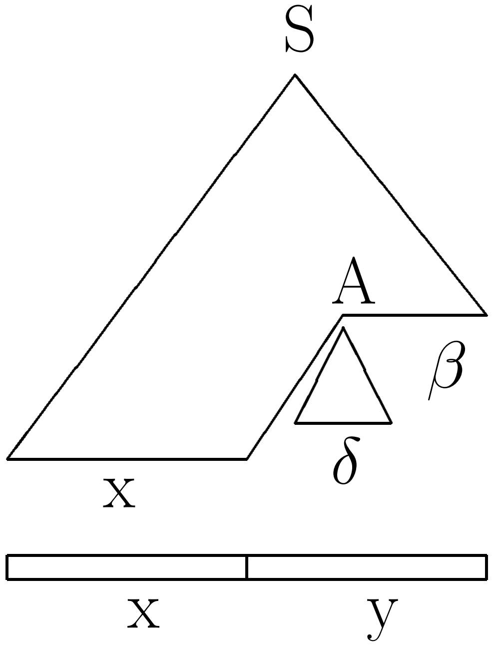
REVIEW – Parse tree, sentential form, frontier

Parse tree:

"Graphical" representation for a derivation; filters out the choice regarding non-terminal replacement order.

$$S \;\Rightarrow^*_{lm}\; xA\beta \;\Rightarrow_{lm}\; x\delta\beta \;\Rightarrow^*_{lm}\; xy$$



Sentential form:

Any α such that $S \Rightarrow^* \alpha$

Fringe or frontier of parse tree:

Labels of leaf nodes of (partial) parse tree read from left to right.

REVIEW – Ambiguity

Ambiguous CFG:

A word in the language has multiple parse trees.

Examples of ambiguous CFG:

- simple expressions grammar (E := E <op> E).
- "dangling else" grammar.

What to do about it?

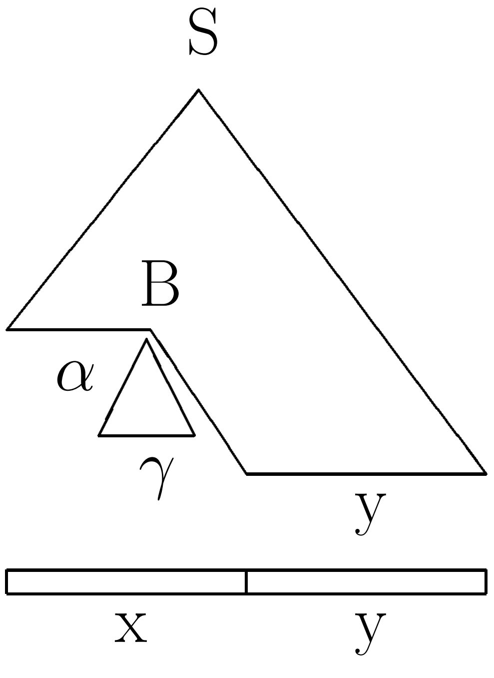
- Add disambiguating productions, if possible.
- There are context-free *languages* for which every context-free *grammar* is ambiguous.
  Example (see Hopcroft&Ullman, p.99):

  $\{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$

REVIEW – Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid prefixes*)
- use a stack to store both state and sentential forms

$$S \quad \Rightarrow^*_{rm} \quad \alpha \mathrm{By} \quad \Rightarrow_{rm} \quad \alpha\gamma y \quad \Rightarrow^*_{rm} \quad xy$$

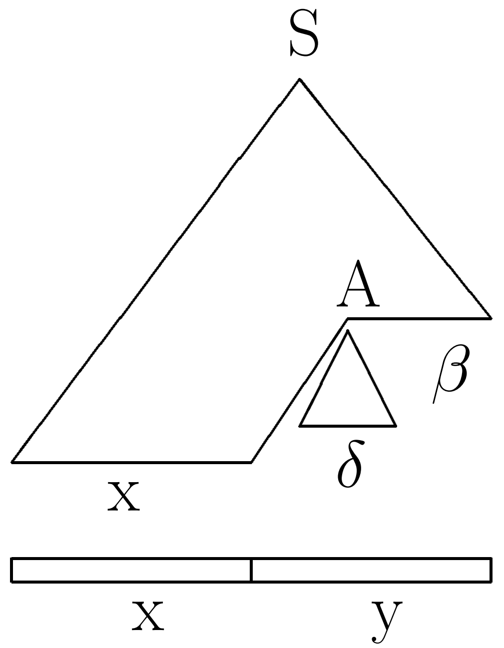


- *LR parsing:* Reads input from *left to right* and constructs *rightmost* derivation (backwards)

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

$$S \Rightarrow^*_{lm} xA\beta \Rightarrow_{lm} x\delta\beta \Rightarrow^*_{lm} xy$$



- *LL Parsing:* reads input from *left to right* and constructs *leftmost* derivation (forwards)

Which parsing technique seems to be more powerful?

