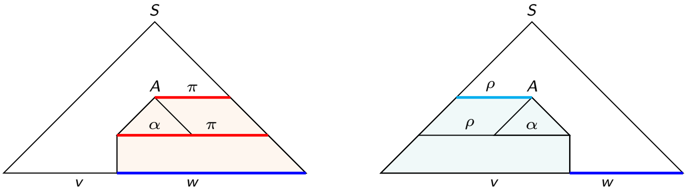
Compilers and Languages/
Compiler Construction

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

October 19, 2016



Contents

Motto

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

Henri Poincaré

Compilers and Languages $\subsetneqq$ Compiler Construction

Computer Science is *both* like mathematics (or theoretical physics) and engineering.

Practice builds on theory (e.g. LR(k) $\to$ yacc, bison, Menhir)

For compiling at Bham, we now have

1. a 10 credit module for the theory:
   Compilers and Languages
2. a 20 credit module for theory *and* building a toy compiler:
   Compiler Construction

One could try to build a compiler without the theory, but that would be awful.

Like building an aircraft using no maths or physics and only medieval ideas.

(But it is fine to have some compiling-inspired exercises in programming modules in Year 2.)

Compilers and Languages $\subsetneq$ Extended

The extended module is aimed mainly at the MEng and MSci at Level 4/M

It has some additional homework, like using a parser generator

It is open to some MSc students

But if you have not been an undergraduate at Bham, you may lack background

Bham students know functional programming

MSc students need to be careful about failing even a single module

Compilers and Languages

Syllabus

1. LL and LR parsing using stack machines
2. Compiling C-like language: stack frames, x86 code
3. Compiling functional languages

More research-led than in previous years.
Smaller syllabus, but more rigorous.

Background from Year 1 and 2 modules

1. Grammars and derivations; automata
2. Computer architecture and C/C++
3. Functional programming

Assessment

- 20% class tests
- 80% exam

What is a compiler?

- A compiler consists of a sequence of transformations between programming languages that preserve meaning of programs
- The meaning of programs is given by steps between (abstract) machine states
- For example: LLVM/clang and its IR

In this module:

- LL and LR
  grammar $\to$ stack machine
- stack frames for function calls
  variable $\to$ indexed addressing
- SECD/CEK machine with closures for functional programs
  function $\to$ code $+$ environment

Today we have a clear overall picture

Parsing section - aims and style

- Some of this material is *implicit* in what you can find in any comprehensive compilers textbook
- such as 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/LR machine, it will be easier to understand the CEK/SECD/Krivine machines
- 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
- Frege's principle

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

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?

Dyck languages idealize the parsing problem

Lisp is like Dyck(1)

```
(lambda (f)
  (cdr (f (cdaddr (cthulhu))
)))
```

C is like Dyck(3) and C++ like Dyck(4)

```
void C<D<int> >::f(int (*p)())
{
   if(g()) {
     x[h[0]()] = p();
   }
}
```

To parse programming languages, we must be able to parse at least Dyck languages; and the techniques scale up.

Grammars: formal definition

A context-free grammar consists of

- terminal symbols `a`, `b`, ..., `+`, `)`,...
- non-terminal symbols A, B, S,...
- a distinguished non-terminal start symbol S
- 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 (some more than others):

α	alpha
β	beta
γ	gamma
ε	epsilon
σ	sigma
π	pi
ρ	rho
λ	lambda

λ will be used only in lambda calculus later in the module

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, ...
- v, w, x, y, z 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 standard in the literature

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:

  $$S \stackrel{*}{\Rightarrow} w$$

The Kleene star $*$ on $\stackrel{*}{\Rightarrow}$ means "any number of steps".

Leftmost and rightmost derivation steps

Given a rule $A \to \alpha$, we have
leftmost derivation steps

$$w\,A\,\gamma \Rightarrow_\ell w\,\alpha\,\gamma$$

rightmost derivation steps

$$\beta\,A\,z \Rightarrow_r \beta\,\alpha\,z$$

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets. Consider this grammar for Dyck(2):

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

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

$$D$$

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets. Consider this grammar for Dyck(2):

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

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

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

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets. Consider this grammar for Dyck(2):

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

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

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

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets.
Consider this grammar for Dyck(2):

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

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

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

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets. Consider this grammar for Dyck(2):

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

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

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

An example derivation in the Dyck language

The Dyck language consists of all strings of matching brackets.
Consider this grammar for Dyck(2):

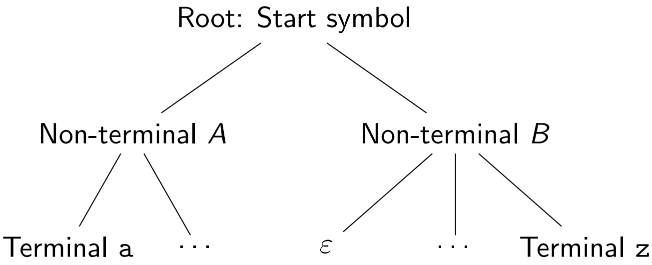
$$
\begin{array}{lcl}
D & \to & [\, D \,]\, D \\
D & \to & (\, D \,)\, D \\
D & \to &
\end{array}
$$

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

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

Parse trees

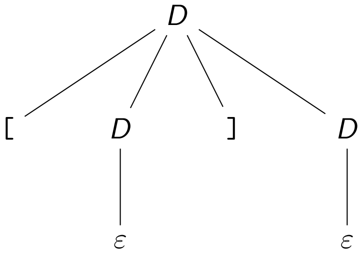
The internal nodes are labelled with nonterminals.
If there is a rule $A \to X_1 \dots X_n$, then an internal node can have the label A and children $X_1, \dots, X_n$.
The root node of the whole tree is labelled with the start symbol.
The leaf nodes are labelled with terminal symbols or ε.



