LL and LR parsing as making stack machines (more) deterministic

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

October 20, 2015

Contents

Should a naturalist who had never studied the elephant except by means of the microscope think himself sufficiently acquainted with that animal?

Henri Poincaré

Why compilers?

Compilers are one of the great success stories of Computer Science:
clear-cut problems
$\Rightarrow$ theory that analyses the problem and its solutions
$\Rightarrow$ efficient and correct implementations and tools

You get a much deeper understanding of programming languages:
C $\rightarrow$ stack frames + heap + pointers
Object-orientation $\rightarrow$ C + this pointer + vtable
OCaml $\rightarrow$ call-by-value λ-calculus + stack + closures
Prolog $\rightarrow$ Warren machine: stack for backtracking + unification

These lectures - aims and style

- This material is implicit in what you can find in any comprehensive compilers textbook
- Dragon Book =
  Aho, Lam, Sethi, Ullman
  Compilers: Principles, Techniques, and Tools
- Hopcroft, John E.; Ullman, Jeffrey D. (1979). Introduction to Automata Theory, Languages, and Computation (1st ed.).
- The way I present it is using abstract machines
- This is much closer to current research in programming languages
- Including Theory group at Birmingham
- If you understand the LL machine, it will be easier to understand the CEK machine later
- Syntax and semantics are really not that different

Why parsing?

We need to sort out the syntax before dealing with semantics (meaning) of programs.

Compare

```
if (bad())
    goto fail;
```

and

```
if (bad());
    goto fail;
```

or

```
if (bad())
    goto fail;
    goto fail;
```

See https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/

Parser and compiler

The parser turns an input file into a tree data structure.
The rest of the compiler works by processing this tree
The idea has various names in different communities:

- tree walking
- syntax-directed translation
- compositional semantics

Parsing is one of the success stories of computer science.
Clear-cut problem; clean theory $\Rightarrow$ practical tools for efficient parsers

Why abstract machines?

- Long successful history, starting from Peter Landin's 1964 paper "The mechanical evaluation of expressions" I SECD machine
- Caml originally came from the Categorical Abstract Machine (CAM)
- Caml compiler based on ZINC machine, itself inspired by the Krivine Abstract Machine
- We will use the CEK machine later
- LLVM originally stood for "low level virtual machine"

- abstract machines are like functional programming: transition relation defined by pattern matching
- abstract machine are like imperative programming: step by step state change; can often be implemented using pointers

Exercise and motivation: Dyck(2) language

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.

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Grammars: formal definition

A context-free grammar consists of

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

$$A \to X_1 \dots X_n$$

where $n \geq 0$, A is a non-terminal, and the X_i are symbols.

Notation: Greek letters

Mathematicians and computer scientists are inordinately fond of Greek letters.

α	alpha
β	beta
γ	gamma
ε	epsilon
σ	sigma

Notational conventions for grammars

- We use Greek letters α, β, γ, σ ..., to stand for strings of symbols that may contain both terminals and non-terminals.
- In particular, ε is used for the empty string (of length 0).
- We write A, B, ... for non-terminals.
- We write S for the start symbol.
- Terminal symbols are usually written as lower case letters a, b, c, ...
- w and v are used for strings of terminal symbols
- X, Y, Z are used for grammar symbols that may be terminal or nonterminal
- These conventions are handy once you get used to them and are found in most books, e.g. the "Dragon Book"

Derivations

- If $A \to \alpha$ is a rule, we can replace A by α for any strings β and γ 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:

$$
\begin{array}{lcl}
D & \to & [\ D\ ]\ D \\
D & \to & (\ D\ )\ D \\
D & \to &
\end{array}
$$

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

$$D$$

An example derivation

Consider this grammar:

$$
\begin{array}{rcl}
D & \to & [\ D\ ]\ D \\
D & \to & (\ D\ )\ D \\
D & \to &
\end{array}
$$

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

$$
\begin{array}{rl}
 & D \\
\Rightarrow & [\ D\ ]\ D
\end{array}
$$

An example derivation

Consider this grammar:

$$
\begin{array}{lcl}
D & \rightarrow & [\ D\ ]\ D \\
D & \rightarrow & (\ D\ )\ D \\
D & \rightarrow &
\end{array}
$$

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

$$
\begin{array}{ll}
 & D \\
\Rightarrow & [\ D\ ]\ D \\
\Rightarrow & [\ ]\ D
\end{array}
$$

An example derivation

Consider this grammar:

$$
\begin{aligned}
D &\rightarrow [\ D\ ]\ D \\
D &\rightarrow (\ D\ )\ D \\
D &\rightarrow
\end{aligned}
$$

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

$$
\begin{aligned}
& D \\
\Rightarrow\ & [\ D\ ]\ D \\
\Rightarrow\ & [\ ]\ D \\
\Rightarrow\ & [\ ]\ [\ D\ ]\ D
\end{aligned}
$$

An example derivation

Consider this grammar:

$$
\begin{array}{lcl}
D & \rightarrow & [\, D\,]\, D \\
D & \rightarrow & (\, D\,)\, D \\
D & \rightarrow &
\end{array}
$$

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

$$
\begin{array}{ll}
 & D \\
\Rightarrow & [\, D\,]\, D \\
\Rightarrow & [\,]\, D \\
\Rightarrow & [\,]\, [\, D\,]\, D \\
\Rightarrow & [\,]\, [\,]\, D
\end{array}
$$

An example derivation

Consider this grammar:

$$\begin{array}{rcl} D & \rightarrow & [\ D\ ]\ D \\ D & \rightarrow & (\ D\ )\ D \\ D & \rightarrow & \end{array}$$

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

$$\begin{array}{rl} & D \\ \Rightarrow & [\ D\ ]\ D \\ \Rightarrow & [\ ]\ D \\ \Rightarrow & [\ ]\ [\ D\ ]\ D \\ \Rightarrow & [\ ]\ [\ ]\ D \\ \Rightarrow & [\ ]\ [\ ] \end{array}$$

Dyck language $\rightarrow$ stacks

How can we parse the Dyck language using a stack?

Dyck language $\rightarrow$ stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:

Dyck language $\rightarrow$ stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack

Dyck language → stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:

Dyck language $\to$ stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
   remove] from the input, pop] off the stack, carry on

Dyck language → stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
   remove] from the input, pop] off the stack, carry on
   (b) If a different symbol is on the top of the stack:

Dyck language $\rightarrow$ stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
   remove] from the input, pop] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
   stop and reject the input

Dyck language $\rightarrow$ stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
       remove] from the input, pop] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
       stop and reject the input
3. If the input is empty:

Dyck language → stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
   remove] from the input, pop] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
   stop and reject the input
3. If the input is empty:
   (a) If the stack is also empty: accept the input

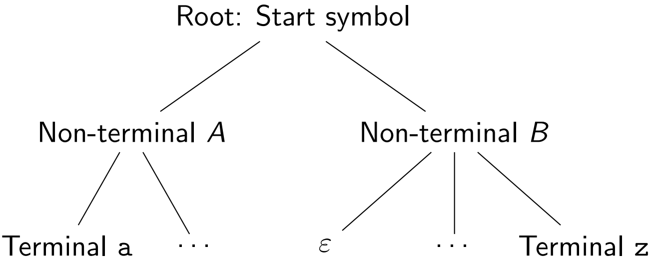
Dyck language → stacks

How can we parse the Dyck language using a stack?

1. If we see a [in the input:
   Remove [from the input, push the corresponding] onto the stack
2. If we see a] in the input:
   (a) If the same] is on the top of the stack:
   remove] from the input, pop] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
   stop and reject the input
3. If the input is empty:
   (a) If the stack is also empty: accept the input
   (b) If the stack is not empty: reject the input

Parse trees abstractly

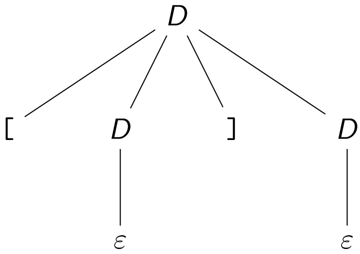
The internal nodes are labelled with nonterminals.
If there is a rule $A \to X_1 \ldots X_n$, then an internal node can have the label A and children $X_1, \ldots, X_n$.
The root node of the whole tree is labelled with the start symbol.
The leaf nodes are labelled with terminal symbols or ε.



