

# Syntax & semantics of Beginning Student

**Readings:** HtDP, Intermezzo 1 (Section 8).

We are covering the ideas of section 8, but not the parts of it dealing with section 6/7 material (which will come later), and in a somewhat different fashion.

## Topics:

- Modelling programming languages
- Racket's semantic model
- Substitution rules (so far)

# Modelling programming languages

- A program has a precise meaning and effect.
- A model of a programming language provides a way of describing the meaning of a program.
- Typically this is done informally, by examples.
- With Racket, we can do better.

## > Advantages in modelling Racket

- Few language constructs, so model description is short
- We don't need anything more than the language itself!
  - No diagrams
  - No vague descriptions of the underlying machine

## > Spelling rules for Beginning Student

Identifiers are the names of constants, parameters, and user-defined functions.

They are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks. They must contain at least one non-number. They can't contain spaces or any of these:

( ) , ; { } [ ] ' ' " " .

Symbols start with a single quote ' followed by something obeying the rules for identifiers.

There are rules for numbers (integers, rationals, decimals) which are fairly intuitive.

There are some built-in constants, like `true` and `false`.

Of more interest to us are the rules describing program structure.

For example: a program is a sequence of definitions and expressions.

# > Syntax, semantics, and ambiguity

There are three problems we need to address:

- 1 **Syntax**: The way we're allowed to say things.  
*'?is This Sentence Syntactically Correct'*
- 2 **Semantics**: the meaning of what we say.  
*'Trombones fly hungrily.'*
- 3 **Ambiguity**: valid sentences have exactly one meaning.  
*'Sally was given a book by Joyce.'*

English rules on these issues are pretty lax. For Racket, we need rules that *always* avoid these problems.

## » Grammars

To enforce syntax and avoid ambiguity, we can use **grammars**.

For example, an English sentence can be made up of a subject, verb, and object, in that order.

We might express this as follows:

$\langle \text{sentence} \rangle = \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle$

The linguist Noam Chomsky formalized grammars in this fashion in the 1950's. The idea proved useful for programming languages.

The textbook describes function definitions like this:

$\langle \text{def} \rangle = (\text{define } ( \langle \text{var} \rangle \langle \text{var} \rangle \dots \langle \text{var} \rangle ) \langle \text{exp} \rangle )$

There is a similar rule for defining constants. Additional rules define **cond** expressions, etc.

The Help Desk presents the same idea as

`definition = (define (id id id ...) expr)`

In CS 135, we will use informal descriptions instead.

CS 241, CS 230, CS 360, and CS 444 discuss the mathematical formalization of grammars and their role in the interpretation of computer programs and other structured texts.



# Racket's semantic model

The second of our three problems (syntax, semantics, ambiguity) we will solve rigorously with a **semantic model**. A semantic model of a programming language provides a method of predicting the result of running any program.

Our model will repeatedly simplify the program via **substitution**.

A **substitution step** finds the leftmost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.

**Every substitution step yields a valid program** (in full Racket), until all that remains is a sequence of definitions and values.

## > Application of built-in functions

We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions like  $(+ 3 5)$  and  $(\text{expt } 2 10)$ .

$(+ 3 5) \Rightarrow 8$

$(\text{expt } 2 10) \Rightarrow 1024$

Formally, the substitution rule is:

$(f v_1 \dots v_n) \Rightarrow v$  where  $f$  is a built-in function and  $v$  is the value of  $f(v_1, \dots, v_n)$ .

Note the two uses of an **ellipsis**  $(. . .)$ . What does it mean?

## > Ellipses

For built-in functions  $f$  with one parameter, the rule is:

$(f\ v_1) \Rightarrow v$  where  $v$  is the value of  $f(v_1)$

For built-in functions  $f$  with two parameters, the rule is:

$(f\ v_1\ v_2) \Rightarrow v$  where  $v$  is the value of  $f(v_1, v_2)$

For built-in functions  $f$  with three parameters, the rule is:

$(f\ v_1\ v_2\ v_3) \Rightarrow v$  where  $v$  is the value of  $f(v_1, v_2, v_3)$

We can't just keep writing down rules forever, so we use ellipses to show a *pattern*:

$(f\ v_1\ \dots\ v_n) \Rightarrow v$  where  $v$  is the value of  $f(v_1, \dots, v_n)$ .

## > Application of user-defined functions

Consider (**define** (term x y) (\* x (sqr y))).

The function application (term 2 3) can be evaluated by taking the body of the function definition and replacing x by 2 and y by 3.

The result is (\* 2 (sqr 3)).

The rule does not apply if an argument is not a value, as in the case of the second argument in (term 2 (+ 1 2)).

Any argument which is not a value must first be simplified to a value using the rules for expressions.

The general substitution rule is:

$(f\ v_1\ \dots\ v_n) \Rightarrow \text{exp}'$

where (**define**  $(f\ x_1\ \dots\ x_n)\ \text{exp}$ ) occurs to the left, and  $\text{exp}'$  is obtained by substituting into the expression  $\text{exp}$ , with all occurrences of the formal parameter  $x_i$  replaced by the value  $v_i$  (for  $i$  from 1 to  $n$ ).

Note we are using a pattern ellipsis in the rules for both built-in and user-defined functions to indicate several arguments.