Example: parse trees

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

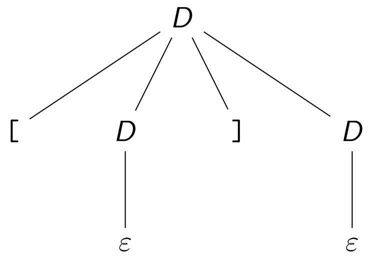
Here is a parse tree for the string `[ ]`:



Parse trees and derivations

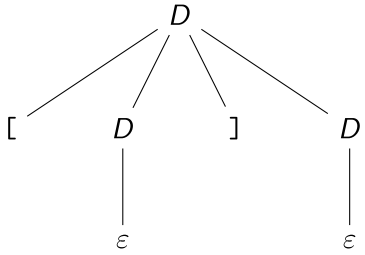
- Parse trees and derivations are related, but not the same.
- Intuition: Parse trees are extended in space (data structure), derivations in time.
- For each derivation of a word, there is a parse tree for the word.
  (Idea: each step using $A \to \alpha$ tells us that the children of some A-labelled 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



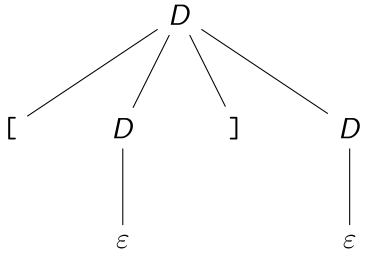
D

Parse tree traversal and derivation



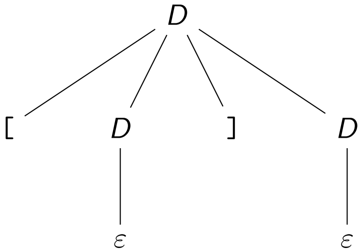
$$
\begin{array}{ll}
 & D \\
\Rightarrow & [\, D \,]\, D
\end{array}
$$

Parse tree traversal and derivation



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

Parse tree traversal and derivation



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

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:

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

Repetition α^* can be expressed as follows:

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

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)

Ambiguous grammars ☹

- A grammar is called ambiguous if there is a word in the language that has more than one parse tree.
- Ambiguous grammars are useless for our purposes here.
- For each program to be compiled, there should be a unique parse tree
- Then each program can have a unique meaning.
- ambiguity $\neq$ non-determinism (of automata or machines)
- non-determinism is not inherently bad, just inefficient

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.

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

Parsing then printing: almost the same string.

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

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:

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

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

Dyck language $\to$ stack machine

How can we parse the Dyck language using a stack?

Dyck language → stack machine

How can we parse the Dyck language using a stack?

1. If we see a `[` in the input:

Dyck language $\rightarrow$ stack machine

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
   we predict that we hope to see the matching closing bracket later

Dyck language $\to$ stack machine

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
   we predict that we hope to see the matching closing bracket later
2. If we see a `]` in the input:
   (a) If the same `]` is on the top of the stack:

Dyck language $\to$ stack machine

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
   we predict that we hope to see the matching closing bracket later
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 → stack machine

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
   we predict that we hope to see the matching closing bracket later
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 → stack machine

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
   we predict that we hope to see the matching closing bracket later
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 → stack machine

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
   we predict that we hope to see the matching closing bracket later
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 → stack machine

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
   we predict that we hope to see the matching closing bracket later
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 $\to$ stack machine

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
   we predict that we hope to see the matching closing bracket later
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

Why abstract machines?

- Long successful history, starting from Peter Landin's 1964 paper "The mechanical evaluation of expressions" 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 machines are like imperative programming: step by step state change; can often be implemented using pointers

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
We do not have deterministic polynomial-time algorithms for NP-complete problems ☹

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

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

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

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 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.

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

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

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

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

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

LL machine run example: list idiom

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

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

Correct input accepted.

