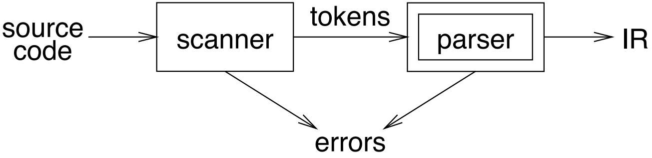
Chapter 3: LL Parsing

The role of the parser



Parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

Syntax analysis

Context-free syntax is specified with a *context-free grammar*.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

- V_t is the set of *terminal* symbols in the grammar. For our purposes, V_t is the set of tokens returned by the scanner.
- V_n, the *nonterminals*, is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.
- S is a distinguished nonterminal ($S \in V_n$) denoting the entire set of strings in $L(G)$. This is sometimes called a *goal symbol*.
- P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the *vocabulary* of G

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

If $A \to \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \to \gamma$
Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps
If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G
$L(G) = \{w \in V_t^* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a *sentence* of G
Note, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).
Example:

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle$
3		$\mid$	`num`
4		$\mid$	`id`
5	$\langle \text{op} \rangle$	$::=$	$+$
6		$\mid$	$-$
7		$\mid$	$*$
8		$\mid$	$/$

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or underline
3. productions as in the example

Scanning vs. parsing

Where do we draw the line?

$$\begin{array}{lcl} \textit{term} & ::= & [\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}]([\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}] \mid [0-9])^* \\ & \mid & 0 \mid [\mathtt{1}-9][\mathtt{0}-9]^* \\ \textit{op} & ::= & + \mid - \mid * \mid / \\ \textit{expr} & ::= & (\textit{term op})^* \textit{term} \end{array}$$

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

Context-free grammars are used to count:

- brackets: `()`, `begin`...`end`, `if`...`then`...`else`
- imparting structure: expressions

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes the compiler more manageable.

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG:

$$\begin{aligned}
\langle\text{goal}\rangle &\Rightarrow \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{aligned}$$

We have derived the sentence $\texttt{x} + 2 * \texttt{y}$.
We denote this $\langle\text{goal}\rangle \Rightarrow^* \texttt{id} + \texttt{num} * \texttt{id}$.
Such a sequence of rewrites is a *derivation* or a *parse*.
The process of discovering a derivation is called *parsing*.

Derivations

At each step, we chose a non-terminal to replace.
This choice can lead to different derivations.

Two are of particular interest:

leftmost derivation
the leftmost non-terminal is replaced at each step

rightmost derivation
the rightmost non-terminal is replaced at each step

The previous example was a leftmost derivation.

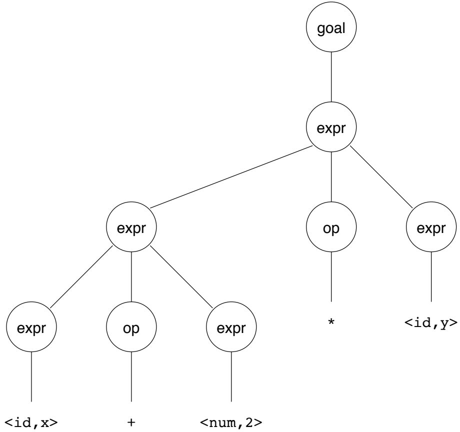
Rightmost derivation

For the string $\mathtt{x} + 2 * \mathtt{y}$:

$$
\begin{array}{rcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{array}
$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.

Precedence



Treewalk evaluation computes $(x + 2) * y$
— the "wrong" answer!
Should be $x + (2 * y)$

Precedence

These two derivations point out a problem with the grammar.
It has no notion of precedence, or implied order of evaluation.
To add precedence takes additional machinery:

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

This grammar enforces a precedence on the derivation:

- terms *must* be derived from expressions
- forces the “correct” tree

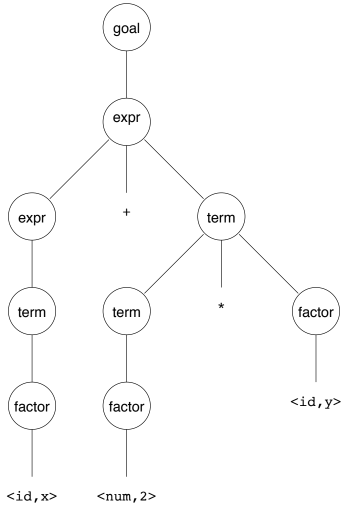
Precedence

Now, for the string $\mathtt{x} + \mathtt{2} * \mathtt{y}$:

$$
\begin{array}{lcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{factor}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{term}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{factor}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle \\
& \Rightarrow & \langle\text{id,x}\rangle + \langle\text{num,2}\rangle * \langle\text{id,y}\rangle
\end{array}
$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$, but this time, we build the desired tree.

Precedence



Treewalk evaluation computes $\mathtt{x} + (2 * \mathtt{y})$

Ambiguity

If a grammar has more than one derivation for a single sentential form, then it is *ambiguous*

Example:

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{then } \langle \text{stmt} \rangle \\ & | & \texttt{if } \langle \text{expr} \rangle \texttt{then } \langle \text{stmt} \rangle \texttt{else } \langle \text{stmt} \rangle \\ & | & \texttt{other stmts} \end{array}$$

Consider deriving the sentential form:

$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & ::= & \langle \text{matched} \rangle \\
 & | & \langle \text{unmatched} \rangle \\
\langle \text{matched} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{matched} \rangle \texttt{ else } \langle \text{matched} \rangle \\
 & | & \texttt{other stmts} \\
\langle \text{unmatched} \rangle & ::= & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{stmt} \rangle \\
 & | & \texttt{if } \langle \text{expr} \rangle \texttt{ then } \langle \text{matched} \rangle \texttt{ else } \langle \text{unmatched} \rangle
\end{array}
$$

This generates the same language as the ambiguous grammar, but applies the common sense rule:

match each `else` with the closest unmatched `then`

This is most likely the language designer's intent.

Ambiguity

Ambiguity is often due to confusion in the context-free specification.
Context-sensitive confusions can arise from *overloading*.
Example:

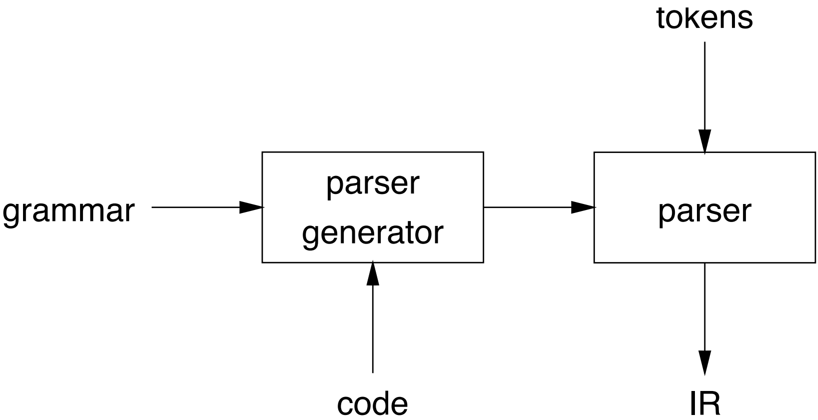
```
a = f(17)
```

In many Algol-like languages, `f` could be a function or subscripted variable.
Disambiguating this statement requires context:

- need *values* of declarations
- not *context-free*
- really an issue of *type*

Rather than complicate parsing, we will handle this separately.

Parsing: the big picture



Our goal is a flexible parser generator system

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, labelled with the start 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 $A \to \alpha$ and construct the appropriate child for each symbol of α
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 V_n)

The key is selecting the right production in step 1

$\Rightarrow$ *should be guided by input string*

Simple expression grammar

Recall our grammar for simple expressions:

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

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

Example

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
8	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
8	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
–	$\langle goal \rangle$	$\uparrow x - 2 * y$
1	$\langle expr \rangle$	$\uparrow x - 2 * y$
2	$\langle expr \rangle + \langle term \rangle$	$\uparrow x - 2 * y$
2	$\langle expr \rangle + \langle term \rangle + \langle term \rangle$	$\uparrow x - 2 * y$
2	$\langle expr \rangle + \langle term \rangle + \cdots$	$\uparrow x - 2 * y$
2	$\langle expr \rangle + \langle term \rangle + \cdots$	$\uparrow x - 2 * y$
2	$\cdots$	$\uparrow x - 2 * y$

