CS 251 Part 1: How to Program

Defining Racket: Expressions and Bindings via the meta-language of PL definitions

Topics / Goals

1. Basic language forms and evaluation model.
2. Foundations of defining syntax and semantics.
   - Informal descriptions (English)
   - Formal descriptions (meta-language):
     • Grammars for syntax.
     • Judgments, inference rules, and derivations for big-step operational semantics.
3. Learn Racket. (an opinionated subset)
   - Not always idiomatic or the full story. Setup for transition to Standard ML.

From AI to language-oriented programming

LISP: List Processing language, 1950s-60s, MIT AI Lab.
Advice Taker: represent logic as data, not just as a program.
Metaprogramming and programs as data:
  • Symbolic computation (not just number crunching)
  • Programs that manipulate logic (and run it too)

Scheme: child of Lisp, 1970s, MIT AI Lab.
Still motivated by AI applications, became more "functional" than Lisp.
Important design changes/additions/cleanup:
  • simpler naming and function treatment
  • lexical scope
  • first-class continuations
  • tail-call optimization, ...

Racket: child of Scheme, 1990s-2010s, PLT group.
Revisions to Scheme for:
  • Rapid implementation of new languages.
  • Education.
  • Became Racket in 2010.
Defining Racket

To define each new language feature:
- Define its **syntax**. How is it written?
- Define its **dynamic semantics** as evaluation rules. How is it evaluated?

Features
1. **Expressions**
   - A few today, more to come.
2. **Bindings**
3. That's all!
   - A couple more advanced features later.

Values

Expressions that cannot be evaluated further.

**Syntax:**
Numbers: 251 240 301
Booleans: #t #f ...

**Evaluation:**
Values evaluate to themselves.

Addition expression

**Syntax:**  (+  e1  e2)
- Parentheses required: no extras, no omissions.
- `e1` and `e2` stand in for *any* expressions.
- Note *prefix* notation.

**Examples:**
(+ 251 240)  (+  (+  251  240)  301)
(+  #t  251)
Addition expression

Syntax: \((+ \ e_1 \ e_2)\)

Evaluation:
1. Evaluate \(e_1\) to a value \(v_1\).
2. Evaluate \(e_2\) to a value \(v_2\).
3. Return the arithmetic sum of \(v_1 + v_2\).

Dynamic type checking

Evaluation:
1. Evaluate \(e_1\) to a value \(v_1\).
2. Evaluate \(e_2\) to a value \(v_2\).
3. If \(v_1\) and \(v_2\) are numbers then return the arithmetic sum of \(v_1 + v_2\).
   Otherwise there is a type error.

The language of languages

Because it pays to be precise.

Syntax:
- Formal grammar notation
- Conventions for writing syntax patterns

A grammar formalizes syntax.

"An expression \(e\) is one of:
- Any value \(v\)
- Any addition expression \((+ \ e \ e)\) of any two expressions"
Racket syntax so far

Expressions
\[ e ::= v \]
\[ \quad \mid ( + e e ) \]

Literal Values
\[ v ::= \#f \mid \#t \mid n \]

Number values
\[ n ::= 0 \mid 1 \mid 2 \mid ... \]

Notation conventions

Outside the grammar:

- Use of a non-terminal symbol, such as \( e \), in syntax examples and evaluation rules means any expression matching one of the productions of \( e \) in the grammar.

- Two uses of \( e \) in the same context are aliases; they mean the same expression.

- Subscripts (or suffixes) distinguish separate instances of a single non-terminal, e.g., \( e_1, e_2, \ldots, e_n \) or \( e_1, e_2, \ldots, e_n \).

The language of languages

Because it pays to be precise.

Syntax:
- Formal grammar notation
- Conventions for writing syntax patterns

Semantics:
- Judgments:
  - formal assertions, like functions
- Inference rules:
  - implications between judgments, like cases of functions
- Derivations:
  - deductions based on rules, like applying functions

Judgments and rules formalize semantics.

Judgment \( e \Downarrow v \)
means "expression \( e \) evaluates to value \( v \)."

