

---

# CS 421 – Spring 2007

## Lecture Notes Set 25: LR Parsing Continued

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 19-lrgrammars (slides 35-end)

Made available: March 30, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 LR Parsing Algorithm

You won't have to ever have to do the algorithm yourself by hand since you don't know how to create the action and goto tables without the help of a computer. But what we will do is go over the algorithm, so you understand the basic idea, so that if you come to point that you need to debug some code that was running LR Parsing, you can at least know what you're looking at.

### 2.1 Idea

So, when you run the parser, two tables are created, the Action Table and Goto Table. So using these tables, you can tell, after seeing the next token, should you shift the token and state onto the stack or should you look back at your stack and state and try to reduce. The Action table tells you if you can do those actions or not. The Goto table tells you, if you do some action, what state should you move to next.

### 2.2 Procedure (slides 35-38)

So the procedure technically begins with step 3, where you look at the next token in the stream. So after you see the next token, you look at your state and do what the action table tells you to do. Rather, we look up what it is we are supposed to do. So, if the action is to shift, we push the token onto the stack and move to the next state that it tells us to. And then we go back to step 3.

If we are told to do a reduce, it tells us which rule we are supposed to use to reduce. We start by removing twice the length of rule symbols from the stack (that is the rule and all the states inbetween each symbol). So after popping the  $2 \times \text{rule}$  items from the stack a state that we passed a long time ago ( $2 \times \text{length of rule stack pushes ago}$ ) is now on the top of the stack. So we use the Goto table with that new state and the rule we just used to find our next state. Then we push the rule onto the stack and the new state we just retrieved. After all that, it's time to go back to step 3. So according the slides, our rule is  $E ::= u$ . So we remove  $2 \times (\text{length}(u))$  symbols. And it's 2 times the length of the rule because for each token we pushed onto the stack we pushed a state on the stack right after it. So after popping all those items off, we are left with a state on top of the stack that we pushed on a while back. We take that state  $m$  and the rule  $E ::= u$  and use the Goto table to find our new state  $p$ . Then we push  $E$  onto the stack and then we push  $p$  onto the stack.

---

<sup>1</sup>© 2007, Share and Enjoy

Now, there are two more possibilities. If we get to the new state and the action is accept, then we finished the string (or we got to some symbol that told us to stop parsing, like an end of line, a double semi-colon, etc) and we stop parsing, and return that the parsing was a success. However, if the action is error, then we stop parsing because the string cannot be parsed and we return that we failed. :(

### 2.3 More than accept (slide 39)

But we want more than this. We don't want to simply accept strings, if we do accept them we want to do something interesting with them, like generate a parse tree. For example, with the use of attributes, yacc can become a simple calculator, compiler, or interpreter.

So, to do this, every time you see a reduction, then when you replace the string with the non-terminal, you have to extract from the stack, the attribute that is associated with that non-terminal. And as you move to the bottom of the stack, you are going to have to use the function for composing those sub-attributes to get the attribute for non-terminal one node up in the parse tree. So instead of just pushing onto the stack, the non-terminal, you also push the attribute with it.

So when you get to the accept state, you will then extract the last attribute pushed onto the stack, and that is considered the accept attribute.

### 2.4 Conflicts (slides 40-44)

Remember, before the second midterm, we discussed ambiguous grammars and how to disambiguate them. Well, you are going to need to remember that for this topic.

Parsers run into problems when they can't figure, with the use of the current state and the next token, whether the action should be to shift or to reduce. This is called a shift-reduce conflict. It is caused by ambiguity in the grammar. There is either a lack of associativity or precedence in the grammar.

Let's look at an example, here is the grammar we are working with:

$\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle) \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

And we want to parse the string  $0 + 1 + 0$ . So we start:

Stack:	String: $0 + 1 + 0$	Action: shift
Stack: 0	String: $+ 1 + 0$	Action: reduce
Stack: $\langle \text{Sum} \rangle$	String: $+ 1 + 0$	Action: shift
Stack: $\langle \text{Sum} \rangle +$	String: $1 ) + 0$	Action: shift
Stack: $\langle \text{Sum} \rangle + 1$	String: $+ 0$	Action: reduce
Stack: $\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$	String: $+ 0$	Action: ??

We we get to that last step...we see that we can potentially reduce the stack or we also see that we can shift over the  $+$ . But we don't know which one is more important, because we don't know which way associative the  $+$  is. If we reduce, it corresponds to  $+$  being left-associative. If we shift, it means  $+$  is right-associative. Now, when parsers hit a shift-reduce conflict, the parser by default chooses shift. But that can potentially mean the parser is making the wrong decision many times.

To avoid this, you must make sure the associativity and precedence are made clear.

Another kind of conflict is the reduce-reduce conflict. The problem here is when you can't decide, between different rules, which one to use to reduce with. This problem is really easy to run into so it is extremely important to deal with this problem. So this conflict stems from the idea that you can look back at your stack and remove  $x$  number of characters because of one rule and try to move on ... or ... you can move  $y$  number of characters because of a different rule and then try from there. Each rule takes you to a different state, but only one will get you where you want to go.