LL machine run example 2

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

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

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

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

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

LL machine run example 2

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

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

LL machine run example 2

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

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

LL machine run example: what should not happen

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

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

LL machine run example: what should not happen

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

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

LL machine run example: what should not happen

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

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

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

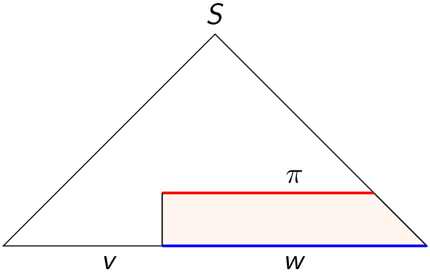
Lookahead will be used to avoid such bad predictions

LL is top down: invariant

A state $\langle \pi , w \rangle$ can be visualized as a horizontal slice of the parse tree

The machine tries to make the slice smaller until it goes away upon acceptance

Initially, the slice is the whole tree from S to $v\,w$



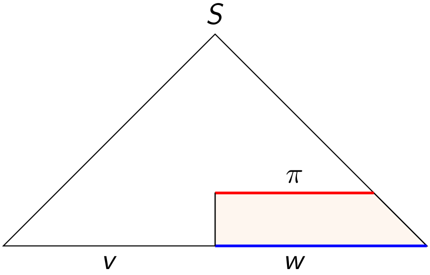
LL stack = horizontal cut across the parse tree above remaining input

LL is top down: invariant

A state $\langle \pi , w \rangle$ can be visualized as a horizontal slice of the parse tree

The machine tries to make the slice smaller until it goes away upon acceptance

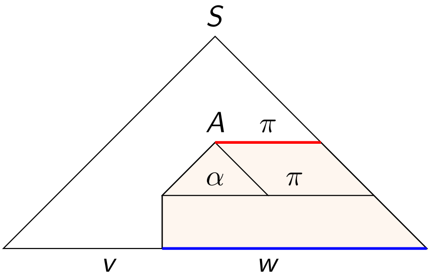
Initially, the slice is the whole tree from S to $v\,w$



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

LL is top down: predict steps

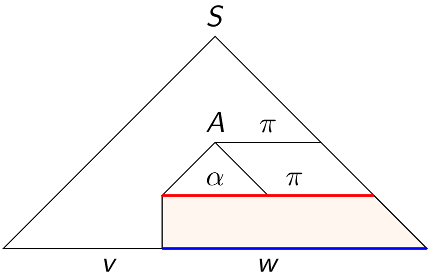
A predict step pushes the red line down towards the blue one.



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

LL is top down: predict steps

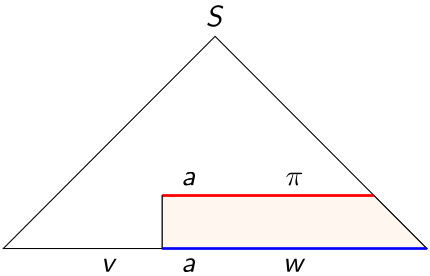
A predict step pushes the red line down towards the blue one.



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

LL is top down: match steps

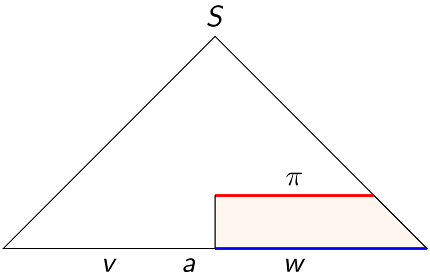
A match step shortens both lines



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

LL machine soundness proof

Lemma: If there is a step of the LL machine

$$\langle \pi_1 , w_1 \rangle \xrightarrow{*} \langle \pi_2 , w_2 \rangle$$

$$\text{and } \pi_2 \overset{*}{\Rightarrow}_\ell w_2$$

then

$$\pi_1 \overset{*}{\Rightarrow}_\ell w_1$$

Proof: exercise! Hint: case analysis on the LL machine steps and induction on the number of steps.

Theorem: If there is a run of the LL machine of the form

$$\langle S , w_1 \rangle \xrightarrow{*} \langle \varepsilon , \varepsilon \rangle$$

then

$$S \overset{*}{\Rightarrow}_\ell w_1$$

Proof: by the lemma as the special case

$$\pi_1 = S \qquad \pi_2 = \varepsilon \qquad w_2 = \varepsilon$$

LL exercise: stuck

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.

LL machine example: Dyck language

Consider this grammar

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

Write down the LL machine transitions for this grammar.
Show how the LL machine for this grammar can accept the input

`[ ( ) ]`

LR machine

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

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

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

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

$\langle S \,,\, \varepsilon \rangle$ 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:

$$\langle \varepsilon , w \rangle \xrightarrow{*} \langle S , \varepsilon \rangle$$

Same idea as for LL, but with different initial and accepting states.

Is greed good?

Should the LR machine be "greedy"[1]
keep shifting from input onto the stack;
as soon as the right-hand-side α of a rule

$$A \to \alpha$$

is at the top of the stack, do a reduce step

$$\langle\, \rho\, \alpha \;,\; w \,\rangle \xrightarrow{\text{reduce}} \langle\, \rho\, A \;,\; w \,\rangle$$

[1] Not a technical term, just used for some examples here

LR machine run example 1: accepting

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

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

LR machine run example 1: accepting

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

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

LR machine run example 1: accepting

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

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

LR machine run example 1: accepting

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

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

LR machine run example 1: accepting

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

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

LR machine run example 1: accepting

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

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

LR machine run example 2

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

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

LR machine run example 2

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

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

LR machine run example 2

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

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

LR machine run example 2

$$
\begin{array}{rcl}
S & \to & A \\
S & \to & B \\
D & \to & a \\
A & \to & a\,b \\
B & \to & a\,c
\end{array}
\qquad
\begin{array}{rl}
 & \langle \varepsilon\ ,\ a\,b \rangle \\
\xrightarrow{\text{shift}} & \langle a\ ,\ b \rangle \\
\xrightarrow{\text{reduce}} & \langle D\ ,\ b \rangle\ ☹
\end{array}
$$

We can reduce, but it would have been better to shift.