Top-down parsing

First try: Let's use guessing and backtracking

A top-down parser starts with the root of the parse tree. It is labelled with the start symbol or goal symbol of the grammar.

To build a parse tree, it repeats the following steps until the fringe of the parse tree matches the input string.

1. At leftmost node labelled (non-terminal) A, select a production with A on its lhs and for each symbol on its rhs, construct the appropriate child. Let's ignore the actual input string for the selection.
2. When a terminal is added to the fringe that doesn't match the input string, backtrack.
3. Find the next node to be expanded. (Must have a label in NT)

Simple expression grammar

Recall our grammar for simple expressions:

$$
\begin{array}{r|rcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle\ \texttt{+}\ \langle\text{term}\rangle \\
3 & & | & \langle\text{expr}\rangle\ \texttt{-}\ \langle\text{term}\rangle \\
4 & & | & \langle\text{term}\rangle \\
5 & \langle\text{term}\rangle & ::= & \langle\text{term}\rangle\ \texttt{*}\ \langle\text{factor}\rangle \\
6 & & | & \langle\text{term}\rangle\ /\ \langle\text{factor}\rangle \\
7 & & | & \langle\text{factor}\rangle \\
8 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
9 & & | & \texttt{id}
\end{array}
$$

Consider the input string `x - 2 * y`

Example – guesses next production

Prod'n	Sentential form	Input
–	<goal>	↑x - 2 * y
1	<expr>	↑x - 2 * y
2	<expr> + <term>	↑x - 2 * y
4	<term> + <term>	↑x - 2 * y
7	<factor> + <term>	↑x - 2 * y
9	<id> + <term>	↑x - 2 * y
–	<id> + <term>	x ↑- 2 * y
–	<expr>	↑x - 2 * y
3	<expr> - <term>	↑x - 2 * y
4	<term> - <term>	↑x - 2 * y
7	<factor> - <term>	↑x - 2 * y
9	<id> - <term>	↑x - 2 * y
–	<id> - <term>	x ↑- 2 * y
–	<id> - <term>	x - ↑2 * y
7	<id> - <factor>	x - ↑2 * y
9	<id> - <num>	x - ↑2 * y
–	<id> - <num>	x - 2 ↑* y
–	<id> - <term>	x - ↑2 * y
5	<id> - <term> * <factor>	x - ↑2 * y
7	<id> - <factor> * <factor>	x - ↑2 * y
9	<id> - <num> * <factor>	x - ↑2 * y
–	<id> - <num> * <factor>	x - 2 ↑* y
–	<id> - <num> * <factor>	x - 2 * ↑y
9	<id> - <num> * <id>	x - 2 * ↑y
–	<id> - <num> * <id>	x - 2 * y↑

Example

Another possible parse for `x - 2 * y`

Prod'n	Sentential form	Input
–	<goal>	↑`x - 2 * y`
1	<expr>	↑`x - 2 * y`
2	<expr> + <term>	↑`x - 2 * y`
2	<expr> + <term> + <term>	↑`x - 2 * y`
2	<expr> + <term> + ⋯	↑`x - 2 * y`
2	<expr> + <term> + ⋯	↑`x - 2 * y`
2	⋯	↑`x - 2 * y`

If the parser makes the wrong choices, the expansion doesn't terminate. This isn't a good property for a parser to have. (Parsers should terminate!)

The key is selecting the right production in step 1.

⇒ should be guided by input string

Question: How much of the input string would we have to look at in order to make the "right choices" for non-terminal <expr>?

Example: `x + x + x + x + x`.

Left Recursion

Top-down parsers cannot handle left-recursion in a grammar.