Example: parse trees

$$
\begin{aligned}
D &\rightarrow [\, D \,]\, D \\
D &\rightarrow (\, D \,)\, D \\
D &\rightarrow
\end{aligned}
$$

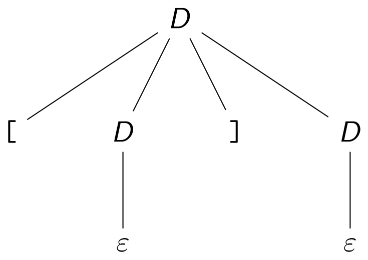
Here is a parse tree for the string `[ ]`:



Parse trees and derivations

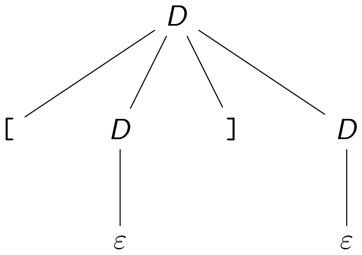
- Parse trees and derivations are related, but not the same.
- Intuition: Parse trees are extended in space (data structure), derivations in time.
- For each derivation of a word, there is a parse tree for the word.
  (Idea: each step using $A \to \alpha$ tells us that the children of some A-labelled node are labelled with the symbols in α.)
- For each parse tree, there is a (unique leftmost) derivation.
  (Idea: walk over the tree in depth-first order; each internal node gives us a rule.)

Parse tree traversal and derivation



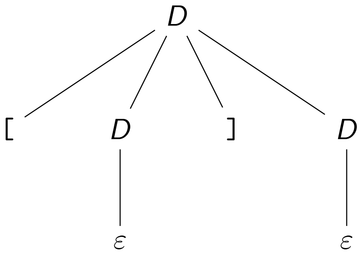
D

Parse tree traversal and derivation



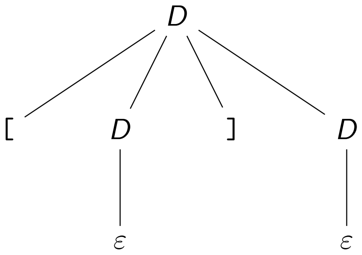
$$
\begin{aligned}
& {\color{blue}D} \\
\Rightarrow \quad & [\ {\color{blue}D}\ ]\ D
\end{aligned}
$$

Parse tree traversal and derivation



$$
\begin{array}{rl}
 & D \\
\Rightarrow & [\ D\ ]\ D \\
\Rightarrow & [\ ]\ D
\end{array}
$$

Parse tree traversal and derivation



$$
\begin{aligned}
& D \\
\Rightarrow\ & [\ D\ ]\ D \\
\Rightarrow\ & [\ ]\ D \\
\Rightarrow\ & [\ ]
\end{aligned}
$$

Definition of a parser

Suppose a grammar is given.
A parser for that grammar is a program such that for any input string w:

- If the string w is in the language of the grammar, the parser finds a derivation of the string. In our case, it will always be a leftmost or a rightmost derivation.
- If string w is **not** in the language of the grammar, the parser must reject it, for example by raising a parsing exception.
- The parser always terminates.

Abstract characterization of parsing as an inverse

Printing then parsing: same tree.

$$\begin{tikzcd}
\texttt{Tree} \arrow[rr, "\texttt{toString}"] \arrow[drr, "="'] & & \texttt{String} \arrow[d, "\texttt{parse}"] \\
& & \texttt{Tree} \cup \texttt{SyntaxError}
\end{tikzcd}$$

Parsing then printing: almost the same string.

$$\begin{tikzcd}
\texttt{String} \arrow[r, "\texttt{parse}"] \arrow[dr, "\texttt{prettyPrint}"'] & \texttt{Tree} \cup \texttt{SyntaxError} \arrow[d, "\texttt{toString}"] \\
& \texttt{String}
\end{tikzcd}$$

Push-down automata vs LL and LR machines

Much of the classic parsing theory uses push-down-automata (PDAs)

Difference to our LL and LR machines:

The LL and LR machines can push and pop several grammar symbols in one step; PDAs can push or pop only one symbol at a time

LL and LR machines could be translated to PDAs

The LL and LR machines only have a stack and input; PDAs also have a "state" component

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.
Classic automata theory:

$$\begin{aligned} & \text{regular expression} \\ \to\ & \text{nondeterministic finite automaton (NFA)} \\ \to\ & \text{deterministic finite automaton (DFA)} \end{aligned}$$

Lexers can be automagically generated (just like parsers by parser generators.
Example: lex tool in Unix and variants

Grammars can encode regular expressions

The alternative $\alpha \mid \beta$ can be expressed as follows:

Grammars can encode regular expressions

The alternative $\alpha \mid \beta$ can be expressed as follows:

$$\begin{array}{rcl} A & \to & \alpha \\ A & \to & \beta \end{array}$$

Grammars can encode regular expressions

The alternative $\alpha \mid \beta$ can be expressed as follows:

$$\begin{array}{rcl} A & \rightarrow & \alpha \\ A & \rightarrow & \beta \end{array}$$

Repetition α^* can be expressed as follows:

$$\begin{array}{rcl} A & \rightarrow & \alpha\, A \\ A & \rightarrow & \end{array}$$

Grammars can encode regular expressions

The alternative $\alpha \mid \beta$ can be expressed as follows:

$$
\begin{aligned}
A &\rightarrow \alpha \\
A &\rightarrow \beta
\end{aligned}
$$

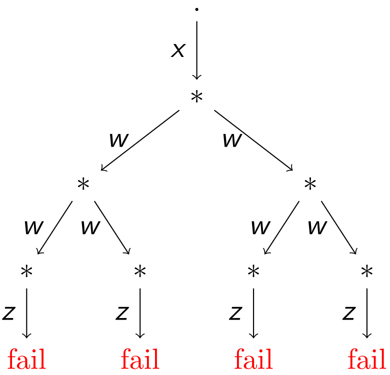
Repetition α^* can be expressed as follows:

$$
\begin{aligned}
A &\rightarrow \alpha\, A \\
A &\rightarrow
\end{aligned}
$$

Hence we can use $\mid$ and $*$ in grammars;
sometimes called BNF, for Backus-Naur-form.
Reg exps are still useful, as they are simple and efficient (in grep and lex)

Exponential blowup due to backtracking ☹

Note: regular expression matchers in Java, Perl etc use an NFA and are vulnerable to exponential blowup denial of service attacks; see `http://www.cs.bham.ac.uk/~hxt/research/rxxr2`



We used nondeterministic abstract machines for reg exp matching.

Determinism vs nondeterminism

The more powerful machines become, the greater the gap between determinism and nondeterminism

Finite automata : can always construct DFA from NFA

Pushdown automata : not always possible to make stack machine deterministic
But we get deterministic machines in practice

Polynomial time Turing machines : P vs NP

Parsing and (non-)deterministic stack machines

We decompose the parsing problem into to 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

Parsing stack machines

The states of the machines are of the form

$$\langle \sigma \ , \ w \rangle$$

where

- σ 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

$$\underbrace{\langle \sigma_1 \ , \ w_1 \rangle}_{\text{old state}} \longrightarrow \underbrace{\langle \sigma_2 \ , \ w_2 \rangle}_{\text{new state}}$$

- pushing or popping the stack changes σ_1 to σ_2
- consuming input changes w_1 to w_2

LL and LR parsing terminology

The first L in LL and LR means that the input is read from the left.
The second letter refers to what the parser does:

LL	machine run $\cong$ leftmost derivation
LR	machine run $\cong$ rightmost derivation in reverse

Moreover,

LL(1)	means LL with one symbol of lookahead
LL(k)	means LL with k symbols of lookahead
LL(*)	means LL(k) for a large enough k
LR(0)	means LR with zero symbols of lookahead
LR(1)	means LR with one symbol of lookahead
LALR(1)	is a variant of LR(1) that uses less memory
LR(k)	for $k > 1$ is not needed because LR(1) is already powerful enough

Parser generators try to increase determinism

- For each grammar, we can construct an LL and an LR machine for that grammar
- The machines are partially correct by construction
- The machines are non-deterministic, so we cannot implement them efficiently.
- There are constructions (FIRST/FOLLOW and LR items) that make the machines more deterministic
- Parser generators perform these constructions automatically
- If the construction succeeds, we get an efficient parser
- If not, we get errors from the parser generator
- The errors mean there was still some nondeterminism that could not be eliminated

LL vs LR idea intuitively

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?

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.

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

$$\langle \sigma_1 , w_1 \rangle \longrightarrow \langle \sigma_2 , w_2 \rangle$$

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.

