CS 251 Part 1: How to Program

Defining Racket: Expressions and Bindings

+define.rkt

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: \((+ e_1 e_2)\)
- Parentheses required: no extras, no omissions.
- \(e_1\) and \(e_2\) 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 \(v_1 + v_2\).

Note recursive structure!

Dynamic type checking

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. 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
\[ e ::= v \]
\[ \quad | \quad ( + e e ) \]

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

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

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

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:

\[ \text{value rule:} \quad \frac{}{v \downarrow v} \]

\[ \text{addition rule:} \]
\[ \frac{e_1 \downarrow n_1 \quad \text{and} \quad e_2 \downarrow n_2}{( + e_1 e_2 ) \downarrow n} \quad \text{where} \quad n = n_1 + n_2 \]

...
**Inference rules**

Axiom (no premises)

Bar is optional for axioms.

If all premises hold then the conclusion holds.

- Conclusion
- Premises
- Rule name

**Expressions, Bindings, Meta-language**

- \( v \downarrow v \) [value]
- \( e_1 \downarrow n_1 \)
- \( e_2 \downarrow n_2 \)
- \( n = n_1 + n_2 \) [add]

\[ (+e_1 e_2) \downarrow n \]

**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
- and \( (+ 5 4) \downarrow 9 \) by the addition rule, where 9 is a number, because:
  - \( 5 \downarrow 5 \) by the value rule, where 5 is a number
  - and \( 4 \downarrow 4 \) 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.

**Evaluation derivations**

Rules defining the evaluation judgment

\[ e \downarrow v \quad v \downarrow v \] [value]

- \( e_1 \downarrow n_1 \)
- \( e_2 \downarrow n_2 \)
- \( n = n_1 + n_2 \) [add]

\[ (+e_1 e_2) \downarrow n \]

**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 \textit{if} expressions

Syntax: \((\text{if } e_1 e_2 e_3)\)

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 \textit{if} expressions.

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

\[
\begin{align*}
& e_1 \downarrow \#f \\
& e_3 \downarrow v_3 \\
& [\text{if false}] \\
& (\text{if } e_1 e_2 e_3) \downarrow v_3
\end{align*}
\]

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

\textit{if} expressions are \textit{expressions}.

Racket has no "statements!"

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

\[
\begin{align*}
& (+ 4 (* 3 (\text{if } (< 9 (\# - 251 240)) 2 3))) \\
& (\text{if } (\text{if } (< 1 2) (> 4 3) (> 5 6)) \\
& \quad (+ 7 8) \\
& \quad (* 9 10))
\end{align*}
\]
**if expression evaluation**

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

\[
\text{(if (> 251 240) 251 (/ 251 0))}
\]

\[
\text{(if #f (+ #t 251) 251)}
\]

**Language design choice: if semantics**

- **v1 not required to be a Boolean value**

\[
\begin{align*}
&\text{\( e_1 \downarrow v_1 \)} \\
&\text{\( e_2 \downarrow v_2 \)} \\
&\text{\( v_1 \text{ is not } \#f \)} \\
\end{align*}
\]

\[
\text{(if \( e_1 \ e_2 \ e_3 \) \downarrow v_2)}
\]

- **Alternative design**

\[
\begin{align*}
&\text{\( e_1 \downarrow \#t \)} \\
&\text{\( e_2 \downarrow v_2 \)} \\
\end{align*}
\]

\[
\text{(if \( e_1 \ e_2 \ e_3 \) \downarrow v_2)}
\]

**Variables and environments**

How do we know the value of a variable?

