LAMBDA CALCULUS

(untyped)

What is computation?

A representation of a value, which we can simplify to its simplest ("normal") form.

- We need expressions that can be simplified
- We need functions to pass around these expressions
- We need variables (parameters to functions)

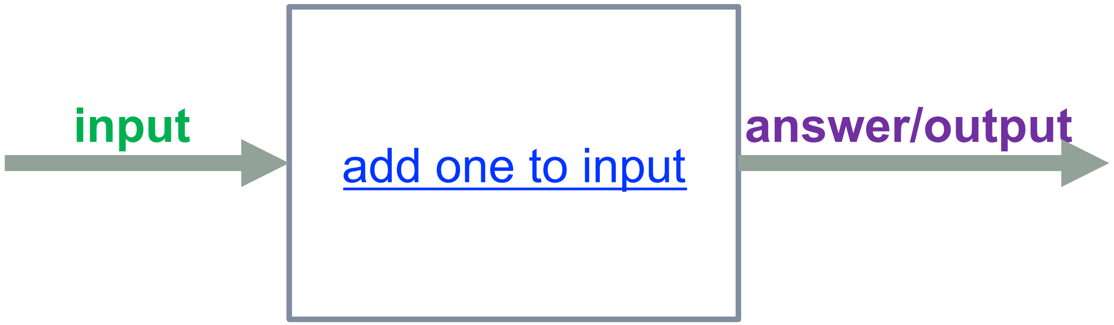
Let's name the input x, and represent this functionality:

λ x . x + 1

A function,

With parameter x,

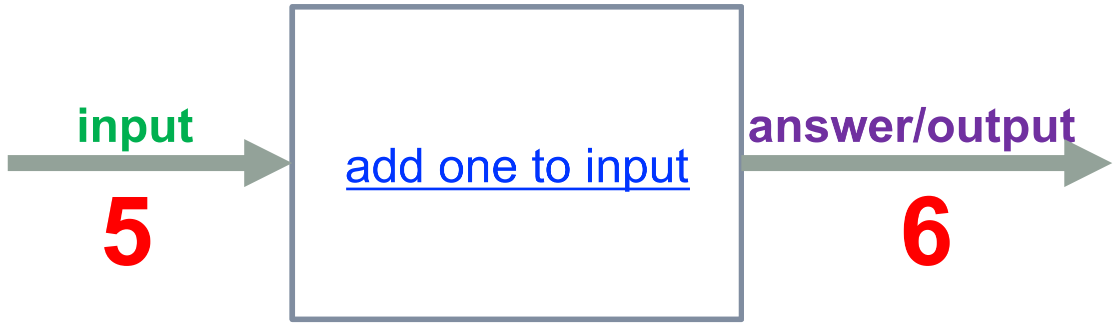
That will calculate (x+1)



What is computation?

A representation of a value, which we can simplify to its simplest ("normal") form.

- We need expressions that can be simplified
- We need functions to pass around these expressions
- We need variables (parameters to functions)



Let's name the input x, and represent this functionality:

λ x . x + 1

Let's feed 5 to the function:

((λx.x+1) 5)

Simplify ("solve"/"evaluate"):

⟶ **5+1**

⟶ **6**

RoadMap

- We want to understand the **idea of evaluation**, *based directly on the evaluation rules*

- We learn how to **extend the language**: add terms, add evaluation rules, play with the outcome until it behaves how we want
- Eventually, the language is ready to use – we **encode** expressions and evaluate them.
  - Writing encodings is just … coding! You write expressions in the language.
  - Anything you need that's not an extension (part of the language) can be implemented.

- We can explore what **no-extensions** feels like later on.
- "Church booleans", encoding libraries of functionality
- We can explore **weirder extensions** like recursion.
- Quick introductory video: https://www.youtube.com/watch?v=eis11j_iGMs (1st 7 minutes)

The Untyped Lambda Calculus (λ)

Designed by Alonzo Church (1930s)

- Turing Complete *(Turing was his doctoral student. Small world!)*
- **Models functions**, always as 1-input

- Definition: terms, values, and evaluation
  - **t ::= x | λ x . t | t t** ← ***adjacent terms are an application (t t)***
  - **v ::= λ x . t**

- Notes
  - terms **t** are variables, lambdas, or applications
  - only lambdas are values.
  - this language is untyped!

