Abstract stack machines for LL and LR parsing

Hayo Thielecke

August 13, 2015
We decompose the parsing problem into two parts:

**What** the parser does: a stack machine, possibly nondeterministic. The “what” is very simple and elegant and has not changed in 50 years.

**How** the parser knows which step to take, making the machine deterministic. The “how” can be very complex, e.g. LALR(1) item or LL(*) constructions; there is still ongoing research, even controversy.

There are tools for computing the “how”, e.g. yacc and ANTLR. You need to understand some of the theory for really using tools, e.g. what does it mean when yacc complains about a reduce/reduce conflict?
Exercise

Consider the language of matching round and square brackets. For example, these strings are in the language:

\[
[(\)]
\]

and

\[
[(\)](\)(\)([])
\]

but this is not:

\[
[(\)]
\]

How would you write a program that recognizes this language, so that a string is accepted if and only if all the brackets match? It is not terribly hard. But are you sure your solution is correct? You will see how the LL and LR machines solve this problem, and are correct by construction.
Parsing stack and function call stack

A useful analogy: a grammar rule

\[ A \rightarrow B \ C \]

is like a function definition

```c
void A()
{
    B();
    C();
}
```

Function calls also use a stack
ANTLR uses the function call stack as its parsing stack
yacc maintains its own parsing stack
A context-free grammar consists of

- some terminal symbols $a, b, \ldots, +, )$, \ldots
- some non-terminal symbols $A, B, S,$ \ldots
- a distinguished non-terminal start symbol $S$
- some rules of the form

$$A \rightarrow X_1 \ldots X_n$$

where $n \geq 0$, $A$ is a non-terminal, and the $X_i$ are symbols.
Mathematicians and computer scientists are inordinately fond of Greek letters.

\[ \alpha \quad \text{alpha} \]
\[ \beta \quad \text{beta} \]
\[ \gamma \quad \text{gamma} \]
\[ \varepsilon \quad \text{epsilon} \]
\[ \sigma \quad \text{sigma} \]
Notational conventions for grammars

- We will use Greek letters $\alpha, \beta, \gamma, \sigma \ldots$, to stand for strings of symbols that may contain both terminals and non-terminals.
- In particular, $\varepsilon$ is used for the empty string (of length 0).
- We will write $A, B, \ldots$ for non-terminals.
- We will write $S$ for the start symbol.
- Terminal symbols are usually written as lower case letters $a, b, c, \ldots$
- These conventions are handy once you get used to them and are found in most books, e.g. the “Dragon Book”
Derivations

- If $A \rightarrow \alpha$ is a rule, we can replace $A$ by $\alpha$ for any strings $\beta$ and $\gamma$ on the left and right:

  $$\beta A \gamma \Rightarrow \beta \alpha \gamma$$

  This is one derivation step.

- A string $w$ consisting only of terminal symbols is generated by the grammar if there is a sequence of derivation steps leading to it from the start symbol $S$:

  $$S \Rightarrow \cdots \Rightarrow w$$
An example derivation

Consider this grammar:

\[
\begin{align*}
  D & \rightarrow [ D ] D \\
  D & \rightarrow ( D ) D \\
  D & \rightarrow 
\end{align*}
\]

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[
D \Rightarrow [ D ] D \\
D \Rightarrow [ ] D \\
D \Rightarrow [ ] [ ] \\
D \Rightarrow [ ] [ ]
\]
An example derivation

Consider this grammar:

\[
D \rightarrow [D]D \quad (1) \\
D \rightarrow (D)D \quad (2) \\
D \rightarrow \quad (3)
\]

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[
D \\
\Rightarrow [D]D
\]
An example derivation

Consider this grammar:

\[ D \rightarrow \left[ D \right] D \quad (1) \]
\[ D \rightarrow \left( D \right) D \quad (2) \]
\[ D \rightarrow \quad (3) \]

There is a unique leftmost derivation for each string in the language. For example, we derive \[ [ ] [ ] \] as follows:

\[
\begin{align*}
D & \Rightarrow \left[ D \right] D \\
& \Rightarrow \left[ D \right] D \\
& \Rightarrow [ ] D
\end{align*}
\]
An example derivation

Consider this grammar:

\[ D \rightarrow [D]D \quad (1) \]
\[ D \rightarrow (D)D \quad (2) \]
\[ D \rightarrow \quad (3) \]

There is a unique leftmost derivation for each string in the language. For example, we derive \([\quad] [\quad]\) as follows:

\[
D \\
\Rightarrow [D]D \\
\Rightarrow [\quad]D \\
\Rightarrow [\quad][D]D
\]
An example derivation

