Functions in Racket

Solutions

CS251 Programming Languages
Spring 2019, Lyn Turbak
Department of Computer Science
Wellesley College

Racket Functions

Functions: the most important building block in Racket (and 251)

- Functions/procedures/methods/subroutines abstract over computations
- Like Java methods & Python functions, Racket functions have arguments and result
- But no classes, this, return, etc.
- The most basic Racket function are anonymous functions specified with lambda

Examples:

\[
\begin{align*}
> & ((\text{lambda} \ (x) \ (* \ x \ 2)) \ 5) \\
& 10 \\
> & (\text{define} \ \text{dbl} \ (\text{lambda} \ (x) \ (* \ x \ 2))) \\
& (\text{dbl} \ 21) \\
& 42 \\
> & (\text{define} \ \text{quad} \ (\text{lambda} \ (x) \ (\text{dbl} \ (\text{dbl} \ x)))) \\
& (\text{quad} \ 10) \\
& 40 \\
> & (\text{define} \ \text{avg} \ (\text{lambda} \ (a \ b) \ (/ \ (+ \ a \ b) \ 2))) \\
& (\text{avg} \ 8 \ 12) \\
& 10
\end{align*}
\]

\[\text{lambda} \] denotes an anonymous function

Syntax: \((\text{lambda} \ (I_{d1} \ ... \ I_{dn}) \ E_{body})\)

- \text{lambda}: keyword that introduces an anonymous function
  (the function itself has no name, but you’re welcome to name it using define)
- \(I_{d1} \ ... \ I_{dn}\): any identifiers, known as the parameters of the function.
- \(E_{body}\): any expression, known as the body of the function.
  It typically (but not always) uses the function parameters.

Evaluation rule:

- A \text{lambda} expression is just a value (like a number or boolean),
  so a \text{lambda} expression evaluates to itself!
- What about the function body expression? That’s not evaluated until later,
  when the function is called. (Synonyms for called are applied and invoked.)

Function applications (calls, invocations)

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

Syntax: \((E_{0} \ E_{1} \ ... \ E_{n})\)

- A function application expression has no keyword. It is the only parenthesized expression that doesn’t begin with a keyword.
- \(E_{0}\): any expression, known as the rator of the function call
  (i.e., the function position).
- \(E_{1} \ ... \ E_{n}\): any expressions, known as the rands of the call
  (i.e., the argument positions).

Evaluation rule:

1. Evaluate \(E_{0} \ ... \ E_{n}\) in the current environment to values \(V_{0} \ ... \ V_{n}\).
2. If \(V_{0}\) is not a \text{lambda} expression, raise an error.
3. If \(V_{0}\) is a \text{lambda} expression, returned the result of applying it to the argument values \(V_{1} \ ... \ V_{n}\) (see following slides).
**Function application**

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

\[
\begin{align*}
((\text{lambda } (x) \ (\times x 2)) \ 3) \\
((\text{lambda } (a \ b) \ (/ \ (+ a \ b) 2)) \ 8 \ 12)
\end{align*}
\]

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.

**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:

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

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

**Avoid this common substitution bug**

Students sometimes **incorrectly** substitute the argument values into the parameter positions:

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

Makes no sense

When substituting argument values for parameters, **only the modified body should remain. The lambda and params disappear!**

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

\[
(/ (+ 8 12) 2)
\]
Small-step function application rule: substitution model

\[
\begin{align*}
    \text{Small-step substitution model semantics:} \\
    \text{substitution model} \\
    \text{function call (a.k.a. apply)}
\end{align*}
\]

\[
( \text{lambda (Id1 \ldots Idn) Ebody) V1 \ldots Vn ) \Rightarrow Ebody[V1, \ldots, Vn/Id1, \ldots, Idn] \text{[function call]}
\]

Note: could extend this with notion of “current environment”

Small-step function application rule: substitution model

\[
\begin{align*}
    \text{Small-step substitution model semantics:} \\
    \text{function call (a.k.a. apply)}
\end{align*}
\]

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

Small-step semantics: function example

\[
\begin{align*}
    \text{Suppose env2 = quad \rightarrow (lambda (x) (dbl (dbl x))),} \\
    \text{dbl \rightarrow (lambda (x) (* x 2))}
\end{align*}
\]

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

Stepping back: name issues

Answers

Do the particular choices of function parameter names matter?
No, the substitution model implies that as long as the parameter names are used consistently in the body and do not conflict with other names, they can be any names you like.

Is there any confusion caused by the fact that dbl and quad both use x as a parameter?
No, the substitution model shows that these two different x’s do not interact in any way.

Are there any parameter names that we can’t change x to in quad?
x can be any name except dbl; a dbl parameter would “capture” the references to the function dbl and change the meaning of the body.

