LAMBDA CALCULUS

(untyped)
The Untyped Lambda Calculus ($\lambda$)

Designed by Alonzo Church (1930s)
- Turing Complete *(Turing was his doctoral student!)*
- Models functions, always as 1-input

Definition: terms, values, and evaluation
- $t ::= x | \lambda x . t | (t t)$
- $v ::= \lambda x . t$

Notes
- terms $t$ are variables, lambdas, or applications
- only lambdas are values.
- this language is untyped!
Evaluation: applying these rules to simplify your term until you have a value (no more evaluation possible).

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

- **E-App2**: \( \frac{t \rightarrow t'}{(v \; t) \rightarrow (v \; t')} \)

- **E-App-Abs**: \( \frac{((\lambda \; x \; . \; t) \; v) \rightarrow t[x \rightarrow v]}{} \)

**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)."
Sample Expressions

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

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

- \(((\lambda a . (\lambda b . a)) 10) 20\)
- \((\lambda a . a*a) ((\lambda x . x + 1) 6))\)
- \(((\lambda x . (\lambda y . x – y + 1)) 10) 6)\)
- \((\lambda x . (\lambda y . x – y + 1)) 10)\)
Think in Trees

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

$$(((\lambda a \cdot (\lambda b \cdot a)) \, 10) \, 20)$$
\( \lambda : \text{Evaluation Rules as Trees} \)

**E-App1**

\( t_1 \rightarrow t_1' \)

\[
\begin{array}{c}
\text{@} \\
t_1 \\
t_2 \\
\end{array} \rightarrow \\
\begin{array}{c}
\text{@} \\
t_1' \\
t_2 \\
\end{array}
\]

**E-App2**

\( t_2 \rightarrow t_2' \)

\[
\begin{array}{c}
\text{@} \\
v \\
t_2 \\
\end{array} \rightarrow \\
\begin{array}{c}
\text{@} \\
v \\
t_2' \\
\end{array}
\]
$\lambda : \text{Evaluation Rules as Trees}$

$\text{E-App-Abs}$
Reading Expressions

- 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:

- $(\lambda x. (\lambda y. x+y))$
- $\lambda x. \lambda y. x+y$

These are not:

- $((\lambda x. x+1) 3)$
- $(\lambda x. x+1 3)$
Extending λ

- we will add more terms and values, to have more primitives in our language.
- The core lambda calculus is actually pretty painful/nearly useless on its own (we already sneaked numbers/ops in!)

Adding Booleans

| t ::= ... | true | false | if t t t |
| v ::= ... | true | false |
Extending evaluation: Booleans

Add these rules:

• **E-if**

  \[
  \frac{t_1 \rightarrow t_1'}{\text{(if } t_1 \text{ } t_2 \text{ } t_3 \text{)} \rightarrow \text{(if } t_1' \text{ } t_2 \text{ } t_3 \text{)}}
  \]

• **E-if-true**

  \[
  \frac{}{\text{(if true } t_2 \text{ } t_3 \text{)} \rightarrow t_2}
  \]

• **E-if-false**

  \[
  \frac{}{\text{(if false } t_2 \text{ } t_3 \text{)} \rightarrow t_3}
  \]
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 ::= ... | \( \mathbb{Z} \) | t + t | t – t | t * t |
| v ::= ... | \( \mathbb{Z} \) |

**E-Add-1**

\[
\begin{align*}
\text{t}_1 + \text{t}_2 & \rightarrow \text{t}_1' + \text{t}_2 \\
\text{t}_1 & \rightarrow \text{t}_1'
\end{align*}
\]

**E-Add-2**

\[
\begin{align*}
\text{v} + \text{t}_2 & \rightarrow \text{v} + \text{t}_2' \\
\text{t}_2 & \rightarrow \text{t}_2'
\end{align*}
\]

**E-Add**

\[
\begin{align*}
\text{v1} + \text{v2} & \rightarrow \text{<perform addition>}
\end{align*}
\]

Same idea for subtraction and multiplication
Sample Expressions – Bools and Ints

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

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

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

- (((λ a . (λ b . if b 100 200)) (2+3)) (if (a>4) true false))
  - Assume we've added > operator. Uses numbers and booleans!
Using Definitions

We can name expressions and use them.
Using these definitions:

- definition: \(\text{not} = (\lambda x . \text{if } x \text{ false true})\)
- definition: \(\text{and} = (\lambda a . (\lambda b . \text{if } a b \text{ false}))\)

Simplify these expressions.