Consider this grammar:

\[ D \rightarrow [ D ] D \]  \hspace{1cm} (1)
\[ D \rightarrow ( D ) D \]  \hspace{1cm} (2)
\[ D \rightarrow \]  \hspace{1cm} (3)

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[
\begin{align*}
D &\Rightarrow [ D ] D \\
&\Rightarrow [ ] D \\
&\Rightarrow [ ] [ D ] D \\
&\Rightarrow [ ] [ ] D
\end{align*}
\]
An example derivation

Consider this grammar:

\[
D \rightarrow [D]D \\
D \rightarrow (D)D \\
D \rightarrow
\]

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[
D \\
\Rightarrow [D]D \\
\Rightarrow [ ]D \\
\Rightarrow [ ] [D]D \\
\Rightarrow [ ] [ ] D \\
\Rightarrow [ ] [ ]
\]
Lexer and parser

The raw input is processed by the lexer before the parser sees it. For instance, `while` or `4223666` count as a single symbol for the purpose of parsing. Lexers can be automagically generated (just like parsers by parser generators. Example: lex

```
regular expression → deterministic finite automaton
```

We’ll skip this phase of the compiler. 😞 Parsing is much harder and has more interesting ideas. 😊
Parsing stack machines

The states of the machines are of the form

\[ \langle \sigma, w \rangle \]

where

- \( \sigma \) is the stack, a string of symbols which may include non-terminals
- \( w \) is the remaining input, a string of input symbols no non-terminal symbols may appear in the input

Transitions or steps are of the form

\[
\begin{align*}
\langle \sigma_1, w_1 \rangle & \quad \rightarrow \quad \langle \sigma_2, w_2 \rangle \\
\text{old state} & \quad \rightarrow \quad \text{new state}
\end{align*}
\]

- pushing or popping the stack changes \( \sigma_1 \) to \( \sigma_2 \)
- consuming input changes \( w_1 \) to \( w_2 \)
LL vs LR idea

There are two main classes of parsers: LL and LR. Both use a parsing stack, but in different ways.

**LL** the stack contains a prediction of what the parser expects to see in the input

**LR** the stack contains a reduction of what the parser has already seen in the input

Which is more powerful, LL or LR?
LL vs LR idea

There are two main classes of parsers: LL and LR. Both use a parsing stack, but in different ways.

- **LL** the stack contains a prediction of what the parser expects to see in the input
- **LR** the stack contains a reduction of what the parser has already seen in the input

Which is more powerful, LL or LR?
LL: Never make predictions, especially about the future.
LR: Benefit of hindsight
Theoretically, LR is much more powerful than LL.
But LL is much easier to understand.
Abstract and less abstract machines

You could easily implement these parsing stack machines when they are deterministic.

- In OCAML, Haskell, Agda:
  state = two lists of symbols
  transitions by pattern matching

- In C:
  state = stack pointer + input pointer
  yacc does this, plus an LALR(1) automaton
Deterministic and nondeterministic machines

The machine is deterministic if for every $\langle \sigma_1, w_1 \rangle$, there is at most one state $\langle \sigma_2, w_2 \rangle$ such that

$\langle \sigma_1, w_1 \rangle \xrightarrow{} \langle \sigma_2, w_2 \rangle$

In theory, one uses non-deterministic parsers (see PDA in Models of Computation).
In compilers, we want deterministic parser for efficiency (linear time).
Some real parsers (ANTLR) tolerate some non-determinism and so some backtracking.
Parser generators and the LL and LR machines

The LL and LR machines:
- encapsulate the main ideas (stack = prediction vs reduction)
- can be used for abstract reasoning, like partial correctness
- cannot be used off the shelf, since they are nondeterministic

A parser generator:
- computes information that makes these machines deterministic
- does not work on all grammars
- some grammars are not suitable for some (or all) deterministic parsing techniques
- parser generators produce errors such as reduce/reduce conflicts
- we may redesign our grammar to make the parser generator work
Examples of parser generators and their parsing principles

ANTLR: LL(k) for any k, also called LL(*)
Menhir for OCAML: LR(1)
yacc/bison, ocamlyacc: LALR(1)
The numbers refer to the amount of lookahead.
LALR(1) is an version of LR(1) that uses less main memory
—useful in the 1970s. Now, not so much.
LL parsing stack machine

Assume a fixed context-free grammar. We construct the LL machine for that grammar.
The top of stack is on the left.

\[ \langle A \sigma, w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]
\[ \langle a \sigma, aw \rangle \xrightarrow{\text{match}} \langle \sigma, w \rangle \]

\[ \langle S, w \rangle \quad \text{is the initial state for input } w \]
\[ \langle \varepsilon, \varepsilon \rangle \quad \text{is the accepting state} \]
Accepting a given input in the LL machine

