Semantics: Redex

A redex is a term of the form
A function $(\lambda x \rightarrow e_1) e_2$

- $x$ is the parameter
- $e_1$ is the returned expression

Applied to an argument $e_2$

- $e_2$ is the argument
Semantics: $\beta$-Reduction

A redex $b$-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 $e_2$"

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

Redex Examples
\[(\lambda x \rightarrow x) \text{apple} \Rightarrow \text{apple} = b \rightarrow \text{apple}\]

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

\(\text{Body } [x := \text{ARG}]\)

A. apple
B. \(y \to \text{apple}\)
C. \(x \to \text{apple}\)
D. \(y \to y\)
E. \(x \to y\)
**QUIZ**

\[
\lambda x \to y \times y \times \text{apple}
\]

- **A.** apple apple apple apple apple
- **B.** y apple y apple

NEW

https://ucsd-cse230.github.io/sp20/lectures/01-lambda.html
• C. y y y y
• D. apple

QUIZ

https://ucsd-cse230.github.io/sp20/lectures/01-lambda.html
\((\lambda x \rightarrow x (\lambda x \rightarrow x))\) apple

= b

A. apple (\(\lambda x \rightarrow x\))
B. apple (\(\lambda apple \rightarrow apple\))
C. apple (\(\lambda x \rightarrow apple\))
D. apple
E. \(\lambda x \rightarrow x\)
EXERCISE

What is a $\lambda$-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

$(\lambda x \rightarrow (\lambda y \rightarrow x)) \ y = b > \lambda y \rightarrow y = \text{BOUND}$
Is this right?

Something is Fishy
\( (\lambda x \rightarrow (\lambda y \rightarrow x)) \; y \)

= \( \lambda y \rightarrow y \)

Is this right?

**Problem:** The free \( y \) in the argument has been captured by \( \lambda y \) 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 $\beta$-reduction:

$$(\lambda x \to e1) \ e2 =_{\beta} e1[x := e2]$$

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:
\[
x[x := e] = e
\]
\[
y[x := e] = y
\quad -- \text{as } x /= y
\]
\[
(e1 \ e2)[x := e] = (e1[x := e]) \ (e2[x := e])
\]
\[
(\lambda x \to e1)[x := e] = \lambda x \to e1
\quad -- \text{Q: Why leave `e1 `unchanged?}
\]
\[
(\lambda y \to e1)[x := e]
\quad | \text{not } (y \ \text{in } \text{FV}(e)) = \lambda y \to e1[x := e]
\]

Oops, but what to do if \( y \) is in the \text{free-variables} of \( e \)?

- i.e. if \( \lambda y \to \ldots \) may capture those free variables?
Rewrite Rules of Lambda Calculus

1. $\beta$-step (aka function call)
2. $\alpha$-step (aka renaming formals)
Semantics: $\alpha$-Renaming

- We rename a formal parameter $x$ to $y$
- By replace all occurrences of $x$ in the body with $y$
• We say that \( x \rightarrow e \) \( \alpha \)-steps to \( y \rightarrow e[ x := y ] \)

Example:

\[
\begin{align*}
\text{fun}(y) \ \& \ \text{ret} \ y^3 \\
\hline
\text{x} \rightarrow \ x & = a > \quad \text{y} \rightarrow \ y & = a > \quad \text{z} \rightarrow \ z
\end{align*}
\]

All these expressions are \( \alpha \)-equivalent

\[
\begin{align*}
\text{fun}(x) \ \& \ \text{ret} \ x^3 \\
\text{fun}(z) \ \& \ \text{ret} \ z^3
\end{align*}
\]

What’s wrong with these?

\[
\begin{align*}
\text{-- (A)}
\hline
f \rightarrow f \ x & = a > \quad x \rightarrow x \ x
\end{align*}
\]

\[
\begin{align*}
\text{-- (B)}
\hline
( \text{x} \rightarrow \ y \rightarrow y ) \ y & = a > \quad ( \text{x} \rightarrow \ z \rightarrow z ) \ z
\end{align*}
\]

Do not replace the FREE \('y'\)
Tricky Example Revisited
To avoid getting confused,

- you can always rename formals,
- so different variables have different names!
Normal Forms = Has No Redex

Recall \textbf{redex} is a \(\lambda\)-term of the form

\[ (\lambda x \to e_1) \, e_2 \]

A \(\lambda\)-term is in \textbf{normal form} if it contains no redexes.
QUIZ

Which of the following terms are not in normal form?

A. \( x \) \( \text{NF} \)

B. \( x \ y \) \( \text{NF} \)

C. \( (\lambda x \to x) \ y \) \( \not\text{NF} \)

D. \( x \ (\lambda y \to y) \) \( \text{NF} \)

ie do NOT contain ANY REDEX

\( (\lambda x \to \varepsilon_1) \ v \)
**Semantics: Evaluation**

A $\lambda$-term $e$ evaluates to $e'$ if

1. There is a sequence of steps

\[ e \Rightarrow e_1 \Rightarrow e_2 \]
e =?> e_1 =?> ... =?> e_N =?> e' => e'

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

2. e' is in normal form

---

**Examples of Evaluation**

\[
\lambda x . x \text{ apple} =_b \text{ apple}
\]
\((f \rightarrow f ((x \rightarrow x)) (\x \rightarrow x))\) 
=???

\((\x \rightarrow x x) (\x \rightarrow x)\) 
=???

**Elsa shortcuts**

Named \(\lambda\)-terms:

```
let ID = \x \rightarrow x -- abbreviation for \x \rightarrow x
```
To substitute name with its definition, use a \( \Rightarrow \) step:

\[
\text{ID apple} \\
\Rightarrow (\lambda x \to x) \text{apple} \quad -- \text{expand definition} \\
\Rightarrow \text{apple} \quad -- \text{beta-reduce}
\]

Evaluation:

- \( e_1 \Rightarrow^* e_2 \): \( e_1 \) reduces to \( e_2 \) in 0 or more steps
  - where each step is \( \Rightarrow a \), \( \Rightarrow b \), or \( \Rightarrow d \)
- \( e_1 \Rightarrow e_2 \): \( e_1 \) evaluates to \( e_2 \) and \( e_2 \) 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

\((\lambda x \rightarrow x \, x) \, (\lambda x \rightarrow x \, x)\)
\[=b\] \((\lambda x \rightarrow x \, x) \, (\lambda x \rightarrow x \, x)\)

Some programs loop back to themselves...

... and *never* reduce to a normal form!

This combinator is called \(\Omega\)

What if we pass \(\Omega\) 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 $\lambda$-calculus

Real languages have lots of features
- Booleans ✓
- Records (structs, tuples) ✓
- Numbers ✓
- **Functions** [we got those]
- Recursion ✓

Let's see how to *encode* all of these features with the \(\lambda\)-calculus.