

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

```

e ::= x           -- variable 'x'
   | (\x -> e)   -- function that takes a parameter 'x' and returns 'e'
   | (e1 e2)     -- call (function) 'e1' with argument 'e2'

```

Programs are **expressions** e (also called λ -terms) of one of three kinds:

- **Variable**
 - x, y, z
- **Abstraction** (aka *nameless function definition*)
 - $(\lambda x \rightarrow e)$
 - x is the *formal* parameter, e is the *body*
 - “for any x compute e ”
- **Application** (aka function call)
 - $(e1\ e2)$
 - $e1$ is the *function*, $e2$ is the *argument*
 - in your favorite language: $e1(e2)$

(Here each of $e, e1, e2$ can itself be a variable, abstraction, or application)

- OO-lambda
- ieng6.
 - ↗ 'cs130wi21'
 - ↘ looks like $rm -rf \sim/.stack_work$
 - $e ::= x, y, z, \dots$
 - | $(\lambda x \rightarrow e)$
 - | $(\underline{e}_1\ \underline{e}_2)$

Examples

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

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

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

QUIZ

Which of the following terms are syntactically **incorrect**?

A. $(\lambda (\lambda x \rightarrow x) \rightarrow y)$
not a variable

B. $(\lambda x \rightarrow (x x))$

C. $(\lambda x \rightarrow (x (y x)))$

D. A and C



E. all of the above

Examples

`(\x -> x)` -- The identity function (*id*) that returns its input



`(\x -> (\y -> y))` -- A function that returns (*id*)

`(\f -> (f (\x -> x)))` -- A function that applies its argument to *id*

func  *arg* 

How do I define a function with two arguments?

- e.g. a function that takes x and y and returns y?

(e₁ e₂)
 