Greed will cause pain. (Taiwanese proverb)

LR machine run example 3

For any ρ, $\rho = \rho\,\varepsilon$. Thus ε is always at the top of the stack.

$$\begin{array}{lcl} S & \to & A \\ A & \to & \varepsilon \end{array}$$

$$\langle \varepsilon \,, \varepsilon \rangle$$

LR machine run example 3

For any ρ, $\rho = \rho\,\varepsilon$. Thus ε is always at the top of the stack.

$$
\begin{array}{lcl}
S & \to & A \\
A & \to & \varepsilon
\end{array}
\qquad\qquad
\begin{array}{rl}
 & \langle \varepsilon \,,\, \varepsilon \rangle \\
\xrightarrow{\text{reduce}} & \langle A\,\varepsilon \,,\, \varepsilon \rangle
\end{array}
$$

LR machine run example 3

For any ρ, $\rho = \rho\,\varepsilon$. Thus ε is always at the top of the stack.

$$\begin{aligned} S &\to A \\ A &\to \varepsilon \end{aligned}$$

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

LR machine run example 3

For any ρ, $\rho = \rho\,\varepsilon$. Thus ε is always at the top of the stack.

$$\begin{array}{lcl} S & \to & A \\ A & \to & \varepsilon \end{array}$$

$$\begin{array}{cl} & \langle \varepsilon\ ,\, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,\varepsilon\ ,\, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,A\,\varepsilon\ ,\, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,A\,A\,\varepsilon\ ,\, \varepsilon \rangle \end{array}$$

LR machine run example 3

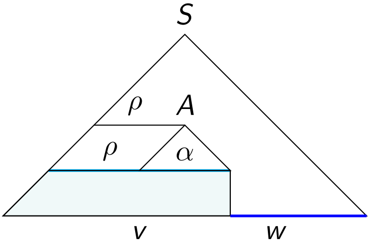
For any ρ, $\rho = \rho\,\varepsilon$. Thus ε is always at the top of the stack.

$$\begin{array}{rcl} S & \to & A \\ A & \to & \varepsilon \end{array}$$

$$\begin{array}{rl} & \langle \varepsilon \;, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,\varepsilon \;, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,A\,\varepsilon \;, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \langle A\,A\,A\,\varepsilon \;, \varepsilon \rangle \\ \xrightarrow{\text{reduce}} & \ldots \end{array}$$

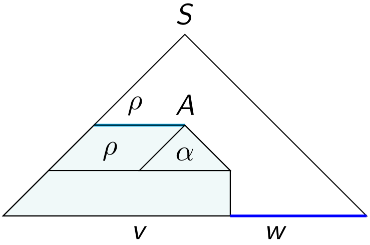
We can reduce $A \to \varepsilon$ at the second step
but it would have been better to reduce with $S \to A$.

LR machine is bottom-up: reduce step



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

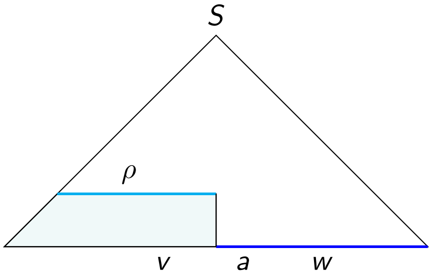
LR machine is bottom-up: reduce step



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

LR is bottom up: shift steps

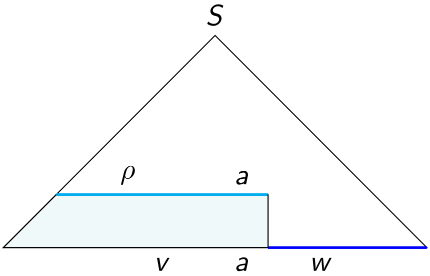
A shift step moves the boundary from blue to cyan



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

Is the LR machine always deterministic?

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

Is the LR machine always deterministic?

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

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

For some grammars, there may be:

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

LR machine soundness proof

Lemma: If there is a run of the LR machine of the form

$$\langle \rho_1 \, , \, w_1 \rangle \xrightarrow{*} \langle \rho_2 \, , \, w_2 \rangle$$

then

$$\rho_2 \, w_2 \stackrel{*}{\Rightarrow} \rho_1 \, w_1$$

Proof: by induction on the number of steps and case analysis on the last step.

