Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 10

ML examples II: Recursive Descent Parsing

Topics covered:

- The parsing problem
- Recursive descent
- Parsers in ML
- Higher order parser combinators
- Efficiency and limitations.

Grammar for terms

We would like to have a parser for our terms, so that we don't have to write them in terms of type constructors.

$$
\begin{array}{rcl}
\mathit{term} & \longrightarrow & \mathit{name}\texttt{(}\mathit{termlist}\texttt{)} \\
 & | & \mathit{name} \\
 & | & \texttt{(}\mathit{term}\texttt{)} \\
 & | & \mathit{numeral} \\
 & | & \texttt{-}\mathit{term} \\
 & | & \mathit{term}\ \texttt{+}\ \mathit{term} \\
 & | & \mathit{term}\ \texttt{*}\ \mathit{term} \\
\mathit{termlist} & \longrightarrow & \mathit{term}\texttt{,}\mathit{termlist} \\
 & | & \mathit{term}
\end{array}
$$

Here we have a grammar for terms, defined by a set of production rules.

Ambiguity

The task of *parsing*, in general, is to reverse this, i.e. find a sequence of productions that could generate a given string.

Unfortunately the above grammar is *ambiguous*, since certain strings can be produced in several ways, e.g.

$$
\begin{array}{lcl}
term & \longrightarrow & term \; \texttt{+} \; term \\
 & \longrightarrow & term \; \texttt{+} \; term \; \texttt{*} \; term
\end{array}
$$

and

$$
\begin{array}{lcl}
term & \longrightarrow & term \; \texttt{*} \; term \\
 & \longrightarrow & term \; \texttt{+} \; term \; \texttt{*} \; term
\end{array}
$$

These correspond to different 'parse trees'. Effectively, we are free to interpret $x \; \texttt{+} \; y \; \texttt{*} \; z$ either as $x \; \texttt{+} \; (y \; \texttt{*} \; z)$ or $(x \; \texttt{+} \; y) \; \texttt{*} \; z$.

Encoding precedences

We can encode operator precedences by introducing extra categories, e.g.

$$
\begin{array}{rcl}
atom & \longrightarrow & name\texttt{(}termlist\texttt{)} \\
 & | & name \\
 & | & numeral \\
 & | & \texttt{(}term\texttt{)} \\
 & | & \texttt{-}atom \\
mulexp & \longrightarrow & atom \texttt{ * } mulexp \\
 & | & atom \\
term & \longrightarrow & mulexp \texttt{ + } term \\
 & | & mulexp \\
termlist & \longrightarrow & term\texttt{,}termlist \\
 & | & term
\end{array}
$$

Now it's unambiguous. Multiplication has higher precedence and both infixes associate to the right.

Recursive descent

A *recursive descent* parser is a series of mutually recursive functions, one for each syntactic category (*term*, *mulexp* etc.).

The mutually recursive structure mirrors that in the grammar.

This makes them quite easy and natural to write — especially in ML, where recursion is the principal control mechanism.

For example, the procedure for parsing terms, say `term` will, on encountering a `-` symbol, make a recursive call to itself to parse the subterm, and on encountering a name followed by an opening parenthesis, will make a recursive call to `termlist`. This in itself will make at least one recursive call to `term`, and so on.

Parsers in ML

We assume that a parser accepts a list of input characters or tokens of arbitrary type.

It returns the result of parsing, which has some other arbitrary type, and also the list of input objects not yet processed. Therefore the type of a parser is:

$$(\alpha)list \to \beta \times (\alpha)list$$

For example, when given the input characters `(x + y) * z` the function `atom` will process the characters `(x + y)` and leave the remaining characters `* z`. It might return a parse tree for the processed expression using our earlier recursive type, and hence we would have:

```
atom "(x + y) * z" =
  Fn("+",[Var "x", Var "y"]),"* z"
```

Parser combinators

In ML, we can define a series of *combinators* for plugging parsers together and creating new parsers from existing ones.

By giving some of them infix status, we can make the ML parser program look quite similar in structure to the original grammar.

First we declare an exception to be used where parsing fails:

```
exception Noparse;
```

`p1 ++ p2` applies `p1` first and then applies `p2` to the remaining tokens; `many` keeps applying the same parser as long as possible.

`p >> f` works like `p` but then applies `f` to the result of the parse.

`p1 || p2` tries `p1` first, and if that fails, tries `p2`. These are automatically infix, in decreasing order of precedence.