If the parser makes the wrong choices, 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

$\exists A \in V_n$ *such that* $A \Rightarrow^{+} A\alpha$ *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:

$$\begin{array}{lcl} \langle \mathrm{foo} \rangle & ::= & \langle \mathrm{foo} \rangle \alpha \\ & | & \beta \end{array}$$

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

We can rewrite this as:

$$\begin{array}{lcl} \langle \mathrm{foo} \rangle & ::= & \beta \langle \mathrm{bar} \rangle \\ \langle \mathrm{bar} \rangle & ::= & \alpha \langle \mathrm{bar} \rangle \\ & | & \varepsilon \end{array}$$

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

This fragment contains no left-recursion

Example

Our expression grammar contains two cases of left-recursion

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

Applying the transformation gives

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

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

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

It is

- right-recursive
- free of ε productions

Unfortunately, it generates different associativity
Same syntax, different meaning

Example

Our long-suffering 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	$\langle\text{expr}'\rangle$	$::=$	$+\langle\text{term}\rangle\langle\text{expr}'\rangle$
4		$\mid$	$-\langle\text{term}\rangle\langle\text{expr}'\rangle$
5		$\mid$	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		$\mid$	$/\langle\text{factor}\rangle\langle\text{term}'\rangle$
9		$\mid$	ε
10	$\langle\text{factor}\rangle$	$::=$	`num`
11		$\mid$	`id`

Recall, we factored out 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): **l**eft to right scan, **l**eft-most derivation, **1**-token lookahead; and

LR(1): **l**eft to right scan, **r**ight-most derivation, **1**-token lookahead

Predictive parsing

Basic idea:

For any two productions $A \rightarrow \alpha \mid \beta$, 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 first in some string derived from α
That is, for some $w \in V_t^*$, $w \in \text{FIRST}(\alpha)$ iff. $\alpha \Rightarrow^* w\gamma$.

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

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

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 α common to two or more of its alternatives.

if $\alpha \neq \varepsilon$ then replace all of the A productions
$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$
with

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where A' 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	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle + \langle \text{expr} \rangle$
3		$\mid$	$\langle \text{term} \rangle - \langle \text{expr} \rangle$
4		$\mid$	$\langle \text{term} \rangle$
5	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle * \langle \text{term} \rangle$
6		$\mid$	$\langle \text{factor} \rangle / \langle \text{term} \rangle$
7		$\mid$	$\langle \text{factor} \rangle$
8	$\langle \text{factor} \rangle$	$::=$	`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 $/$.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \phi$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals that must be left factored:

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle + \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle - \langle \text{expr} \rangle \\
& | & \langle \text{term} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle * \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle / \langle \text{term} \rangle \\
& | & \langle \text{factor} \rangle
\end{array}
$$

Applying the transformation gives us:

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
\langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
& | & - \langle \text{expr} \rangle \\
& | & \varepsilon
\end{array}
$$

$$
\begin{array}{lcl}
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
\langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
& | & / \langle \text{term} \rangle \\
& | & \varepsilon
\end{array}
$$

Example

Substituting back into the grammar yields

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		$\mid$	$-\langle \text{expr} \rangle$
5		$\mid$	ε
6	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
7	$\langle \text{term}' \rangle$	$::=$	$*\langle \text{term} \rangle$
8		$\mid$	$/\langle \text{term} \rangle$
9		$\mid$	ε
10	$\langle \text{factor} \rangle$	$::=$	`num`
11		$\mid$	`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	idε ⟨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⟨term′⟩⟨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.

Back to left-recursion elimination

Given a left-factored CFG, to eliminate left-recursion:

if $\exists\ A \rightarrow A\alpha$ then replace all of the A productions

$$A \rightarrow A\alpha \mid \beta \mid \ldots \mid \gamma$$

with

$$A \rightarrow NA'$$
$$N \rightarrow \beta \mid \ldots \mid \gamma$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

where N and A' are new productions.

Repeat until there are no left-recursive productions.

