

---

# CS 421 – Spring 2007

## Lecture Notes Set 24:

### Recursive Decent Parsing Conclusion

### LR Parsing Introduction

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

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 18-rec-dec-parsing (slides 33-39 19-lrgrammars slides (2-36)  
Made available: March 28, 2007  
Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Final notes on Recursive Decent Parsing

### 2.1 Problems (slides 33-34)

We have already mentioned that left recursive grammars cause problems. This is because we will end up in an infinite loop of recursive calls when trying to parse the grammar. So if our rule was  $A ::= Aw$ , we would first see an  $A$ , and make a call to the rule for  $A$ , and see an  $Aw$ , so we would make another call to the  $A$  rule and see  $Aw$  again, so we would...you get the idea. We can't even have indirect left recursion. That is, if  $A ::= Bw$  and  $B ::= Av$ , and we wanted to parse something with an  $A$  in it, we would go and see the rule  $Bw$ , and call the rule for  $B$ , and then see  $Av$ , and then call the rule for  $A$  again, and see a  $Bw$  and....again, you get the idea.

So, along with that issue, we need a grammar that will allow us to figure out what rule we are going to use to parse with after only having had seen one token. One way that we saw for dealing with this was to extend the grammar and not make a choice until it was absolutely obvious which rule we needed to be using. But we need to test our grammar to see if it is even possible to write a parser for the grammar.

### 2.2 Pairwise Disjoint Test (slides 34-36)

The test we are going to use is called the Pairwise disjointedness Test. And note, that this is an approximate test, so we may not get perfect results every time. And this test simply checks to see if it can make a decision on which rule to use based on the first token seen and not extending the grammar like we mentioned before. So this test is a little strict because it would have told us that the grammar we looked at last time would not have a parser, but in fact, we did have one.

So for this test, we are going to calculate what we will call 'First' sets. That is, for each rule, we want to know what first letters could possibly be in a string and use that rule. So we want to calculate what  $FIRST(y)$  equals and to do that  $FIRST(y)$  becomes the set of characters  $a$  such that for all  $y$  the  $a$  is the first part of the rule. But we also want to account for the fact that the character may also be the empty string, so we also allow for  $\epsilon$ . So we found  $FIRST(y)$ , that is the characters for the rule  $y$ . Now we do the same for all rule  $z$  and we want to see that for  $FIRST$  sets for all pairs of rules,  $y$  and  $z$  that all the characters are different.

---

<sup>1</sup>© 2007, Share and Enjoy

## 2.3 Pairwise Disjoint Example (slides 34-36)

So let's look at an example. We have the following grammar:

$$\begin{aligned}\langle S \rangle &::= \langle A \rangle a \langle B \rangle b \\ \langle A \rangle &::= \langle A \rangle b \mid b \\ \langle B \rangle &::= a \langle B \rangle \mid a\end{aligned}$$

Before we go, let's look at what the grammar accepts.  $\langle A \rangle$  will see any sequence of b's that has at least one b in it. Then  $\langle B \rangle$  will accept any sequence of a's that has at least one a in it. So  $\langle S \rangle$  will accept a sequence of one or more b's then an 'a' then a sequence of one or more a's and then a 'b'.

So, for rule  $\langle S \rangle$  there is no conflict because  $\langle S \rangle$  only has one rule associated with it. Now, let's look at  $\langle A \rangle$ . The way it is written, it looks like we have a potential infinite loop. But what we do is this. We say if we have  $\langle A \rangle$  we can go to an  $\langle A \rangle b$  from there. In only one step, we have this call to  $\langle A \rangle$  again, but that means we haven't generated any string yet so we have the  $\{\}$ . Then let's look at the next rule,  $\langle A \rangle$  can be a 'b' so we have  $\{b\}$ . Now, that we know we can get a 'b' the first time around from  $\langle A \rangle$ , we can plug that into the  $\langle A \rangle b$  rule and again, see that 'b' is a potential first character, but we already knew that.