func *arg*

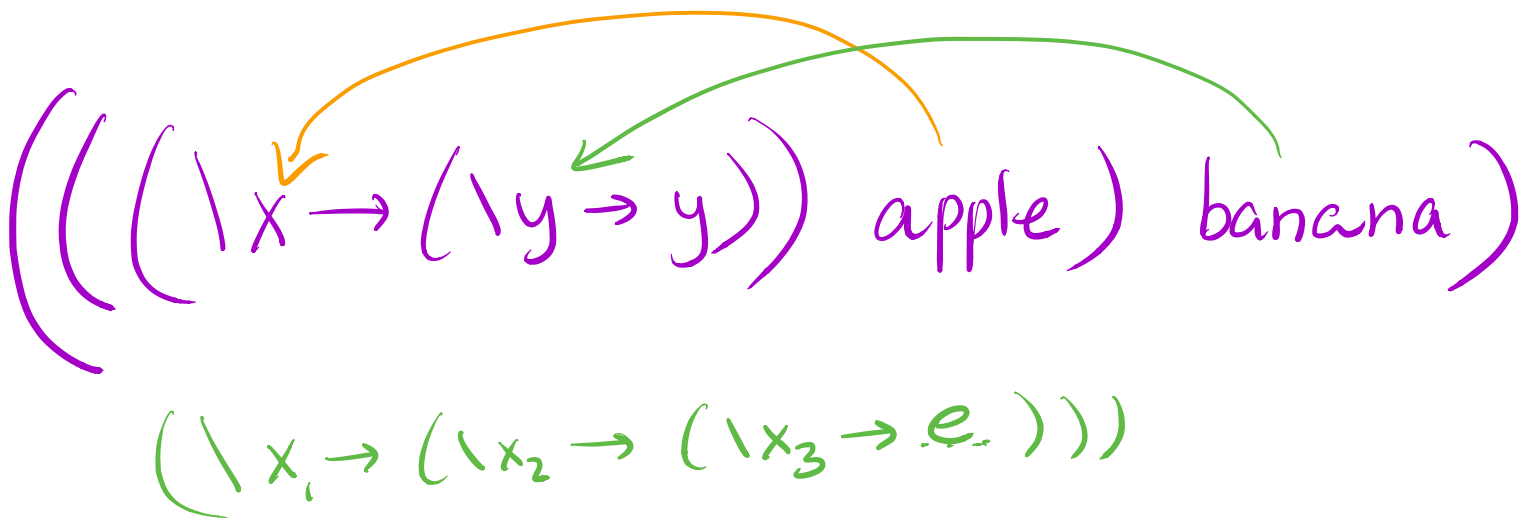
func (x, y) { return y }

(\x -> (\y -> y))

```
(\x -> (\y -> y))  -- A function that returns the identity function
on
-- 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?



$((\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple}) \text{ banana}$ -- *first apply to apple,*
 -- *then apply the result to banana*
ana

Syntactic Sugar

instead of

we write

instead of	we write
$\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow e))$	$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$
$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$	$\lambda x y z \rightarrow e$
$((e1 e2) e3) e4$	$e1 e2 e3 e4$

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

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

Syntax "look like"

Semantic "mean"

Semantics : What Programs Mean

How do I "run" / "execute" a λ -term?

Think of middle-school algebra:

$$\begin{aligned}
 & (1 + 2) * ((3 * 8) - 2) \\
 = & 3 * ((3 * 8) - 2) \\
 = & 3 * (24 - 2) \\
 = & 3 * 22 \\
 = & 66
 \end{aligned}$$

e
 \downarrow
 e_1
 \downarrow
 e_2
 \downarrow
 \vdots
 \downarrow
 66
 "result"

Execute = rewrite step-by-step

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

Rewrite Rules of Lambda Calculus

Programming
"Scope"

1. β -step (aka function call)
2. α -step (aka renaming formals)

But first we have to talk about scope

