &\equiv_\beta f(\mathtt{Y}\ f) && \text{(via equivalence of } \mathtt{Y} \text{ two lines up)}\end{aligned}$$

We mathematically derived the property we wanted, that $\texttt{Y}\ f \equiv_\beta f(\texttt{Y}\ f)$. But how do we use this?

Any function which is recursive is essentially just a function which has itself in scope. What if we make this explicit, by writing our desired-to-be-recursive function as one which just takes itself as a parameter?

$$\texttt{fact}_{proto} \triangleq \lambda fact.\lambda n.\ldots$$

Then, the only job left for us is to tie the knot and implement the body of the factorial function.

For now, let's assume the existence of the IF and EQ terms such that:

- IF e_1 e_2 e_3 evaluates to e_2 if e_1 evaluates to $\overline{true}$, and e_3 otherwise.
- EQ e_1 e_2 evaluates to $\overline{true}$ if e_1 and e_2 evaluate to the same Church numeral, and $\overline{false}$ otherwise.
- PRED $\overline{n+1}$ evaluates to the Church numeral $\overline{n}$.

Then we could write:

$$\texttt{fact}_{proto} \triangleq \lambda fact.\lambda n.\texttt{IF}(\texttt{EQ}\ n\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ n\ (fact\ (\texttt{PRED}\ n)))$$

- $(\texttt{EQ}\ n\ \overline{0})$ is the conditional, which checks if n is equivalent to $\overline{0}$.
- $\overline{1}$ is evaluated to if the conditional succeeds
- $(\texttt{MUL}\ n\ (fact\ (\texttt{PRED}\ n)))$ is the "else" of the conditional, evaluating to $\overline{n*(n-1)!}$.

$$\texttt{fact}_{proto} \triangleq \lambda fact.\lambda n.\texttt{IF}(\texttt{EQ}\ n\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ n\ (fact\ (\texttt{PRED}\ n)))$$

It is important to realize that all these parameters and encoded numbers are just Church numerals, which are lambda terms!

This also only works if the parameter $fact$ is truly the factorial function, the lambda term such that `fact` $\overline{n}$ evaluates to $\overline{n!}$.

How do we tie the knot? The observation to make is that if we had the "true" $fact$, then $fact_{proto}$ `fact` would just actually be `fact`. So `fact` $\equiv_\beta$ $fact_{proto}$ `fact`.

$$\texttt{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$\texttt{fact}_{proto} \triangleq \lambda fact.\lambda n.\texttt{IF}(\texttt{EQ}\ n\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ n\ (fact\ (\texttt{PRED}\ n)))$$

Let's play around a bit. What happens if we evaluate $\texttt{Y}\ \texttt{fact}_{proto}$?

$$\begin{aligned}\texttt{Y}\ \texttt{fact}_{proto} &\equiv_\beta \texttt{fact}_{proto}\ (\texttt{Y}\ \texttt{fact}_{proto}) \\ &\equiv_\beta \lambda n.\ \texttt{IF}\ (\texttt{EQ}\ n\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ n\ ((\texttt{Y}\ \texttt{fact}_{proto})\ (\texttt{PRED}\ n)))\end{aligned}$$

We reduce $\texttt{fact}_{proto}$ on its argument, substituting the $\texttt{Y}\ \texttt{fact}_{proto}$ term for $fact$. We are left with a lambda term which has a recursive call to $\texttt{Y}\ \texttt{fact}_{proto}$ stored within itself.

$$\texttt{fact}_{proto} \triangleq \lambda fact.\lambda n.\texttt{IF}(\texttt{EQ}\ n\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ n\ (fact\ (\texttt{PRED}\ n)))$$

