CS 251 Part 1:
How to Program

WELLESLEY

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

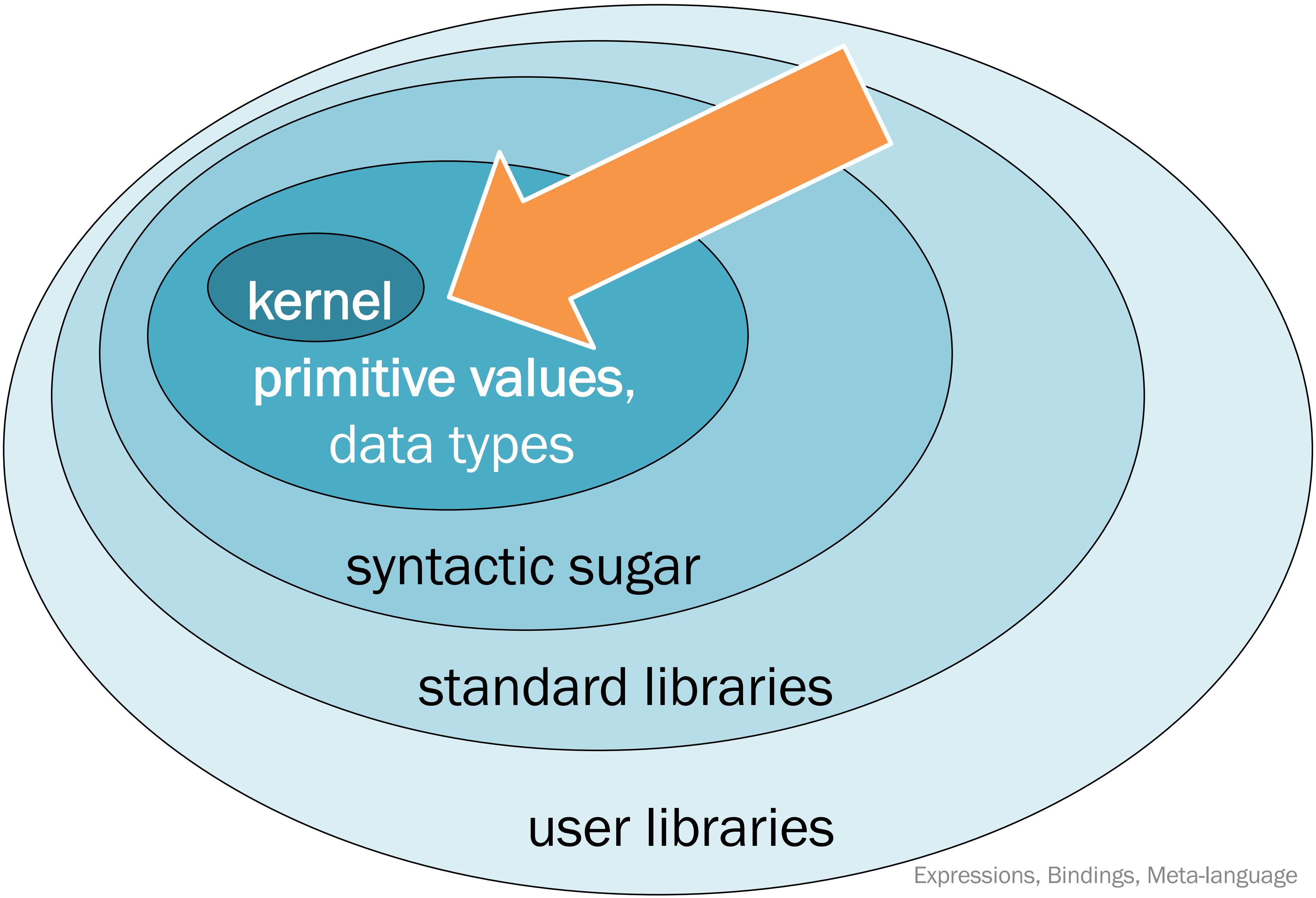
To define each new language feature:

- 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.

PL design/implementation: layers



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.

Note recursive structure!

Examples:

`(+ 251 240)` `(+ (+ 251 240) 301)`

`(+ #t 251)`

Addition expression

Not quite!

Syntax: `(+ e1 e2)`

Note recursive structure!