```

{ var x = "cat"
  { var x = "dog"
    // x is "dog"
  }
  // x is "cat"
}

```

Diagram illustrating variable scope with curly braces and arrows. A blue arrow points from the word "scope" in the text above to the inner brace. A green arrow points from the word "x" in the text to the inner brace. A green arrow points from the word "x" in the text to the outer brace.

Semantics: Scope of a Variable

The part of a program where a variable is visible

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

$\text{func}(x) \{ \text{return } e \}$

- 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** λx)

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:

$(x\ y)$ -- no binders at all!

$(\lambda y \rightarrow (x\ y))$ -- no λx binder

$((\lambda x \rightarrow (\lambda y \rightarrow y))\ x)$ -- x is outside the scope of the λx binder

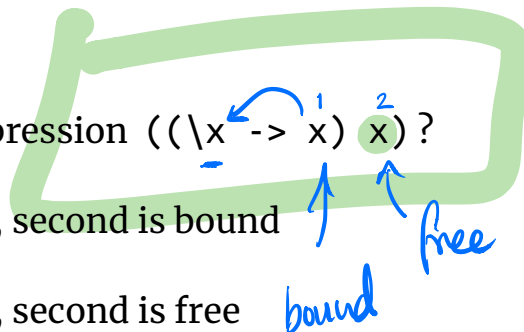
-- intuition: it's not "the same" x

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:

$$\begin{aligned}
 FV(x) &= \{x\} & FV(\lambda x. e) &= \{e\} \\
 FV(\lambda x. e) &= \{e\} & FV(e_1 e_2) &= \{e_1, e_2\} \\
 FV(e_1 e_2) &= \{e_1, e_2\} & FV(\lambda x. \lambda y. e) &= \{e\} \\
 & & FV(\lambda x. \lambda y. e) &= \{e\}
 \end{aligned}$$

Handwritten examples:

- $FV(\lambda x. e) = \{e\}$ (with x above e)
- $FV(\text{apple banana}) = \{\text{apple, banana}\}$
- $FV(\lambda \text{apple} \rightarrow \text{apple}) = \{\}$ (with arrow from apple to apple)
- $FV(\lambda \text{apple} \rightarrow \text{banana}) = \{\text{banana}\}$ (with banana underlined)

$$FV(x) = \{x\}$$

$$FV(\lambda x.e) = FV(e) - x$$

$$FV(e_1 e_2) = FV(e_1) + FV(e_2)$$

↑
func

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?

$$\lambda x.x$$

Rewrite Rules of Lambda Calculus

1. β -step (aka function call)
2. α -step (aka renaming formals)

Semantics: Redex

A redex is a term of the form

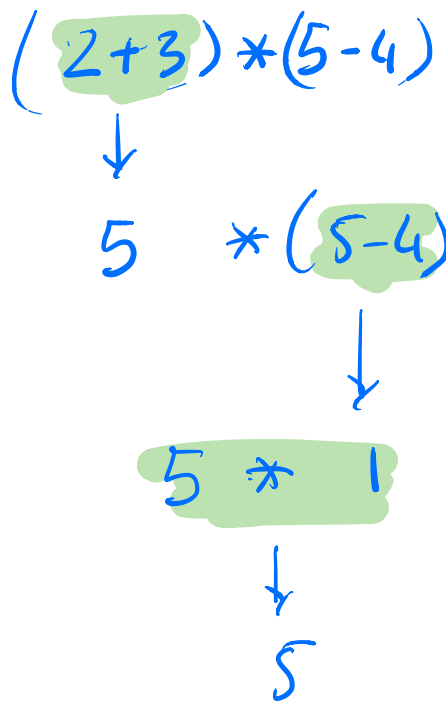
$$((\lambda x \rightarrow e1) e2)$$

A function $(\lambda x \rightarrow e1)$
func *arg*

- x is the parameter
- $e1$ is the returned expression

Applied to an argument $e2$

- $e2$ is the argument



Semantics: β -Reduction

A redex β -steps to another term ...

$$(\lambda x \rightarrow e1) e2 \quad \beta \rightarrow \quad e1[x := e2]$$

where $e1[x := e2]$ means

“ $e1$ with all *free* occurrences of x replaced with $e2$ ”

$$(\lambda x \rightarrow x) \text{ apple} \quad \beta \rightarrow \quad \text{apple}$$

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 $(\lambda x \rightarrow e_1) e_2$ β -steps to $e_1[x := e_2]$

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

Redex Examples

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

\Rightarrow apple

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

QUIZ

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

=b> ???

- A. apple
- B. $\lambda y \rightarrow \text{apple}$
- C. $\lambda x \rightarrow \text{apple}$
- D. $\lambda y \rightarrow y$
- E. $\lambda x \rightarrow y$

$(\lambda x \rightarrow e_1) e_2$

=b> $e_1 [x := e_2]$

$(\lambda y \rightarrow y) [x := \text{apple}]$

QUIZ

$$(\lambda x \rightarrow e_1) e_2$$

$$\Rightarrow e_1[x := e_2]$$

$(\lambda x \rightarrow ((y x) y) x) \text{ apple}$
 \Rightarrow ???

A. (((apple apple) apple) apple)

B. (((y apple) y) apple) ✓

C. (((y y) y) y)

D. apple

QUIZ

QUIZ

$((\lambda x \rightarrow (x (\lambda x \rightarrow x))) \text{apple})$
 $(\lambda x \rightarrow \dots)$ e_1 e_2
 =b> ???

$e_1[x := e_2]$

- A. (apple ($\lambda x \rightarrow x$))
- B. (apple (λ apple \rightarrow apple))
- C. (apple ($\lambda x \rightarrow$ apple))
- D. apple
- E. ($\lambda x \rightarrow x$)

$(\lambda x \rightarrow (x (\lambda x \rightarrow x)))$
 def (binder)

func (x) Σ
 return $x+1$ \swarrow use (occurrences)

$(1+2)+3$
 $= 3+3$
 $= 6$

$(\lambda x \rightarrow \boxed{e_1}) e_2$

EXERCISE

What is a λ -term fill_this_in such that

fill_this_in apple
 =b> banana

$(\dots \text{apple})$
 $=b> \text{banana}$

ELSA: <https://goto.ucsd.edu/elsa/index.html>

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434473_24432.lc)

$(\lambda x \rightarrow ?) \text{ apple} = b \rightarrow \underline{\text{banana}} \quad x$

$? [x := \text{apple}] \equiv \underline{\text{banana}}$
 banana

A Tricky One

$((\lambda x \rightarrow (\lambda y \rightarrow x)) y)$

$= b \rightarrow \lambda y \rightarrow y$

Is this right?

x, y, z
 $FV(x)$

$FV(e_1, e_2)$

$FV(\lambda x \rightarrow e)$

$e = \textcircled{1} \quad \underline{x}, \underline{y}, \underline{z}$

$\textcircled{2} \quad (e_1, e_2)$

$\textcircled{3} \quad (\lambda x \rightarrow \underline{e})$