Describing a Programming Language

- Syntax: what do programs look like?
- Semantics: what do programs mean?
  - Operational semantics: how do programs execute step-by-step?

Syntax: What Programs Look Like
Programs are expressions $e$ (also called \( \lambda \)-terms) of one of three kinds:

- **Variable**
  - $x, y, z$

- **Abstraction** (aka nameless function definition)
  - $\lambda x \to e$
  - $x$ is the formal parameter, $e$ is the body
  - “for any $x$ compute $e$”

- **Application** (aka function call)
  - $(e_1 e_2)$
  - $e_1$ is the function, $e_2$ is the argument
  - in your favorite language: $e_1(e_2)$

(Here each of $e$, $e_1$, $e_2$ can itself be a variable, abstraction, or application)
Examples

$$(\mathbf{x} \rightarrow \mathbf{x}) \quad \text{-- The identity function (id) that returns its input}$$

$$(\mathbf{x} \rightarrow (\mathbf{y} \rightarrow \mathbf{y})) \quad \text{-- A function that returns (id)}$$

$$(\mathbf{f} \rightarrow (\mathbf{f} (\mathbf{x} \rightarrow \mathbf{x}))) \quad \text{-- A function that applies its argument to id}$$

QUIZ

Which of the following terms are syntactically incorrect?

A. $$((\mathbf{x} \rightarrow \mathbf{x}) \rightarrow \mathbf{y})$$

D. A and C
E. all of the above

Examples

\((x \rightarrow x)\)  
-- The identity function (id) that returns its input

\((x \rightarrow (y \rightarrow y))\)  
-- A function that returns (id)

\((f \rightarrow (f (x \rightarrow x)))\)  
-- A function that applies its argument to id

How do I define a function with two arguments?

- e.g. a function that takes \(x\) and \(y\) and returns \(y\)?
(\x -> (\y -> y)) -- A function that returns the identity function

-- OR: a function that takes two arguments
-- and returns the second one!

How do I apply a function to two arguments?

• e.g. apply (\x -> (\y -> y)) to apple and banana?
Syntactic Sugar

```latex
(((\lambda x. (\lambda y. y)) \text{apple}) \text{banana})
```

```latex
((\lambda x_1. (\lambda x_2. (\lambda x_3. e_1))))
```

```latex
(((\lambda x. (\lambda y. y)) \text{apple}) \text{banana}) -- first apply to apple,
-- then apply the result to banana
```

Instead of

<table>
<thead>
<tr>
<th>instead of</th>
<th>we write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
<table>
<thead>
<tr>
<th>instead of</th>
<th>we write</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \rightarrow (y \rightarrow (z \rightarrow e)))</td>
<td>(x \rightarrow y \rightarrow z \rightarrow e)</td>
</tr>
<tr>
<td>(x \rightarrow y \rightarrow z \rightarrow e)</td>
<td>(x \ y \ z \rightarrow e)</td>
</tr>
<tr>
<td>(((e_1 \ e_2) \ e_3) \ e_4)</td>
<td>(e_1 \ e_2 \ e_3 \ e_4)</td>
</tr>
</tbody>
</table>

\(x \rightarrow y\)  
--- *A function that takes two arguments*  
--- *and returns the second one...*

\((x \ y \rightarrow y)\) apple banana  
--- *applied to two arguments*

**Syntax** "Look like"

**Semantic** "mean"

**Semantics : What Programs Mean**

How do I “run” / “execute” a \(\lambda\)-term?
Think of middle-school algebra:

\[(1 + 2) \times ((3 \times 8) - 2)\]

\[= 3 \times ((3 \times 8) - 2)\]

\[= 3 \times (24 - 2)\]

\[= 3 \times 22\]

\[= 66\]

**Execute** = rewrite step-by-step

- Following simple *rules*
- until no more rules *apply*

---

**Rewrite Rules of Lambda Calculus**
But first we have to talk about **scope**

The part of a program where a variable is visible

In the expression $(\lambda x \rightarrow e)$

- $x$ is the newly introduced variable
- $e$ is the scope of $x$
- Any occurrence of $x$ in $(\lambda x \rightarrow e)$ is bound (by the binder $\lambda x$)

For example, $x$ is bound in:

$(\lambda x \rightarrow x)$

$(\lambda x \rightarrow (\lambda y \rightarrow x))$
An occurrence of \( x \) in \( e \) is **free** if it’s **not bound** by an enclosing abstraction.

