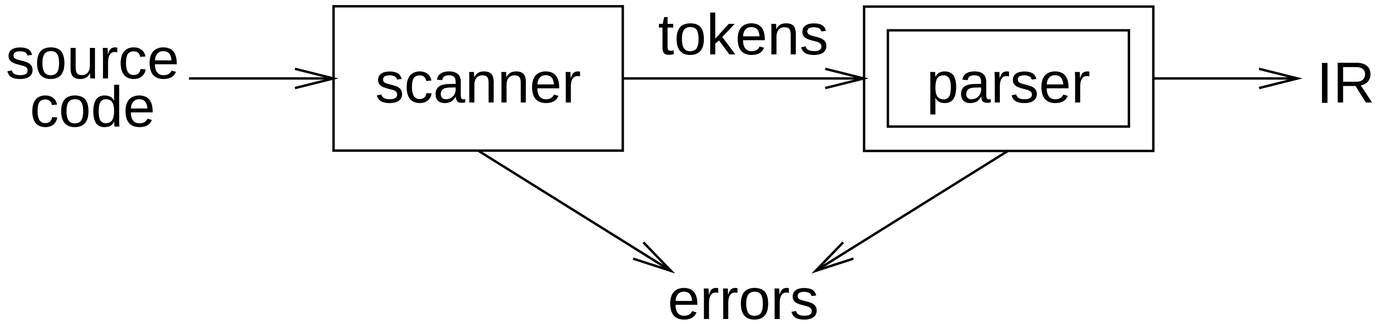
Chapter 3: LL Parsing

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

For the next few weeks, we will look at parser construction

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:

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

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?

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

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 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.

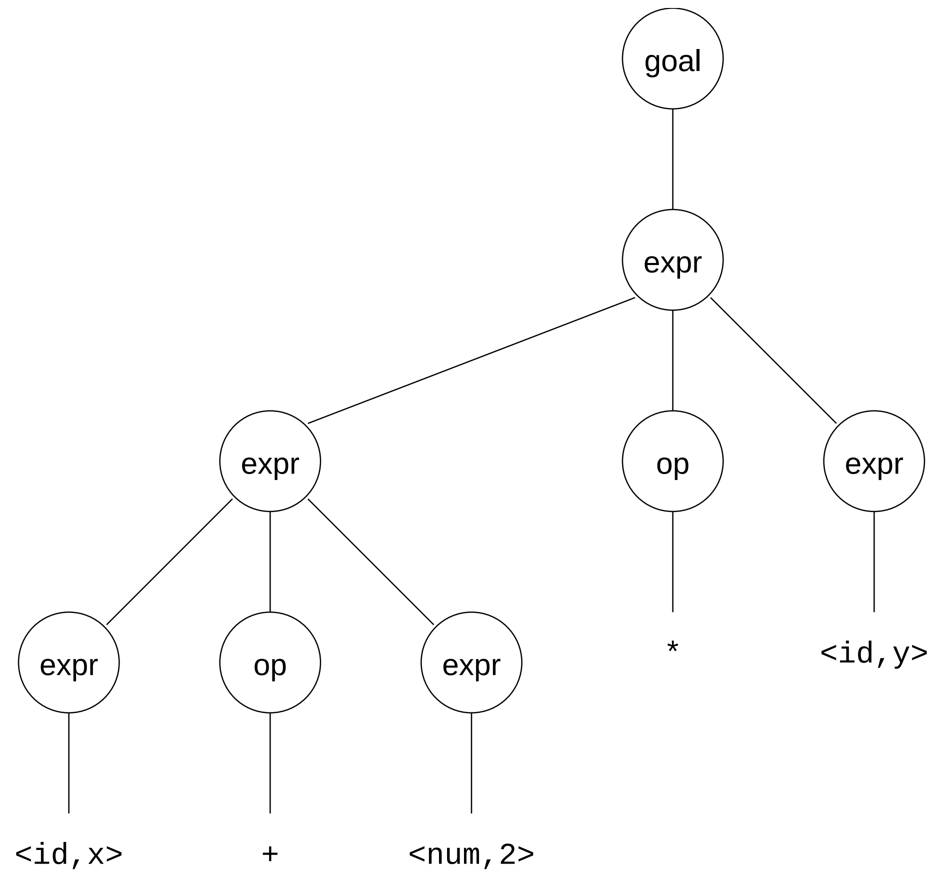
Rightmost derivation

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

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

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:

