LL(*k*) Grammars

LL(*k*) grammars are CFGs that we can build a top-down deterministic parser for. This is a subset of CFGs. The first L means we scan the input **l**eft-to-right. The second L means we create a **l**eftmost derivation. ***k*** represents the # of look ahead allowed to determine the correct production to apply.

Examples:

```
1.)  S -> aAS | b
     A -> a | bSA
```

This is a simple LL(1) grammar; we only need to look ahead 1 symbol at a time to determine which S-production or which A-production to use.

```
2.)  S -> ε | abA
     A -> Saa | b
```

Here, we have an LL(2) grammar. We look ahead (at most) 2 symbols in order to determine a string's derivation. S can be followed by ε or by a string beginning with the terminals aa. So with 2 symbols look ahead, one can deterministically decide if the production to be applied is S -> ε or S -> abA. (If there is no more input or the next 2 symbols are aa, then S -> ε must be applied. If the next 2 symbols are ab, S -> abA must be applied.)
Since S can only generate ε or something beginning with a, one symbol look ahead is sufficient to distinguish A -> Saa and A -> b. (If the next symbol is a, use A -> Saa. If the next symbol is b, then use A -> b.)

```
3.)  S -> A | B
     A -> aAb | 0
     B -> aBbb | 1
```

This grammar will either derive strings of the form a^n0b^n or of the form a^n1b^{2n}. This CFG is not LL(*k*) for any size *k*. One needs arbitrarily large look ahead in order to decide initially to use S -> A or S -> B (i.e. one needs to look past all the a's to see if they are followed by a 0 or a 1. Since the number of a's is arbitrarily large, no fixed amount of look ahead suffices.)

Note that it is much harder to determine if a ***language*** is LL(***k***) than to prove if a ***grammar*** is LL(***k***), because a given language may have several different grammars that will generate it.

Definition: Let G = (N, ∑, P, S) be a CFG. G is an **LL(*k*) grammar** if, whenever there are two leftmost derivations:

1. $S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \underset{lm}{\Rightarrow} w\beta\alpha \overset{*}{\underset{lm}{\Rightarrow}} wx$ AND
2. $S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \underset{lm}{\Rightarrow} w\gamma\alpha \overset{*}{\underset{lm}{\Rightarrow}} wy$

such that $FIRST_k(x) = FIRST_k(y)$, then $\beta = \gamma$. The function $FIRST_k()$ is described below.

$FIRST_k(\alpha)$ – This is a function that maps α into the set of terminal strings of length $\leq k$. In other words, we take all terminal strings that can be generated from α, throw away all but the first k symbols of each string and put everything that remains in $FIRST_k(\alpha)$.

If $\alpha = x_1x_2 \ldots x_k$, then $FIRST_k(\alpha) = FIRST_k(x_1) \oplus_k FIRST_k(x_2) \oplus_k \ldots FIRST_k(x_k)$.

Examples of calculating $FIRST_k(\alpha)$ are given in class.

There is a systematic way to determine if a CFG is LL(k). Basically, we try to create an LL(k) parsing table for the given grammar. If we are able to complete the table with k symbols look ahead at each step, the grammar is LL(k). Otherwise, if we find any conflict in constructing the parsing table, then the grammar is not LL(k).

Note that if G_1 is not LL(k), it does not imply that $L(G_1)$ is not LL(k). If there exists a CFG G_2 such that $L(G_1) = L(G_2)$, then the language $L(G_1) = L(G_2)$ is LL(k).

Construct an LL(1) parsing table.

Example: Given the following CFG:

1. S -> aAS
2. S -> b
3. A -> a
4. A -> bSa

We can easily convert this grammar into its equivalent nondeterministic PDA recognizer. With one symbol look ahead, we make the process of the PDA deterministic.

LL(1) Parser:

| lookahead | ≤ 1

	A	**b**	**ε**
S	aAS (1)	b (2)	error
A	A (3)	bSa (4)	error
a	POP	error	error
b	error	POP	error
$	error	error	ACCEPT

top of stack

Note that any left recursive grammar ($A \overset{+}{\rightarrow} AB$) cannot be LL(*k*) for any fixed value of k.

