CPSC 311: Functions **(DRAFT)**
("lec-functions")

Joshua Dunfield
University of British Columbia

September 26, 2016

Updated 2016-09-25 with substitution (Figure 2) and some discussion from the 2016-09-23 lecture (Section 4.3)

1 Topics discussed

- Recipe for extending a language
- The "Fun language": concrete syntax, abstract syntax
- The Fun language: evaluation rules
  - why evaluation in Fun produces a *value*, not just a number
  - evaluation rule for `lam`
  - updated evaluation rules for features already in our language
  - rule for function application?
- rule for function application: Eval-app-expr
- the "expression strategy"
- example with Eval-app-expr
- the "method of hope"; complete derivations
- another example with Eval-app-expr
- the "value strategy" and Eval-app-value
- advantages and disadvantages of each strategy
- interpreter for Fun (**Racket code**: `dynsem-fun.rkt` ⇐ **2015 VERSION, OUT OF DATE**)
- first-class functions
  - mathematicians also think they're weird
  - functions are values...
  - ...but are displayed in an unhelpful way by most language implementations
  - when are functions equal?

2 Functions

We started our journey through evaluation semantics with a language of arithmetic expressions. We got a slightly larger language by adding the Let construct, which lets us give names to values and then use those names. To model Let in the evaluation semantics, we had to define substitution.

Adding just one more feature will get us to a surprisingly powerful language.

To add functions, we'll try to follow the same recipe as for Let:

1. **Extend the concrete syntax** (EBNF grammar).
2. **Extend the abstract syntax** (**define-type**).
3. **Add evaluation rules.**

After finishing these steps, we will have extended our *language*: a language is defined by its syntax and semantics. To implement the language, we'll need to extend our parsing function (to reflect the new syntax) and our interpreter (to reflect the new evaluation rules).

Remark. In real life, or at least in real programming languages research, there would probably be a fourth step: **Prove that the new evaluation rules have good properties.** Exactly what properties are "good" depends on the language. For our language so far, one good property (the only one I can think of right now) would be *determinism*, which says that evaluating the same expression should give consistent results:

$$\text{"If } e \Downarrow n_1 \text{ and } e \Downarrow n_2\text{, then } n_1 = n_2\text{."}$$

Later in 311, we'll discuss these sorts of properties (without actually proving them).

3 The Fun language: syntax

It's time to add functions to our language. Nearly a century of tradition would have us use the syntax "lambda" or λ, but to help distinguish the lambda in Fun from the lambda in Racket (which can also be written λ), we'll use "Lam".

Instances of the identifier bound by `Lam` can be represented by `Id`, just as we did for `Let`.

⟨E⟩ ::= ⟨num⟩
| {+ ⟨E⟩ ⟨E⟩}
| {− ⟨E⟩ ⟨E⟩}
| ⟨symbol⟩
| {Let ⟨symbol⟩ ⟨E⟩ ⟨E⟩}
| {Lam ⟨symbol⟩ ⟨E⟩}

And here's the abstract syntax:

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Id (name symbol?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Lam (name symbol?) (body E?)]
)
```

We're not done with the syntax, though. In a previous lecture, I mentioned that a language can be organized by what kinds of data it has, and how it "introduces" and "eliminates" each kind of data. Until now, our languages only had one kind of data, numbers; functions are a new kind of data. We can introduce functions with "Lam", but we need a way to use them. We'll call this "App". The first expression will represent the function being applied, and the second expression will represent the argument—what the function is being applied to.

⟨E⟩ ::= ⟨num⟩
| {+ ⟨E⟩ ⟨E⟩}
| {− ⟨E⟩ ⟨E⟩}
| ⟨symbol⟩
| {Let ⟨symbol⟩ ⟨E⟩ ⟨E⟩}
| {Lam ⟨symbol⟩ ⟨E⟩}
| {App ⟨E⟩ ⟨E⟩}

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
)
```

4 The Fun language: Evaluation rules

We'll try to reuse all the rules from before. Because we don't know yet whether that will work, we'll write a ? before each rule name to emphasize its provisional status.