For example, \( x \) is free in:

\[
\begin{align*}
(x \ y) & \quad \text{-- no binders at all!} \\
(\lambda y \rightarrow (x \ y)) & \quad \text{-- no } \lambda x \text{ binder} \\
((\lambda x \rightarrow (\lambda y \rightarrow y)) \ x) & \quad \text{-- } x \text{ is outside the scope of the } \lambda x \text{ binder;} \\
& \quad \text{-- intuition: it's not "the same" } x
\end{align*}
\]

**QUIZ**

Is \( x \) **bound** or **free** in the expression \( ((\lambda x \rightarrow x) \ x) \)?

A. first occurrence is bound, second is bound

B. first occurrence is bound, second is free
C. first occurrence is free, second is bound
D. first occurrence is free, second is free

EXERCISE: Free Variables

An variable $x$ is **free** in $e$ if there exists a free occurrence of $x$ in $e$

We can formally define the set of all free variables in a term like so:

- $FV(x) = ???$  
  $FV(x) = ???$  
  $FV(x) = ???$
- $FV(\lambda x \rightarrow e) = ???$  
  $FV(\lambda x \rightarrow e) = ???$  
  $FV(\lambda x \rightarrow e) = ???$
- $FV(e_1 e_2) = ???$  
  $FV(e_1 e_2) = ???$  
  $FV(e_1 e_2) = ???$
Closed Expressions

If \( e \) has no free variables, it is said to be closed.

- Closed expressions are also called combinators.

What is the shortest closed expression?

\[ \lambda x \to x \]
1. $\beta$-step (aka function call)
2. $\alpha$-step (aka renaming formals)

**Semantics: Redex**

A **redex** is a term of the form

$$((\text{\textbackslash}x \rightarrow \text{e}1) \text{\textbackslash}e2)$$

A function $(\text{\textbackslash}x \rightarrow \text{e}1)$

- $\text{x}$ is the **parameter**
- $\text{e}1$ is the **returned** expression

Applied to an argument $\text{e}2$

- $\text{e}2$ is the **argument**

\[\begin{align*}
(2+3) \times (5-4) & \\
\downarrow & \\
5 \times (5-4) & \\
\downarrow & \\
5 \times 1 & \\
\downarrow & \\
5 & \\
\end{align*}\]
Semantics: $\beta$-Reduction

A redex $\beta$-steps to another term ...

$$(\lambda x \rightarrow e_1)\ e_2 =_b e_1[x := e_2]$$

where $e_1[x := e_2]$ means

"$e_1$ with all free occurrences of $x$ replaced with $e_2$"

$$\left(\lambda x \rightarrow \underline{\text{apple}}\right)\ \underline{\text{apple}} =_b \underline{\text{apple}}$$

Computation by search-and-replace:

If you see an abstraction applied to an argument,

- In the body of the abstraction
- Replace all free occurrences of the formal by that argument
We say that \((\lambda x \rightarrow e_1) e_2\) \(\beta\)-steps to \(\lambda x := e_2\)

\[ (\lambda x \rightarrow e_1) e_2 \rightarrow b > e_1[x:=e_2] \]

**Redex Examples**

\(((\lambda x \rightarrow x) \text{apple})\)

\(= b > \text{apple} \)

Is this right? Ask Elsa (https://goto.ucsd.edu/elsa/index.html)
QUIZ

`(((\x -> (\y -> y)) apple)`

=⇒ ???

A. apple
B. \y -> apple
C. \x -> apple
D. \y -> y
E. \x -> y
QUIZ

\[ (\lambda x \rightarrow ((y \; x) \; y) \; x)) \; apple = b > ??? \]

A. \(((\text{apple} \; \text{apple}) \; \text{apple}) \; \text{apple})\)

B. \(((y \; \text{apple}) \; y) \; \text{apple})\)

C. \(((y \; y) \; y) \; y)\)

D. apple

QUIZ
EXERCISE

What is a \( \lambda \)-term \( \text{fill\_this\_in} \) such that

\[
(\text{fill\_this\_in} \text{ apple}) = \text{banana}
\]
A Tricky One

\[(\lambda x \rightarrow ?) \text{apple } = b \rightarrow \text{banana} \times \]

\[? \left[ x := \text{apple} \right] = \text{banana} \]

\[\text{banana} \]

\[x, y, z \]

\[\text{FV}(x)\]

\[\text{FV}(e_1, e_2)\]

\[\text{FV}(\lambda x \rightarrow e)\]

Is this right?

\[e = 0 \quad x, y, z\]

\[\text{FV}(e_1, e_2)\]

\[\text{FV}(\lambda x \rightarrow e)\]