PL: Lecture #3

Tuesday, September 17th

BNF, Grammars, the AE Language

Getting back to the theme of the course: we want to investigate programming languages, and we want to do that *using* a programming language.

The first thing when we design a language is to specify the language. For this we use BNF (Backus-Naur Form). For example, here is the definition of a simple arithmetic language:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

Explain the different parts. Specifically, this is a mixture of low-level (concrete) syntax definition with parsing.

We use this to derive expressions in some language. We start with `<AE>`, which should be one of these:

- a number `<num>`
- an `<AE>`, the text "`+`", and another `<AE>`
- the same but with "`-`"

`<num>` is a terminal: when we reach it in the derivation, we're done. `<AE>` is a non-terminal: when we reach it, we have to continue with one of the options. It should be clear that the `+` and the `-` are things we expect to find in the input — because they are not wrapped in `<>`s.

We could specify what `<num>` is (turning it into a `<NUM>` non-terminal):

```
<AE> ::= <NUM>
       | <AE> + <AE>
       | <AE> - <AE>

<NUM> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
        | <NUM> <NUM>
```

But we don't — why? Because in Racket we have numbers as primitives and we want to use Racket to implement our languages. This makes life a lot easier, and we get free stuff like floats, rationals etc.

To use a BNF formally, for example, to prove that `1-2+3` is a valid `<AE>` expression, we first label the rules:

```
<AE> ::= <num>        (1)
       | <AE> + <AE>  (2)
       | <AE> - <AE>  (3)
```

and then we can use them as formal justifications for each derivation step:

```
<AE>
<AE> + <AE>         ; (2)
<AE> + <num>        ; (1)
<AE> - <AE> + <num> ; (3)
<AE> - <AE> + 3     ; (num)
<num> - <AE> + 3    ; (1)
<num> - <num> + 3   ; (1)
```

```
    1 - <num> + 3        ; (num)
    1 - 2 + 3            ; (num)
```

This would be one way of doing this. Alternatively, we can can visualize the derivation using a tree, with the rules used at the nodes.

These specifications suffer from being ambiguous: an expression can be derived in multiple ways. Even the little syntax for a number is ambiguous — a number like `123` can be derived in two ways that result in trees that look different. This ambiguity is not a “real” problem now, but it will become one very soon. We want to get rid of this ambiguity, so that there is a single (= deterministic) way to derive all expressions.

There is a standard way to resolve that — we add another non-terminal to the definition, and make it so that each rule can continue to exactly one of its alternatives. For example, this is what we can do with numbers:

```
    <NUM>   ::= <DIGIT> | <DIGIT> <NUM>

    <DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Similar solutions can be applied to the `<AE>` BNF — we either restrict the way derivations can happen or we come up with new non-terminals to force a deterministic derivation trees.

As an example of restricting derivations, we look at the current grammar:

```
    <AE> ::= <num>
           | <AE> + <AE>
           | <AE> - <AE>
```

and instead of allowing an `<AE>` on both sides of the operation, we force one to be a number:

```
    <AE> ::= <num>
           | <num> + <AE>
           | <num> - <AE>
```

Now there is a single way to derive any expression, and it is always associating operations to the right: an expression like `1+2+3` can only be derived as `1+(2+3)`. To change this to left-association, we would use this:

```
    <AE> ::= <num>
           | <AE> + <num>
           | <AE> - <num>
```

But what if we want to force precedence? Say that our AE syntax has addition and multiplication:

```
    <AE> ::= <num>
           | <AE> + <AE>
           | <AE> * <AE>
```

We can do that same thing as above and add new non-terminals — say one for “products”:

```
    <AE>   ::= <num>
             | <AE> + <AE>
             | <PROD>

    <PROD> ::= <num>
             | <PROD> * <PROD>
```

Now we must parse any AE expression as additions of multiplications (or numbers). First, note that if `<AE>` goes to `<PROD>` and that goes to `<num>`, then there is no need for an `<AE>` to go to a `<num>`, so this is the same syntax:

```
    <AE>   ::= <AE> + <AE>
             | <PROD>
```

```
<PROD> ::= <num>
         | <PROD> * <PROD>
```

Now, if we want to still be able to multiply additions, we can force them to appear in parentheses:

```
<AE>   ::= <AE> + <AE>
         | <PROD>

<PROD> ::= <num>
         | <PROD> * <PROD>
         | ( <AE> )
```

Next, note that `<AE>` is still ambiguous about additions, which can be fixed by forcing the left hand side of an addition to be a factor:

```
<AE>   ::= <PROD> + <AE>
         | <PROD>

<PROD> ::= <num>
         | <PROD> * <PROD>
         | ( <AE> )
```

We still have an ambiguity for multiplications, so we do the same thing and add another non-terminal for "atoms":