Non-deterministic machines

Non-deterministic does not mean that the machine flips a coin or uses a random number generator.

It means some of the details of what the machine does are not known to us.

Compare malloc in C. It gives you some pointer to newly allocated memory. What matters is that the memory is newly allocated. Do you care whether the pointer value is 68377378 or 37468562?

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

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.

$$\langle A\,\sigma\,,\,w\rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma\,,\,w\rangle \qquad \text{if there is a rule } A \to \alpha$$

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

$\langle S\,,\,w\rangle$ is the initial state for input w

$\langle \varepsilon\,,\,\varepsilon\rangle$ is the accepting state

Note: $A\,\sigma$ means A consed onto σ, whereas $\alpha\,\sigma$ means α concatenated with σ.
Compare OCaml: A::σ versus α@σ.

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:

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

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
Exercise: prove this in Agda

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

LL machine run example: list idiom

$$\begin{array}{rcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}$$

$$\langle S \ , \ a\,a\,b \rangle$$

LL machine run example: list idiom

$$
\begin{aligned}
S &\to L\, b \\
L &\to a\, L \\
L &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S \,,\, a\, a\, b \rangle \\
\xrightarrow{\text{predict}} \; & \langle L\, b \,,\, a\, a\, b \rangle
\end{aligned}
$$

LL machine run example: list idiom

$$
\begin{aligned}
S &\rightarrow L\,b \\
L &\rightarrow a\,L \\
L &\rightarrow \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}}\ & \langle L\,b\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}}\ & \langle a\,L\,b\ ,\ a\,a\,b \rangle
\end{aligned}
$$

LL machine run example: list idiom

$$
\begin{aligned}
S &\to L\, b \\
L &\to a\, L \\
L &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S \,,\, a\, a\, b \rangle \\
\xrightarrow{\text{predict}} \;& \langle L\, b \,,\, a\, a\, b \rangle \\
\xrightarrow{\text{predict}} \;& \langle a\, L\, b \,,\, a\, a\, b \rangle \\
\xrightarrow{\text{match}} \;& \langle L\, b \,,\, a\, b \rangle
\end{aligned}
$$

LL machine run example: list idiom

$$
\begin{aligned}
S &\to L\, b \\
L &\to a\, L \\
L &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S\ ,\ a\, a\, b \rangle \\
\xrightarrow{\text{predict}}\ & \langle L\, b\ ,\ a\, a\, b \rangle \\
\xrightarrow{\text{predict}}\ & \langle a\, L\, b\ ,\ a\, a\, b \rangle \\
\xrightarrow{\text{match}}\ & \langle L\, b\ ,\ a\, b \rangle \\
\xrightarrow{\text{predict}}\ & \langle a\, L\, b\ ,\ a\, b \rangle
\end{aligned}
$$

LL machine run example: list idiom

$$
\begin{array}{rcl}
S & \to & L\,b \\
L & \to & a\,L \\
L & \to & \varepsilon
\end{array}
$$

$$
\begin{array}{rl}
 & \langle S\,,\,a\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\,,\,a\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,a\,b\rangle \\
\xrightarrow{\text{match}} & \langle L\,b\,,\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,b\rangle \\
\xrightarrow{\text{match}} & \langle L\,b\,,\,b\rangle
\end{array}
$$

LL machine run example: list idiom

$$
\begin{aligned}
S &\to L\, b \\
L &\to a\, L \\
L &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S\,,\, a\,a\,b \rangle \\
\xrightarrow{\text{predict}}\; & \langle L\,b\,,\, a\,a\,b \rangle \\
\xrightarrow{\text{predict}}\; & \langle a\,L\,b\,,\, a\,a\,b \rangle \\
\xrightarrow{\text{match}}\; & \langle L\,b\,,\, a\,b \rangle \\
\xrightarrow{\text{predict}}\; & \langle a\,L\,b\,,\, a\,b \rangle \\
\xrightarrow{\text{match}}\; & \langle L\,b\,,\, b \rangle \\
\xrightarrow{\text{predict}}\; & \langle b\,,\, b \rangle
\end{aligned}
$$

LL machine run example: list idiom

$$
\begin{array}{rcl}
S & \to & L\,b \\
L & \to & a\,L \\
L & \to & \varepsilon
\end{array}
$$

$$
\begin{array}{rl}
 & \langle S\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,b \rangle \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ b \rangle \\
\xrightarrow{\text{predict}} & \langle b\ ,\ b \rangle \\
\xrightarrow{\text{match}} & \langle \varepsilon\ ,\ \varepsilon \rangle
\end{array}
$$

LL machine run example: list idiom

$$
\begin{array}{rcl}
S & \to & L\,b \\
L & \to & a\,L \\
L & \to & \varepsilon
\end{array}
$$

$$
\begin{array}{rl}
 & \langle S\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,b \rangle \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ b \rangle \\
\xrightarrow{\text{predict}} & \langle b\ ,\ b \rangle \\
\xrightarrow{\text{match}} & \langle \varepsilon\ ,\ \varepsilon \rangle \quad ☺
\end{array}
$$

Correct input accepted.

LL machine run example 2

$$
\begin{aligned}
S &\rightarrow L\,b \\
L &\rightarrow a\,L \\
L &\rightarrow \varepsilon
\end{aligned}
$$

$$\langle S\ ,\ b\,a \rangle$$

LL machine run example 2

$$\begin{aligned} S &\rightarrow L\,b \\ L &\rightarrow a\,L \\ L &\rightarrow \varepsilon \end{aligned}$$

$$\begin{aligned} & \langle S\ ,\ b\,a \rangle \\ \xrightarrow{\text{predict}}\ & \langle L\,b\ ,\ b\,a \rangle \end{aligned}$$

LL machine run example 2

$$
\begin{aligned}
S &\rightarrow L\, b \\
L &\rightarrow a\, L \\
L &\rightarrow \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S \,,\, b\, a \rangle \\
\xrightarrow{\text{predict}} \; & \langle L\, b \,,\, b\, a \rangle \\
\xrightarrow{\text{predict}} \; & \langle b \,,\, b\, a \rangle
\end{aligned}
$$

LL machine run example 2

$$
\begin{array}{rcl}
S & \to & L\,b \\
L & \to & a\,L \\
L & \to & \varepsilon
\end{array}
$$

$$
\begin{array}{rl}
 & \langle S \;,\; b\,a \rangle \\
\xrightarrow{\text{predict}} & \langle L\,b \;,\; b\,a \rangle \\
\xrightarrow{\text{predict}} & \langle b \;,\; b\,a \rangle \\
\xrightarrow{\text{match}} & \langle \varepsilon \;,\; a \rangle
\end{array}
$$

LL machine run example 2

$$S \to L\,b$$
$$L \to a\,L$$
$$L \to \varepsilon$$

$$\langle S\,,\,b\,a\rangle \xrightarrow{\text{predict}} \langle L\,b\,,\,b\,a\rangle \xrightarrow{\text{predict}} \langle b\,,\,b\,a\rangle \xrightarrow{\text{match}} \langle \varepsilon\,,\,a\rangle \quad ☺$$

Incorrect input should not be accepted. The machine is not to blame for it.

LL machine run example: what should not happen

$$
\begin{aligned}
S &\to L\,b \\
L &\to a\,L \\
L &\to \varepsilon
\end{aligned}
$$

$$\langle S \, , \, a\,a\,b \rangle$$

LL machine run example: what should not happen

$$
\begin{aligned}
S &\rightarrow L\, b \\
L &\rightarrow a\, L \\
L &\rightarrow \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
& \langle S \,,\, a\, a\, b \rangle \\
\xrightarrow{\text{predict}} \; & \langle L\, b \,,\, a\, a\, b \rangle
\end{aligned}
$$

LL machine run example: what should not happen

$$\begin{array}{lcl} S & \to & L\, b \\ L & \to & a\, L \\ L & \to & \varepsilon \end{array}$$

$$\begin{array}{rl} & \langle\, S\ ,\ a\, a\, b \,\rangle \\ \xrightarrow{\text{predict}} & \langle\, L\, b\ ,\ a\, a\, b \,\rangle \\ \xrightarrow{\text{predict}} & \langle\, b\ ,\ a\, a\, b \,\rangle \quad ☹ \end{array}$$

When it makes bad nondeterministic choices, the LL machine gets stuck even on correct input.

LL Exercise

Describe in general what the stuck states of the LL machine are, in which the machine can make no further steps, but cannot accept the input.
Hint: there are 3 cases, plus another one if the grammar is silly.

