LL(k) Grammars

See pages 110-115 of the text.

We need a bunch of terminology.

For any terminal string α we write

$First_k(\alpha)$ is the prefix of α of length k (or all of α if its length is less than k)

For any string γ of terminal and non-terminal symbols and any non-terminal symbol A, we say $A =^{*}> \gamma$ if we can derive γ from A. In English we say A *derives* γ.

For any non-terminal symbol

$First_k(A)$ is the set of $First_k(\alpha)$ of all terminal strings A derives.

Note that for recursive descent to work, if A ::= B1 | B2 is a grammar rule we need $First_k(B1)$ disjoint from $First_k(B2)$.

For any two sets S1 and S2 of strings of terminal symbols $S1 \oplus_k S2$ is the set of prefixes of length k of all of the strings you an get by concatenating a string from S1 with a string from S2.

For any non-terminal symbol A, $Follow_k(A)$ is the set of prefixes of length k of all the terminal strings that could come after strings generated from symbol A. To be precise

$$Follow_k(A) = \{FIrst_k(x) \mid S =*> wAx \text{ any strings w and x}\}$$

We say a grammar is LL(k) if for every non-terminal symbol a and every pair of rules A ::= α | β we have

$$[First_k(\alpha) \oplus_k Follow_k(A)] \cap [First_k(\beta) \oplus_k Follow_k(A)] = \varnothing$$

The idea is that by looking ahead k token we can decide if we should use the rule A ::= a or the rule A::= b.

We will see algorithms for generating the $First_k$ and $Follow_k$ sets.

For now, suppose we know how to generate these sets. If the grammar is LL(k) we can build a table-driven parser for it.

Assume first that k=1. We build a table whose columns are indexed by tokens and whose rows are indexed by the non-terminal symbols of the grammar. The entries of the table are the grammar rules:

1. If A ::=α is a grammar rule and a is in First(α), then
   Table[A, a] is the rule A ::= α.
2. If A ::= α is a rule, and First(α) contains the empty string, and if b is in Follow(A), then Table[A, b] is the rule A ::= α.
3. If A ::= α is a rule, and First(α) contains the empty string, and if EOF is in Follow(A), then Table[A, EOF] is the rule A ::= α.

To use the table we maintain a stack that contains the current sentential form (string of symbols derived from the Start symbol):

1. Begin by pushing the Start symbol on an empty stack.
2. If the current token is on top of the stack, pop the stack and get the next token.
3. If there is a non-terminal symbol A on top of the stack and the current token is a, and if there is a rule in Table[A, a], pop the A off the stack and push on the right side of the rule from the table.
4. In (3), if there is no rule in Table[A, a] issue an error.
5. The stack should be empty when you reach the EOF token.

Example. S ::= if (C) S fi | if (C) S else S fi | a
C ::= b

We would need too many tokens to disambiguate the two if-rules, so we "factor" the grammar into

S ::= if (C) S S' | a
S' ::= fi | else S fi
C ::= b

This is actually 5 rules:

(P1)	S ::= if (C) S S'	First(if (C) S S') = { if }
(P2)	S ::= a	First(a) = { a }
(P3)	S ' ::= fi	First(fi) = { fi }
(P4)	S' ::= else S fi	First(else S fi) = { else }
(P5)	C ::= b	First(b) = { b }

This gives the following parse table:

	if	**(**	**)**	**fi**	**else**	**a**	**b**	**EOF**
S	P1					P2		
S'				P3	P4			
C							P5	

It is easy to walk through parsing an expression like

```
if (b)
    if (b)
        a
    else
        a
    fi
fi
```

Note that if we had the more familiar grammar

S ::= if (C) S | if (C) S else S | a

C ::= b

We would factor it into

(P1) S ::= if (C) S S'	First(if (C) S S') = { if }
(P2) S ::= a	First(a) = { a }
(P3) S ' ::= else S	First(else S) = { else }
(P4) S' ::= ε	First(ε) $\oplus$ Follow(S')= { else, EOF }
(P5) C ::= b	First(b) = { b }

There is an ambiguity between P3 and P4 on the token *else*.

This time the table is

	if	**(**	**)**	**else**	**a**	**b**	**EOF**
S	P1				P2		
S'				P3/P4			P4
C						P5	

We can effectively "disambiguate" the language by choosing which rule we use for Table[S', else]. The standard choice is to use P3, which puts a nested *else* with the nearest possible *if*.

We can build parse trees with this parser. Each time we pop a non-terminal string off the stack, we replace it by a tree node pointing at the right-side elements that are on the stack.

