QuasiPatterns

We can also use quasiquoting in a match pattern
We call this a quasi\textit{pattern}

It turns out this lets us build an implementation of a \textit{little language}!
(define (interpret-binary-arith e)
  (match e
    [`(+ ,e1 ,e2) (+ (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [`(- ,e1 ,e2) (- (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [(? number? n) n]
    [else (error "bad expression...")]]))

Exercise: call interpret-binary-arith on the following...

3
(+ 2 3)
(+ (- (+ 2 3) 5) (+ 1 (- 2 3)))
Quiz

What’s the difference between the following two expressions?

(\text{interpret-binary-arith}
 (\texttt{(+ (- (+ 2 3) 5) (+ 1 (- 2 3)))))

(\text{interpret-binary-arith}
 \texttt{('(+ (- (+ 2 3) 5) (+ 1 (- 2 3))))})

Answer: in one we’re cheating. We’re not really using our interpreter, we’re just using Racket
The Lambda Calculus

• A system for calculating based entirely on computing with functions.

• Developed as a foundation for mathematics (originally used to model the natural numbers) by Alonzo Church in 1936.

• Church’s thesis: “Every effectively calculable function (effectively decidable predicate) is general recursive”, i.e., can be computed using the λ-calculus. Used to show there exist unsolvable problems.

• One of the simplest Turing-equivalent languages!
  
  • Church, with his student Alan Turing, proved the equivalent expressiveness of Turing machines and the λ-calculus (called the Church-Turing thesis).

• Still makes up the heart of all functional programming languages!
The Lambda Calculus

* lambdas are just anonymous functions! 

\[
e \in \text{Exp} ::= (\lambda (x) \ e) \quad \text{\(\lambda\)-abstraction} \\
| (e \ e) \quad \text{function application} \\
| x \quad \text{variable reference}
\]

\[
x \in \text{Var} ::= \langle \text{variables} \rangle
\]
Textual-reduction semantics

• One way of designing a formal semantics is as a relation over terms in the language—one that reduces the term textually.

• This is usually **small-step**—each eval step must terminate (meaning there are no *premises above the line* in our rules of inference and no recursive use of the interpreter within a step.)

• Consider a small-step semantics for our arithmetic language:

\[
\begin{align*}
a \in \text{AExp} & ::= n | a + a | a - a | a \times a \\
n, m \in \text{Num} & ::= \langle \text{integer constants} \rangle
\end{align*}
\]
Textual-reduction semantics

\[ a \in A\text{Exp} ::= n \mid a + a \mid a - a \mid a \times a \]

\[ n, m \in \text{Num} ::= \langle \text{integer constants} \rangle \]

- Rules to reduce terms in this language match operations that have two numeric operands already and apply the operation, textually substituting a numeric value for the operation; e.g.:

\[ a_0 \times a_1 \Rightarrow n_0 \times n_1 \]

where \( a_0 \) is \( n_0 \) and \( a_1 \) is \( n_1 \)

- For example: \( 2 \times 3 + 4 \times 5 \Rightarrow 2 \times 3 + 20 \Rightarrow 6 + 20 \Rightarrow 26 \)

- Is there another way to evaluate \( 2 \times 3 + 4 \times 5 \) using similar rules?
The Lambda Calculus

*lambda* are just anonymous functions!

\[
e \in \text{Exp} ::= (\lambda (x) e) \quad \lambda\text{-abstraction}
\]
\[
| \quad (e \ e) \quad \text{function application}
\]
\[
| \quad x \quad \text{variable reference}
\]

\[
x \in \text{Var} ::= \langle\text{variables}\rangle
\]
The Lambda Calculus

The lambda-calculus is the functional core of Racket (as of other functional languages).