In (small? (sqr n)), is there any confusion between the global variable named n and the parameter n in sqr?
No. The substitution model handles references to the the parameter n in the body of sqr and the [varref] rule handles references to the global variable n.

Is there any parameter name we can’t use instead of num in small?
Yes: changing the parameter num to n or <= would change the meaning of the function.
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.

\[
\begin{align*}
\text{(lambda (x) (+ (* 4 5) x)) ( + 1 2 )}
\end{align*}
\]

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

Substitution model derivation

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

\[
\begin{align*}
\text{quad} & \xrightarrow{\text{env2}} (\text{lambda} (x) (\text{dbl} (\text{dbl} x))) \\
3 & \xrightarrow{\text{env2}} 3 \\
\text{dbl} & \xrightarrow{\text{env2}} (\text{lambda} (x) (* x 2)) \\
\text{dbl} & \xrightarrow{\text{env2}} (\text{lambda} (x) (* x 2)) \\
3 & \xrightarrow{\text{env2}} 3 \\
(* 3 2) & \xrightarrow{\text{env2}} 6 \text{ [multiplication rule, subparts omitted]} \\
\text{dbl} & \xrightarrow{\text{env2}} 6 \\
(* 6 2) & \xrightarrow{\text{env2}} 12 \text{ [multiplication rule, subparts omitted]} \\
\text{dbl} & \xrightarrow{\text{env2}} 12 \\
3 & \xrightarrow{\text{env2}} 12 \\
\text{quad} & \xrightarrow{\text{env2}} 12
\end{align*}
\]

Big step function call rule: substitution model

\[
\begin{align*}
E0 \xrightarrow{\text{env}} (\text{lambda} (\text{Id1} \ldots \text{Idn}) \text{Ebody}) \\
E1 \xrightarrow{\text{env}} V1 \\
& \vdots \\
En \xrightarrow{\text{env}} Vn \\
\text{Ebody}[V1 \ldots Vn/\text{Id1} \ldots \text{Idn}] \xrightarrow{\text{env}} Vbody \text{ (function call)} \\
\langle E0 E1 \ldots En \rangle \xrightarrow{\text{env}} Vbody
\end{align*}
\]

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

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.

\[
\begin{align*}
\text{(define } \text{fact} & \text{ (lambda} (n) \\
& \quad (\text{if} (= n 0) \\
& \quad \quad 1 \\
& \quad \quad (* n (\text{fact} (- n 1))))))
\end{align*}
\]

What is the value of \((\text{fact 3})\)?
Small-step recursion derivation for (fact 4) [1]

Let's use the abbreviation \( \lambda_{\text{fact}} \) for the expression
\[
\lambda \ (n) \ (\text{if } (\ = \ n \ 0) \ 1 \ (* \ n \ (\text{fact} \ (- \ n \ 1))))
\]

\[
((\text{fact}) \ 4) \\
\Rightarrow ((\lambda_{\text{fact}} \ 4)) \\
\Rightarrow (\text{if } ((\ = \ 4 \ 0)) \ 1 \ (* \ 4 \ (\text{fact} \ (- \ 4 \ 1)))) \\
\Rightarrow ((\text{if } #\ n \ 1 \ (* \ 4 \ (\text{fact} \ (- \ 4 \ 1)))) \\
\Rightarrow (* \ 4 \ ((\text{fact}) \ (- \ 4 \ 1))) \\
\Rightarrow (* \ 4 \ (\lambda_{\text{fact}} \ ((- \ 4 \ 1)))) \\
\Rightarrow (* \ 4 \ ((\lambda_{\text{fact}} \ 3))) \\
\Rightarrow (* \ 4 \ (\text{if } ((\ = \ 3 \ 0)) \ 1 \ (* \ 3 \ (\text{fact} \ (- \ 3 \ 1)))) \\
\Rightarrow (* \ 4 \ ((\text{if } #\ n \ 1 \ (* \ 3 \ (\text{fact} \ (- \ 3 \ 1)))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ ((\text{fact}) \ (- \ 3 \ 1)))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (\lambda_{\text{fact}} \ ((- \ 3 \ 1)))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (\lambda_{\text{fact}} \ 2))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (\text{if } ((\ = \ 2 \ 0)) \ 1 \ (* \ 2 \ (\text{fact} \ (- \ 2 \ 1)))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ ((\text{if } #\ n \ 1 \ (* \ 2 \ (\text{fact} \ (- \ 2 \ 1)))))))
\]

… continued on next slide …

Abbreviating derivations with \( \Rightarrow^* \)

