**Racket Functions**

Functions: most important building block in Racket (and 251)
- Functions/procedures/methods/subroutines abstract over computations
- Like Java methods, Python functions have arguments and result
- But no classes, this, return, etc.

Examples:

```
(define dbl (lambda (x) (* x 2)))
(define quad (lambda (x) (dbl (dbl x))))
(define avg (lambda (a b) (/ (+ a b) 2)))
(define sqr (lambda (n) (* n n)))
(define n 10)
(define small? (lambda (num) (<= num n)))
```

---

**Function calls (applications)**

To use a function, you call it on arguments (apply it to arguments).
E.g. in Racket: `(dbl 3), (avg 8 12), (small? 17)`

Syntax: `(e0 e1 ... en)`
- A function call expression has no keyword. A function call because it’s the only parenthesized expression that doesn’t begin with a keyword.
- `e0`: any expression, known as the rator of the function call (i.e., the function position).
- `e1` ... `en`: any expressions, known as the rands of the function call (i.e., the argument positions).

Evaluation rule:
1. Evaluate `e0` ... `en` in the current environment to values `v0` ... `vn`
2. If `v0` is not a lambda expression, raise an error.
3. If `v0` is a lambda expression, returned the result of applying it to the argument values `v1` ... `vn` (see following slides).
Function application

What does it mean to apply a function value (lambda expression) to argument values? E.g.

((lambda (x) (* x 2)) 3)
((lambda (a b) (/ (+ a b) 2)) 8 12)

We will explain function application using two models:
1. The substitution model: substitute the argument values for the parameter names in the function body.
2. The environment model: extend the environment of the function with bindings of the parameter names to the argument values.

Function application: substitution model

Example 1:

((lambda (x) (* x 2)) 3)
Substitute 3 for x in (* x 2) and evaluate the result:
(* 3 2) ↓ 6 (environment doesn’t matter in this case)

Example 2:

((lambda (a b) (/ (+ a b) 2)) 8 12)
Substitute 3 for x in (* x 2) and evaluate the result:
(/ (+ 8 12) 2) ↓ 10 (environment doesn’t matter in this case)

Substitution notation

We will use the notation

\[ e[v_1, \ldots, v_n/\text{id}_1, \ldots, \text{id}_n] \]

to indicate the expression that results from substituting the values \( v_1, \ldots, v_n \) for the identifiers \( \text{id}_1, \ldots, \text{id}_n \) in the expression \( e \).

For example:

- \( (* x 2)[3/x] \) stands for \( (* 3 2) \)
- \( (/ (+ a b) 2)[8,12/a,b] \) stands for \( (/ (+ 8 12) 2) \)
- \( (if (< x z) (+ (* x x) (* y y)) (/ x y))[3,4/x,y] \)
  stands for \( (if (< 3 z) (+ (* 3 3) (* 4 4)) (/ 3 4)) \)

It turns out that there are some very tricky aspects to doing substitution correctly. We’ll talk about these when we encounter them.

Function call rule: substitution model

\[ e_0 \# env \downarrow (\text{lambda} \ (\text{id}_1 \ldots \text{id}_n) \ e_{\text{body}}) \]
\[ e_1 \# env \downarrow v_1 \]
\[ \vdots \]
\[ en \# env \downarrow v_n \]
\[ e_{\text{body}}[v_1 \ldots v_n/\text{id}_1 \ldots \text{id}_n] \# env \downarrow v_{\text{body}} \]

(function call)

\( e_0 \ e_1 \ \ldots \ en \# env \downarrow v_{\text{body}} \)

Note: no need for function application frames like those you’ve seen in Python, Java, C,...
Substitution model derivation

