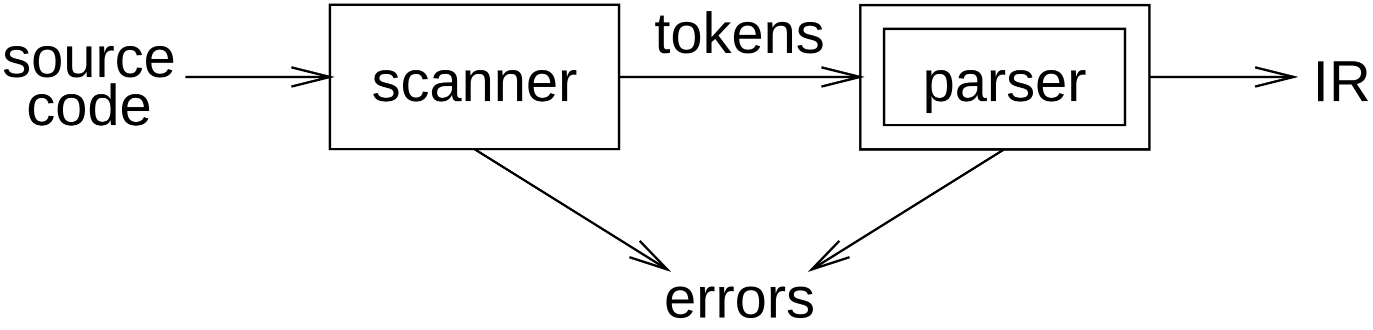
The role of the parser



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 $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \rightarrow \gamma$

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, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Why it is called ”context free grammar”?

Syntax analysis

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

Example:

$$\begin{array}{r|rcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
3 & & \mid & \texttt{num} \\
4 & & \mid & \texttt{id} \\
5 & \langle\text{op}\rangle & ::= & + \\
6 & & \mid & - \\
7 & & \mid & * \\
8 & & \mid & /
\end{array}$$

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 <u>underline</u>
3. productions as in the example

Scanning vs. parsing

Where do we draw the line?

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

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 compiler more manageable.

Derivations

We can view the productions of a CFG as rewriting rules.

Using our example CFG:

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

We have derived the sentence $\mathtt{x} + \mathtt{2} * \mathtt{y}$.
We denote this $\langle\text{goal}\rangle\Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.

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.

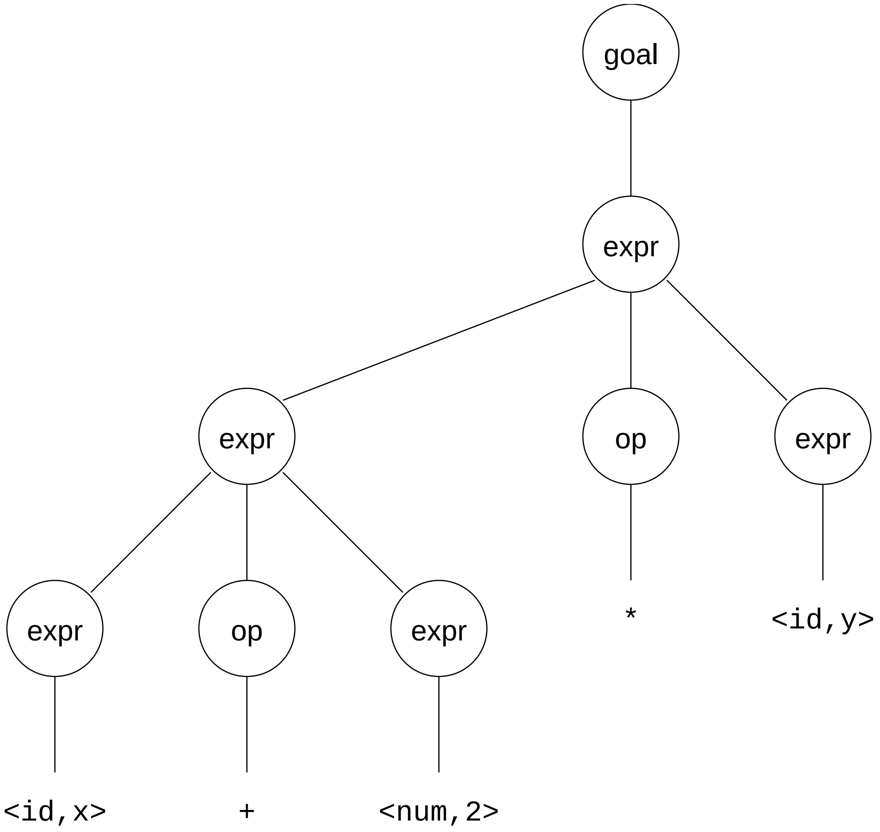
Rightmost derivation

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

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

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

Precedence



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

— the “wrong” answer!

