Semantics (= Evaluation)

[[[PLAI Chapter 2]]]

Back to BNF -- now, **meaning**.

An important feature of these BNF specifications: we can use the derivations to specify *meaning* (and meaning in our context is "running" a program (or "interpreting", "compiling", but we will use "evaluating")). For example:

```
  <AE> ::= <num>         ; <AE> evaluates to the number
         | <AE1> + <AE2> ; <AE> evaluates to the sum of
evaluating
                         ;      <AE1> and <AE2>
         | <AE1> - <AE2> ; ... the subtraction of <AE2>
from <AE1>
                                 (... roughly!)
```

To do this a little more formally:

```
  a. eval(<num>) = <num> ; <-- special rule: moves syntax
into a value
  b. eval(<AE1> + <AE2>) = eval(<AE1>) + eval(<AE2>)
  c. eval(<AE1> - <AE2>) = eval(<AE1>) - eval(<AE2>)
```

Note the completely different roles of the two "+"s and "-"s. In fact, it might have been more correct to write:

```
  a. eval("<num>") = <num>
  b. eval("<AE1> + <AE2>") = eval("<AE1>") + eval("<AE2>")
  c. eval("<AE1> - <AE2>") = eval("<AE1>") - eval("<AE2>")
```

or even using a marker to denote meta-holes in these strings:

```
  a. eval("$<num>") = <num>
  b. eval("$<AE1> + $<AE2>") = eval("$<AE1>") +
eval("$<AE2>")
  c. eval("$<AE1> - $<AE2>") = eval("$<AE1>") -
eval("$<AE2>")
```

but we will avoid pretending that we're doing that kind of string manipulation. (For example, it will require specifying what does it mean to return <num> for "$<num>" (involves `string->number'), and the fragments on the right side mean that we need to specify these as substring operations.)

Note that there's a similar kind of informality in our BNF specifications, where we assume that "<foo>" refers to some terminal or non-terminal. In texts, where more formality is required (for example, in RFC specifications), each literal part of the BNF is usually marked with double quotes, so we'd get

```
  <AE> ::= <num> | <AE1> "+" <AE2> | <AE1> "-" <AE2>
```

An alternative popular notation for eval(X) is [[X]]:

```
  a. [[<num>]] = <num>
  b. [[<AE1> + <AE2>]] = [[<AE1>]] + [[<AE2>]]
  c. [[<AE1> - <AE2>]] = [[<AE1>]] - [[<AE2>]]
```

Is there a problem with this definition? **Ambiguity**:

```
  eval(1 - 2 + 3) = ?
```

Depending on the way the expression is parsed, we get either 2 or -4:

```
  eval(1 - 2 + 3) = eval(1 - 2) + eval(3)           [b]
                  = eval(1) - eval(2) + eval(3)     [c]
                  = 1 - 2 + 3                       [a,a,a]
                  = 2

  eval(1 - 2 + 3) = eval(1) - eval(2 + 3)           [c]
                  = eval(1) - (eval(2) + eval(3))   [a]
                  = 1 - (2 + 3)                     [a,a,a]
                  = -4
```

Again, be very aware of confusing subtleties which are extremely important: We need parens around a sub-expression only on one side, why?
-- When we write:

```
  eval(1 - 2 + 3) = ... = 1 - 2 + 3
```

we have two expressions, but one stands for an *input syntax*, and one stands for a `real' mathematical expression.

In a case of a computer implementation, the syntax on the left is (as always) an AE syntax, and the `real' expression on the right is an expression in whatever language we use to implement our AE language.

Like we said earlier, ambiguity is **not a real problem** until the actual parse tree matters. With `**eval**' it **definitely matters**, so we must **not** make it possible to derive any syntax in multiple ways or our evaluation will be non-deterministic.

Compositionality – an important feature of a syntax

(Compositionality: The meaning of a complex expression is determined by its structure and the meanings of its constituents.)

Quick exercise:

We can define a meaning for <digit>s and then <num>s in a similar way:

```
  <NUM> ::= <digit> | <digit> <NUM>

  eval(0) = 0
  eval(1) = 1
  eval(2) = 2
  ...
  eval(9) = 9

  eval(<digit>) = <digit>
  eval(<digit> <NUM>) = 10*eval(<digit>) + eval(<NUM>)
```

Is this exactly what we want? -- Depends on what we actually want...