We have built the parse table for an LL(1) grammar. An LL(k) table is exactly the same, only the columns are indexed by strings of k tokens. The primary rule for building the table is:

If $A ::= \alpha$ is a grammar rule and w is in $[First_k(\alpha) \oplus_k Follow_k(A)]$, then Table[A, w] is the rule $A ::= \alpha$.

Left-recursive rules like E ::= E+T | T are not LL(k) for any k. Here is a way to eliminate them. This is called "left factoring".

Example: Consider the grammar

E ::= E + T | T
T ::= T * F | F
F ::= id

We factor it into

E ::= T E'
E' ::= + T E' | ε
T ::= F T'
T' ::= * F T' | ε
F ::= id

This is completely unambiguous. Unfortunately, it makes right-associative trees, but we can deal with that as we complete the tree.

Algorithm to find the $First_1$ sets (you can generalize to $First_k$)

Step I: First compute First(X) for every individual symbol X

1. If x is a terminal symbol, First(x) = {x}
2. For the non-terminal symbols X, start with First(X) empty for every X. Apply the following rules until nothing changes:
   a) If X ::= ε is a rule, add ε to First(X).
   b) If X ::= $Y_1Y_2..Y_n$ is a rule and ε is in $First(Y_1)$.. $First(Y_j)$, then add all the symbols of $First(Y_{j+1})$ to First(X).
   c) If X ::= $Y_1Y_2..Y_n$ is a rule and ε is in every $First(Y_i)$, then add ε to First(X).

Step 2: Now find the First set for the right hand side of every rule:

Suppose X ::= $X_1X_2..X_n$ is a grammar rule

1. Start with First($X_1X_2..X_n$) = First(X_1)
2. Let i be the smallest index so ε is not in First(X_i). Then all of the symbols in First(X_j) for j <= i are in First($X_1X_2..X_n$) .
3. If ε is in all of the First(X_i) then it is also in First($X_1X_2..X_n$).

Algorithm: To compute Follow(X) for every non-terminal symbol X:

1. Include EOF in Follow(Start).
2. If there is a rule X ::= $\alpha B \beta$, include First(β)-ε in Follow(B).

Now apply the following rules until nothing changes:

3. If X ::= αB is a grammar rule, then include all of Follow(X) in Follow(B).
4. If X ::= $\alpha B \beta$ is a grammar rule and ε is in First(β), then include all of Follow(X) in Follow(B).

Example: Consider our calculator grammar:

E ::= E+T | E-T | T

T ::= T*F | T/F | F

F ::= (E) | id

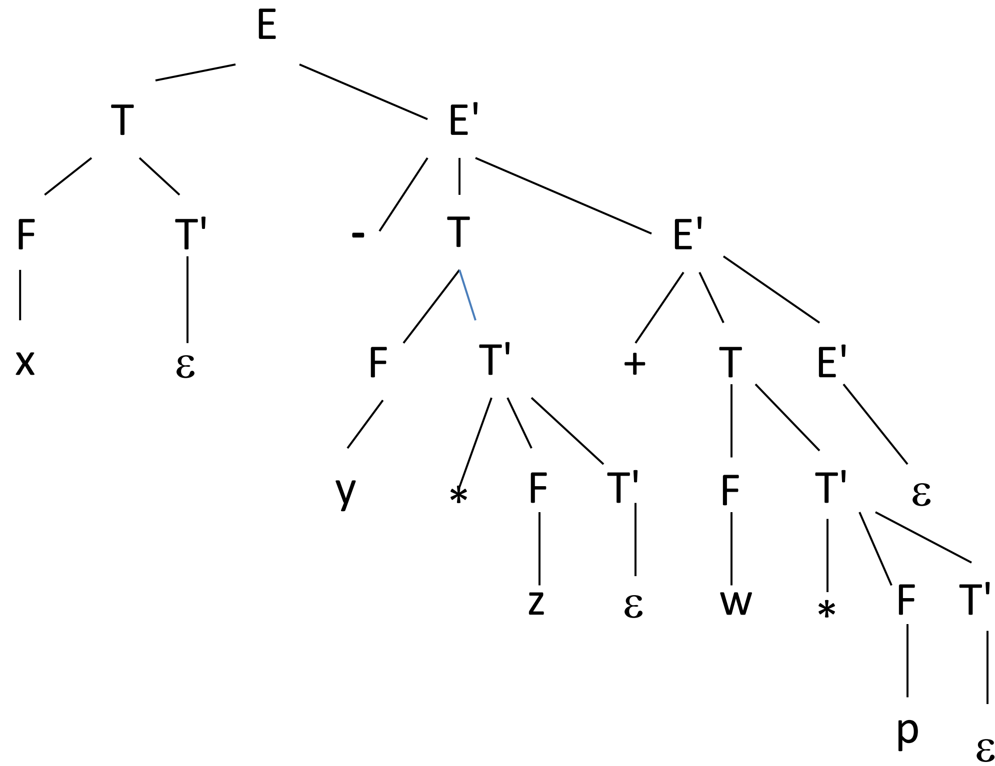
We left factor to get the following rules:

(P1) E ::= TE'	First(TE') = {id, EOF}
(P2) E' ::= +TE'	First(+TE') = {+}
(P3) E' ::= -TE'	First(-TE') = {-}
(P4) E' ::= ε	Follow(E') = {), EOF}
(P5) T ::= FT'	First(FT') = {id, (}
(P6) T' ::= *FT'	First(*FT') = {*}
(P7) T' ::= /FT'	First(/FT') = {/}
(P8) T' ::= ε	Follow(T')={+, -,), EOF}
(P9) F ::= id	First(id) = {id}
(P10) F ::= (E)	First((E)) = {(}

This gives the following LL(1) parse table:

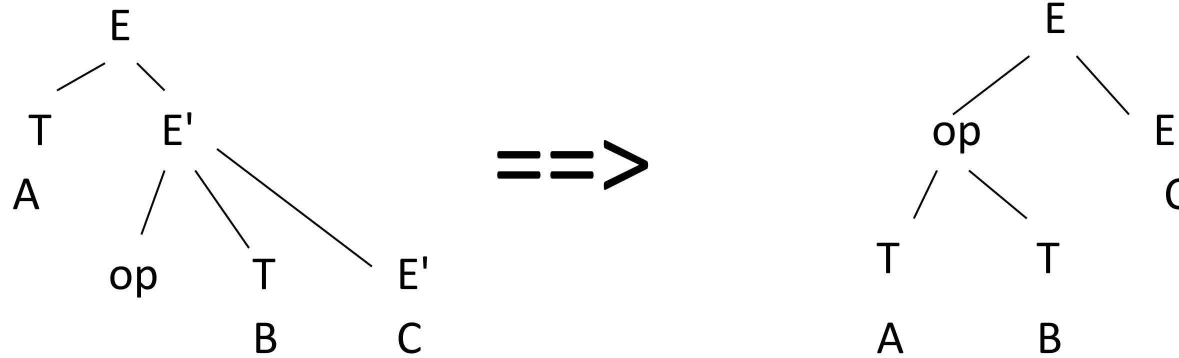
	+	-	*	/	(	)	id	EOF
E					**P1**		**P1**	
E'	**P2**	**P3**				**P4**		**P4**
T					**P5**		**P5**	
T'	**P8**	**P8**	**P6**	**P7**		**P8**		**P8**
F					**P10**		**P9**	

The expression x-y*z +w*p gives the tree

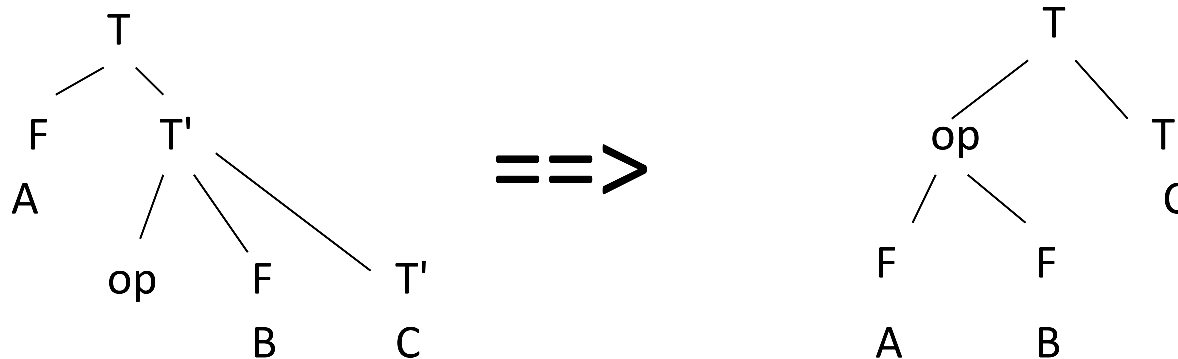


To make this look like our other parse trees, apply the following transformations:

1) Any treenode with only one child can be replaced by that child.
2) The tree can be replace by



3) The tree can be replaced by



With these transformations, our tree for x-y*z+w*p becomes