Should be $\mathtt{x} + (2 * \mathtt{y})$

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:

$$
\begin{array}{r|lcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
3 & & | & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
4 & & | & \langle\text{term}\rangle \\
5 & \langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
6 & & | & \langle\text{term}\rangle / \langle\text{factor}\rangle \\
7 & & | & \langle\text{factor}\rangle \\
8 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
9 & & | & \texttt{id}
\end{array}
$$

This grammar enforces a precedence on the derivation:

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

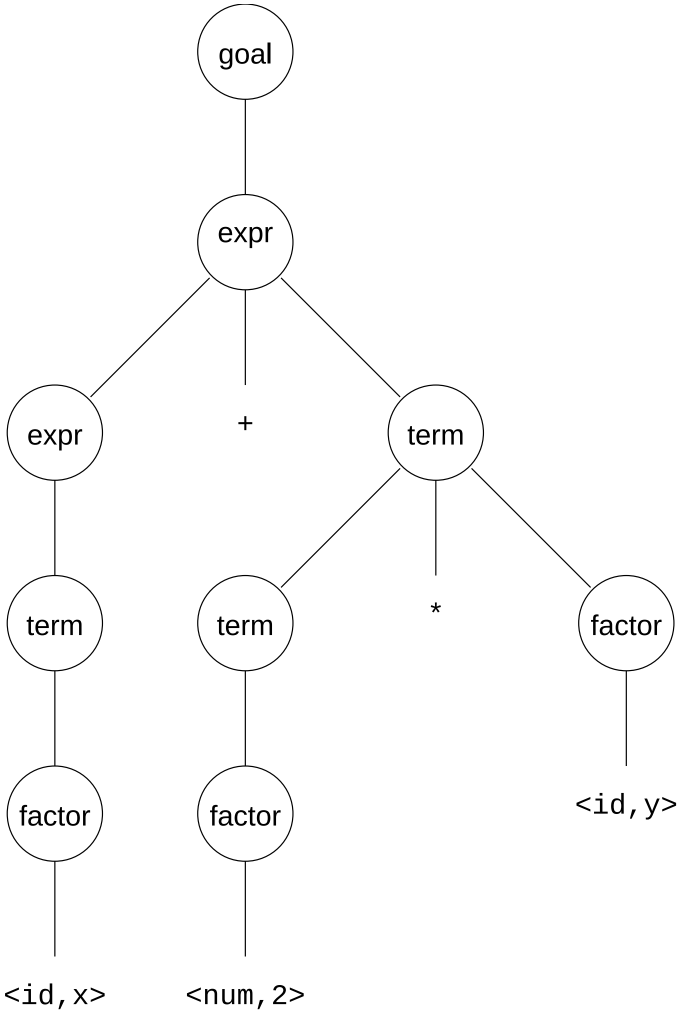
Precedence

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

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

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:

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

Consider deriving the sentential form:

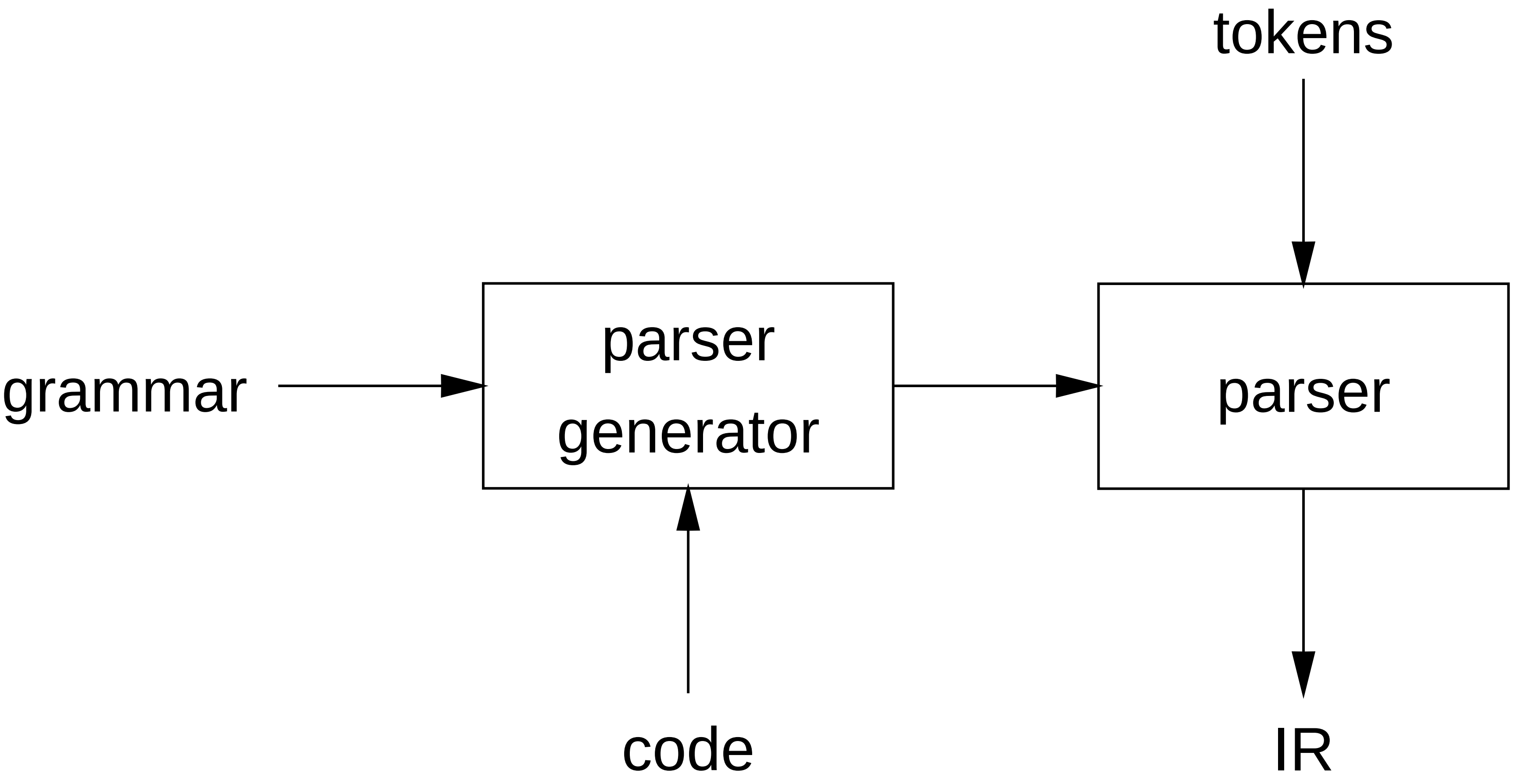
$$\texttt{if}\ E_1\ \texttt{then}\ \texttt{if}\ E_2\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