Our chosen rules:

- Must simplify the function down to a value before simplifying the argument
- Can recursively explore subterms to find simplifications, with some caveats
- Can only feed values to functions (eval argument before calling)

λ : Evaluation Semantics

Evaluation: applying these rules to simplify your term until you have a value (no more evaluation possible).

→ *is* evaluation; these rules define the functionality from input to output.

$$t ::= x \mid \lambda x . t \mid (t\ t)$$
$$v ::= \lambda x . t$$

- **E-App1** $$\frac{t_1 \to t_1'}{(t_1\ t_2) \to (t_1'\ t_2)}$$
- **E-App2** $$\frac{t \to t'}{(v\ t) \to (v\ t')}$$
- **E-App-Abs** $$\frac{}{((\lambda x . t)\ v) \to t[x \mapsto v]}$$

E-App1: "If I know that a term t_1 can be evaluated to t_1', then when I've got an application (t_1 t_2), I can evaluate just t_1 to t_1', and then I've got (t_1' t_2) left over."

E-App2: "When an application has a value first and a term second, and that term can evaluate further, we are allowed to evaluate that second term."

E-App-Abs: "When a lambda term is applied to a value v, it can simplify down to the body of the lambda, t, with all occurrences of the lambda parameter x being replaced with the argument v. We don't need any simplifications available (nothing above the line)."

Confusing at first? That's ok! This is like the DNA of evaluation; a lot is packed into just a few rules.

λ : Evaluation Semantics

$\rightarrow$ *is* evaluation; these rules define the functionality from input to output.

How to read these rules:

- If you have this,
- And can show that this can be done (often using more evaluation recursively),
- Then what you started with can evaluate to this.

- **E-App1** $$\frac{t_1 \rightarrow t_1'}{(t_1\ t_2) \rightarrow (t_1'\ t_2)}$$

Pattern Matching:

- When we need a term, we use placeholders with names like t_1, t_2, t_1', etc. to represent any valid term.
  - when we see the same name (e.g. t_1 again), it must be the same term.
  - If we need a specific term, we write it out – e.g. "λx.x", 5, true, and so on.
- Parentheses represent application. For convenience, adjacency is also used.
  - (((f a) b) c) is the same as (f a b c)

Better with Extensions!

- Though not required, some extensions really help understand how to *use* the lambda calculus.

t ::= x | λx.t | (t t) | t + t | t – t | t * t | t > t | t < t | <ℤ's> | true | false | if t t t | ~t

v ::= λx.t | <ℤ's> | true | false

E-Add1 $\dfrac{t_1 \rightarrow t_1'}{t_1 + t_2 \ \rightarrow \ t_1' + t_2}$ **E-Add2** $\dfrac{t_2 \rightarrow t_2'}{v + t_2 \ \rightarrow \ v + t_2'}$ **E-Add** $\dfrac{}{v_1 + v_2 \ \rightarrow (perform\ addition)}$

E-GT1 $\dfrac{t_1 \rightarrow t_1'}{t_1 > t_2 \ \rightarrow \ t_1' > t_2}$ **E-GT2** $\dfrac{t_2 \rightarrow t_2'}{v > t_2 \ \rightarrow \ v > t_2'}$ **E-GT** $\dfrac{}{v_1 > v_2 \ \rightarrow (check\ relation)}$

E-Mul/1/2, E-Sub/1/2, E-LT/1/2 follow the same patterns as E-Add/1/2, E-GT/1/2.

E-If $\dfrac{t_1 \rightarrow t_1'}{if\ t_1\ t_2\ t_3 \rightarrow if\ t_1'\ t_2\ t_3}$ **E-If-true** $\dfrac{}{if\ true\ t_2\ t_3 \rightarrow t_2}$ **E-If-false** $\dfrac{}{if\ false\ t_2\ t_3 \rightarrow t_3}$

E-Neg1 $\dfrac{t \rightarrow t'}{\sim t \rightarrow \sim t'}$ **E-Neg-T** $\dfrac{}{\sim true \ \rightarrow \ false}$ **E-Neg-F** $\dfrac{}{\sim false \ \rightarrow \ true}$

Sample Expressions

- Consider each expression. Are they already values? If not, show each reduction, and name the rule used.