Theorem: If there is a run of the LR machine of the form

$$\langle \varepsilon \, , \, w_1 \rangle \xrightarrow{*} \langle S \, , \, \varepsilon \rangle$$

then

$$S \stackrel{*}{\Rightarrow} w_1$$

Proof: by the lemma as the special case

$$\rho_1 = \varepsilon \qquad \rho_2 = S \qquad w_2 = \varepsilon$$

LL vs LR exercise

Consider this grammar

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

Show how the LL and LR machine can accept the input $c\,d$.
Compare the machine runs to a leftmost and (backwards) rightmost derivation. Draw them side by side like this:

LL run	leftmost derivation
LR run	rightmost derivation upside down

Draw the parse tree and compare how the machines traverse it

LL vs LR comparison

LL

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

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

LR

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

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

LL vs LR comparison

LL

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

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

LR

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

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

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

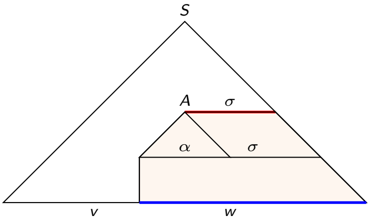
Here is a simple grammar:

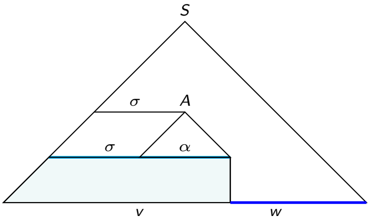
$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow a\, b \\
B &\rightarrow a\, c
\end{aligned}
$$

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.

Geometric intuition about LL vs LR

- 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

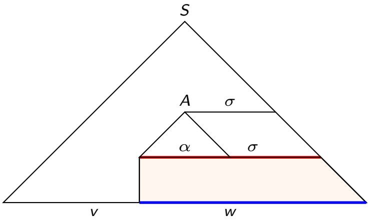




Switch back and forth in the PDF to attain enlightenment. It's almost, but not quite, duality. Left $\leftrightarrow$ right and down $\leftrightarrow$ up.

Geometric intuition about LL vs LR

- 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 almost, but not quite, duality. Left $\leftrightarrow$ right and down $\leftrightarrow$ up.

LEVEL UP!

Making the LL machine deterministic using lookahead

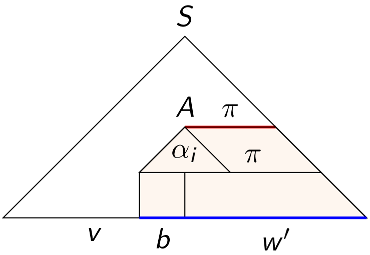
- Deterministic refinement of nondeterministic LL machine $\Rightarrow$ LL(1) machine.
- We use one symbol of lookahead to guide the predict moves, to avoid predict moves that get the machine stuck soon after
- Formally: FIRST and FOLLOW construction.
- Can be done by hand, though tedious
- The construction does not work for all grammars!
- Real-world: ANTLR does a more powerful version: LL(k) for any k.

LL and lookahead

The LL machine must decide between

$$\begin{array}{rcl} A & \to & \alpha_1 \\ A & \to & \alpha_2 \\ & \vdots & \end{array}$$

It can use lookahead = first symbol of the remaining input



Check whether

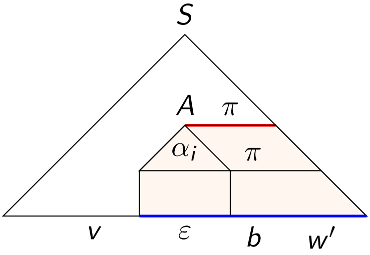
$$\alpha_i \overset{*}{\Rightarrow} b\,\gamma$$

LL and lookahead where nullable right-hand sides

The LL machine must decide between

$$
\begin{array}{lcl}
A & \to & \alpha_1 \\
A & \to & \alpha_2 \\
 & \vdots &
\end{array}
$$

It can use lookahead = first symbol of the remaining input



May happen if

$$\alpha_i \stackrel{*}{\Rightarrow} \varepsilon$$

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:

$$\langle\, E\,\pi\,,\, d\,w \,\rangle \longrightarrow \ldots$$

What should the next 4 steps be, and why?

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

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

Derivations for ε exercise

Suppose

$$\varepsilon \stackrel{*}{\Rightarrow} \alpha$$

What is α?

Hint:
CORDELIA Nothing, my lord.
KING LEAR Nothing!
CORDELIA Nothing.
KING LEAR Nothing will come of nothing.

FIRST and FOLLOW examples

Consider

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

Then

$$a \in \mathit{FIRST}(L)$$

$$b \in \mathit{FOLLOW}(L)$$

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?

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

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

$$\langle A\,\pi \,,\, b\,w \rangle \xrightarrow{\text{predict}} \langle \alpha\,\pi \,,\, b\,w \rangle \quad \text{if there is a rule } A \to \alpha$$

and $b \in \mathit{FIRST}(\alpha)$

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

and β is nullable

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

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

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

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

ε-rules and FOLLOW

Consider this rule:

$$\langle A\,\pi\ ,\ b\,w \rangle \quad \stackrel{\text{predict}}{\longrightarrow} \quad \langle \beta\,\pi\ ,\ b\,w \rangle \qquad \text{if there is a rule } A \to \beta$$

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

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

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

$$\langle A\,\pi\ ,\ b\,w \rangle \quad \stackrel{\text{predict}}{\longrightarrow} \quad \langle \pi\ ,\ b\,w \rangle \qquad \text{if there is a rule } A \to \varepsilon$$

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

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

$$\langle\, a\,\pi \;,\; b\,w \,\rangle$$

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

$$\langle\, \pi \;,\; \varepsilon \,\rangle$$

where $\pi \neq \varepsilon$, 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

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

LL(1) machine run example

Just like LL machine, but now deterministic

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

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

Is the LL(1) machine always deterministic?

