

Top-down versus bottom-up

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*)

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

Top-down parsing

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

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

1. At a node labeled A , select a production with A on its *lhs* and for each symbol on its *rhs*, construct the appropriate child.
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)

The key is selecting the right production in step 1.

⇒ should be guided by input string

Simple expression grammar

Recall our grammar for simple expressions:

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle + \langle \text{term} \rangle$ $\langle \text{expr} \rangle - \langle \text{term} \rangle$
3			$\langle \text{term} \rangle$
4	$\langle \text{term} \rangle$	$::=$	$\langle \text{term} \rangle * \langle \text{factor} \rangle$ $\langle \text{term} \rangle / \langle \text{factor} \rangle$
5			$\langle \text{factor} \rangle$
6			number
7			id
8	$\langle \text{factor} \rangle$	$::=$	number
9			id

Consider the input string $x - 2 * y$

Example

Prod'n	Sentential form	Input
-	$\langle \text{goal} \rangle$	$\uparrow x - 2 * y$
1	$\langle \text{expr} \rangle$	$\uparrow x - 2 * y$
2	$\langle \text{expr} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
4	$\langle \text{term} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
7	$\langle \text{factor} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
9	$\langle \text{id} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
-	$\langle \text{id} \rangle + \langle \text{term} \rangle$	$x \uparrow - 2 * y$
3	$\langle \text{expr} \rangle$	$\uparrow x - 2 * y$
4	$\langle \text{expr} \rangle - \langle \text{term} \rangle$	$\uparrow x - 2 * y$
7	$\langle \text{term} \rangle - \langle \text{term} \rangle$	$\uparrow x - 2 * y$
9	$\langle \text{factor} \rangle - \langle \text{term} \rangle$	$\uparrow x - 2 * y$
-	$\langle \text{id} \rangle - \langle \text{term} \rangle$	$\uparrow x - 2 * y$
-	$\langle \text{id} \rangle - \langle \text{term} \rangle$	$x \uparrow - 2 * y$
5	$\langle \text{id} \rangle - \langle \text{term} \rangle * \langle \text{factor} \rangle$	$x - \uparrow 2 * y$
7	$\langle \text{id} \rangle - \langle \text{factor} \rangle * \langle \text{factor} \rangle$	$x - \uparrow 2 * y$
9	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{factor} \rangle$	$x - \uparrow 2 * y$
-	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{factor} \rangle$	$x - 2 \uparrow * y$
-	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{factor} \rangle$	$x - 2 \uparrow * y$
-	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{factor} \rangle$	$x - 2 * \uparrow y$
9	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{id} \rangle$	$x - 2 * \uparrow y$
-	$\langle \text{id} \rangle - \langle \text{num} \rangle * \langle \text{id} \rangle$	$x - 2 * y \uparrow$

Example

Another possible parse for $x - 2 * y$

Prod'n	Sentential form	Input
-	<goal>	$\uparrow x - 2 * y$
1	<expr>	$\uparrow x - 2 * y$
2	<expr> + <term>	$\uparrow x - 2 * y$
2	<expr> + <term> + <term>	$\uparrow x - 2 * y$
2	<expr> + <term> + ...	$\uparrow x - 2 * y$
2	<expr> + <term> + ...	$\uparrow x - 2 * y$
2	...	$\uparrow x - 2 * y$

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!)

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 left recursion

To remove left recursion, we can transform the grammar.

Consider the grammar fragment:

$$\begin{aligned} \langle \text{foo} \rangle &::= \langle \text{foo} \rangle \alpha \\ &| \beta \end{aligned}$$

where α and β do not start with $\langle \text{foo} \rangle$.

We can rewrite this as:

$$\begin{aligned} \langle \text{foo} \rangle &::= \beta \langle \text{bar} \rangle \\ \langle \text{bar} \rangle &::= \alpha \langle \text{bar} \rangle \\ &| \epsilon \end{aligned}$$

where $\langle \text{bar} \rangle$ is a new non-terminal.

This fragment contains no left recursion.

Example

Our expression grammar contains two cases of left recursion

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{expr} \rangle + \langle \text{term} \rangle \\ &| \langle \text{expr} \rangle - \langle \text{term} \rangle \\ &| \langle \text{term} \rangle \\ \langle \text{term} \rangle &::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\ &| \langle \text{term} \rangle / \langle \text{factor} \rangle \\ &| \langle \text{factor} \rangle \end{aligned}$$

Applying the transformation gives

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\ \langle \text{expr}' \rangle &::= + \langle \text{term} \rangle \langle \text{expr}' \rangle \\ &| \epsilon \\ \langle \text{term} \rangle &::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\ \langle \text{term}' \rangle &::= * \langle \text{factor} \rangle \langle \text{term}' \rangle \\ &| \epsilon \\ &| / \langle \text{factor} \rangle \langle \text{term}' \rangle \end{aligned}$$

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Eliminating left recursion

A general technique for removing left recursion

arrange the non-terminals in some order

A_1, A_2, \dots, 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

$A_i ::= \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma,$

where $A_j ::= \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$

are all the current A_j productions.

eliminate any immediate left recursion on A_i

using the direct transformation

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

Also, 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_j has no non-terminal A_j with $j < i$
4. It forward substitutes those away
5. last step in the outer loop converts any direct recursion on A_i to right recursion using the simple transformation showed earlier
6. 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 expanding A_k
that has A_l in its rhs, for $l < k$.