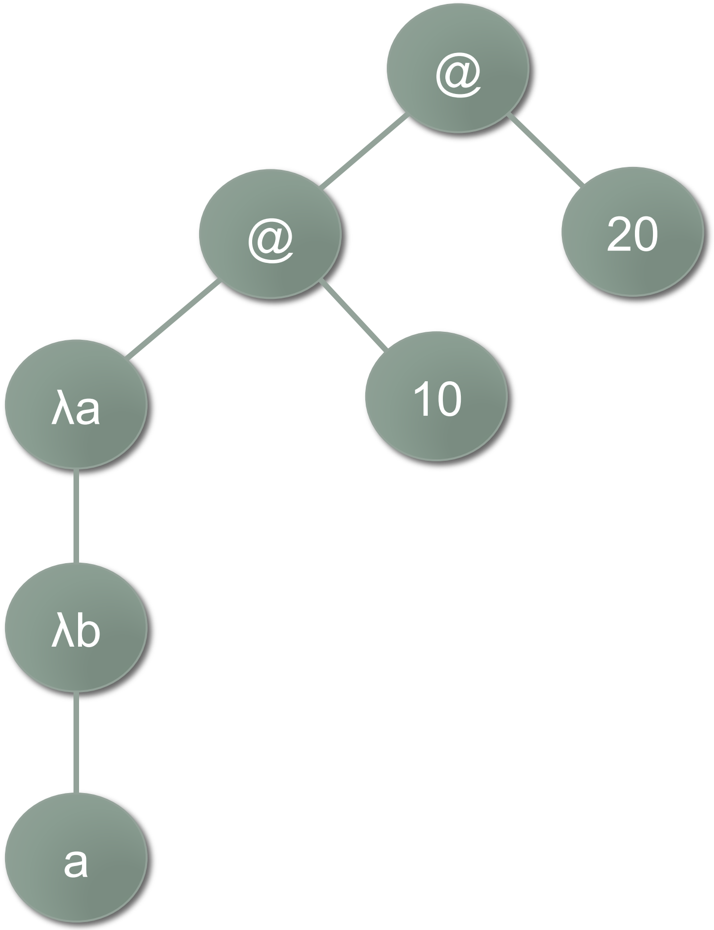
- ((λ x . x + 1) 3)
- (λ e . e + 1)
- ((λ z . z * z) 5)

- (((λa . (λb . a)) 10) 20)
- ((λx . x-(1+2)) (3*4)) ← *why is there always only one next possible step?*
- ((λa . a*a) ((λx . x + 1) 6))
- (((λx . (λy . (x – y) + 1)) 10) 6)
- ((λx . (λy . (x – y) + 1)) 10)

Think in Trees

- Drawing out an abstract syntax tree for terms can help understand them:

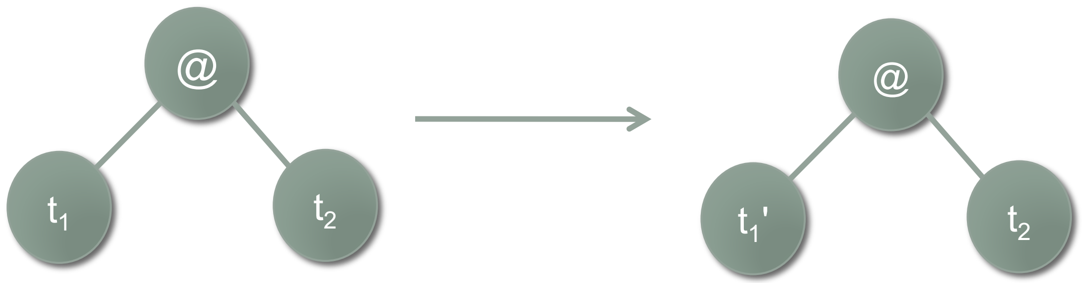
(((λa . (λb . a)) 10) 20)