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

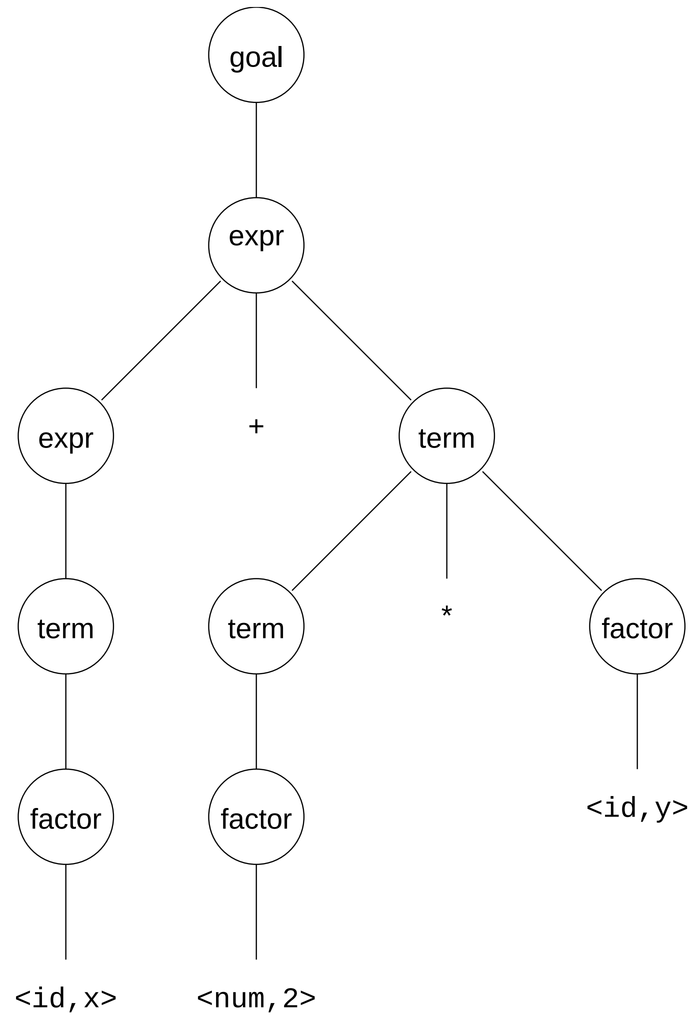
Precedence

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

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

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.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