When we write a meta-variable e in a rule, it will stand for a Fun expression.

$$\frac{}{(\mathsf{Num}\ n) \Downarrow n}\ \text{?Eval-num} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\mathsf{Add}\ e1\ e2) \Downarrow n_1 + n_2}\ \text{?Eval-add} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\mathsf{Sub}\ e1\ e2) \Downarrow n_1 - n_2}\ \text{?Eval-sub}$$

$$\frac{e1 \Downarrow n1 \qquad [n1/x]e2 \Downarrow n2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow n2}\ \text{?Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \text{free-variable-error}}\ \text{?Eval-free-identifier}$$

We added two productions to the EBNF and two variants to the abstract syntax, so we expect to need two new evaluation rules. We might also expect (following the pattern of Eval-add, Eval-sub, and Eval-let) that, since a Lam has one expression inside it and an App has two expressions inside it, the rule for Eval-lam will have one premise and the rule for Eval-app will have two premises:

$$\frac{??}{(\mathsf{Lam}\ x\ e1) \Downarrow ??}\ \text{??Eval-lam} \qquad \frac{?? \qquad ??}{(\mathsf{App}\ e1\ e2) \Downarrow ??}\ \text{??Eval-app}$$

Let's think about ??Eval-lam. What should its premise be? The only expression we have is $e1$, so maybe we should evaluate $e1$ in the premise.

$$\frac{e1 \Downarrow n1}{(\mathsf{Lam}\ x\ e1) \Downarrow n1??}\ \text{??Eval-lam}$$

This seems to follow the pattern of our other rules, but (as with programming) we should think about examples (test cases). What is the *simplest possible* function? I claim it is the identity function, $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$. So let's try that function with our proposed rule:

$$\frac{(\mathsf{Id}\ x) \Downarrow \_\_\_}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow \_\_\_}$$

Now we have a problem: the expression $(\mathsf{Id}\ x)$ doesn't evaluate to anything—instead, we get a "free-variable-error". So we can't derive $(\mathsf{Id}\ x) \Downarrow \_\_\_$.

The problem is that we're trying to evaluate what's *inside* the function before we've applied it *to* anything! We don't know what x is until we pass the function an argument. A function is a machine that turns arguments into results; evaluating a function without an argument is like running a dishwasher when it's empty.

We already have numbers that evaluate to themselves (Eval-num), so we'll make functions evaluate to themselves as well.

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam}$$

Irritatingly, this doesn't quite match what we've done so far. Instead of always evaluating to numbers—deriving

$$\ldots \Downarrow n$$

where n is a number, we can now use Eval-lam to derive

$$\ldots \Downarrow (\mathsf{Lam}\ x\ e1)$$

We could say that evaluation produces either a number n or an expression of the form $(\mathsf{Lam}\ x\ e1)$. It will be a little easier to say that evaluation produces a *value*, where a value is a particular kind of expression: one that is either $(\mathsf{Num}\ n)$, or $(\mathsf{Lam}\ x\ e1)$.[1]

Making this change requires several changes to our rules:

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam}$$

Interestingly, one of the rules—Eval-let—became simpler, and more general: it should now work with any kind of value $v1$, not just numbers.

Figure 1 summarizes all our rules so far—we're still missing a rule for App.

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\mathsf{Num}\ n_1) \qquad e2 \Downarrow (\mathsf{Num}\ n_2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\mathsf{Id}\ x)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\mathsf{Lam}\ x\ e1) \Downarrow (\mathsf{Lam}\ x\ e1)}\ \text{Eval-lam}$$

Figure 1 Evaluation rules for Fun (still missing a rule for App)

Figure 2 defines substitution for this language. The rules for substitution into a Lam echo the rules for substitution into a Let, except that a Lam has only one expression inside it.

4.1 Evaluating application

Here's our guess at the shape of the Eval-app rule, with a lot of ??'s.

$$\frac{?? \qquad ??}{(\mathsf{App}\ e1\ e2) \Downarrow ??}\ ??\text{Eval-app}$$

Ideas?