Formally,

> a grammar is *left recursive* if $\exists\ A \in NT$ such that $\exists$ a derivation $A \Rightarrow^{+} A\alpha$ for some string α.

Our simple expression grammar is left recursive.

Eliminating immediate left recursion

To remove ***immediate*** *left recursion, we can transform the grammar.*

Consider the grammar fragment:

$$\begin{array}{rcl} \text{<foo>} & ::= & \text{<foo>}\ \alpha \\ & \mid & \beta \end{array}$$

where α and β do not start with <foo>.

We can rewrite this as:

$$\begin{array}{rcl} \text{<foo>} & ::= & \beta\ \text{<bar>} \\ \text{<bar>} & ::= & \alpha\ \text{<bar>} \\ & \mid & \epsilon \end{array}$$

where <bar> is a new non-terminal.

This fragment contains no left recursion.

Example

Our expression grammar contains two cases of left recursion

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{expr}> \texttt{+} <\text{term}> \\
 & | & <\text{expr}> \texttt{-} <\text{term}> \\
 & | & <\text{term}> \\
<\text{term}> & ::= & <\text{term}> \texttt{*} <\text{factor}> \\
 & | & <\text{term}> \texttt{/} <\text{factor}> \\
 & | & <\text{factor}>
\end{array}
$$

Applying the transformation gives

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{term}> <\text{expr}'> \\
<\text{expr}'> & ::= & \texttt{+} <\text{term}> <\text{expr}'> \\
 & | & \texttt{-} <\text{term}> <\text{expr}'> \\
 & | & \epsilon \\
<\text{term}> & ::= & <\text{factor}> <\text{term}'> \\
<\text{term}'> & ::= & \texttt{*} <\text{factor}> <\text{term}'> \\
 & | & \texttt{/} <\text{factor}> <\text{term}'> \\
 & | & \epsilon
\end{array}
$$

With this grammar, a top-down parser will terminate

Eliminating left recursion

A general technique for removing left recursion

arrange the non-terminals in some order
    $A_1, A_2, \ldots, A_n$
for i $\leftarrow$ 1 to n
    for j $\leftarrow$ 1 to i-1
        replace each production of the form
            $A_i ::= A_j\gamma$ with the productions
            $A_i ::= \delta_1\gamma \ \mid \ \delta_2\gamma \ \mid \ \ldots \ \mid \ \delta_k\gamma$,
            where $A_j ::= \delta_1 \ \mid \ \delta_2 \ \mid \ \ldots \ \mid \ \delta_k$
            are all the current A_j productions.
    endfor
    eliminate any <u>immediate</u> left recursion on A_i
        using the direct transformation
endfor

This assumes that the grammar has no cycles
$(A \Rightarrow^+ A)$ or ϵ productions $(A ::= \epsilon)$.

Aho, Sethi, and Ullman, Figure 4.7

Eliminating left recursion

How does this algorithm work?

1. impose an arbitrary order on the non-terminals
2. outer loop cycles through NT in order
3. inner loop ensures that a production expanding A_i has no non-terminal A_j with $j < i$ as the first symbol on its right hand side (it forward substitutes those away)
4. last step in the outer loop converts any direct recursion on A_i to right recursion using the simple transformation showed earlier
5. new non-terminals are added at the end of the order and only involve right recursion

At the start of the i^{th} outer loop iteration

> for all $k < i$, $\nexists$ a production $A_k ::= A_l \ \alpha$
> such that $l \leq k$.

At the end of the process ($n = i$), the grammar has no remaining left recursion.

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

  Aho, Hopcroft, and Ullman, Problem 2.34

  Parsing, Translation and Compiling, Chapter 4

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are LL(1) and LR(1).

Predictive Parsing

Basic idea:

> For any two productions $A \ ::= \ \alpha \ | \ \beta$, we would like a distinct way of choosing the correct production to expand.

For some *rhs* $\alpha \ \in \ G$, define **FIRST**(α) as the set of tokens that appear as the first symbol in some string derived from α.

That is

$x \ \in \mathrm{FIRST}(\alpha)$ *iff* $\alpha \ \Rightarrow^* \ x\gamma$ for some γ, and

$\epsilon \ \in \mathrm{FIRST}(\alpha)$ *iff* $\alpha \ \Rightarrow^* \ \epsilon$.

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it

A terminal symbol has no FOLLOW set

Predictive Parsing (cont.)