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

$$\text{and } b \in \mathit{FIRST}(\alpha)$$

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

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

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

For some grammars, there may be:

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

FIRST/FIRST conflict $\Rightarrow$ nondeterminism ☹

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

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 ☹

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

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

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

The LL(1) machine is deterministic if there are none of:

1. FIRST/FIRST conflicts
2. FIRST/FOLLOW conflicts
3. more than one nullable right-hand sides for the same nonterminal

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

LL(1) refines LL machine

The computations of the LL machine form a tree
The LL(1) machine prunes the tree
predict steps that lead only to stuck states are eliminated

$$
\begin{tikzcd}
& \langle b\,\beta\,\pi \,,\, c\,w \rangle \\
\langle A\,\pi \,,\, c\,w \rangle \arrow[ur, dashed] \arrow[dr] & \\
& \langle c\,\gamma\,\pi \,,\, c\,w \rangle
\end{tikzcd}
$$

No conflicts $\Rightarrow$ LL(1) machine deterministic

Grammar:

$$\begin{array}{ccc} & \nearrow \alpha_1 & \text{where } a \in \mathrm{FIRST}(\alpha_1) \\ A & = & \\ & \searrow \alpha_2 & \text{where } a \in \mathrm{FIRST}(\alpha_2) \end{array}$$

LL(1) machine:

$$\begin{array}{cc} & \nearrow \langle\, \alpha_1\, \pi \;,\; a\, w \,\rangle \\ \langle\, A\, \pi \;,\; a\, w \,\rangle & = \\ & \searrow \langle\, \alpha_2\, \pi \;,\; a\, w \,\rangle \end{array}$$

FIRST/FIRST conflict $\neq$ 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

$$
\begin{aligned}
A &\to ab \\
A &\to ac
\end{aligned}
$$

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

Left factoring

This grammar has a FIRST/FIRST conflict ☹

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

Left factorize as follows:

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

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

Pre-computing FIRST and FOLLOW

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

$$\text{and } b \in \mathit{FIRST}(\alpha)$$

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

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

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

The grammar is fixed and known when the parser is constructed
whereas the input is not
Idea: FIRST and FOLLOW are not function calls that are recomputed every time
but array accesses
The FIRST, FOLLOW, and nullable arrays are computed only once when the parser is built
not every time it is run

FIRST and FOLLOW closure properties

Consider a grammar rule of the form

$$A \to \alpha\ B\ \beta\ C\ \gamma$$

What set inclusions hold for FIRST and/or FOLLOW if any of α, β, γ are nullable?

FIRST and FOLLOW closure properties

Consider a grammar rule of the form

$$A \to \alpha\ B\ \beta\ C\ \gamma$$

What set inclusions hold for FIRST and/or FOLLOW if any of α, β, γ are nullable?

α nullable $\Rightarrow$ FIRST(B) $\subseteq$ FIRST(A)

β nullable $\Rightarrow$ FIRST(C) $\subseteq$ FOLLOW(B)

γ nullable $\Rightarrow$ FOLLOW(A) $\subseteq$ FOLLOW(C)

FIRST and FOLLOW closure properties 1

Consider a rule

$$X \to Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k$$

Assume $Y_1, \dots, Y_{i-1}$ are all nullable. Hence $Y_1 \dots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Now suppose b is in $\textsc{first}(Y_i)$, that is, $Y_i \stackrel{*}{\Rightarrow} b\alpha$. Then

$$\begin{array}{rcl} X & \Rightarrow & Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k \\ & \stackrel{*}{\Rightarrow} & Y_i Y_{i+1} \dots Y_k \\ & \stackrel{*}{\Rightarrow} & b\alpha Y_{i+1} \dots Y_k \end{array}$$

Hence b is also in $\textsc{first}(X)$.

FIRST and FOLLOW closure properties 2

Consider a rule

$$X \to Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k$$

Now assume that $Y_{i+1}, \dots, Y_k$ are all nullable. Let some terminal symbol b be in $\textsc{follow}(X)$, that is $S \stackrel{*}{\Rightarrow} \alpha X b \gamma$. Then

$$\begin{array}{rcl} S & \stackrel{*}{\Rightarrow} & \alpha X b \gamma \\ & \stackrel{*}{\Rightarrow} & \alpha Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k b \gamma \\ & \stackrel{*}{\Rightarrow} & \alpha Y_1 \dots Y_{i-1} Y_i b \gamma \end{array}$$

Hence b is also in $\textsc{follow}(Y_i)$.

FIRST and FOLLOW closure properties 3

Consider a rule

$$X \to Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_k$$

Now assume $Y_{i+1}, \dots, Y_{j-1}$ are all nullable. Let b be in $\text{FIRST}(Y_j)$. Assuming that X is reachable, we have

$$\begin{array}{rcl} S & \stackrel{*}{\Rightarrow} & \alpha X \gamma \\ & \Rightarrow & \alpha Y_1 \dots Y_{i-1} Y_i Y_{i+1} \dots Y_{j-1} Y_j Y_{j+1} \dots Y_k \gamma \\ & \stackrel{*}{\Rightarrow} & \alpha Y_1 \dots Y_{i-1} Y_i Y_j Y_{j+1} \dots Y_k \gamma \\ & \stackrel{*}{\Rightarrow} & \alpha Y_1 \dots Y_{i-1} Y_i b \alpha Y_{j+1} \dots Y_k \gamma \end{array}$$

Hence b is also in $\text{FOLLOW}(Y_i)$.