[1] By themselves, values are inert. They don't do anything. A function does nothing until it's called. Unfortunately, many "real" programming languages make this hard to remember. For example, DrRacket claims that the result of evaluating `(lambda (x) x)` is "`#<procedure>`". But you should think of that as just a strange notation for the function you entered. It's still `(lambda (x) x)`.

$$
\begin{array}{rcll}
[v/x](\mathsf{Num}\ n) & = & (\mathsf{Num}\ n) & \\
[v/x](\mathsf{Add}\ e1\ e2) & = & (\mathsf{Add}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{Sub}\ e1\ e2) & = & (\mathsf{Sub}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{App}\ e1\ e2) & = & (\mathsf{App}\ [v/x]e1\ [v/x]e2) & \\
[v/x](\mathsf{Id}\ y) & = & v & \text{if } x = y \\
[v/x](\mathsf{Id}\ y) & = & (\mathsf{Id}\ y) & \text{if } x \neq y \\
[v/x](\mathsf{Let}\ y\ e1\ e2) & = & (\mathsf{Let}\ y\ [v/x]e1\ e2) & \text{if } x = y \\
[v/x](\mathsf{Let}\ y\ e1\ e2) & = & (\mathsf{Let}\ y\ [v/x]e1\ [v/x]e2) & \text{if } x \neq y \\
[v/x](\mathsf{Lam}\ y\ eB) & = & (\mathsf{Lam}\ y\ eB) & \text{if } x = y \\
[v/x](\mathsf{Lam}\ y\ eB) & = & (\mathsf{Lam}\ y\ [v/x]eB) & \text{if } x \neq y
\end{array}
$$

Figure 2 Substitution for Fun

At this point, I was expecting to (eventually) reach a particular rule, but it's not the one that you suggested during lecture. (I should have expected this, since I suggested that we have a rule with two premises, and the one I was thinking of has three premises!) However, the rule you suggested is quite reasonable.

The first suggestion was to evaluate $e1$ to a Lam. Why does it have to be a Lam? Well, evaluation produces a value. We have two kinds of values so far: numbers and Lams. Applying a number as a function doesn't make much sense, so it's got to be a Lam. Also, requiring $e1$ to evaluate to a Lam corresponds to Eval-add, where the expressions being added have to evaluate to Nums.

$$\frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad ??}{(\mathsf{App}\ e1\ e2) \Downarrow ??}\ \text{?Eval-app}$$

I can't think of any other way of writing this first premise. This is good progress, so I dropped one of the ?'s from the name of the rule.

Now we come to a "fork in the road". The choice we make now will decide what *evaluation strategy* this language has. This is usually considered an important and fundamental language design choice, especially for functional languages (like Racket and this Fun language).

4.1.1 The "expression strategy"

The next suggestion during lecture was to substitute $e2$ for x, like this:[2]

$$\frac{e1 \Downarrow (\mathsf{Lam}\ x\ eB) \qquad [e2/x]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\ \text{Eval-app-expr}$$

We'll call this the *expression strategy*[3], because we're taking the expression $e2$—the argument being passed to $(\mathsf{Lam}\ x\ eB)$—and substituting it for x, *without* evaluating $e2$.

[2] During lecture, I called the result of evaluation $v2$ rather than v. It will be less confusing to call it v.

[3] There is a more traditional name, which we'll talk about in due course.

Let's look at some examples.

- **Evaluating an application of the identity function.**

  Suppose we apply the identity function $(\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x}))$ to some simple expression, like $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$.

  (It would be even simpler to apply the identity function to a Num, but that wouldn't illustrate what I'm trying to illustrate.)

  So we need to derive

  $$\big(\mathsf{App}\ (\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x}))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow \cdots$$

  First we "match" expressions to meta-variables in Eval-app-expr:

  $$\big(\mathsf{App}\ \underbrace{(\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x}))}_{e1}\ \underbrace{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))}_{e2}\big) \Downarrow \cdots$$

  Then we plug in those expressions:

  $$\frac{(\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x})) \Downarrow (\mathsf{Lam}\ \mathrm{x}\ e\mathrm{B}) \qquad \big[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/\mathrm{x}\big]e\mathrm{B} \Downarrow v}{\big(\mathsf{App}\ (\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x}))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow v}\ \text{Eval-app-expr}$$

  We need to be careful about Gentzen's notation. We *want* to apply the rule Eval-app-expr, but we haven't really applied it yet, because we haven't derived its two premises. A rule is an "if…then …" statement; you don't get to conclude the "then" part without showing that the "if" holds. What I wrote above is based on the power of hope: I *want* to derive $\big(\mathsf{App}\ \cdots\ \cdots\big) \Downarrow v$, so I'm going to *try* to apply Eval-app-expr. To apply Eval-app-expr, I need to derive each premise, using the same method of hope.

  At each step in this process, any leaf in my derivation tree that doesn't have a horizontal line over it is a *goal* that remains to be proved.[4]

  If I can get a derivation tree whose leaves all have horizontal lines over them, I will know that I have derived $\big(\mathsf{App}\ \cdots\ \cdots\big) \Downarrow v$, but not before.