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LL machine example: Dyck language

Consider this grammar

$$
\begin{array}{lcl}
D & \to & [\, D \,]\, D \\
D & \to & (\, D \,)\, D \\
D & \to &
\end{array}
$$

Show how the LL machine for this grammar can accept the input

`[ ( ) ]`

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LL revision exercise

You will need to know the LL machine.
Memorizing it by brute force is not recommended.
Instead, remember what the machine does and why.
Try to write down the LL using these points as a guideline:
The parsing stack is a prediction of future input.
The predict moves refine the prediction at the top of the stack
The match moves check that the prediction is true

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LL(1) and FIRST/FOLLOW motivating example

Consider the LL machine for the following grammar (which may be part of some larger grammar):

$$
\begin{array}{lcl}
E & \to & A\,B \\
E & \to & f \\
A & \to & c \\
A & \to & \varepsilon \\
B & \to & d
\end{array}
$$

Machine state:

$$\langle\, E\,\sigma \;,\; d\,w \,\rangle \longrightarrow \ldots$$

What should the next 4 steps be, and why?

We need FIRST for AB and FOLLOW for the ε rule for A.

Making the LL machine deterministic using lookahead

- Deterministic refinement of nondeterministic LL machine $\Rightarrow$ LL(1) machine.
- We use one symbol of lookahead to guide the predict moves, to avoid predict moves that get the machine stuck soon after
- 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 $FIRST(\alpha)$ if there exist a β such that
  $$\alpha \stackrel{*}{\Rightarrow} b\,\beta$$
  that is, b is the first symbol in something derivable from α
- A terminal symbol b is in $FOLLOW(X)$ if there exist α and β such that
  $$S \stackrel{*}{\Rightarrow} \alpha\,X\,b\,\beta$$
  that is, b follows X in some derivation
- α is nullable if
  $$\alpha \stackrel{*}{\Rightarrow} \varepsilon$$
  that is, we can derive the empty string from it

FIRST and FOLLOW examples

Consider

$$\begin{array}{rcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}$$

Then

$$a \in FIRST(L)$$

$$b \in FOLLOW(L)$$

L is nullable

Exercise on FIRST and FOLLOW

What is FIRST(a)?
What is FIRST(aB)?
What is FIRST(ε)?
What is FIRST(AB) written in terms of FIRST(A) and FIRST(B), if A is nullable?

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LL(1) machine: LL with 1 symbol of lookahead

The LL1(1) machine is like the LL machine with additional conditions.

$$\langle A\,\sigma \;,\; b\,w \rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma \;,\; b\,w \rangle \qquad \text{if there is a rule } A \to \alpha$$
$$\text{and } b \in \mathit{FIRST}(\alpha)$$

$$\langle A\,\sigma \;,\; b\,w \rangle \xrightarrow{\text{predict}} \langle \beta\,\sigma \;,\; b\,w \rangle \qquad \text{if there is a rule } A \to \beta$$
$$\text{and } \beta \text{ is nullable}$$
$$\text{and } b \in \mathit{FOLLOW}(A)$$

$$\langle a\,\sigma \;,\; a\,w \rangle \xrightarrow{\text{match}} \langle \sigma \;,\; w \rangle$$

$\langle S \;,\; w \rangle$ is the initial state for input w

$\langle \varepsilon \;,\; \varepsilon \rangle$ is the accepting state

ε-rules and FOLLOW

Consider this rule:

$$\langle A\,\sigma\ ,\ b\,w\rangle \xrightarrow{\text{predict}} \langle \beta\,\sigma\ ,\ b\,w\rangle \qquad \text{if there is a rule } A \to \beta$$

$$\text{and } \beta \text{ is nullable}$$

$$\text{and } b \in \mathit{FOLLOW}(A)$$

This is a common special case for $\beta = \varepsilon$:

$$\langle A\,\sigma\ ,\ b\,w\rangle \xrightarrow{\text{predict}} \langle \sigma\ ,\ b\,w\rangle \qquad \text{if there is a rule } A \to \varepsilon$$

$$\text{and } b \in \mathit{FOLLOW}(A)$$

In English: the machine can delete A when it sees a symbol b in the lookahead that can follow A.

Parsing errors as stuck states in the LL(1) machine

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".
If it reaches a state of the form

$$\langle\, \sigma \;,\; \varepsilon \,\rangle$$

where $\sigma \neq \varepsilon$, it can report premature end of input.
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

$$\boxed{\begin{array}{lcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}$$

$$\langle S\;,\; a\,a\,b \rangle$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\boxed{\begin{array}{lcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}$$

$$\begin{array}{ll} & \langle\, S\ ,\ a\,a\,b \,\rangle \\ \xrightarrow{\text{predict}} & \langle\, L\,b\ ,\ a\,a\,b \,\rangle \ \text{as } a \in \mathit{FIRST}(L\,b) \end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\begin{array}{|ccc|} \hline & & \\ S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \\ \hline \end{array}$$

$$\begin{array}{ll} & \langle S\ ,\ a\,a\,b \rangle \\ \xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b \rangle \text{ as } a \in FIRST(L\,b) \\ \xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b \rangle \text{ as } a \in FIRST(a\,L) \end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\boxed{\begin{array}{rcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}$$

$$\begin{array}{rl}
& \langle\, S \,,\, a\,a\,b \,\rangle \\
\xrightarrow{\text{predict}} & \langle\, L\,b \,,\, a\,a\,b \,\rangle \ \text{as } a \in \mathit{FIRST}(L\,b) \\
\xrightarrow{\text{predict}} & \langle\, a\,L\,b \,,\, a\,a\,b \,\rangle \ \text{as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle\, L\,b \,,\, a\,b \,\rangle
\end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$
\boxed{\begin{array}{ccc} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}
$$

$$
\begin{array}{ll}
 & \langle S\,,\,a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\,,\,a\,a\,b \rangle \text{ as } a \in \mathit{FIRST}(L\,b) \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,a\,b \rangle \text{ as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\,,\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,b \rangle \text{ as } a \in \mathit{FIRST}(a\,L)
\end{array}
$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\boxed{\begin{array}{lcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}$$

$$\begin{array}{ll}
 & \langle S\,,\,a\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\,,\,a\,a\,b\rangle \text{ as } a \in \mathit{FIRST}(L\,b) \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,a\,b\rangle \text{ as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\,,\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\,,\,a\,b\rangle \text{ as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\,,\,b\rangle
\end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\begin{array}{|lcl|} \hline & & \\ S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \\ \hline \end{array}$$

$$\begin{array}{rll}
 & \langle S\ ,\ a\,a\,b \rangle & \\
\xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b \rangle & \text{as } a \in FIRST(L\,b) \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b \rangle & \text{as } a \in FIRST(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ a\,b \rangle & \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,b \rangle & \text{as } a \in FIRST(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ b \rangle & \\
\xrightarrow{\text{predict}} & \langle b\ ,\ b \rangle & \text{as } b \in FOLLOW(L)
\end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\begin{array}{|ccc|} & & \\ S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \\ \end{array}$$

$$\begin{array}{ll}
 & \langle S\ ,\ a\,a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b \rangle \text{ as } a \in FIRST(L\,b) \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b \rangle \text{ as } a \in FIRST(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ a\,b \rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,b \rangle \text{ as } a \in FIRST(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ b \rangle \\
\xrightarrow{\text{predict}} & \langle b\ ,\ b \rangle \text{ as } b \in FOLLOW(L) \\
\xrightarrow{\text{match}} & \langle \varepsilon\ ,\ \varepsilon \rangle
\end{array}$$

LL(1) machine run example

Just like LL machine, but now deterministic

$$\boxed{\begin{array}{lcl} S & \to & L\,b \\ L & \to & a\,L \\ L & \to & \varepsilon \end{array}}$$

$$\begin{array}{rl}
 & \langle S\ ,\ a\,a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle L\,b\ ,\ a\,a\,b\rangle \text{ as } a \in \mathit{FIRST}(L\,b) \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,a\,b\rangle \text{ as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ a\,b\rangle \\
\xrightarrow{\text{predict}} & \langle a\,L\,b\ ,\ a\,b\rangle \text{ as } a \in \mathit{FIRST}(a\,L) \\
\xrightarrow{\text{match}} & \langle L\,b\ ,\ b\rangle \\
\xrightarrow{\text{predict}} & \langle b\ ,\ b\rangle \text{ as } b \in \mathit{FOLLOW}(L) \\
\xrightarrow{\text{match}} & \langle \varepsilon\ ,\ \varepsilon\rangle \ ☺
\end{array}$$

