Syntax and Semantics (Ch. 8)

- Vocabulary + Grammar -> Syntax

  - The rules for what is allowed.

- Not all syntactically correct sentences (programs) make sense:

  - "Green is my favorite number."

  - (+ 'Hi 'There)

- There are formal rules that define what is (isn't) syntactically correct.

Syntax vs. Semantics

- Syntax: rules of proper grammar/vocabulary.

- Semantics: meaning of a sentence (program).

- The computer is very good at determining whether or not a program is syntactically correct

  - if not it couldn't even evaluate our program.

- The computer is not very good at determining whether the program semantics are correct (how would the computer even know "what we want the program to do"?

Basic Scheme Vocabulary

```
<var>   =      x | area-of-disk | perimeter | ...

<con>   =    true | false

             'a | 'doll | 'sum | ...

             1 | -1 | 3/5 | 1.22 | ...

<prm>   =    + | - | ...
```

var is *variables*

con is *constants* (also called *literals*)

prm is *primitive operations*

Other parts of a program

- We've also seen things like:

```
(    )    define    cond    else
```

Keywords

- These are punctuation!

  they have no meaning, they just identity context.

- Note: you cannot use a *keyword* as the name of a variable.

Grammar for "Beginning Student Scheme"

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

<exp>  =    <var>

          | <con>

          | (<prm> <exp> ...<exp>)

          | (<var> <exp> ...<exp>)

          | (cond (<exp> <exp>) ...(<exp> <exp>))

          | (cond (<exp> <exp>) ...(else <exp>))
```

Valid Syntax?

```
(x)

(+ 1 (not x))

(+ 1 2 3)


(define (f x) 'x)

(define (f 'x) x)

(define (f x y) (+ 'y (not x)))
```

Program Meaning

- Scheme *arithmetic*

```
(+ 2 3) -> 2      (* 3 4) -> 12

     (not true) -> false

(symbol=? 'a 'b) -> false
```

And lots of others...

More program meaning

- algebraic substitution:

```
(define (f x-1 ... x-n)
  exp)
```

```
(f v-1 ... v-n) ->
```
  exp with all `x-1 ... x-n` replaced by `v-1 ... v-n`

Substitution Example

```
(define (poly x y)
  (+ (expt 2 x) y))




(poly 3 5) ->
  (+ (expt 2 3) 5)) ->
  (+ 8 5)  ->
  13
```

and more Program Meaning: cond

cond_false: when the first condition is false:

```
  (cond

    [false ...]

    [exp1 exp2]

    ...)   ->

  (cond

      ; The first line disappeared.

      [exp1 exp2]

      ...)
```

More of cond

cond_true: when the first condition is true:

```
(cond

   [true exp]

    ...)  ->   exp
```

cond_else: else is the only line left

```
(cond

   [else exp])  ->   exp
```

Some examples to try

```
(cond
  [false 1]
  [true (+ 1 1)]
  [else 3])


(cond
    [(= 1 0) 0]
    [else (+ 1 1)])
```

More things to try

```
(+ (* (/ 12 8) 2/3)

   (- 20 (sqrt 4)))



(cond

      [(= 2 0) false]

      [(> 2 1) (symbol=? 'a 'a)]

      [else (= (/  1 2) 9)]) ;
```

and more...

```
(define (f x y)
  (+ (* 3 x) (* y y)))


(+ (f 1 2) (f 2 1))


(f 1 (* 2 3))


(f (f 1 (* 2 3)) 19)
```

Errors

- Syntax Errors: DrScheme will complain before attempting to evaluate anything.

- Logic/Runtime Errors
  - `(/ 1 0)     (+ 'Fred 'Hello)`

  - These are not errors until they have been evaluted!

Error or not?

```
;; my-divide : number  ->  number
(define (my-divide n)
  (cond
    [(= n 0) 'inf]
    [else (/ 1 n)]))


(my-divide 0)
```

Scheme Expression Evalution

- Rule of thumb:

Simplify the outermost (and left-most) subexpression that is ready for evaluation.

Raising an Error

- Programmer can include expressions that tell scheme to signal that an error has occurred.

```
(define (add x y)

    (cond

        [ (and (number? x) (number? y))

            (+ x y) ]

        [ else (error 'add "numbers expected")]))

(add 'hi fred)

add: numbers expected
```

Boolean Functions

- We did not explicitly look at the functions `and` and `or`.

  - these are actually special and we need to understand the rules.

```
(and exp1 exp2)
```

 exp2 is never evaluated if `exp1` is `false`

```
(or exp1 exp2)
```

 exp2 is never evaluated if `exp1` is `true`

Try these:

```
(and (= 0 1) (/ 1 0))

(and (= 0 0) (/ 1 0))

(or (= 0 1) (/ 1 0))

(or (= 0 0) (/ 1 0))
```

Variable definition

```
(define varname exp)
```

Very different than function definition!

   scheme evaluates exp right away.

```
(define foo (/ 1 0))
(define (foo x (/ x 0)))
```

Structures

- Define-struct is also special:

```
(define-struct  sname (fld1 fld2 ...))
```

Not only is evaulation of this expression special, it also results in the creation of new primitive operations:

```
make-sname

sname-fld1

sname-fld2  ...

sname?
```

Complete Grammar (Beginning Student)

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

        |  (define <var> <exp>)

        |  (define-struct <var0> (<var-1> ...<var-n>))

<exp>   =   <var>

        |  <con>

        |  (<prm> <exp> ...<exp>)

        |  (<var> <exp> ...<exp>)

        |  (cond (<exp> <exp>) ...(<exp> <exp>))

        |  (cond (<exp> <exp>) ...(else <exp>))

        |  (and <exp> <exp>)

        |  (or <exp> <exp>)
```