Eliminating Left-Recursion – Let A be any nonterminal and all A-productions be of the form

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots A\alpha_n \mid ß_1 \mid \ldots ß_m$$

where none of the $ß_i$s begin with the nonterminal A. Then we can replace the set of A-productions by:

$$A \rightarrow ß_1 \mid ß_2 \mid \ldots ß_m \mid ß_1A' \mid ß_2A' \mid \ldots ß_mA'$$
$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n \mid \alpha_1A' \mid \alpha_2A' \mid \ldots \alpha_nA'$$

Example:

```
1. E -> E + T  |  T
2. T -> T * F  |  F
3. F -> ( E )  |  a
```

Removing left recursion...

```
1. E -> T | TE'
2. E' -> +T | +TE'
3. T -> F | FT'
4. T' -> *F | *FT'
5. F -> (E) | a
```

However, this is still not LL(*k*). For instance, there are 2 E-productions and both of them generate something beginning with T. Whatever T can generate in E -> T, T can generate the same thing in E -> TE', one needs arbitrarily large look ahead in order to decide initially to use E -> T or E -> TE'. However, we can change it to the following and make it LL(1).

```
1. E -> TE'
2. E' -> ε | +TE'
3. T -> FT'
4. T' -> ε | *FT'
5. F -> (E) | a
```

Now, the grammar is LL(1). To build the LL(1) parser, we need to calculate FIRST and FOLLOW (a function akin to FIRST) for all nonterminals.

$\text{FIRST}_1(E) = \{ \; (\; , \; a \; \}$

$\text{FIRST}_1(E') = \{ \; \varepsilon \; , \; + \; \}$

$\text{FIRST}_1(T) = \{ \; (\; , \; a \; \}$

$\text{FIRST}_1(T') = \{ \; \varepsilon \; , \; * \; \}$

$\text{FIRST}_1(F) = \{ \; (\; , \; a \; \}$

The function **FOLLOW$_k$(B)** is defined as follows:

1. If B is the initial nonterminal, include ε in FOLLOW$_k$(B).
2. IF A -> αBß, FOLLOW$_k$(B) = FIRST$_k$(ß) $\oplus_k$ FOLLOW$_k$(A).

For the example just given, here are the values for FOLLOW_1.

```
FOLLOW₁(E)      = { ε , ) }
FOLLOW₁(E')     = { ε , ) }
FOLLOW₁(T)      = { + , ) , ε }
FOLLOW₁(T')     = { + , ) , ε }
FOLLOW₁(F)      = { * , +, ) , ε }
```

The LL(1) parser for this grammar:

	(	**)**	**+**	*****	**a**	**ε**
E	TE' (1)				TE' (1)	
E'		ε (3)	+TE' (2)			ε (3)
T	FT' (4)				FT' (4)	
T'		ε (6)	ε (6)	*FT' (5)		ε (6)
F	(E) (7)				a (8)	
(	POP					
)		POP				
+			POP			
*****				POP		
a					POP	
$ (empty)						ACCEPT

Top down parsing: To perform top down parsing, we need to keep track of three things - the contents of the stack, the input string, and the output of productions. For instance:

<u>Stack</u>	<u>Input</u>	<u>Output</u>
E$	(a+a)*a	ε

This is the initial state: the input string is unaltered, the stack starts with the initial nonterminal and the empty stack symbol ($), and there is no output. Now, we look at the top item in the stack (E), and look at the first character of the input ("("). In the table, this corresponds to the action TE' and production rule #1, so we replace the top item of the stack with TE'.

<u>STack</u>	<u>Input</u>	<u>Output</u>
E$	(a+a)*a	ε
TE'	(a+a)*a	1

Our next move uses T and the "(" input character again. In the table, these two values intersect at FT' (4).

Stack	Input	Output
E$	(a+a)*a	ε
TE'$	(a+a)*a	1
FT'E'$	(a+a)*a	1, 4

The next move, F and the left parenthesis, corresponds to (E) (7).

(E)T'E'$	(a+a)*a	1, 4, 7

...and the next move matches two left parentheses. This results in a pop both from the stack and the input string. Nothing is added to the output at this step, since the output merely gives us the sequence of productions to follow.

