
CS 421 – Spring 2007

Lecture Notes Set 23: Recursive Decent Parsing

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 18 - rec-dec-parsing (slides 2-31)

Made available: March 26, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Writing parsing programs

2.1 Programs for parsing (slide 2)

Now, we are going to start looking at a more systematic way of being able to tell if a string is accepted by a grammar. We don't want the program just to tell us if the syntax works, we want it to actually do something with the strings we send it. We are going to do this in a couple of phases. The first phase is to get the parse tree, in the form of an abstract syntax tree. Then with the abstract syntax, we are going to add meaning, which tells us how to evaluate something of that form.

So what we want is a program that can produce elements of a datatype that represent the parse trees for expressions. This process is broken up into a couple pieces. There is the lexer that generates the tokens from a string or character stream. It tells if the string/stream satisfies being part of a syntactic category. Essentially, we are checking to see if the sentences are 'correct'. Do they have a subject and a verb? From that, we are going to build our parse tree. We are going to check what role each word plays in the sentence.

2.2 Idea of Recursive Decent Parsing (slides 3-5)

There is an algorithm that takes a string and a grammar and can decide whether or not that string is parsed by that grammar but there is a great deal of search and backtracking. Because of this, we will cut down our space of grammars to, instead of containing every BNF grammar, only those that are more well-formed than just being an arbitrary BNF grammar.

The first class of grammars that we are going to look at are ones that can be done by what is called LL-parsing. We are going to read the string from left to right, and as we do so, we are going to build a parse tree that corresponds to a left-most derivation. This is going to be done in a top down fashion. That is, we are going to find the structure of the tree starting with the top most leaf and then working our way down. You may be thinking, well, how else would you do this? And the answer to that is LR parsing, which we will learn next time, parses in a bottom-up manner.

So we start out by setting things up to get our parse tree. What we do, is that we have each syntactic category become a type and each rule becomes a data constructor. So, your non-terminals become the data types and then your rules become the constructors of those data types. To write a program over this and produce this. Since each type has different properties, they each need different subprogram to handle them. So if we have a program that returns a <factor> parse tree, it can't also return a <id> parse tree, since those are of different types.

¹© 2007, Share and Enjoy

Then each rule will become a clause in our recursive decent parser, that will say if you see something that matches this pattern, you will give this result. But in that rule, you will have various non-terminals that you will have to deal with and those require building parse trees and that's where the recursive decent comes in. When you hit a hit a non-terminal in your production on the right hand side, that corresponds to a recursive call to one of the mutually recursive family of parsers you have for parsing a grammar.

Recursive decent parsing doesn't work for all BNF grammars; there are some problems. It has to be able to avoid nontermination, that is, it has to be able to decide which rule it's going to call, simply based on looking at the left most character. You don't get to look at your whole string, all you have is that one character and that's it. It's not that you can't write algorithms that look at entire strings, it's that looking at the whole string every time can get very expensive. Imagine trying to parse a code that thousands of lines long and having to look at all of it for every word in your code. Our goal is to parse in one scan of the string.

Because of this restriction, left recursive grammars are very hard to parse. For example, if your grammar had the rule `<Sum> ::= <Sum> + <Atom>`, if you went to parse this, the parse would see the first `<Sum>` so it would immediately call the rule for `<Sum>` and then it see another `<Sum>` and make the same call, and it would do this repeatedly, forever, or at least until you fill up your stack with recursive calls. So you can't have a left-associative grammar.

3 An example parser (slides 6-11)

So let's look at this example. It is at right associative disambiguous grammar. Here, `*` and `/` have higher precedence than `+` and `-`.

```
<expr> ::= <term> | <term> + <expr> | <term> - <expr>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= <id> | ( <expr> )
```

Let's assume we already have a lexer for this and the lexer produced a bunch of tokens for us. We take the characters `+` `*` `/` `(` `)` and `<id>` and turn them into `Id_token` of string | `Left_parenthesis` | `Right_parenthesis` | `Times_token` | `Divide_token` | `Plus_token` | `Minus_token`. These are different constructors for the characters that have type token.

