Parsing expressions top down
Supplementary notes for lecture 9, CS 421, Spring 2013

Prof. Kamin

January 19, 2013

Toward the end of lecture 9, we discussed how to use top-down parsing for expressions. The purpose of this note is just to give the entire example.

To repeat: We want a left-recursive expression grammar, because then the shape of the (concrete) parse tree will enforce left-associativity, which is the correct associativity for almost all operations. However, a left-recursive grammar cannot be parsed top-down. So, we are stuck with right-recursive grammars, which means we are stuck with right-associative operations, which is wrong. The point about the final part of lecture 9 is that we can handle this by going ahead and parsing with the right-recursive grammar but then re-balancing the AST to "lean left." Here we give the details of our simple example.

First, start with a right-recursive, stratified grammar for plus and times. This grammar enforces precedence of times, but also gives right-associativity for both plus and times:

$$\begin{array}{l} \mathrm{E} \rightarrow \mathrm{T} + \mathrm{E} \mid \mathrm{T} \\ \mathrm{T} \rightarrow \mathrm{id} \mid \mathrm{id} * \mathrm{T} \end{array}$$

As noted in class, this grammar is not LL(1), but it can be transformed to an LL(1) grammar by left-factoring:

$$\begin{array}{l} \mathrm{E} \rightarrow \mathrm{T} + \mathrm{E} \mid \mathrm{T} \\ \mathrm{T} \rightarrow \mathrm{id} \mid \mathrm{id} * \mathrm{T} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} \mathrm{E} & \rightarrow & \mathrm{T\ E'} \\ \mathrm{E'} & \rightarrow & \epsilon \mid + \mathrm{E} \\ \mathrm{T} & \rightarrow & \mathrm{id\ T'} \\ \mathrm{T'} & \rightarrow & \epsilon \mid * \mathrm{T} \end{array}$$

Here is a recognizer for this grammar, written in the straightforward style we introduced at the start of class:

```
type token = IDENT of string | PLUS | TIMES | EOF

let rec parseE toklis = parseE' (parseT toklis)

and parseE' toklis = if hd toklis = PLUS then parseE (tl toklis) else toklis

and parseT toklis = parseT' (parseIDENT toklis)

and parseT' toklis = if hd toklis = TIMES then parseT (tl toklis) else toklis

and parseIDENT toklis = match hd toklis with IDENT _ -> tl toklis
                                           | _ -> raise SyntaxError ;;
```

So far, so good. ASTs for this grammar are really simple: they are trees whose leaf nodes are identifiers and whose internal nodes are either pluses or timeses; since we're already using `PLUS` and `TIMES` for tokens, we'll call these ast constructors `ADD` and `MULT`:

```
type ast = ID of string | ADD of ast * ast | MULT of ast * ast | NONE
```

We have also added NONE for a technical reason. We never really want to return an AST that includes NONE — and we won't — but we use it to indicate that E' or T' has not recognized anything (i.e. chose the ϵ production).

Now change our recognizer to a parser in the way suggested earlier in lecture 9:

```
let rec parseE toklis = let (r,t) = parseT toklis
                        in let (r',t') = parseE' r
                           in (r', if t'=NONE then t else ADD(t,t'))

and parseE' toklis = if hd toklis = PLUS
                     then parseE (tl toklis)
                     else (toklis, NONE)

and parseT toklis = let (r,t) = parseIDENT toklis
                    in let (r',t') = parseT' r
                       in (r', if t'=NONE then t else MULT(t,t'))

and parseT' toklis = if hd toklis = TIMES
                     then parseT (tl toklis)
                     else (toklis, NONE)

and parseIDENT toklis = match hd toklis with IDENT x -> (tl toklis, ID x)
                                           | _ -> raise SyntaxError ;;
```

This works great:

```
# parseE [IDENT "x"; PLUS; IDENT "y"; PLUS; IDENT "z"; PLUS; IDENT "w"; EOF];;
- : token list * ast = ([EOF], ADD (ID "x", ADD (ID "y", ADD (ID "z", ID "w"))))
```

except that the result is, as expected, right-associating. (Again, think of how `eval` would evaluate this: it would first add z to w, then add y to that, then add x to that — i.e. it would go from right to left)

We proposed to fix that in this way: After `parseE` makes the recursive calls to `parseT` and `parseE'`, instead of simply putting the two trees together with an `ADD` constructor on top, it should combine them in a way that makes the resulting tree "lean left." This function does that:

```
let rec addplus t1 t2 = match t2 with
          ADD(ADD(t21, t22) as a, t2') -> ADD(addplus t1 a, t2')
        | ADD(t21, t22) -> ADD(ADD(t1, t21), t22)
        | _ -> ADD(t1, t2)
```

We will leave it up to the reader to figure this out. But here are a couple of examples:

```
#addplus (ID "x") (ADD (ID "y", ID "z"));;
- : ast = ADD (ADD (ID "x", ID "y"), ID "z")
#addplus (ID "v") (ADD(ADD(ADD(ID "x", ID "y"), ID "z"), ID "w"));;
- : ast = ADD (ADD (ADD (ADD (ID "v", ID "x"), ID "y"), ID "z"), ID "w")
```

You can see that if the second argument is in left-associating form, the first argument is placed in the tree "at the bottom," so that the entire thing is in left-associating form.

Now we just make one change to `parseE`:

```
let rec parseE toklis = let (r,t) = parseT toklis
                        in let (r',t') = parseE' r
                           in (r', if t'=NONE then t else addplus t t')
```

Consider `t'`, the tree returned from the call to `parseE'`: Looking at `parseE'`, we see that if `t'` isn't `NONE`, then it is an AST returned from a call to `parseE`. Let's assume, by the recursion fairy, that AST's returned from `parseE` are in left-associating form. Then, looking at the new version of `parseE`, we see that the tree `t` returned from `parseT` will be added to `t'` in such a way as to make the whole thing left-associating. So, by the usual "circular" — or rather, inductive — reasoning associated with recursion, we conclude that `parseE` always returns trees in left-associating form. And we see that this is so:

```
# parseE [IDENT "x"; PLUS; IDENT "y"; PLUS; IDENT "z"; EOF];;
- : token list * ast = ([EOF], ADD (ADD (ID "x", ID "y"), ID "z"))
```

Note that the precedence of multiplication is still preserved:

```
# parseE [IDENT "x"; PLUS; IDENT "y"; PLUS; IDENT "z"; TIMES; IDENT "w"; EOF];;
- : token list * ast =j([EOF], ADD (ADD (ID "x", ID "y"), MULT (ID "z", ID "w")))
```

The only remaining problem is that multiplication is still right-associating:

```
# parseE [IDENT "x"; TIMES; IDENT "y"; TIMES; IDENT "z"; EOF];;
- : token list * ast = ([EOF], MULT (ID "x", MULT (ID "y", ID "z")))k
```

Obviously, this can be handled in the same way, with a new function `addmult`. We leave this as an exercise for the reader.