It is implemented by inference rules for different cases:

- value rule:
  \[ v \Downarrow v \]

- addition rule:
  \[ \text{if } e_1 \Downarrow n_1 \]
  \[ \text{and } e_2 \Downarrow n_2 \]
  \[ \text{and } n \text{ is the arithmetic sum of } n_1 \text{ and } n_2 \]
  \[ \text{then } (+ e_1 e_2) \Downarrow n \]

...
**Evaluation derivations**

An evaluation derivation is a "proof" that an expression evaluates to a value using the evaluation rules.

\[
(+ 3 (+ 5 4)) \Downarrow 12
\]

by the addition rule because:

- \(3 \Downarrow 3\) by the value rule, where 3 is a number
- \(+ 5 4 \Downarrow 9\) by the addition rule, where 9 is a number, because:
  - \(5 \Downarrow 5\) by the value rule, where 5 is a number
  - \(4 \Downarrow 4\) by the value rule, where 4 is a number
  - and 9 is the sum of 5 and 4
- and 12 is the sum of 3 and 9.

**Errors are modeled by “stuck” derivations.**

How to evaluate
\[
(+ \#t (+ 5 4))?
\]

Stuck. Can’t apply the [add] rule because there is no rule that allows \#t to evaluate to a number.

How to evaluate
\[
(+ (+ 1 2) (+ 5 \#f))?
\]

Stuck. Can’t apply the [add] rule because there is no rule that allows \#f to evaluate to a number.
Other number expressions

Similar syntax and evaluation for:

+ - * / quotient < > <= >= =

Some small differences.

Build syntax and evaluation rules for:

quotient and >

Conditional if expressions

Syntax: \((if\ e_1\ e_2\ e_3)\)

Evaluation:

1. Evaluate \(e_1\) to a value \(v_1\).
2. If \(v_1\) is not the value \(#f\) then evaluate \(e_2\) and return the result otherwise evaluate \(e_3\) and return the result

Evaluation rules for if expressions.

\[
\begin{align*}
e_1 & \downarrow v_1 \\
e_2 & \downarrow v_2 \\
\text{\textbf{v1 is not \#f}} & \quad [\text{if nonfalse}] \\
(if\ e_1\ e_2\ e_3) & \downarrow v_2
\end{align*}
\]

\[
\begin{align*}
e_1 & \downarrow \#f \\
e_3 & \downarrow v_3 \\
\text{\textbf{e2 is not evaluated!}} \\
(if\ e_1\ e_2\ e_3) & \downarrow v_3
\end{align*}
\]

Notice: at most one of these rules can have its premises satisfied!

if expressions

if expressions are expressions.

Racket has no "statements!"

\[
\begin{align*}
(if\ (<\ 9\ (-\ 251\ 240))) \\
(+\ 4\ (*\ 3\ 2)) \\
(+\ 4\ (*\ 3\ 3))
\end{align*}
\]

\[
\begin{align*}
(if\ (if\ (<\ 1\ 2)\ (>\ 4\ 3)\ (>\ 5\ 6)) \\
(+\ 7\ 8) \\
(*\ 9\ 10)
\end{align*}
\]
**if expression evaluation**

Will either of these expressions result in an error (stuck derivation) when evaluated?

\[(if \ (>\ 251\ 240)\ 251\ (/\ 251\ 0))\]

\[(if\ \#f\ (+\ \#t\ 251)\ 251)\]

**Variables and environments**

How do we know the value of a variable?

\[(define\ x\ (+\ 1\ 2))\]

\[(define\ y\ (*\ 4\ x))\]

\[(define\ diff\ (-\ y\ x))\]

\[(define\ test\ (<\ x\ diff))\]

\[(if\ test\ (+\ (*\ x\ y)\ diff)\ 17)\]

Keep a **dynamic environment**:
- A sequence of **bindings** mapping **identifier** (variable name) to **value**.
- “Context” for evaluation, used in evaluation rules.

**Language design choice: if semantics**

\[
\begin{align*}
\text{v1} & \quad \text{not required to be a Boolean value} \\
\text{if} & \quad \text{nonfalse} \\
\text{if} & \quad \text{true} \\
\text{if} \quad & \text{false}
\end{align*}
\]

**More Racket syntax**

**Bindings**

\[b ::= (define\ x\ e)\]

**Expressions**

\[e ::= v \mid x \mid ( + e e ) \mid ... \mid (if\ e\ e\ e)\]

**Literal Values (booleans, numbers)**

\[v ::= \#f \mid \#t \mid n\]

**Identifiers** (variable names)

\[x\] (see valid identifier explanation)
Dynamic environments

Grammar for environment notation:

\[ E ::= . \quad \text{ (empty environment) } \]
\[ | x \mapsto v, E \quad \text{ (one binding, rest of environment) } \]

where:
- \( x \) is any legal variable identifier
- \( v \) is any value

Concrete example:

\[ \text{num} \mapsto 17, \text{absZero} \mapsto -273, \text{true} \mapsto \#t, \ . \]

