Functions in Racket

CS251 Programming Languages
Spring 2016, Lyn Turbak

Department of Computer Science
Wellesley College
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)))
```
**lambda** denotes a anonymous function

Syntax: \( \texttt{(lambda \ (id1 \ ... \ idn) \ e)} \)

- **lambda**: keyword that introduces an anonymous function (the function itself has no name, but you’re welcome to name it using `define`)
- **id1 ... idn**: any identifiers, known as the **parameters** of the function.
- **e**: any expression, known as the **body** of the function.
  It typically (but not always) uses the function parameters.

**Evaluation rule:**

- A **lambda** expression is just a value (like a number or boolean), so a **lambda** expression evaluates to itself!
- What about the function body expression? That’s not evaluated until later, when the function is **called**.
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 (\texttt{lambda expression}) to argument values? E.g.

\begin{itemize}
  \item \((\texttt{lambda (x) \ast x 2)} 3)\)
  \item \((\texttt{lambda (a b) \div (\ast a b 2)} 8 12)\)
\end{itemize}

We will explain function application using two models:

1. The \textbf{substitution model}: substitute the argument values for the parameter names in the function body.

2. The \textbf{environment model}: extend the environment of the function with bindings of the parameter names to the argument values.
Function application: substitution model

Example 1:

\(((\text{lambda } (x) (* x 2)) 3)\)

Substitute 3 for \(x\) in \(* x 2\) and evaluate the result:

\((* 3 2) \downarrow 6\) \(\text{(environment doesn’t matter in this case)}\)

Example 2:

\(((\text{lambda } (a b) (/ (+ a b) 2) 8 12))\)

Substitute 3 for \(x\) in \(* x 2\) and evaluate the result:

\((/ (+ 8 12) 2) \downarrow 10\) \(\text{(environment doesn’t matter in this case)}\)
Substitution notation

We will use the notation

\[ e[v_1, \ldots, v_n/id_1, \ldots, id_n] \]

to indicate the expression that results from substituting the values \( v_1, \ldots, v_n \) for the identifiers \( id_1, \ldots, id_n \) 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

\[
\begin{align*}
e_0 \# \text{env} \downarrow (\lambda (id_1 \ldots id_n) e_{body}) \\
e_1 \# \text{env} \downarrow v_1 \\
\vdots \\
en \# \text{env} \downarrow v_n \\
e_{body}[v_1 \ldots v_n/id_1 \ldots id_n] \# \text{env} \downarrow v_{body} \\
(e_0 e_1 \ldots en) \# \text{env} \downarrow v_{body}
\end{align*}
\]

Note: no need for function application frames like those you’ve seen in Python, Java, C, ...
Suppose $env2 = \text{dbl} \rightarrow (\lambda x \ (\times x \ 2))$, $\text{quad} \rightarrow (\lambda x \ ((\text{dbl} \ (\text{dbl} \ x))))$

<table>
<thead>
<tr>
<th>quad # env2</th>
<th>(lambda (x) (dbl (dbl x)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 # env2</td>
<td>3</td>
</tr>
<tr>
<td>dbl # env2</td>
<td>(lambda (x) (* x 2))</td>
</tr>
<tr>
<td></td>
<td>(lambda (x) (* x 2))</td>
</tr>
<tr>
<td>3 # env2</td>
<td>3</td>
</tr>
<tr>
<td>(* 3 2) # env2</td>
<td>6 (multiplication rule, subparts omitted)</td>
</tr>
<tr>
<td></td>
<td>(function call)</td>
</tr>
<tr>
<td>(dbl 3) # env2</td>
<td>6</td>
</tr>
<tr>
<td>(* 6 2) # env2</td>
<td>12 (multiplication rule, subparts omitted)</td>
</tr>
<tr>
<td></td>
<td>(function call)</td>
</tr>
<tr>
<td>(dbl (dbl 3)) # env2</td>
<td>12</td>
</tr>
<tr>
<td>(quad 3) # env2</td>
<td>12</td>
</tr>
</tbody>
</table>
Substitution model derivation: your turn

Suppose $\texttt{env3} = n \rightarrow 10,$

$\texttt{small?} \rightarrow (\text{lambda } (\text{num}) (\leq \text{num } n))$

$\texttt{sqr} \rightarrow (\text{lambda } (\text{n}) (\ast \text{n } n))$

Give an evaluation derivation for $(\texttt{small? } (\texttt{sqr } n)) \# \texttt{env3}$
Stepping back: name issues

Do the particular choices of function parameter names matter?

Is there any confusion caused by the fact that `dbl` and `quad` both use `x` as a parameter?

Are there any parameter names that we can’t change `x` to in `quad`?

In `(small? (sqr n))`, is there any confusion between the global parameter name `n` and parameter `n` in `sqr`?

Is there any parameter name we can’t use instead of `num` in `small`?
Small-step vs. big-step semantics

The evaluation derivations we’ve seen so far are called a **big-step semantics** because the derivation $e \# \text{env2} \downarrow v$ 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 **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:

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

\[
\text{id} \Rightarrow v, \text{ where id} \rightarrow v \text{ in the current environment}^* \text{ (varref)}
\]

\[
(+ v_1 v_2) \Rightarrow v, \text{ where } v \text{ is the sum of } v_1 \text{ and } v_2 \text{ (addition)}
\]

There are similar rules for other arithmetic operators

\[
(\text{if } \#t e_\text{\_then } e_\text{\_else}) \Rightarrow e_\text{\_then} \text{ (if true)}
\]

\[
(\text{if } \#f e_\text{\_then } e_\text{\_else}) \Rightarrow e_\text{\_false} \text{ (if false)}
\]

\[
((\text{lambda } (id_1 \ldots id_n) e_\text{\_body}) v_1 \ldots v_n)
\Rightarrow e_\text{\_body}[v_1 \ldots v_n/id_1 \ldots id_n] \text{ (function call)}
\]

* In a more formal approach, the notation would make the environment explicit.
E.g., \(e \# env \Rightarrow v\)
Small-step semantics: conditional example

\[
(+ \ (if \ (< \ 1 \ 2) \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7) \\
\Rightarrow \ (+ \ (if \ #t \ (* \ 3 \ 4) \ (/ \ 5 \ 6)) \ 7) \\
\Rightarrow \ (+ \ (* \ 3 \ 4) \ 7) \\
\Rightarrow \ (+ \ 12 \ 7) \\
\Rightarrow \ 19
\]
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

\[
(- (* (+ 2 3) #t) (/ 18 6))
\]

\[
\Rightarrow (- (* 5 #t) (/ 18 6))
\]

\[
(if (= 2 (/ (+ 3 4) (- 5 5))) 8 9)
\]

\[
\Rightarrow (if (= 2 (/ 7 (- 5 5))) 8 9)
\]

\[
\Rightarrow (if (= 2 (/ 7 0)) 8 9)
\]
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.

\[
(((\text{lambda} \ (x) \ (+ \ (\cdot \ 4 \ 5) \ x)) \ (+ \ 1 \ 2))
\]

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.