  We now want to derive

  $$(\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x})) \Downarrow (\mathsf{Lam}\ \mathrm{x}\ e\mathrm{B}) \qquad \text{(first goal)}$$

  and

  $$\big[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/\mathrm{x}\big]e\mathrm{B} \Downarrow v \qquad \text{(second goal)}$$

  For the first evaluation, we have a rule that evaluates Lams: Eval-lam. Plugging in for the meta-variable $e1$ in that rule, we get

  $$\frac{}{(\mathsf{Lam}\ \mathrm{x}\ (\mathsf{Id}\ \mathrm{x})) \Downarrow (\mathsf{Lam}\ \mathrm{x}\ \underbrace{(\mathsf{Id}\ \mathrm{x})}_{e\mathrm{B}})}$$

[4] If you have used a logic programming language, such as Prolog (taught in CPSC 312), yes, there is a connection; we may not have time to explore it in 311, though.

Note that this tells us what eB is, so we can revise our second goal by plugging in $(\mathsf{Id}\ x)$ for eB:

$$\big[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x\big]\,(\mathsf{Id}\ x) \Downarrow v \qquad \text{(second goal, revised)}$$

Since *subst* is now applied to "real" arguments (without unknown meta-variables), we can use the definition of *subst*. We really should revise the definition of *subst* to our extended language, but the old version works for this example: replacing all instances of the identifier x in $(\mathsf{Id}\ x)$ with $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$ is just $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$.

$$(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow v \qquad \text{(second goal, revised again)}$$

Exercise 1. Derive this second goal, following the "method of hope" as above. (This is *not* exactly like a similar example for the AE language, because we changed our notion of evaluation to produce expressions—more specifically, values—rather than numbers.) I left some space above for you to write the derivation tree.

I'll assume you've derived the second goal, and to keep track, I'll put a checkmark ✓ above it. I'm also assuming you got $(\mathsf{Num}\ 5)$, so I'm plugging that in for the meta-variable v.

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-lam} \qquad \overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)}}{\big(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ 5)}\ \text{Eval-app-expr}$$

Everything in this tree has either a horizontal line or a checkmark, so we have a complete evaluation derivation.

Remember: Following the method of hope, a derivation tree is **complete** if each leaf of the tree is either (1) an application of a rule with no premises (for example, Eval-num), or (2) a conclusion of some other complete derivation tree.

You can tell (1) because the leaf has a horizontal line with nothing above it.

To keep track of (2), write a checkmark above the leaf.

If a leaf doesn't have a horizontal line or a checkmark, that leaf is a goal that needs to be be derived, and the derivation is **incomplete**.

- **Evaluating an application of the doubling function.**

  In Fun, we can write a function that doubles its argument:

  $$\big(\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))\big)$$

  What happens when we apply this function to $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$? Hopefully, we will get 10, since $2 \cdot (2+3) = 10$. But remember that we are evaluating to expressions now, so we actually want to evaluate to $(\mathsf{Num}\ 10)$.

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\mathsf{Lam}\,x\,(\mathsf{Add}\,(\mathsf{Id}\,x)\,(\mathsf{Id}\,x))\big)}\ \text{Eval-lam} \qquad \big[(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))/x\big](\mathsf{Add}\,(\mathsf{Id}\,x)\,(\mathsf{Id}\,x)) \Downarrow v}{\big(\mathsf{App}\ \big(\mathsf{Lam}\,x\,(\mathsf{Add}\,(\mathsf{Id}\,x)\,(\mathsf{Id}\,x))\big)\ \big(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)\big)\big) \Downarrow v}\ \text{Eval-app-expr}$$

To save space, I wrote "…", but since Eval-lam has the same thing on both sides of the "⇓", it has to be $\big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)$.

The second premise of Eval-app-expr has neither a horizontal line above nor a checkmark, so we have an incomplete derivation. To figure out which rule to try, we need to know what expression we have, so we look up the definition of *subst*. That gives:

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)}\text{Eval-lam} \qquad \big(\mathsf{Add}\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow v}{\big(\mathsf{App}\ \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow v}\text{Eval-app-expr}$$

We now know exactly what expression we have, so we can try to apply a rule. We have an Add, so we'll try Eval-add.