Just the following subset of Racket is Turing-equivalent!

\[ e \in \text{Exp ::= (}\lambda (x) \; e) \quad \text{\text{(lambda } (x) \; e)} \]
\[ \quad | \; (e \; e) \quad \text{\text{(e} \; \text{e)}} \]
\[ \quad | \; x \quad \text{x} \]

\[ x \in \text{Var ::= } \langle \text{variables} \rangle \]
Lambda Abstraction

An expression, abstracted over all possible values for a formal parameter, in this case, x.

\[
(\lambda (x) \ e)
\]

Formal parameter  Function body
Application

When the first expression is evaluated to a value (in this language, all values are functions!) it may be invoked / applied on its argument.

\[ (e \ e) \]

Expression in *function position*  
Expression in *argument position*
Variable

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.
We define a rule for step-by-step evaluation called *Beta-reduction*.
Textual substitution. This says: replace every $x$ in $E_0$ with $E_1$.

$$(\lambda (x) E_0) E_1) \rightarrow_{\beta} E_0 [x \leftarrow E_1]$$

(\text{reducible expression})
\[( ((\lambda \ (x) \ x) \ (\lambda \ (x) \ x)) ) \]

\[
\xleftarrow{\beta} \]

\[
\ x [ x \leftarrow (\lambda \ (x) \ x) ] \]
Try an example. Can you beta-reduce this term?
Can you beta-reduce it more than once?

(((\(x\) \((x\ x)\)) \((\lambda \(x\) \((x\ x)\))\))
\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]

\(\beta\) reduction may continue indefinitely (i.e., in non-terminating programs)

\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]

\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]

\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]

\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]

\[
(((\lambda x)(x x)) (\lambda x)(x x))
\]
This specific program is known as $\Omega$ (Omega).
\[ ((\lambda (x) (x\ x))\ (\lambda (x) (x\ x))) \]

\[ \Rightarrow \beta \]

\[ \Omega \text{ is the smallest non-terminating program!} \]

\[ ((\lambda (x) (x\ x))\ (\lambda (x) (x\ x))) \]

\[ \Rightarrow \beta \]

Note how it reduces to itself in a single step!
Evaluation with $\beta$ reduction is nondeterministic!

$$(((\lambda (w) \ w) (\lambda (x) \ x)) \ ((\lambda (y) \ y) (\lambda (z) \ z)))) \\Downarrow \beta \\Downarrow ((\lambda (x) \ x) \ ((\lambda (y) \ y) (\lambda (z) \ z))))$$
Evaluation with $\beta$ reduction is nondeterministic!

$$((\lambda (w) \ w) \ (\lambda (x) \ x)) \ (\lambda (y) \ y) \ (\lambda (z) \ z))$$

$\beta$ or! $\beta$

$$((\lambda (x) \ x) \ (\lambda (y) \ y) \ (\lambda (z) \ z))$$

$$(((\lambda (w) \ w) \ (\lambda (x) \ x)) \ (\lambda (z) \ z))$$
Try an example. Perform each possible β-reduction

\(((\lambda(x))((\lambda(y)(x\ y))\ x))\ (\lambda(z)(z\ z)))\)

How many different β-reductions are possible from the above?
Answer

\[
(\lambda x \ ((\lambda y (x y)) x)) \ (\lambda z (z z))
\]

\[
\downarrow \beta
\]

\[
(\lambda x \ ((x x)) \ (\lambda z (z z))
\]

Can reduce inner redex...
Answer

$$\beta$$

Or the outer redex.
Can’t reduce this since we don’t (yet) know about the particular value (function) $z$ in call position.
Free variables

\[ \text{FV} : \text{Exp} \rightarrow \mathcal{P}(\text{Var}) \]

\[ \text{FV}(x) \triangleq \{ x \} \]

\[ \text{FV}((\lambda (x) \; e_b)) \triangleq \text{FV}(e_b) \setminus \{ x \} \]

\[ \text{FV}(e_f \; e_a) \triangleq \text{FV}(e_f) \cup \text{FV}(e_a) \]
Free variables

\[ \text{FV}(((x\ y))) = \{x, y\} \]

\[ \text{FV}(((\lambda (x) x)\ y)) = \{y\} \]

\[ \text{FV}(((\lambda (x) x)\ x)) = \{x\} \]

\[ \text{FV}(((\lambda (y) ((\lambda (x) (z\ x))\ x))) = \{z, x\} \]
Try an example. What are the free variables of each of the following terms?