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
- requires the capability of predicting the right rule

Bottom-up parsers

- start at the leaves and fill in the derivation tree in a bottom-up fashion
- an intermediate node is inserted if the body (right hand side) appears.

A simple grammar

$$\begin{array}{r|lcl}
1 & S & ::= & \textbf{data}\ H\ B \\
2 & H & ::= & id\ num \\
3 & B & ::= & R\ B \mid \varepsilon \\
4 & R & ::= & (\ num\ )
\end{array}$$

Example string: **data** Grade 2 (100) (90)

A top down parser for the simple grammar

```
void eat (Token s) {
   if (s!=scanner.getNextToken()) {
      error();
   }
}

int main () {
   eat (data);
   parseH();
   parseB();
}

void parseH() {
   eat(id);
   eat(num);
}
```

```
void parseB() {
   if (!endOfFile()) {
      parseR();
      parseB();
   }
}

void parseR() {
   eat(leftParenthesis);
   eat(num);
   eat(rightParentheis);
}
```

Problem 1:Left Recursion

$$
\begin{array}{r|lcl}
1 & S & ::= & \textbf{data}\ H\ B \\
2 & H & ::= & id\ num \\
3 & B & ::= & B\ R \mid \varepsilon \\
4 & R & ::= & (\ num\ )
\end{array}
$$

Formally, a grammar is *left-recursive* if

$\exists A \in V_n$ such that $A \Rightarrow^{+} A\alpha$ for some string α

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

Consider the grammar fragment:

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

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

We can rewrite this as:

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

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

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

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

Problem 2: deciding production rules

$$\begin{array}{r|lll}
1 & S & ::= & \textbf{data}\ H\ B \\
2 & H & ::= & id\ num \\
3 & B & ::= & R\ B\ |N\ B\ |\ \varepsilon \\
4 & R & ::= & (\ num\ ) \\
5 & N & ::= & ''\ id\ ''
\end{array}$$

Example string: **data** Grade 2 (100) “Wendy”

For some RHS $\alpha \in G$, define $\textsc{first}(\alpha)$ as the set of tokens that appear first in some string derived from α.
That is, for some $w \in V_t^*$, $w \in \textsc{first}(\alpha)$ 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

$$\textsc{first}(\alpha) \cap \textsc{first}(\beta) = \phi$$

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

Deciding production rules (cont.)

$$\begin{array}{r|ccl}
1 & S & ::= & \mathbf{data}\ H\ B \\
2 & H & ::= & id\ num \\
3 & B & ::= & R\ B\ |N\ B\ |\ \varepsilon \\
4 & R & ::= & (\ num\ )\ |(\ ) \\
5 & N & ::= & ''\ id\ ''
\end{array}$$

Two solutions:

1. Multiple tokens lookahead. Simple but expensive.
2. Left factoring.

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
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$
with

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where A' is a new non-terminal.

Repeat until no two alternatives for a single
non-terminal have a common prefix.

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.

The simplest way to construct a top-down parser.

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.