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 symantic 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: ( ) , ; { } [ ] ‘ ’ “ ”

**Symbols** start with a single quote ’ followed by something obeying the rules for identifiers.
Spelling rules for Beginning Student

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

There are some **built-in constants**, such as **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 *form or structure* we use to say things.
   
   Not: ‘*is This Sentence Syntactically Correct*’

2. **Semantics**: the *meaning* of what we say.
   
   Not: ‘*Trombones fly hungrily.*’

3. **Ambiguity**: valid sentences have exactly *one meaning*.
   
   Not: ‘*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, use grammars, i.e. rules used to construct valid programs (in Racket) or sentences (in English).

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:

\[
\text{\langle def\rangle} = (\text{define (\langle var\rangle \langle var\rangle \ldots \langle var\rangle) \langle exp\rangle})
\]

There are also rules for defining constants, \text{cond} expressions, etc.

The Help Desk presents the same idea as

\[
\text{definition} = (\text{define (id id id \ldots) expr})
\]

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

A substitution step finds the leftmost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.
Application of built-in functions

Use the rules for the arithmetic expressions to substitute the appropriate value for expressions like $\left(\mathbf{+} \ 3 \ 5\right)$ and $\left(\mathrm{expt} \ 2 \ 10\right)$.

$\left(\mathbf{+} \ 3 \ 5\right) \Rightarrow 8$

$\left(\mathrm{expt} \ 2 \ 10\right) \Rightarrow 1024$

Formally, the substitution rule for a built-in function $f$ is:

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

Note: Rules 1 and 2 were covered in Module 02 slide 13.

Note the two uses of an ellipsis ($\ldots$). 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:

$$(f \ v_1 \ldots \ v_n) \Rightarrow v$$

where $v$ is the value of $f(v_1, \ldots, v_n)$.
Application of user-defined functions

As an example, consider \( \text{define (term x y) (\ast x (sqr y))} \).

The *function application* \( \text{(term 2 3)} \) can be evaluated by taking the body of the function definition and *replacing the parameters* \( x \) and \( y \) *with the arguments* \( 2 \) and \( 3 \).

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

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

Any *argument* which is not a value *must first be simplified to a value* using the rules for expressions.
Application of user-defined functions

The *general substitution* rule is:

4. \((f \, v_1 \ldots \, v_n) \Rightarrow \text{exp}'\) where \((\text{define} \, (f \, x_1 \ldots \, x_n) \, \text{exp})\) 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:

\[
\begin{align*}
\text{(define (term x y) (\times x (sqr y)))} \\
\text{(term (\neg 3 1) (\plus 1 2))} & ; \text{apply built-in function (to 1st arg)} \\
\Rightarrow (\text{term 2 (\plus 1 2)}) & ; \text{apply built-in function (to 2nd arg)} \\
\Rightarrow (\text{term 2 3}) & ; \text{replace parameters with arguments} \\
\Rightarrow (\times 2 (\text{sqr 3})) & ; \text{apply built-in function (to 2nd arg)} \\
\Rightarrow (\times 2 9) & ; \text{apply built-in function} \\
\Rightarrow 18
\end{align*}
\]

Note: The comments were included here to clarify each step. On a midterm or final, do not put the comments.
Application of user-defined constants

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

We add the substitution rule:

5. \( id \mapsto val \), where \( (\text{define } id \text{ val}) \) occurs to the left.
Example:

\[(\text{define } x 3) \ (\text{define } y \ (+ \ x \ 1)) \ y\]

⇒ ; substitute 3 for x

\[(\text{define } x 3) \ (\text{define } y \ (+ \ 3 \ 1)) \ y\]

⇒ ; apply built-in function

\[(\text{define } x 3) \ (\text{define } y 4) \ y\]

⇒ ; substitute 4 for y

\[(\text{define } x 3) \ (\text{define } y 4) \ 4\]

Note: In order to fit this onto one slide, we have put the program onto one line and added comments, but students should avoid both of these on a midterm or final (if possible).
Substitution in \texttt{cond} expressions

There are three rules: when the first expression is (1) \texttt{false}, when it is (2) \texttt{true}, and when it is (3) \texttt{else}.

6. \texttt{(cond [false exp] \ldots \Rightarrow (cond \ldots).}

7. \texttt{(cond [true exp] \ldots \Rightarrow exp.}

8. \texttt{(cond [else exp]) \Rightarrow exp.}

These rules suffice to simplify any \texttt{cond} expression.

Here the \textit{ellipses} are serving a different role. They are not showing a pattern, but \textit{showing an omission}. The rule just says “whatever else appeared after the \texttt{[false exp]}, you just copy it over.”
The Simplification Rules for cond

- \((\text{cond } [\text{false exp}] \ldots) \Rightarrow (\text{cond } \ldots)\) means the substitution for \((\text{cond } [\text{false exp}] \ldots)\) (i.e. when the first condition is false) would be \((\text{cond } \ldots)\) (i.e. the rest of the conditions).