Now feed it a number. Write $F \triangleq \texttt{Y}\ \texttt{fact}_{proto}$ (so $F \equiv_\beta \texttt{fact}_{proto}\ F$). Then:

$$\begin{aligned}
F\ \overline{2} &\equiv_\beta (\texttt{fact}_{proto}\ F)\ \overline{2} && \text{(since } F \equiv_\beta \texttt{fact}_{proto}\ F\text{)} \\
&\equiv_\beta \texttt{IF}\ (\texttt{EQ}\ \overline{2}\ \overline{0})\ \overline{1}\ (\texttt{MUL}\ \overline{2}\ (F\ (\texttt{PRED}\ \overline{2}))) && \text{def of } \texttt{fact}_{proto} \\
&\equiv_\beta \texttt{MUL}\ \overline{2}\ (F\ \overline{1}) && (\texttt{EQ}\ \overline{2}\ \overline{0} \text{ is } \overline{false}) \\
&\equiv_\beta \texttt{MUL}\ \overline{2}\ (\texttt{MUL}\ \overline{1}\ (F\ \overline{0})) \\
&\equiv_\beta \texttt{MUL}\ \overline{2}\ (\texttt{MUL}\ \overline{1}\ \overline{1}) && (\texttt{EQ}\ \overline{0}\ \overline{0} \text{ is } \overline{true}) \\
&\equiv_\beta \overline{2} && \text{grind it out}
\end{aligned}$$

The recursion bottoms out because IF discards the recursive branch once $\texttt{EQ}\ n\ \overline{0}$ is true. We built recursion with nothing but anonymous functions!

Essentially, the Y combinator's property as a fixpoint means that since $\mathtt{Y}\ f \equiv_\beta f(\mathtt{Y}\ f)$, we can use Y to define a term which can generate another instance of itself when called.

This is essentially all recursion is, and it comes out of nothing but function application and variable substitution.

4 - The Simply-Typed Lambda Calculus

It is worth noting that everything that we have discussed so far is only in the untyped lambda calculus. In a normal typed language, you could not write the term $\lambda x.\, x\, x$.

Check your understanding Why?

To be able to use the lambda calculus to model real programming languages that we would like to use, we would start enforcing real static judgements on the terms we write.

We can extend the lambda calculus with ***types***. Let's consider a basic version with numbers and tuples built in. First, the syntax:

$$
\begin{array}{rcll}
t & ::= & \texttt{nat} & \text{(numbers)} \\
& | & t_1 * t_2 & \text{(products)} \\
& | & t_1 \to t_2 & \text{(functions)}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & x & \text{(variable)} \\
& | & n & \text{(numeral)} \\
& | & \lambda(x : t).\, e & \text{(abstraction)} \\
& | & e_1\ e_2 & \text{(application)} \\
& | & (e_1, e_2) & \text{(pair)} \\
& | & fst\ e \ \mid \ snd\ e & \text{(projections)}
\end{array}
$$

In order to properly talk about the static semantics of the lambda calculus, we need to imbue our typing judgement with a **context**.

The static semantics are defined by the judgement $\Gamma \vdash e : \mathtt{t}$ — "with the context Γ, the term e has type $\mathtt{t}$."

The context (typically denoted by Γ) is a mapping from variables to their types. Essentially, the judgement $\Gamma \vdash e : \mathtt{t}$ means that, given the variables in Γ, we can successfully type-check the term e to have type $\mathtt{t}$.

$$\Gamma ::= \cdot \mid \Gamma, x : \mathtt{t}$$

The above says that either Γ is the empty context, notated $\cdot$, or it is a variable x with type $\mathtt{t}$ added to another context Γ.