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

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)
```

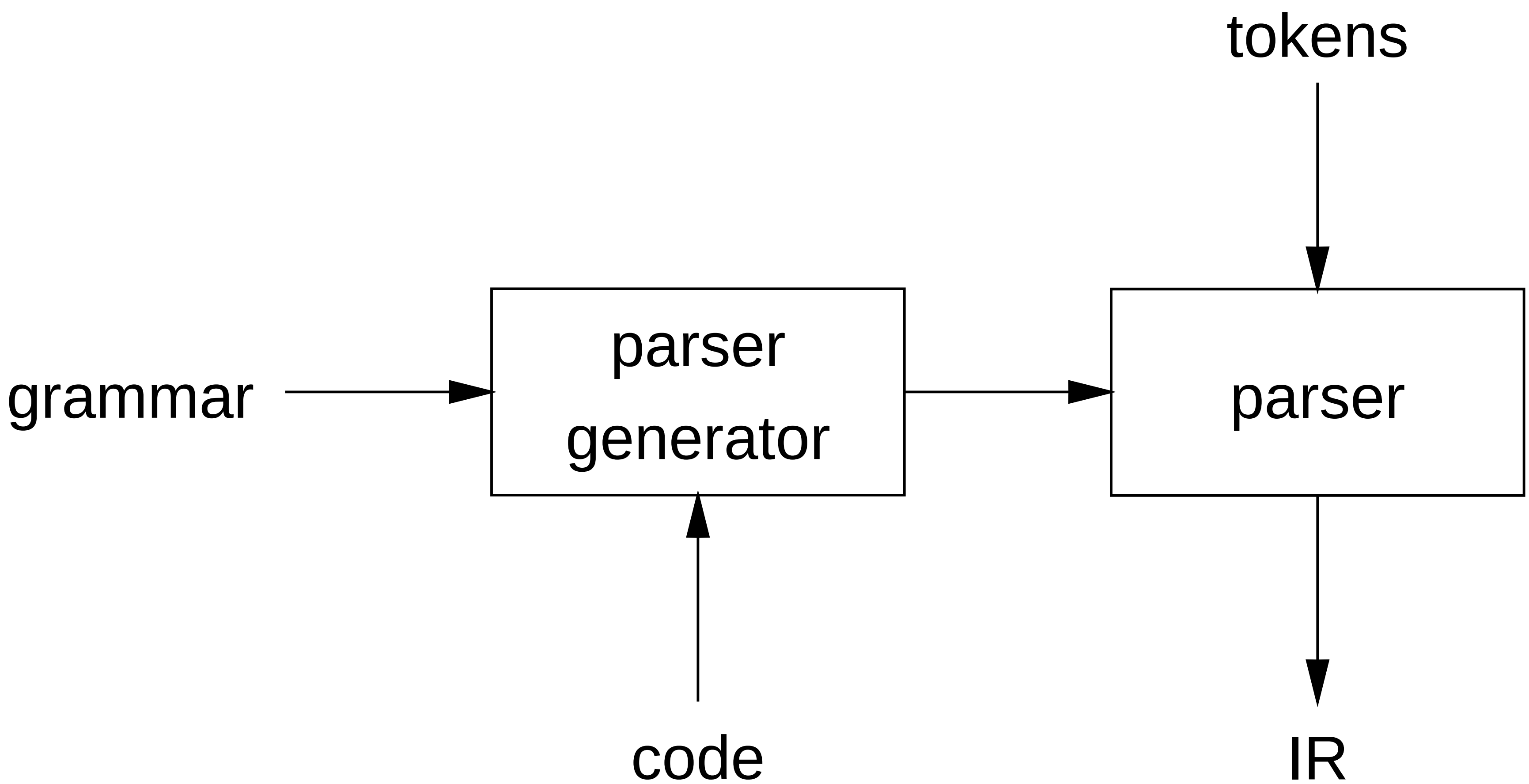
In many Algol-like languages, `f` could be a function or subscripted variable.

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 prefi xes*)
- 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:

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

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

Example

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
4	$\langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
7	$\langle\text{factor}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
9	$\mathtt{id} + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} + \langle\text{term}\rangle$	$\mathtt{x}\ \uparrow-\ 2\ *\ \mathtt{y}$
–	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
3	$\langle\text{expr}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
4	$\langle\text{term}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
7	$\langle\text{factor}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
9	$\mathtt{id} - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ \uparrow-\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
7	$\mathtt{id} - \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
8	$\mathtt{id} - \mathtt{num}$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num}$	$\mathtt{x}\ -\ 2\ \uparrow *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
5	$\mathtt{id} - \langle\text{term}\rangle * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
7	$\mathtt{id} - \langle\text{factor}\rangle * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
8	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ 2\ \uparrow *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ 2\ *\ \uparrow\mathtt{y}$
9	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	$\mathtt{x}\ -\ 2\ *\ \uparrow\mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	$\mathtt{x}\ -\ 2\ *\ \mathtt{y}\ \uparrow$

Example

Another possible parse for $\mathtt{x} - \mathtt{2} * \mathtt{y}$

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$
2	$\cdots$	$\uparrow\mathtt{x}\ -\ \mathtt{2}\ *\ \mathtt{y}$

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:

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

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

We can rewrite this as:

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

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

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

Applying the transformation gives

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

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

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

It is

- right-recursive
- free of ε productions

Unfortunately, it generates different associativity

Same syntax, different meaning

Example

Our long-suffering expression grammar:

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

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

Predictive parsing

Basic idea:

> For any two productions $A \to \alpha \mid \beta$, 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 $A \to \alpha$ and $A \to \beta$ 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:

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

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:

$$
\begin{array}{rcl}