\[ (((\lambda (x) x) y) \]

\[ (((\lambda (x) (x x)) (\lambda (x) (x x))) \]

\[ (((\lambda (x) (z y)) x) \]
Try an example. What are the free variables of each of the following terms?

\[ ((\lambda (x) x) y) \]
\[ \{y\} \]

\[ (((\lambda (x) (x x)) \ (\lambda (x) (x x))) \}
\[ \{\} \]

\[ (((\lambda (x) (z y)) x) \}
\[ \{x, y, z\} \]
The problem with (naive) textual substitution

\[( (\lambda \ (a) \ (\lambda \ (a) \ a)) \ (\lambda \ (b) \ b) ) \]
The problem with (naive) textual substitution

\[
((\lambda (a) (\lambda (a) a)) (\lambda (b) b))
\]

\[
\beta
\]

\[
(\lambda (a) a)[a \leftarrow (\lambda (b) b)]
\]
The problem with (naive) textual substitution

\[(\lambda (a) (\lambda (a) a)) (\lambda (b) b))\]

\[\beta\]

\[(\lambda (a) (\lambda (b) b))\]
Capture-avoiding substitution

\[ E_\theta[x \leftarrow E_1] \]
\[ x[x \leftarrow E] = E \]

\[ y[x \leftarrow E] = y \text{ where } y \neq x \]

\[ (E_0 \ E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \ E_1[x \leftarrow E]) \]

\[ (\lambda (x) \ E_0)[x \leftarrow E] = (\lambda (x) \ E_0) \]

\[ (\lambda (y) \ E_0)[x \leftarrow E] = (\lambda (y) \ E_0[x \leftarrow E]) \]

where \( y \neq x \) and \( y \not\in \text{FV}(E) \)

\[ \beta\text{-reduction cannot occur when } y \in \text{FV}(E) \]
Capture-avoiding substitution

\(((\lambda\ (a)\ (\lambda\ (a)\ a))\ (\lambda\ (b)\ b))\)

\[\Downarrow\beta\]

\((\lambda\ (a)\ a)\)

✓
Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

\[
(((\lambda \ y)\\((\lambda \ z)\ ((\lambda \ y)\ (z \ y)))\ y))\\((\lambda \ x)\ x))
\]
Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

\[
( ((\lambda \ (y) \\
\quad ((\lambda \ (z) \ (\lambda \ (y) \ (z \ y))) \ y)) \ y) \\
(\lambda \ (x) \ x))
\]

\[\xrightarrow{\beta}\]

\[
(((\lambda \ (z) \ (\lambda \ (y) \ (z \ y))) \ (\lambda \ (x) \ x))
\]
Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

$$(\lambda y)((\lambda z)(\lambda y)z)(\lambda x)y)$$
Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

\[(\lambda (y) (((\lambda (z) (\lambda (y) z)) (\lambda (x) y))))\]

You cannot! This redex would require:

\[(\lambda (y) z)[z \leftarrow (\lambda (x) y)]\]

(y is free here, so it would be captured)
Try an example. How can you beta-reduce the following expression using capture-avoiding substitution?

\[
(\lambda (y) (((\lambda (z) (\lambda (y) z)) (\lambda (x) y))))
\]

\[\rightarrow_\alpha (\lambda (y) (((\lambda (z) (\lambda (w) z)) (\lambda (x) y))))\]

\[\rightarrow_\beta (\lambda (y) (\lambda (w) (\lambda (x) y))))\]

Instead we alpha-convert first.
\(\alpha\)-renaming

\[(\lambda (x) (\lambda (y) x)) \quad (\lambda (a) (\lambda (b) a))\]