Stack	Input	Output
E$	(a+a)*a	ε
TE'$	(a+a)*a	1
FT'E'$	(a+a)*a	1, 4
(E)T'E'$	(a+a)*a	1, 4, 7
E)T'E'$	a+a)*a	1, 4, 7

Continuing in this manner gives us the final output of 1, 4, 7, 1, 4, 8, 6, 2, 4, 8, 6, 3, 5, 8, 6, 3. At this point, the entire input string has been read and the stack is empty, so the string is, indeed, in the language. Note that this sequence of productions on the output tape is actually the leftmost derivation of the input string.

Any undefined entry in the table is an error, and if we find that we are in a situation where one of these entries must be used, the parser fails: the string is not in the language.

Construct an LL(*k*) parser, for *k* ≥ 2.

Building an LL(*k*) parser, for *k* ≥ 2, involves replacing the nonterminals in the parser with "tables" or "items". These tables describe a nonterminal and its ***local follow set***.

Example: Given the following CFG:

1. S -> aAaa
2. S -> bAba
3. A -> b
4. A -> ε

We show how to obtain the local follow set.

$T_0 = T_S, \{\varepsilon\}$

$\mathrm{FIRST}_2(aAaa) \oplus_2 \{\varepsilon\} = \{ab, aa\}$

$\mathrm{FIRST}_2(bAba) \oplus_2 \{\varepsilon\} = \{bb\}$

$\{ab, aa\} \cap \{bb\} = \emptyset$

Since there is no conflict, we can continue...

lookahead	production	local follow set
ab }	S -> a$\underline{A}$aa	for A: $\{aa\}\oplus_2\{\varepsilon\} = \{aa\}$: T_1
aa }		
bb	S -> b$\underline{A}$ba	for A: $\{ba\}\oplus_2\{\varepsilon\} = \{ba\}$: T_2

$T_1 = T_A, \{aa\}$

$\mathrm{FIRST}_2(b) \oplus_2 \{aa\} = \{ba\}$

$\mathrm{FIRST}_2(\varepsilon) \oplus_2 \{aa\} = \{aa\}$

$\{ba\} \cap \{aa\} = \emptyset$

Since there is no conflict, we can continue...

lookahead	production	local follow set
ba	A -> b	no nonterminals on the RHS
aa	A -> ε	no local follow set

$T_2 = T_A, \{ba\}$

$\mathrm{FIRST}_2(b) \oplus_2 \{ba\} = \{bb\}$

$\mathrm{FIRST}_2(\varepsilon) \oplus_2 \{ba\} = \{ba\}$

$\{bb\} \cap \{ba\} = \emptyset$

Since there is no conflict, we can continue...

lookahead	production	local follow set
bb	A -> b	no nonterminals on the RHS
ba	A -> ε	no local follow set

Using these Ti's, we build the following LL(2) parser:

	aa	**ab**	**ba**	**bb**	**A**	**b**	**ε**
T_0	aT_1aa (1)	aT_1aa(1)		bT_2ba(2)			
T_1	ε (4)		b (3)				
T_2			ε (4)	b (3)			
a	POP	POP			POP		
b			POP	POP		POP	
$							ACCEPT

Top down parsing for the input string "abaa":

Stack	Input	Output
$T_0$$	abaa	ε
$aT_1aa$$	abaa	1
$T_1aa$$	baa	1
baa$	baa	1, 3
aa$	aa	1, 3
a$	a	1, 3
$	ε	1, 3

This accepts the string "abaa".

Example: Determining if the following grammar is LL(2)? LL(3)?

1. S -> AB
2. S -> BC
3. A -> ab
4. A -> CBba
5. A -> baaS
6. B -> bb
7. B -> bbAS
8. B -> baB
9. C -> aaa
10. C -> aabS

We can tell, from looking at productions (6) and (7) that this grammar is not LL(2); that is, looking ahead two characters is not sufficient to determine what B-productions should be applied to the string. Can we determine if it is LL(3)?