Is the LL(1) machine always deterministic?

$$\langle A\,\sigma\ ,\ b\,w\rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma\ ,\ b\,w\rangle \qquad \text{if there is a rule } A \to \alpha$$
$$\text{and } b \in \mathit{FIRST}(\alpha)$$

$$\langle A\,\sigma\ ,\ b\,w\rangle \xrightarrow{\text{predict}} \langle \beta\,\sigma\ ,\ b\,w\rangle \qquad \text{if there is a rule } A \to \beta$$
$$\text{and } \beta \text{ is nullable}$$
$$\text{and } b \in \mathit{FOLLOW}(A)$$

For some grammars, there may be:

- FIRST/FIRST conflicts ☹
- FIRST/FOLLOW conflicts ☹

FIRST/FIRST conflict $\Rightarrow$ nondeterminism ☹

$$\langle A\,\sigma\ ,\ b\,w\rangle \quad \xrightarrow{\text{predict}} \quad \langle \alpha\,\sigma\ ,\ b\,w\rangle \qquad \text{if there is a rule } A \to \alpha$$
$$\text{and } b \in \mathit{FIRST}(\alpha)$$

FIRST/FIRST conflicts:
There exist

- terminal symbol b
- grammar rule $A \to \alpha_1$ with $b \in \mathit{FIRST}(\alpha_1)$
- grammar rule $A \to \alpha_2$ with $b \in \mathit{FIRST}(\alpha_2)$ and $\alpha_1 \neq \alpha_2$

If A is on the top of the stack and b in the lookahead, the LL(1) machine can do two different steps.

FIRST/FOLLOW conflict $\Rightarrow$ nondeterminism ☹

$$\langle A\,\sigma\;,\;b\,w\rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma\;,\;b\,w\rangle \qquad \text{if there is a rule } A \to \alpha$$
$$\text{and } b \in \mathit{FIRST}(\alpha)$$

$$\langle A\,\sigma\;,\;b\,w\rangle \xrightarrow{\text{predict}} \langle \beta\,\sigma\;,\;b\,w\rangle \qquad \text{if there is a rule } A \to \beta$$
$$\text{and } \beta \text{ is nullable}$$
$$\text{and } b \in \mathit{FOLLOW}(A)$$

FIRST/FOLLOW conflicts:
There exist

- terminal symbol b
- grammar rule $A \to \alpha$ with $b \in \mathit{FIRST}(\alpha)$
- grammar rule $A \to \beta$ where β is nullable and $b \in \mathit{FOLLOW}(A)$ and $\alpha \neq \beta$

If A is on the top of the stack and b in the lookahead, the LL(1) machine can do two different steps.

LL(1) construction may fail

- If there are FIRST/FIRST or FIRST/FOLLOW conflicts, the LL(1) machine is not deterministic
- The LL(1) parser construction methods has failed to give a deterministic parser
- A parser generator may produce error messages when given such a grammar
- Note: this is different from parse errors due to inputs that are not in the language of the grammar
- If there are no conflicts, the grammar is LL(1) and we get a deterministic, linear-time parser

FIRST/FIRST $\neq$ ambiguity

NB: FIRST/FIRST conflict do not mean that the grammar is ambiguous.
Ambiguous means different parse trees for the same string.
See `https://www.youtube.com/watch?v=ldT2g2qDQNQ`
This grammar has a FIRST/FIRST conflict

$$
\begin{array}{lcl}
A & \rightarrow & ab \\
A & \rightarrow & ac
\end{array}
$$

But parse trees are unique.

Lookahead and programming language design

Many constructs start with a keyword telling us immediately what it is.
Keywords "`if`", "`while`", produce tokens that tell the parser to expect a conditional, a loop, etc
$\Rightarrow$ these symbols are typically in FIRST
Many constructs end with a terminator like ";"
Such tokens tell the parser to stop reading an expression, statement etc
$\Rightarrow$ these symbols are typically in FOLLOW
Now you know why there are some many semicolons in CS. ☺
Opening brackets are often in FIRST
Closing brackets are often in FOLLOW

Stretch exercise

In C, the symbol $*$ can be both
a binary operator (for multiplication) and
a unary operator (for pointer dereferencing).
This double syntactic use of star is confusing to many students.
Explain whether it is a problem for LL(1) parsing or not.

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Stretch exercise

Does the LL(1) machine always terminate?
Are there reasonable conditions on grammars that cause the LL(1) machine to terminate?

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Left factoring

This grammar has a FIRST/FIRST conflict ☹

$$\begin{array}{rcl} A & \rightarrow & ab \\ A & \rightarrow & ac \end{array}$$

Left factorize as follows:

$$\begin{array}{rcl} A & \rightarrow & a\,B \\ B & \rightarrow & b \\ B & \rightarrow & c \end{array}$$

No conflict ☺
LL(1) machine can postpone its decision until after the a is read.

Computing FIRST and FOLLOW depending on nullable

Consider a grammar rule of the form

$$A \to \underbrace{Y_1 \dots Y_{i-1}}_{\text{nullable?}}\ Y_i\ \underbrace{Y_{i+1} \dots Y_{j-1}}_{\text{nullable?}}\ Y_j\ \underbrace{Y_{j+1} \dots Y_k}_{\text{nullable?}}$$

For any of the nullable cases above, what follows for FIRST(A), FIRST(Y$_i$), FOLLOW(A), and FOLLOW(Y$_j$)?
$\subseteq$ is easy to see; but be careful which way it goes

Computing FIRST and FOLLOW iteratively

for each symbol X, `nullable`$[X]$ is initialised to `false`
for each symbol X, `follow`$[X]$ is initialised to the empty set
for each terminal symbol a, `first`$[a]$ is initialised to $\{a\}$
for each non-terminal symbol A, `first`$[A]$ is initialised to the empty set
repeat
  for each grammar rule $A \to Y_1 \dots Y_k$
    if all the Y_i are nullable
      then set `nullable`$[A]$ to `true`
  for each i from 1 to k, and j from $i+1$ to k
    if $Y_1, \dots, Y_{i-1}$ are all nullable
      then add all symbols in `first`$[Y_i]$ to `first`$[A]$
    if $Y_{i+1}, \dots, Y_{j-1}$ are all nullable
      then add all symbols in `first`$[Y_j]$ to `follow`$[Y_i]$
    if $Y_{j+1}, \dots, Y_k$ are all nullable
      then add all symbols in `follow`$[A]$ to `follow`$[Y_j]$
until `first`, `follow` and `nullable` did not change in this iteration

LL(1) machine exercise

Consider the grammar

$$
\begin{array}{lcl}
D & \to & [\, D \,]\, D \\
D & \to & (\, D \,)\, D \\
D & \to &
\end{array}
$$

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.

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.
4. A non-terminal symbol occurs multiple times on the right-hand side of one of its rules.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.
4. A non-terminal symbol occurs multiple times on the right-hand side of one of its rules.
5. The same symbol occurs on the right-hand side of two rules.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.
4. A non-terminal symbol occurs multiple times on the right-hand side of one of its rules.
5. The same symbol occurs on the right-hand side of two rules.
6. There are no symbols on the right-hand side of some rules.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.
4. A non-terminal symbol occurs multiple times on the right-hand side of one of its rules.
5. The same symbol occurs on the right-hand side of two rules.
6. There are no symbols on the right-hand side of some rules.
7. A non-terminal symbol occurs as the first symbol on the right-hand side of one of its rules.

Quiz: which of these are a problem for LL(1), and why?

1. The same symbols occur on the right-hand side of two rules for the same non-terminal symbol.
2. The same symbol occurs as the first symbol on the right-hand side of two rules for the same non-terminal symbol.
3. A non-terminal symbol occurs on the right-hand side of one of its rules.
4. A non-terminal symbol occurs multiple times on the right-hand side of one of its rules.
5. The same symbol occurs on the right-hand side of two rules.
6. There are no symbols on the right-hand side of some rules.
7. A non-terminal symbol occurs as the first symbol on the right-hand side of one of its rules.
8. A non-terminal symbol A occurs as the first symbol of a rule for a non-terminal B, such that B occurs as the first symbol of a rule for A

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LR machine

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

$$
\begin{array}{rcll}
\langle \sigma \,,\, a\, w \rangle & \xrightarrow{\text{shift}} & \langle \sigma\, a \,,\, w \rangle & \\
\langle \sigma\, \alpha \,,\, w \rangle & \xrightarrow{\text{reduce}} & \langle \sigma\, A \,,\, w \rangle & \quad \text{if there is a rule } A \to \alpha
\end{array}
$$

$$
\begin{array}{rl}
\langle \varepsilon \,,\, w \rangle & \quad \text{is the initial state for input } w \\
\langle S \,,\, \varepsilon \rangle & \quad \text{is the accepting state}
\end{array}
$$

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:

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

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 in reverse
Stretch exercise: prove this in Agda

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

LL vs LR exercise

Consider this grammar

$$\begin{array}{lcl} S & \to & A\,B \\ A & \to & c \\ B & \to & d \end{array}$$

Show how the LL and LR machine can accept the input $c\,d$.
Compare the machine runs to a leftmost and rightmost derivation.