\( E_1 \Rightarrow^* E_2 \) means \( E_1 \) reduces to \( E_2 \) in zero or more steps

\[
((\text{fact}) \ 4) \\
\Rightarrow ((\lambda_{\text{fact}} \ 4)) \\
\Rightarrow (* \ 4 \ ((\lambda_{\text{fact}} \ 3))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ ((\lambda_{\text{fact}} \ 2)))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ ((\lambda_{\text{fact}} \ 1)))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 ((\lambda_{\text{fact}} \ 0)))))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ (* \ 1 \ 1)))))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (* \ 2 \ 1)))) \\
\Rightarrow (* \ 4 \ (* \ 3 \ (((2 \ 2)))) \\
\Rightarrow (* \ 4 \ (((3 \ 2)))) \\
\Rightarrow ((* \ 4 \ 6)) \\
\Rightarrow 24
\]

Recursion: your turn:

Show an abbreviated small-step evaluation of \((\text{pow} \ 5 \ 3)\) where \text{pow} is defined as:

\[
\text{(define pow}
\begin{align*}
\text{ (lambda} & \text{ (base exp)} \\
\text{ (if } & (\ = \ \text{exp} \ 0) \\
& 1 \\
& (* \ \text{base} \ (\text{pow} \ \text{base} \ (- \ \text{exp} \ 1))))
\end{align*}
\]

How many multiplications are performed in
\[(\text{pow} \ 2 \ 10)? \]
\[(\text{pow} \ 2 \ 100)? \]
\[(\text{pow} \ 2 \ 1000)?

What is the stack depth (\# pending multiplies) in these cases?
Recursion: your turn Answers
Show an abbreviated small-step evaluation of \((\text{pow} \ 5 \ 3)\):

\[
\begin{align*}
((\text{pow}) \ 5 \ 3) & \Rightarrow ((\lambda \text{pow} \ 5 \ 3)) \\
& \Rightarrow (* \ 5 ((\lambda \text{pow} \ 5 \ 2))) \\
& \Rightarrow (* \ 5 (* \ 5 ((\lambda \text{pow} \ 5 \ 1)))) \\
& \Rightarrow (* \ 5 (* \ 5 (* \ 5 ((\lambda \text{pow} \ 5 \ 0)))))) \\
& \Rightarrow (* \ 5 (* \ 5 ((* \ 5 1)))) \\
& \Rightarrow (((* \ 5 25)) \\
& \Rightarrow 125
\end{align*}
\]

<table>
<thead>
<tr>
<th>Call</th>
<th># multiplications</th>
<th>stack depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pow 2 10)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>(pow 2 100)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>(pow 2 1000)</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

linear in \(\exp\), i.e. \(O(\exp)\)

---

Recursion: your turn 2 Answers
Show an abbreviated small-step evaluation of \((\text{fast-pow} \ 2 \ 10)\) with the following definitions:

\[
\begin{align*}
\text{(define square)} & \quad \text{(lambda) (n) (* n n))} \\
\text{(define even?)} & \quad \text{(lambda) (n) (= 0 (remainder n 2)))} \\
\text{(define fast-pow)} & \quad \text{(lambda) (base exp) } \\
& \hspace{1cm} \text{(if (= exp 0)} 1 \\
& \hspace{2.5cm} \text{(if (even? exp)} \\
& \hspace{4cm} \text{(fast-pow (square base) (/ exp 2))} \\
& \hspace{5.5cm} (* base (fast-pow base (- exp 1))))))
\end{align*}
\]

How many multiplications (including in \text{square}) are performed in
\(\text{fast-pow} \ 2 \ 10)\?
\(\text{fast-pow} \ 2 \ 100)\?
\(\text{fast-pow} \ 2 \ 1000)\?

What is the stack depth (# pending multiplies) in these cases?

---

Tree Recursion: Fibonacci

Show an abbreviated small-step evaluation of \((\text{fast-pow} \ 2 \ 10)\):

\[
\begin{align*}
((\text{fast-pow}) \ 2 \ 10) & \Rightarrow ((\lambda \text{fast-pow} \ 2 \ 10)) \\
& \Rightarrow (* \ 4 ((\lambda \text{fast-pow} \ 4 \ 5))) \\
& \Rightarrow (* \ 4 ((\lambda \text{fast-pow} \ 16 \ 2))) \\
& \Rightarrow (* \ 4 ((\lambda \text{fast-pow} \ 256 \ 1))) \\
& \Rightarrow (* \ 4 (* \ 256 ((\lambda \text{fast-pow} \ 256 \ 0)))) \\
& \Rightarrow (* \ 4 (* \ 256 1))) \\
& \Rightarrow (((* \ 4 256)) \\
& \Rightarrow 1024
\end{align*}
\]

number of 1s in binary rep

\[1 + \text{number of bits in binary rep}\]

