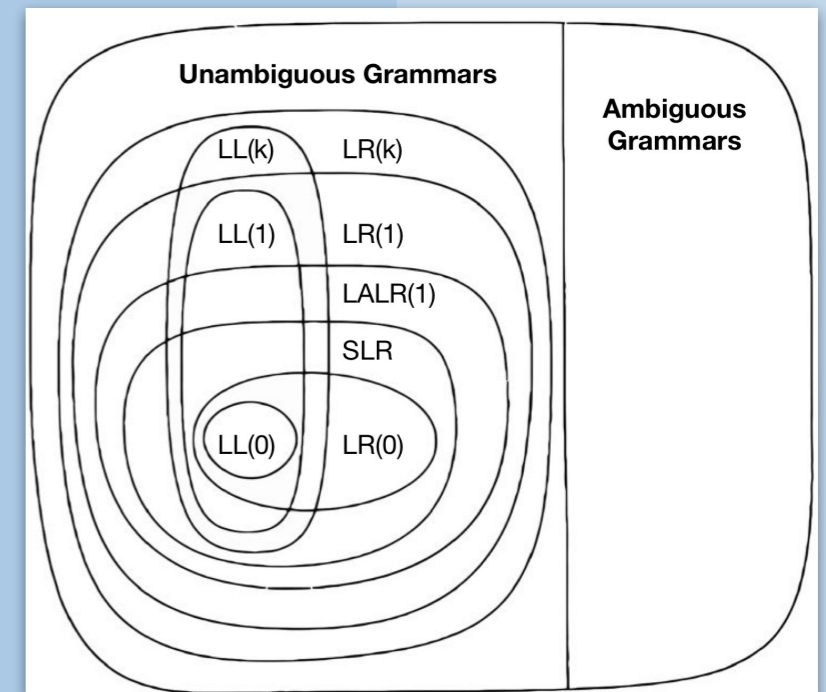


3. Parsing

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
<http://www.cs.ucla.edu/~palsberg/>
<http://www.cs.purdue.edu/homes/hosking/>



Roadmap

- > Context-free grammars
- > Derivations and precedence
- > Top-down parsing
- > Left-recursion
- > Look-ahead
- > Table-driven parsing



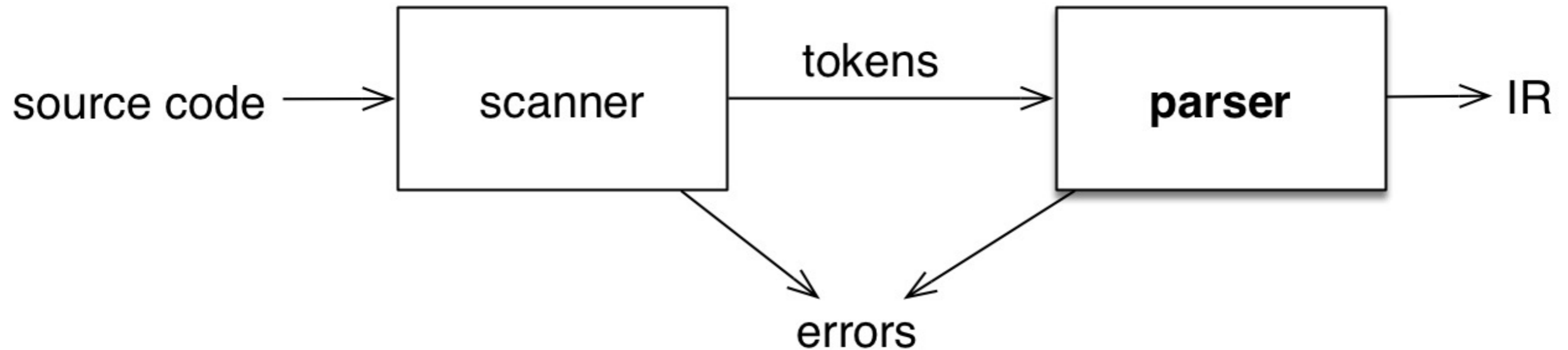
See, Modern compiler implementation in Java (Second edition), chapter 3.

Roadmap

- > **Context-free grammars**
- > Derivations and precedence
- > Top-down parsing
- > Left-recursion
- > Look-ahead
- > Table-driven parsing



The role of the parser



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

The role of the parser is to *recognize structure* in the stream of tokens produced by the scanner. To do this, the parser produces a parse — a derivation of the parse tree showing how the rules of the grammar can be used to produce the given input stream.

The output of the parser is some form of intermediate representation for the back end. This could be a parse tree, or it could be some other kind of intermediate language.

One important practical consideration is that the parser should not quit at the first error, but rather recover from errors to give as much useful feedback as possible concerning the entire input.

Languages and Operations (from lecture 2)

A language is a set of strings

<i>Operation</i>	<i>Definition</i>
Union	$L \cup M = \{ s \mid s \in L \text{ or } s \in M \}$
Concatenation	$LM = \{ st \mid s \in L \text{ and } t \in M \}$
Kleene closure	$L^* = \cup_{i=0, \infty} L^i$
Positive closure	$L^+ = \cup_{i=1, \infty} L^i$

Formally, a language is a set of strings (or “sentences”). We can perform various operations over languages, such as union, concatenation etc.

In the slide, L and M are languages, while s and t are strings. Operations over languages produce new languages by iterating over strings they contain.

The Kleene closure produces all possible concatenations of strings in a language L.

Examples:

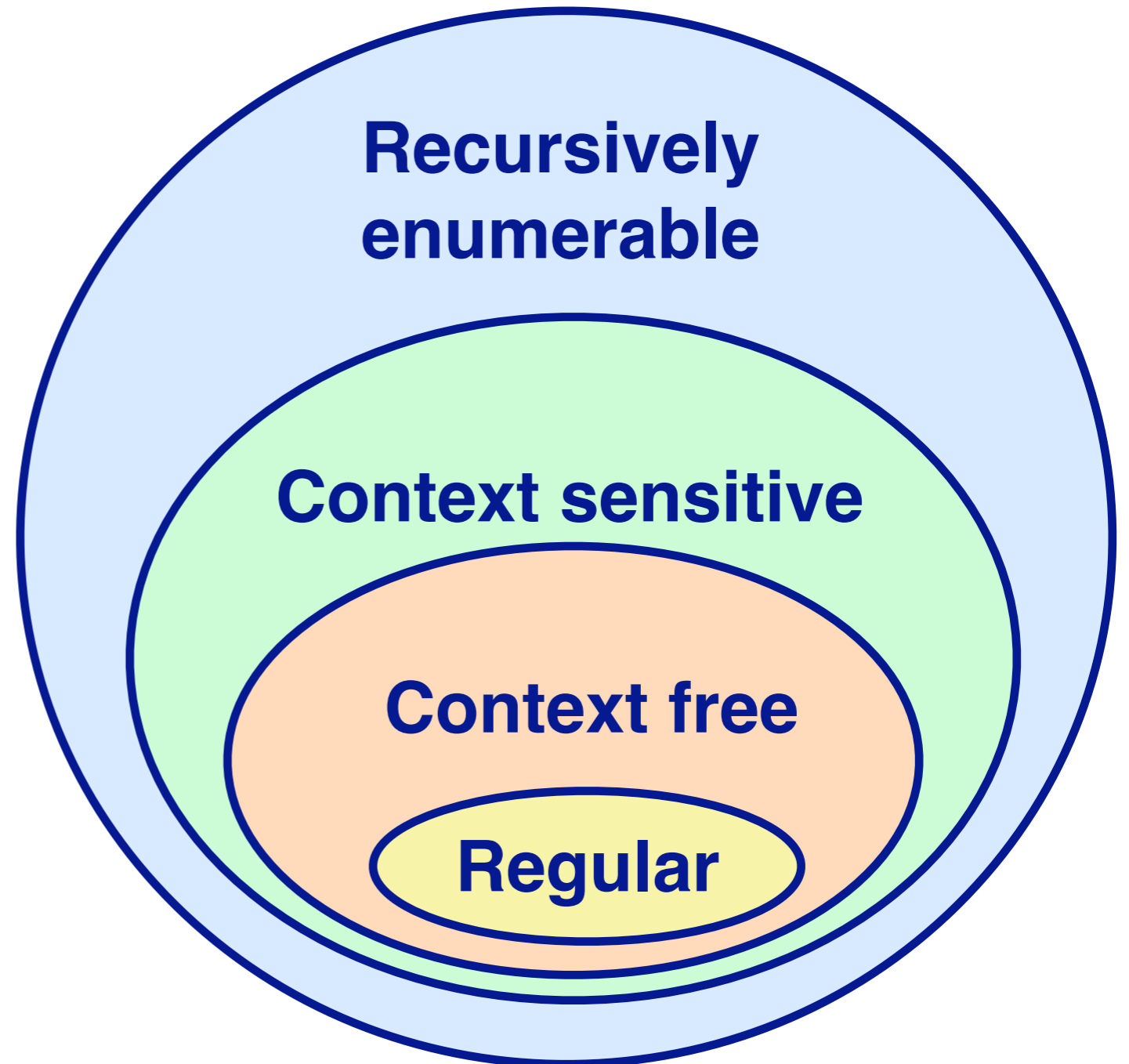
$$L = \{ a, b \}, M = \{ c, d \}$$

$$LM = \{ ac, ad, bc, bd \}$$

$$L^* = \{ \wedge, a, b, aa, ab, ba, bb, aaa, aab, aba, \dots \}$$

Production Grammars (from lecture 2)

- > Powerful formalism for language description
 - Start symbol (S_0)
 - Production rules ($A \rightarrow abA$)
 - Non-terminals (A, B)
 - Terminals (a, b)
- > Rewriting



A common way to specify languages is with the help of *production grammars*. These consist of a set of *rewrite rules* that allow you *generate* all possible strings in a language.

A grammar starts with a *start symbol* S_0 , and consists of a number of rules of the form

$$A \rightarrow abA$$

consisting of *non-terminals*, like S_0 and A , that can be *expanded* using further production rules, and *terminals*, like a and b , that cannot.

By repeated expanding terminals using different rules, one can generate all possible strings in the language specified by the grammar.

Detail: The Chomsky Hierarchy (from lecture 2)

> Type 0: $\alpha \rightarrow \beta$

—Unrestricted grammars generate recursively enumerable languages. Minimal requirement for recognizer: Turing machine.

> Type 1: $\alpha A \beta \rightarrow \alpha \gamma \beta$

—Context-sensitive grammars generate context-sensitive languages, recognizable by linear bounded automata

> Type 2: $A \rightarrow \gamma$

—Context-free grammars generate context-free languages, recognizable by non-deterministic push-down automata

> Type 3: $A \rightarrow a$ and $A \rightarrow aB$

—Regular grammars generate regular languages, recognizable by finite state automata

NB: A is a non-terminal; α , β , γ are strings of terminals and non-terminals

Since compilers need to recognize languages rather than generate them, we need a way to turn a grammar into a recogniser.

The Chomsky Hierarchy (named after Noam Chomsky) formalizes how different constraints over the production rules produce very different classes of languages. Unrestricted grammars (i.e., where the left and right-hand sides of the rules may contain a mix of terminals and non-terminals) are the hardest to parse, and require a Turing machine to recognize them.

Programming languages are mostly context-free (only non-terminals on the left-hand side), with occasionally some context-sensitive features. Typically the tokens of a programming language (i.e., identifiers, strings, comments etc.) are Type 3 and can be recognized by a FSA.

https://en.wikipedia.org/wiki/Chomsky_hierarchy

Syntax analysis

- > *Context-free syntax* is specified with a *context-free grammar*.
- > Formally a CFG $G = (V_t, V_n, S, P)$, where:
 - V_t is the set of *terminal* symbols in the grammar (i.e., the set of tokens returned by the scanner)
 - V_n , the *non-terminals*, are variables that denote sets of (sub)strings occurring in the language. These impose a structure on the grammar.
 - S is the *goal symbol*, a distinguished non-terminal in V_n denoting the entire set of strings in $L(G)$.
 - 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

Recalling the Chomsky hierarchy, most modern programming languages have a (mostly) context-free syntax. This makes them relatively easy to parse efficiently.

Notation and terminology

- > $a, b, c, \dots \in V_t$ (terminals)
- > $A, B, C, \dots \in V_n$ (non-terminals)
- > $U, V, W, \dots \in V$ (both)
- > $\alpha, \beta, \gamma, \dots \in V^*$ (sequences of terminals and non-terminals)
- > $u, v, w, \dots \in V_t^*$ (sequences of terminals only)

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

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

Syntax analysis

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

Example:

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> <op> <expr>
3.			num
4.			id
5.	<op>	::=	+
6.			-
7.			*
8.			/

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

This BNF describes simple expressions over numbers and identifiers. The terminals are num, id, +, -, * and /. Non-terminals are <goal> etc. The goal symbol here is <goal>. The BNF grammar rules can easily be re-written as formal productions:

$$\langle \text{goal} \rangle \rightarrow \langle \text{expr} \rangle$$
$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle$$
$$\langle \text{expr} \rangle \rightarrow \text{num}$$
$$\langle \text{expr} \rangle \rightarrow \text{id}$$

etc.

A sentence is any valid arithmetic expression, such as 3+4

Scanning vs. parsing

Where do we draw the line?

```
term ::= [a-zA-Z] ( [a-zA-Z] | [0-9] )*  
      | 0 | [1-9][0-9]*  
op   ::= + | - | * | /  
expr ::= (term op)* term
```

Regular expressions:

- Normally used to classify identifiers, numbers, keywords ...
- Simpler and more concise for tokens than a grammar
- More efficient scanners can be built from REs

CFGs are used to impose *structure*

- Brackets: (), begin ... end, if ... then ... else
- Expressions, declarations ...

Factoring out lexical analysis simplifies the compiler

As we have seen, context-free grammars are strictly more powerful than regular grammars. It is therefore always possible to express a grammar for both the lexical and syntactic aspects of a language with a single (scannerless) grammar.

The advantages are (1) only a single formalism (and tool) is needed, and (2) different tokens can be used for different sublanguages, such as embedded domain-specific languages (e.g., a query language).

The disadvantages are (1) the grammar will be much more complex (syntactic analysis is complicated enough: the grammar for C has around 200 productions), and (2) parsing becomes less efficient.

