Limitations of LL(1) Parsing

- Given an LL(1) grammar we should be able to construct a parse tree given two pieces of information:
  - The current pointer into the sentence to be parsed
  - A single token lookahead

- However, just as we saw in LR(1) parsing there are certain CFGs that cannot be used to construct an LL(1) parser.
  - example: left-recursive grammars
Left-Recursive Grammars

Grammar:

\[ L : A \quad B \quad C \]
\[ A : \ 'a' \]
\[ B : \quad B \quad 'b' \quad | \quad 'b' \]
\[ C : \ 'c' \]

Start symbol: L

Parse the sentence:

\[ a \quad b \quad b \quad c \]
Left-Recursive Grammars

Grammar:

L : A B C
A : 'a'
B : { 'b' } B 'b'
| { 'b' } 'b'
C : 'c'

Parse the sentence:

a b b c

Start symbol: L

Can’t be done - problem choosing the correct alternative.
Left-Recursive Grammars

Grammar:

```
A : 'a' | B
B : A 'b'
```

Be aware, sometimes left-recursion is hidden, but it is easily exposed by computing the lookahead sets.

Start symbol: A
Left-Recursive Grammars

- Left-recursive grammars are not LL(1)!
- In most cases we can rewrite a left-recursive grammar to make it LL(1)
  - NOTE: the rewriting has to preserve the language the original grammar defined!
Eliminating Left-Recursion

- Left-recursive grammars often describe lists and other recursive structures
- In most cases those recursive structures can be described with alternative LL(1) grammars, consider:

```
A : A 'b' | 'a'
```

This left-recursive grammar defines the language of all strings that start with a single ‘a’ and finish with zero or more ‘b’ s.

E.g. a, ab, abbb, ...

```
A : B ('b')*
B : 'a'
```

This is a LL(1) grammar and defines the same language.
Let's try to eliminate left-recursion in this grammar:

```
L :    A B C
A :   'a'
B :   B 'b'
      | 'b'
C :   'c'
```
Left-Recursion in Algebraic Terms

- Left-recursion appears naturally in the grammar definitions of algebraic terms, consider:

```
exp : exp ' + ' exp
    | NUM
    | VAR
VAR : ('a'..'z')+
NUM : ('0'..'9')+
```

```
exp : exp ' + ' atom
    | atom
atom : NUM
    | VAR
VAR : ('a'..'z')+
NUM : ('0'..'9')+
```

```
exp : B (' + ' atom)*
B : atom
atom : NUM
    | VAR
VAR : ('a'..'z')+
NUM : ('0'..'9')+
```

```
exp : atom (' + ' atom)*
atom : NUM
    | VAR
VAR : ('a'..'z')+
NUM : ('0'..'9')+
```
Left-Recursion

● Here is another example which typically occurs in programming languages:

```
list      :    '[' listElements ']'  
listElements: listElements ',' element 
             |   element  
element   :    'a' | 'b' | 'c'
```

```
list      :    '[' listElements ']'  
listElements:      element (',' element)*  
element   :    'a' | 'b' | 'c'
```
The precedence of operators specifies the strength of the binding to operands relative to other operators. From algebra we know that multiplication binds tighter than addition, consider

\[ 3 \times 2 + 4 \]

We would expect that the expression \( 3 \times 2 + 4 \) gives rise to the parse tree below (why?),

```
+   
  /  
*   4
  /  
3   2
```
LL(1) Grammars and Operator Precedence

- The problem is that straight forward grammars do not give rise to the desired parse tree
- Consider the left-recursive grammar,

```
exp : exp ' + ' exp  
    | exp ' * ' exp  
    | ID  
    | INT

ID   : ('a'..'z')+
INT  : ('0'..'9')+
```

Now consider our sentence
3 * 2 + 4

Note that the obvious choice of rule constructs the wrong parse tree
We can rewrite this grammar with the precedence information (and correct associativity).

```
addexp : addexp '+' multexp
       | multexp

multexp : multexp '*' atom
         | atom

atom : ID
    | INT

ID : ('a'..'z')+
INT : ('0'..'9')+
```

Now consider our sentence

```
3 * 2 + 4
```

Now our parse trees have the correct precedence and associativity.