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)}\text{Eval-lam} \qquad \dfrac{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ \_\_\_) \qquad (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ \_\_\_)}{\big(\mathsf{Add}\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ \_\_\_+\_\_\_)}\text{Eval-add}}{\big(\mathsf{App}\ \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ \_\_\_+\_\_\_)}\text{Eval-app-expr}$$

Conveniently, you already did a complete derivation for $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$ and found that this expression evaluated to $(\mathsf{Num}\ 5)$, so we can fill in all of the blanks with 5.

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)}\text{Eval-lam} \qquad \dfrac{\overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)} \qquad \overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)}}{\big(\mathsf{Add}\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ 5+5)}\text{Eval-add}}{\big(\mathsf{App}\ \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ 5+5)}\text{Eval-app-expr}$$

Finally, we replace $5 + 5$ with 10 in the derivation tree.

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)}\text{Eval-lam} \qquad \dfrac{\overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)} \qquad \overset{\checkmark}{(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)) \Downarrow (\mathsf{Num}\ 5)}}{\big(\mathsf{Add}\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ 10)}\text{Eval-add}}{\big(\mathsf{App}\ \big(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x}))\big)\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))\big) \Downarrow (\mathsf{Num}\ 10)}\text{Eval-app-expr}$$

Notice that in last example, we ended up with *two* copies of the same evaluation derivation (the one you did as an exercise). This is because we had two instances of x in the body of the function being applied, and our rule Eval-app-expr substitutes the entire expression $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$.

Does that really matter, except for making the derivation tree bigger? Well, maybe. If the evaluation semantics seems to be doing work twice, an interpreter based on the semantics will *also* do that work twice! That doesn't matter much for this small example, but imagine if, instead of the argument being $(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))$, it were some expression that computed the sum of the first million prime numbers. Our interpreter would evaluate that huge expression twice, even though it only appears once in the input expression $(\mathsf{App}\ \cdots\ \cdots)$.

4.2 The "value strategy"

An alternative strategy, which we'll call the *value strategy*, is to have this rule instead of Eval-app-expr:

$$\dfrac{e1 \Downarrow (\mathsf{Lam}\ \mathsf{x}\ eB) \qquad e2 \Downarrow v2 \qquad \big[v2/\mathsf{x}\big]eB \Downarrow v}{(\mathsf{App}\ e1\ e2) \Downarrow v}\text{Eval-app-value}$$

The first premise is the same as Eval-app-expr, but a new second premise evaluates *e2 immediately*, and in a third premise, we substitute the *result* of that evaluation for x.

We can see the difference from Eval-app-expr by evaluating the same Fun expression we evaluated just above, the doubling function applied to (Add (Num 2) (Num 3)). The expression still evaluates to (Num 10), but the derivation looks rather different:

$$\dfrac{\dfrac{}{\ldots \Downarrow (\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})))}\text{Eval-lam} \quad (\mathsf{Add}\ (\mathsf{Num}\ 2)\ \overset{\checkmark}{(\mathsf{Num}\ 3)}) \Downarrow \overbrace{(\mathsf{Num}\ 5)}^{v2} \quad (\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5)) \Downarrow (\mathsf{Num}\ 10)}{(\mathsf{App}\ (\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})))\ (\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3))) \Downarrow (\mathsf{Num}\ 10)}\text{Eval-app-value}$$

4.3 Other possible rules

Based on a suggestion during lecture (in 2016), another possible rule would be

$$\dfrac{e1 \Downarrow (\mathsf{Lam}\ \mathsf{x}\ e\mathsf{B}) \qquad e2 \Downarrow v2}{(\mathsf{App}\ e1\ e2) \Downarrow [v2/\mathsf{x}]e\mathsf{B}}\ \text{Eval-app-alternate}$$

In this rule, the substitution $[v2/\mathsf{x}]$ happens in the conclusion, not in a premise.