```
(define pow
  (lambda (base exp)
    (if (= exp 0)
      1
      (* base (pow base (- exp 1)))))
```

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

Define and test the following recursive functions in Racket:

\[
\text{(fact } n) : \text{ Return the factorial of the nonnegative integer } n
\]

\[
\text{(fib } n) : \text{ Return the } n\text{th Fibonacci number}
\]

\[
\text{(sum-between } lo \text{ hi)} : \text{return the sum of the integers between integers } lo \text{ and } hi \text{ (inclusive)}
\]
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:

```racket
(define (id_funName id1 ... idn) e_body)
```

desugars to

```racket
(define id_funName (lambda (id1 ... idn) e_body))
```

```racket
(define (dbl x) (* x 2))
(define (quad x) (dbl (dbl x)))
(define (pow base exp)
  (if (< exp 1)
    1
    (* base (pow base (- exp 1))))))
```
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,
...

4-23
Summary So Far

Racket declarations:
• definitions: \(\text{define } \text{id } e\)  

Racket expressions:
• conditionals: \(\text{if } e_{\text{test}} e_{\text{then}} e_{\text{else}}\)
• function values: \(\text{lambda } (id1 \ldots idn) e_{\text{body}}\)
• Function calls: \(e_{\text{rator}} e_{\text{rand1}} \ldots e_{\text{randn}}\)
  Note: arithmetic and relation operations are just function calls

What about?
• Assignment? Don’t need it!
• Loops? Don’t need them! Use tail recursion, coming soon.
• Data structures? Glue together two values with \text{cons} (next time)