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 \(\text{recognize valid prefixes}\)
- 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 labelled 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 labelled $A$, 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  \texttt{<goal>} ::= \texttt{<expr>}
2  \texttt{<expr>} ::= \texttt{<expr>} + \texttt{<term>}
3  \quad \mid \texttt{<expr>} - \texttt{<term>}
4  \quad \mid \texttt{<term>}
5  \texttt{<term>} ::= \texttt{<term>} \ast \texttt{<factor>}
6  \quad \mid \texttt{<term>} / \texttt{<factor>}
7  \quad \mid \texttt{<factor>}
8  \texttt{<factor>} ::= \texttt{number}
9  \quad \mid \texttt{id}

Consider the input string \texttt{x - 2 * y}
## Example

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>&lt;goal&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; + &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;factor&gt; + &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; + &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; + &lt;term&gt;</td>
<td>(x \uparrow - 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;expr&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>3</td>
<td>&lt;expr&gt; - &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; - &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;factor&gt; - &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(\uparrow x - 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(x \uparrow - 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>(x - 2 \uparrow\ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id&gt; - &lt;term&gt; \ast &lt;factor&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt; \ast &lt;factor&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt; \ast &lt;factor&gt;</td>
<td>(x - \uparrow 2 \ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; \ast &lt;factor&gt;</td>
<td>(x - 2 \uparrow\ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; \ast &lt;factor&gt;</td>
<td>(x - 2 \uparrow\ast y)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt; \ast &lt;id&gt;</td>
<td>(x - 2 \uparrow\ast y)</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; \ast &lt;id&gt;</td>
<td>(x - 2 \ast y)</td>
</tr>
</tbody>
</table>
Example

Another possible parse for \( x - 2 * y \)

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>&lt;goal&gt;</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt;</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + &lt;term&gt;</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + ⋯</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; + &lt;term&gt; + ⋯</td>
<td>↑( x - 2 * y )</td>
</tr>
<tr>
<td>2</td>
<td>⋯</td>
<td>↑( x - 2 * y )</td>
</tr>
</tbody>
</table>

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 \( \alpha \).

*Our simple expression grammar is left recursive.*
Eliminating left recursion

To remove left recursion, we can transform the grammar.

Consider the grammar fragment:

\[
<\text{foo}> ::= <\text{foo}> \alpha \\
| \beta
\]

where \(\alpha\) and \(\beta\) do not start with \(<\text{foo}>\).

We can rewrite this as:

\[
<\text{foo}> ::= \beta <\text{bar}> \\
<\text{bar}> ::= \alpha <\text{bar}> \\
| \epsilon
\]

where \(<\text{bar}>\) is a new non-terminal.

This fragment contains no left recursion.
Example

Our expression grammar contains two cases of left recursion

\[
\begin{align*}
<\text{expr}> &::= <\text{expr}> + <\text{term}> \\
&\quad | <\text{expr}> - <\text{term}> \\
&\quad | <\text{term}>
\end{align*}
\]

\[
\begin{align*}
<\text{term}> &::= <\text{term}> \times <\text{factor}> \\
&\quad | <\text{term}> / <\text{factor}> \\
&\quad | <\text{factor}>
\end{align*}
\]

Applying the transformation gives

\[
\begin{align*}
<\text{expr}> &::= <\text{term}> <\text{expr}'> \\
<\text{expr}'> &::= + <\text{term}> <\text{expr}'> \\
&\quad | \epsilon \\
&\quad | - <\text{term}> <\text{expr}'> \\
<\text{term}> &::= <\text{factor}> <\text{term}>'
\end{align*}
\]

\[
\begin{align*}
<\text{term}>' &::= \times <\text{factor}> <\text{term}>' \\
&\quad | \epsilon \\
&\quad | / <\text{factor}> <\text{term}>'
\end{align*}
\]

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs
Example

A temptation is to clean up the grammar like this instead:

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

This grammar

- accepts the same language
- uses right recursion
- has no \( \epsilon \) productions

*Unfortunately, it generates different associativity*

*Same syntax, different meaning*
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.
    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 ($A ::= \epsilon$).