Definition: An input string \( w \) is accepted if and only if there is a sequence of machine steps leading to the accepting state:

\[
\langle S, w \rangle \rightarrow \cdots \rightarrow \langle \varepsilon, \varepsilon \rangle
\]

Theorem: an input string is accepted if and only if it can be derived by the grammar.
More precisely: LL machine run \( \cong \) leftmost derivation in the grammar
Stretch exercise: prove this in Agda
Consider this grammar

\[ S \rightarrow L \, b \]
\[ L \rightarrow a \, L \]
\[ L \rightarrow \]

Show how the LL machine for this grammar can accept the input \textit{a a a b}. 
LL machine run example

\[ S \rightarrow Lb \quad \langle S, aab \rangle \]
\[ L \rightarrow aL \]
\[ L \rightarrow \]
LL machine run example

\[
\begin{align*}
S & \to Lb \\
L & \to aL \quad \langle S, a a b \rangle \\
L & \to \quad \langle Lb, a a b \rangle \\
\end{align*}
\]
LL machine run example

\[
\begin{align*}
S & \rightarrow Lb \\
L & \rightarrow aL \\
L & \rightarrow \\
\langle S, aab \rangle & \xrightarrow{\text{predict}} \langle Lb, aab \rangle \\
\langle aLb, aab \rangle & \xrightarrow{\text{predict}} \langle aLb, aab \rangle \\
\langle Lb, ab \rangle & \xrightarrow{\text{match}} \langle Lb, b \rangle \\
\langle b, b \rangle & \xrightarrow{\text{match}} \langle \varepsilon, \varepsilon \rangle \\
\end{align*}
\]
LL machine run example

\[
S \rightarrow L b \\
L \rightarrow a L \\
L \rightarrow \langle S, a a b \rangle \\
predict \rightarrow \langle L b, a a b \rangle \\
predict \rightarrow \langle a L b, a a b \rangle \\
match \rightarrow \langle L b, a b \rangle \\
predict \rightarrow \langle b, b \rangle \\
match \rightarrow \langle \epsilon, \epsilon \rangle \checkmark
\]
LL machine run example

\[ S \rightarrow L b \]
\[ L \rightarrow a L \]
\[ L \rightarrow \]

\[ \langle S, a a b \rangle \]
\[ \text{predict} \rightarrow \langle L b, a a b \rangle \]
\[ \text{predict} \rightarrow \langle a L b, a a b \rangle \]
\[ \text{match} \rightarrow \langle L b, a b \rangle \]
\[ \text{predict} \rightarrow \langle a L b, a b \rangle \]

\[ \rightarrow \langle \epsilon, \epsilon \rangle \]
\[ \checkmark \]
LL machine run example

\[
\begin{align*}
S & \rightarrow L \ b \\
L & \rightarrow a \ L \\
L & \rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\langle S , a \ a \ b \rangle & \rightarrow \langle L \ b , a \ a \ b \rangle \\
\text{predict} & \rightarrow \langle a \ L \ b , a \ a \ b \rangle \\
\text{match} & \rightarrow \langle L \ b , a \ b \rangle \\
\text{predict} & \rightarrow \langle a \ L \ b , a \ b \rangle \\
\text{match} & \rightarrow \langle L \ b , b \rangle \\
\end{align*}
\]
LL machine run example

\[ S \rightarrow Lb \]
\[ L \rightarrow aL \]
\[ L \rightarrow \]

\[ \langle S, aab \rangle \]
\[ \text{predict} \rightarrow \langle Lb, aab \rangle \]
\[ \text{predict} \rightarrow \langle aLb, aab \rangle \]
\[ \text{match} \rightarrow \langle Lb, ab \rangle \]
\[ \text{predict} \rightarrow \langle aLb, ab \rangle \]
\[ \text{match} \rightarrow \langle Lb, b \rangle \]
\[ \text{predict} \rightarrow \langle b, b \rangle \]
\[ \checkmark \]
LL machine run example

\[
\begin{align*}
S & \rightarrow L \ b \\
L & \rightarrow a \ L \\
L & \rightarrow
\end{align*}
\]

\[
\begin{align*}
\langle S, a \ a \ b \rangle \quad & \text{predict} \quad \langle L \ b, a \ a \ b \rangle \\
\langle L \ b, a \ a \ b \rangle \quad & \text{predict} \quad \langle a \ L \ b, a \ a \ b \rangle \\
\langle a \ L \ b, a \ a \ b \rangle \quad & \text{match} \quad \langle L \ b, a \ b \rangle \\
\langle a \ L \ b, a \ b \rangle \quad & \text{predict} \quad \langle L \ b, b \rangle \\
\langle L \ b, b \rangle \quad & \text{predict} \quad \langle b, b \rangle \\
\langle b, b \rangle \quad & \text{match} \quad \langle \varepsilon, \varepsilon \rangle
\end{align*}
\]
LL machine run example