Computing FIRST and FOLLOW as least fixpoint

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

Soundness/partial correctness of the LL(1) machine

The soundness of the LL(1) machine follows from that of the LL machine.
There is a (quite trivial) simulation relation between the machines.

$$
\begin{array}{cc}
\textbf{LL(1)} & \textbf{LL} \\
\\
\langle \pi_1 \, , \, w_1 \rangle & \langle \pi_1 \, , \, w_1 \rangle \\
\big\downarrow \forall & \big\downarrow \exists \\
\langle \pi_2 \, , \, w_2 \rangle & \langle \pi_1 \, , \, w_1 \rangle
\end{array}
$$

Hence:
LL(1) accepts input
$\Rightarrow$ LL accepts input
$\Rightarrow$ there is a derivation for it

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.

LEVEL UP!

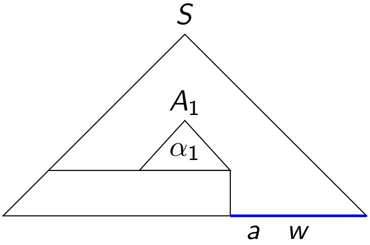
Making the LR machine deterministic using items

- Construction of LR(k) items
- Much more complex than FIRST/FOLLOW construction
- LR(1) items Menhir
- LALR(1) items, to consume less memory (yacc, bison)
- 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 nondeterminism

Should the LR machine:

- shift
- reduce with $A_1 \to \alpha_1$
- reduce with $A_2 \to \alpha_2$



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

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

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

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

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

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.

$$\langle \sigma\, s \,,\, a\, w \rangle \quad \xrightarrow{\text{shift}} \quad \langle \sigma\, s\, s' \,,\, w \rangle \quad \text{if } [B \to \alpha \bullet c\, \beta] \in s \text{ and } s \xrightarrow{a} s'$$

$$\langle \sigma\, s_0\, s_1 \,\ldots\, s_n \,,\, w \rangle \quad \xrightarrow{\text{reduce}} \quad \langle \sigma\, s_0\, s' \,,\, w \rangle \quad \text{if } [B \to X_1 \ldots X_n \bullet] \in s_n \text{ and } s_0 \xrightarrow{B} s'$$

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 track what may appear on the LR stack:

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

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

   Note that the input alphabet of this machine includes nonterminals
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)

BOSS LEVEL

LR machine run example with LR(0) items

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

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

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

Points to note in the LR(0) example

Having the same symbol on the right-hand side of

$$A \to a\, b \text{ and } B \to a\, c$$

causes no problems.
After the machine has read a, it considers both

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

and only decides after having seen either b or c. The powerset DFA allows many possibilities to be tracked at the same time.

The rule $D \to a$ causes no problem. As there is no item $[D \to \bullet a]$ before the a is shifted, there is no item $[D \to a \bullet]$ after a is shifted. Hence no reduce.

Exercise

What happens for

$$\langle s_1 \,,\, a\, c \rangle$$

What happens for

$$\langle s_1 \,,\, a\, a \rangle$$

What happens if we add the rule

$$S \to D\, c$$

What happens if we add the rule

$$A \to A\, c$$

Would the LL(1) machine work for this grammar?

Shift/reduce and reduce/reduce conflicts in LR(0)

A state s has a shift/reduce conflict if

$$[A_1 \to \alpha_1 \bullet] \in s \text{ and } [A_2 \to \alpha_2 \bullet b\,\beta] \in s$$

A state s has a reduce/reduce conflict if

$$[A_1 \to \alpha_1 \bullet] \in s \text{ and } [A_2 \to \alpha_2 \bullet] \in s$$

and not both $A_1 = A_2$ and $\alpha_1 = \alpha_2$
A parser generator will detect these when constructing the DFA and report an error.
Compare: FIRST/FIRST and FIRST/FOLLOW for LL(1)

LR(k)

LR(k) items are like LR(0) items combined with a lookahead set L:

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

The L can be compared to the lookahead and guide the parser
The basic insight of item sets on the stack is already in LR(0)

Efficient imlementation of LR(k)

Aren't the sets of items on the stack big and cumbersome?
Like lists of items, need to be malloced ...?
Not inefficient if you encode sets of items properly (strictly speaking, we implement an automaton isomorphic to the LR(0) one)
Finite grammar $\Rightarrow$ finitely many items $\Rightarrow$ finitely many sets of items $\Rightarrow$ can encode as integers
$\Rightarrow$ the LR(0) stack holds integers
The machine steps are encoded as two arrays ACTION and GOTO
Requires lots of memory by 1960s standards (OMG kilobytes!)
Nowadays, memory is cheap

