Semantics: Redex

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

`(\x -> e1) e2`

REDEX

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

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

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]
```

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
=b> apple
```

$\Rightarrow$ apple

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

DONE

QUIZ

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

BODY ARG

BODY[x := ARG]

↓

BODY

A. `apple`

B. `\y -> apple`

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

NEW

QUIZ

formal body arg

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

- **A.** `apple apple apple apple`
- $\boxed{\textbf{B. } \texttt{y apple y apple}}$ ✓

- **C.** `y y y y`
- **D.** `apple`

QUIZ

x (\x -> x x)

free

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

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
```

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

DIFF FREE

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

\y → y

= BOUND

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*!

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

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 leave `e1
` unchanged?

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

DEF

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*)

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

↓ α-STEP

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

func (x) { return x+1 }

"a-renam"

func(y){return y+1}

(\x -> y) "glob" y

\y -> y "captur" y

Semantics: α-Renaming

```
\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:

fun(y) {ret y}

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

All these expressions are α-**equivalent**

fun(x) {ret x}

fun(z) {ret z}

What's wrong with these?

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

Do not replace the FREE 'y'

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

FREE

Tricky Example Revisited

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

=a> (\x -> (\z -> x)) y      -- rename 'y' to 'z' to avoid capture

=b> \z -> y                  -- now do b-step without capture!
```

To avoid getting confused,

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

Normal Forms = Has No Redex

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

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

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

QUIZ

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

A. `x` NF

B. `x y` NF

C. `(\x -> x) y` not NF

D. `x (\y -> y)` NF

ie do NOT CONTAIN ANY REDEX

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

E. ~~C and D~~

Semantics: Evaluation

A λ-term `e` **evaluates to** `e'` if

1. There is a sequence of steps

$$
\begin{aligned}
& e \\
& \overset{?}{\Rightarrow} e_1 \\
& \overset{?}{\Rightarrow} e_2 \\
& \vdots
\end{aligned}
$$

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

$\Rightarrow$ e'

NF

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)
  =?> ???
```

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

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

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

What if we pass Ω as an argument to another function?

```
let OMEGA = (\x -> x x) (\x -> x x)
```

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

Does this reduce to a normal form? Try it at home!

Programming in λ-calculus

Real languages have lots of features

$$\boxed{e ::= x \mid \backslash x \to e \mid (e_1\ e_2)}$$

- Booleans ✓
- Records (structs, tuples) ✓
- Numbers ✓ LIST
- **Functions** [we got those]
- Recursion ✓

Lets see how to *encode* all of these features with the λ-calculus.