In other words, we want to find all the first characters that  $\langle A \rangle$  accepts. So we want to compute  $\text{FIRST}(\langle A \rangle)$ . So we have to look at every rule for  $\langle A \rangle$ , that is we will look to see what the first character is that we can get from  $\text{FIRST}(\langle A \rangle ::= \langle A \rangle b)$  equals and what  $\text{FIRST}(\langle A \rangle ::= b)$  equals. Well, it turns out, that we first are able to find that  $\text{FIRST}(b) = \{b\}$ . We can plug that result for  $\langle A \rangle$  in the first rule and get that  $\text{FIRST}(\langle A \rangle ::= \langle A \rangle b) = \{b\}$ . Then if you take the intersection of the two sets, we want them to be disjoint (we want to get the empty set back) but it turns out that we get  $\{b\}$  back. And so, the rules for  $\langle A \rangle$  are not pairwise disjoint.

Important note: If the grammar had the rule  $\langle A \rangle ::= \langle A \rangle b \mid \langle A \rangle$ . By simply glancing at these rules, one can see enormous possibilities for an infinite loop. However, if one just does the pairwise disjoint test, both rules would return the empty set as what they can parse. So if you take the intersection of these two, empty sets, the empty set would be returned and you would end up thinking that this grammar is okay. So the point here is, that you need to do both tests.

Let's look at another example. If we had some grammar that had the rules  $\langle \text{term} \rangle ::= \langle \text{id} \rangle * \langle \text{term} \rangle \mid \langle \text{id} \rangle$  and  $\langle \text{id} \rangle ::= a \mid b$ , and we used the pairwise disjoint test on it, we would have the problem that since both rules for  $\langle \text{term} \rangle$  start with  $\langle \text{id} \rangle$ , both rules can start with an 'a' or a 'b', so the test fails and we get back that a parser can't be made. However, remember, like we saw last time, that you can pass the  $\langle \text{id} \rangle$  and then look for a character that tells you which rule to use. And from last time, we know that if we see the \*, then we will follow the first rule, and then for any other character, we will follow the second rule. These secondary characters, called *followsets* need to also be calculated as well.

## 2.4 Fixing the problems (slides 37-38)

So what do we do when we find these problems? Well, the answer is that we generate a new grammar.

If we have a grammar that is left recursive, we will convert it to a right recursive grammar. But we can't simply flip the non-terminals around because that may cause the meaning of the grammar to change. What we do to fix this is add new non-terminals.

Say we had the starting grammar of:

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$

We will add a new non-terminal, we'll call it  $\langle e \rangle$  and it will have the following rules:

$$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle \mid \langle \epsilon \rangle$$

And then we will replace the original  $\langle \text{expr} \rangle$  with:

$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$$

So, this is somewhat similar to what we were just talking about, where we said that with *followsets* we can simply look disregard the first term when it comes to checking to see what rule we use, and then we check the *followsets* to see, do we see a + or do we see an  $\epsilon$ . And looking at those characters will determine which rule we use.

## 3 LR Parsing

### 3.1 Introduction (slide 2)

LL parsing (recursive decent) was parsing from left to right on a leftmost derivation. That is where the LL came from. Now we are going to look at another parsing algorithm that goes from left to right, but does a rightmost derivation. This sounds a little strange at first, but the key is that we are going to be working from the bottom and moving up the tree. So, our left most token of the string will be found at the left most branch of the tree, which will also be the lowest branch of the tree. Then as we move to the right, we will be moving up the tree instead of down, like we were before.

So, as we move up the tree, when we see that we can replace a terminal with a non-terminal, we do so. When we get to the top of the tree, we should only have one replace meant left to make and that is to get us to the start symbol.

### 3.2 Example (slides 3-16)

Take the grammar:

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

and the string that we want to parse:

$$(0 + 1) + 0$$

So our first step is to take the first character we see and push it on a stack. And when we find a character or a sequence of characters that we think we can deal with, we pull it off the stack and deal with it. The action of pushing characters on the stack is called shift. If we find something that we can replace with a non-terminal, we call that reduce.

We see a ( first, so we shift that onto the stack and we have:

Stack:(	String: 0 + 1) + 0	Action: shift
---------	--------------------	---------------

So we look at the characters in the stack and we have a ( and can do nothing with it, so we move on and shift the 0.