$FIRST_3(S)$ = {abb, baa, aaa, aab, bab, bba, bbb}
$FIRST_3(A)$ = {ab, baa, aaa, aab}
$FIRST_3(B)$ = {bab, bba, bbb, bb}
$FIRST_3(C)$ = {aaa, aab}

$T_0 = T_S$, {ε}

$FIRST_3(AB) \oplus_3 \{\varepsilon\}$ = {abb, baa, aaa, aab}
$FIRST_3(BC) \oplus_3 \{\varepsilon\}$ = {bba, bbb, bab}

{abb, baa, aaa, aab} ∩ {bba, bbb, bab} = ∅
Since there is no conflict, we can continue...

lookahead	production	local follow set
abb }	S -> AB	for A: $FIRST_3(B)\oplus_3\{\varepsilon\}$: T_1
baa }		for B: $\{\varepsilon\}\oplus_3\{\varepsilon\}$: T_2
aaa }		
aab }		
bba }	S -> BC	for B: $FIRST_3(C)\oplus_3\{\varepsilon\}$: T_3
bbb }		for C: $\{\varepsilon\}\oplus_3\{\varepsilon\}$: T_4
bab }		

$T_1 = T_A, \{bb, bba, bbb, bab\}$

$\mathrm{FIRST}_3(ab) \oplus_3 \{bb, bba, bbb, bab\} = \{abb\}$

$\mathrm{FIRST}_3(CBba) \oplus_3 \{bb, bba, bbb, bab\} = \{aaa, aab\}$

$\mathrm{FIRST}_3(baaS) \oplus_3 \{bb, bba, bbb, bab\} = \{baa\}$

$\{abb\}\cap\{aaa, aab\}$, $\{aaa, aab\}\cap\{baa\}$, and $\{baa\}\cap\{abb\} = \emptyset$
Since there is no conflict, we can continue...

lookahead	production	local follow set
abb	A -> ab	no follow set
aaa }	A -> CBba	for C: $\mathrm{FIRST}_3(Bba) \oplus_3 \{bb, bba, bbb, bab\} = \{bbb, bba, bab\}$: T_5
aab }		for B: $\mathrm{FIRST}_3(ba) \oplus_3 \{bb, bba, bbb, bab\} = \{bab\}$: T_6
baa	A -> baaS	for S: $\{\varepsilon\} \oplus_3 \{bb, bba, bbb, bab\} = \{bb, bba, bbb, bab\}$: T_7

$T_2 = T_B, \{\varepsilon\}$

$\mathrm{FIRST}_3(bb) \oplus_3 \{\varepsilon\} = \{bb\}$

$\mathrm{FIRST}_3(bbAS) \oplus_3 \{\varepsilon\} = \{bba, bbb\}$

$\mathrm{FIRST}_3(baB) \oplus_3 \{\varepsilon\} = \{bab\}$

$\{bb\}\cap\{bba, bbb\}$, $\{bba, bbb\}\cap\{bab\}$, and $\{bab\}\cap\{bb\} = \emptyset$
Since there is no conflict, we can continue...

lookahead	production	local follow set
bb	B -> bb	no follow set
bba }	B -> bbAS	for A: $\mathrm{FIRST}_3(S) \oplus_3 \{\varepsilon\} = \{abb, baa, aaa, aab, bba, bbb, bab\}$: T_8
bbb }		for S: $\{\varepsilon\} \oplus_3 \{\varepsilon\} = \{\varepsilon\}$: T_0
bab	B -> baB	for B: $\{\varepsilon\} \oplus_3 \{\varepsilon\} = \{\varepsilon\}$: T_2

$T_3 = T_B$, {aaa, aab}

$FIRST_3(bb) \oplus_3 \{aaa, aab\} = \{bba\}$

$FIRST_3(bbAS) \oplus_3 \{aaa, aab\} = \{bba, bbb\}$

$FIRST_3(baB) \oplus_3 \{aaa, aab\} = \{bab\}$

Since bba can be generated from either B -> bb or B -> bbAS, 3 symbols look ahead is not enough. Therefore, G is not LL(3).

Theorem: A CFG G is LL(*k*) iff the following holds: if A -> ß and A -> γ are distinct productions, then $FIRST_k(ß\alpha) \cap FIRST_k(\gamma\alpha) = \emptyset$ for all wAα such that $S \overset{*}{\rightarrow} wA\alpha$.

Example: S -> (S)S | [S]S | ε   Is G LL(1)?