```
<AE>   ::= <PROD> + <AE>
         | <PROD>

<PROD> ::= <ATOM> * <PROD>
         | <ATOM>

<ATOM> ::= <num>
         | ( <AE> )
```

And you can try to derive several expressions to be convinced that derivation is always deterministic now.

But as you can see, this is exactly the cosmetics that we want to avoid — it will lead us to things that might be interesting, but unrelated to the principles behind programming languages. It will also become much much worse when we have a real language rather such a tiny one.

Is there a good solution? — It is right in our face: do what Racket does — always use fully parenthesized expressions:

```
<AE> ::= <num>
       | ( <AE> + <AE> )
       | ( <AE> - <AE> )
```

To prevent confusing Racket code with code in our language(s), we also change the parentheses to curly ones:

```
<AE> ::= <num>
       | { <AE> + <AE> }
       | { <AE> - <AE> }
```

But in Racket *everything* has a value — including those `+`s and `-`s, which makes this extremely convenient with future operations that might have either more or less arguments than 2 as well as treating these arithmetic operators as plain functions. In our toy language we will not do this initially (that is, `+` and `-` are second order operators: they cannot be used as values). But since we will get to it later, we'll adopt the Racket solution and use a fully-parenthesized prefix notation:

```
<AE> ::= <num>
       | { + <AE> <AE> }
       | { - <AE> <AE> }
```

(Remember that in a sense, Racket code is written in a form of already-parsed syntax…)

Simple Parsing

On to an implementation of a “parser”:

Unrelated to what the syntax actually looks like, we want to parse it as soon as possible — converting the concrete syntax to an abstract syntax tree.

No matter how we write our syntax:

- `3+4` (infix),
- `3 4 +` (postfix),
- `+(3,4)` (prefix with args in parens),
- `(+ 3 4)` (parenthesized prefix),

we always mean the same abstract thing — adding the number `3` and the number `4`. The essence of this is basically a tree structure with an addition operation as the root and two leaves holding the two numerals.

With the right data definition, we can describe this in Racket as the expression `(Add (Num 3) (Num 4))` where `Add` and `Num` are constructors of a tree type for syntax, or in a C-like language, it could be something like `Add(Num(3),Num(4))`.

Similarly, the expression `(3-4)+7` will be described in Racket as the expression:

```
(Add (Sub (Num 3) (Num 4)) (Num 7))
```

Important note: “expression” was used in two *different* ways in the above — each way corresponds to a different language, and the result of evaluating the second “expression” is a Racket value that *represents* the first expression.

To define the data type and the necessary constructors we will use this:

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

- Note — Racket follows the tradition of Lisp which makes syntax issues almost negligible — the language we use is almost as if we are using the parse tree directly. Actually, it is a very simple syntax for parse trees, one that makes parsing extremely easy.

  [This has an interesting historical reason… Some Lisp history — *M-expressions* vs. *S-expressions*, and the fact that we write code that is isomorphic to an AST. Later we will see some of the advantages that we get by doing this. See also “*The Evolution of Lisp*”, section 3.5.1. Especially the last sentence:

  > *Therefore we expect future generations of Lisp programmers to continue to reinvent Algol-style syntax for Lisp, over and over and over again, and we are equally confident that they will continue, after an initial period of infatuation, to reject it. (Perhaps this process should be regarded as a rite of passage for Lisp hackers.)*

  And an interesting & modern *counter*-example of this **here**.]

To make things very simple, we will use the above fact through a double-level approach:

- we first “parse” our language into an intermediate representation — a Racket list — this is mostly done by a modified version of Racket’s **`read`** function that uses curly `{}` braces instead of round `()` parens,
- then we write our own **`parse`** function that will parse the resulting list into an instance of the `AE` type — an abstract syntax tree (AST).

This is achieved by the following simple recursive function:

```
(: parse-sexpr : Sexpr -> AE)
;; parses s-expressions into AEs
(define (parse-sexpr sexpr)
  (cond [(number? sexpr) (Num sexpr)]
        [(and (list? sexpr) (= 3 (length sexpr)))
         (let ([make-node
                (match (first sexpr)
                  ['+ Add]
                  ['- Sub]
                  [else (error 'parse-sexpr "unknown op: ~s"
                               (first sexpr))])
                #| the above is the same as:
                (cond [(equal? '+ (first sexpr)) Add]
                      [(equal? '- (first sexpr)) Sub]
                      [else (error 'parse-sexpr "unknown op: ~s"
                                   (first sexpr))])
                |#])
           (make-node (parse-sexpr (second sexpr))
                      (parse-sexpr (third sexpr))))]
        [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