If the body $e\mathsf{B}$ of the Lam being applied is simple enough, this rule works in the same way as Eval-app-value. But for a function like $(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})))$, we would derive

$$\dfrac{e1 \Downarrow \underbrace{(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})}\ \ (\mathsf{Num}\ 5) \Downarrow (\mathsf{Num}\ 5)}{(\mathsf{App}\ \underbrace{(\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})))}_{e1}\ \underbrace{(\mathsf{Num}\ 5)}_{e2}) \Downarrow \underbrace{(\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))}_{[v2/\mathsf{x}]e\mathsf{B}}}\ \text{Eval-app-alternate}$$

(I left out the details of evaluating the premises.) Here the result of evaluation is $[v2/\mathsf{x}]e\mathsf{B} = [(\mathsf{Num}\ 5)/\mathsf{x}](\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})) = (\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))$, which is neither a Num nor a Lam; to get the expected answer, (Num 10), we would need to evaluate (Add (Num 5) (Num 5)). The rule gets us closer to the answer, but not all the way, and evaluation semantics is supposed to give us a final answer.

It's possible to design a system of rules that takes smaller steps from an expression to a value. In that kind of semantics, called a "small-step" semantics, we might say that

$$(\mathsf{App}\ (\mathsf{Lam}\ \mathsf{x}\ (\mathsf{Add}\ (\mathsf{Id}\ \mathsf{x})\ (\mathsf{Id}\ \mathsf{x})))\ (\mathsf{Num}\ 5)) \longrightarrow (\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5))$$

and that

$$(\mathsf{Add}\ (\mathsf{Num}\ 5)\ (\mathsf{Num}\ 5)) \longrightarrow (\mathsf{Num}\ 5)$$

Such a semantics has several nice features, including being "closer to the machine" (in fact, assembly language can be modelled using a small-step semantics) and allowing us to specify the behaviour of programs that don't terminate. We will use a small-step semantics for some later parts of 311. However, the rules of evaluation semantics (or "big-step semantics") are easier to implement. Since the first part of 311 emphasizes the connection between rules, which specify language behaviour, and interpreters that implement that behaviour, we are using the easier-to-implement style—evaluation semantics—rather than small-step semantics.

4.4 Advantages and disadvantages

Think about the "expression strategy" and the "value strategy". The value strategy gave us a smaller derivation for one of our examples—and would also give us a faster interpreter for that example.

- Both strategies seem to be giving us the same answers—evaluation is giving us the same values. Is that true in general?
  - There are different ways to read "in general". For our language[5], we will indeed get the same value regardless of which strategy we use, for any expression. (Caveat: I haven't proved this.) The size of the derivations, and the amount of time the interpreter will take, may be very different, but we'll get the same value (either a Num or a Lam).
  - If we mean languages generally, there are languages where this doesn't hold. In a language with *effects*, the argument *e2* might do something that allows us to distinguish the two evaluation strategies. For example, *e2* might print a string, and a different number of strings would be printed depending on the evaluation strategy. The values returned wouldn't change, however: either you print once, and return a value, or you print twice and return the same value.

    We *could* get different values, however, if our language had *mutable state*, such as an incrementable counter. If *e2* increments this counter, the contents of the counter might affect the value returned.

    Languages with mutable state have more complicated semantics, which we'll look at later on.

- Are there any Fun expressions where the *expression* strategy (Eval-app-expr) would be faster?
  - Yes—expressions that apply a function that doesn't use its argument:

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Num}\ 4)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))}\ \text{Eval-lam} \qquad \big[\big(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)\big)/x\big](\mathsf{Num}\ 4) \Downarrow v}{\big(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))\ \big(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)\big)\big) \Downarrow v}\ \text{Eval-app-expr}$$

    Since x doesn't occur in (Num 4), the result of *subst* on (Num 4) is just (Num 4):

$$\dfrac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Num}\ 4)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))}\ \text{Eval-lam} \qquad (\mathsf{Num}\ 4) \Downarrow v}{\big(\mathsf{App}\ (\mathsf{Lam}\ x\ (\mathsf{Num}\ 4))\ \underbrace{\big(\mathsf{Add}\ (\mathsf{Num}\ 2)\ (\mathsf{Num}\ 3)\big)}_{e2}\big) \Downarrow v}\ \text{Eval-app-expr}$$

    Here, we never evaluate the argument *e2* at all! The value strategy would evaluate *e2* even though it's not needed.
  - There's another evaluation strategy, *lazy evaluation*, which doesn't evaluate the argument *e2* until it's used inside the Lam. At that time, it evaluates *e2* and remembers the value it gets. Other instances of x will reuse that value instead of evaluating *e2* again. This strategy is a little more complicated to define, but we'll come back to it later in 311.