These two expressions are equivalent—they only differ by their variable names \((x = a; y = b)\)
\( \alpha \)-renaming

\[
(\lambda (x) \ E_\theta) \rightarrow_\alpha (\lambda (y) \ E_\theta[x \leftarrow y])
\]

\(=\alpha\)

\(\alpha\) renaming/conversions can be run backward, so you might think of it as an equivalence relation.
\(\alpha\) - renaming

\(\alpha\) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda\ (x)\ (\lambda\ (x)\ x))\ (\lambda\ (y)\ y))
\]
\(\alpha\)-renaming

\(\alpha\) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda (x) (\lambda (x) x)) (\lambda (y) y))
\]

Can’t perform naive substitution w/o capturing x.
α-re naming

α renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[ ( (\lambda (x) (\lambda (x) x)) (\lambda (y) y) ) \]

Fix by α renaming to z
\( \alpha \)-renaming

\( \alpha \) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
(((\lambda (x) (\lambda (z) z)) \ (\lambda (y) y))
\]

\( \uparrow \)

Fix by \( \alpha \) renaming to \( z \)
\[ \alpha \text{-renaming} \]

\[ \alpha \text{ renaming/conversions can be used to implement capture-avoiding substitution} \]

Rename variables that would break naive substitution!

\[ (((\lambda x)(\lambda z)z)) (\lambda y) y) \]

Could now perform beta-reduction with naive substitution
\[ \eta \text{-reduction} \]

\[
(\lambda (x) \ (E_\theta \ x)) \ \rightarrow_\eta \ E_\theta \ \text{where} \ x \not\in \text{FV}(E_\theta)
\]
$\eta$-expansion

$$E_\theta \rightarrow_\eta (\lambda (x) (E_\theta \ x)) \text{ where } x \not\in \text{FV}(E_\theta)$$
Reduction

\[(\rightarrow) = (\rightarrow_\beta) \cup (\rightarrow_\alpha) \cup (\rightarrow_\eta)\]

\[(\rightarrow^*)\]

reflexive/transitive closure
Evaluation

\[ E_0 \]
\[ * \]
\[ \rightarrow \]
\[ E_8 \]
\[ * \]
\[ \rightarrow \]
\[ ? \]
Evaluation to *normal form*

\[ E_0 \]

\[ * \]

\[ (\lambda \ (x) \ ... ) \]
Evaluation to *normal form*

\[
E_0 \\ * \\
(\lambda (x) \ldots (\lambda (z) ((a \ldots) \ldots)))
\]

In *normal form*, no function position can be a lambda; this is to say: *there are no unreduced redexes left!*
Evaluation Strategy

E₀

*      *

E₁     E₂
Evaluation Strategy

\[
\text{\ensuremath{\left( \left( \lambda \left( x \right) \left( \left( \lambda \left( y \right) y \right) x \right) \right) \left( \lambda \left( z \right) z \right) } \rightarrow_{\eta} \left( \left( \lambda \left( y \right) y \right) \left( \lambda \left( z \right) z \right) \right) \rightarrow_{\beta} \left( \lambda \left( z \right) z \right) \right)}
\]
Evaluation Strategy

\[( ((\lambda (x)) ((\lambda (y) y) x)) \ (\lambda (z) z)) \]

\[\rightarrow_{\beta} \ ((\lambda (y) y) \ (\lambda (z) z)) \]

\[\rightarrow_{\beta} \ (\lambda (z) z) \]
Evaluation Strategy

\(((\lambda\ x)\ ((\lambda\ (y)\ y)\ x))\ (\lambda\ (z)\ z))\)

\[\rightarrow_\beta\ ((\lambda\ (x)\ x)\ (\lambda\ (z)\ z))\]

\[\rightarrow_\beta\ (\lambda\ (z)\ z)\]
Confluence

Diverging paths of evaluation must eventually join back together.

\[
\begin{array}{c}
E_0 \\
& \uparrow \ast \\
E_1 & \ast & E_2 \\
& \ast \downarrow & \\
E_3 & & \\
\end{array}
\]

