Syntax, semantics, and ambiguity

There are three problems we need to address:

• **Syntax**: The way we are allowed to say things.
  
  *is This Sentence Syntactically Correct*

• **Semantics**: the meaning of what we say.
  
  *Trombones fly recursively.*

• **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:

\[(\text{sentence}) = (\text{subject}) (\text{verb}) (\text{object})\]

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:
<def> = (define ( <var> <var> ... <var>) <exp>)

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)
Grammars

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.
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.
Substitution rules (so far)
Substitution 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 $(\text{expt}\ 2\ 10)$.

$(+\ 3\ 5) => 8$
$(\text{expt}\ 2\ 10) => 1024$

Formally, the substitution rule is: $(f\ v1\ \ldots\ vn) => v$, where $f$ is a built-in function and $v$ is the value of $f(v_1, \ldots, v_n)$.

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

For built-in functions $f$ with 1 parameter, the rule is:
$$(f \ v1) \Rightarrow v, \text{ where } v \text{ is the value of } f(v_1)$$

For built-in functions $f$ with 2 parameters, the rule is:
$$(f \ v1 \ v2) \Rightarrow v, \text{ where } v \text{ is the value of } f(v_1, v_2)$$

For built-in functions $f$ with 3 parameters, the rule is:
$$(f \ v1 \ v2 \ v3) \Rightarrow v, \text{ where } v \text{ is the value of } f(v_1, v_2, v_3)$$

We can’t just keep writing down rules forever, so we use ellipses to show a pattern: For built-in functions $f$ with $n$ parameters, the rule is:
$$(f \ v1 \ldots \ vn) \Rightarrow v, \text{ where } v \text{ is the value of } f(v_1, ..., v_n)$$
Substitution of user-defined functions

As an example, 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.
Substitution of user-defined functions

The general substitution rule is

\[(f \ v1 \ldots \ vn) => exp\]

where (\textit{define} (\textit{f} \ x1 \ldots \ xn) \ exp) occurs to the left, and exp is obtained by substituting into the expression exp, with all occurrences of the formal parameter \(x_i\) replaced by the value \(v_1\) (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.
Substitution of user-defined functions – Example

\[(\text{define} \ (\text{term} \ x \ y) \ (** \ x \ (\text{sqr} \ y)))\]

\[(\text{term} \ (- \ 3 \ 1) \ (+ \ 1 \ 2))\]
\[=> \ (\text{term} \ 2 \ (+ \ 1 \ 2))\]
\[=> \ (\text{term} \ 2 \ 3)\]
\[=> \ (** \ 2 \ (\text{sqr} \ 3))\]
\[=> \ (** \ 2 \ 9)\]
\[=> \ 18\]
Substitution of constants

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

We add the substitution rule:

\[ \text{id} \Rightarrow \text{val} \]
where \( (\text{define} \ \text{id} \ \text{val}) \) occurs to the left
Substitution of constants – 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
```
Substitution of \texttt{cond} expressions

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

\begin{itemize}
  \item \((\text{cond} \ [\text{false exp}] \ldots) \Rightarrow (\text{cond} \ldots)\).
  \item \((\text{cond} \ [\text{true exp}] \ldots) \Rightarrow \text{exp}\)
  \item \((\text{cond} \ [\text{else exp}]) \Rightarrow \text{exp}\)
\end{itemize}

These suffice to simplify any \texttt{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 \([\text{false exp}], you just copy it over.”
Substitution of cond expressions – 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 5) y])

=> y

=> 7
Substitution of **cond** expressions – 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 5) y])

=> y

=> y: this variable is not defined
Occurrence of errors

A syntax error occurs when a sentence cannot be interpreted using the grammar, e.g., `(define my-func a b (a + b))`

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 substitution 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** – Example

(\text{and true false true})
(\text{or false true true})

\Rightarrow (\text{and false true})
(\text{or false true true})

\Rightarrow \text{false}
(\text{or false true true})

\Rightarrow \text{false}
(\text{or true true})

\Rightarrow \text{false}
\text{true}

As in the rewriting rules for \textit{cond}, we might use omission ellipses.
## Substitution Rules (so far)

### General rules

1. Apply functions only when all arguments are values.
2. When given a choice, evaluate the leftmost expression first.

### Resolving built-in functions

3. \((f \ v1 \ldots \ vn) \ v\) when \(f\) is built-in

### Resolving define

4. \((f \ v1 \ldots \ vn) \ exp\) when \((\text{define} \ (f \ x1 \ldots \ xn) \ exp)\) occurs to the left
5. \(id \ val\) when \((\text{define} \ id \ val)\) occurs to the left
## Substitution Rules (so far)

### Resolving $\text{cond}$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$(\text{cond} \ [\text{false} \ \text{exp}] \ ...) \ (\text{cond} \ ...))</td>
</tr>
<tr>
<td>7</td>
<td>$(\text{cond} \ [\text{true} \ \text{exp}] \ ...) \ \text{exp}$</td>
</tr>
<tr>
<td>8</td>
<td>$(\text{cond} \ [\text{else} \ \text{exp}]) \ \text{exp}$</td>
</tr>
</tbody>
</table>
## Substitution Rules (so far)

<table>
<thead>
<tr>
<th>Resolving <strong>and</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (and false ...)</td>
<td>false</td>
</tr>
<tr>
<td>10 (and true ...)</td>
<td>(and ...)</td>
</tr>
<tr>
<td>11 (and)</td>
<td>true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resolving <strong>or</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12 (or true ...)</td>
<td>true</td>
</tr>
<tr>
<td>13 (or false ...)</td>
<td>(or ...)</td>
</tr>
<tr>
<td>14 (or)</td>
<td>false</td>
</tr>
</tbody>
</table>
Substitution Rules (so far)

1. Apply functions only when all arguments are values.
2. When given a choice, evaluate the leftmost expression first.
3. \((f \ v_1 \ldots \ v_n) \Rightarrow v\) when \(f\) is built-in
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
5. \(\text{id} \Rightarrow \text{val}\) when \((\text{define} \ \text{id} \ \text{val})\) occurs to the left
6. \((\text{cond} \ [\text{false} \ \text{exp}] \ldots) \Rightarrow (\text{cond} \ldots)\)
7. \((\text{cond} \ [\text{true} \ \text{exp}] \ldots) \Rightarrow \text{exp}\)
8. \((\text{cond} \ [\text{else} \ \text{exp}]) \Rightarrow \text{exp}\)
9. \((\text{and} \ \text{false} \ldots) \Rightarrow \text{false}\)
10. \((\text{and} \ \text{true} \ldots) \Rightarrow (\text{and} \ldots)\)
11. \((\text{and}) \Rightarrow \text{true}\)
12. \((\text{or} \ \text{true} \ldots) \Rightarrow \text{true}\)
13. \((\text{or} \ \text{false} \ldots) \Rightarrow (\text{or} \ldots)\)
14. \((\text{or}) \Rightarrow \text{false}\)
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.
Semantics – End of module
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.