CS 251 Part 1:
How to Program
Defining Racket: Expressions and Bindings

via the meta-language of PL definitions
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.
PL design/implementation: layers

- kernel
- primitive values, data types
- syntactic sugar
- standard libraries
- user libraries
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: \( (+ \ e1 \ e2) \)

Evaluation:
1. Evaluate \( e1 \) to a value \( v1 \).
2. Evaluate \( e2 \) to a value \( v2 \).
3. Return the **arithmetic sum** of \( v1 + v2 \).
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.

Non-terminal symbol $e$ 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
\[ e ::= v \]
\[ \quad | ( + e e ) \]

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

Number values
\[ n ::= 0 \quad | \quad 1 \quad | \quad 2 \quad | \quad \ldots \]
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., e₁, e₂, ..., eₙ or e₁, e₂, ..., eₙ.
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 $\text{e } \downarrow \text{v}$ means "expression e evaluates to value v."

It is implemented by inference rules for different cases:

**value rule:**

<table>
<thead>
<tr>
<th>$\text{v } \downarrow \text{v}$</th>
</tr>
</thead>
</table>

**addition rule:**

if $\text{e1 } \downarrow \text{n1}$
and $\text{e2 } \downarrow \text{n2}$
and n is the arithmetic sum of n1 and n2
then $(+ \text{e1 e2}) \downarrow \text{n}$

| $\text{e1 } \downarrow \text{n1}$ |
| $\text{e2 } \downarrow \text{n2}$ |
| $\text{n } = \text{n1 } + \text{n2}$ |
| $(+ \text{e1 e2}) \downarrow \text{n}$ |
Inference rules

Axiom (no premises)
Bar is optional for axioms.

Premises

Conclusion

Rule name

Inference rule notation and meaning

If all premises hold then the conclusion holds.

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

"v is the arithmetic sum of the numbers n1 and n2." (not Racket syntax)

Expressions, Bindings, Meta-language
Evaluation derivations

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

\[
(+ 3 (+ 5 4)) \downarrow 12 \text{ by the addition rule because:}
\]

\begin{itemize}
  \item 3 \downarrow 3 \text{ by the value rule, where 3 is a number}
  \item and (+ 5 4) \downarrow 9 \text{ by the addition rule, where 9 is a number, because:}
    \begin{itemize}
      \item 5 \downarrow 5 \text{ by the value rule, where 5 is a number}
      \item and 4 \downarrow 4 \text{ by the value rule, where 4 is a number}
      \item and 9 is the sum of 5 and 4
    \end{itemize}
  \item and 12 is the sum of 3 and 9.
\end{itemize}
Evaluation derivations

Adding vertical bars helps clarify nesting.

3 \downarrow 3 \quad \text{[value]}
5 \downarrow 5 \quad \text{[value]}
4 \downarrow 4 \quad \text{[value]}
9 = 5 + 4
( + 5 4 ) \downarrow 9
12 = 3 + 9
( + 3 ( + 5 4 ) ) \downarrow 12

Rules defining the evaluation judgment

\[
\begin{align*}
e & \downarrow v \\
v & \downarrow v \quad \text{[value]}
\end{align*}
\]

\[
\begin{align*}
e1 & \downarrow n1 \\
e2 & \downarrow n2 \\
n & = n1 + n2 \\
( + e1 e2 ) & \downarrow n \quad \text{[add]}
\end{align*}
\]
Errors are modeled by “stuck” derivations.

How to evaluate

(+ #t (+ 5 4))?

#t ↓ n X
5 ↓ 5 [value]
4 ↓ 4 [value]
9 = 5 + 4 [add]
(+ 5 4) ↓ 9 [add]

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))?

1 ↓ 1 [value]
2 ↓ 2 [value]
3 = 1 + 2
(+ 1 2) ↓ 3 [add]
5 ↓ 5 [value]
#f ↓ n X [add]

Stuck. Can’t apply the [add] rule because there is no rule that allows #t 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 \( i f \) expressions.

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

\[\text{[if nonfalse]}\]

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

\[\text{[if false]}\]

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

**if expressions are expressions.**
Racket has no "statements!"

```
(if (< 9 (- 251 240))
  (+ 4 (* 3 2))
  (+ 4 (* 3 3)))

(+ 4 (* 3 (if (< 9 (- 251 240)) 2 3))))

(if (if (< 1 2) (> 4 3) (> 5 6))
  (+ 7 8)
  (* 9 10))
```
**if** expression evaluation

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

```scheme
(if (> 251 240) 251 (/ 251 0))
```

```scheme
(if #f (+ #t 251) 251)
```
Language design choice: *if* semantics

\[
e_1 \downarrow v_1 \\
_2 \downarrow v_2 \\
_1 \text{ is not } \#f \\
\frac{\text{(if } e_1 \ e_2 \ e_3) \downarrow v_2}{[\text{if nonfalse}]}
\]

\[
e_1 \downarrow \#t \\
_2 \downarrow v_2 \\
\frac{\text{(if } e_1 \ e_2 \ e_3) \downarrow v_2}{[\text{if true}]}
\]

*v* not required to be a Boolean value

Alternative design
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.
More Racket syntax

**Bindings**

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

**Expressions**

\[
e ::= v \mid x \mid ( + e e e ) \mid \ldots \mid (\texttt{if } e e e e)\]

**Literal Values (booleans, numbers)**

\[
v ::= \texttt{#f} \mid \texttt{#t} \mid n\]

**Identifiers (variable names)**

\[
x \text{ (see valid identifier explanation)}\]
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: \[ \mathbf{x} \]

\( \mathbf{x} \) is any identifier

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

\[
\mathcal{E} \vdash e \downarrow \mathbf{v}
\]

Revised expression evaluation judgment uses environment.

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

\[
\mathcal{E}(\mathbf{x}) = \mathbf{v}
\]

[\text{[var]}]
Expression evaluation rules must pass the environment.

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

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

\[
\begin{align*}
E &\vdash test \downarrow \#t \quad [\text{var}] \\
E &\vdash x \downarrow 3 \quad [\text{var}] \\
E &\vdash 5 \downarrow 5 \quad [\text{value}] \\
E &\vdash (* \ x \ 5) \downarrow 15 \quad [\text{mult}] \\
E &\vdash diff \downarrow 9 \quad [\text{var}] \\
E &\vdash (+ \ (* \ x \ 5) \ diff) \downarrow 24 \quad [\text{add}] \\
E &\vdash (if \ test \ (+ \ (* \ x \ 5) \ diff) \ 17) \downarrow 24 \quad [\text{if nonfalse}] \\
\end{align*}
\]
**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 \(x \mapsto v\).

\[
E \vdash b \Downarrow E' \quad \text{[define]}
\]

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

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

Most character sequences are allowed as identifiers, except:

- those containing
  - whitespace
  - special characters ( ) [ ] { } ” ’ ` ; # | \n  - 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.