Church-Rosser Theorem
$e_0 \leftrightarrow \alpha \leftrightarrow e_1 \leftrightarrow \alpha \leftrightarrow e_2 \leftrightarrow \alpha \leftrightarrow e_3 \leftrightarrow \alpha \leftrightarrow e_4 \leftrightarrow \alpha \leftrightarrow e_5$
Confluence (i.e., Church-Rosser Theorem)

\[ e_0 \xrightarrow{\alpha} e_1 \xrightarrow{\alpha} e_2 \xrightarrow{\alpha} e_3 \xrightarrow{\alpha} e_4 \xrightarrow{\alpha} e_5 \]

\[ e_7 \xrightarrow{\beta} e_8 \xrightarrow{\beta} e_9 \]

\[ e_{10} \xrightarrow{\beta} e_{11} \xrightarrow{\beta} e_{12} \]

\[ \text{β} \quad \text{α} \quad \text{β} \quad \text{β} \quad \text{β} \quad \text{β} \]

\[ * \quad * \quad * \]
Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

Normal evaluation order

Always evaluates the *outermost* leftmost redex first.
Applicative evaluation order

$$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$$

Normal evaluation order

$$((((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) (\lambda (w) w))$$
Call-by-value (CBV) semantics

Applicative evaluation order, \textit{but not under lambdas}.

Call-by-name (CBN) semantics

Normal evaluation order, \textit{but not under lambdas}. 
Try an example.

Write a lambda term other than $\Omega$ which also does not terminate

(Hint: think about using some form of self-application)
Write a lambda term other than $\Omega$ which also does not terminate

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

$$(\lambda (u) ((u\ u)\ u))$$
$$(\lambda (u) ((u\ u)\ u)))$$

$$(\lambda (x) x)$$
$$(\lambda (u) (u\ u))$$
$$(\lambda (u) (u\ u))))$$
Evaluation contexts

Restrict the order in which we may simplify a program’s redexes

(\mathcal{E} ::= (\mathcal{E} \ e) \\
| (\nu \mathcal{E}) \\
| \square)

(left-to-right) CBV evaluation

(\mathcal{E} ::= (\mathcal{E} \ e) \\
| \square)

(left-to-right) CBN evaluation

\nu ::= (\lambda (x) \ e)

\mathcal{v} ::= (\lambda (x) \ e) \\
| (e e) \\
| x

\mathcal{e} ::= (\lambda (x) \ e)
Context and redex

\[ r = \left( (\lambda (x) \left( (\lambda (y) y) \ x \right)) \ (\lambda (z) z) \right) \ \left( \lambda (w) w \right) \]

\[ \mathcal{E} = (\Box \ (\lambda (w) w)) \]

For CBV a redex must be \((v \ v)\)
For CVN, a redex must be \((v \ e)\)
Context and redex

\[ \mathcal{E}[r] = \]

\[
((\lambda (x) (\lambda (y) y) x)) (\lambda (z) z) \]

\( \mathcal{E} = (\square (\lambda (w) w)) \)

\[ r = ((\lambda (x) (\lambda (y) y) x)) (\lambda (z) z) \]

\( \rightarrow^\beta ((\lambda (y) y) (\lambda (z) z)) \)
Put the reduced redex back in its evaluation context...

\[ \mathcal{E} = (\Box (\lambda (w) w)) \]

\[ r = ((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) \]

\[ \rightarrow_\beta ((\lambda (y) y) (\lambda (z) z)) \]

\[ \mathcal{E}[r] \]

\[ (((\lambda (y) y) (\lambda (z) z)) (\lambda (w) w)) \]
Exercises—can you evaluate…

1) \(((\lambda (y) y) (\lambda (z) z)) (\lambda (w) w))\)

2) \(((\lambda (u) (u u)) (\lambda (x) (\lambda (x) x)))\)

3) \(((\lambda (x) x) (\lambda (y) y))
   \(((\lambda (u) (u u)) (\lambda (z) (z z))))\)