CS 251 Spring 2020
Principles of Programming Languages
Ben Wood

# CS 251 Part 1: How to Program



CS 251 Spring 2020
Principles of Programming Languages
Ben Wood

# 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:

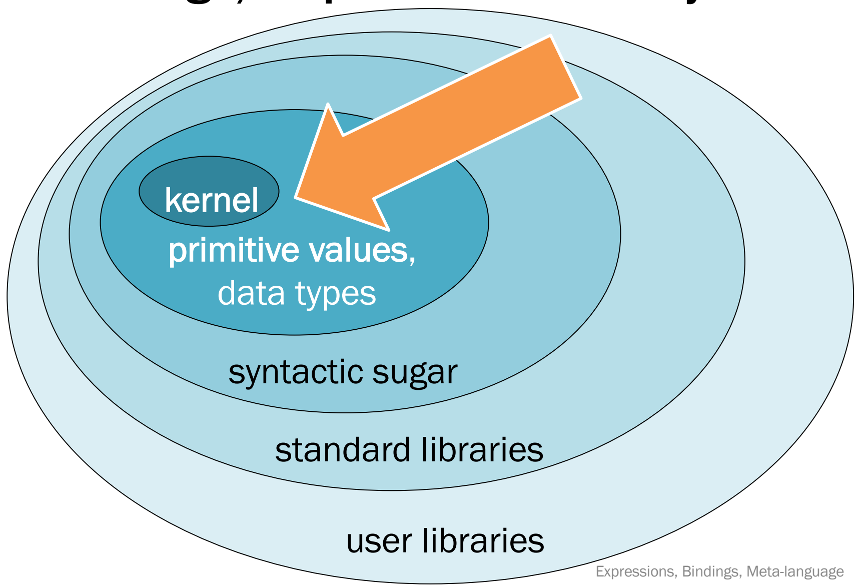
– Define its **syntax.**
  How is it written?
– Define its **dynamic semantics** as **evaluation rules.**
  How is it evaluated?

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.

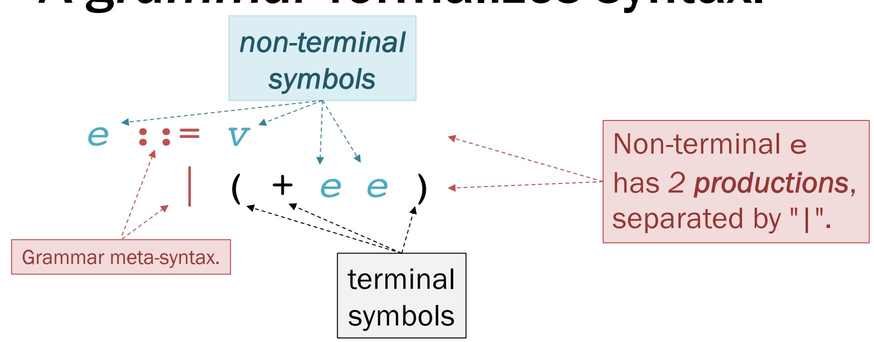
**Syntax:**

– Formal grammar notation

– Conventions for writing syntax patterns



## A *grammar* formalizes syntax.




"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** $e \downarrow v$
means "expression e evaluates to value v."

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

*value rule*:

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

*addition rule*:

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

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

...

# Inference rules

| If | all premises | hold |
|---|---|---|
| then | the conclusion | holds. |

Inference rule notation and meaning

Axiom (no premises)
Bar is optional for axioms.

Conclusion

```
______ [value]
v ↓ v
```

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

```
e1 ↓ n1
e2 ↓ n2
n = n1 + n2
________________ [add]
(+ e1 e2) ↓ n
```

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.*

```
| 3 ↓ 3 [value]
|  | 5 ↓ 5 [value]
|  | 4 ↓ 4 [value]
|  | 9 = 5 + 4
|  |______________ [add]
| (+ 5 4) ↓ 9
| 12 = 3 + 9
|___________________ [add]
(+ 3 (+ 5 4)) ↓ 12
```

Rules defining the evaluation judgment

```
e ↓ v

______ [value]
v ↓ v

e1 ↓ n1
e2 ↓ n2
n = n1 + n2
______________ [add]
(+ e1 e2) ↓ n
```



# 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.

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

e3 is not evaluated!

[if nonfalse]

```
  e1 ↓ #f
  e3 ↓ v3
---------------------
(if e1 e2 e3) ↓ v3
```

e2 is not evaluated!

[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?

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

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



## Language design choice: *if* semantics

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

```
e1 ↓ v1
e2 ↓ v2
v1 is not #f
-----------------------  [if nonfalse]
(if e1  e2  e3) ↓ v2
```

Alternative design

```
e1 ↓ #t
e2 ↓ v2
-----------------------  [if true]
(if e1  e2  e3) ↓ v2
```



## 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:

```
E ::= .                  (empty environment)
    | x ↦ v, E           (one binding, rest of environment)
```

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:** ***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 ⊢ e ↓ v
```

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

*Revised expression evaluation judgment uses environment.*

```
E(x) = v
───────────  [var]
E ⊢ x ↓ v
```



# Expression evaluation rules must pass the environment.

```
E ⊢ x ↓ v
```

```
E ⊢ v ↓ v  [value]
```

```
E(x) = v
───────────  [var]
E ⊢ x ↓ v
```

```
E ⊢ e1 ↓ n1
E ⊢ e2 ↓ n2
n = n1 + n2
─────────────────────  [add]
E ⊢(+ e1  e2) ↓ n
```

```
E ⊢ e1 ↓ v1
E ⊢ e2 ↓ v2
v1 is not #f
──────────────────────────  [if nonfalse]
E ⊢ (if e1  e2  e3) ↓ v2
```

```
E ⊢ e1 ↓ #f
E ⊢ e3 ↓ v3
──────────────────────────  [if false]
E ⊢ (if e1  e2  e3) ↓ v3
```



# Derivation with environments

Let `E = test ↦ #t, diff ↦ 9, y ↦ 12, x ↦ 3`

```
E ⊢ test ↓ #t                    [var]
    E ⊢ x ↓ 3                    [var]
    E ⊢ 5 ↓ 5                    [value]
    15 = 3 * 5
    ─────────────────────        [mult]
  E ⊢ (* x 5) ↓ 15
  E ⊢ diff ↓ 9                   [var]
  24 = 15 + 9
  ──────────────────────────────  [add]
E ⊢ (+ (* x 5) diff) ↓ 24
──────────────────────────────────────  [if nonfalse]
E ⊢ (if test (+ (* x 5) diff) 17) ↓ 24
```

# `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` ↦ `v`.

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

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



# 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.