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:

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

Consider parsing the input string \(x - 2 * y\)
### Backtracking parse example

One possible parse for $x - 2 \times y$

<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 \times y</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
<td>↑x - 2 \times y</td>
</tr>
<tr>
<td>3</td>
<td>&lt;expr&gt; - &lt;term&gt;</td>
<td>↑x - 2 \times y</td>
</tr>
<tr>
<td>4</td>
<td>&lt;term&gt; - &lt;term&gt;</td>
<td>↑x - 2 \times y</td>
</tr>
<tr>
<td>7</td>
<td>&lt;factor&gt; - &lt;term&gt;</td>
<td>↑x - 2 \times y</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>↑x - 2 \times y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>x ↑- 2 \times y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt;</td>
<td>x - 2 ↑* y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;term&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id&gt; - &lt;term&gt; * &lt;factor&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id&gt; - &lt;factor&gt; * &lt;factor&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;factor&gt;</td>
<td>x - ↑2 \times y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;factor&gt;</td>
<td>x - 2 ↑* y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;factor&gt;</td>
<td>x - 2 * ↑y</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;id&gt;</td>
<td>x - 2 * ↑y</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id&gt; - &lt;num&gt; * &lt;id&gt;</td>
<td>x - 2 * 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><code>&lt;goal&gt;</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>1</td>
<td><code>&lt;expr&gt;</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;expr&gt; + &lt;term&gt;</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;expr&gt; + &lt;term&gt; + &lt;term&gt;</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;expr&gt; + &lt;term&gt; + &lt;term&gt; + &lt;term&gt;</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td><code>&lt;expr&gt; + &lt;term&gt; + &lt;term&gt; + &lt;term&gt; + ...</code></td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>$\uparrow x - 2 * y$</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.
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:

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

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

We can rewrite this as:

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

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

This fragment contains no left recursion.
Example

Our expression grammar contains two cases of left recursion

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

\[
<\text{term}>::= <\text{term}> * <\text{factor}>
\]

\[
| <\text{term}> / <\text{factor}>
\]

\[
| <\text{factor}>
\]

Applying the transformation gives

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

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

\[
<\text{term'}>::= * <\text{factor}> <\text{term'}>
\]

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

\[
| \epsilon
\]
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 \( (A ::= \epsilon) \).
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.
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 match(x) for terminal x
• start by invoking procedure for start symbol S

Example

\[ A ::= a \ B \ c \Rightarrow A() \{ \text{match}(a); \ B(); \ \text{match}(c); \} \]
Recursive Descent Parsing

Example grammar

1. \( S ::= \text{a A} \)
2. \[ b \]
3. \( A ::= S \text{ c} \)

Helpers

\[
\begin{align*}
tok; \quad &// \text{current token} \\
\text{match}(x) \{ \\
&\quad \text{if } (tok != x) \\
&\qquad \text{error}(); \\
&\qquad \text{tok} = \text{getToken}(); \\
&\} \\
\end{align*}
\]

Parser

\[
\begin{align*}
S() \{ \\
&\quad \text{if } (tok == \text{a}) \\
&\qquad \text{match(a)}; \ A(); \\
&\quad \text{else if } (tok == \text{b}) \\
&\qquad \text{match(b)}; \\
&\qquad \text{else error}(); \\
&\} \\
A() \{ \\
&\quad \text{S(); match(c)}; \\
&\}
\end{align*}
\]
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 \( \alpha \).