Now we have to describe `<expr>` in terms of OCaml code we have:

```
type expr =
  Term_as_Expr of term
  Plus_Expr of (term * expr)
  Minus_Expr of (term * expr)
```

We say that a term is anything you can get a parse tree for, and `Plus_Expr` is pair of a term and an expression and the same for `Minus_Expr`. The fact that there was an actual `+` or `-` there will be lost. The existence of the either of them is entirely encoded into the constructor.

And `<term>` has the same structure, more or less:

```
and term =
  Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
```

Finally, we have `<factor>`

```
and Factor =
  Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Our next step is to write three mutually recursive functions. Mutually recursive because each has to know about the other two terminals. The functions will parse what they can and then recursive on what ever is left of the tokens. The input is some form of sequencing of tokens. In practice, this is essentially done with a stream. The whole idea of this is that we are not required to use whole token lists. At any point, we only need to see what we can call the head of the token list. We always need to make sure we are returning a token list though. That is because our constructors, mainly the ones that are pairs, will parse something, and then the go to the right side of the pair and say, 'ok, I need to

parse something with this rule'. If you don't return the token stream after the first call to a subprogram in a pair then the second subprogram won't have any tokens to parse.

3.1 Parsing expressions (slides 12-20)

So, let's look at how we parse expressions.

```
<expr> ::= <term> | <term> + <expr> | <term> - <expr>
```

Now, we have already mentioned that you parse by looking at one character at a time, so one thinks, how can you tell the difference between the three different rules for <expr> when they all start with a <term>. Well, the answer is easy when looking at the rules written in a different fashion.

```
<expr> ::= <term> [( + | - ) <expr> ] | <term>
```

One can take this expression, and with the use of match statements, these can be easily parsed. Even though you can't decide at the very beginning, which rule to take, the key here is that you can eventually make a decision. So the first thing we do, is we call term on all our token head and see what it does, that is, we see what we can match it with.

```
let rec expr tokens =  
  (match term tokens  
   with ( term_parse , tokens_after_term) →  
        (match tokens_after_term with ...
```

So here, we say that we just parsed a term, and we get a pair back, the first component is what the term was, the second component was the rest of the tokens list. Now, there are two cases where the tokens_after_term have an expression that we need, and that happens when we see a + or a -. So, that's exactly what we do next, we check for a Plus_token or a Minus_token. If we see the + or the -, then the next token has to be the <expr>. So we take the tokens after the + or the - and parse it as having an expression and then the rest of the tokens left. So for + we have:

```
( Plus_token :: tokens_after_plus) →  
(match expr tokens_after_plus  
 with ( expr_parse, tokens_after_expr) →  
  ( Plus_Expr (term_parse, expr_parse), tokens_after_expr))
```

And but if we saw a - we would use:

```
| ( Minus_token :: tokens_after_minus) →  
(match expr tokens_after_minus  
 with (expr_parse , tokens_after_expr) →  
  ( Minus_Expr ( term_parse, expr_parse), tokens_after_expr))
```

Finally, the last case is that we have a <term> by itself, which means that we didn't see a + or a - so that means we should just return the <term> and what comes after it as the token list.

```
| _ → (Term_as_Expr term_parse, tokens_after_term))
```

Note that the code for <term> is the same except for switching a couple types and changing the constructors, that is, having * and / instead of + and -.

3.2 Parsing factor (slides 21-24)

Parsing <factor> is a little different.

```
<factor> ::= <id> | ( <expr> )
```

We will know how we are going to parse this from the very first token. So if we see an <id> we call the constructor for <id> and then return the rest of the tokens.

```
and factor (Id_token id_name :: tokens) =  
  ( Id_as_Factor id_name, tokens)
```

Otherwise we are going to see a (and what do we do with that...we throw it away. It's done its job, we know we need to deal with parentheses and that's it, we don't need it anymore. The parenthesis then tells us we need to see an <expr> next, followed by a). So that is what we do, we get the (and then we pull out the expression, and match the token list that call returned with a).