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

LR and right-hand-sides

Should the LR machine be "greedy" in the following sense: as soon as the right-hand-side of a rule appears on the stack, the machine does a reduce step?
Find a counterexample where this causes the LR machine to get stuck.

[**Revision exercise:** You are encouraged to attempt this exercise on you own before discussing it on Canvas or Facebook.]

Is the LR machine always deterministic?

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

$$\langle \sigma\, \alpha \, , w \rangle \xrightarrow{\text{reduce}} \langle \sigma\, A \, , w \rangle \qquad \text{if there is a rule } A \to \alpha$$

Is the LR machine always deterministic?

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

$$\langle \sigma\, \alpha \, , w \rangle \xrightarrow{\text{reduce}} \langle \sigma\, A \, , w \rangle \qquad \text{if there is a rule } A \to \alpha$$

For some grammars, there may be:

- shift/reduce conflicts ☹
- reduce/reduce conflicts ☹

LL vs LR in more detail

LL

$$\langle A\,\sigma\ ,\ w\rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma\ ,\ w\rangle \qquad \text{if there is a rule } A \to \alpha$$

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

LR

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

$$\langle \sigma\,\alpha\ ,\ w\rangle \xrightarrow{\text{reduce}} \langle \sigma\,A\ ,\ w\rangle \qquad \text{if there is a rule } A \to \alpha$$

LL vs LR in more detail

LL

$$\langle A\,\sigma \,,\, w \rangle \xrightarrow{\text{predict}} \langle \alpha\,\sigma \,,\, w \rangle \qquad \text{if there is a rule } A \to \alpha$$

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

LR

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

$$\langle \sigma\,\alpha \,,\, w \rangle \xrightarrow{\text{reduce}} \langle \sigma\,A \,,\, w \rangle \qquad \text{if there is a rule } A \to \alpha$$

The LR machines can make its reduce choices after it has seen everything derived from the right hand side of a rule.
The LL machine has less information available when making its predict choices.

LL vs LR example

Here is a simple grammar:

$$\begin{array}{rcl} S & \rightarrow & A \\ S & \rightarrow & B \\ A & \rightarrow & a\,b \\ B & \rightarrow & a\,c \end{array}$$

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

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow a\, b \\
B &\rightarrow a\, c \\
D &\rightarrow a
\end{aligned}
$$

$$\langle \varepsilon \,,\, a\, b \rangle$$

LR machine run example

$$
\begin{aligned}
S &\to A \\
S &\to B \\
A &\to a\,b \\
B &\to a\,c \\
D &\to a
\end{aligned}
$$

$$
\begin{aligned}
& \langle \varepsilon \;,\; a\,b \rangle \\
\xrightarrow{\text{shift}} \; & \langle a \;,\; b \rangle
\end{aligned}
$$

LR machine run example

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow a\,b \\
B &\rightarrow a\,c \\
D &\rightarrow a
\end{aligned}
$$

$$
\begin{aligned}
& \langle \varepsilon \,,\, a\,b \rangle \\
\xrightarrow{\text{shift}} \; & \langle a \,,\, b \rangle \\
\xrightarrow{\text{shift}} \; & \langle a\,b \,,\, \varepsilon \rangle
\end{aligned}
$$

LR machine run example

$$
\begin{aligned}
S &\to A \\
S &\to B \\
A &\to a\,b \\
B &\to a\,c \\
D &\to a
\end{aligned}
$$

$$
\begin{aligned}
& \langle \varepsilon \,,\, a\,b \rangle \\
\xrightarrow{\text{shift}}\; & \langle a \,,\, b \rangle \\
\xrightarrow{\text{shift}}\; & \langle a\,b \,,\, \varepsilon \rangle \\
\xrightarrow{\text{reduce}}\; & \langle A \,,\, \varepsilon \rangle
\end{aligned}
$$

LR machine run example

$$\begin{array}{rcl} S & \to & A \\ S & \to & B \\ A & \to & a\,b \\ B & \to & a\,c \\ D & \to & a \end{array}$$

$$\begin{array}{rl} & \langle \varepsilon\,,\,a\,b \rangle \\ \xrightarrow{\text{shift}} & \langle a\,,\,b \rangle \\ \xrightarrow{\text{shift}} & \langle a\,b\,,\,\varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,,\,\varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle S\,,\,\varepsilon \rangle \end{array}$$

LR machine run example

$$
\begin{aligned}
S &\to A \\
S &\to B \\
A &\to a\, b \\
B &\to a\, c \\
D &\to a
\end{aligned}
$$

$$
\begin{aligned}
& \langle \varepsilon \,,\, a\, b \rangle \\
\xrightarrow{\text{shift}} \quad & \langle a \,,\, b \rangle \\
\xrightarrow{\text{shift}} \quad & \langle a\, b \,,\, \varepsilon \rangle \\
\xrightarrow{\text{reduce}} \quad & \langle A \,,\, \varepsilon \rangle \\
\xrightarrow{\text{reduce}} \quad & \langle S \,,\, \varepsilon \rangle \quad ☺
\end{aligned}
$$

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.

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Making the LR machine deterministic using items

- 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

LR(0) items idea

An LR parser must recognize when and how to reduce.
It uses LR items to guide it.
Ideally, guidance by the LR items should make the machine deterministic.
An LR item is a rule together with an pointer $\bullet$ that tells the parser how much of the right-hand-side of the rule it has pushed onto its stack:

$$[A \to \alpha \bullet \beta]$$

When the bullet reaches the end in an item

$$[A \to \gamma \bullet]$$

then reduce with $A \to \gamma$.

LR(0) items

Let us assume we have a given grammar. An LR(0) item (for that grammar) is of the form:

$$[A \to \alpha \bullet \beta]$$

if there is a rule $A \to \alpha\,\beta$.
Note: α and/or β may be $= \varepsilon$.
Transition steps between items:

$$[A \to \alpha \bullet X\,\beta] \quad \xrightarrow{X} \quad [A \to \alpha\,X \bullet \beta]$$

$$[A \to \alpha \bullet B\,\beta] \quad \xrightarrow{\varepsilon} \quad [B \to \bullet\,\gamma] \text{ if there is a rule } B \to \gamma$$

Compare the ε rule to the FIRST construction for LL(1).

LR(0) automaton, made deterministic

$$[A \to \alpha \bullet X\,\beta] \quad \xrightarrow{X} \quad [A \to \alpha\, X \bullet \beta]$$

$$[A \to \alpha \bullet B\,\beta] \quad \xrightarrow{\varepsilon} \quad [B \to\ \bullet \gamma] \text{ if there is a rule } B \to \gamma$$

This is a finite automaton! But nondeterministic.
The powerset construction gives us a deterministic finite automaton (DFA) with
states = sets of LR(0) items
input alphabet: symbols of our grammar, including nonterminals

Powerset automaton for items

Idea: instead of nondeterministic transitions

$$i \xrightarrow{X} j$$

we collect all the is and js into sets s.
There is a step

$$s_1 \quad \xrightarrow{X} \quad s_2$$

if s_2 is the set of all items j such that there is an item $i \in s_1$ with

$$i \xrightarrow{X} \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} j$$

that is, we can get from i to j with an X step followed by a possibly empty sequence of ε steps.
All the items in the set proceed "in lockstep"

LR(0) machine steps: make LR machine deterministic

The stack σ now holds states s of the LR(0) DFA, which are sets of LR(0) items.
We write $s \xrightarrow{X} s'$ for steps of the LR(0) DFA.