Suppose \( env2 = \text{dbl} \rightarrow (\lambda x \cdot (* x 2)) \),
\( \quad \text{quad} \rightarrow (\lambda x \cdot (\text{dbl} \ (\text{dbl} \ x))) \)

\[
\begin{align*}
\text{quad} \ # \ env2 \downarrow (\lambda x \cdot (\text{dbl} \ (\text{dbl} \ x))) \\
\quad 3 \ # \ env2 \downarrow 3 \\
\quad \text{dbl} \ # \ env2 \downarrow (\lambda x \cdot (* x 2)) \\
\quad \text{dbl} \ # \ env2 \downarrow (\lambda x \cdot (* x 2)) \\
\quad 3 \ # \ env2 \downarrow 3 \\
\quad (* 3 \ 2) \ # \ env2 \downarrow 6 \text{ (multiplication rule, subparts omitted)} \\
\quad (\text{dbl} \ 3) \ # \ env2 \downarrow 6 \text{ (function call)} \\
\quad (* 6 \ 2) \ # \ env2 \downarrow 12 \text{ (multiplication rule, subparts omitted)} \\
\quad (\text{dbl} \ (\text{dbl} \ 3)) \ # \ env2 \downarrow 12 \text{ (function call)} \\
\quad (\text{quad} \ 3) \ # \ env2 \downarrow 12
\end{align*}
\]

Substitution model derivation: your turn

Suppose \( env3 = n \rightarrow 10, \quad \text{small?} \rightarrow (\lambda \text{num} \cdot (\leq \text{num} \ n)) \)
\( \quad \text{sqr} \rightarrow (\lambda n \cdot (* n \ n)) \)

Give an evaluation derivation for \((\text{small?} \ (\text{sqr} \ n)) \ # \ env3\)

Stepping back: name issues

Do the particular choices of function parameter names matter?

Is there any confusion caused by the fact that \text{dbl} and \text{quad} both use \( x \) as a parameter?

Are there any parameter names that we can’t change \( x \) to in \text{quad}? 

In \((\text{small?} \ (\text{sqr} \ n))\), is there any confusion between the global parameter name \( n \) and parameter \( n \) in \text{sqr}? 

Is there any parameter name we can’t use instead of \text{num} in \text{small}? 

Small-step vs. big-step semantics

The evaluation derivations we’ve seen so far are called a \textbf{big-step semantics} because the derivation \( e \ # \ env2 \downarrow v \) explains the evaluation of \( e \) to \( v \) as one “big step” justified by the evaluation of its subexpressions.

An alternative way to express evaluation is a \textbf{small-step semantics} in which an expression is simplified to a value in a sequence of steps that simplifies subexpressions. You do this all the time when simplifying math expressions, and we can do it in Racket, too. E.g;

\[
\begin{align*}
(- (* (+ 2 3) 9) (/ 18 6)) \\
\Rightarrow (- (* 5 9) (/ 18 6)) \\
\Rightarrow (- 45 (/ 18 6)) \\
\Rightarrow (- 45 3) \\
\Rightarrow 42
\end{align*}
\]
Small-step semantics: intuition

Scan left to right to find the first redex (nonvalue subexpression that can be reduced to a value) and reduce it:

\[
\begin{align*}
& (- (*) (+ 2 3) 9) (/ 18 6)) \\
\Rightarrow & (- (*) 5 9) (/ 18 6)) \\
\Rightarrow & (- 45 (/ 18 6)) \\
\Rightarrow & (- 45 3) \\
\Rightarrow & 42
\end{align*}
\]

Small-step semantics: reduction rules

There are a small number of reduction rules for Racket. These specify the redexes of the language and how to reduce them.

The rules often require certain subparts of a redex to be values in order to be applicable.

\[
\begin{align*}
&id \Rightarrow v, \text{ where } id \rightarrow v \text{ in the current environment} & \Rightarrow v, \text{ where } v \text{ is the sum of } v1 \text{ and } v2 \text{ (addition)} \\
&(+) v1 v2 \Rightarrow v, \text{ where } v \text{ is the sum of } v1 \text{ and } v2 \text{ (addition)} \\
&\text{There are similar rules for other arithmetic operators} \\
&(\text{if } \#t e\_then e\_else) \Rightarrow e\_then \text{ (if true)} \\
&(\text{if } \#f e\_then e\_else) \Rightarrow e\_false \text{ (if false)} \\
&(\text{lambda } (id1 \ldots idn) e\_body) v1 \ldots vn \Rightarrow e\_body[v1 \ldots vn/id1 \ldots idn] \text{ (function call)}
\end{align*}
\]