\[
\begin{align*}
S &\to L b \\
L &\to a L \\
L &\to
\end{align*}
\]

\[
\begin{align*}
\langle S, a a b \rangle &\xrightarrow{\text{predict}} \langle L b, a a b \rangle \\
\langle a L b, a a b \rangle &\xrightarrow{\text{match}} \langle L b, a b \rangle \\
\langle a L b, a b \rangle &\xrightarrow{\text{predict}} \langle L b, a b \rangle \\
\langle L b, a b \rangle &\xrightarrow{\text{match}} \langle L b, b \rangle \\
\langle b, b \rangle &\xrightarrow{\text{predict}} \langle b, b \rangle \\
\langle \varepsilon, \varepsilon \rangle &\xrightarrow{\text{match}} \langle \varepsilon, \varepsilon \rangle \checkmark
\end{align*}
\]
LL machine run example: what should not happen

\[ S \rightarrow L b \]
\[ L \rightarrow a L \]
\[ L \rightarrow \]
LL machine run example: what should not happen

\[
\begin{align*}
S & \rightarrow \ L \ b \\
\rightarrow & \\
L & \rightarrow \ a \ L \\
\rightarrow & \\
L & \rightarrow \\
\rightarrow & \\
\rightarrow & \\
\rightarrow & \\
\rightarrow & \langle S, \ a \ a \ b \rangle \\
\rightarrow & \langle L \ b, \ a \ a \ b \rangle \\
\rightarrow & \langle L \ b, \ a \ a \ b \rangle
\end{align*}
\]
LL machine run example: what should not happen

\[ \begin{align*}
S & \rightarrow \ L \ b \\
L & \rightarrow \ a \ L \\
L & \rightarrow \\
\end{align*} \]

\[
\langle S, \ a\ a\ b \rangle
\]

\[
\text{predict} \rightarrow \ \langle L\ b, \ a\ a\ b \rangle
\]

\[
\text{predict} \rightarrow \ \langle b, \ a\ a\ b \rangle \ \frown
\]
LL machine run example: what should not happen

When it makes bad nondeterministic choice, the LL machine gets stuck.
Making the LL machine deterministic

- Idea: we use one symbol of lookahead to guide the predict moves. 
  ⇒ LL(1) machine.
- Formally: FIRST and FOLLOW construction.
- Can be done by hand, though tedious
- The construction does not work for all grammars!
- Real-world: ANTLR does a more powerful version, LL(k) for any k.
FIRST and FOLLOW

We define FIRST, FOLLOW and nullable:

- A terminal symbol $b$ is in $\text{FIRST}(\alpha)$ if there exist a $\beta$ such that
  \[ \alpha \Rightarrow^* b \beta \]
  that is, $b$ is the first symbol in something derivable from $\alpha$

- A terminal symbol $b$ is in $\text{FOLLOW}(X)$ if there exist $\alpha$ and $\beta$ such that
  \[ S \Rightarrow^* \alpha X b \gamma \]
  that is, $b$ follows $X$ in some derivation

- $X$ is nullable if
  \[ X \Rightarrow^* \epsilon \]
  that is, we can derive the empty string from it
FIRST and FOLLOW examples

Consider

\[
S \rightarrow L \, b \\
L \rightarrow a \, L \\
L \rightarrow \ \\
\]

Then

\[
a \in FIRST(L) \\
b \in FOLLOW(L) \\
L \text{ is nullable}
\]
LL(1) machine

This is a predictive parser with 1 symbol of lookahead.

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]
\[ \text{and } b \in \text{FIRST}(\alpha) \]

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \varepsilon \]
\[ \text{and } b \in \text{FOLLOW}(A) \]

\[ \langle a \sigma, a w \rangle \xrightarrow{\text{match}} \langle \sigma, w \rangle \]

\[ \langle S, w \rangle \]
\[ \text{is the initial state for input } w \]

\[ \langle \varepsilon, \varepsilon \rangle \]
\[ \text{is the accepting state} \]
Suppose the LL(1) machine reaches a state of the form

\[ \langle a \sigma , b w \rangle \]

where \( a \neq b \). Then the machine can report an error, like expecting \( a \) found \( b \) in the input instead.