[5] Or rather, our languages: a language is syntax and semantics together, so we should really talk about two languages: the "Fun with Eval-app-expr" language, and the "Fun with Eval-app-value" language.

$$\frac{}{(\text{Num } n) \Downarrow (\text{Num } n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Add } e1\ e2) \Downarrow (\text{Num } (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\text{Num } n_1) \qquad e2 \Downarrow (\text{Num } n_2)}{(\text{Sub } e1\ e2) \Downarrow (\text{Num } (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\text{Let } x\ e1\ e2) \Downarrow v2}\ \text{Eval-let} \qquad \frac{}{(\text{Id } x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\text{Lam } x\ e1) \Downarrow (\text{Lam } x\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\text{Lam } x\ eB) \qquad e2 \Downarrow v2 \qquad [v2/x]eB \Downarrow v}{(\text{App } e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

Figure 3 Evaluation rules for Fun

- Does evaluation in Racket work like the expression strategy, or like the value strategy? How about Java? Haskell? Algol-60?
  - What Racket does isn't exactly the same as either of our evaluation rules, but it's very close to the value strategy.
  - Java: same answer. (In Java, and similar languages, you often pass *pointers* or *references* around, but those are really just values, albeit of a different kind than the values in Fun.)
  - Haskell uses lazy evaluation (see above), so it's kind of like the expression strategy.
  - Algol-60 supports *both* the value strategy and the expression strategy. The expression strategy is the default, but programmers can designate specific function arguments as following the value strategy. (The Algol-60 committee had *just invented the expression strategy*. Years later, several committee members were still angry that the report's editor, Peter Naur—also the 'N' in 'BNF'—decided, on his own, to make the expression strategy be the default.)

5 From the Fun rules to a Fun interpreter

Earlier, we said that an interpreter should do this:

"Given an e, find a number n such that $e \Downarrow n$."

For Fun, we have a more general idea of the result of evaluation that includes functions as well as numbers, and we said that functions (Lam $\cdots$) and numbers (Num $\cdots$) are collectively *values*, so a Fun interpreter should do this instead:

"Given an e, find a value v such that $e \Downarrow v$."

(Live-coding time...)

6 Collected rules for Fun

Figure 3 collects all the rules, showing Eval-app-value rather than Eval-app-expr.

7 Fly first-class, for free

If we wrote our interpreter correctly, we now have a programming language that is quite similar to the λ-calculus (which was invented by Alonzo Church in the 1930s, and extensively studied ever since). As a programming language, the λ-calculus has very few features (it doesn't even do arithmetic. . . at least not in a way that you'd recognize), but it does have functions that are *first-class*—functions that can take other functions as arguments, and return functions. You may not have noticed it, but our rules have no trouble with first-class functions.

7.1 It's not just you

First-class functions are often considered a strange and advanced language feature. Back in 1967, Christopher Strachey (who worked on the semantics of programming languages, and also—rather curiously—set in motion a chain of events that led to C) pointed out that mathematicians rarely treated functions as "first-class" and hadn't even agreed on a notation for first-class functions; mathematicians seemed to have little grasp of how to use functions as values.

Today, the DrRacket "Beginning Student" language doesn't allow functions as arguments, and it doesn't allow a function to return a function. The "Intermediate Student" language allows functions as arguments, but not as results. The "Intermediate Student with Lambda" language adds the ability to return a function (by returning a Lambda).

This distinction may be useful when learning how to program; I'm not sure it's useful when learning how to think about defining programming languages. **Functions are values;** until they are applied, they don't do anything, just as numbers don't do anything until you do arithmetic on them.

Unfortunately, most "real" programming languages make that hard to remember. For example, DrRacket claims that the result of evaluating (**lambda** (x)x) is "#<procedure>". But that's a bad, secretive notation for the function you entered; you should think of it as being (**lambda** (x)x). Two other functional languages, SML and OCaml, also refuse to show you the inside of a function.

```
Standard ML of New Jersey v110.72 [built: Tue Jan 11 13:30:58 2011]
- 2 + 2;
val it = 4 : int
- (fn x => x + x) 2;
val it = 4 : int
- (fn x => x + x);
val it = fn : int -> int
```

```
        OCaml version 4.00.1

# 2 + 2;;
- : int = 4
# (fun x -> x + x) 2;;
- : int = 4
# (fun x -> x + x);;
- : int -> int = <fun>
```