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: `(+ e1 e2)`

Evaluation:

Dynamic type checking

1. Evaluate ***e1*** to a value `v1`.
2. Evaluate ***e2*** to a value `v2`.
3. If **`v1`** and **`v2`** are numbers then
   return the arithmetic sum of `v1 + v2`.
   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 symbols

```
e ::= v
    | (+ e e)
```

Non-terminal e has 2 ***productions***, separated by "|".

Grammar meta-syntax.

terminal symbols

"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

  | (+ e e)

Literal Values

v ::= #f | #t | n

Number values

n ::= 0 | 1 | 2 | …

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 `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 $\boxed{\texttt{e} \downarrow \texttt{v}}$
means "expression e evaluates to value v."

It is implemented by **inference rules** for different cases:

value rule:

$$\frac{}{\texttt{v} \downarrow \texttt{v}}\ [\text{value}]$$

addition rule:

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

$$\frac{\texttt{e1} \downarrow \texttt{n1} \quad \texttt{e2} \downarrow \texttt{n2} \quad \texttt{n} = \texttt{n1} + \texttt{n2}}{\texttt{(+ e1 e2)} \downarrow \texttt{n}}\ [\text{add}]$$

...

Inference rules

If $\dfrac{\text{all premises}}{\text{the conclusion}}$ hold
then ... holds.

Inference rule notation and meaning

Axiom (no premises)
Bar is optional for axioms.

Conclusion

$$\frac{}{\texttt{v} \downarrow \texttt{v}} \; \text{[value]}$$

Rule name

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

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

Premises

Conclusion

$$\frac{\begin{array}{l}\texttt{e1} \downarrow \texttt{n1} \\ \texttt{e2} \downarrow \texttt{n2} \\ \texttt{n} = \texttt{n1} + \texttt{n2}\end{array}}{\texttt{(+ e1 e2)} \downarrow \texttt{n}} \; \text{[add]}$$

Rule name

Evaluation derivations

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

`(+ 3 (+ 5 4))` ↓ `12` by the addition rule because:

- `3` ↓ `3` by the value rule, where `3` is a number
- and `(+ 5 4)` ↓ `9` by the addition rule , where `9` is a number, because:
  - `5` ↓ `5` by the value rule, where `5` is a number
  - and `4` ↓ `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

Adding vertical bars helps clarify nesting.

$$\dfrac{3 \downarrow 3 \ \text{[value]} \qquad \dfrac{5 \downarrow 5 \ \text{[value]} \qquad 4 \downarrow 4 \ \text{[value]} \qquad 9 = 5 + 4}{(+\ 5\ 4) \downarrow 9}\ \text{[add]} \qquad 12 = 3 + 9}{(+\ 3\ (+\ 5\ 4)) \downarrow 12}\ \text{[add]}$$

Rules defining the evaluation judgment

$\boxed{e \downarrow v}$

$$\dfrac{}{v \downarrow v}\ \text{[value]}$$

$$\dfrac{e1 \downarrow n1 \qquad e2 \downarrow n2 \qquad n = n1 + n2}{(+\ e1\ e2) \downarrow n}\ \text{[add]}$$

Errors are modeled by "stuck" derivations.

How to evaluate
`(+ #t (+ 5 4))?`

```
#t ↓ n
    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
---------------- [add]
(+ 1 2) ↓ 3

    5 ↓ 5 [value]
    #f ↓ n
    ------------- [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 `if` expressions

Syntax: `(if e1 e2 e3)`

Evaluation:

1. Evaluate `e1` to a value `v1`.
2. If `v1` is not the value `#f` then

   evaluate `e2` and return the result

   otherwise

   evaluate `e3` and return the result

Evaluation rules for *if* expressions.

$$\frac{\texttt{e1} \downarrow \texttt{v1} \quad \texttt{e2} \downarrow \texttt{v2} \quad \texttt{v1 is not \#f}}{\texttt{(if e1 e2 e3)} \downarrow \texttt{v2}} \text{ [if nonfalse]}$$

e3 is not evaluated!

$$\frac{\texttt{e1} \downarrow \texttt{\#f} \quad \texttt{e3} \downarrow \texttt{v3}}{\texttt{(if e1 e2 e3)} \downarrow \texttt{v3}} \text{ [if false]}$$

e2 is not evaluated!

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?

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

```
(if #f (+ #t 251) 251)
```

Language design choice: *if* semantics

```
e1 ↓ v1
e2 ↓ v2
v1 is not #f
────────────────────────
(if e1  e2  e3) ↓ v2
```

[if nonfalse]

v1 not required to be a Boolean value