This approach is called ‘precedence climbing’

Problem: still left-recursive!
The LL(1) grammar below does not fare much better,

\[
\text{exp} : \text{atom} (('+' \text{atom}) | ('*' \text{atom}))^*
\]
\[
\text{atom} : \text{ID} \mid \text{INT}
\]
\[
\text{ID} : (\text{'a'..'z'})^+
\]
\[
\text{INT} : (\text{'0'..'9'})^+
\]

Consider our sentence

\[3 \times 2 + 4\]

The ANTLR parse tree has no precedence information what so ever:

We need to build operator precedence info into the grammar.
LL(1) Grammars and Operator Precedence

- Below we have a LL(1) grammar with the precedence info built in:

```
addexp : multexp ('+' multexp)* ;
multexp : atom ('*' atom)*;
atom : ID | INT ;
ID : ('a'..'z')+;
INT : ('0'..'9')+;
```

Consider our sentence

```
3 * 2 + 4
```

The ANTLR parse tree now has precedence information embedded in it:
Hints for Programming

Consider building a syntax directed interpreter for this language…

HINT: use the expression return values as an “accumulator”

```plaintext
addexp returns [int value]
  : el=multexp {$value=el.value;} ('+' e2=multexp {$value+=e2.value;})*

multexp returns [int value]
  : el=atom {$value=el.value;} ('*' e2=atom {$value*+=e2.value;})*

atom returns [int value]
  : INT {$value=Integer.parseInt($INT.text);} 

INT : ('0'..'9')+
```
Operator Associativity

- Most arithmetic operators are left-associative
  - I.e., $1 + 2 + 3$ is interpreted as $(1 + 2) + 3$
- Grammars need to respect this.

```
exp : exp '+' exp
atom : ID
    | INT
ID   : ('a'..'z')+
INT  : ('0'..'9')+
```

```
exp : exp '+' atom
    | atom
atom : ID
    | INT
ID   : ('a'..'z')+
INT  : ('0'..'9')+
```
Operator Associativity

- In ANTLR this is usually not an issue because parse trees of operators with the same precedence are represented as lists.

\[
\begin{align*}
\text{exp} &: \text{exp} \; \text{`}+` \; \text{exp} \\
&\quad | \; \text{ID} \\
&\quad | \; \text{INT}
\end{align*}
\]

\[
\begin{align*}
\text{ID} &: (\text{'a'..'z'}) + \\
\text{INT} &: (\text{'0'..'9'}) +
\end{align*}
\]

\[
\begin{align*}
\text{exp} &: \text{atom} \; (\text{`+`} \; \text{atom})^* \\
\text{atom} &: \text{ID} \\
&\quad | \; \text{INT}
\end{align*}
\]

\[
\begin{align*}
\text{ID} &: (\text{'a'..'z'}) + \\
\text{INT} &: (\text{'0'..'9'}) +
\end{align*}
\]
Operator Associativity

- However, you need to be careful that the wrong associativity “doesn’t sneak in”, consider

```
exp  :  atom ('+' exp)? ;
atom  :  ID
      |  INT
   ;
ID    :  ('a'..'z')+;
INT   :  ('0'..'9')+;
```

```
exp  :  exp '+' exp
      |  ID
      |  INT
ID    :  ('a'..'z')+;
INT   :  ('0'..'9')+;
```
Other Non-LL(1) Issues

- Left-recursion is not the only problem that make grammars non-LL(1), consider:

```plaintext
grammar declN;
options{k=1;}

decl : declSpec ID '=' INT
    | declSpec ID '=' FLOAT
    | declSpec ID
    ;

declSpec : 'int'
    | 'float'
    ;

ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
INT : ('0'..'9')+;
FLOAT : ('0'..'9')+ '.' ('0'..'9')*;
```
Other Non-LL(1) Issues

- The grammar in the previous slide could be fixed in a number of ways to be processed by ANTLR
  - If we keep the option \{k=1;\} then we will have to rewrite the grammar to a LL(1) grammar. How?
  - We can rewrite the options statement to \{k=4;\} turning the grammar into a LL(4) grammar. Why would that work?
  - We can remove the option statement, in that case ANTLR will revert to its default parsing algorithm which supports LL(*) grammars, i.e., grammars that can be disambiguated with arbitrary length look ahead
    - How much look ahead is needed in this grammar to make a decision on which rule to pick?

- Note: Left-recursive grammars cannot be considered LL(*) grammar, there is no amount of look ahead that can disambiguate a left-recursive rule. (Why?)
Consider the following grammar:

\[
\text{exp : } \begin{cases} \text{exp + 'b'} \\ \text{b'} \end{cases}
\]

Problematic LL(2) parsing:
1. start with exp
2. lookahead = “b+”
3. pick first rule: exp + ‘b’
4. need to expand leftmost non-terminal
5. expand exp
6. lookahead = “b+”
7. pick first rule: exp + ‘b’
8. need to expand leftmost non-terminal
9. expand exp
10. lookahead = “b+”
11. …

Rules are uniquely identified, but…when we try to use these rules in LL(2) parsing our parser gets stuck in an infinite loop!
Exercise: Construct a grammar that encodes operator precedence correctly for the two operators @ and #; each of which operates on integer values. The operator @ has a higher precedence than operator #.

- E.g. 3 # 4 @ 5 should be interpreted as 3 # (4 @ 5)

Write a grammar where we consider all the operators to be left-assoc. and one where we consider them to be right-assoc.
Assignment

- Assignment #3 – see website