This is again, caused by ambiguity. It usually stems from one rule being the suffix of another. That is, if you had a rule that went to either  $\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$  or just  $\langle \text{Sum} \rangle$ . (Note this is not the best example, because one can just look for a  $+$  after the first  $\langle \text{Sum} \rangle$ ). But here, if you had the stack  $\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$ , one can reduce the second  $\langle \text{Sum} \rangle$  or go all the way back to the first one and reduce the whole stack. So there are two potential reductions that can be done, but you don't know which one to do.

So, a better example is this, you have the grammar:

$S ::= A \mid aB$        $A ::= abc$        $B ::= bc$

And your stack had:

Stack:abc

What do you do? You could reduce by  $B ::= bc$  and get  $aB$  in the stack and then reduce with  $S ::= aB$  and be done with having  $S$  alone. Or you can start with reducing with  $A ::= abc$  and have  $A$  be in the stack and then reduce with the rule  $S ::= A$  and be done with  $S$  in the stack.

## 3 OCaml yacc

### 3.1 Starting things off (slides 45-46)

How do we go about writing such a parser in yacc? Well, the way we are going to do this is we are going to put an attribute grammar in a file that is called `<filename>.mly`. The name of the file is going to be the name of the module that you will generate for creating your parser. The next step you do (or the next step the makefile will do for you) is run `ocaml yacc` on your `mly` file. This will generate several bits of output. You will get an `.ml` file that contains ML code for doing your parsing. Among other things, it will contain that parsing tables (Action and Goto). It will also generate an interface for you that will give you the names of the functions that were produced and their types, one for each entry point. Those functions use the lexer. So the function takes in a buffer and returns a token. And the parser uses these tokens.

One more thing it will produce is a `.output` file (not mentioned in the slide). This file can be ignored if your writing of the yacc code was completely successful. That is there are no conflicts. However, you will see that most of the time, there will be conflicts and you will need this file to figure out what your code is doing. So, this file contains stylized ASCII printouts of your tables and marks where your conflicts are.

### 3.2 OCaml yacc Input (slides 47-51)

At the top of the file, you are allowed to put some code as your header. The code is used for specific attributes that you are trying to generate. Note that the functions you put here are local to the file. You can also do an `Open <file>` here.

Afer the header, you have declarations. They will tell you what the entry points are, types of things are, as well as give you the opportunity to say things about precedence and associativity. So the declarations begin with you saying that you have a collection of tokens. You can put all your tokens on a line, or break them up and even have only ony token on a line. This allows for multiple instances of token and symbol. When you have a token that contains information, like a string within a string, you have to declare that here. So you write `%token <type of information>` symbols that correspond to information. After this, you can declare what your start smybolns are (your entry points). You are allowed to have more than one if you want to use mutually recursive grammars. These start symbols will be outputted to your `<grammar>.ml` file. So, next you have to specify the types of these symbols. If you want more than one output type, you will need more than one line. Now, you can declare which symbols are left associative and then which ones are right associative with the `%left` and `%right` declaration. It will complain if you say a symbol is both. However, you can declare that a symbol has no associativity with the `%nonassoc` declaration. Now, depending on if you put things on different lines or not determines precedence. That is, symbols on the same line are of the same precedence while something on an earlier line has lower precedence. So the later the line, the higher the precedence, the tighter it binds. Note that if you have two rules with the same precedence, but different associativities, you will have a conflict.

The main section you will spend most of your time on are the rules. The rule are the BNF rules together with the associated attributes. The rules are similar to how you wrote `ocamllex`. You have your symbol (instead of regular expressions), and then in curly brackets the action associated with seeing that symbol. So for example, you might have something that looks like:

```
<Sum>:  
<Sum> PLUS <Sum> { $1 + $3 }
```

The dollar sign refers to a symbol in the list of symbols. So `$2` would refer to the `PLUS` symbol in that rule. Think of them like variables representing the symbols in that rule.

Lastly, you have trailer or footer code that could be used for using that function, when you are all done. For example, maybe it will call the parser, so that way, as soon as the code is ready to go, you actually start parsing.

### 3.3 Example (slides 52-57)

Recall the grammar we used when we talked about recursive decent parsing, with, `expr`, `term`, `factor`, and `id` (see slide 52 for grammar). The lexer for this would turn in the input into the tokens for the parser to use (see slide 53 for parser). So after producing all these tokens, we have to introduce them to the parser environment. So, for each token, we have the line:

```
%token symbol
```

In the example, the start symbol is called `main`.

```
%start main
```

And then, the type of `main` is `expr`:

```
%type <expr> main
```

Now, the rules, if we see an `expr`, we check for a `term`, a `term + expr` or a `term - expr`:

```
expr:
```

```
  term Term_as_Expr $1
```

```
  | term Plus_token expr {Plus_Expr($1, $3)}
```

```
  | term Minus_token expr {Minus_Expr($1, $3)}
```

Then you have similar code for `term` and `factor`. And then when you get to `main`, you have:

```
main:
```

```
  | expr EOL {$1}
```

This checks for the end of line token.

### 3.4 Running the parser (slides 58-59)

So, to use the parser, you need to load your grammar, your parser, and your lexer and then write a your test. In the slides, it's called, `s`. Then you can use your test, and the output will be your parse tree written in OCaml fashion.