$$\langle \sigma\, s \,,\, a\, w \rangle \quad \xrightarrow{\text{shift}} \quad \langle \sigma\, s\, s' \,,\, w \rangle \quad \text{if there is no } [B \to \gamma \bullet] \in s \text{ and } s \xrightarrow{a} s'$$

$$\langle \sigma\, s\, s_1 \,\ldots\, s_n \,,\, w \rangle \quad \xrightarrow{\text{reduce}} \quad \langle \sigma\, s\, s' \,,\, w \rangle \quad \text{if } [B \to X_1 \ldots X_n \bullet] \in s_n \text{ and } s \xrightarrow{B} s'$$

Idea: LR(0) as machine layers

I like to think of the LR construction as building a big machine from little machines, in three layers:

1. Items give a whole-grammar analyis of what the machine may see

$$[A \to \alpha \bullet B\,\beta] \quad \xrightarrow{\varepsilon} \quad [B \to \ \bullet \gamma] \text{ if there is a rule } B \to \gamma$$

2. DFA from sets of items to get a deterministic automaton (the items are run in parallel "in lockstep")
3. LR(0) machine saves states of the automaton on its stack and runs the one at the top to guide its moves (the DFAs are run sequentially)

LR machine run example with LR(0) items

$$
\begin{array}{rcl}
S & \to & A \\
S & \to & B \\
A & \to & a\,b \\
B & \to & a\,c \\
D & \to & a
\end{array}
$$

$$
\begin{array}{rrl}
 & \langle\, s_1 \ ,\ a\,b \,\rangle & \\
\xrightarrow{\text{shift}} & \langle\, s_1\ s_2 \ ,\ b \,\rangle & \text{because } s_1 \xrightarrow{a} s_2 \\
\xrightarrow{\text{shift}} & \langle\, s_1\ s_2\ s_3 \ ,\ \varepsilon \,\rangle & \text{because } s_2 \xrightarrow{b} s_3 \\
\xrightarrow{\text{reduce}} & \langle\, s_1\ s_4 \ ,\ \varepsilon \,\rangle & \text{because } s_1 \xrightarrow{A} s_4
\end{array}
$$

$$
\begin{array}{rcl}
[S \to \bullet A], [S \to \bullet B] & \in & s_1 \\
[A \to \bullet a\,b], [B \to \bullet a\,c] & \in & s_1 \\
[A \to a \bullet b], [B \to a \bullet c] & \in & s_2 \\
[A \to a\,b \bullet] & \in & s_3 \\
[S \to A \bullet] & \in & s_4
\end{array}
$$

Books on LR

I have tried to give you the idea of LR as simply as possible.
For further reading, I recommend:
Hopcroft, John E.; Ullman, Jeffrey D. (1979). Introduction to Automata Theory, Languages, and Computation (1st ed.).
Pages 248–264
Not second edition!
See the book for proofs on the power of LR(0) or LR(1)
Primary source: On the translation of languages from left to right, by Donald E. Knuth, 1965
`http://www.sciencedirect.com/science/article/pii/S0019995865904262`

Stretch exercise on LR(0)

Prove (or argue informally) that the LR(0) machine simulates the LR machine.
The main point is that the stacks are different.
When the LR machine stack contains a sequence of symbols

$$X_1 \dots X_n$$

then the LR(0) stack contains a sequence of sets of LR(0) items

$$s_0 \dots s_n$$

such that

$$s_0 \xrightarrow{X_1} s_1 \xrightarrow{X_2} \cdots \xrightarrow{X_n} s_n$$

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

LL vs LR revisited

$$
\begin{array}{rcl}
S & \to & A \\
S & \to & B \\
A & \to & a\,b \\
B & \to & a\,c
\end{array}
$$

FIRST/FIRST conflict
$\Rightarrow$ LL(1) machine cannot predict A or B
based on a in the input ☹

By contrast:
LR(0) machine makes decision after shifting either ab or ac
and looking at the resulting item 😎

$$[A \to a\, b\, \bullet] \text{ or } [B \to a\, c\, \bullet]$$

LL vs LR ideas revisited

The nondeterministic LL and LR machines are equally simple.
Making them deterministic is very different.
LL(1) is essentially common sense: avoid predictions that get the machine stuck
LR(0) and LR(1) took years of research by top computer scientists
Build automaton from sets of items and put states of that automaton on the parsing stack
(In current research, the idea of a big machine made up from smaller machines may be used in abstract machines for concurrency, for example.)

LR(1) machine

- LR(0) parsers are already powerful
- LR(1) uses lookahead in the items for even more power.
- Compare: FIRST/FOLLOW construction for LL(1) machine
- Not something one would wish to calculate by hand
- LR(1) parser generators like Menhir construct the automaton and parser from a given grammar. ☺
- The construction will not always work, in which case the parser generator will complain about shift/reduce or reduce/reduce conflicts ☹
- The grammar may have to be refactored to make it suitable for LR(1) parsing
- Some tools (e.g. yacc) have ad-hoc ways to resolve conflicts

Implementing LR parser generators

The LR construction may seem heavyweight, but is not really
LR became practical with yacc
Table-driven parser
sets of items can be represented as integers (only finitely many)
automaton = lookup-table, 2-dimensional array
GOTO and ACTION functions guide the stack machine
For details, see compiler textbooks, e.g. Aho et al or Appel

Stretch exercise

For finite automata, the powerset construction always gives us a DFA from an NFA.
Explain why we cannot just use this construction to make stack machines deterministic.
(If we could, it would drastically simplify parser generators; but no such luck.)

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

Problem: ambiguous grammars ☹

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

$$
\begin{array}{rcl}
E & \rightarrow & E - E \\
E & \rightarrow & 1
\end{array}
$$

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.

Left recursion ☹

In fact, this grammar also has a FIRST/FIRST conflict.

$$\begin{array}{lcl} E & \rightarrow & E - E \\ E & \rightarrow & 1 \end{array}$$

1 is in FIRST of both rules
$\Rightarrow$ predictive parser construction fails
Standard solution: left recursion elimination
(Note: ANTLR v4 can deal with left recursion)

Left recursion elimination example

$$
\begin{array}{rcl}
E & \rightarrow & E \texttt{ - } E \\
E & \rightarrow & \texttt{1}
\end{array}
$$

We observe that $E \Rightarrow^* \texttt{1 - 1 - } \ldots \texttt{ - 1}$
Idea: 1 followed by 0 or more " - 1"

$$
\begin{array}{rcl}
E & \rightarrow & \texttt{1 } F \\
F & \rightarrow & \texttt{ - 1 } F \\
F & \rightarrow &
\end{array}
$$

This refactored grammar also eliminates the ambiguity. Yay. ☺

Problem: C/C++ syntax sins against parsing

C borrowed declaration syntax from Algol 60.

```
int *p;
```

Fine as long as there was only `int`, `char` etc.
But then came typedef.

```
x * p;
```

Is that a pointer declaration or a multiplication?
Depends on whether there was

```
typedef int x;
```

C/C++ compilers may have to look at the symbol table for parsing. ☹

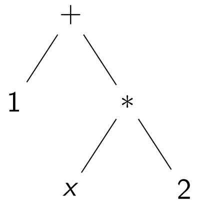
Pascal syntax is more LL-friendly: `var x : T;` ☺

Abstract syntax tree

In principle, a parser could build the whole parser tree:

$$\begin{array}{rl} & \text{LL or LR machine run} \\ \to & \text{leftmost or rightmost derivation} \\ \to & \text{parse tree} \end{array}$$

In practice, parsers build more compact abstract syntax trees.
Leave out syntactic details.
Just enough structure for the semantics of the language. For example:



Parser generators

Except when the grammar is very simple, one typically does not program a parser by hand from scratch. ¡ Instead, one uses a *parser generator*.
Compare:

$$\text{C code} \xrightarrow{\text{Compiler}} \text{x86 binary}$$

$$\text{Grammar} \xrightarrow{\text{Parser generator}} \text{Parser}$$

$$\text{Grammar+ annotations} \xrightarrow{\text{Parser generator}} \text{Parser+semantic actions}$$

Examples of parser generators: yacc, bison, ANLTR, Menhirm JavaCC, SableCC, ...

A concluding remark on nondeterminism

Refining programs by making them more deterministic is a useful technique in programming in general, not just parsing.
See for example,
Edsger Dijkstra: "Guarded commands, nondeterminacy and formal derivation of programs" (google it)
Nondeterminism seems weird at first, but is in fact a useful generalization for thinking about programs, even when you want a deterministic programs in the end.
Even Dijkstra writes:
"I had to overcome a considerable mental resistance before I found myself willing to consider nondeterministic programs seriously."