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 $j < i$
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 $k < i$, $\not\exists$ 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  \( <\text{goal}> ::= <\text{expr}> \)
2  \( <\text{expr}> ::= <\text{term}> <\text{expr'}> \)
3  \( <\text{expr'}> ::= + <\text{term}> <\text{expr'}> \)
4  \( \quad | - <\text{term}> <\text{expr'}> \)
5  \( \quad | \epsilon \)
6  \( <\text{term}> ::= <\text{factor}> <\text{term'}> \)
7  \( <\text{term'}> ::= \ast <\text{factor}> <\text{term'}> \)
8  \( \quad | / <\text{factor}> <\text{term'}> \)
9  \( \quad | \epsilon \)
10 \( <\text{factor}> ::= \text{number} \)
11 \( \quad | \text{id} \)

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

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 \( A ::= \alpha \mid \beta \), we would like a distinct way of choosing the correct production to expand.

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 \( \alpha \).

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

Key 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!

The example grammar has this property!
Left Factoring

What if a grammar does not have this 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.

Aho, Sethi, and Ullman, Algorithm 4.2
Example

Consider a right-recursive version of the expression grammar:

1. \(<\text{goal}> \ ::= \ <\text{expr}>\)
2. \(<\text{expr}> \ ::= \ <\text{term}> + \ <\text{expr}>\)
3. \(<\text{term}> \ ::= \ <\text{term}> - \ <\text{expr}>\)
4. \(<\text{term}> \ ::= \ <\text{term}>\)
5. \(<\text{term}> \ ::= \ <\text{factor}> \ast \ <\text{term}>\)
6. \(<\text{term}> \ ::= \ <\text{factor}> / \ <\text{term}>\)
7. \(<\text{factor}> \ ::= \ <\text{factor}>\)
8. \(<\text{factor}> \ ::= \ \text{number}\)
9. \(<\text{factor}> \ ::= \ \text{id}\)

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

\[
\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \emptyset
\]

This grammar fails the test.

Note: \text{This grammar is right-associative}.\]
Example

There are two nonterminals that must be left factored:

\[
<\text{expr}> ::= <\text{term}> + <\text{expr}>
\]
\[
| \quad <\text{term}> - <\text{expr}>
\]
\[
| \quad <\text{term}>
\]

\[
<\text{term}> ::= <\text{factor}>' <\text{term}>
\]
\[
| \quad <\text{factor}>' / <\text{term}>
\]
\[
| \quad <\text{factor}>'
\]

Applying the transformation gives us:

\[
<\text{expr}> ::= <\text{term}>' <\text{expr}>'
\]
\[
<\text{expr}>' ::= + <\text{expr}>
\]
\[
| \quad - <\text{expr}>
\]
\[
| \quad \epsilon
\]

\[
<\text{term}>' ::= <\text{factor}>' <\text{term}>'
\]
\[
<\text{term}>' ::= * <\text{term}>
\]
\[
| \quad / <\text{term}>
\]
\[
| \quad \epsilon
\]
Example

Substituting back into the grammar yields

\[
\begin{align*}
1 & \quad <\text{goal}> & ::= & <\text{expr}> \\
2 & \quad <\text{expr}> & ::= & <\text{term}> <\text{expr}'> \\
3 & \quad <\text{expr}'> & ::= & + <\text{expr}> \\
4 & \quad & | & - <\text{expr}> \\
5 & \quad & | & \epsilon \\
6 & \quad <\text{term}> & ::= & <\text{factor}> <\text{term}'> \\
7 & \quad <\text{term}>' & ::= & * <\text{term}> \\
8 & \quad & | & / <\text{term}> \\
9 & \quad & | & \epsilon \\
10 & \quad <\text{factor}> & ::= & \text{number} \\
11 & \quad & | & \text{id}
\end{align*}
\]

Now, selection requires only a single token lookahead.