- \((\text{cond } [\text{true exp}] \ldots) \Rightarrow \text{exp}\) means the substitution for \((\text{cond } [\text{true exp}] \ldots)\) (i.e. when the first condition is true) would be \(\text{exp}\) (i.e. the expression for that condition).

- \((\text{cond } [\text{else exp}] \ldots) \Rightarrow \text{exp}\) means the substitution for \((\text{cond } [\text{else exp}] \ldots)\) (i.e. the else condition) would be \(\text{exp}\) (i.e. the expression for the else).
Simplification Examples for cond

- Using substitutions \((\text{cond } [\text{false } \text{exp}] \ldots ) \Rightarrow (\text{cond } \ldots )\) and \((\text{cond } [\text{true } \text{exp}] \ldots ) \Rightarrow \text{exp.}\)

\[
(\text{cond } [(= 1 2) (+ 0 1)] [ (< 2 3) (+ 0 2)]) \Rightarrow
\]

\[
(\text{cond } [\text{false } (+ 0 1)] [ (< 2 3) (+ 0 2)]) \Rightarrow
\]

\[
(\text{cond } [(< 2 3) (+ 0 2)]) \Rightarrow \quad \text{; by rule for false}
\]

\[
(\text{cond } [\text{true } (+ 0 2)]) \Rightarrow
\]

\[
(+ 0 2) \Rightarrow \quad \text{; by rule for true}
\]

2
Simplification Examples for cond

- Using substitutions: 
  \[(\text{cond} \ [\text{false exp}] \ldots) \Rightarrow (\text{cond} \ldots)\]
  and 
  \[(\text{cond} \ [\text{else exp}] \ldots) \Rightarrow \text{exp}\]

\[(\text{cond} \ [(=\ 1\ 2)\ (+\ 0\ 1)]\ [\text{else}\ (+\ 0\ 2)]) \Rightarrow\]

\[(\text{cond} \ [\text{false}\ (+\ 0\ 1)]\ [\text{else}\ (+\ 0\ 2)] \Rightarrow\]