Realistic grammars

Grammars for real programming languages are a few pages long.
Example:
`https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`
Note: the colon ":" in yacc is used like the arrow $\to$ in grammars.
Exercise: derive this string in the C grammar:

```
int f(int *p)
{
  p[10] = 0;
}
```

[**Stretch exercise:** You may attempt it in groups or discuss it on Canvas or Facebook.]

More on parser generators

- Parser generators use some ASCII syntax rather than symbols like $\rightarrow$.
- With yacc, one attaches parsing actions to each production that tell the parser what to do.
- Some parsers construct the parse tree automatically. All one has to do is tree-walking.
- Parser generators often come with a collection of useful grammars for Java, XML, HTML and other languages
- If you need to parse a non-standard language, you need a grammar suitable for input to the parser generator
- Pitfalls: ambiguous grammars, left recursion

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

LL machine and ANTLR

- We could extend the LL machine to become more realistic
- Use k symbols of lookahead, as needed
- Compute semantic actions in addition to just parsing
- Use semantic predicates to guide the parsing decisions for grammars that are not LL(k)
- For each grammar rule, ANTLR generates a C function
- ANTLR uses the C (or Java, or ...) call stack as its parsing stack

Parser generator overview

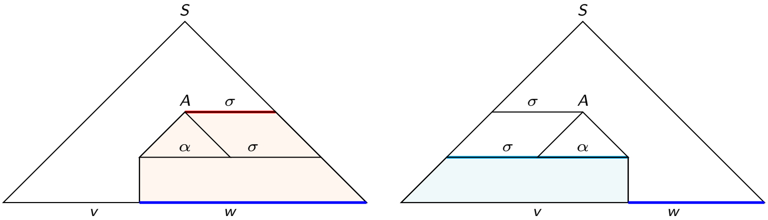
Both LL and LR generators exist.
There are various way to interface the parser and the rest of the compiler.

Parser generator	LR or LL	Tree processing
Yacc/Bison	LALR(1)	Parsing actions in C
Menhir	LR(1)	Parse tree in Ocaml
ANTLR	LL(k)	Tree grammars + Java or C++
SableCC	LALR(1)	Visitor Pattern in Java
JavaCC	LL(k)	JJTree + Visitors in Java

Some geometric intuition about LL and LR

If you have made it this far, you deserve some pictures.

- Symbols are points (0 dimensional)
- Strings of symbols are horizontal lines (1 dimensional)
- Derivations are horizontal slices (2 dimensional)
- LL tries to make the red-blue slice smaller until it vanishes
- LR tries to make the cyan-topped slice bigger until it fills the tree

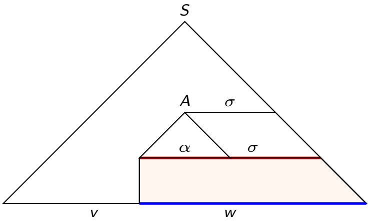


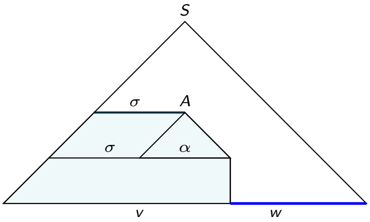
Switch back and forth in the PDF to attain enlightenment. It's, like, duality. Left ↔ right and down ↔ up.

Some geometric intuition about LL and LR

If you have made it this far, you deserve some pictures.

- Symbols are points (0 dimensional)
- Strings of symbols are horizontal lines (1 dimensional)
- Derivations are horizontal slices (2 dimensional)
- LL tries to make the red-blue slice smaller until it vanishes
- LR tries to make the cyan-topped slice bigger until it fills the tree

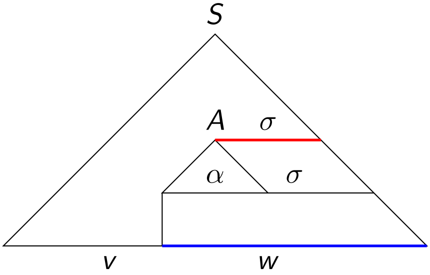




Switch back and forth in the PDF to attain enlightenment. It's, like, duality. Left ↔ right and down ↔ up.

LL is top down: predict steps

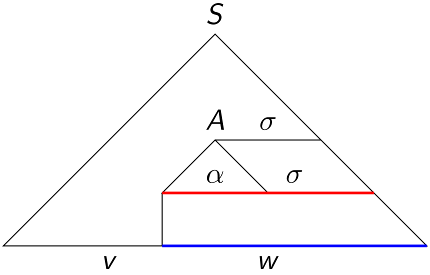
A predict step pushes the red line down towards the blue one.



LL stack = horizontal cut across the parse tree
above remaining input

LL is top down: predict steps

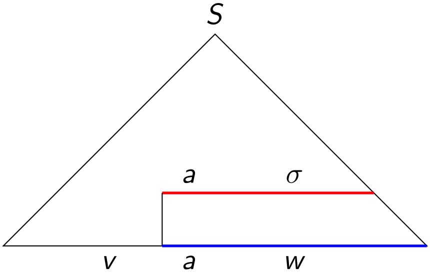
A predict step pushes the red line down towards the blue one.



LL stack = horizontal cut across the parse tree
above remaining input

LL is top down: match steps

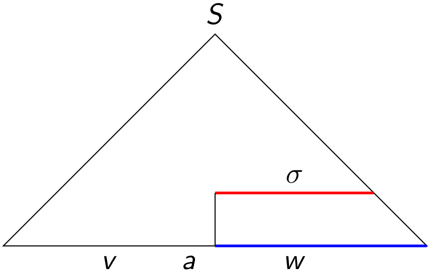
A match step shortens both lines



LL stack = horizontal cut across the parse tree above remaining input

LL is top down: match steps

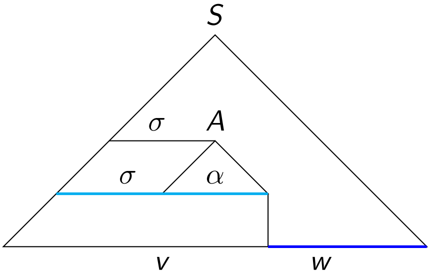
A match step shortens both lines



LL stack = horizontal cut across the parse tree
above remaining input

LR is bottom up: reduce steps

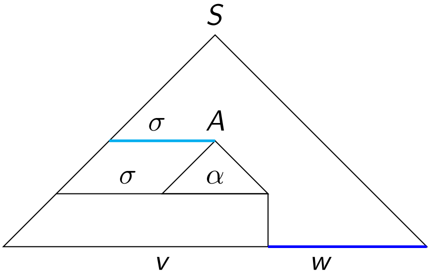
A reduce step pushes the combined cyan and blue lines towards the root at the top.



LR stack = horizontal cut across the parse tree above consumed input

LR is bottom up: reduce steps

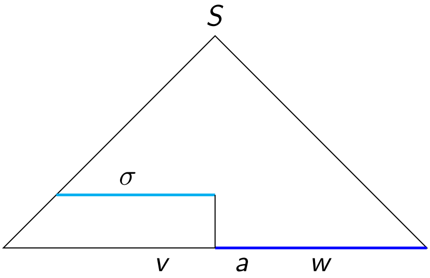
A reduce step pushes the combined cyan and blue lines towards the root at the top.



LR stack = horizontal cut across the parse tree above consumed input

LR is bottom up: shift steps

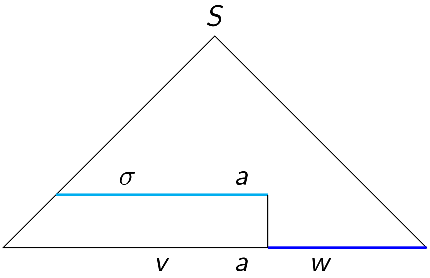
A shift step moves the boundary from blue to cyan



LR stack = horizontal cut across the parse tree above consumed input

LR is bottom up: shift steps

A shift step moves the boundary from blue to cyan



LR stack = horizontal cut across the parse tree above consumed input

Summary of parsing ideas

1. The parser must find a derivation for a given input
2. Use stack machines to simulate derivations
3. Two ways to use the stack: LL or LR
4. LL: prediction of future input to build derivation
5. LR: reduction of past input to build derivation in reverse
6. At first, the machines are nondeterministic, but correct
7. Make machine deterministic for efficient parsing
8. Use lookahead to make LL more deterministic
9. LL parser may fail to become deterministic due to FIRST/FIRST or FIRST/FOLLOW conflicts
10. Use items to make LR more deterministic
11. LR parser may fail to become deterministic due to reduce/reduce or shift/reduce conflicts