Generality

Question:

> *By left factoring and eliminating left-recursion, 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:

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

Must look past an arbitrary number of *a*'s to discover the 0 or the 1 and so determine the derivation.

Recursive descent parsing

Now, we can produce a simple recursive descent parser from the (right-associative) grammar.

```
Token token;
void eat(char a) {
    if (token == a){ token = next_token(); }
                   { error(); }
}
void goal() { token = next_token(); expr(); eat(EOF); }
void expr() { term(); expr_prime(); }
void expr_prime() {
    if (token == PLUS)
        { eat(PLUS); expr(); }
    else if (token == MINUS)
        { eat(MINUS); expr(); }
    else { }
}
```

Recursive descent parsing

```
void term() { factor(); term_prime(); }
void term_prime() {
   if (token = MULT)
      { eat(MULT); term(); }
   else if (token = DIV)
      { eat(DIV); term(); }
   else { }
}
void factor() {
   if (token = NUM)
      { eat(NUM); }
   else if (token = ID)
      { eat(ID); }
   else error();
}
```

Nullable

For a string α of grammar symbols, define $\textsc{Nullable}(\alpha)$ as

> α *can go to* ε.

$\textsc{Nullable}(\alpha)$ if and only if $(\alpha \Rightarrow^* \varepsilon)$

How to compute $\textsc{Nullable}(U)$, for $U \in V_t \cup V_n$.

1. For each U, let $\textsc{Nullable}(U)$ be a Boolean variable.
2. Derive the following constraints:
   2.1 If $a \in V_t$,
      - $\textsc{Nullable}(a) = \text{false}$

   2.2 If $A \to Y_1 \cdots Y_k$ is a production:
      - $[\,\textsc{Nullable}(Y_1) \wedge \cdots \wedge \textsc{Nullable}(Y_k)\,] \Longrightarrow \textsc{Nullable}(A)$
3. Solve the constraints.

$$\textsc{nullable}(X_1 \cdots X_k) \;=\; \textsc{Nullable}(X_1) \wedge \cdots \wedge \textsc{Nullable}(X_k)$$

FIRST

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

> *the set of terminal symbols that begin strings derived from* α.

$$\text{FIRST}(\alpha) \;=\; \{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$$

How to compute $\text{FIRST}(U)$, for $U \in V_t \cup V_n$.

1. For each U, let $\text{FIRST}(U)$ be a set variable.
2. Derive the following constraints:
   2.1 If $a \in V_t$,
      - $\text{FIRST}(a) = \{ a \}$

   2.2 If $A \to Y_1 Y_2 \cdots Y_k$ is a production:
      - $\text{FIRST}(Y_1) \subseteq \text{FIRST}(A)$
      - $\forall i : 1 < i \leq k$, if $\text{NULLABLE}(Y_1 \cdots Y_{i-1})$, then $\text{FIRST}(Y_i) \subseteq \text{FIRST}(A)$
3. Solve the constraints. Go for the $\subseteq$-least solution.

$$\text{FIRST}(X_1 \cdots X_k) \;=\; \bigcup_{i:1 \leq i \leq k \wedge \text{NULLABLE}(X_1 \cdots X_{i-1})} \text{FIRST}(X_i)$$

FOLLOW

For a non-terminal B, define $\textsc{follow}(B)$ as

> *the set of terminals that can appear immediately to the right of B in some sentential form*

$$\textsc{follow}(B) = \{a \in V_t \mid G \Rightarrow^* \alpha B \beta \wedge a \in \textsc{first}(\beta\,\$)\}$$

How to compute $\textsc{follow}(B)$.

1. For each non-terminal B, let $\textsc{follow}(B)$ be a set variable.
2. Derive the following constraints:
   - 2.1 If G is the start symbol and \$ is the end-of-file marker, then
     - $\{\,\$\,\} \subseteq \textsc{follow}(G)$
   - 2.2 If $A \rightarrow \alpha B \beta$ is a production:
     - $\textsc{first}(\beta) \subseteq \textsc{follow}(B)$
     - if $\textsc{Nullable}(\beta)$, then $\textsc{follow}(A) \subseteq \textsc{follow}(B)$
3. Solve the constraints. Go for the $\subseteq$-least solution.

