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 retur
ns '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)
  - `(\x -> 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)

- 00-lambda
- ieng6. 'cs130wi21'

rm -rf ~/.stack_wrk

looks like

$e ::= x, y, z, \ldots$
$\quad | \ (\backslash x \to e)$
$\quad | \ (e_1 \ e_2)$

Examples

```
(\x -> x)             -- The identity function (id) that returns i
ts input

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

(\f -> (f (\x -> x))) -- A function that applies its argument to i
d
```

QUIZ

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

not a variable

A. `(\(\x -> x) -> y)`

B. `(\x -> (x x))`

C. `(\x -> (x (y x)))`

D. A and C

E. all of the above

Examples

```
(\x -> x)                -- The identity function (id) that returns i
ts input

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

(\f -> (f (\x -> x)))    -- A function that applies its argument to i
d
```

func arg

(e1 e2)

func arg

How do I define a function with two arguments?

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

func (x, y) { return y }

(\ x → (\y → y))

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

$(((\backslash x \to (\backslash y \to y))\ apple)\ banana)$

$(\backslash x_1 \to (\backslash x_2 \to (\backslash x_3 \to e)))$

```
(((\x -> (\y -> y)) apple) banana) -- first apply to apple,
                                   -- then apply the result to banana
```

Syntactic Sugar

instead of	we write

instead of	we write
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

```
\x y -> y      -- A function that that takes two arguments
               -- and returns the second one...

(\x y -> 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:

```
    (1 + 2) * ((3 * 8) - 2)
==
    3       * ((3 * 8) - 2)
==
    3       * ( 24     - 2)
==
    3       *  22
==
    66
```

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

Execute = rewrite step-by-step

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

Rewrite Rules of Lambda Calculus

1. β-step (aka $\boxed{\textit{function call}}$)
2. α-step (aka $\boxed{\textit{renaming formals}}$)

Programming
"Scope"

But first we have to talk about **scope**

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

x ?

Semantics: Scope of a Variable

The part of a program where a **variable is visible**

In the expression `(\x -> e)`   func (x) { return e }

- `x` is the newly introduced variable
- `e` is **the scope** of `x`
- any occurrence of `x` in `(\x -> e)` is **bound** (by the **binder** `\x`)

For example, `x` is bound in:

```
(\x -> x)
```

```
(\x -> (\y -> 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!

(\y -> (x y))            -- no \x binder

((\x -> (\y -> y)) x)  -- x is outside the scope of the \x binde
r;
                         -- intuition: it's not "the same" x
```

QUIZ

Is `x` *bound* or *free* in the expression `((\x -> x) x)`?

1 2

free

bound

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)         = ??? {x}
FV(\x -> e)   = ???
FV(e1 e2)     = ???
```

```
FV(apple) = {apple}
FV(apple banane) = {apple, bana}
FV(\apple -> apple) = { }
FV(\apple -> banana) = {banana}
```

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

$FV(\backslash x \to e) = FV(e) - x$

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

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?

$\backslash x \to x$

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

```
((\x -> e1) e2)
```

func arg

A *function* `(\x -> e1)`

- `x` is the *parameter*
- `e1` is the *returned* expression

Applied to an argument `e2`

- `e2` is the *argument*

$(2+3) * (5-4)$

$\downarrow$

$5 * (5-4)$

$\downarrow$

$5 * 1$

$\downarrow$

5

Semantics: β-Reduction

A **redex** b-steps to another term …

```
(\x -> e1) e2   =b>   e1[x := e2]
```

where `e1[x := e2]` means

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

(\x → x) apple =b> 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 $\boxed{\texttt{(\textbackslash x -> e1) e2}}$ β-steps to $\boxed{\texttt{e1[x := e2]}}$

(\x->e1) e2 =b> e1[x:=e2]

Redex Examples

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

=b> apple
```

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

QUIZ

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

=b> ???
```

A. `apple`

B. `\y -> apple`

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

$(\backslash x \to e_1)\ e_2$

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

$\underbrace{(\backslash y \to y)}\ [x := apple]$

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

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

QUIZ

apple apple

```
(\x -> (((y x) y) x)) apple
=b> ???
```

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

B. `(((y apple) y) apple)` ✓

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

D. `apple`

QUIZ

Quiz

a ✓ b ✗ c ✗

```
((\x -> (x (\x -> x))) apple)
=b> ???
```

e_1 e_2

$e_1[x := e_2]$

A. `(apple (\x -> x))`

B. `(apple (\apple -> apple))`

C. `(apple (\x -> apple))`

D. `apple`

E. `(\x -> x)`

(\x → (x (\x → x)))

def (binder)

```
func (x) {
  return x+1
}
```

use (occurrence)

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

$(\backslash x \to \boxed{e_1})\ e_2$

EXERCISE

What is a λ-term `fill_this_in` such that

```
fill_this_in apple
=b> banana
```

(??? apple)
=b> 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)

(\x-> ?) apple =b> banana   x

? [x := apple] ≡ banana

banana

x, y, z

A Tricky One

FV(x)

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

FV(e1 e2)

```
=b> \y -> y
```

?

FV(\x-> e)

Is this right?

e = ① x, y, z

② (e1 e2)

③ (\x-> e)