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

Further reading and history on LR

- I have tried to give you the ideas behind LR as simply as possible but not all the details
- For further reading, I recommend:
- Hopcroft, John E.; Ullman, Jeffrey D. (1979). Introduction to Automata Theory, Languages, and Computation (1st ed.).
  Not second edition!
  Pages 248–264
- See the book for automata and proofs on the power of LR(0)
- Primary source: "On the translation of languages from left to right", by Donald E. Knuth, 1965
  `http://www.sciencedirect.com/science/article/pii/S0019995865904262`
- Fun fact: Knuth also made TeX
  TeX is a kind of compiler: text + markup → PDF
  like a Dyck language `{{{{}[]{}}}{}{{}{}}}`
  and these slides are in Beamer, on top of LaTeX

LL vs LR revisited

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

FIRST/FIRST conflict
⇒ 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.)

Problem: ambiguous grammars ☹

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

$$
\begin{aligned}
E &\rightarrow E - E \\
E &\rightarrow 1
\end{aligned}
$$

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.
Infamous example of ambiguity: "dangling else" `if` ... `if` ... `else`

Left recursion ☹

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

$$
\begin{aligned}
E &\rightarrow E - E \\
E &\rightarrow 1
\end{aligned}
$$

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

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

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

$$
\begin{array}{lcl}
E & \to & \texttt{1}\ F \\
F & \to & \texttt{ - 1}\ F \\
F & \to &
\end{array}
$$

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;
```

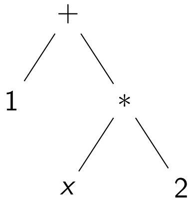
C/C++ compilers may have to look at the symbol table for parsing.
Pascal syntax is more LL-friendly: `var x :  T;` ☺

Abstract syntax tree

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

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

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



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."

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:

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

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

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

More on parser generators

- Parser generators use some ASCII syntax rather than symbols like $\to$.
- 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 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

Realistic grammars

Grammars for real programming languages have 100s of rules and are a few pages long

Example:

`https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`

clickable grammar for C in yacc notation

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;
}
```

Recursive descent parsing

- One may still write a parser by hand instead of using a parser generator
- E.g. C front end in clang
- "Recursive descent" : grammar rule $\rightarrow$ recursive function
- can use any modern language: recursive functions are required, exceptions and datatypes are nice to have
- lots of mutual recursion
- his gives a "predictive" top-down parser, like LL(k)
- as since it just code, you can do whatever you want ☺
- unlike LL and LR, not correct by construction ☹

Parsing stack and function call stack

A useful analogy: a grammar rule

$$A \rightarrow B\ C$$

is like a function definition

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

In top-down parsing, the stack works like a simpler version of the function call stack
ANTLR uses the function call stack as its parsing stack

Recursive methods

From grammars to mutually recursive methods:

- For each non-terminal A there is a method A. The method body is a switch statement that chooses a rule for A.
- For each rule $A \to X_1 \dots X_n$, there is a branch in the switch statement. There are method calls for all the non-terminals among $X_1, \dots, X_n$.

Each grammar gives us some recursive methods.
For each derivation in the language, we have a sequence of method calls.

The `lookahead` and `match` methods

- A predictive parser relies on two methods for accessing the input string:
- `char lookhead()` returns the next symbol in the input, without removing it.
- `void match(char c)` compares the next symbol in the output to `c`. If they are the same, the symbol is removed from the input. Otherwise, the parsing is stopped with an error; in Java, this can be done by throwing an exception.

FIRST and FOLLOW give the case labels

- FIRST and FOLLOW gives us the case labels for the branches of the switch statement.
- A branch for $A \to \alpha$ gets the labels in $\mathrm{FIRST}(\alpha)$.
- A branch for $A \to \varepsilon$ gets the labels in $\mathrm{FOLLOW}(A)$.

Parsing with lookahead

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

We also need to know where else in the grammar a D could occur:

$$S \to D\ \$$$

Idea: suppose you are trying to parse a D. Look at the first symbol in the input:
if it is a [, use the first rule;
if it is a] or $, use the second rule.

Recursive descent parser in Java for Dyck(1)

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

```
void parseD() throws SyntaxError
{
    switch(lookahead())  { // what is in the input?
        case '[':          // If I have seen a [
           match('[');     // remove the [
           parseD();       // now parse what is inside
           match(']');     // make sure there is a ]
           parseD();       // now parse what follows
           break;          // done in this case
        case ']': case '$':   // If I have seen a ] or $
           break;          // just return
        default: throw new SyntaxError();
    }
}
```

The dollar sign marks the end of the input

Summary of parsing ideas

1. The parser must find a derivation for a given input if possible
2. Use stack machines to simulate derivations
3. Two ways to use the stack: LL or LR
4. LL: stack = prediction of future input
5. LR: stack = reduction of past input
6. At first, the machines are nondeterministic, but correct
7. Make machine deterministic for efficient parsing
8. LL(1) uses lookahead to make LL more deterministic
9. LL(1) parser may fail to become deterministic due to FIRST/FIRST or FIRST/FOLLOW conflicts
10. LR(0) uses items to make LR more deterministic
11. LR(0) parser may fail to become deterministic due to reduce/reduce or shift/reduce conflicts