$$e ::= \underset{\text{var}}{\underset{\uparrow}{x}} \mid \underset{\text{func}}{\underset{\uparrow}{(\lambda x. e)}} \mid \underset{\text{call}}{\underset{\uparrow}{(e_1\ e_2)}}$$

Syntactic Sugar

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))

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

x y

```
(\x y -> y) apple banana -- ... applied to two arguments
```

(((\x→(\y→y)) apple) banana)

$\hookrightarrow^*$ banana

Semantics : What Programs Mean

How do I "run" / "execute" a λ-term?

Think of middle-school algebra:

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

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

Execute = rewrite step-by-step

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

Rewrite Rules of Lambda Calculus

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

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

```
{ int x;
   :
   :
}
```

Semantics: Scope of a Variable

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

In the expression `\x -> 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)
```

B

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 bind
er;
                      -- intuition: it's not "the same" x
```

A BOUND

B FREE

QUIZ

FREE occ of x

1 2

In the expression `(\x -> x) x`, is `x` *bound* or *free*?

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

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

$FV(\backslash x \to \underbrace{(\backslash y \to z)}) = FV(\backslash y \to z) = FV(z) = \{z\}$

e

x, y, z

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

y is free (\x->x) NO

y is free (\y->y) NO

y is free (\x->y) YES

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

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

{x}

FV(e) - x

FV(e_1) U FV(e_2)

FV : Expr → Set of free vars in Expr

not expr

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?

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
```

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

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

Applied to an argument `e2`

- `e2` is the *argument*

Semantics: β-Reduction

A **redex** b-steps to another term …

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

free occ of x replaced by e_2

where `e1[x := e2]` means

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

Computation by *search-and-replace*:

- If you see an *abstraction* applied to an *argument*, take the *body* of the abstraction and replace all free occurrences of the *formal* by that *argument*
- We say that `(\x -> e1) e2` β-steps to `e1[x := e2]`

Redex Examples

(\x → x) apple

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

Is this right? Ask Elsa (http://goto.ucsd.edu:8095/index.html#?demo=blank.lc)!

QUIZ

formal body arg

(\x → (\y → (y apple)))

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

body [formal := arg]

A. `apple`

(\y→y)

B. `\y -> apple`

C. `\x -> apple`

[✓ **D.** `\y -> y`]

E. `\x -> y`

QUIZ

form body arg

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

body [form := arg]

y x y x [x := apple]

y apple y apple

A. `apple apple apple apple`

[✓ **B.** `y apple y apple`] ✓

C. `y y y y`

D. `apple`

QUIZ

form free body bound arg

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

body[form := arg]

✓ x (\x -> x)

x

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

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

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

D. `apple`

E. `\x -> x`

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)

A Tricky One

form

"local"

body

arg

"global variable"

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

free? or bound?

body [x:= arg]

(\y→x) [x:= y]

=> (\y→y)

Is this right?

Something is Fishy

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

Is this right?

Problem: The *free* `y` in the argument has been **captured** by `\y` in *body*!

" become bound "

Solution: Ensure that *formals* in the body are **different from** *free-variables* of argument!

iDE (A) YES
intellij (B) NEVER
eclipse

Capture-Avoiding Substitution

We have to fix our definition of β-reduction:

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

where `e1[x := e2]` means " ~~`e1` with all *free* occurrences of `x` replaced with `e2`~~ "

- `e1` with all *free* occurrences of `x` replaced with `e2`
- **as long as** no free variables of `e2` get captured

Formally:

```
x[x := e]            = e

y[x := e]            = y                           -- as x /=
y

(e1 e2)[x := e]      = (e1[x := e]) (e2[x := e])

(\x -> e1)[x := e]   = \x -> e1                    -- Q: Why l
eave `e1` unchanged?

(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
```

Oops, but what to do if `y` is in the *free-variables* of `e`?

- i.e. if `\y -> ...` may *capture* those free variables?

Rewrite Rules of Lambda Calculus

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

Semantics: α-Renaming

\x → y

```
\x -> e   =a>   \y -> e[x := y]
  where not (y in FV(e))
```

- We rename a formal parameter `x` to `y`
- By replace all occurrences of `x` in the body with `y`
- We say that `\x -> e` α-steps to `\y -> e[x := y]`

Example:

```
\x -> x   =a>   \y -> y   =a>   \z -> z
```

All these expressions are α-**equivalent**

What's wrong with these?

```
-- (A)
\f -> f x    =a>   \x -> x x
```

?

x is free in f x

```
-- (B)
(\x -> (\y -> y)) y   =a>   (\x -> \z -> z) z
```

1

2

NOT ALLOW

Tricky Example Revisited

```
    (\x -> (\y -> x)) y
                                      -- rename 'y' to 'z' to avoid
capture
    =a> (\x -> (\z -> x)) y
                                      -- now do b-step without capt
ure!
    =b> \z -> y
```

To avoid getting confused,

- you can **always rename** formals,
- so different **variables** have different **names**!

Normal Forms

Recall **redex** is a λ-term of the form

`(\x -> e1) e2`

REDEX

A λ-term is in **normal form** if it *contains no redexes.*

$(2+3) * 5$

redex

22

normal-form
no redex!

QUIZ

Which of the following term are **not** in *normal form* ?

ie contain a redex

A. `x` no red

B. `x y` no red

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

D. `x (\y -> y)` no red

E. C and D

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

left right

$$e \overset{a}{\Rightarrow} e_1 \overset{b}{\Rightarrow} e_2 \overset{a}{\Rightarrow} e_3 \overset{a}{\Rightarrow} \dots \Rightarrow \underbrace{e'}_{\text{normal form}}$$

Semantics: Evaluation

A λ-term `e` **evaluates to** `e'` if

1. There is a sequence of steps

```
e =?> e_1 =?> ... =?> e_N =?> e'
```

where each `=?>` is either `=a>` or `=b>` and `N >= 0`

2. `e'` is in *normal form*

Examples of Evaluation

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

```
(\f -> f (\x -> x)) (\x -> x)
  =?> ??? (\b → b)
```

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

Elsa shortcuts

Named λ-terms:

```
let ID = \x -> x  -- abbreviation for \x -> x
```

To substitute name with its definition, use a `=d>` step:

```
ID apple
  =d> (\x -> x) apple     -- expand definition
  =b> apple               -- beta-reduce
```

Evaluation:

- `e1 =*> e2` : `e1` reduces to `e2` in 0 or more steps
  - where each step is `=a>` , `=b>` , or `=d>`
- `e1 =~> e2` : `e1` evaluates to `e2` and `e2` is **in normal form**

EXERCISE

Fill in the definitions of `FIRST` , `SECOND` and `THIRD` such that you get the following behavior in `elsa`

```
let FIRST  = fill_this_in
let SECOND = fill_this_in
let THIRD  = fill_this_in
```

On THU

```
eval ex1 :
(((FIRST apple) banana) orange)
  =*> apple

eval ex2 :
(((SECOND apple) banana) orange)
  =*> banana

eval ex3 :
(((THIRD apple) banana) orange)
  =*> orange
```

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

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434130_24421.lc)

Non-Terminating Evaluation

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

Some programs loop back to themselves…

… and *never* reduce to a normal form!

This combinator is called Ω