Two different Haskell implementations, hugs and ghci, not only won't show it, but print mysterious error messages, just to taunt you.

```
__   __ __  __  ____   ___      ____________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ____________________________________________

Hugs> 2 + 2
4
Hugs> (\x -> x + x) 2
4
Hugs> (\x -> x + x)
ERROR - Cannot find "show" function for:
*** Expression : \x -> x + x
*** Of type    : Integer -> Integer


GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Prelude> 2 + 2
4
Prelude> (\x -> x + x) 2
4
Prelude> (\x -> x + x)

<interactive>:4:1:
    No instance for (Show (a0 -> a0)) arising from a use of ~print~
    In a stmt of an interactive GHCi command: print it
Prelude>
```

Python displays the function along with its address in memory:

```
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
>>> 2 + 2
4
>>> (lambda x: x + x) (2)
4
>>> (lambda x: x + x)
<function <lambda> at 0x1023d3938>
```

Why is this? I'm not sure. It kind of makes sense for a compiler to throw away abstract syntax, but Hugs isn't even a compiler. The point of a REPL (read-eval-print loop; also known in DrRacket as "Interactions", and in many other languages as a "toplevel") is not to be efficient, but to allow "playing" with a language by typing in expressions and seeing what happens. It doesn't seem that difficult for something like SML to preserve the abstract syntax of code, at least code that you enter in the REPL. But I haven't implemented it myself, so there are probably issues I haven't thought of.

7.2 Looking behind the curtain

The distinction between first-class functions and "lesser" functions may matter, however, when we try to write *efficient* interpreters and compilers. But not worrying about efficiency is helpful now: Because our interpreter follows a (relatively) very simple evaluation semantics, you can get an idea of how first-class functions work *in general* by writing Fun expressions that evaluate to functions, and *looking at the functions*—our Fun language always shows you what's inside a Lam! Then you can take that *general* idea and use it when programming in Racket, OCaml, or Haskell.

This may not be too effective yet, because our Fun language is so small, but you'll add several features to it in the next assignment.

7.3 Unparsing

A parse function takes concrete syntax (for us, an S-expression) and builds abstract syntax. An "unparse" function takes the abstract syntax, and turns it back into concrete syntax.

I'm doing this now so that when you look at the Lams your Fun expressions evaluate to, you can read them more easily.

For convenience, I'm using quasiquote and unquote. (A fun (?) exercise: write a version of unparse that *doesn't* use quasiquote and unquote.) The file dynsem-fun.rkt has some useful links about these features.

7.4 Digression: equality of functions

When are functions equal? Not an easy question!

In Racket, after defining a function using **define**, that function is equal to itself. But if we write identical expressions using **lambda**, they are not equal.

```
> (equal? unparse unparse)
#t
> (equal? (lambda (x) x) (lambda (x) x))
#f
```

Python works similarly.

SML, rather characteristically, just refuses to let you compare functions at all:

```
- (fn x => x) = (fn x => x);
stdIn:1.1-1.26 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) * ('X -> 'X)
  in expression:
    (fn x => x) = (fn x => x)
```

Its complaint is that functions don't have "equality type"; they don't have a type for which SML defines equality. The principle here is that the answer to whether two functions are equal is most likely useless—it would be reasonable to expect that `(lambda (x) x)` would be equal to itself, but it's not, so SML just doesn't define equality on functions at all.

OCaml defines two (at least) kinds of equality: = doesn't work for functions (with an exception, rather than a type error), and == works like equal? in Racket: it *might* return true for functions that are the same, but it might just return false.

```
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
# (fun x -> x) == (fun x -> x);;
- : bool = false
# let identity = fun x -> x;;
val identity : 'a -> 'a = <fun>
# identity == identity;;
- : bool = true
```

Neither approach to function equality (not defining it at all, or defining an equality test that often says "no, they're not equal" even for functions with identical source code) is totally satisfying. A mathematician's answer to the question, "Are the function $f(x) = x + 1$ and the function $g(y) = y + 2 - 1$ the same?" would be (I think—I'm not really a mathematician) "yes", because both functions are *extensionally equal*: given the same arguments, they produce the same results.

In general, the question of whether two functions are extensionally equal is undecidable, and certainly difficult: imagine that the bodies of the functions `f` and `g` above were each 100,000 lines long. An interesting approach would be to implement a version of function equality that has *three* possible answers: "these functions are obviously extensionally equal", "these functions are obviously extensionally not equal", and "I don't know". I don't know if anyone has tried this approach.