## » Example:

```
(define (term x y) (* x (sqr y)))
```

```
(term (- 3 1) (+ 1 2))
```

```
⇒ (term 2 (+ 1 2))
```

```
⇒ (term 2 3)
```

```
⇒ (* 2 (sqr 3))
```

```
⇒ (* 2 9)
```

```
⇒ 18
```

A constant definition binds a name (the constant) to a value (the value of the expression).

We add the substitution rule:

`id ⇒ val`

where (**define** `id val`) occurs to the left.

## » Example:

```
(define x 3)
(define y (+ x 1))
y ⇒
(define x 3)
(define y (+ 3 1))
y ⇒
(define x 3)
(define y 4)
y ⇒
(define x 3)
(define y 4)
4
```

To avoid a lot of repetition, we adopt the convention that we stop repeating a definition once its expression has been reduced to a value (since it cannot change after that).

```
(define x 3)
(define y (+ x 1))
y ⇒
(define y (+ 3 1))
y ⇒
(define y 4)
y ⇒
4
```



## > Substitution in cond expressions

There are three rules: when the first expression is false, when it is true, and when it is **else**.

```
(cond [false exp] ...) ⇒ (cond ...)
```

```
(cond [true exp] ...) ⇒ exp
```

```
(cond [else exp]) ⇒ exp
```

These suffice to simplify any **cond** expression.

Here the ellipses are serving a different role. They are not showing a pattern, but showing an **omission**. The rule just says “whatever else appeared after the [false exp], you just copy it over.”

## » Example:

```
(define n 5) (define x 6) (define y 7)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ 7
```

What happens if *y* is not defined?

## » Example:

```
(define n 5) (define x 6)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ y: this variable is not defined
```

DrRacket's rules differ. It scans the whole **cond** expression before it starts, notes that **y** is not defined, and shows an error. That's hard to explain with substitution rules!

# Exercise 1

Given the definition:

```
(define (foo x)
  (cond [(odd? x) "odd"]
        [(= 2 (remainder x 10)) "strange"]
        [(> x 100) "big"]
        [(even? x) "even"])))
```

First evaluate the expression by hand:

```
(foo 102)
```

Then run the code to check your answer.

## Exercise 2

Figure out what you think the following program will display.

Then run it in Racket to check your understanding.

```
(define (waldo x)
  (cond
    [(even? x) "even"]
    [true "neither even nor odd"]
    [(odd? x) "odd"]
  ))
```

```
(waldo 4)
```

```
(waldo 3)
```

## > Errors

A **syntax error** occurs when a sentence cannot be interpreted using the grammar.

Example: `(10 + 1)`

A **run-time error** occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules.

Example:

```
(cond [(> 3 4) x])  
⇒ (cond [false x])  
⇒ (cond )  
⇒ cond: all question results were false
```

## > Substitution rules for **and** and **or**

The simplification rules we use for Boolean expressions involving **and** and **or** are different from the ones the Stepper in DrRacket uses.

The end result is the same, but the intermediate steps are different.

The implementers of the Stepper made choices which result in more complicated rules, but whose intermediate steps appear to help students in lab situations.

## > Substitution rules for **and** and **or**

**(and false ...)**  $\Rightarrow$  false

**(and true ...)**  $\Rightarrow$  **(and ...)**

**(and)**  $\Rightarrow$  true

**(or true ...)**  $\Rightarrow$  true

**(or false ...)**  $\Rightarrow$  **(or ...)**

**(or)**  $\Rightarrow$  false

As in the rewriting rules for **cond**, we are using an omission ellipsis.



## Exercise 3

Perform a trace of

```
(and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))
```

## Exercise 4

Perform a trace of

```
(or (< 7 4) (= 3 3) (> 7 4) (> 0 (/ 3 0)))
```

# Substitution rules (so far)

- 1 Apply functions only when all arguments are values.
- 2 When given a choice, evaluate the leftmost expression first.
- 3  $(f\ v_1 \dots v_n) \Rightarrow v$  when  $f$  is built-in...
- 4  $(f\ v_1 \dots v_n) \Rightarrow \text{exp}'$  when  $(\mathbf{define}\ (f\ x_1 \dots x_n)\ \text{exp})$  occurs to the left...
- 5  $\text{id} \Rightarrow \text{val}$  when  $(\mathbf{define}\ \text{id}\ \text{val})$  occurs to the left.

6 (**cond** [false exp] ...)  $\Rightarrow$  (**cond** ...)

7 (**cond** [true exp] ...)  $\Rightarrow$  exp

8 (**cond** [else exp])  $\Rightarrow$  exp

9 (**and** false ...)  $\Rightarrow$  false

10 (**and** true ...)  $\Rightarrow$  (**and** ...)

11 (**and**)  $\Rightarrow$  true

12 (**or** true ...)  $\Rightarrow$  true

13 (**or** false ...)  $\Rightarrow$  (**or** ...)

14 (**or**)  $\Rightarrow$  false

# Importance of the model

We will add to the semantic model when we introduce a new feature of Racket.

Understanding the semantic model is very important in understanding the meaning of a Racket program.

Doing a step-by-step reduction according to these rules is called **tracing** a program.

It is an important skill in any programming language or computational system.

We will test this skill on assignments and exams.

# Goals of this module

- You should understand the substitution-based semantic model of Racket, and be prepared for future extensions.
- You should be able to trace the series of simplifying transformations of a Racket program.