Hierarchy of grammar classes

Unambiguous Grammars

LL(k)

LR(k)

LL(1)

LR(1)

LALR(1)

SLR

LL(0)

LR(0)

Ambiguous Grammars

LL(k):

- Left-to-right, **L**eftmost derivation, k tokens lookahead, *top-down*

LR(k):

- Left-to-right, **R**ightmost derivation, k tokens lookahead, *bottom-up*

SLR:

- **S**imple **LR** (uses “follow sets”)

LALR:

- **L**ook**A**head **LR** (uses “lookahead sets”)



There exist many different sub-categories of context-free grammars. For practical purposes it is important that a grammar be *unambiguous*, i.e., that it always produces a unique parse for a given valid input.

Although parsers read their input Left to Right (the first “L” in most of these categories), they may work either *top-down* — producing a *leftmost derivation* — or *bottom-up* — producing a *rightmost derivation*. (More on this later.)

They may also require some number of tokens of “*lookahead*” to decide which production rule to apply at any point without backtracking.

LL(1) and LR(1) are “sweet spots” that allow interesting languages to be specified, but can also be parsed efficiently.

Roadmap

- > Context-free grammars
- > **Derivations and precedence**
- > Top-down parsing
- > Left-recursion
- > Look-ahead
- > Table-driven parsing



Derivations

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

$\langle \text{goal} \rangle$	\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	$\langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

We have derived the sentence: $x + 2 * y$

We denote this derivation (or parse) as: $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$

The process of discovering a derivation is called parsing.

Derivation

- > At each step, we choose a non-terminal to replace.
— *This choice can lead to different derivations.*
- > Two strategies are especially interesting:
 1. *Leftmost derivation*: replace the leftmost non-terminal at each step
 2. *Rightmost derivation*: replace the rightmost non-terminal at each step

The previous example was a leftmost derivation.

As we shall see, a *leftmost derivation* corresponds to *top-down parsing*, and is especially well-suited to recursive descent parsers. A *rightmost derivation*, on the other hand, is produced *bottom-up*, and is better-suited to table-driven parsing.

Rightmost derivation

For the string: $x + 2 * y$

$\langle \text{goal} \rangle$	\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	$\langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

Again we have: $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$

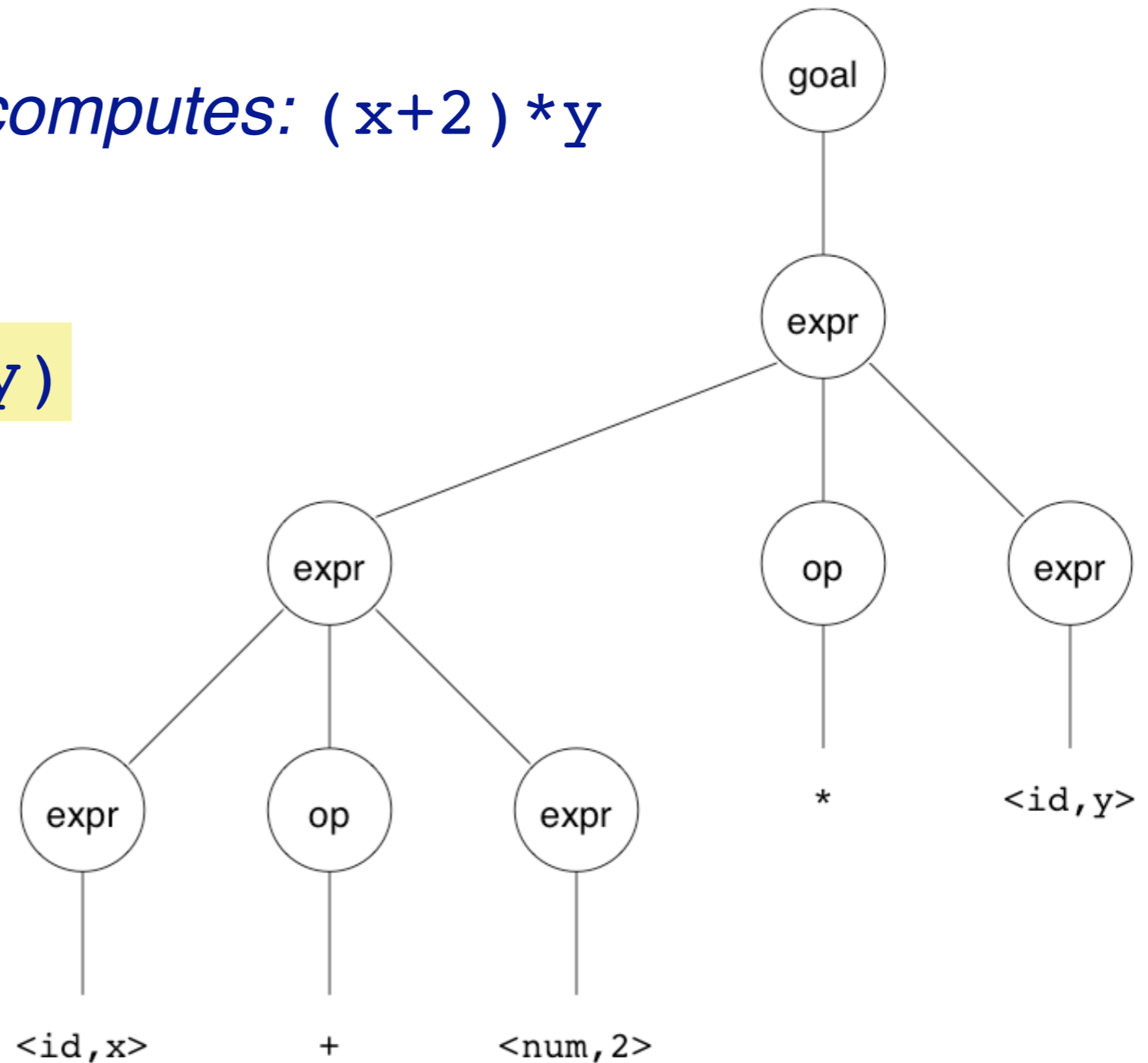
Here we see that the rightmost non-terminal (in **bold**) is expanded in the following line.

As we shall see, the trick is know which rule to use to expand the non-terminal. This requires some analysis of the grammar to generate lookup tables, and some lookahead of input symbols.

Precedence

*Treewalk evaluation computes: $(x+2) * y$*

*Should be: $x + (2 * y)$*



Precedence

- > **Our grammar has a problem:** it has *no notion of precedence*, or implied order of evaluation.
- > To add precedence takes additional machinery:

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

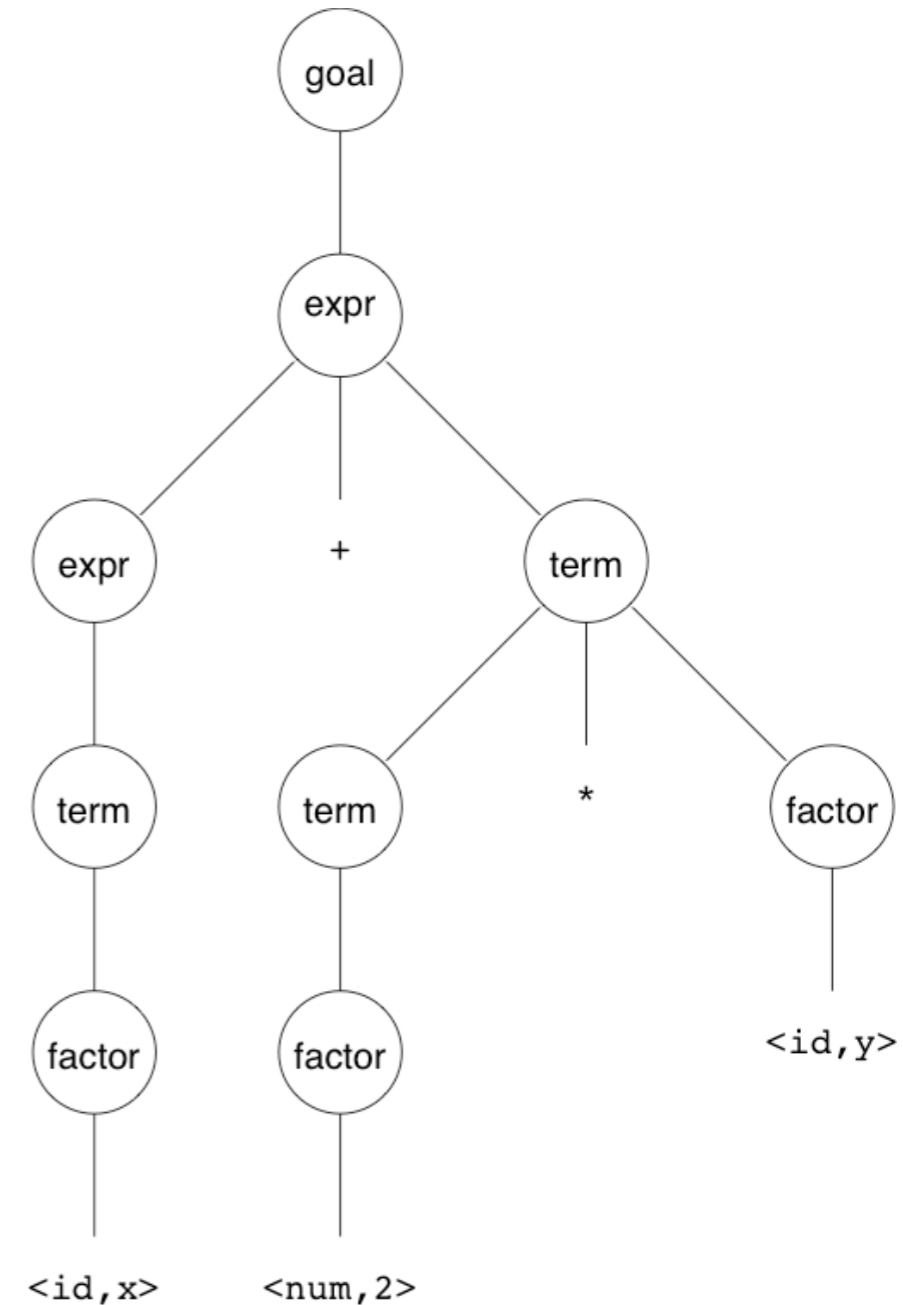
- > This grammar enforces a precedence on the derivation:
 - terms *must* be derived from expressions
 - forces the “correct” tree

Forcing the desired precedence

Now, for the string: $x + 2 * y$

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

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



This time it is impossible to go wrong. Not only do we force the right precedence, but there is only a unique parse possible.

However it is still not clear how we choose to expand the rightmost non-terminal without backtracking.

Ambiguity

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

```
<stmt> ::= if <expr> then <stmt>
        |  if <expr> then <stmt> else <stmt>
        |  ...
```

- > Consider: $\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$
 - This has two derivations
 - The ambiguity is purely grammatical
 - It is called a context-free ambiguity

“context-free ambiguity” = ambiguity in a context-free grammar.

Note that the rules share a large common prefix, which means that you have to proceed quite far before you could detect a mistake or an ambiguity in parsing.

Resolving ambiguity

Ambiguity may be eliminated by rearranging the grammar:

```
<stmt> ::= <matched>
        | <unmatched>
<matched> ::= if <expr> then <matched> else <matched>
           | ...
<unmatched> ::= if <expr> then <stmt>
              | if <expr> then <matched> else <unmatched>
```

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

— *match each else with the closest unmatched then*

Ambiguity

- > Ambiguity is often due to confusion in the context-free specification. Confusion can arise from *overloading*, e.g.:

$$a = f(17)$$

- > In many Algol-like languages, f could be a function or a subscripted variable.
- > Disambiguating this statement *requires context*:
 - need *values* of declarations
 - not *context-free*
 - really an issue of *type*

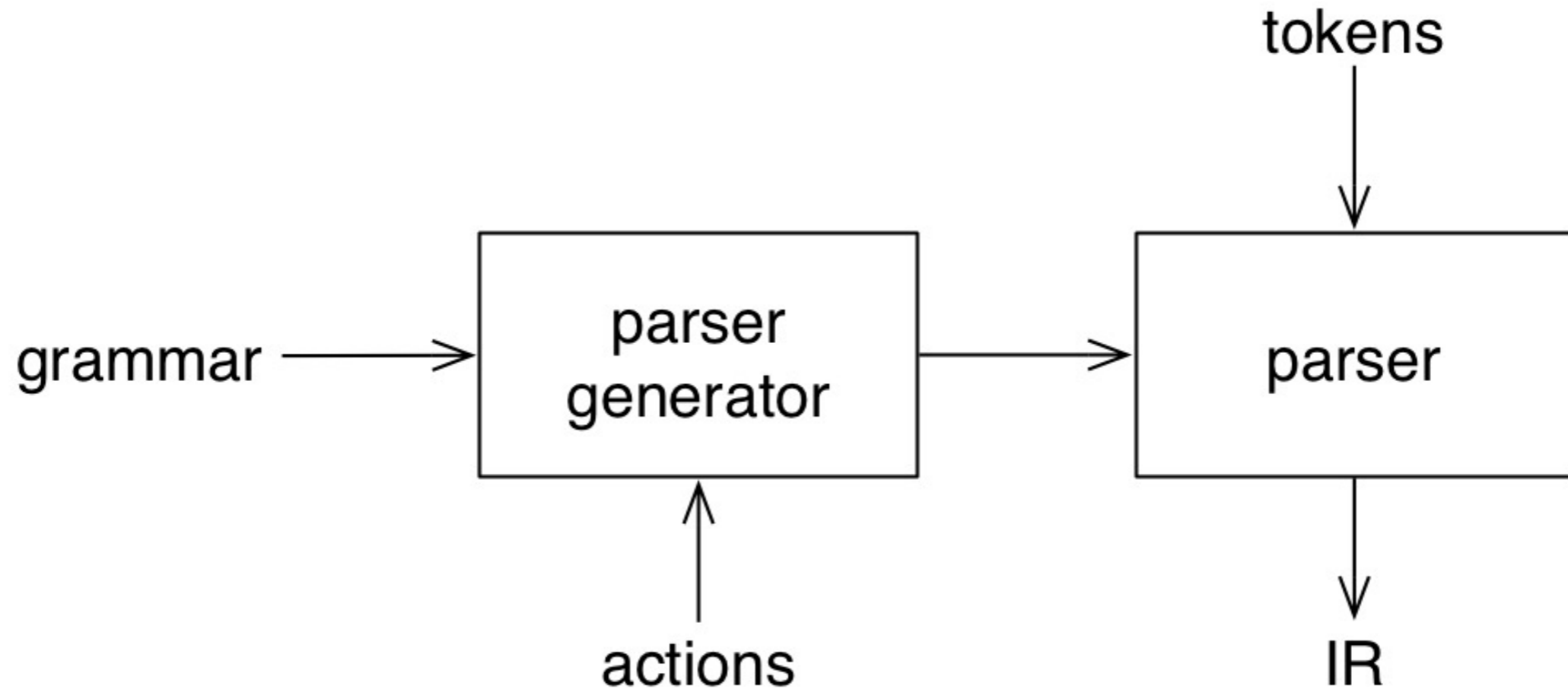
Rather than complicate parsing, we will handle this separately.