\[(\text{cond} \ [\text{else}\ (+\ 0\ 2)]) \Rightarrow\] ; by rule for false

\[(+\ 0\ 2) \Rightarrow\] ; by rule for else

2
Example:

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

(cond [(even? n) x][(odd? n) y]) ; substitute 5 for n
⇒ (cond [(even? 5) x] [(odd? n) y]) ; apply built-in function
⇒ (cond [false x][(odd? n) y]) ; apply rule false cond
⇒ (cond [(odd? n) y]) ; substitute 5 for n
⇒ (cond [(odd? 5) y]) ; apply built-in function
⇒ (cond [true y]) ; apply rule for true cond
⇒ y ; substitute 7 for y
⇒ 7
```
Example:

Similar to the previous example except here \( y \) has not been defined.

\[
(\text{define } n \ 5) \ (\text{define } x \ 6)
\
(\text{cond } [(\text{even? } n) \ x][(\text{odd? } n) \ y])
\Rightarrow (\text{cond } [(\text{even? } \ 5) \ x] [(\text{odd? } n) \ y])
\Rightarrow (\text{cond } [\text{false } x][(\text{odd? } n) \ y])
\Rightarrow (\text{cond } [(\text{odd? } n) \ y])
\Rightarrow (\text{cond } [(\text{odd? } \ 5) \ y])
\Rightarrow (\text{cond } [\text{true } y])
\Rightarrow y
\Rightarrow y: \text{ this variable is not defined}
\]
Two types of errors

A **syntax error** occurs when *a sentence cannot be interpreted using the grammar*. Example: \( ) 1 ( 2 + \)

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

Example:

\[
(\text{cond } [(> 3 4) \ x])
\Rightarrow (\text{cond } [\text{false } x])
\Rightarrow (\text{cond } )
\Rightarrow \text{cond: all question results were false}
\]
Simplification 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.
Simplification rules for and and or

9. \((\text{and false } \ldots) \Rightarrow \text{false}\).

10. \((\text{and true } \ldots) \Rightarrow (\text{and } \ldots)\).

11. \((\text{and}) \Rightarrow \text{true}\).

12. \((\text{or true } \ldots) \Rightarrow \text{true}\).

13. \((\text{or false } \ldots) \Rightarrow (\text{or } \ldots)\).

14. \((\text{or}) \Rightarrow \text{false}\).

As in the rewriting rules for \text{cond}, we are using an omission ellipsis.

Note: In DrRacket \text{and} and \text{or} expect two or more arguments.
Simplification Rules for \textbf{and}

- For both DrRacket and CS135: $(\text{and} \; \text{false} \ldots) \Rightarrow \text{false}$
  but for CS135: $(\text{and} \; \text{true} \ldots) \Rightarrow (\text{and} \ldots)$

  i.e. the first argument of \text{and} gets removed. E.g.

  $(\text{and} \; (< \; 1 \; 2) \; (< \; 2 \; 3) \; (< \; 3 \; 4)) \Rightarrow$

  $(\text{and} \; \text{true} \; (< \; 2 \; 3) \; (< \; 3 \; 4)) \Rightarrow$

  $(\text{and} \; (< \; 2 \; 3) \; (< \; 3 \; 4)) \Rightarrow$

  $(\text{and} \; \text{true} \; (< \; 3 \; 4)) \Rightarrow$

  $(\text{and} \; (< \; 3 \; 4)) \Rightarrow$

  $(\text{and} \; \text{true}) \Rightarrow$

  $(\text{and}) \Rightarrow \text{true}$
Simplification Rules for and

• For both DrRacket and CS135: \((\text{and false . . .}) \Rightarrow \text{false}\)
  but for DrRacket: \((\text{and true . . .}) \Rightarrow (\text{and true . . .})\)
  i.e. the first argument of and does not get removed. E.g.
  \((\text{and } (< 1 2) (< 2 3) (< 3 4)) \Rightarrow\)
  \((\text{and true } (< 2 3) (< 3 4)) \Rightarrow\)
  \((\text{and true true } (< 3 4)) \Rightarrow\)
  \((\text{and true true true}) \Rightarrow\)
  true
Simplification Rules for or

- For both DrRacket and CS135: (or true . . . ) ⇒ true
  but for CS135: (or false . . . ) ⇒ (or . . . )
i.e. the first argument of or gets removed. E.g.

  (or (= 1 2) (= 2 3) (≤ 3 4)) ⇒
  (or false (= 2 3) (≤ 3 4)) ⇒
  (or (= 2 3) (≤ 3 4)) ⇒
  (or false (≤ 3 4)) ⇒
  (or (≤ 3 4)) ⇒
  (or true) ⇒
  true
Simplification Rules for or

- For both DrRacket and CS135: \((\text{or} \ \text{false} \ldots) \Rightarrow \text{false}\)
  but for DrRacket: \((\text{or} \ \text{false} \ldots) \Rightarrow (\text{or} \ \text{false} \ldots)\)
  i.e. the first argument of or does not get removed. E.g.
  \[(\text{or} \ (= \ 1 \ 2) (= \ 2 \ 3) (\lt \ 3 \ 4)) \Rightarrow\]
  \[(\text{or} \ \text{false} (= \ 2 \ 3) (\lt \ 3 \ 4)) \Rightarrow\]
  \[(\text{or} \ \text{false} \ \text{false} (\lt \ 3 \ 4)) \Rightarrow\]
  \[(\text{or} \ \text{false} \ \text{false} \ \text{true}) \Rightarrow\]
  true
Substitution rules (so far)

1. Apply functions only when all arguments are values. [Mod 02:13]

2. When given a choice, evaluate the leftmost expression first. [Mod 02:13]

3. \((f \, v_1\ldots v_n) \Rightarrow v\) when \(f\) is built-in function... [10-11]

4. \((f \, v_1\ldots v_n) \Rightarrow \text{exp}'\) when \((\text{define} \ (f \, x_1\ldots x_n) \ \text{exp})\) occurs to the left... [12–14]

5. \(id \Rightarrow \text{val}\) when \((\text{define} \ id \ \text{val})\) occurs to the left. [15–16]
6. (cond [false exp] ...) ⇒ (cond ...) [17–19].

7. (cond [true exp] ...) ⇒ exp [17–19].

8. (cond [else exp]) ⇒ exp [17–19].

9. (and false ...) ⇒ false [21–22].

10. (and true ...) ⇒ (and ...) [21–22].

11. (and) ⇒ true [21–22].

12. (or true ...) ⇒ true [21–22].

13. (or false ...) ⇒ (or ...) [21–22].

14. (or) ⇒ false. [21–22].
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.
Module 05 Summary

Syntax, Semantics and Grammar

1. **Syntax**: The correct format (nothing unexpected or missing). [6]
2. **Semantics**: the meaning of what we say. [6]
3. **Ambiguity**: valid sentences must have exactly one meaning. [6]
4. **Grammars** are formal rules to enforce syntax and prevent ambiguity. [7]
5. A **semantic model** of a programming language specifies a method of predicting the result of running any program. [9]
6. For Racket, our model is to repeatedly simplify the program via substitution. [9]
Module 05 Summary

Errors

7. **Ellipsis** are used to show *several arguments* [11] or to show *omissions*, e.g. the rest of the conditions. [17]

8. A **syntax error** occurs when a sentence cannot be interpreted using the grammar (i.e. it is not in a correct format). [20]

9. A **run-time error** occurs when an expression cannot be reduced to a value by application of our (still incomplete) evaluation rules. [20]
Module 05 Summary

Substitution Rules

10. Our substitution rules for and and or are different from DrRacket’s rules. [22]

11. Know the **Substitution Rules** [23-24] and how to apply them.

12. Doing a step-by-step reduction using to these rules is called **tracing a program**. [25]