Abstract example:

\[ x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n, \ . \]

Variable reference expressions

Syntax:

\[ x \]

\( x \) is any identifier

Evaluation rule:

Look up \( x \) in the current environment, \( E \), and return the value, \( v \), to which \( x \) is bound. If there is no binding for \( x \), a name error occurs.

\[ E \vdash x \downarrow v \]

Expression evaluation rules must pass the environment.

\[ E \vdash v \downarrow v \quad \text{[value]} \]
\[ E \vdash e_1 \downarrow n_1 \]
\[ E \vdash e_2 \downarrow n_2 \]
\[ n = n_1 + n_2 \quad \text{[add]} \]
\[ E \vdash x \downarrow v \]
\[ E \vdash e_1 \downarrow v_1 \]
\[ E \vdash e_2 \downarrow v_2 \]
\[ v_1 \text{ is not } \#f \quad \text{[if nonfalse]} \]
\[ E \vdash \text{(if } e_1 \text{ e_2 e_3) } \downarrow v_2 \]
\[ E \vdash e_1 \downarrow \#f \]
\[ E \vdash e_3 \downarrow v_3 \]
\[ E \vdash \text{(if } e_1 \text{ e_2 e_3) } \downarrow v_3 \quad \text{[if false]} \]

Derivation with environments

Let \( E = \text{test} \mapsto \#t, \text{diff} \mapsto 9, y \mapsto 12, x \mapsto 3 \)

\[ E \vdash \text{test} \downarrow \#t \quad \text{[var]} \]
\[ E \vdash x \downarrow 3 \quad \text{[var]} \]
\[ E \vdash 5 \downarrow 5 \quad \text{[value]} \quad \text{[add]} \]
\[ E \vdash (*) \text{x} \downarrow 15 \quad \text{[mult]} \]
\[ E \vdash \text{diff} \downarrow 9 \quad \text{[var]} \]
\[ E \vdash (+ (* x 5) \text{diff}) \downarrow 24 \quad \text{[add]} \]
\[ E \vdash \text{(if test} (+ (* x 5) \text{diff) 17) } \downarrow 24 \quad \text{[if nonfalse]} \]
define bindings

Syntax:  (define \textit{x e})

define is a keyword, \textit{x} is any identifier, \textit{e} is any expression

Evaluation rule:
1. Under the current environment, \textit{E}, evaluate \textit{e} to a value \textit{v}.
2. Produce a new environment, \textit{E}', by extending the current environment, \textit{E}, with the binding \textit{x} \mapsto \textit{v}.

\[
\begin{align*}
\text{E} & \vdash \text{b} \Downarrow \text{E}' \\
\text{E} & \vdash \text{e} \Downarrow \text{v} \\
\text{E}' & = \text{x} \mapsto \text{v}, \text{E} \\
\text{E} & \vdash (\text{define x e}) \Downarrow \text{E}'
\end{align*}
\]

Environment example

\[
\begin{align*}
\text{E0} & = . \\
(\text{define x (+ 1 2)}) & ; \text{E1} = \text{x} \mapsto 3, . & \text{abbreviated x} \mapsto 3; \text{write as x --> 3 in text} \\
(\text{define y (* 4 x)}) & ; \text{E2} = \text{y} \mapsto 12, \text{x} \mapsto 3 & \text{(most recent binding first)} \\
(\text{define diff (- y x)}) & ; \text{E3} = \text{diff} \mapsto 9, \text{y} \mapsto 12, \text{x} \mapsto 3 \\
(\text{define test (< x diff)}) & ; \text{E4} = \text{test} \mapsto \text{#t}, \text{diff} \mapsto 9, \text{y} \mapsto 12, \text{x} \mapsto 3 \\
(\text{if test (+ (* 5 diff) 17)}) & ; \text{(environment here is still E4)}
\end{align*}
\]

Racket identifiers

Most character sequences are allowed as identifiers, except:
- those containing
  - whitespace
  - special characters ( ) [ ]lbracerbrace ’ “ , ‘ ; # \ \\
- identifiers syntactically indistinguishable from numbers (e.g., -45)

Fair game: ! @ $ % ^ & * . - + _ : < = > ? / \\
- myLongName, my_long__name, my-long-name \\
- is_a+b<c*d-e? \\
- 64 bits

Why are other languages less liberal with legal identifiers?

Big-step vs. small-step semantics

We defined a big-step operational semantics: evaluate "all at once"

A small-step operational semantics defines step by step evaluation:

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

A small-step view helps define evaluation orders later in 251.