* First, there's a bug in this code -- having a BNF derivation like

```
    <NUM> ::= <digit> | <digit> <NUM>
```

  is unambiguous, but makes it **hard** to parse a number. We get:

```
     eval(123) = 10*eval(1) + eval(23)
               = 10*1 + 10*eval(2) + eval(3)
```

```
                = 10*1 + 10*2 + 3
                = 33
```

  Changing the order of the last rule works much better:

```
    <NUM> ::= <digit> | <NUM> <digit>
```

  and then:

```
    eval(<NUM> <digit>) = 10*eval(<NUM>) + eval(<digit>)
```

* As a concrete example see how you would make it work with "107", which demonstrates how the compositionality is important.

* Example for free stuff that looks trivial: if we were to define the meaning of numbers this way, would it always work? Think of an average language that does not give you **bignums**, making the above rules fail when the numbers are too big. In Racket, we happen to be using an integer representation for the syntax of integers, and both are unlimited. But what if we wanted to write a Racket compiler in C or a C compiler in Racket? What about a C compiler in C, where the compiler runs on a 64 bit machine, and the result needs to run on a 32 bit machine?

Side comment on compositionality

The example of

```
    <NUM> ::= <digit> | <NUM> <digit>
```

being a language that is easier to write an evaluator for leads us to an important concept -- **compositionality.** This definition is easier to write an evaluator for, since the **resulting language is compositional:** the meaning of an expression -- for example \`123' -- is composed out of the meaning of its two parts, which in this BNF are \`12' and \`3'.

Specifically, the evaluation of `<NUM> <digit>' is 10 * the evaluation of the first, plus the evaluation of the second. In the `<digit> <NUM>' case this is more difficult -- the meaning of such a number **depends not only on the *meaning* of the two parts**, but also on the `<NUM>' *syntax*:

eval(<digit> <NUM>) = eval(<digit>) * 10^length(<NUM>) + eval(<NUM>)

This case can be **tolerable**, since the meaning of the expression is still made out of its parts -- but **imperative programming** (when you use side effects) is much more problematic since it is not compositional (at least not in the obvious sense). This is compared to functional programming, where the meaning of an expression is a combination of the meanings of its sub-expressions. For example, every sub-expression in a functional program has some known meaning, and these all make up the meaning of the expression that contains them -- but in an imperative program we can have a part of the code be `x++' – and that doesn't have a meaning by itself, at least not one that contributes to the meaning of the whole program in a direct way.

(Actually, we can have a well-defined meaning for such an expression: the meaning is going from a world where `x' is a container of some value N, to a world where the same container has a different value N+1. You can probably see now how this can make things more complicated. On an intuitive level -- **if we look at a random part of a functional program we can tell its meaning**, so building up the meaning of the whole code is easy, but **in an imperative program, the meaning of a random part is pretty much useless.)**

Implementing an Evaluator

Now continue to implement the semantics of our syntax -- we express that through an `**eval**' function that evaluates an expression.

We use a basic programming principle -- **splitting the code into two layers**, one for parsing the input, and one for doing the evaluation. Doing this avoids the mess we'd get into otherwise, for example:

```
  (define (eval sexpr)
    (match sexpr
      [(number: n) n]
      [(list '+ left right) (+ (eval left) (eval right))]
      [(list '- left right) (- (eval left) (eval right))]
      [else (error 'eval "bad syntax in ~s" sexpr)]))
```

This is messy because it **combines two very different things** - **syntax** and **semantics** -- into a single lump of code. For this particular kind of evaluator it looks simple enough, but this is only because it's simple enough that all we do is replace constructors by arithmetic operations. Later on things will get more complex, and bundling the evaluator with the parser will be more problematic. (Note: The fact, that we can replace constructors with the run-time operators, means that we have a very simple, calculator-like language, and that we can, in face, "compile" all programs down to a number.)