Stack:(0	String: + 1) + 0	Action: shift
----------	------------------	---------------

Now we see that our grammar parses 0's from the  $\langle \text{Sum} \rangle$  non-terminal, so we can replace it.

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

Now there is nothing else to do, so we move on to shifting over the +

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

We can't do anything here, so we shift.

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

Now we see a 1 and know that we can reduce that to  $\langle \text{Sum} \rangle$ .

Stack:( $\langle \text{Sum} \rangle$ + $\langle \text{Sum} \rangle$	String: ) + 0	Action: reduce
---	---------------	----------------

But before we move on again, we see we have another rule to parse the  $\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$ , so we can do another reduction.

Stack:( $\langle \text{Sum} \rangle$	String: ) + 0	Action: reduce
--------------------------------------	---------------	----------------

Now there is nothing we can do again, so we shift.

Stack:( $\langle \text{Sum} \rangle$ )	String: + 0	Action: shift
--	-------------	---------------

Here we can do another reduction.

Stack: $\langle \text{Sum} \rangle$	String: + 0	Action: reduce
-------------------------------------	-------------	----------------

Nothing more we can do with this, so we shift.

Stack: $\langle \text{Sum} \rangle$ +	String: 0	Action: shift
---------------------------------------	-----------	---------------

Again, nothing we can do here so we shift.

Stack: $\langle \text{Sum} \rangle$ + 0	String:	Action: shift
---	---------	---------------

We can reduce the 0.

Stack: $\langle \text{Sum} \rangle$ + $\langle \text{Sum} \rangle$	String:	Action: reduce
--	---------	----------------

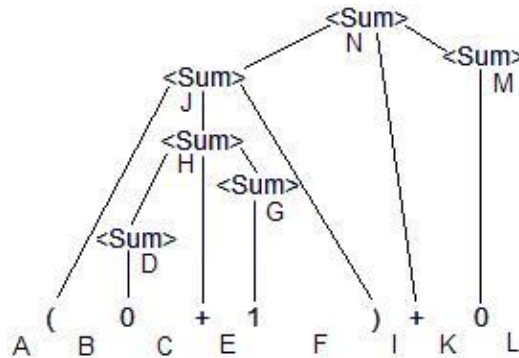
We can do another reduction.

Stack: $\langle \text{Sum} \rangle$	String:	Action: reduce
-------------------------------------	---------	----------------

The string is empty so we are done and look, we have completely reduced the string to the start symbol, so everything worked. If you end up with the case that there is more to the string but you can't reduce anything, then the string can't be parsed by that grammar.

### 3.3 Graphically (slides 17-31)

The letters will guide you from step to step of the algorithm. So, you start at point A with the whole string left to parse. Then you move to point B, and that move stands for a shift. Then there is the move to point C, which is another shift. Now, we move to point D, which stand for a reduction. We were able to move up the tree with this move, etc, until the whole string was parsed.



### 3.4 Tables (slides 32-33)

So to do this parsing, we need to build a couple tables. One is called the Action table and the other is called the Goto table. The building of these tables from your grammar is the main thing that yacc does for you. The rows of these tables are labeled by states. For the Actions table, the columns are labeled by terminals and end-tokens. In the Goto table, the columns are only the non-terminals.

So how do we fill out these tables? Well, for the actions table if we have a state and the next input, we either can say, shift and go to the next state that comes after the shift, reduce by some rule, or accept the string or there was an error. In the Goto table, if you have a state and non-terminal you are simply have what state you are supposed to go from here. The Goto table finishes off a step for the Action table. If you get a chance to do a reduction, then you have to know what state to go to next, so you take your current state and the non-terminal that you just got and you will be able to find out what state you're going to next.

### 3.5 The Algorithm (slides 34-36)

Once you have your tables, the first step is to check to make sure your string ends with an acceptable end of string character. So you know when you have completed parsing the whole string. So after this you go to the initial state with an empty stack. Now, you push your current state, onto the stack. Then you can look at the token in the string and if the top character is part of state( $n$ ) then you look up the associated action in the Action table at position ( $n$ ,  $toks$ ). If the action says shift, then you remove the token from the string and push it on the stack, and then push the state that came with the action onto the stack and then go back to the step that looks at the next token in the string.

We will continue the algorithm during the next class.