Similarly, if it reaches a state of the form

\[ \langle \varepsilon , w \rangle \]

where \( w \neq \varepsilon \), the machine can report unexpected input \( w \) at the end.
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow L b \\
L & \rightarrow a L \\
L & \rightarrow
\end{align*}
\]

\[
\langle S, a a b \rangle
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{array}{l}
S \rightarrow Lb \\
L \rightarrow aL \\
L \rightarrow \langle S, a a b \rangle \\
\end{array}
\]

\[
\langle Lb, a a b \rangle \quad \text{as } a \in \text{FIRST}(L)
\]
**LL(1) machine run example**

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow Lb \\
L & \rightarrow aL \\
L & \rightarrow \\
\end{align*}
\]

\[
\langle S, aab \rangle
\]

\[
\langle Lb, aab \rangle \quad \text{as} \quad a \in \text{FIRST}(L)
\]

\[
\langle aLb, aab \rangle \quad \text{as} \quad a \in \text{FIRST}(L)
\]

\[
\langle Lb, ab \rangle \\
\langle aLb, ab \rangle \\
\langle ab, b \rangle \quad \text{as} \quad b \in \text{FOLLOW}(L)
\]

\[
\langle \epsilon, \epsilon \rangle \quad \checkmark
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow L b \\
L & \rightarrow a L \\
L & \rightarrow \\
\end{align*}
\]

\[
\begin{array}{c}
\langle S, a a b \rangle \\
\text{predict} \quad \langle L b, a a b \rangle \text{ as } a \in \text{FIRST}(L) \\
\text{predict} \quad \langle a L b, a a b \rangle \text{ as } a \in \text{FIRST}(L) \\
\text{match} \quad \langle L b, a b \rangle \\
\end{array}
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{array}{c}
S \rightarrow L b \\
L \rightarrow a L \\
L \rightarrow \\
\end{array}
\]

\[
\begin{align*}
\langle S, a a b \rangle \\
\text{predict} & \rightarrow \langle L b, a a b \rangle \text{ as } a \in \text{FIRST}(L) \\
\text{predict} & \rightarrow \langle a L b, a a b \rangle \text{ as } a \in \text{FIRST}(L) \\
\text{match} & \rightarrow \langle L b, a b \rangle \\
\text{predict} & \rightarrow \langle a L b, a b \rangle \text{ as } a \in \text{FIRST}(L) \\
\end{align*}
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{array}{c}
S & \rightarrow & L \ b \\
L & \rightarrow & a \ L \\
L & \rightarrow &
\end{array}
\]

\[
\begin{align*}
\langle S, a \ a \ b \rangle \\
\rightarrow \text{ predict} & \Rightarrow \langle L \ b, a \ a \ b \rangle \text{ as } a \in \text{FIRST}(L) \\
\rightarrow \text{ predict} & \Rightarrow \langle a \ L \ b, a \ a \ b \rangle \text{ as } a \in \text{FIRST}(L) \\
\rightarrow \text{ match} & \Rightarrow \langle L \ b, a \ b \rangle \\
\rightarrow \text{ predict} & \Rightarrow \langle a \ L \ b, a \ b \rangle \text{ as } a \in \text{FIRST}(L) \\
\rightarrow \text{ match} & \Rightarrow \langle L \ b, b \rangle \\
\rightarrow \text{ match} & \Rightarrow \langle \epsilon, \epsilon \rangle \\
\end{align*}
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow L b \\
L & \rightarrow a L \\
L & \rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\langle S, a a b \rangle \\
\xrightarrow{\text{predict}} \langle L b, a a b \rangle & \quad \text{as } a \in \text{FIRST}(L) \\
\xrightarrow{\text{predict}} \langle a L b, a a b \rangle & \quad \text{as } a \in \text{FIRST}(L) \\
\xrightarrow{\text{match}} \langle L b, a b \rangle \\
\xrightarrow{\text{predict}} \langle a L b, a b \rangle & \quad \text{as } a \in \text{FIRST}(L) \\
\xrightarrow{\text{match}} \langle L b, b \rangle \\
\xrightarrow{\text{predict}} \langle b, b \rangle & \quad \text{as } b \in \text{FOLLOW}(L)
\end{align*}
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow Lb \\
L & \rightarrow aL \\
L & \rightarrow
\end{align*}
\]

\[
\text{predict} \rightarrow \langle Lb, aab \rangle \text{ as } a \in \text{FIRST}(L)
\]

\[
\text{predict} \rightarrow \langle aLb, aab \rangle \text{ as } a \in \text{FIRST}(L)
\]

\[
\text{match} \rightarrow \langle Lb, ab \rangle
\]

\[
\text{predict} \rightarrow \langle aLb, ab \rangle \text{ as } a \in \text{FIRST}(L)
\]

\[
\text{match} \rightarrow \langle Lb, b \rangle
\]

\[
\text{predict} \rightarrow \langle b, b \rangle \text{ as } b \in \text{FOLLOW}(L)
\]

\[
\text{match} \rightarrow \langle \varepsilon, \varepsilon \rangle
\]
LL(1) machine run example

