

## Types of parsing

---

### *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

## Example grammar

---

This is a grammar for simple expressions:

1	<goal> ::= <expr>
2	<expr> ::= <expr> + <term>
3	<expr> - <term>
4	<term>
5	<term> ::= <term> * <factor>
6	<term> / <factor>
7	<factor>
8	<factor> ::= <b>number</b>
9	<b>id</b>

Consider parsing the input string  $x - 2 * y$

## Backtracking parse example

---

One possible parse for  $x - 2 * y$

Prod'n	Sentential form	Input
-	<goal>	↑x - 2 * y
1	<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
-	$\langle \text{goal} \rangle$	$\uparrow x - 2 * y$
1	$\langle \text{expr} \rangle$	$\uparrow x - 2 * y$
2	$\langle \text{expr} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
2	$\langle \text{expr} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
2	$\langle \text{expr} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle$	$\uparrow x - 2 * y$
2	$\langle \text{expr} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle + \langle \text{term} \rangle + \dots$	$\uparrow x - 2 * y$
2	$\dots$	$\uparrow x - 2 * y$

If the parser makes the wrong choices, the expansion doesn't terminate.

This isn't a good property for a parser to have.

## 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  $\alpha$ .

*Our simple expression grammar is left recursive.*

## Eliminating left recursion

---

To remove left recursion, we can transform the grammar.

Consider the grammar fragment:

$$\begin{array}{l} \langle \text{foo} \rangle ::= \langle \text{foo} \rangle \alpha \\ \quad \quad | \beta \end{array}$$

where  $\alpha$  and  $\beta$  do not start with  $\langle \text{foo} \rangle$ .

We can rewrite this as:

$$\begin{array}{l} \langle \text{foo} \rangle ::= \beta \langle \text{bar} \rangle \\ \langle \text{bar} \rangle ::= \alpha \langle \text{bar} \rangle \\ \quad \quad | \epsilon \end{array}$$

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

$$\begin{array}{l} \langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \\ \quad \quad | \langle \text{expr} \rangle - \langle \text{term} \rangle \\ \quad \quad | \langle \text{term} \rangle \\ \langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \\ \quad \quad | \langle \text{term} \rangle / \langle \text{factor} \rangle \\ \quad \quad | \langle \text{factor} \rangle \end{array}$$

Applying the transformation gives

$$\begin{array}{l} \langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\ \langle \text{expr}' \rangle ::= + \langle \text{term} \rangle \langle \text{expr}' \rangle \\ \quad \quad | - \langle \text{term} \rangle \langle \text{expr}' \rangle \\ \quad \quad | \epsilon \\ \langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\ \langle \text{term}' \rangle ::= * \langle \text{factor} \rangle \langle \text{term}' \rangle \\ \quad \quad | / \langle \text{factor} \rangle \langle \text{term}' \rangle \\ \quad \quad | \epsilon \end{array}$$

## 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  $\epsilon$  productions ( $A ::= \epsilon$ ).

### 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  $j < i$
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.

## 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

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  $\text{match}(\mathbf{x})$  for terminal  $\mathbf{x}$
- start by invoking procedure for start symbol  $S$

Example

$A ::= a B c \quad \Rightarrow A() \{ \text{match}(a); B(); \text{match}(c); \}$

## Recursive Descent Parsing

---

Example grammar

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

Helpers

```
tok; // current token

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

Parser

```
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  $FIRST(\alpha)$  as the set of tokens that appear as the first symbol in some string derived from  $\alpha$ .

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

LL(1) property

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

$$FIRST(\alpha) \cap FIRST(\beta) = \epsilon$$

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  $\alpha$  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.

### Example

---

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

1		<goal>	::=	<expr>
2		<expr>	::=	<term> + <expr>
3				<term> - <expr>
4				<term>
5		<term>	::=	<factor> * <term>
6				<factor> / <term>
7				<factor>
8		<factor>	::=	<b>number</b>
9				<b>id</b>

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:

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

Applying the transformation gives us:

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

## Example

---

Substituting back into the grammar yields

$$\begin{array}{l|l} 1 & \langle \text{goal} \rangle ::= \langle \text{expr} \rangle \\ 2 & \langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\ 3 & \langle \text{expr}' \rangle ::= + \langle \text{expr} \rangle \\ 4 & \quad | - \langle \text{expr} \rangle \\ 5 & \quad | \epsilon \\ 6 & \langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\ 7 & \langle \text{term}' \rangle ::= * \langle \text{term} \rangle \\ 8 & \quad | / \langle \text{term} \rangle \\ 9 & \quad | \epsilon \\ 10 & \langle \text{factor} \rangle ::= \mathbf{number} \\ 11 & \quad | \mathbf{id} \end{array}$$

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.