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

<table>
<thead>
<tr>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;goal&gt;</td>
<td>( \uparrow x - 2 \ast y )</td>
</tr>
<tr>
<td>1 &lt;expr&gt;</td>
<td>( \uparrow x - 2 \ast y )</td>
</tr>
<tr>
<td>2 &lt;term&gt; &lt;expr’&gt;</td>
<td>( \uparrow x - 2 \ast y )</td>
</tr>
<tr>
<td>6 &lt;factor&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( \uparrow x - 2 \ast y )</td>
</tr>
<tr>
<td>11 &lt;id&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( \uparrow x - 2 \ast y )</td>
</tr>
<tr>
<td>(-) &lt;id&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>9 &lt;id&gt; (\epsilon) &lt;expr’&gt;</td>
<td>( x \uparrow 2 )</td>
</tr>
<tr>
<td>4 &lt;id&gt; - &lt;expr&gt;</td>
<td>( x \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>(-) &lt;id&gt; - &lt;expr&gt;</td>
<td>( x \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>2 &lt;id&gt; - &lt;term&gt; &lt;expr’&gt;</td>
<td>( x - \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>6 &lt;id&gt; - &lt;factor&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x - \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>10 &lt;id&gt; - &lt;num&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x - \uparrow 2 \ast y )</td>
</tr>
<tr>
<td>(-) &lt;id&gt; - &lt;num&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x - 2 \uparrow \ast y )</td>
</tr>
<tr>
<td>7 &lt;id&gt; - &lt;num&gt; \ast &lt;term&gt; &lt;expr’&gt;</td>
<td>( x - 2 \uparrow \ast y )</td>
</tr>
<tr>
<td>(-) &lt;id&gt; - &lt;num&gt; \ast &lt;term&gt; &lt;expr’&gt;</td>
<td>( x - 2 \ast \uparrow y )</td>
</tr>
<tr>
<td>6 &lt;id&gt; - &lt;num&gt; \ast &lt;factor&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x - 2 \ast \uparrow y )</td>
</tr>
<tr>
<td>11 &lt;id&gt; - &lt;num&gt; \ast &lt;id&gt; &lt;expr’&gt;</td>
<td>( x - 2 \ast \uparrow y )</td>
</tr>
<tr>
<td>(-) &lt;id&gt; - &lt;num&gt; \ast &lt;id&gt; &lt;term’&gt; &lt;expr’&gt;</td>
<td>( x - 2 \ast y \uparrow )</td>
</tr>
<tr>
<td>9 &lt;id&gt; - &lt;num&gt; \ast &lt;id&gt; &lt;expr’&gt;</td>
<td>( x - 2 \ast y \uparrow )</td>
</tr>
<tr>
<td>5 &lt;id&gt; - &lt;num&gt; \ast &lt;id&gt;</td>
<td>( x - 2 \ast y \uparrow )</td>
</tr>
</tbody>
</table>

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.

\[
\{a^n0b^n \mid n \geq 1\} \cup \{a^n1b^{2n} \mid n \geq 1\}
\]
Recursive Descent Parsing

Now, we can produce a simple recursive descent parser from this grammar.

goal:
    token ← next_token();
    if (expr() = ERROR | token ≠ EOF) then
        return ERROR;

expr:
    if (term() = ERROR) then
        return ERROR;
    else return expr_prime();

expr_prime:
    if (token = PLUS) then
        token ← next_token();
        return expr();
    else if (token = MINUS) then
        token ← next_token();
        return expr();
    else return OK;
Recursive Descent Parsing

term:
    if (factor() = ERROR) then
        return ERROR;
    else return term_prime();

term_prime:
    if (token = MULT) then
        token ← next_token();
        return term();
    else if (token = DIV) then
        token ← next_token();
        return term();
    else return OK;

factor:
    if (token = NUM) then
        token ← next_token();
        return OK;
    else if (token = ID) then
        token ← next_token();
        return OK;
    else return ERROR;
Building the Tree

One of the key jobs of the parser is to build an intermediate representation of the source code.

To build an abstract syntax tree, we can simply insert code at the appropriate points:

- `factor()` can stack nodes `id`, `num`
- `term_prime()` can stack nodes `*`, `/`
- `term()` can pop 3, build and push subtree
- `expr_prime()` can stack nodes `+`, `-`
- `expr()` can pop 3, build and push subtree
- `goal()` can pop and return tree