Roadmap

- > Context-free grammars
- > Derivations and precedence
- > **Top-down parsing**
- > Left-recursion
- > Look-ahead
- > Table-driven parsing



Parsing: the big picture



Our goal is a flexible parser generator system

Top-down versus bottom-up

> *Top-down parser (LL):*

- starts at the root of derivation tree and fills in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

> *Bottom-up parser (LR):*

- starts at the leaves and fills in
- starts in a state valid for legal first tokens
- as input is consumed, changes state to encode possibilities (*recognize valid prefixes*)
- uses a *stack* to store both state and sentential forms

LL parsers are top-down. LR parsers are bottom-up.

Bottom-up parsers are normally built by parser generators.

Top-down parsers can be either hand-written or generated.

We are interested in automating the construction of both top-down and bottom-up parsers directly from the grammar specifications.

Since a parser does not just parse, but must also produce a suitable IR, we must decorate the grammar with additional actions that the generated parser will take.

Top-down parsing

A top-down parser starts with the root of the parse tree, labeled 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 labeled A , select a production $A \rightarrow \alpha$ 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. $\langle \text{goal} \rangle ::= \langle \text{expr} \rangle$
2. $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$
3. $\quad \quad \quad | \langle \text{expr} \rangle - \langle \text{term} \rangle$
4. $\quad \quad \quad | \langle \text{term} \rangle$
5. $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle$
6. $\quad \quad \quad | \langle \text{term} \rangle / \langle \text{factor} \rangle$
7. $\quad \quad \quad | \langle \text{factor} \rangle$
8. $\langle \text{factor} \rangle ::= \text{num}$
9. $\quad \quad \quad | \text{id}$

Consider the input string $x - 2 * y$

Top-down derivation

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

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. $\langle \text{factor} \rangle ::= \text{num}$
9. | id

The horizontal lines denote the backtracking points. Whenever a token cannot be read, or input is left, then we must backtrack to an alternative rule.

- start with $\langle \text{goal} \rangle$, input before x
- $\langle \text{goal} \rangle \rightarrow \langle \text{expr} \rangle$
- $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle$ [NB: choice of productions]
- ...
- consume $\text{id} = x$; fail at $+ = -$
- backtrack and redo: $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle - \langle \text{term} \rangle$

NB: This example does not show how we pick which rule to expand! (be patient)

Roadmap

- > Context-free grammars
- > Derivations and precedence
- > Top-down parsing
- > **Left-recursion**
- > Look-ahead
- > Table-driven parsing



Non-termination

Another possible parse for $x - 2 * y$

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

If the parser makes the wrong choices, expansion doesn't terminate!

Left-recursion

Top-down parsers cannot handle left-recursion in a grammar

Formally, a grammar is left-recursive if

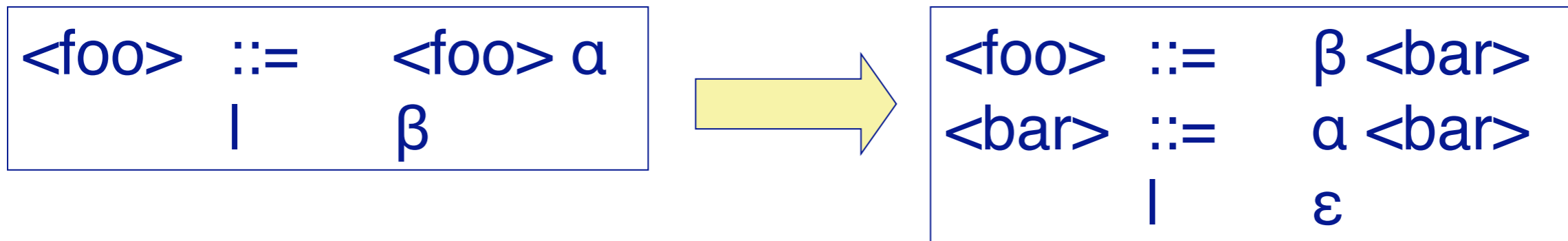
$\exists A \in V_n$ such that $A \Rightarrow^+ A\alpha$ for some string α

Our simple expression grammar is left-recursive!

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

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

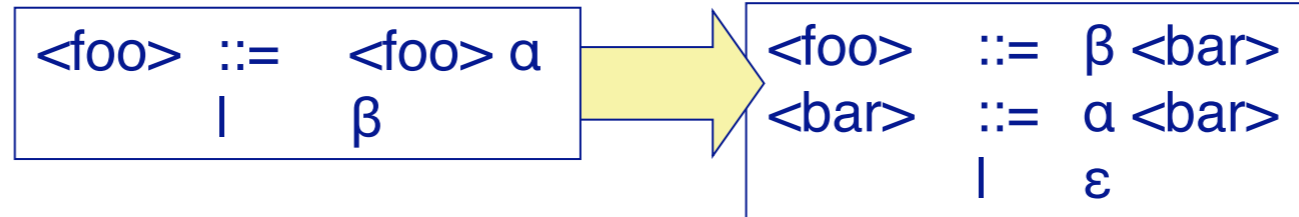


NB: α and β do not start with $\langle \text{foo} \rangle$!

*How would you write $\langle \text{foo} \rangle$
as a regular expression?*

It can be easier to see how this works by reasoning about what `<foo>` actually represents. How would you write it as a regular expression?

Example



$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$
 $| \langle \text{expr} \rangle - \langle \text{term} \rangle$
 $| \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $| \langle \text{term} \rangle / \langle \text{factor} \rangle$
 $| \langle \text{factor} \rangle$

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle \text{expr}' \rangle$
 $\langle \text{expr}' \rangle ::= + \langle \text{term} \rangle \langle \text{expr}' \rangle$
 $| - \langle \text{term} \rangle \langle \text{expr}' \rangle$
 $| \epsilon$

$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle$
 $\langle \text{term}' \rangle ::= * \langle \text{factor} \rangle \langle \text{term}' \rangle$
 $| / \langle \text{factor} \rangle \langle \text{term}' \rangle$
 $| \epsilon$

Example

Our long-suffering expression grammar :

With this grammar, a top-down parser will

- *terminate*
- *backtrack on some inputs*

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.			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.			$/ \langle \text{factor} \rangle \langle \text{term}' \rangle$
9.			ϵ
10.	$\langle \text{factor} \rangle$::=	num
11.			id

Example

This cleaner grammar defines the same language:

1.	<code><goal></code>	<code>::=</code>	<code><expr></code>
2.	<code><expr></code>	<code>::=</code>	<code><term> + <expr></code>
3.		<code> </code>	<code><term> - <expr></code>
4.		<code> </code>	<code><term></code>
5.	<code><term></code>	<code>::=</code>	<code><factor> * <term></code>
6.		<code> </code>	<code><factor> / <term></code>
7.		<code> </code>	<code><factor></code>
8.	<code><factor></code>	<code>::=</code>	<code>num</code>
9.		<code> </code>	<code>id</code>

It is:

- *right-recursive*
- *free of ϵ productions*

Unfortunately, it generates different associativity.