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle + \langle\text{expr}\rangle \\
& | & \langle\text{term}\rangle - \langle\text{expr}\rangle \\
& | & \langle\text{term}\rangle \\[4pt]
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle * \langle\text{term}\rangle \\
& | & \langle\text{factor}\rangle / \langle\text{term}\rangle \\
& | & \langle\text{factor}\rangle
\end{array}
$$

Applying the transformation gives us:

$$
\begin{array}{rcl}
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\langle\text{expr}'\rangle \\
\langle\text{expr}'\rangle & ::= & +\langle\text{expr}\rangle \\
& | & -\langle\text{expr}\rangle \\
& | & \varepsilon \\[4pt]
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\langle\text{term}'\rangle \\
\langle\text{term}'\rangle & ::= & *\langle\text{term}\rangle \\
& | & /\langle\text{term}\rangle \\
& | & \varepsilon
\end{array}
$$

Example

Substituting back into the grammar yields

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

Now, selection requires only a single token lookahead.

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

Example

	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\mathtt{x} - \mathtt{2} * \mathtt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x} - \mathtt{2} * \mathtt{y}$
2	$\langle\text{term}\rangle\langle\text{expr}'\rangle$	$\uparrow\mathtt{x} - \mathtt{2} * \mathtt{y}$
6	$\langle\text{factor}\rangle\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\uparrow\mathtt{x} - \mathtt{2} * \mathtt{y}$
11	$\mathtt{id}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\uparrow\mathtt{x} - \mathtt{2} * \mathtt{y}$
–	$\mathtt{id}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x}\ \uparrow\mathtt{-}\ \mathtt{2} * \mathtt{y}$
9	$\mathtt{id}\varepsilon\ \langle\text{expr}'\rangle$	$\mathtt{x}\ \uparrow\mathtt{-}\ \mathtt{2}$
4	$\mathtt{id}-\ \langle\text{expr}\rangle$	$\mathtt{x}\ \uparrow\mathtt{-}\ \mathtt{2} * \mathtt{y}$
–	$\mathtt{id}-\ \langle\text{expr}\rangle$	$\mathtt{x} - \uparrow\mathtt{2} * \mathtt{y}$
2	$\mathtt{id}-\ \langle\text{term}\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \uparrow\mathtt{2} * \mathtt{y}$
6	$\mathtt{id}-\ \langle\text{factor}\rangle\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \uparrow\mathtt{2} * \mathtt{y}$
10	$\mathtt{id}-\ \mathtt{num}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \uparrow\mathtt{2} * \mathtt{y}$
–	$\mathtt{id}-\ \mathtt{num}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2}\ \uparrow\mathtt{*}\ \mathtt{y}$
7	$\mathtt{id}-\ \mathtt{num}*\ \langle\text{term}\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2}\ \uparrow\mathtt{*}\ \mathtt{y}$
–	$\mathtt{id}-\ \mathtt{num}*\ \langle\text{term}\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2} * \uparrow\mathtt{y}$
6	$\mathtt{id}-\ \mathtt{num}*\ \langle\text{factor}\rangle\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2} * \uparrow\mathtt{y}$
11	$\mathtt{id}-\ \mathtt{num}*\ \mathtt{id}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2} * \uparrow\mathtt{y}$
–	$\mathtt{id}-\ \mathtt{num}*\ \mathtt{id}\langle\text{term}'\rangle\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2} * \mathtt{y}\uparrow$
9	$\mathtt{id}-\ \mathtt{num}*\ \mathtt{id}\langle\text{expr}'\rangle$	$\mathtt{x} - \mathtt{2} * \mathtt{y}\uparrow$
5	$\mathtt{id}-\ \mathtt{num}*\ \mathtt{id}$	$\mathtt{x} - \mathtt{2} * \mathtt{y}\uparrow$

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.

