PL: Lecture #4

Tuesday, September 17th

The `define-type` Form

The class language that we're using, `#lang pl`, is based on *Typed Racket*: a statically-typed dialect of Racket. It is not exactly the same as Typed Racket — it is restricted in many ways, and extended in a few ways. (You should therefore try to avoid looking at the Typed Racket documentation and expect things to be the same in `#lang pl`.)

The most important extension is `define-type`, which is the construct we will be using to create new user-defined types. In general, such definitions looks like what we just used:

```
(define-type AE
  [Num Number]
  [Add AE AE]
  [Sub AE AE])
```

This defines a *new type* called `AE`, an `AE?` predicate for this type, and a few *variants* for this type: `Num`, `Add`, and `Sub` in this case. Each of these variant names is a constructor, taking in arguments with the listed types, where these types can include the newly defined type itself in (the very common) case we're defining a recursive type. The return type is always the newly defined type, `AE` here.

To summarize, this definition gives us a new `AE` type, and three constructors, as if we wrote the following type declarations:

- `(: Num : Number -> AE)`
- `(: Add : AE AE -> AE)`
- `(: Sub : AE AE -> AE)`

The newly defined types are known as *"disjoint unions"*, since values in these types are disjoint — there is no overlap between the different variants. As we will see, this is what makes this such a useful construct for our needs: the compiler knows about the variants of each newly defined type, which will make it possible for it to complain if we extend a type with more variants but not update all uses of the type.

Furthermore, since the return types of these constructors are all the new type itself, there is *no way* for us to write code that expects just *one* of these variants. We will use a second form, `cases`, to handle these values.

The `cases` Form

A `define-type` declaration defines *only* what was described above: one new type name and a matching predicate, and a few variants as constructor functions. Unlike HtDP, we don't get predicates for each of the variants, and we don't get accessor functions for the fields of the variants.

The way that we handle the new kind of values is with `cases`: this is a form that is very similar to `match`, but is specific to instances of the user-defined type.

> *Many students find it confusing to distinguish `match` and `cases` since they are so similar. Try to remember that `match` is for primitive Racket values (we'll mainly use them for S-expression values), while `cases` is for user-defined values. The distinction between the two forms is unfortunate, and doesn't serve any purpose. It is just technically difficult to unify the two.*

For example, code that handles `AE` values (as defined above) can look as follows:

```
(cases some-ae-value
  [(Num n)   "a number"]
  [(Add l r) "an addition"]
  [(Sub l r) "a subtraction"])
```

As you can see, we need to have patterns for each of the listed variants (and the compiler will throw an error if some are missing), and each of these patterns specifies bindings that will get the field values contained in a given variant object.

We can also use nested patterns:

```
(cases some-ae-value
  [(Num n)                 "a number"]
  [(Add (Num m) (Num n)) "a simple addition"]
  [(Add l r)               "an addition"]
  [(Sub (Num m) (Num n)) "a simple subtraction"]
  [(Sub l r)               "a subtraction"])
```

but this is a feature that we will not use too often.

The final clause in a `cases` form can be an `else` clause, which serves as a fallback in case none of the previous clauses matched the input value. However, using an `else` like this is ***strongly discouraged!*** The problem with using it is that it effectively eliminates the advantage in getting the type-checker to complain when a type definition is extended with new variants. Using these `else` clauses, we can actually mimic all of the functionality that you expect in HtDP-style code, which demonstrates that this is equivalent to HtDP-style definitions. For example:

```
(: Add? : AE -> Boolean)
;; identifies instances of the `Add` variant
(define (Add? ae)
  (cases ae
    [(Add l r) #t]
    [else #f]))

(: Add-left : AE -> AE)
;; get the left-hand subexpression of an addition
(define (Add-left ae)
  (cases ae
    [(Add l r) l]
    [else (error 'Add-left "expecting an Add value, got ~s" ae)]))

...
```

Important reminder: this is code that ***you should not write!*** Doing so will lead to code that is more fragile than just using `cases`, since you'd be losing the protection the compiler gives you in the form of type errors on occurrences of `cases` that need to be updated when a type is extended with new variants. You would therefore end up writing a bunch of boiler-plate code only to end up with lower-quality code. The core of the problem is in the prevalent use of `else` which gives up that protection.

In these examples the `else` clause is justified because even if `AE` is extended with new variants, functions like `Add?` and `Add-left` should not be affected and treat the new variants as they treat all other non-`Add` instances. (And since `else` is inherent to these functions, using them in our code is inherently a bad idea.) We will, however, have a few (*very few!*) places where we'll need to use `else` — but this will always be done only on some specific functionality rather than a wholesale approach of defining a different interface for user-defined types.

Semantics (= Evaluation)

👉 ***PLAI §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: translate syntax to 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 that require more formal specifications (for example, in RFC specifications), each literal part of the BNF is usually double-quoted, 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 can get either a result of `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 in 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.

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 why 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 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-note: 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 this case this 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 subexpressions. 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 mean that we have a very simple, calculator-like language, and that we can, in fact, "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:

```
(equal? 3 (eval (parse "3")))
(equal? 7 (eval (parse "{+ 3 4}")))
(equal? 6 (eval (parse "{+ {- 3 4} 7}")))
```

which from now on we will write 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:

```
#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)
;; parses 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.