$FOLLOW_1(S) = \{ \varepsilon,),] \}$

$FIRST_1((S)S)\oplus_1\{\varepsilon\} \cap FIRST_1([S]S)\oplus_1\{\varepsilon\} = \{(\} \cap \{[\} = \emptyset$

$FIRST_1((S)S)\oplus_1\{)\} \cap FIRST_1([S]S)\oplus_1\{)\} = \{(\} \cap \{[\} = \emptyset$

$FIRST_1((S)S)\oplus_1\{]\} \cap FIRST_1([S]S)\oplus_1\{]\} = \{(\} \cap \{[\} = \emptyset$

$FIRST_1([S]S)\oplus_1\{\varepsilon\} \cap \{\varepsilon\}\oplus_1\{\varepsilon\} = \{[\} \cap \{\varepsilon\} = \emptyset$

$FIRST_1([S]S)\oplus_1\{)\} \cap \{\varepsilon\}\oplus_1\{)\} = \{[\} \cap \{)\} = \emptyset$

$FIRST_1([S]S)\oplus_1\{]\} \cap \{\varepsilon\}\oplus_1\{]\} = \{[\} \cap \{]\} = \emptyset$

$FIRST_1((S)S)\oplus_1\{\varepsilon\} \cap \{\varepsilon\}\oplus_1\{\varepsilon\} = \{(\} \cap \{\varepsilon\} = \emptyset$

$FIRST_1((S)S)\oplus_1\{)\} \cap \{\varepsilon\}\oplus_1\{)\} = \{(\} \cap \{)\} = \emptyset$

$FIRST_1((S)S)\oplus_1\{]\} \cap \{\varepsilon\}\oplus_1\{]\} = \{(\} \cap \{]\} = \emptyset$

This grammar, therefore, is LL(1).

Example: S -> Aa   Is G LL(1)?
A -> bAb | ε

For S: ok
For A: $FOLLOW_1(A) = \{a, b\}$

$FIRST_1(bAb)\oplus_1\{a\} \cap FIRST_1(\varepsilon)\oplus_1\{a\} = \{b\} \cap \{a\} = \emptyset$

$FIRST_1(bAb)\oplus_1\{b\} \cap FIRST_1(\varepsilon)\oplus_1\{b\} = \{b\} \cap \{b\} \neq \emptyset$

Therefore, G is not LL(1).

Example: S -> aAaa | bAba
A -> b | ε

LL(2)?

$FOLLOWS_2(S) = \{ \varepsilon \}$
$FOLLOWS_2(A) = \{aa, ba\}$

S: $FIRST_2(aAaa) \oplus_2 \{\varepsilon\} \cap FIRST_2(bAba) \oplus_2 \{\varepsilon\}$
$= \{ab, aa\} \cap \{bb\} = \emptyset$
A: $FIRST_2(b) \oplus_2 \{aa\} \cap FIRST_2(\varepsilon) \oplus_2 \{aa\} = \{ba\} \cap \{aa\} = \emptyset$
$FIRST_2(b) \oplus_2 \{ba\} \cap FIRST_2(\varepsilon) \oplus_2 \{ba\} = \{bb\} \cap \{ba\} = \emptyset$

This language is LL(2).

Example: S -> AB | BC
A -> a | CBS | bSS
B -> b | bAS | bB
C -> a | aSBC

LL(2)?

$FIRST_2(S) = \{ab, aa, ba, bb\}$
$FIRST_2(A) = \{a, ab, ba, bb, aa\}$
$FIRST_2(B) = \{b, bb, ba\}$
$FIRST_2(C) = \{a, aa, ab\}$

$FOLLOW_2(S) = \{\varepsilon, ab, ba, bb, aa, b, a\}$
$FOLLOW_2(A) = \{b, ba, bb, aa, ab\}$
$FOLLOW_2(B) = \{aa, ab, ba, bb, \varepsilon, b, a\}$
$FOLLOW_2(C) = \{bb, ba, aa, ab, \varepsilon, b, a\}$

$FIRST_2(AB) \oplus_2 \{\varepsilon\} \cap FIRST_2(BC) \oplus_2 \{\varepsilon\}$
$= \{aa, ab, ba, bb\} \cap \{ba, bb\} \neq \emptyset$

We can stop here: this language is not LL(2).