* In a more formal approach, the notation would make the environment explicit. E.g., \( e#env \Rightarrow v \)

Small-step semantics: conditional example

\[
\begin{align*}
& (+ (\text{if } (< 1 2) (* 3 4) (/ 5 6)) 7) \\
\Rightarrow & (+ (\text{if } \#t (* 3 4) (/ 5 6)) 7) \\
\Rightarrow & (+ (* 3 4) 7) \\
\Rightarrow & (+ 12 7) \\
\Rightarrow & 19
\end{align*}
\]

Small-step semantics: errors as stuck expressions

Similar to big-step semantics, we model errors (dynamic type errors, divide by zero, etc.) in small-step semantics as expressions in which the evaluation process is stuck because no reduction rule is matched. For example

\[
\begin{align*}
& (- (*) (+ 2 3) #t) (/ 18 6)) \\
\Rightarrow & (- (*) 5 #t) (/ 18 6)) \\
&(\text{if } (= 2 (/ (+ 3 4) (- 5 5))) 8 9) \\
\Rightarrow & (\text{if } (= 2 (/ 7 (- 5 5))) 8 9) \\
\Rightarrow & (\text{if } (= 2 (/ 7 0))) 8 9)
\end{align*}
\]
**Small-step semantics: function example**

\[
\text{quad } 3
\]

\[
\Rightarrow ((\lambda (x) (\text{dbl} (\text{dbl } x))) 3)
\]

\[
\Rightarrow (\text{dbl} (\text{dbl } 3))
\]

\[
\Rightarrow ((\lambda (x) (* x 2)) (\text{dbl } 3))
\]

\[
\Rightarrow ((\lambda (x) (* x 2)) ((\lambda (x) (* x 2)) 3))
\]

\[
\Rightarrow ((\lambda (x) (* x 2)) (* 3 2))
\]

\[
\Rightarrow ((\lambda (x) (* x 2)) 6)
\]

\[
\Rightarrow (* 6 2)
\]

\[
\Rightarrow 12
\]

---

**Evaluation Contexts**

Although we will not do so here, it is possible to formalize exactly how to find the next redex in an expression using so-called **evaluation contexts**.

For example, in Racket, we never try to reduce an expression within the body of a lambda.

\[
((\lambda (x) (+ (* 4 5) x)) (+ 1 2))
\]

\[
\text{not this}
\]

\[
\text{this is the first redex}
\]

We’ll see later in the course that other choices are possible (and sensible).

---

**Small-step semantics: your turn**

Use small-step semantics to evaluate \((\text{small? } (\text{sqr } n))\)

Assume this is evaluated with respect to the same global environment used earlier.

---

**Recursion**

Recursion works as expected in Racket using the substitution model (both in big-step and small-step semantics).

There is no need for any special rules involving recursion!

The existing rules for definitions, functions, and conditionals explain everything.

\[
\text{(define pow}
\]

\[
(\lambda (\text{base exp})
\]

\[
(\text{if } (= \exp 0) 1 (* \text{base} (\text{pow} \text{base} (- \exp 1))))
\]

What is the value of \((\text{pow } 5 2)\)?
Recursion: your turn

Define and test the following recursive functions in Racket:

(fact n): Return the factorial of the nonnegative integer n.

(fib n): Return the nth Fibonacci number.

(sum-between lo hi): Return the sum of the integers between integers lo and hi (inclusive).

Syntactic sugar: function definitions

Syntactic sugar: simpler syntax for common pattern.
– Implemented via textual translation to existing features.
– i.e., not a new feature.

Example: Alternative function definition syntax in Racket:

(dbl x) (* x 2))
(quad x) (dbl (dbl x)))

Racket Operators are Actually Functions!

Surprise! In Racket, operations like (+ e1 e2), (< e1 e2) are, and (not e) are really just function applications!

There is an initial top-level environment that contains bindings like:
+ → addition function,
− → subtraction function,
* → multiplication function,
< → less-than function,
not → boolean negation function,
...