```
e1 ↓ #t
e2 ↓ v2
────────────────────────
(if e1  e2  e3) ↓ v2
```

[if true]

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 ::= (define x e)`

Expressions

`e ::= v | x | ( + e e ) | … | (if e e e)`

Literal Values (booleans, numbers)

`v ::= #f | #t | n`

Identifiers (variable names)

`x` (see valid identifier explanation)

Dynamic environments

Grammar for environment notation:

$$
\begin{array}{lll}
E ::= & . & \text{(empty environment)} \\
\quad\mid & x \mapsto v,\ E & \text{(one binding, rest of environment)}
\end{array}
$$

where:

- x is any legal variable identifier
- v is any value

Concrete example:

```
num ↦ 17, absZero ↦ -273, true ↦ #t, .
```

Abstract example:

```
x1 ↦ v1, x2 ↦ v2, …, xn ↦ vn, .
```

Variable reference expressions

Syntax: $\boldsymbol{x}$

$\boldsymbol{x}$ 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.

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

$$\boxed{\mathbf{E} \vdash e \downarrow v}$$

Revised expression evaluation judgment uses environment.

$$\frac{E(x) = v}{E \vdash x \downarrow v}\ \text{[var]}$$

Expression evaluation rules must pass the environment.

$$\texttt{E} \vdash \texttt{x} \downarrow \texttt{v}$$

$$\texttt{E} \vdash \texttt{v} \downarrow \texttt{v} \quad \text{[value]}$$

$$\frac{\texttt{E(x) = v}}{\texttt{E} \vdash \texttt{x} \downarrow \texttt{v}} \quad \text{[var]}$$

$$\frac{\texttt{E} \vdash \texttt{e1} \downarrow \texttt{n1} \quad \texttt{E} \vdash \texttt{e2} \downarrow \texttt{n2} \quad \texttt{n = n1 + n2}}{\texttt{E} \vdash \texttt{(+ e1 e2)} \downarrow \texttt{n}} \quad \text{[add]}$$

$$\frac{\texttt{E} \vdash \texttt{e1} \downarrow \texttt{v1} \quad \texttt{E} \vdash \texttt{e2} \downarrow \texttt{v2} \quad \texttt{v1 is not \#f}}{\texttt{E} \vdash \texttt{(if e1 e2 e3)} \downarrow \texttt{v2}} \quad \text{[if nonfalse]}$$

$$\frac{\texttt{E} \vdash \texttt{e1} \downarrow \texttt{\#f} \quad \texttt{E} \vdash \texttt{e3} \downarrow \texttt{v3}}{\texttt{E} \vdash \texttt{(if e1 e2 e3)} \downarrow \texttt{v3}} \quad \text{[if false]}$$

Derivation with environments

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

$$\dfrac{E \vdash \texttt{test} \downarrow \texttt{\#t} \;\; \text{[var]} \qquad \dfrac{\dfrac{E \vdash \texttt{x} \downarrow 3 \;\; \text{[var]} \qquad E \vdash \texttt{5} \downarrow 5 \;\; \text{[value]} \qquad 15 = 3 * 5}{E \vdash \texttt{(* x 5)} \downarrow 15}\,\text{[mult]} \qquad E \vdash \texttt{diff} \downarrow 9 \;\; \text{[var]} \qquad 24 = 15 + 9}{E \vdash \texttt{(+ (* x 5) diff)} \downarrow 24}\,\text{[add]}}{E \vdash \texttt{(if test (+ (* x 5) diff) 17)} \downarrow 24}\,\text{[if nonfalse]}$$

`define` bindings

Syntax: `(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 $x \longmapsto v$.

$$\boxed{E \vdash b \Downarrow E'}$$

$$\frac{E \vdash e \downarrow v \qquad E' = x \longmapsto v,\ E}{E \vdash (\texttt{define}\ x\ e) \Downarrow E'}\ \text{[define]}$$

Environment example (`define.rkt`)

```
    ; E0 = .
(define x (+ 1 2))
    ; E1 = x ↦ 3, .
(define y (* 4 x))
    ; E2 = y ↦ 12, x ↦ 3
(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)
    ;
```

(abbreviated `x ↦ 3`; write `x --> 3` in text)

(most recent binding first)

(environment here is still E4)

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:

```
(- (* (+ 2 3) 9) (/ 18 6))
→ (- (* 5 9) (/ 18 6))
→ (- 45 (/ 18 6))
→ (- 45 3)
→ 42
```

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