```
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

Non-recursive predictive parsing

Observation:

> *Our recursive descent parser encodes state information in its run-time stack, or call stack.*

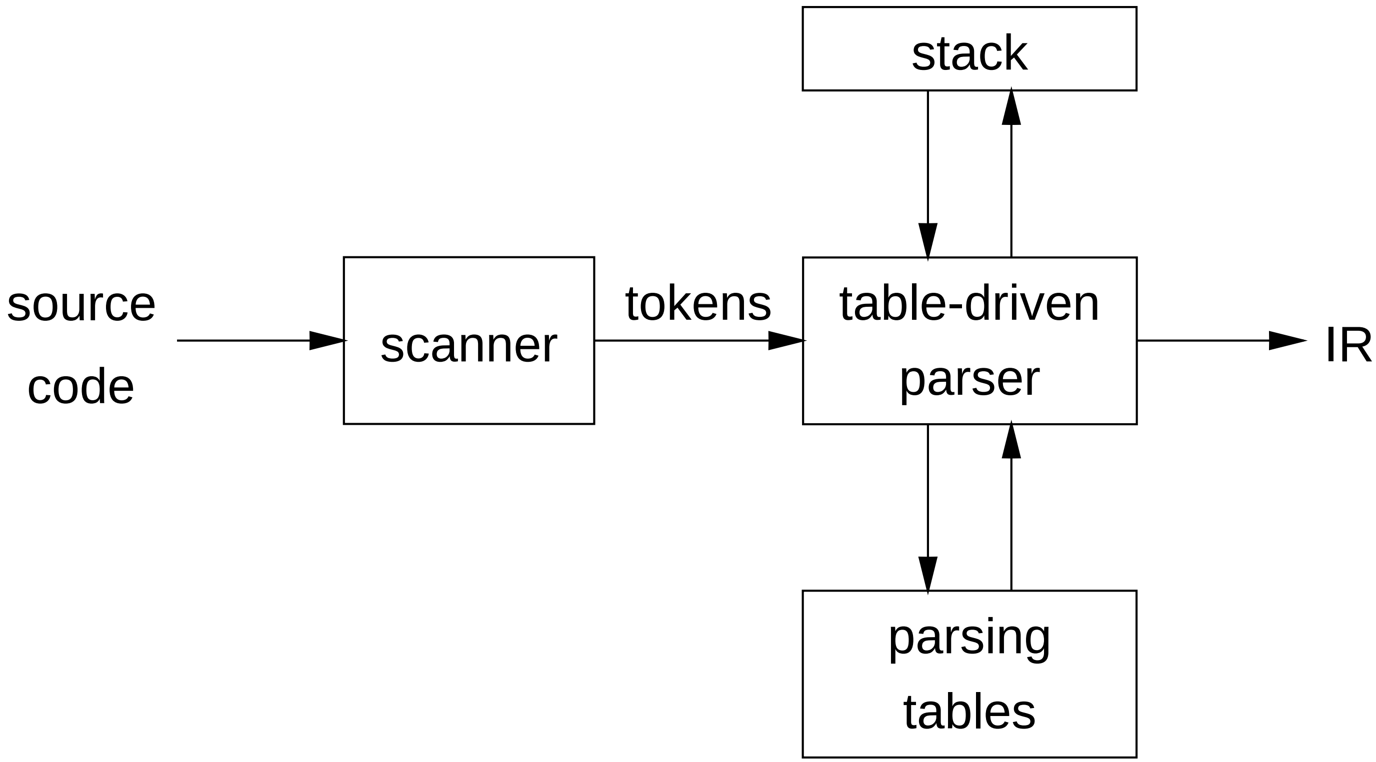
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