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

For a string of grammar symbols  $\alpha$ , define  $\text{FIRST}(\alpha)$  as

- the set of terminal symbols that begin strings derived from  $\alpha$
- if  $\alpha \Rightarrow^* \epsilon$ , then  $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$  contains the set of tokens valid in the first position of  $\alpha$

To build  $\text{FIRST}(X)$ :

1. if  $X$  is a terminal,  $\text{FIRST}(X)$  is  $\{X\}$
2. if  $X ::= \epsilon$ , then  $\epsilon \in \text{FIRST}(X)$
3. if  $X ::= Y_1 Y_2 \cdots Y_k$  then put  $\text{FIRST}(Y_1)$  in  $\text{FIRST}(X)$
4. if  $X$  is a non-terminal and  $X ::= Y_1 Y_2 \cdots Y_k$ , then  $a \in \text{FIRST}(X)$  if  $a \in \text{FIRST}(Y_i)$  and  $\epsilon \in \text{FIRST}(Y_j)$  for all  $1 \leq j < i$   
(If  $\epsilon \notin \text{FIRST}(Y_1)$ , then  $\text{FIRST}(Y_i)$  is irrelevant, for  $1 < i$ )

*Our 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		$\langle \text{expr}' \rangle ::= + \langle \text{expr} \rangle$
4		$- \langle \text{expr} \rangle$
5		$\epsilon$
6		$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle$
7		$\langle \text{term}' \rangle ::= * \langle \text{term} \rangle$
8		$/ \langle \text{term} \rangle$
9		$\epsilon$
10		$\langle \text{factor} \rangle ::= \text{num}$
11		$\text{id}$

<i>rule</i>	1	2	3	4	<i>FIRST</i>
<i>goal</i>	–	–	num, id	–	{num, id}
<i>expr</i>	–	–	num, id	–	{num, id}
<i>expr'</i>	–	$\epsilon$	+, –	–	{ $\epsilon$ , +, –}
<i>term</i>	–	–	num, id	–	{num, id}
<i>term'</i>	–	$\epsilon$	*, /	–	{ $\epsilon$ , *, /}
<i>factor</i>	–	–	num, id	–	{num, id}
num	num	–	–	–	{num}
id	id	–	–	–	{id}
+	+	–	–	–	{+}
–	–	–	–	–	{–}
*	*	–	–	–	{*}
/	/	–	–	–	{/}

The FOLLOW set

---

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

To build FOLLOW( $X$ ):

1. place **eof** in FOLLOW( $\langle$ goal $\rangle$ )
2. if  $A ::= \alpha B \beta$ , then put  $\{FIRST(\beta) - \epsilon\}$  in FOLLOW( $B$ )
3. if  $A ::= \alpha B$  then put FOLLOW( $A$ ) in FOLLOW( $B$ )
4. if  $A ::= \alpha B \beta$  and  $\epsilon \in FIRST(\beta)$ , then put FOLLOW( $A$ ) in FOLLOW( $B$ )

<i>rule</i>	1	2	3	4	<i>FOLLOW</i>
<i>goal</i>	eof	-	-	-	{eof}
<i>expr</i>	-	-	eof	-	{eof}
<i>expr'</i>	-	-	eof	-	{eof}
<i>term</i>	-	+,-	-	eof	{eof,+,-}
<i>term'</i>	-	-	eof,+,-	-	{eof,+,-}
<i>factor</i>	-	*,/	-	eof,+,-	{eof,+,-,*,/}

Using FIRST and FOLLOW

---

To build a predicative recursive-descent parser:

For each production  $A ::= \alpha$  and lookahead *token*

- expand  $A$  using production if  $token \in \text{FIRST}(\alpha)$
- if  $\epsilon \in \text{FIRST}(\alpha)$  expand  $A$  using production if  $token \in \text{FOLLOW}(A)$
- all other tokens return *error*

If multiple choices, the grammar is not  $LL(1)$  (predicative).

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

### Features

- input parsed from left to right
- leftmost derivation
- one token lookahead

### Definition

A grammar  $G$  is  $LL(1)$  if and only if, for all non-terminals  $A$ , each distinct pair of productions  $A ::= \beta$  and  $A ::= \gamma$  satisfy the condition  $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \emptyset$

A grammar  $G$  is  $LL(1)$  if and only if for each set of productions  $A ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

1.  $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \dots, \text{FIRST}(\alpha_n)$  are all pairwise disjoint
2. if  $\alpha_i \Rightarrow^* \epsilon$ , then  $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ , for all  $1 \leq j \leq n, i \neq j$ .

If  $G$  is  $\epsilon$ -free, condition 1 is sufficient.

### Provable facts about $LL(1)$ grammars:

- no left recursive grammar is  $LL(1)$
- no ambiguous grammar is  $LL(1)$
- $LL(1)$  parsers operate in linear time
- an  $\epsilon$ -free grammar where each alternative expansion for  $A$  begins with a distinct terminal is a *simple*  $LL(1)$  grammar

### Not all grammars are $LL(1)$

- $S ::= aS \mid a$   
is not  $LL(1)$   
 $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$
- $S ::= aS'$   
 $S' ::= aS' \mid \epsilon$   
accepts the same language and is  $LL(1)$

*LL(1) grammars*

- may need to rewrite grammar (left recursion, left factoring)
- resulting grammar larger, less maintainable

*LL(k) grammars*

- $k$ -token lookahead, more powerful than *LL(1)* grammars
- example:  
 $S ::= ac \mid abc$  is *LL(2)*

Not all grammars are *LL(k)*

- example:  
 $S ::= a^i b^j$  where  $i \geq j$
- equivalent to dangling else problem
- problem - must choose production after  $k$  tokens of lookahead

Bottom-up parsers avoid this problem