That is, \( x \in \text{FIRST}(\alpha) \) 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

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

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 $\alpha$ 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> ::= number`
9. | `id`

To choose between productions 2, 3, & 4, the parser must see past the *number* or *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:** *This grammar is right-associative.*
Example

There are two nonterminals that must be left factored:

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

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

Applying the transformation gives us:

\[
\begin{align*}
\text{<expr>} & ::= \text{<term>} \text{<expr’>}
\end{align*}
\]

\[
\begin{align*}
\text{<expr’>} & ::= + \text{<expr>}
\end{align*}
\]

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

\[
\begin{align*}
& \quad | \quad \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{<term>} & ::= \text{<factor>} \text{<term’>}
\end{align*}
\]

\[
\begin{align*}
\text{<term’>} & ::= \ast \text{<term>}
\end{align*}
\]

\[
\begin{align*}
& \quad | \quad \div \text{<term>}
\end{align*}
\]

\[
\begin{align*}
& \quad | \quad \epsilon
\end{align*}
\]
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:

<table>
<thead>
<tr>
<th>Sentential form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;goal&gt;</td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>1 &lt;expr&gt;</td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>2 &lt;term&gt; &lt;expr'&gt;</td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>6 &lt;factor&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>11 &lt;id&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>- &lt;id&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x ↑- 2 * y</td>
</tr>
<tr>
<td>9 &lt;id&gt; ε &lt;expr'&gt;</td>
<td>x ↑- 2</td>
</tr>
<tr>
<td>4 &lt;id&gt; - &lt;expr&gt;</td>
<td>x ↑- 2 * y</td>
</tr>
<tr>
<td>- &lt;id&gt; - &lt;expr&gt;</td>
<td>x - ↑2 * y</td>
</tr>
<tr>
<td>2 &lt;id&gt; - &lt;term&gt; &lt;expr'&gt;</td>
<td>x - ↑2 * y</td>
</tr>
<tr>
<td>6 &lt;id&gt; - &lt;factor&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x - ↑2 * y</td>
</tr>
<tr>
<td>10 &lt;id&gt; - &lt;num&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x - ↑2 * y</td>
</tr>
<tr>
<td>- &lt;id&gt; - &lt;num&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x - 2 ↑* y</td>
</tr>
<tr>
<td>7 &lt;id&gt; - &lt;num&gt; * &lt;term&gt; &lt;expr'&gt;</td>
<td>x -2 ↑* y</td>
</tr>
<tr>
<td>- &lt;id&gt; - &lt;num&gt; * &lt;term&gt; &lt;expr'&gt;</td>
<td>x -2 * ↑y</td>
</tr>
<tr>
<td>6 &lt;id&gt; - &lt;num&gt; * &lt;factor&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x -2 * ↑y</td>
</tr>
<tr>
<td>11 &lt;id&gt; - &lt;num&gt; * &lt;id&gt; &lt;expr'&gt;</td>
<td>x -2 * ↑y</td>
</tr>
<tr>
<td>- &lt;id&gt; - &lt;num&gt; * &lt;id&gt; &lt;term'&gt; &lt;expr'&gt;</td>
<td>x -2 * y↑</td>
</tr>
<tr>
<td>9 &lt;id&gt; - &lt;num&gt; * &lt;id&gt; &lt;expr'&gt;</td>
<td>x -2 * y↑</td>
</tr>
<tr>
<td>5 &lt;id&gt; - &lt;num&gt; * &lt;id&gt;</td>
<td>x -2 * y↑</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.

\[
\{a^n0b^n \mid n \geq 1\} \cup \{a^n1b^{2n} \mid n \geq 1\}
\]
The \textit{FIRST} set

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 \( \epsilon \not\in \text{FIRST}(Y_1) \), then \( \text{FIRST}(Y_i) \) is irrelevant, for \( 1 < i \))
Our example grammar

1  \langle goal \rangle ::= \langle expr \rangle
2  \langle expr \rangle ::= \langle term \rangle \langle expr' \rangle
3  \langle expr' \rangle ::= + \langle expr \rangle
4  \langle expr' \rangle ::= - \langle expr \rangle
5  \langle expr' \rangle ::= \epsilon
6  \langle term \rangle ::= \langle factor \rangle \langle term' \rangle
7  \langle term' \rangle ::= * \langle term \rangle
8  \langle term' \rangle ::= / \langle term \rangle
9  \langle term' \rangle ::= \epsilon
10 \langle factor \rangle ::= \text{num}
11 \langle factor \rangle ::= \text{id}
## The FIRST construction