This suggests other implementation methods:

- explicit stack, hand-coded parser
- stack-based, table-driven parser

Non-recursive predictive parsing

Now, a predictive parser looks like:

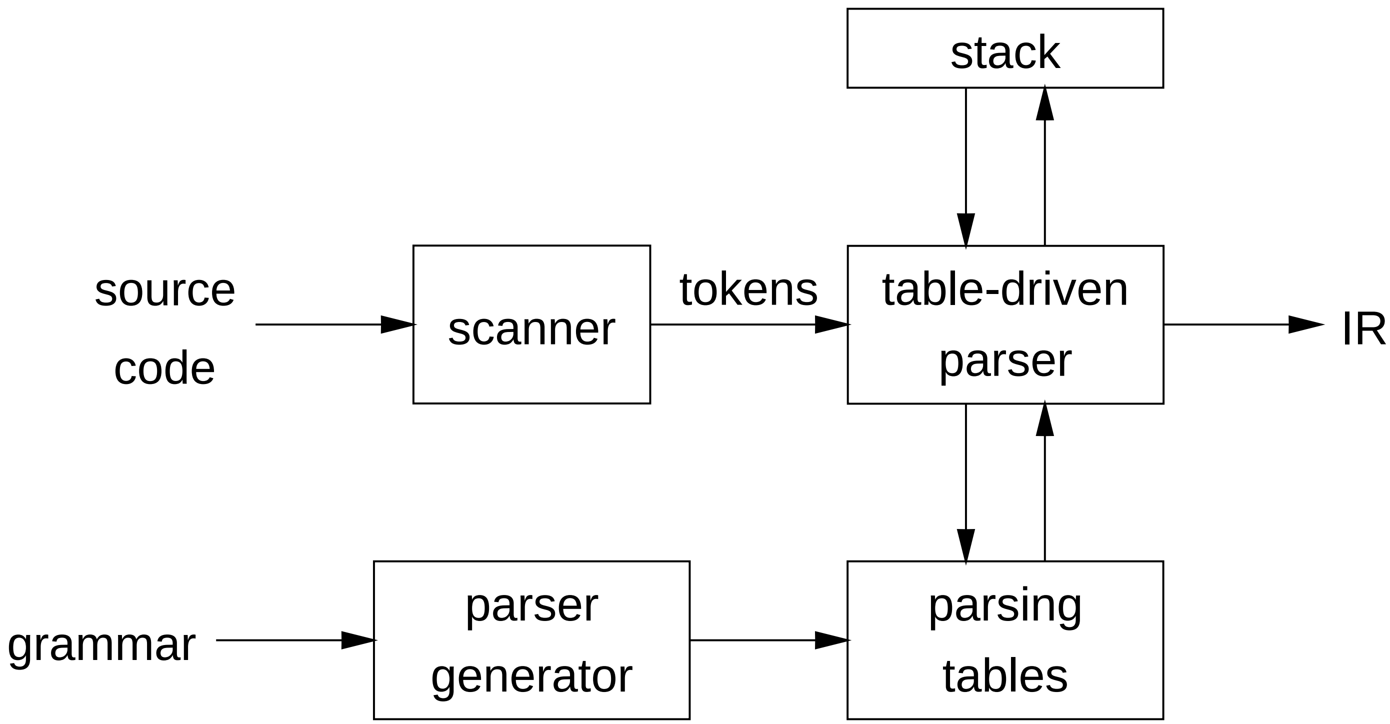


Rather than writing code, we build tables.

Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



This is true for both top-down (LL) and bottom-up (LR) parsers

Non-recursive predictive parsing

Input: a string w and a parsing table M for G

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk,Yk−1,···,Y1
      else error()
until X = EOF
```

Non-recursive predictive parsing

What we need now is a parsing table M.

Our 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	$\langle\text{expr}'\rangle$	$::=$	$+\langle\text{expr}\rangle$
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`

Its parse table:

	`id`	`num`	$+$	$-$	$*$	$/$	\$†
$\langle\text{goal}\rangle$	1	1	–	–	–	–	–
$\langle\text{expr}\rangle$	2	2	–	–	–	–	–
$\langle\text{expr}'\rangle$	–	–	3	4	–	–	5
$\langle\text{term}\rangle$	6	6	–	–	–	–	–
$\langle\text{term}'\rangle$	–	–	9	9	7	8	9
$\langle\text{factor}\rangle$	11	10	–	–	–	–	–

† we use \$ to represent `EOF`

FIRST

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

- the set of terminal symbols that begin strings derived from α:
  $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in \text{FIRST}(\alpha)$

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

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

1. If $X \in V_t$ then $\text{FIRST}(X)$ is $\{X\}$
2. If $X \to \varepsilon$ then add ε to $\text{FIRST}(X)$.
3. If $X \to Y_1 Y_2 \cdots Y_k$:
   (a) Put $\text{FIRST}(Y_1) - \{\varepsilon\}$ in $\text{FIRST}(X)$
   (b) $\forall i : 1 < i \leq k$, if $\varepsilon \in \text{FIRST}(Y_1) \cap \cdots \cap \text{FIRST}(Y_{i-1})$
   (i.e., $Y_1 \cdots Y_{i-1} \Rightarrow^* \varepsilon$)
   then put $\text{FIRST}(Y_i) - \{\varepsilon\}$ in $\text{FIRST}(X)$
   (c) If $\varepsilon \in \text{FIRST}(Y_1) \cap \cdots \cap \text{FIRST}(Y_k)$ then put ε in $\text{FIRST}(X)$

   Repeat until no more additions can be made.

FOLLOW

For a non-terminal A, define $\textsc{follow}(A)$ 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 $\textsc{follow}(A)$:

1. Put \$ in $\textsc{follow}(\langle\text{goal}\rangle)$
2. If $A \rightarrow \alpha B \beta$:
   (a) Put $\textsc{first}(\beta) - \{\varepsilon\}$ in $\textsc{follow}(B)$
   (b) If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in \textsc{first}(\beta)$ (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW$(A)$ in FOLLOW(B)

   Repeat until no more additions can be made

LL(1) grammars

Previous definition

A grammar G is LL(1) iff. for all non-terminals A, each distinct pair of productions $A \to \beta$ and $A \to \gamma$ satisfy the condition $\text{FIRST}(\beta) \bigcap \text{FIRST}(\gamma) = \phi$.

What if $A \Rightarrow^* \varepsilon$?

Revised 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 all pairwise disjoint
2. If $\alpha_i \Rightarrow^* \varepsilon$ then $\text{FIRST}(\alpha_j) \bigcap \text{FOLLOW}(A) = \phi, \forall 1 \leq j \leq n, i \neq j$.

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 $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$

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

accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   (a) $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$
   (b) If $\varepsilon \in \text{FIRST}(\alpha)$:
      i. $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$
      ii. 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:

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

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

A grammar that is not LL(1)

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

Left-factored:

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

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

On seeing `else`, conflict between choosing

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

$\Rightarrow$ grammar is not LL(1)!

The fix:

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

Error recovery

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize
- When an error occurs looking for A, scan until an element of $\textsc{synch}(A)$ is found

Building SYNCH:

1. $a \in \textsc{follow}(A) \Rightarrow a \in \textsc{synch}(A)$
2. place keywords that start statements in $\textsc{synch}(A)$
3. add symbols in $\textsc{first}(A)$ to $\textsc{synch}(A)$

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

(i.e., $\textsc{synch}(a) = V_t - \{a\}$)