Just like LL machine, but now deterministic

\[
\begin{align*}
S & \rightarrow Lb \\
L & \rightarrow aL \\
L & \rightarrow
\end{align*}
\]

\[
\begin{align*}
\langle S, aab \rangle & \rightarrow \langle Lb, aab \rangle \text{ as } a \in \text{FIRST}(L) \\
\rightarrow \langle aLb, aab \rangle & \text{ as } a \in \text{FIRST}(L) \\
\text{match} & \rightarrow \langle Lb, ab \rangle \\
\rightarrow \langle aLb, ab \rangle & \text{ as } a \in \text{FIRST}(L) \\
\text{match} & \rightarrow \langle Lb, b \rangle \\
\rightarrow \langle b, b \rangle & \text{ as } b \in \text{FOLLOW}(L) \\
\text{match} & \rightarrow \langle \varepsilon, \varepsilon \rangle \checkmark
\end{align*}
\]
Is the LL(1) machine deterministic?

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \quad \text{if there is a rule} \quad A \rightarrow \alpha \]
\[ \text{and} \quad b \in \text{FIRST}(\alpha) \]

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \quad \text{if there is a rule} \quad A \rightarrow \varepsilon \]
\[ \text{and} \quad b \in \text{FOLLOW}(A) \]
Is the LL(1) machine deterministic?

\[
\langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \\
\text{and } b \in \text{FIRST}(\alpha)
\]

\[
\langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \varepsilon \\
\text{and } b \in \text{FOLLOW}(A)
\]

For some grammars, there may be:

\[
\text{FIRST/FIRST conflicts} \quad A \rightarrow \alpha_1 \quad A \rightarrow \alpha_2 \\
\text{FIRST/FOLLOW conflicts} \quad A \rightarrow \alpha \quad \text{FIRST}(\alpha) \cap \text{FOLLOW}(A) \neq \emptyset
\]

NB: FIRST/FIRST conflict do not mean that the grammar is ambiguous. Ambiguous means different parse trees for the same string.
Is the LL(1) machine deterministic?

\[
\langle A \sigma , b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma , b w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \\
\text{and } b \in \text{FIRST}(\alpha)
\]

\[
\langle A \sigma , b w \rangle \xrightarrow{\text{predict}} \langle \sigma , b w \rangle \quad \text{if there is a rule } A \rightarrow \varepsilon \\
\text{and } b \in \text{FOLLOW}(A)
\]

For some grammars, there may be:

- FIRST/FIRST conflicts 😞

NB: FIRST/FIRST conflict do not mean that the grammar is ambiguous. Ambiguous means different parse trees for the same string.
Is the LL(1) machine deterministic?

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \text{ if there is a rule } A \rightarrow \alpha \]
and \( b \in \text{FIRST}(\alpha) \)

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \text{ if there is a rule } A \rightarrow \varepsilon \]
and \( b \in \text{FOLLOW}(A) \)

For some grammars, there may be:

- FIRST/FIRST conflicts \( \bigcirc \)

\[ A \rightarrow \alpha_1 \quad A \rightarrow \alpha_2 \quad \text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \neq \emptyset \]
Is the LL(1) machine deterministic?

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]
and \( b \in \text{FIRST}(\alpha) \)

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \varepsilon \]
and \( b \in \text{FOLLOW}(A) \)

For some grammars, there may be:

- **FIRST/FIRST conflicts 😞**
  
  \[ A \rightarrow \alpha_1 \quad A \rightarrow \alpha_2 \quad \text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \neq \emptyset \]

- **FIRST/FOLLOW conflicts 😞**
Is the LL(1) machine deterministic?

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]
\[ \text{and } b \in \text{FIRST}(\alpha) \]

\[ \langle A \sigma, b w \rangle \xrightarrow{\text{predict}} \langle \sigma, b w \rangle \quad \text{if there is a rule } A \rightarrow \varepsilon \]
\[ \text{and } b \in \text{FOLLOW}(A) \]

For some grammars, there may be:

- **FIRST/FIRST conflicts**:
  \[ A \rightarrow \alpha_1 \quad A \rightarrow \alpha_2 \quad \text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \neq \emptyset \]

- **FIRST/FOLLOW conflicts**:  
  \[ A \rightarrow \alpha \quad \text{FIRST}(\alpha) \cap \text{FOLLOW}(A) \neq \emptyset \]