- \(\text{(define x (+ 1 2))}\)
- \(\text{(define y (* 4 x))}\)
- \(\text{(define diff (- y x))}\)
- \(\text{(define test (< x diff))}\)
- \(\text{(if test (+ (* x y) diff) 17)}\)

Keep a **dynamic environment:**

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

**More Racket syntax**

**Binders**

\[
\begin{array}{c}
\text{\( b ::= (\text{define x e)} \)}
\end{array}
\]

**Expressions**

\[
\begin{array}{c}
\text{\( e ::= v \ | \ x \ | \ ( + e e ) \ | \ ... \ | \ (\text{if} \ e \ e \ e) \)}
\end{array}
\]

**Literal Values (booleans, numbers)**

\[
\begin{array}{c}
\text{\( v ::= \#f \ | \ \#t \ | \ n \)}
\end{array}
\]

**Identifiers (variable names)**

\[
\begin{array}{c}
\text{\( x \) (see valid identifier explanation)}
\end{array}
\]
Dynamic environments

Grammar for environment notation:
\[ E ::= \] (empty environment)
\[ \mid x \mapsto v, E \] (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.

Expression evaluation rules must pass the environment.

\[ E \vdash x \downarrow v \]

\[ E \vdash v \downarrow v \quad [\text{value}] \]
\[ E \vdash e_1 \downarrow n_1 \]
\[ E \vdash e_2 \downarrow n_2 \]
\[ n = n_1 + n_2 \quad [\text{add}] \]
\[ E \vdash (+ e_1 e_2) \downarrow n \]
\[ E \vdash test \downarrow \#t \quad [\text{var}] \]
\[ E \vdash x \downarrow 3 \quad [\text{var}] \]
\[ E \vdash 5 \downarrow 5 \quad [\text{value}] \]
\[ 15 = 3 \times 5 \quad [\text{mult}] \]
\[ E \vdash (* x 5) \downarrow 15 \]
\[ E \vdash \text{diff} \downarrow 9 \quad [\text{var}] \]
\[ 24 = 15 + 9 \quad [\text{add}] \]
\[ E \vdash (+ (* x 5) \text{diff}) \downarrow 24 \quad [\text{if nonfalse}] \]
\[ E \vdash (\text{if} test (+ (* x 5) \text{diff}) 17) \downarrow 24 \quad [\text{if nonfalse}] \]

Expression evaluation rules

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.

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

Abstract example:
\[ x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n, \]

Derivation with environments

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

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

Syntax:  \( (\text{define } x \ e) \)

define is a keyword, \( x \) is any identifier, \( e \) is any expression

Evaluation rule:
1. Under the current environment, \( E \), evaluate \( e \) to a value \( v \).
2. Produce a new environment, \( E' \), by extending the current environment, \( E \), with the binding \( x \leadsto v \).

\[
\begin{align*}
E \triangleright b \downarrow E' & \quad \text{E \triangleright e \downarrow v} \\
E' = x \leadsto v, E & \quad \text{[define]} \\
E \triangleright (\text{define } x \ e) \downarrow E'
\end{align*}
\]

Racket identifiers

Most character sequences are allowed as identifiers, except:
- those containing
  - whitespace
  - special characters ( )[]{}",:;\|
- identifiers syntactically indistinguishable from numbers (e.g., -45)

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

Why are other languages less liberal with legal identifiers?

Environment example (define.rkt)

; E0 = .
(define x (+ 1 2)) ; E1 = x \rightarrow 3, . (abbreviated x \rightarrow\rightarrow 3; write x \rightarrow\rightarrow\rightarrow 3 in text)
(define y (* 4 x)) ; E2 = y \rightarrow 12, x \rightarrow 3 (most recent binding first)
(define diff (- y x)) ; E3 = diff \rightarrow 9, y \rightarrow 12, x \rightarrow 3
(define test (< x diff)) ; E4 = test \rightarrow #t, diff \rightarrow 9, y \rightarrow 12, x \rightarrow 3
(if test (+ (* x 5) diff) 17) ; (environment here is still E4)

Big-step vs. small-step semantics

We defined a \textbf{big-step operational semantics}: evaluate "all at once"

A \textbf{small-step operational semantics} defines step by step evaluation:

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

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