Same syntax, different meaning!

How would you parse “ $9 - 5 + 3$ ” with the two grammars?

Roadmap

- > Context-free grammars
- > Derivations and precedence
- > Top-down parsing
- > Left-recursion
- > **Look-ahead**
- > Table-driven parsing



How much look-ahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary look-ahead 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, Left-most derivation, 1-token look-ahead; and
- LR(1)**: Left to right scan, Right-most derivation, 1-token look-ahead

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 $\text{FIRST}(\alpha)$ as the set of tokens that appear first in some string derived from α

I.e., 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:

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

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

The example grammar has this property!

Example — computing the FIRST set

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

$\text{FIRST}(S) = \text{FIRST}(E)$
 $= \text{FIRST}(TE')$
 $= \text{FIRST}(T)$ *since $\varepsilon \notin \text{FIRST}(T)$*

$\text{FIRST}(T) = \text{FIRST}(FT')$
 $= \text{FIRST}(F)$ *since $\varepsilon \notin \text{FIRST}(F)$*

$\text{FIRST}(E') = \{ +, -, \varepsilon \}$

$\text{FIRST}(T') = \{ *, /, \varepsilon \}$

$\text{FIRST}(F) = \{ \text{num}, \text{id} \}$

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

To compute the FIRST sets, we simply start from the goal and recursively compute the FIRST sets of all the non-terminals by expanding the rules.

If $A \rightarrow B|C$ then $\text{FIRST}(A) = \text{FIRST}(B) \cup \text{FIRST}(C)$.

If $A \rightarrow BC$ then $\text{FIRST}(A) = \text{FIRST}(BC)$.

If $\varepsilon \notin \text{FIRST}(B)$ then $\text{FIRST}(A) = \text{FIRST}(B)$, else $\text{FIRST}(A) = \text{FIRST}(C)$.

In the example we see that $\text{FIRST}(S) = \text{FIRST}(TE')$. Since we (later) show that $\varepsilon \notin \text{FIRST}(T)$, we can conclude that $\text{FIRST}(S) = \text{FIRST}(T) = \text{FIRST}(F) = \{ \text{num}, \text{id} \}$

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 $\alpha \neq \epsilon$ then replace all of the A productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$$

with

$$A \rightarrow \alpha A'$$

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

where A' is fresh

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

Example

Consider our *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>	::=	num
9.			id

To choose between productions 2, 3, & 4, the parser must see past the num or id and look at the +, −, * or /.

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

This grammar *fails* the test.

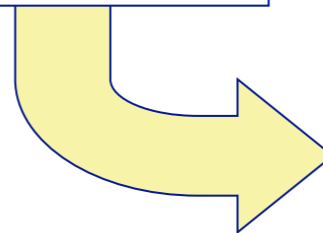
I.e., they all have the same FIRST set, namely { num, id }

NB: This grammar is right-associative.

Example

Two non-terminals must be left-factored:

<code><expr></code>	<code>::=</code>	<code><term> + <expr></code>
	<code> </code>	<code><term> - <expr></code>
	<code> </code>	<code><term></code>
<code><term></code>	<code>::=</code>	<code><factor> * <term></code>
	<code> </code>	<code><factor> / <term></code>
	<code> </code>	<code><factor></code>



<code><expr></code>	<code>::=</code>	<code><term> <expr'></code>
<code><expr'></code>	<code>::=</code>	<code>+ <expr></code>
	<code> </code>	<code>- <expr></code>
	<code> </code>	<code>ϵ</code>
<code><term></code>	<code>::=</code>	<code><factor> <term'></code>
<code><term'></code>	<code>::=</code>	<code>* <term></code>
	<code> </code>	<code>/ <term></code>
	<code> </code>	<code>ϵ</code>

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.		$ $	$- \langle \text{expr} \rangle$
5.		$ $	ϵ
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.		$ $	ϵ
10.	$\langle \text{factor} \rangle$	$::=$	num
11.		$ $	id

Now, selection requires only a single token look-ahead.

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

NB: This is a *different grammar* than the one we obtained by factoring out left recursion in the previous chapter.

Example derivation

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

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. $\quad \quad \quad | - \langle \text{expr} \rangle$
5. $\quad \quad \quad | \varepsilon$
6. $\langle \text{term} \rangle ::= \langle \text{factor} \rangle \langle \text{term}' \rangle$
7. $\langle \text{term}' \rangle ::= * \langle \text{term} \rangle$
8. $\quad \quad \quad | / \langle \text{term} \rangle$
9. $\quad \quad \quad | \varepsilon$
10. $\langle \text{factor} \rangle ::= \text{num}$
11. $\quad \quad \quad | \text{id}$

The next symbol determines 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

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

with

$$A \rightarrow NA'$$

$$N \rightarrow \beta \mid \dots \mid \gamma$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

where N and A' are fresh

—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 look-ahead?

> **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:

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

S	:=	R0 R1
R0	:=	a R0 b 0
R1	:=	a R1 bb 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;
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

Roadmap

- > Context-free grammars
- > Derivations and precedence
- > Top-down parsing
- > Left-recursion
- > Look-ahead
- > **Table-driven parsing**