λ : Evaluation Rules as Trees

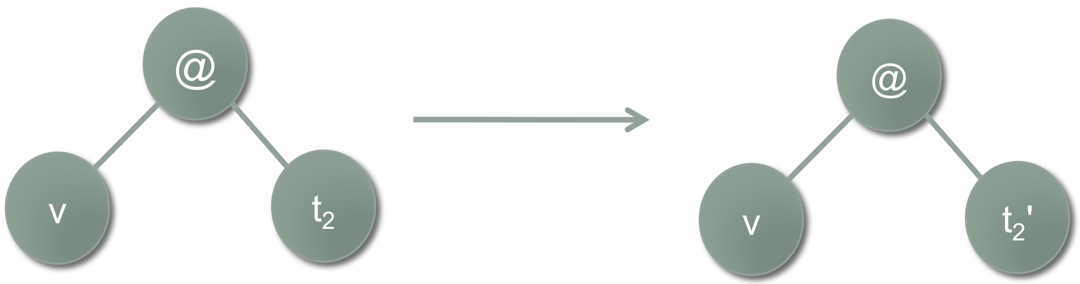
E-App1

$t_1 \rightarrow t_1'$



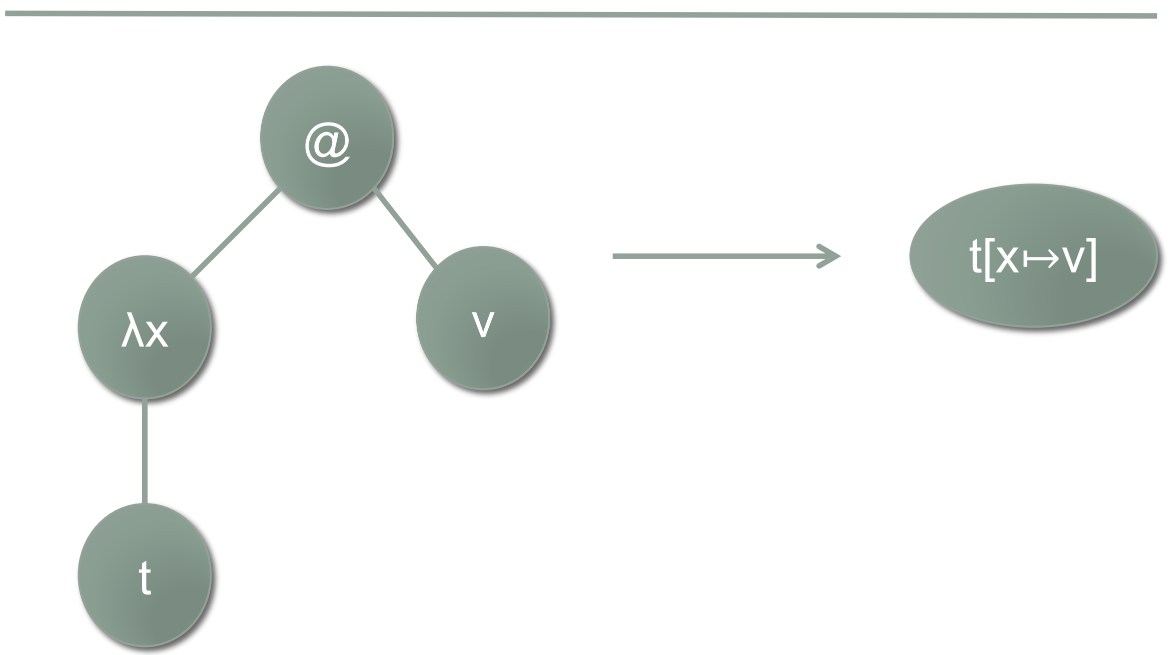
E-App2

$t_2 \rightarrow t_2'$



λ : Evaluation Rules as Trees

E-App-Abs



Reading Expressions

- Too many parentheses is a pain! How can we omit some?
- Parentheses around an application are very common, but not strictly needed: $(t_1\ t_2)$
- the body of a lambda expression grabs as much as it can (it reads through until it hits a close-parenthesis it didn't open, or the end of the expression).
- We can remove some parentheses for slightly easier reading.

These are equivalent:

- (λx. (λy. x+y))
- λx. λy. x+y

These are not:

- ((λx. x+1) 3)
- (λx. x+1   3)

Practice: Draw Trees

Draw the trees for these terms.