<table>
<thead>
<tr>
<th>rule</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td></td>
<td></td>
<td>num,id</td>
<td></td>
<td>{num,id}</td>
</tr>
<tr>
<td>expr</td>
<td></td>
<td></td>
<td>num,id</td>
<td></td>
<td>{num,id}</td>
</tr>
<tr>
<td>expr'</td>
<td></td>
<td>ε</td>
<td>+,-</td>
<td></td>
<td>{ε,+,-}</td>
</tr>
<tr>
<td>term</td>
<td></td>
<td></td>
<td>num,id</td>
<td></td>
<td>{num,id}</td>
</tr>
<tr>
<td>term'</td>
<td></td>
<td>ε</td>
<td>*,/</td>
<td></td>
<td>{ε,*,/}</td>
</tr>
<tr>
<td>factor</td>
<td></td>
<td></td>
<td>num,id</td>
<td></td>
<td>{num,id}</td>
</tr>
<tr>
<td>num</td>
<td>num</td>
<td></td>
<td></td>
<td></td>
<td>{num}</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td>{id}</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>{+}</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td>{−}</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>{*}</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td>{/}</td>
</tr>
</tbody>
</table>
**The FOLLOW set**

For a non-terminal $A$, define $\text{FOLLOW}(A)$ 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 $\text{FOLLOW}(X)$:

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

<table>
<thead>
<tr>
<th>rule</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>goal</td>
<td>eof</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>{eof}</td>
</tr>
<tr>
<td>expr</td>
<td>–</td>
<td>–</td>
<td>eof</td>
<td>–</td>
<td>{eof}</td>
</tr>
<tr>
<td>expr’</td>
<td>–</td>
<td>–</td>
<td>eof</td>
<td>–</td>
<td>{eof}</td>
</tr>
<tr>
<td>term</td>
<td>–</td>
<td>+,-</td>
<td>–</td>
<td>eof</td>
<td>{eof,+,-}</td>
</tr>
<tr>
<td>term’</td>
<td>–</td>
<td>–</td>
<td>eof,+,-</td>
<td>–</td>
<td>{eof,+,-}</td>
</tr>
<tr>
<td>factor</td>
<td>–</td>
<td>*,/</td>
<td>–</td>
<td>eof,+,-</td>
<td>{eof,+,-,*,/}</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>num</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨goal⟩</td>
<td>$g \rightarrow e$</td>
<td>$g \rightarrow e$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>⟨expr⟩</td>
<td>$e \rightarrow te'$</td>
<td>$e \rightarrow te'$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>⟨expr'⟩</td>
<td>-</td>
<td>-</td>
<td>$e' \rightarrow +e$</td>
<td>$e' \rightarrow -e$</td>
<td>-</td>
<td>-</td>
<td>$e' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>⟨term⟩</td>
<td>$t \rightarrow ft'$</td>
<td>$t \rightarrow ft'$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>⟨term'⟩</td>
<td>-</td>
<td>-</td>
<td>$t' \rightarrow \epsilon$</td>
<td>$t' \rightarrow \epsilon$</td>
<td>$t' \rightarrow *t$</td>
<td>$t' \rightarrow /t$</td>
<td>$t' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>⟨factor⟩</td>
<td>$f \rightarrow id$</td>
<td>$f \rightarrow num$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**LL(1) grammars**

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 $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ 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.
**LL(1) grammars**

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 ε–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 | a*
  
  is not *LL(1)*
  
  \[
  \text{FIRST}(aS) = \text{FIRST}(a) = \{a\}
  \]

- \[
  \begin{align*}
  S & ::= aS' \\
  S' & ::= aS' | \epsilon
  \end{align*}
  \]

  accepts the same language and is *LL(1)*
LL grammars

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:
  \[ S ::= a^i b^j \text{ where } i \geq j \]
• equivalent to dangling else problem
• problem - must choose production after k tokens of lookahead

Bottom-up parsers avoid this problem