This function is pretty simple, but as our languages grow, they will become more verbose and more difficult to write. So, instead, we use a new special form: `match`, which is matching a value and binds new identifiers to different parts (try it with “Check Syntax”). Re-writing the above code using `match`:

```
(: parse-sexpr : Sexpr -> AE)
;; parses s-expressions into AEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    [(list '+ left right)
     (Add (parse-sexpr left) (parse-sexpr right))]
    [(list '- left right)
     (Sub (parse-sexpr left) (parse-sexpr right))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

And finally, to make it more uniform, we will combine this with the function that parses a string into a sexpr so we can use strings to represent our programs:

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

The `match` Form

The syntax for `match` is

```
(match value
  [pattern result-expr]
  ...)
```

The value is matched against each pattern, possibly binding names in the process, and if a pattern matches it evaluates the result expression. The simplest form of a pattern is simply an identifier — it always matches and binds that identifier to the value:

```
(match (list 1 2 3)
  [x x]) ; evaluates to the list
```

Another simple pattern is a quoted symbol, which matches that symbol. For example:

```
(match foo
  ['x "yes"]
  [else "no"])
```

will evaluate to `"yes"` if `foo` is the symbol `x`, and to `"no"` otherwise. Note that `else` is not a keyword here — it happens to be a pattern that always succeeds, so it behaves like an else clause except that it binds `else` to the unmatched-so-far value.

Many patterns look like function application — but don't confuse them with applications. A `(list x y z)` pattern matches a list of exactly three items and binds the three identifiers; or if the "arguments" are themselves patterns, `match` will descend into the values and match them too. More specifically, this means that patterns can be nested:

```
(match (list 1 2 3)
  [(list x y z) (+ x y z)]) ; evaluates to 6
(match (list 1 2 3)
  [(cons x (list y z)) (+ x y z)]) ; matches the same shape (also 6)
(match '((1) (2) 3)
  [(list (list x) (list y) z) (+ x y z)]) ; also 6
```

As seen above, there is also a `cons` pattern that matches a non-empty list and then matches the first part against the head for the list and the second part against the tail of the list.

In a `list` pattern, you can use `...` to specify that the previous pattern is repeated zero or more times, and bound names get bound to the list of respective matching. One simple consequent is that the `(list hd tl ...)` pattern is exactly the same as `(cons hd tl)`, but being able to repeat an arbitrary pattern is very useful:

```
> (match '((1 2) (3 4) (5 6) (7 8))
    [(list (list x y) ...) (list x y)])
'((1 3 5 7) (2 4 6 8))
```

A few more useful patterns:

```
id              -- matches anything, binds `id' to it
_               -- matches anything, but does not bind
(number: n)     -- matches any number and binds it to `n'
(symbol: s)     -- same for symbols
(string: s)     -- strings
(sexpr: s)      -- S-expressions (needed sometimes for Typed Racket)
(and pat1 pat2) -- matches both patterns
(or pat1 pat2)  -- matches either pattern (careful with bindings)
```

Note that the `foo:` patterns are all specific to our `#lang pl`, they are not part of `#lang racket` or `#lang typed/racket`.

The patterns are tried one by one *in-order*, and if no pattern matches the value, an error is raised.

Note that `...` in a `list` pattern can follow *any* pattern, including all of the above, and including nested list patterns.

Here are a few examples — you can try them out with `#lang pl untyped` at the top of the definitions window. This:

```
(match x
  [(list (symbol: syms) ...) syms])
```

matches `x` against a pattern that accepts only a list of symbols, and binds `syms` to those symbols. If you want to match only a list of, say, one or more symbols, then just add one before the `...`-ed pattern variable:

```
(match x
  [(list (symbol: sym) (symbol: syms) ...) syms])
;; same as:
(match x
  [(cons (symbol: sym) (list (symbol: syms) ...)) syms])
```

which will match such a non-empty list, where the whole list (on the right hand side) is `(cons sym syms)`.

Here's another example that matches a list of any number of lists, where each of the sub-lists begins with a symbol and then has any number of numbers. Note how the `n` and `s` bindings get values for a list of all symbols and a list of lists of the numbers:

```
> (define (foo x)
    (match x
      [(list (list (symbol: s) (number: n) ...) ...)
       (list 'symbols: s 'numbers: n)]))
> (foo (list (list 'x 1 2 3) (list 'y 4 5)))
'(symbols: (x y) numbers: ((1 2 3) (4 5)))
```

Here is a quick example for how `or` is used with two literal alternatives, how `and` is used to name a specific piece of data, and how `or` is used with a binding:

```
> (define (foo x)
    (match x
      [(list (or 1 2 3)) 'single]
      [(list (and x (list 1 _)) 2) x]
      [(or (list 1 x) (list 2 x)) x]))
> (foo (list 3))
'single
> (foo (list (list 1 99) 2))
'(1 99)
> (foo (list 1 10))
10
> (foo (list 2 10))
10
```