- λx. x+1
- λf. λ x. f x *same as: (λf. (λx. (f x)))*
- λa. ((λb.b) a)
- λx. λy. if (x<y) x 0

Might want to look at our "full language" extensions for these last few terms.

- cons 1 (cons 2 (cons 3 nil))
- λxs . if (isnil xs) 0 (head xs)
- λself . \n . if (n=1) 1 (self (n-1))

Extending λ

- we will add more terms and values, to have more primitives in our language.
- We then also add more evaluation rules that put those new terms to use.
- The core lambda calculus is actually pretty painful/nearly useless on its own (we already sneaked numbers/ops. in!)

Extending evaluation: Booleans

Adding Booleans

t ::= … | true | false | if t t t
v ::= … | true | false

so, we also still have:

t ::= x | λx.t | t t …
v ::= λx.t …

- **E-if**

$$\frac{t_1 \to t_1'}{(\text{if } t_1\ t_2\ t_3) \to (\text{if } t_1'\ t_2\ t_3)}$$

- **E-if-true**

$$\frac{}{(\text{if true } t_2\ t_3) \to t_2}$$

E-if-false

$$\frac{}{(\text{if false } t_2\ t_3) \to t_3}$$

Extension: Natural Numbers

We borrow integers from the void (just kidding – from your years and years of mathematics), and then define some operators. (Here, $\mathbb{Z}$ means all integer values)

Adding Naturals

$$t ::= \ldots \mid \mathbb{Z} \mid t + t \mid t - t \mid t * t$$

$$v ::= \ldots \mid \mathbb{Z}$$

E-Add-1

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2}$$

E-Add-2

$$\frac{t_2 \rightarrow t_2'}{v + t_2 \rightarrow v + t_2'}$$

E-Add

$$\frac{}{v1 + v2 \rightarrow \text{<perform addition>}}$$

Same idea for subtraction and multiplication

Sample Expressions – Bools and Ints

- Consider each expression. Are they already values?
  - If not, show each evaluation step down to a value, and name the rules used.

- if true 5 10
- if false true 20 ← *what's weird about this one?*

- λ x . if x 3 6
- (6+5)*(4-3)
- ((λ x . if x 4 7) false)
- ((λ z . if true z (9+z)) 5)

Alternate Extension: Natural Numbers

This is a far more manual approach than relying on $\mathbb{Z}$: we create our own numbers as **zero**, and **successor** of a number. Adding operations such as predecessor of a number, addition, etc. would be an even further set of term extensions and evaluation rules.

Previously, we just used integers from mathematics "off the shelf", and addition's eval rule just says "do the addition". This is like using **addq** instead of re-implementing addition in a compiler, but we can do it manually if for some reason we want/need…

Extending evaluation: Naturals

Adding Naturals Manually
t ::= … | zero | succ t | pred t
v ::= … | zero | succ v

- **E-succ** $$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'}$$ **E-pred** $$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'}$$
- **E-pred-succ** $$\frac{}{\text{pred(succ t)} \rightarrow \text{t}}$$

Extending evaluation: Pairs

Adding Pairs

t ::= … | pair t t | fst t | snd t
v ::= … | pair v v

E-pair1
$$\frac{t_1 \to t_1'}{\text{pair } t_1\ t_2 \to \text{pair } t_1'\ t_2}$$

E-fst
$$\frac{t \to t'}{\text{fst } t \to \text{fst } t'}$$

E-pair2
$$\frac{t_2 \to t_2'}{\text{pair } v\ t_2 \to \text{pair } v\ t_2'}$$

E-snd
$$\frac{t \to t'}{\text{snd } t \to \text{snd } t'}$$

E-pair-fst
$$\frac{}{\text{fst } (\text{pair } t_1\ t_2) \to t_1}$$

E-pair-snd
$$\frac{}{\text{snd } (\text{pair } t_1\ t_2) \to t_2}$$

Practice: simplifying pair terms

Reduce each to a value. Name the rules used.
(if it can't get to a value, state "no normal form").