LL(1) grammars

Intuition: A grammar G is LL(1) iff
for all non-terminals A, each distinct pair of productions

$A \to \beta$

$A \to \gamma$

satisfy the condition $\text{FIRST}(\beta) \bigcap \text{FIRST}(\gamma) = \emptyset$.

Question: What if $\text{NULLABLE}(A)$?

Definition: A grammar G is LL(1) iff
for each set of productions $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$:

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \ldots, \text{FIRST}(\alpha_n)$ are pairwise disjoint, and
2. If $\text{NULLABLE}(\alpha_i)$, then for all j, such that $1 \leq j \leq n \land j \neq i$: $\text{FIRST}(\alpha_j) \bigcap \text{FOLLOW}(A) = \emptyset$.

If G is ε-free, condition 1 is sufficient.

LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for *A* begins with a distinct terminal is a *simple* LL(1) grammar.

Example

$S \rightarrow aS \mid a$

is not LL(1) because $\textsc{first}(aS) = \textsc{first}(a) = \{a\}$

$S \rightarrow aS'$

$S' \rightarrow aS' \mid \varepsilon$

accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G
Output: Parsing table M
Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   1.1 $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$
   1.2 If $\varepsilon \in \text{FIRST}(\alpha)$:
       1.2.1 $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$
       1.2.2 If $\$ \in \text{FOLLOW}(A)$ then add $A \rightarrow \alpha$ to $M[A,\$]$
2. Set each undefined entry of M to `error`

If $\exists M[A,a]$ with multiple entries then grammar is not LL(1).

Note: recall $a,b \in V_t$, so $a,b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$$
\begin{array}{l|l}
S \to E & T \to FT' \\
E \to TE' & T' \to *T \mid /T \mid \varepsilon \\
E' \to +E \mid -E \mid \varepsilon & F \to \texttt{id} \mid \texttt{num}
\end{array}
$$

	FIRST	FOLLOW
S	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E'	$\{\varepsilon, +, -\}$	$\{\$\}$
T	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, \$\}$
T'	$\{\varepsilon, *, /\}$	$\{+, -, \$\}$
F	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, *, /, \$\}$
id	$\{\texttt{id}\}$	–
num	$\{\texttt{num}\}$	–
$*$	$\{*\}$	–
$/$	$\{/\}$	–
$+$	$\{+\}$	–
$-$	$\{-\}$	–

	id	num	$+$	$-$	$*$	$/$	$\$$
S	$S \to E$	$S \to E$	–	–	–	–	–
E	$E \to TE'$	$E \to TE'$	–	–	–	–	–
E'	–	–	$E' \to +E$	$E' \to -E$	–	–	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	–	–	–	–	–
T'	–	–	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \texttt{id}$	$F \to \texttt{num}$	–	–	–	–	–

A grammar that is not LL(1)

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle \\ & | & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \texttt{else}\ \langle \text{stmt} \rangle \\ & | & \ldots \end{array}$$

Left-factored:

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \langle \text{stmt}' \rangle \mid \ldots \\ \langle \text{stmt}' \rangle & ::= & \texttt{else}\ \langle \text{stmt} \rangle \mid \varepsilon \end{array}$$

Now, FIRST($\langle \text{stmt}' \rangle$) $= \{\varepsilon, \texttt{else}\}$
Also, FOLLOW($\langle \text{stmt}' \rangle$) $= \{\texttt{else}, \$\}$
But, FIRST($\langle \text{stmt}' \rangle$) $\bigcap$ FOLLOW($\langle \text{stmt}' \rangle$) $= \{\texttt{else}\} \neq \phi$
On seeing `else`, conflict between choosing

$$\langle \textit{stmt}' \rangle ::= \texttt{else}\ \langle \textit{stmt} \rangle \quad \textit{and} \quad \langle \textit{stmt}' \rangle ::= \varepsilon$$

$\Rightarrow$ grammar is not LL(1)!
The fix:

Put priority on $\langle \textit{stmt}' \rangle ::= \texttt{else}\ \langle \textit{stmt} \rangle$ *to associate* `else` *with closest previous* `then`*.*