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first condition, at most one of α or β can derive ϵ.

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example grammar has this property!

Example: Problem?

What if a grammar does not have this property?

Consider a *right-recursive* version of the expression grammar:

1	<goal>	::=	<expr>
2	<expr>	::=	<term> `+` <expr>
3		$\mid$	<term> `-` <expr>
4		$\mid$	<term>
5	<term>	::=	<factor> `*` <term>
6		$\mid$	<factor> `/` <term>
7		$\mid$	<factor>
8	<factor>	::=	`num`
9		$\mid$	`id`

To choose between productions 2, 3, & 4, the parser must see past the `num` or `id` and look at the `+`, `-`, `*`, `/`, or `eof`.

$$\mathit{FIRST}(2) \; \cap \; \mathit{FIRST}(3) \; \cap \; \mathit{FIRST}(4) \neq \emptyset$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Left Factoring

Sometimes, we can transform a grammar to have the desired property.

> For each non-terminal A find the longest prefix α common to two or more of its alternatives.
>
> if $\alpha \neq \epsilon$, then replace all of the A productions
> $A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$
> with
> $$A ::= \alpha L \mid \gamma$$
> $$L ::= \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$
> where L is a new non-terminal.
>
> Repeat until no two alternatives for a single non-terminal have a common prefix.

Aho, Sethi, and Ullman, Algorithm 4.2

Example

There are two nonterminals that must be left factored:

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{term}>\ \texttt{+}\ <\text{expr}> \\
 & | & <\text{term}>\ \texttt{-}\ <\text{expr}> \\
 & | & <\text{term}> \\
<\text{term}> & ::= & <\text{factor}>\ \texttt{*}\ <\text{term}> \\
 & | & <\text{factor}>\ \texttt{/}\ <\text{term}> \\
 & | & <\text{factor}>
\end{array}
$$

Applying the transformation gives us:

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{term}>\ <\text{expr}'> \\
<\text{expr}'> & ::= & \texttt{+}\ <\text{expr}> \\
 & | & \texttt{-}\ <\text{expr}> \\
 & | & \epsilon \\
<\text{term}> & ::= & <\text{factor}>\ <\text{term}'> \\
<\text{term}'> & ::= & \texttt{*}\ <\text{term}> \\
 & | & \texttt{/}\ <\text{term}> \\
 & | & \epsilon
\end{array}
$$

Example

Substituting back into the grammar yields

1	<goal>	::=	<expr>
2	<expr>	::=	<term> <expr′ >
3	<expr′ >	::=	`+` <expr>
4		|	`-` <expr>
5		|	ϵ
6	<term>	::=	<factor> <term′ >
7	<term′ >	::=	`*` <term>
8		|	`/` <term>
9		|	ϵ
10	<factor>	::=	`num`
11		|	`id`

Now, selection requires only a single token lookahead.

LL(1) parse table for example

	id	num	+	-	*	/	eof
$\langle$goal$\rangle$	$g \rightarrow e$	$g \rightarrow e$	–	–	–	–	–
$\langle$expr$\rangle$	$e \rightarrow te'$	$e \rightarrow te'$	–	–	–	–	–
$\langle$expr$'\rangle$	–	–	$e' \rightarrow \texttt{+}e$	$e' \rightarrow \texttt{-}e$	–	–	$e' \rightarrow \epsilon$
$\langle$term$\rangle$	$t \rightarrow ft'$	$t \rightarrow ft'$	–	–	–	–	–
$\langle$term$'\rangle$	–	–	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow \texttt{*}t$	$t' \rightarrow \texttt{/}t$	$t' \rightarrow \epsilon$
$\langle$factor$\rangle$	$f \rightarrow$ id	$f \rightarrow$ num	–	–	–	–	–

Symbol	FIRST	FOLLOW
<goal>	{ id,number }	{ eof }
<expr>	{ id,number }	{ eof }
<expr$'$ >	{ ϵ, +, - }	{ eof }
<term>	{ id,number }	{ eof,+,- }
<term$'$ >	{ ϵ, *, / }	{ eof,+,- }
<factor>	{ id,number }	{ eof,+,-,*,/ }
+	{ + }	—
-	{ - }	—
*	{ * }	—
/	{ / }	—
id	{ id }	—
number	{ number }	—