NB: FIRST/FIRST conflict do not mean that the grammar is ambiguous. Ambiguous means different parse trees for the same string.
Computing FIRST and FOLLOW

for each symbol $X$, $\text{nullable}[X]$ is initialised to false
for each symbol $X$, $\text{FOLLOW}[X]$ is initialised to the empty set
for each terminal symbol $a$, $\text{FIRST}[a]$ is initialised to $\{a\}$
for each non-terminal symbol $A$, $\text{FIRST}[A]$ is initialised to the empty set
repeat
  for each production $X \rightarrow Y_1 \ldots Y_k$
    if all the $Y_i$ are nullable
      then set $\text{nullable}[X]$ to true
  for each $i$ from 1 to $k$, and $j$ from $i+1$ to $k$
    if $Y_1, \ldots, Y_{i-1}$ are all nullable
      then add all symbols in $\text{FIRST}[Y_i]$ to $\text{FIRST}[X]$
    if $Y_{i+1}, \ldots, Y_k$ are all nullable
      then add all symbols in $\text{FOLLOW}[X]$ to $\text{FOLLOW}[Y_i]$
    if $Y_{i+1}, \ldots, Y_{j-1}$ are all nullable
      then add all symbols in $\text{FIRST}[Y_j]$ to $\text{FOLLOW}[Y_i]$
until $\text{FIRST}$, $\text{FOLLOW}$ and $\text{nullable}$ did not change in this iteration
LL(1) machine exercise

Consider the grammar

\[
\begin{align*}
D & \rightarrow [ D ] D \\
D & \rightarrow ( D ) D \\
D & \rightarrow 
\end{align*}
\]

Implement the LL(1) machine for this grammar in a language of your choice, preferably C.
Bonus for writing the shortest possible implementation in C.
LR machine

Assume a fixed context free grammar. We construct the LL machine for it.
The top of stack is on the right.

\[
\langle \sigma, aw \rangle \xrightarrow{\text{shift}} \langle \sigma a, w \rangle \\
\langle \sigma \alpha, w \rangle \xrightarrow{\text{reduce}} \langle \sigma A, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha
\]

\[
\langle \varepsilon, w \rangle \quad \text{is the initial state for input } w \\
\langle S, \varepsilon \rangle \quad \text{is the accepting state}
\]
Accepting a given input in the LR machine

Definition: An input string $w$ is accepted if and only if there is a sequence of machine steps leading to the accepting state:

$$\langle \varepsilon, w \rangle \rightarrow \cdots \rightarrow \langle S, \varepsilon \rangle$$

Theorem: an input string is accepted if and only if it can be derived by the grammar.
More precisely: LR machine run $\cong$ rightmost derivation in the grammar
Stretch exercise: prove this in Agda
LL vs LR example

Consider this grammar

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow c \\
B & \rightarrow d
\end{align*}
\]

Show how the LL and LR machine can accept the input \( c\ d \).
Is the LR machine deterministic?

\[
\langle \sigma, aw \rangle \xrightarrow{\text{shift}} \langle \sigma a, w \rangle
\]

\[
\langle \sigma \alpha, w \rangle \xrightarrow{\text{reduce}} \langle \sigma A, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha
\]
Is the LR machine deterministic?

\[ \langle \sigma, aw \rangle \xrightarrow{\text{shift}} \langle \sigma a, w \rangle \]

\[ \langle \sigma \alpha, w \rangle \xrightarrow{\text{reduce}} \langle \sigma A, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]

For some grammars, there may be:

- shift/reduce conflicts
- reduce/reduce conflicts
LL vs LR in more detail

LL

\[ \langle A\sigma, w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]

\[ \langle a\sigma, aw \rangle \xrightarrow{\text{match}} \langle \sigma, w \rangle \]

LR

\[ \langle \sigma, aw \rangle \xrightarrow{\text{shift}} \langle \sigma a, w \rangle \]

\[ \langle \sigma \alpha, w \rangle \xrightarrow{\text{reduce}} \langle \sigma A, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]

The LR machines can make its choices after it has seen the right hand side of a rule.
LL vs LR in more detail

LL

\[ \langle A \sigma, w \rangle \xrightarrow{\text{predict}} \langle \alpha \sigma, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]

\[ \langle a \sigma, a w \rangle \xrightarrow{\text{match}} \langle \sigma, w \rangle \]

LR

\[ \langle \sigma, a w \rangle \xrightarrow{\text{shift}} \langle \sigma a, w \rangle \]

\[ \langle \sigma \alpha, w \rangle \xrightarrow{\text{reduce}} \langle \sigma A, w \rangle \quad \text{if there is a rule } A \rightarrow \alpha \]

The LR machines can make its choices after it has seen the right hand side of a rule.
Here is a simple grammar:

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow B \\
A & \rightarrow a b \\
B & \rightarrow a c
\end{align*}
\]

