CS 251 Part 1: How to Program

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.

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

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 \(v_1 + v_2\).

Note recursive structure!

Expressions, Bindings, Meta-language

The language of languages

Because it pays to be precise.

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

A grammar formalizes syntax.

Non-terminal \(e\) has 2 productions, separated by "|".

"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., \( e_1, e_2, ..., e_n \) or \( e_1, e_2, ..., e_n \).

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:

\[ v \Downarrow v \] [value]

addition rule:

\[
\begin{align*}
\text{if } & e_1 \Downarrow n_1 \\
\text{and } & e_2 \Downarrow n_2 \\
\text{then } & (+ e_1 e_2) \Downarrow n \\
& \text{where } n = n_1 + n_2
\end{align*}
\] [add]
### Inference rules

- **Axiom (no premises)**
  - Bar is optional for axioms.

- **Conclusion**
  - \( \downarrow \)

- **Premises**
  - \( \downarrow \)

- **Rule name**
  - [add]

- **Expressions, Bindings, Meta-language**
  - \( v \)
  - \( n \)

- **Inference rule notation and meaning**

### 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 derivation examples

**How to evaluate** 

\[(+ \ #t \ (+ \ 5 \ 4))\]?

**How to evaluate**

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

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

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

**Number values, not just any values.** Models dynamic type checking.

- \(v\) is the arithmetic sum of the numbers \(n_1\) and \(n_2\) (not Racket syntax)

- \(n = n_1 + n_2\)

- \(e_1 \downarrow n_1\)
- \(e_2 \downarrow n_2\)

- \(n = n_1 + n_2\)

- \((+ e_1 e_2) \downarrow n\)

**Rules defining the evaluation judgment**

Adding vertical bars helps clarify nesting.

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

- \(3 \downarrow 3\) [value]
- \(5 \downarrow 5\) [value]
- \(4 \downarrow 4\) [value]
- \(9 = 5 + 4\) [add]
- \((+ 5 4) \downarrow 9\) [add]
- \(12 = 3 + 9\) [add]
- \((+ 3 (+ 5 4)) \downarrow 12\) [add]
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

\textit{if} expressions are \textit{expressions}. Racket has no "statements!"

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

\[
\begin{align*}
(\text{if } (\text{if } (<\ 9\ (-\ 251\ 240))\ 2\ 3))\)
& (\text{if } (<\ 1\ 2)\ (>\ 4\ 3)\ (>\ 5\ 6))
\end{align*}
\]

\[
\begin{align*}
& (+\ 7\ 8) \\
& (*\ 9\ 10)
\end{align*}
\]
**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**

\[
\begin{align*}
\text{e1} & \downarrow v1 \\
\text{e2} & \downarrow v2 \\
\text{v1 is not } \#f
\end{align*}
\]

\[
(\text{if e1 e2 e3} \downarrow v2)
\]

- Alternative design

<table>
<thead>
<tr>
<th>e1</th>
<th>#t</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>v2</td>
</tr>
</tbody>
</table>

\[
(\text{if e1 e2 e3} \downarrow v2)
\]

- [if nonfalse]

<table>
<thead>
<tr>
<th>e1</th>
<th>#f</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>v2</td>
</tr>
</tbody>
</table>

| e3 | v2 |

\[
(\text{if e1 e2 e3} \downarrow v2)
\]

- [if true]

---

**Variables and environments**

How do we know the value of a variable?

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

Keep a *dynamic environment*:

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

---

**More Racket syntax**

### Bindings

\[
\text{b ::= (define x e)}
\]

### Expressions

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

### Literal Values (booleans, numbers)

\[
\text{v ::= \#f | \#t | n}
\]

### Identifiers (variable names)

\[
\text{x (see valid identifier explanation)}
\]
## 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.

\[ E \vdash e \Downarrow v \]

Search from most to least recent (left to right).

## Expression evaluation rules must pass the environment.

\[ E \vdash v \Downarrow v \quad \text{[value]} \]
\[ E \vdash e_1 \Downarrow n_1 \]
\[ E \vdash e_2 \Downarrow n_2 \]
\[ n = n_1 + n_2 \]

**[add]**

\[ E \vdash x \Downarrow v \]

\[ E \vdash e_1 \Downarrow v_1 \]
\[ E \vdash e_2 \Downarrow v_2 \]

\( v_1 \) is not \( \#f \)

**[if nonfalse]**

\[ E \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow v_2 \]

\[ E \vdash e_1 \Downarrow \#f \]
\[ E \vdash e_3 \Downarrow v_3 \]

**[if false]**

\[ E \vdash (\text{if } e_1 \ e_2 \ e_3) \Downarrow v_3 \]

## Derivation with environments

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

\[ E \vdash \text{test} \Downarrow \#t \quad \text{[var]} \]
\[ E \vdash x \Downarrow 3 \quad \text{[var]} \]
\[ E \vdash 5 \Downarrow 5 \quad \text{[value]} \]

**[mult]**

\[ E \vdash (\ast \ x \ 5) \Downarrow 15 \]
\[ E \vdash \text{diff} \Downarrow 9 \quad \text{[var]} \]

**[add]**

\[ E \vdash (\ast \ x \ 5 \ \text{diff}) \Downarrow 24 \]

**[if nonfalse]**

\[ E \vdash (\text{if } \text{test} \ (\ast \ x \ 5 \ \text{diff}) \ 17) \Downarrow 24 \]
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 \( x \mapsto v \).

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

Environment example

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

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<c*d-e?
- 64bits

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.