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

Example grammar

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle \langle \text{expr}' \rangle$
3	$\langle \text{expr}' \rangle$	$::=$	$+$ $\langle \text{term} \rangle \langle \text{expr}' \rangle$
4		$ $	$-$ $\langle \text{term} \rangle \langle \text{expr}' \rangle$
5		$ $	ϵ
6	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
7	$\langle \text{term}' \rangle$	$::=$	$*$ $\langle \text{factor} \rangle \langle \text{term}' \rangle$
8		$ $	$/$ $\langle \text{factor} \rangle \langle \text{term}' \rangle$
9		$ $	ϵ
10	$\langle \text{factor} \rangle$	$::=$	number
11		$ $	id

Transformed to eliminate 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

Alho, 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).

Recursive Descent Parsing

Properties

- top-down parsing algorithm
- parser built on procedure calls
- procedures may be (mutually) recursive

Algorithm

- write procedure for each non-terminal
- turn each production into clause
- insert call
 - to procedure A() for non-terminal A
 - to match(x) for terminal x
- start by invoking procedure for start symbol S

Example

```
A ::= a B c
⇒ A() { match(a); B(); match(c); }
```

Recursive Descent Parsing

Example grammar

1		S ::= a A
2		b
3		A ::= S c

Helpers

Parser

```
tok; // current token
match(x) {
  if (tok != x)
    error();
  tok = getToken();
}

S() {
  if (tok == a)
    match(a); A();
  else if (tok == b)
    match(b);
  else error();
}

A() {
  S(); match(c);
}
```

Predictive Parsing

Basic idea

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

FIRST sets

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

That is, $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some γ .

LL(1) property

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

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \epsilon$$

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

Pursuing this idea leads to *predictive LL(1)* parsers.

Left Factoring

What if a grammar does not have the LL(1) property?

Sometimes, we can transform a grammar to have this 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 $A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ with

$$A ::= \alpha L \mid \gamma$$

$$L ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

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

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

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle + \langle \text{expr} \rangle$
3			$\langle \text{term} \rangle - \langle \text{expr} \rangle$
4			$\langle \text{term} \rangle$
5	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle * \langle \text{term} \rangle$
6			$\langle \text{factor} \rangle / \langle \text{term} \rangle$
7			$\langle \text{factor} \rangle$
8	$\langle \text{factor} \rangle$	$::=$	number
9			id

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

$$FIRST(2) \cap FIRST(3) \cap FIRST(4) \neq \emptyset$$

This grammar fails the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals that must be left factored:

$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle + \langle \text{expr} \rangle$
		$\langle \text{term} \rangle - \langle \text{expr} \rangle$
		$\langle \text{term} \rangle$
$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle * \langle \text{term} \rangle$
		$\langle \text{factor} \rangle / \langle \text{term} \rangle$
		$\langle \text{factor} \rangle$

Applying the transformation gives us:

$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle \langle \text{expr}' \rangle$
$\langle \text{expr}' \rangle$	$::=$	$+ \langle \text{expr} \rangle$
		$- \langle \text{expr} \rangle$
		ϵ
$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
$\langle \text{term}' \rangle$	$::=$	$* \langle \text{term} \rangle$
		$/ \langle \text{term} \rangle$
		ϵ

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>	::=	number
11			id

Now, selection requires only a single token lookahead.

Note: *This grammar is still right-associative.*

Example:

	Sentential form	Input
—	<goal>	x - 2 * y
1	<expr>	↑x - 2 * y
2	<term> <expr'>	↑x - 2 * y
6	<factor> <term'> <expr'>	↑x - 2 * y
11	<id> <term'> <expr'>	↑x - 2 * y
—	<id> <term'> <expr'>	x ↑ - 2 * y
9	<id> ϵ <expr'>	x ↑ - 2
4	<id> - <expr>	x ↑ - 2 * y
—	<id> - <expr>	x - ↑2 * y
2	<id> - <term> <expr'>	x - ↑2 * y
6	<id> - <factor> <term'> <expr'>	x - ↑2 * y
10	<id> - <num> <term'> <expr'>	x - ↑2 * y
—	<id> - <num> <term'> <expr'>	x - 2 ↑ * y
7	<id> - <num> * <term> <expr'>	x - 2 ↑ * y
—	<id> - <num> * <term> <expr'>	x - 2 * ↑y
6	<id> - <num> * <factor> <term'> <expr'>	x - 2 * ↑y
11	<id> - <num> * <id> <expr'>	x - 2 * ↑y
—	<id> - <num> * <id> <term'> <expr'>	x - 2 * y ↑
9	<id> - <num> * <id> <expr'>	x - 2 * y ↑
5	<id> - <num> * <id>	x - 2 * y ↑

The next symbol determined each choice correctly.

Generality

Question:

By eliminating left recursion and left factoring, can we transform an arbitrary context free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

Given a context free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many context free languages do not have such a grammar.

$$\{a^n0b^n \mid n \geq 1\} \cup \{a^n1b^{2n} \mid n \geq 1\}$$