- pair (2+4) 6
- fst (pair 2 4)
- fst 5

- [(λp. (fst p) + (snd p))   (pair 2 3)]
- [(λp. if (fst p) (snd p) 0) (pair true 7)]
- if (pair true false) (pair 2 3) (pair 4 5)

Side-topic:

Encoding instead of Extending the language

Encoding (Using Definitions)

The language itself shouldn't be extended with every single calculation you want to do as native terms; we can also just build expressions and use them.

- ***We'll name them for convenience, but the names are only shorthand, not a lang. feature!***

- definition: **not** = λ x . if x false true
- definition: **and** = λ a . (λ b . if a b false)

Simplify these expressions.

- (not true)
- ((λ x . and x true) true) *note, and is just a function!*
- ((λ x . and x true) false) *thus it isn't used infix, as in (x and y)*

Define these:

- or
- nor

Encoding Boolean values/operations

Start again, now with **only the core untyped lambda calculus** – no true/false values, no if-term, no E-Bool rules.

- We just encode everything we wish we had ourselves!

- Encoding 'true' and 'false' as functions:
  - ***true*** = λ x . λ y . x
  - ***false*** = λ x . λ y . y

- Operator encodings
  - **not** = λ a . (a *false*) *true*
  - **and** = λ a . (λ b . ((a b) a))
  - **or** = λ a . (λ b . ((a a) b))
  - **if** = λ b . (λ t . (λ e . ((b t) e)))

note: all ()'s are optional on this slide

evaluate:
- ((and false) true)
- ((or false) true)

The entire language:

t::= x | λx.t | (t t)
v::= λx.t

E-App1
E-App2
E-App-Abs

No if's, and's, or other extensions – we have to encode booleans from only those def'ns!

Implementing the untyped lambda calculus

Investigate implementing the untyped lambda calculus in Haskell:

- **data** for our terms
- **is_value** function to check if a term is a value
  - remember, our values are just a subset of our terms.
- **eval** function to perform evaluation
  - this needs substitution capability (see the **subst** function)

Until we extend our language, it'll seem ungainly – the only values we have are functions!
no booleans, numbers, nothing.

→ see `ULC_bools_nums.hs`, which extends with only bools and nums.

→ see `ULC_full.hs`, which also extends w/recursion and lists, as well as some renaming/equality.
→ see `ULC_core.hs`, which has no extensions but does encodings.

Formal Language extension recipe

- Create more terms **t::= …**
  - both **constructors** and **observers**
    - Constructors represent data(values)
    - Observers represent operations over that data
- add some new terms
  - some that are new values, others that operate on those values
- add some new values **v::= …**
  - probably a subset of the terms you added (the ones that are "answers")
- add more evaluation rules
  - enough that all "proper" terms can become values
  - Probably both of these styles:
    - "make progress on a subterm": see E-App1, E-App2, E-If, etc.
    - "consume a specific structure": see E-App-Ab s, E-If-True, E-If-False, etc.

Coded Language extension recipe

(implementing in Haskell)

- add to datatype Tm (extend terms)
- add cases to is_val (extend the values)
- add cases to eval, subst, etc.

→ language is extended! What other features can we add?

Side topic: Evaluation Strategy

As written, our evaluation rules require that functions' arguments are evaluated first:

- **E-App-Abs**

$$\frac{}{((\lambda\ x\ .\ t)\ \mathbf{v}) \rightarrow t[x \mapsto v]}$$

We could have implemented lazy evaluation (and **removed E-App2**):

- **E-App-Abs-lazy**

$$\frac{}{((\lambda\ x\ .\ t_1)\ \mathbf{t_2}) \rightarrow t_1\ [x \mapsto \mathbf{t_2}]}$$

Evaluation Strategy

How/where does the chosen evaluation strategy affect:

- your implementation?
- Your language usage?

Building an Interpreter

Start with simple core features

- define terms, values, evaluation
- expand with more features

Implementation choice:

- **Domain Specific Language** (DSL)
  - a language designed/dedicated to one task or domain of knowledge
  - write all tools, e.g. parser/compiler
- **Embedded DSL** (EDSL)
  - DSL that is implemented as a library directly in some other language.
  - All of the host's features are directly available: we're actually writing code in the host language that heavily uses the library definitions

→ we're exploring an EDSL.

Choosing a value space

We can have eval target a separate value type instead of `Tm`

choose any type that's already available in the host language, like **Int**.