- \((\text{not true})\)
- \((\lambda x . \text{and } x \text{ true}) \text{ true})\)  
  \(\text{note, and is just a function!}\)
- \((\lambda x . \text{and } x \text{ true}) \text{ false})\)  
  \(\text{thus it isn't used infix (x and y)}\)

Define these:

- or
- nor
Alternate Extension: Natural Numbers

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

<table>
<thead>
<tr>
<th>Adding Naturals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t ::= \ldots \mid \text{zero} \mid \text{succ } t \mid \text{pred } t$</td>
</tr>
<tr>
<td>$v ::= \ldots \mid \text{zero} \mid \text{succ } v$</td>
</tr>
</tbody>
</table>
Extending evaluation: Naturals

- **E-succ**
  \[
  \text{succ } t_1 \rightarrow \text{succ } t_1'
  \]

- **E-pred**
  \[
  \text{pred } t_1 \rightarrow \text{pred } t_1'
  \]

- **E-pred-succ**
  \[
  \text{pred}(\text{succ } t) \rightarrow t
  \]
Extension: Pairs

We add primitive support for paired values.

**Adding Pairs**

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

- **E-pair1**
  - \( t_1 \rightarrow t_1' \)
  - \( \text{pair } t_1 t_2 \rightarrow \text{pair } t_1' t_2 \)

- **E-pair2**
  - \( t_2 \rightarrow t_2' \)
  - \( \text{pair } v t_2 \rightarrow \text{pair } v t_2' \)
Extending evaluation: Pairs

- **E-fst**
  \[ t \rightarrow t' \]
  \[ \text{fst} \ t \rightarrow \text{fst} \ t' \]

- **E-snd**
  \[ t \rightarrow t' \]
  \[ \text{snd} \ t \rightarrow \text{snd} \ t' \]

- **E-pair-fst**
  \[ \text{fst} (\text{pair} \ t_1 \ t_2) \rightarrow t_1 \]

- **E-pair-snd**
  \[ \text{snd} (\text{pair} \ t_1 \ t_2) \rightarrow t_2 \]
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)
Encoding Values

• Without any extensions, we can still represent some simple values.
  • we used numbers and mathematical operators on the previous slide, but those aren't actually in our core language!

• Each value shall be represented as a higher-order function

• Note, we much prefer extending the core language!
Encoding Boolean values/operations

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

- Encoding 'true' and 'false' as functions:
  - true = \lambda x . \lambda y . x
  - false = \lambda x . \lambda y . y

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

*note: all ()'s are optional on this slide*
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
- **eval** function to perform evaluation
  - needs substitution capability (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 includes booleans and numbers.
Formal Language extension recipe

• Create more terms \( t::= \ldots \)
  • both **constructors** and **observers**
    • Constructors represent data(values)
    • Observers represent operations over that data

• add some new terms (if any) to be values \( v::= \ldots \)

• add more evaluation rules
  • enough that all "proper" terms can become values
  • observers will inspect/consume constructors

M. Snyder, George Mason University
Coded Language extension recipe

(implementing in Haskell)

- add to datatype \texttt{Tm} (extend terms)
- add cases to \texttt{is\_val} (extend the values)
- add cases to \texttt{eval}, \texttt{subst}, etc.

→ language is extended! What other features can we add?
Aside: Evaluation Strategy

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

- **E-App-Abs**

  \[ ((\lambda x . t) v) \rightarrow t[x \mapsto v] \]

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

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

  \[ ((\lambda x . t_1) t_2) \rightarrow t_1 [x \mapsto t_2] \]
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 choose any type that's already available in the host language, like \texttt{Int}.
• every single expression must result in a value of this type!

→ see \texttt{ExprLang1.hs} \hspace{1cm} \texttt{eval :: Expr -> Int}

We can make our own data type for the value space

→ see \texttt{ExprLang2.hs} \hspace{1cm} \texttt{eval :: Expr -> Val}
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 $\Omega$-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

\[
\begin{align*}
t & ::= \ldots \mid \text{fix } t
\end{align*}
\]

E-fix

\[
\text{fix } (\lambda \text{self. } t) \rightarrow t [\text{self} \mapsto (\text{fix } (\lambda \text{self} . t))]
\]
Using Recursion

- We make a worker function that, when "fixed", will be the recursive function we want.
  - it needs to be fed a copy of itself as the first argument
  - then, recursive calls call that argument
  - "fix worker" ties the recursive knot and gives us our recursive function.
  - 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`

<table>
<thead>
<tr>
<th>The y combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ycomb = \f . ( (\x . (f (\y . x x y))) (\x . (f (\y . x x y))) )</code></td>
</tr>
<tr>
<td><code>evenF = Lam &quot;self&quot; $ Lam &quot;n&quot; $ If (Equal vn (Num 0)) Tru $ If (Equal vn (Num 1)) Fls $ App (Var &quot;self&quot;) (Sub vn (Num 2))</code></td>
</tr>
<tr>
<td><code>iseven = App ycomb evenF</code></td>
</tr>
</tbody>
</table>

personally, I prefer extending the language. The y-combinator is a real headache to watch in action!
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)