If we split the code, we can easily **include decisions** like making

```
  {+ 1 {- 3 "a"}}
```

syntactically invalid. (Which is not, BTW, what Racket does...) (Also, this is like the distinction between XML syntax and well-formed XML syntax.)

An additional advantage is that by using two separate components, **it is simple to replace each one**, making it possible to change the input syntax, and the semantics independently -- we only need to keep the same interface data (the AST) and things will work fine.

Our `**parse**' function converts an input syntax to an abstract syntax tree (AST). It is abstract exactly because

it is independent of any actual concrete syntax that you
type in, print out etc.

Implementing The AE Language

Back to our `eval' -- this will be its (obvious) type:

```
  (: eval : AE -> Number)
  ;; consumes an AE and computes the corresponding number
```

which leads to some obvious test cases (using the new
`test' form that the `#lang pl' language provides):

```
  (test (eval (parse "3"))               => 3)
  (test (eval (parse "{+ 3 4}"))         => 7)
  (test (eval (parse "{+ {- 3 4} 7}")) => 6)
```

Note that we're testing *only* at the interface level --
only running whole functions. For example, you could think
about a test like:

```
  (test (parse "{+ {- 3 4} 7}")
        => (Add (Sub (Num 3) (Num 4)) (Num 7)))
```

but the details of parsing and of the constructor names are
things that nobody outside of our evaluator cares about --
so we're not testing them. In fact, we shouldn't even
mention `**parse**' in these tests, since it is not part of the
public interface of our users; they only care about using
it as a compiler-like black box. (This is sometimes called
"integration tests".) We'll address this shortly.

Like everything else, the structure of the recursive `eval'
code follows the recursive structure of its input. In HtDP
terms, our template is:

```
  (: eval : AE -> Number)
  (define (eval expr)
    (cases expr
      [(Num n)    ... n ...]
      [(Add l r) ... (eval l) ... (eval r) ...]
      [(Sub l r) ... (eval l) ... (eval r) ...]))
```

In this case, filling in the gaps is very simple

```
  (: eval : AE -> Number)
  (define (eval expr)
    (cases expr
      [(Num n)   n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]))
```

We now further combine `eval' and `parse' into a single `run' function that evaluates an AE string.

```
  (: run : String -> Number)
  ;; evaluate an AE program contained in a string
  (define (run str)
    (eval (parse str)))
```

This function becomes the single public entry point into our code, and the only thing that should be used in tests that verify our interface:

```
  (test (run "3")               => 3)
  (test (run "{+ 3 4}")         => 7)
  (test (run "{+ {- 3 4} 7}") => 6)
```

The resulting *full* code is:

<<<AE>>>

```
  #lang pl

  #| BNF for the AE language:
     <AE> ::= <num>
            | { + <AE> <AE> }
            | { - <AE> <AE> }
            | { * <AE> <AE> }
            | { / <AE> <AE> }
  |#

  ;; AE abstract syntax trees
  (define-type AE
    [Num Number]
    [Add AE AE]
    [Sub AE AE]
    [Mul AE AE]
    [Div AE AE])
  (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n) (Num n)]
      [(list '+ lhs rhs)
       (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs)
       (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs)
       (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs)
       (Div (parse-sexpr lhs) (parse-sexpr rhs))]
      [else
       (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

  (: parse : String -> AE)
  ;; parses a string containing an AE expression to an AE
AST
  (define (parse str)
    (parse-sexpr (string->sexpr str)))

  (: eval : AE -> Number)
  ;; consumes an AE and computes the corresponding number
  (define (eval expr)
    (cases expr
      [(Num n)   n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]
      [(Mul l r) (* (eval l) (eval r))]
      [(Div l r) (/ (eval l) (eval r))]))

  (: run : String -> Number)
  ;; evaluate an AE program contained in a string
  (define (run str)
    (eval (parse str)))


  ;; tests
  (test (run "3") => 3)
  (test (run "{+ 3 4}") => 7)
  (test (run "{+ {- 3 4} 7}") => 6)

  ---------------------------------------------------------
```

(Note that the tests are done with a `**test**' form, which we mentioned above.)

For anyone who thinks that Racket is a bad choice, this is a good point to think how much code would be needed in some other language to do the same as above.