- every single expression must result in a value of this type!

→ see `ExprLang1.hs` `eval :: Expr -> Int`

We can make our own data type for the value space

→ see `ExprLang2.hs` `eval :: Expr -> Val`

All the packing and unpacking between Expr vs Val is a it annoying though.

Choosing evaluation semantics

Once we include some notion of functions in our code, we can then choose calling conventions.

- how can we introduce functions?
- where do declarations go?
- what kinds of declarations are allowed? (recursive?)

- we can implement any evaluation strategy, such as pass-by-value, pass-by-name, simply by changing our `eval` definition.

Fun diversion

The Ω-combinator always diverges.

$$\Omega = (\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$$

Try performing the application. What do you get?

Providing primitive recursion

We can provide primitive definitions for recursion.

Adding Fix

t ::= … | fix t

E-fix

$$\frac{}{\text{fix } (\lambda\ \text{self}.\ t) \rightarrow t\,[\text{self} \mapsto (\text{fix } (\lambda\ \text{self}\ .\ t))]}$$

Using Recursion

E-fix

$$\frac{}{\text{fix } (\lambda \text{ self. } t) \longrightarrow t\,[\text{self} \mapsto (\text{fix } (\lambda \text{ self . } t))]}$$

- We make a worker function that will be built into the recursive function we want.
  - it needs to be fed a copy of itself as the first argument
  - then, recursive calls can use that argument
  - "fix worker" ties the recursive knot and gives us our recursive function.
    - It's like "loop unrolling", in a functional context
  - Simplification will use the E-Fix rule to supply another layer for each recursive call.

```
worker =   λself. λn.
            if (n = 0)
               true
               (if  (n = 1)
                    false
                    (self (n-2)))
is_even = fix worker
```

Representing Recursion

In the untyped lambda calculus, we can represent recursion directly, or with a language extension. (see `ycomb.txt`)

The y combinator

```
ycomb = \f . ( (\x . (f (\y . x x y)))
               (\x . (f (\y . x x y)))  )

evenF = Lam "self" $ Lam "n"
          $ If (Equal vn (Num 0)) Tru
          $ If (Equal vn (Num 1)) Fls
          $ App (Var "self")
                (Sub vn (Num 2))

iseven = App ycomb evenF
```

personally, I prefer extending the language.

The y-combinator is a real headache to watch in action! Though it is cool...

Extension: Lists

We add primitive support for singly-linked lists

Adding Pairs

t ::= … | cons t t | nil | isnil t | head t | tail t
v ::= … | cons t t | nil

- **E-isnil** $\dfrac{t \to t'}{\text{isnil } t \to \text{isnil } t'}$
- **E-isnil-nil** $\dfrac{}{\text{isnil nil} \to \text{true}}$
- **E-isnil-cons** $\dfrac{}{\text{isnil (cons } t_1\ t_2) \to \text{false}}$
- **E-head1** $\dfrac{t \to t'}{\text{head } t \to \text{head } t'}$ **E-head** $\dfrac{}{\text{head (cons } t_1\ t_2) \to t_1}$
- **E-tail1** $\dfrac{t \to t'}{\text{tail } t \to \text{tail } t'}$ **E-tail** $\dfrac{}{\text{tail (cons } t_1\ t_2) \to t_2}$

Using Lists: Recursion Needed

- In order to write a function that traverses a list, you will need to use recursion. (Either the fix extension or the Y-combinator work fine).

- Example: calculate the length of a list.

```
len = fix (λself.
                λxs. if (isnil xs)
                         0
                         (1 + (self (tail xs))))
```

- Example: calculate the sum of a list of numbers.

```
sum = fix (λself.
                λxs. if (isnil xs)
                         0
                         ((head xs) + (self (tail xs))))
```

Other features

How might we introduce each of the following?

- case statements
- let expressions
- records
- abstract data types
- variable assignments
- classes and objects
- types
- type inference

What else would you want to add to your language?

Valuable resource

To get a much more thorough treatment of writing interpreters for more advanced language features, look for this book:

Types and Programming Languages, by Benjamin Pierce.

→ you can view it electronically through our library's website for free! (VPN/logged in)