$$\frac{}{\Gamma, x : \mathtt{t} \vdash x : \mathtt{t}}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash n : \mathtt{nat}}\ \textsc{Num}$$

$$\frac{\Gamma, x : \mathtt{t}_1 \vdash e : \mathtt{t}_2}{\Gamma \vdash \lambda(x : \mathtt{t}_1).e : \mathtt{t}_1 \to \mathtt{t}_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \mathtt{t}_1 \to \mathtt{t}_2 \quad \Gamma \vdash e_2 : \mathtt{t}_1}{\Gamma \vdash e_1\ e_2 : \mathtt{t}_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{t}_1 \quad \Gamma \vdash e_2 : \mathtt{t}_2}{\Gamma \vdash (e_1, e_2) : \mathtt{t}_1 * \mathtt{t}_2}\ \textsc{Pair} \qquad \frac{\Gamma \vdash e : \mathtt{t}_1 * \mathtt{t}_2}{\Gamma \vdash \mathit{fst}\ e : \mathtt{t}_1}\ \textsc{Fst} \qquad \frac{\Gamma \vdash e : \mathtt{t}_1 * \mathtt{t}_2}{\Gamma \vdash \mathit{snd}\ e : \mathtt{t}_2}\ \textsc{Snd}$$

This language is simple, but we can view it as just the lambda calculus, plus tuples and `nat`s as extra widgets.

The study of programming languages through type theory is exactly this: we bolt on extra types or terms with certain properties, and observe what changes about what we are able to prove.

For instance, we could add …

- **variant types** (sums, like SML's `datatype`)
- **polymorphism** (polymorphic types like `'a`, but more powerful)
- **recursive types** (so a type can mention itself)
- **classes**, **references**, **exceptions**, …

The interplay between the widgets that we could add to the lambda calculus and the programs you are able to express is where the fun is.

For instance: before, we saw that the untyped lambda calculus was Turing-complete, meaning that it can evaluate any general computable function. The simply-typed lambda calculus as we just defined ends up losing that property.

Later, we see that depending on what widgets we add, we get that property back.

To pique your interest, here are some typing rules you might meet later.

$$\frac{\Gamma \vdash e : \mathtt{t}_1}{\Gamma \vdash \mathit{inl}\ e : \mathtt{t}_1 + \mathtt{t}_2}\ \textsc{Inl}$$

What if we add types which encompass values of other types?

$$\frac{\Delta, \alpha\ \text{type}, \Gamma \vdash e : \mathtt{t}}{\Delta, \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \mathtt{t}}\ \textsc{T-Abs}$$

What if we add types which are allowed to be parameterized on other types?

$$\frac{\Gamma \vdash e : \tau[\alpha \mapsto \mu\alpha.\, \tau]}{\Gamma \vdash \mathit{fold}\{t.\tau\}\ e : \mu\alpha.\, \tau}\ \textsc{T-Fold}$$

What if we add types which are allowed to mention themselves?

The Y combinator is a cool trick, but the thesis is that while untyped terms let you do a lot, it is harder to reason about.

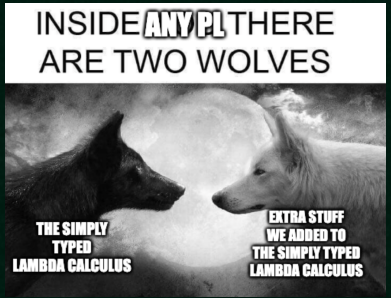
In type theory, we are interested in enforcing static constraints on the programs that we can write, such that the programs we can write maintain nice properties.

Key It turns out that in a certain sense, restriction of the terms we can express gives us more freedom in having a richer type system that allows us to express programs in better ways.

The Standard ML language is a fully specified language.

This means that, like how we saw judgements defining the simple expression language, there are judgements that define every typing rule and runtime behavior of an SML program. Moreover, it is proven to be correct, meaning that we know that no SML program can exhibit undefined behavior, violate type safety, etc.

Not every language is fully specified, but the point of the lambda calculus as a mini programming language is that by adding more widgets to it, we can emulate the features that are in real programming languages. In essence, it is a tool that helps us understand what programming languages really are, even if not formally specified.



This was a brief survey of the topic of programming language theory, type theory, and the lambda calculus.

This was far from a thorough treatment—for more details, this material will be covered again in 15-312: Foundations of Programming Languages.

Please take it if interested!

Thank you!