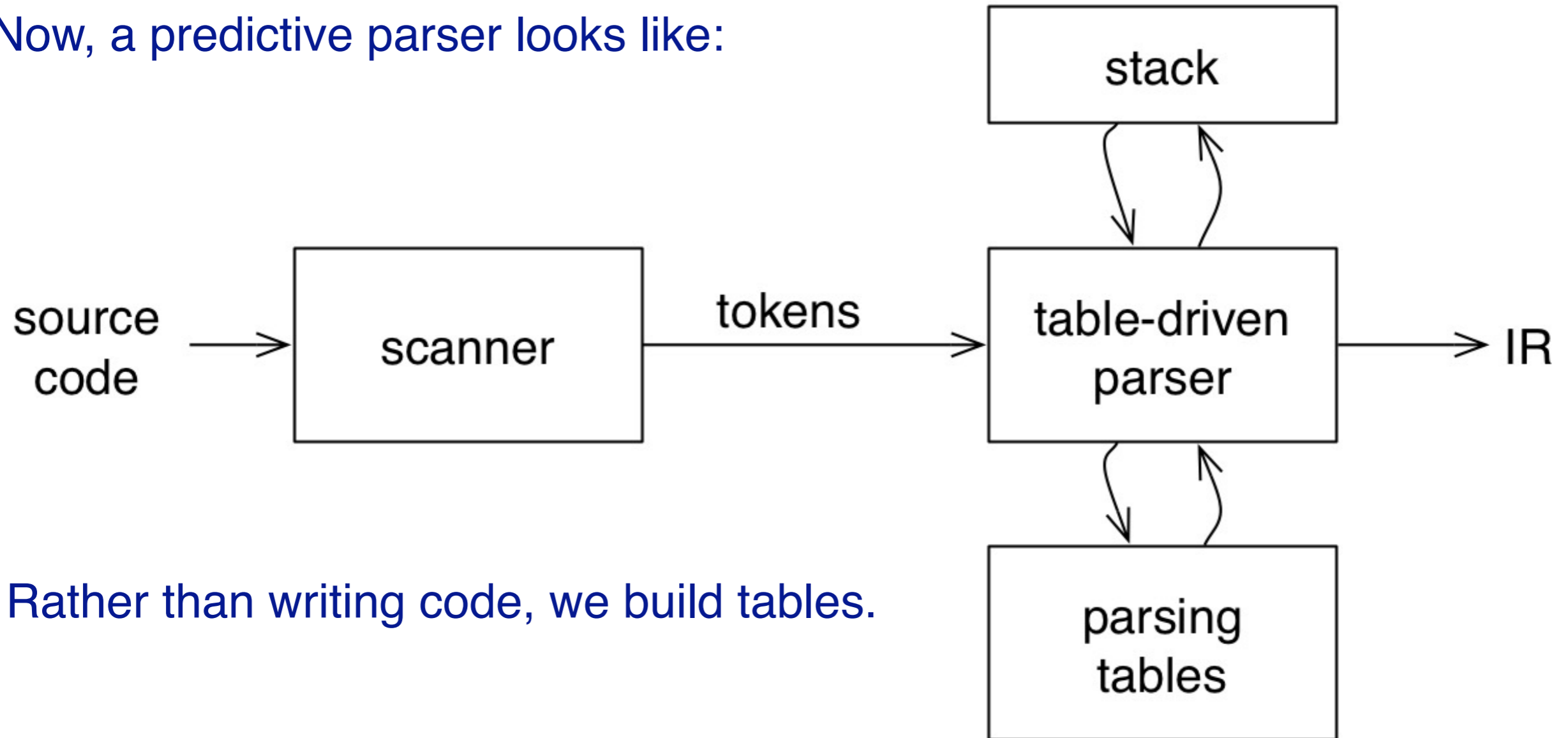


Non-recursive predictive parsing

- > 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:

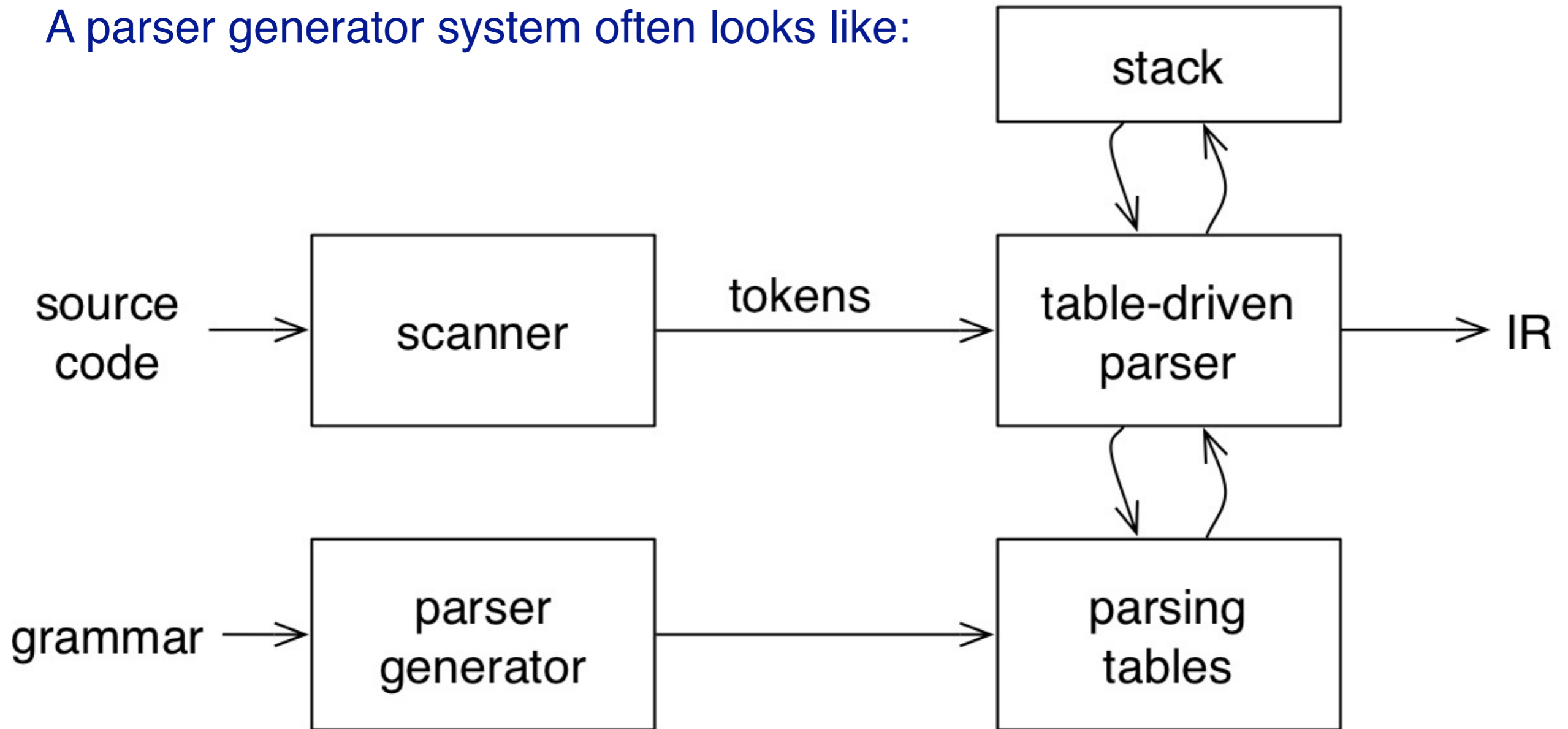


Rather than writing code, we build tables.