```
factor ( Left_parenthesis :: tokens) =  
  (match expr tokens
```

```

with ( expr_parse , tokens_after_expr) →
(match tokens_after_expr
with Right_parenthesis :: tokens_after_rparen →
( Parenthesized_Expr_as_Factor expr_parse , tokens_after_rparen)

```

But wait, what happens if we didn't match our parentheses. That is, what if we see a left parenthesis without a matching right one, or vice versa. In those cases, we have to raise failures.

```

_ → raise (Failure "No matching rparen")
_ → raise (Failure "No id or lparen" );

```

When parsing a <term> or an <expr>, we didn't have to worry about error cases, because the only reason we got in those cases in the first place is that we saw we had a <term> or an <expr>. However, here, if we don't see an <id> we are expecting a set of parentheses, but there is the possibility that it may not be there.

3.3 Parsing examples (slides 25-29)

Let's look at the following example: (a + b) * c - d

We would make this call:

```

expr [Left_parenthesis, Id_token "a", Plus_token, Id_token "b",
Right_parenthesis, Times_token, Id_token "c", Minus_token, Id_token "d"];

```

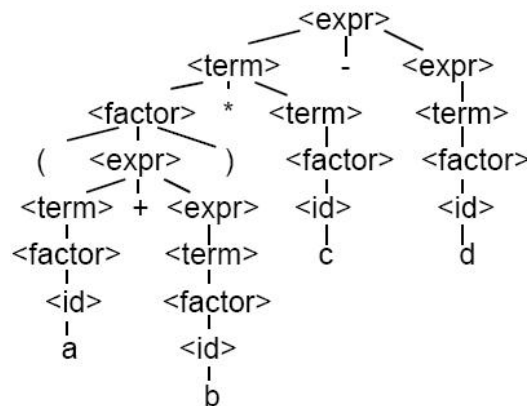
We get a pair returned to us. The first component is the parse tree. It says that we have a Minus.Expr of a Mult.Term of a Parenthesized.Expr.as.Factor of a Plus.Expr of an 'a' with a 'b' with a 'c' (for the other side of the Mult.Term) and then a 'd' (for the other side of the Minus.Expr). The second component is an empty list, and during the running of the program, this held the tokens list. This means our parsing worked. If we didn't get the empty list, our parsing didn't work.

```

- : expr * token list =
(Minus.Expr
(Mult.Term
(Parenthesized.Expr.as.Factor
(Plus.Expr
(Factor.as.Term (Id.as.Factor "a"),
Term.as.Expr (Factor.as.Term (Id.as.Factor "b")))),
Factor.as.Term (Id.as.Factor "c")),
Term.as.Expr (Factor.as.Term (Id.as.Factor "d"))),
[])

```

In picture terms, this expression looks like:



The next example in the lecture slides (slides 28-29) show an example of the same expression above, except without using parentheses. So you can see that it parses differently, because the parentheses aren't there to affect the precedence between the + and the -.

3.4 Errors (slides 30-31)

If we tried to parse $(a + b * c - d)$ we would only get an error returned. That is the very first call we make is for a `<factor>`. That call never gets finished because we never see a right parentheses.

If we tried to parse $a + b) * c - d$ we would have a different result. The beginning of the expression is fine, and we are able to parse it. However, we then see a right parenthesis and the parser says 'dude...I can't parse this'. And when everything gets returned from the error we get only what was able to be parsed and then the list that should be empty has the rest of the tokens still in it. The reason that we do not get the error back is because the parser thinks that whoever called it will do something else with that right parenthesis. So, it simply returns things like normal.

```
expr [Id_token "a; Plus_token; Id_token "b; Right_parenthesis;
      Times_token; Id_token "c; Minus_token; Id_token "d"];
- : expr * token list =
  (Plus_Expr
   (Factor_as_Term (Id_as_Factor "a"),
    Term_as_Expr (Factor_as_Term (Id_as_Factor "b"))),
   [Right_parenthesis; Times_token; Id_token "c"; Minus_token; Id_token "d"])
```