Definitions of the combinators

```
fun ++ (parser1,parser2) input =
  let val (result1,rest1) = parser1 input
      val (result2,rest2) = parser2 rest1
   in ((result1,result2),rest2)
  end;

fun many parser input =
  let val (result,next) = parser input
      val (results,rest) = many parser next
   in ((result::results),rest)
  end handle Noparse => ([],input);

fun >> (parser,treatment) input =
  let val (result,rest) = parser input
   in (treatment(result),rest) end;

fun || (parser1,parser2) input =
  parser1 input
  handle Noparse => parser2 input;
```

Auxiliary functions

We make some of these infix:

```
infixr 8 ++; infixr 7 >>; infixr 6 ||;
```

We will use the following general functions below:

```
fun itlist f [] b = b
      | itlist f (h::t) b =
          f h (itlist f t b);

fun K x y = x;

fun fst(x,y) = x;

fun snd(x,y) = y;

val explode = map str o explode;
```

Atomic parsers

We need a few primitive parsers to get us started.

```
fun some p [] = raise Noparse
  | some p (h::t) =
      if p h then (h,t)
      else raise Noparse;

fun a tok = some (fn item => item = tok);

fun finished input =
   if input = [] then (0,input)
   else raise Noparse;
```

The first two accept something satisfying `p`, and something equal to `tok`, respectively. The last one makes sure there is no unprocessed input.

Lexical analysis

First we want to do lexical analysis, i.e. split the input characters into tokens. This can also be done using our combinators, together with a few character discrimination functions. First we declare the type of tokens:

```
datatype token = Name of string
               | Num of string
               | Other of string;
```

We want the lexer to accept a string and produce a list of tokens, ignoring spaces, e.g.

```
- lex "sin(x + y) * cos(2 * x + y)";
> val it =
    [Name "sin", Other "(", Name "x", Other "+",
     Name "y", Other ")", Other "*", Name "cos",
     Other "(", Num "2", Other "*", Name "x",
     Other "+", Name "y", Other ")"] : token list;
```

Definition of the lexer

```
val lex = let
 fun several p = many (some p)
 fun lowercase_letter s = "a" <= s andalso s <= "z"
 fun uppercase_letter s = "A" <= s andalso s <= "Z"
 fun letter s =
   lowercase_letter s orelse uppercase_letter s
 fun alpha s = letter s orelse s = "_" orelse s = "'"
 fun digit s = "0" <= s andalso s <= "9"
 fun alphanum s = alpha s orelse digit s
 fun space s = s = " " orelse s = "\n" orelse s = "\t"
 fun collect(h,t) =
   h^(itlist (fn s1 => fn s2 => s1^s2) t "")
 val rawname =
    some alpha ++ several alphanum
    >> (Name o collect)
 val rawnumeral =
    some digit ++ several digit
    >> (Num o collect)
 val rawother = some (K true) >> Other
 val token =
   (rawname || rawnumeral || rawother) ++
   several space >> fst
 val tokens = (several space ++ many token) >> snd
 val alltokens = (tokens ++ finished) >> fst
in fst o alltokens o explode end;
```

Parsing terms

In order to parse terms, we start with some basic parsers for single tokens of a particular kind:

```
fun name (Name s::rest) = (s,rest)
  | name _ = raise Noparse;

fun numeral (Num s::rest) = (s,rest)
  | numeral _ = raise Noparse;

fun other (Other s::rest) = (s,rest)
  | other _ = raise Noparse;
```

Now we can define a parser for terms, in a form very similar to the original grammar. The main difference is that each production rule has associated with it some sort of special action to take as a result of parsing.

The term parser (take 1)

```
fun atom input
  = (name ++
      a (Other "(") ++ termlist ++ a (Other ")")
          >> (fn (f,(_,(a,_))) => Fn(f,a))
   || name
          >> (fn s => Var s)
   || numeral
          >> (fn s => Const s)
   || a (Other "(") ++ term ++ a (Other ")")
          >> (fst o snd)
   || a (Other "-") ++ atom
          >> snd) input
and mulexp input
  = (atom ++ a(Other "*") ++ mulexp
          >> (fn (a,(_,m)) => Fn("*",[a,m]))
   || atom) input
and term input
  = (mulexp ++ a(Other "+") ++ term
          >> (fn (a,(_,m)) => Fn("+",[a,m]))
   || mulexp) input
and termlist input
  = (term ++ a (Other ",") ++ termlist
          >> (fn (h,(_,t)) => h::t)
   || term
          >> (fn h => [h])) input;
```

Examples

Let us package everything up as a single parsing function:

```
val parser =
  fst o (term ++ finished >> fst) o lex;
```