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  $M[X, token] = X \rightarrow Y_1 Y_2 \dots Y_k$  then
      pop X
      push  $Y_k, Y_{k-1}, \dots, Y_1$ 
    else error()
until X = EOF
```


tos = “top of stack”

The top of the stack holds the current symbol (terminal or non-terminal) you are trying match.

The bottom of the stack is the target. The lookahead tells you which rule to use to expand a NT, and then the top of stack is replaced by pushing all the symbols of the RHS of the rule.

The algorithm repeatedly compares the input token to the top of the stack. If the top is a token, then we pop and proceed, else we have an error. If the top is a non-terminal, then we use the parsing table to decide how to expand the NT, and we update the stack accordingly.

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. | $- \langle \text{expr} \rangle$
5. | ε
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. | ε
10. $\langle \text{factor} \rangle ::= \text{num}$
11. | 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

Read the parse table as follows:

If the currently expected NT is $\langle \text{goal} \rangle$, then the next token must be either an id or a num (everything else is an error). In either case we expand rule 1. $\langle \text{expr} \rangle$ is similar.

With $\langle \text{expr}' \rangle$ we expect $+$, $-$ or $\$$ (end of input), and expand, respectively, rule 3, 4 or 5.

LL(1) grammars

Previous definition:

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

> But what if $A \Rightarrow^* \varepsilon$?

Revised definition:

—A grammar G is LL(1) iff for each set of productions

$$A \rightarrow a_1 \mid a_2 \mid \dots \mid a_n$$

1. $\text{FIRST}(a_1), \text{FIRST}(a_2), \dots, \text{FIRST}(a_n)$ are pairwise disjoint
2. If $a_i \Rightarrow^* \varepsilon$ then $\text{FIRST}(a_j) \cap \text{FOLLOW}(A) = \emptyset, \forall 1 \leq j \leq n, i \neq j$

NB: If G is ε -free, condition 1 is sufficient

FOLLOW(A) must be disjoint from FIRST(a_j), else we do not know whether to go to a_j or to take a_j and skip to what follows.

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

Repeat until no more additions can be made.

This is a straightforward recursive algorithm: to compute $\text{FIRST}(X)$, expand each individual rule till you reach an initial terminal. Take care to only add ϵ if the entire rule can reduce to ϵ .

FOLLOW

- > For a non-terminal A , define $FOLLOW(A)$ as:
 - the set of terminals that can appear immediately to the right of A in some sentential form
 - I.e., a non-terminal's $FOLLOW$ set specifies the tokens that can legally appear after it.
 - A terminal symbol has no $FOLLOW$ set.
- > To build $FOLLOW(A)$:
 1. Put $\$$ in $FOLLOW(\langle goal \rangle)$
 2. If $A \rightarrow \alpha B \beta$:
 - a) Put $FIRST(\beta) - \{\epsilon\}$ in $FOLLOW(B)$
 - b) If $\beta = \epsilon$ (i.e., $A \rightarrow \alpha B$) or $\epsilon \in FIRST(\beta)$ (i.e., $\beta \Rightarrow^* \epsilon$) then put $FOLLOW(A)$ in $FOLLOW(B)$

Repeat until no more additions can be made

Example — computing the FOLLOW set

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

	FIRST	FOLLOW
S	{num, id}	{ $\$$ }
E	{num, id}	{ $\$$ }
E'	{ ε , +, -}	{ $\$$ }
T	{num, id}	{+, -, $\$$ }
T'	{ ε , *, /}	{+, -, $\$$ }
F	{num, id}	{+, -, *, /, $\$$ }
id	{id}	—
num	{num}	—
*	{*}	—
/	{/}	—
+	{+}	—
-	{-}	—

(start)

$S \rightarrow E$

$E \rightarrow TE'$

Add $\$$ to FOLLOW(S).

Add FOLLOW(S) to FOLLOW(E).

Add FIRST(E') - { ε } to FOLLOW(T).

Add FOLLOW(E) to FOLLOW(E').

$\varepsilon \in \text{FIRST}(E')$, so add FOLLOW(E) to FOLLOW(T).

$E' \rightarrow +E \mid -E \mid \varepsilon$

Add FOLLOW(E') to FOLLOW(E). [noop]

$T \rightarrow FT'$

Add FIRST(T') - { ε } to FOLLOW(F).

Add FOLLOW(T) to FOLLOW(T').

$\varepsilon \in \text{FIRST}(T')$, so add FOLLOW(T) to FOLLOW(F).

$T' \rightarrow *T \mid /T \mid \varepsilon$

Add FOLLOW(T') to FOLLOW(T). [noop]

$F \rightarrow \text{num} \mid \text{id}$

[noop]

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. \forall production $A \rightarrow \alpha$:

a) $\forall a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$

b) If $\epsilon \in \text{FIRST}(\alpha)$:

i. $\forall b \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$

ii. If $\$ \in \text{FOLLOW}(A)$, 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 G is not LL(1).

NB: recall that $a, b \in V_t$, so $a, b \neq \epsilon$

Example

Our long-suffering expression grammar:

$$\begin{aligned}
 S &\rightarrow E \\
 E &\rightarrow TE' \\
 E' &\rightarrow +E \mid -E \mid \varepsilon \\
 T &\rightarrow FT' \\
 T' &\rightarrow *T \mid /T \mid \varepsilon \\
 F &\rightarrow \text{num} \mid \text{id}
 \end{aligned}$$

	FIRST	FOLLOW
S	{num, id}	{ $\$$ }
E	{num, id}	{ $\$$ }
E'	{ ε , +, -}	{ $\$$ }
T	{num, id}	{+, -, $\$$ }
T'	{ ε , *, /}	{+, -, $\$$ }
F	{num, id}	{+, -, *, /, $\$$ }
id	{id}	-
num	{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	$F \rightarrow \text{id}$	$F \rightarrow \text{num}$	-	-	-	-	-

$S \rightarrow E$

FIRST(E) = { id,num } so add to M[S,id] and M[S,num]

$E \rightarrow TE' \text{ — } \textit{easy}$

$T \rightarrow FT' \text{ — } \textit{easy}$

$E' \rightarrow +E \mid -E \mid \varepsilon$

Add to M[E',+] and M[E',-]. $\varepsilon \in \text{FIRST}(\varepsilon)$, so add to M[E',\\$]

$T' \rightarrow *T \mid /T \mid \varepsilon$

Add to M[T',+] and M[T',-], but also to each b in FOLLOW(T').

$F \rightarrow \text{num} \mid \text{id} \text{ — } \textit{easy}$

Properties of 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. An 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)

A grammar that is not LL(1)

```
<stmt> ::= if <expr> then <stmt>
          | if <expr> then <stmt> else <stmt>
          | ...
```

Left-factored:

```
<stmt> ::= if <expr> then <stmt> <stmt'> | ...
<stmt'> ::= else <stmt> | ε
```

Now, $\text{FIRST}(\langle \text{stmt}' \rangle) = \{ \varepsilon, \text{else} \}$

Also, $\text{FOLLOW}(\langle \text{stmt}' \rangle) = \{ \text{else}, \$ \}$

But, $\text{FIRST}(\langle \text{stmt}' \rangle) \cap \text{FOLLOW}(\langle \text{stmt}' \rangle) = \{ \text{else} \} \neq \emptyset$

On seeing `else`, conflict between choosing

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

\Rightarrow grammar is not LL(1)!

Note that since $\langle \text{stmt} \rangle$ precedes $\langle \text{stmt}' \rangle$, by recursion $\langle \text{stmt}' \rangle$ precedes $\langle \text{stmt}' \rangle$, so $\text{FIRST}(\langle \text{stmt}' \rangle)$ is in $\text{FOLLOWS}(\langle \text{stmt}' \rangle)$.

NB: The fix, as before, is to put priority on $\langle \text{stmt}' \rangle ::= \text{else}$ $\langle \text{stmt} \rangle$ 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 $\text{SYNC}(A)$ is found

Building $\text{SYNC}(A)$:

1. $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNC}(A)$
2. place keywords that start statements in $\text{SYNC}(A)$
3. add symbols in $\text{FIRST}(A)$ to $\text{SYNC}(A)$









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., $\text{SYNC}(a) = V_t - \{a\}$

NB: popping the terminal means we matched it – since it wasn't really there, in effect we have inserted it

What you should know!

-  *What are the key responsibilities of a parser?*
-  *How are context-free grammars specified?*
-  *What are leftmost and rightmost derivations?*
-  *When is a grammar ambiguous? How do you remove ambiguity?*
-  *How do top-down and bottom-up parsing differ?*
-  *Why are left-recursive grammar rules problematic?*
-  *How do you left-factor a grammar?*
-  *How can you ensure that your grammar only requires a look-ahead of 1 symbol?*

Can you answer these questions?

- ✎ Why is it important for programming languages to have a context-free syntax?*
- ✎ Which is better, leftmost or rightmost derivations?*
- ✎ Which is better, top-down or bottom-up parsing?*
- ✎ Why is look-ahead of just 1 symbol desirable?*
- ✎ Which is better, recursive descent or table-driven top-down parsing?*
- ✎ Why is LL parsing top-down, but LR parsing is bottom up?*



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<http://creativecommons.org/licenses/by-sa/4.0/>