<table>
<thead>
<tr>
<th>Call</th>
<th>(\exp) in binary</th>
<th># multiplications</th>
<th>stack depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pow 2 10)</td>
<td>1010</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(pow 2 100)</td>
<td>11000100</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>(pow 2 1000)</td>
<td>1111010000</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

logarithmic in \(\exp\), i.e. \(O(\lg(\exp))\)

---

How many additions as a function of \(n\)? \(O(2^n) = \text{exponential}\)

What is the stack depth as a function of \(n\)? \(O(n) = \text{linear}\)
Fibonacci with small-step semantics

Suppose the global env contains binding \( \text{fib} \mapsto \lambda_\text{fib} \), where \( \lambda_\text{fib} \) abbreviates
\[
(\lambda (n) (if (<= n 1) n (+ (\text{fib} (- n 1)) (\text{fib} (- n 2)))))
\]

\[
\{\text{fib} \ 4\} \\
\Rightarrow \{\lambda_\text{fib} \ 4\} \\
\Rightarrow* (+ \{\lambda_\text{fib} \ 3\} \ (\text{fib} (- 4 2))) \\
\Rightarrow* (+ (+ \{\lambda_\text{fib} \ 2\} \ (\text{fib} (- 3 2))) \ (\text{fib} (- 4 2))) \\
\Rightarrow* (+ (+ (+ \{\lambda_\text{fib} \ 1\} \ (\text{fib} (- 2 2))) \ (\text{fib} (- 3 2))) \ (\text{fib} (- 4 2))) \\
\Rightarrow* (+ (+ (+ 1 \{\lambda_\text{fib} \ 0\}) \ (\text{fib} (- 2 2))) \ (\text{fib} (- 3 2))) \\
\Rightarrow* (+ (+ 1 \{\lambda_\text{fib} \ 1\}) \ (\text{fib} (- 4 2))) \\
\Rightarrow* (+ \{+ 1 \} \ (\text{fib} (- 4 2))) \\
\Rightarrow* (+ 2 \{\lambda_\text{fib} \ 2\}) \\
\Rightarrow* (+ 2 (+ \{\lambda_\text{fib} \ 1\} \ (\text{fib} (- 2 2)))) \\
\Rightarrow* (+ 2 (+ 1 \{\lambda_\text{fib} \ 0\})) \\
\Rightarrow* (+ 2 (+ 1 0)) \\
\Rightarrow* (\{+ 2 \}) \\
\Rightarrow* (\{+ 2 \}) \\
\Rightarrow 3
\]

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:

\[
\begin{align*}
(\text{define} \ (\text{Id}_\text{funName} \ \text{Id}_1 \ \ldots \ \text{Id}_n) \ E_{\text{body}}) \\
\end{align*}
\]
desugars to

\[
(\text{define} \ \text{Id}_\text{funName} \ \text{lambda} \ (\text{Id}_1 \ \ldots \ \text{Id}_n) \ E_{\text{body}}))
\]

Answers

- How many additions? 4
- What is the stack depth? 3 (# pending mults)

Racket Operators are Actually Functions!

Surprise! In Racket, operations like \((+ \ e1 \ e2)\), \((< \ e1 \ e2)\), and \((\text{not} \ e)\) are really just function calls!

There is an initial top-level environment that contains bindings for built-in functions like:
- \(+ \mapsto \text{addition function},\)
- \(- \mapsto \text{subtraction function},\)
- \(* \mapsto \text{multiplication function},\)
- \(< \mapsto \text{less-than function},\)
- \(\text{not} \mapsto \text{boolean negation function},\)

(where some built-in functions can do special primitive things that regular users normally can’t do --- e.g. add two numbers)

Racket Language Summary So Far

**Racket declarations:**
- definitions: \((\text{define} \ \text{Id} \ E)\)

**Racket expressions** (this is most of the kernel language!)
- literal values (numbers, boolean, strings): e.g. 251, 3.141, #t, "Lyn"
- variable references: e.g., \(x\), fact, positive?, \(\text{fib}_n\)
- conditionals: \((\text{if} \ \text{E}_{\text{test}} \ \text{E}_{\text{then}} \ \text{E}_{\text{else}})\)
- function values: \((\text{lambda} \ (\text{Id}_1 \ \ldots \ \text{Id}_n) \ E_{\text{body}})\)
- function calls: \((\text{E}_{\text{operator}} \ \text{E}_{\text{rand}1} \ \ldots \ \text{E}_{\text{randn}})\)

**Note**: arithmetic and relational operations are really 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 cons (next time).
  - Can even implement data structures with lambda! (See Wacky Lists on PS4, Functional Sets on PS8)
  - Motto: lambda is all you need!