To see it in action, we try with and without the printer (see above) installed:

```
  - parser "sin(x + y) * cos(2 * x + y)";
  > val it =
      Fn("*",
          [Fn("sin", [Fn("+", [Var "x", Var "y"])]),
           Fn("cos", [Fn("+", [Fn("*",
                      [Const "2", Var "x"]), Var "y"])])])
      : term
  - installPP print_term;
  > val it = () : unit
  - parser "sin(x + y) * cos(2 * x + y)";
  > val it = `sin(x + y) * cos(2 * x + y)` : term
```

Automating precedence parsing

We can easily let ML construct the 'fixed-up' grammar from our dynamic list of infixes:

```
fun binop opr parser input =
  let val (result as (atom1,rest1)) = parser input
   in if rest1 <> [] andalso hd rest1 = Other opr then
        let val (atom2,rest2) =
              binop opr parser (tl rest1)
         in (Fn(opr,[atom1, atom2]),rest2) end
      else result end;

fun findmin l = itlist
  (fn (p1 as (_,pr1)) => fn (p2 as (_,pr2)) =>
     if pr1 <= pr2 then p1 else p2) (tl l) (hd l);

fun delete x (h::t) =
  if h = x then t else h::(delete x t);

fun precedence ilist parser input =
  if ilist = [] then parser input else
  let val opp = findmin ilist
      val ilist' = delete opp ilist
   in binop (fst opp) (precedence ilist' parser) input
  end;
```

The term parser (take 2)

Now the main parser is simpler *and* more general.

```
fun atom input
  = (name ++
      a (Other "(") ++ termlist ++ a (Other ")")
           >> (fn (f,(_,(a,_))) => Fn(f,a))
   || name
           >> (fn s => Var s)
   || numeral
           >> (fn s => Const s)
   || a (Other "(") ++ term ++ a (Other ")")
           >> (fst o snd)
   || a (Other "-") ++ atom
           >> snd) input
and term input = precedence (!infixes) atom input
and termlist input
  = (term ++ a (Other ",") ++ termlist
           >> (fn (h,(_,t)) => h::t)
   || term
              >> (fn h => [h])) input;
```

This will dynamically construct the precedence parser using the list of infixes active when it is actually used. Now the basic grammar is simpler.

Backtracking and reprocessing

Some productions for the same syntactic category have a common prefix. Note that our production rules for *term* have this property:

$$\begin{array}{rcl} term & \longrightarrow & name(termlist) \\ & | & name \\ & | & \cdots \end{array}$$

We carefully put the longer production first in our actual implementation, otherwise success in reading a name would cause the abandonment of attempts to read a parenthesized list of arguments.

However, this backtracking can lead to our processing the initial name twice.

This is not very serious here, but it could be in `termlist`.

An improved treatment

We can easily replace:

```
fun ...
and termlist input
     = (term ++ a (Other ",") ++ termlist
            >> (fn (h,(_,t)) => h::t)
     || term
            >> (fn h => [h])) input;
```

with

```
let ...
and termlist input
    = (term ++
       many (a (Other ",") ++ term >> snd)
             >> (fn (h,t) => h::t)) input;
```

This gives another improvement to the parser, which is now more efficient and slightly simpler. The final version is:

The term parser (take 3)

```
fun atom input
  = (name ++
     a (Other "(") ++ termlist ++ a (Other ")")
         >> (fn (f,(_,(a,_))) => Fn(f,a))
  || name
         >> (fn s => Var s)
  || numeral
         >> (fn s => Const s)
  || a (Other "(") ++ term ++ a (Other ")")
         >> (fst o snd)
  || a (Other "-") ++ atom
         >> snd) input
and term input = precedence (!infixes) atom input
and termlist input
  = (term ++ many (a (Other ",") ++ term >> snd)
    >> (fn (h,t) => h::t)) input;
```

General remarks

With care, this parsing method can be used effectively. It is a good illustration of the power of higher order functions.

The code of such a parser is highly structured and similar to the grammar, therefore easy to modify.

However it is not as efficient as LR parsers; ML-Yacc is capable of generating good LR parsers automatically.

Recursive descent also has trouble with *left recursion*. For example, if we had wanted to make the addition operator left-associative in our earlier grammar, we could have used:

$$\begin{array}{rcl} term & \longrightarrow & term \; \texttt{+} \; mulexp \\ & | & mulexp \end{array}$$

The naive transcription into ML would loop indefinitely. However we can often replace such constructs with explicit repetitions.