One symbol of lookahead is not enough for the LL machine. An LR machine can look at the top of its stack and base its choice on that.
LR machine run example 😊

\[
\begin{align*}
S &\rightarrow A & \langle \varepsilon, a b \rangle \\
S &\rightarrow B \\
A &\rightarrow a b \\
B &\rightarrow a c
\end{align*}
\]
LR machine run example 😊

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow B \\
A & \rightarrow ab \\
B & \rightarrow ac
\end{align*}
\]

\[\langle \varepsilon, ab \rangle \]

\[\text{shift} \rightarrow \langle a, b \rangle\]
LR machine run example 😊

\[
\begin{align*}
S & \rightarrow A & \langle \varepsilon, a b \rangle \\
S & \rightarrow B & \text{shift} \quad \langle a, b \rangle \\
A & \rightarrow a b & \text{shift} \quad \langle a b, \varepsilon \rangle \\
B & \rightarrow a c
\end{align*}
\]
LR machine run example 😊

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow B \\
A & \rightarrow a \ b \\
B & \rightarrow a \ c \\
\langle \varepsilon, a \ b \rangle & \xrightarrow{\text{shift}} \langle a, b \rangle \\
\langle a \ b, \varepsilon \rangle & \xrightarrow{\text{shift}} \langle a \ b, \varepsilon \rangle \\
\langle A, \varepsilon \rangle & \xrightarrow{\text{reduce}} \langle A, \varepsilon \rangle 
\end{align*}
\]
LR machine run example 😊

\[
\begin{align*}
S & \rightarrow A \\
S & \rightarrow B \\
A & \rightarrow ab \\
B & \rightarrow ac \\
\langle \varepsilon, ab \rangle & \xrightarrow{\text{shift}} \langle a, b \rangle \\
\langle a b, \varepsilon \rangle & \xrightarrow{\text{shift}} \langle A, \varepsilon \rangle \\
\langle S, \varepsilon \rangle & \xrightarrow{\text{reduce}} \langle S, \varepsilon \rangle
\end{align*}
\]
LR machine run example 😊

\[
\begin{align*}
S & \rightarrow A & \langle \varepsilon, a b \rangle \\
S & \rightarrow B & \text{shift} \quad \langle a, b \rangle \\
A & \rightarrow a b & \text{shift} \quad \langle a b, \varepsilon \rangle \\
B & \rightarrow a c & \text{reduce} \quad \langle A, \varepsilon \rangle \\
& & \text{reduce} \quad \langle S, \varepsilon \rangle \checkmark
\end{align*}
\]
Experimenting with ANTLR and Menhir errors

Construct some grammar rules that are not:
LL(k) for any k and feed them to ANTLR
LR(1) and feed them to Menhir
and observe the error messages.
The error messages are allegedly human-readable.
It helps if you understand LL and LR.
How to make the LR machine deterministic

- Construction of LR items
- Much more complex than FIRST/FOLLOW construction
- Even more complex: LALR(1) items, to consume less memory
- You really want a tool to compute it for you
- Real world: yacc performs LALR(1) construction.
- Generations of CS students had to simulate the LALR(1) automaton in the exam
- Hardcore: compute LALR(1) items by hand in the exam
  fun: does not fit on a sheet; if you make a mistake it never stops
- But I don’t think that teaches you anything
Problem: ambiguous grammars 😞

A grammar is ambiguous if there is a string that has more than one parse tree.

Standard example:

\[
E \rightarrow E - E \\
E \rightarrow 1
\]

One such string is 1-1-1. It could mean (1-1)-1 or 1-(1-1) depending on how you parse it.

Ambiguous grammars are a problem for parsing, as we do not know which tree is intended.

Note: do not confuse ambiguous with FIRST/FIRST conflict.
In fact, this grammar also has a FIRST/FIRST conflict.

\[ E \rightarrow E - E \]
\[ E \rightarrow 1 \]

1 is in FIRST of both rules
\Rightarrow predictive parser construction fails
Standard solution: left recursion elimination
Left recursion elimination example

\[ E \rightarrow E - E \]
\[ E \rightarrow 1 \]

We observe that \( E \Rightarrow^* 1 - 1 - \ldots - 1 \)
Idea: 1 followed by 0 or more “- 1”

\[ E \rightarrow 1 \ F \]
\[ F \rightarrow - 1 \ F \]
\[ F \rightarrow \]

This refactored grammar also eliminates the ambiguity. Yay. 😊
This grammar has a FIRST/FIRST conflict

\[
A \rightarrow a \ b \\
A \rightarrow a \ c
\]

No left recursion.
No ambiguity.
FIRST/FIRST ≠ ambiguity

This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.
This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.