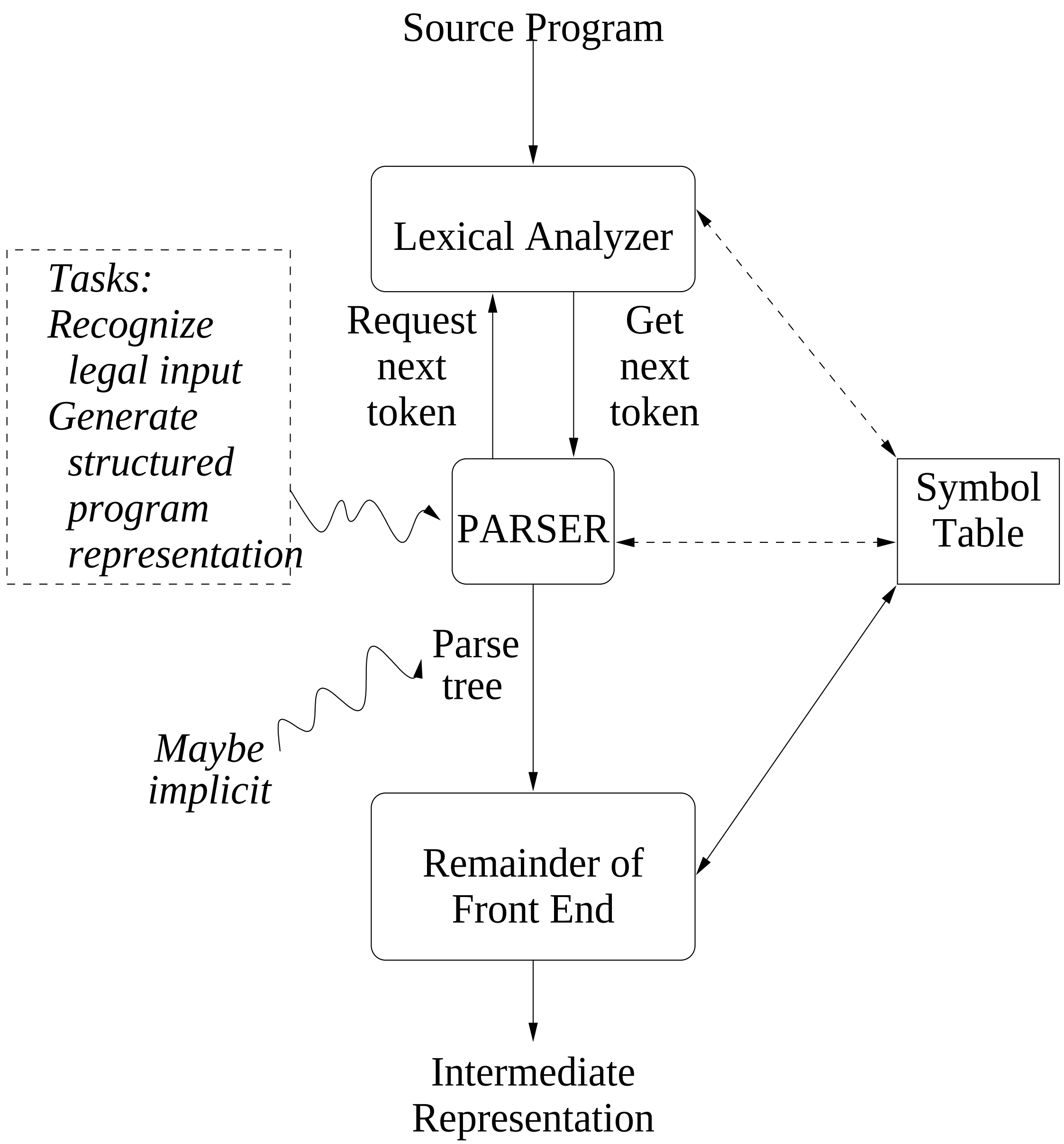
CS321 Languages and Compiler Design I

Fall 2010

Lecture 6

Syntax Analysis (Parsing)



SYNTAX ANALYSIS

Specify legal program formats using **context-free grammar (CFG)**

- Use **B**ackus-**N**aur **F**orm (BNF) as notation.
- Gives precise, readable specification.
- Many CFG's have efficient **parsers**.
- Parser **recognizes** syntactically legal programs and **rejects** illegal ones.

Successful parse also captures **hierarchical** structure of programs (expressions, blocks, etc.).

- Convenient representation for further semantic checking (e.g., types) and for code generation.

We can use another program generator (e.g., `yacc` or `CUP`) to generate parser automatically from grammar.

GRAMMARS

A **Context-free Grammar** is described by a set of **productions** (also called **rewrite rules**), e.g.:

$$stmt \rightarrow \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt$$

$$expr \rightarrow expr\ \texttt{+}\ expr \mid expr\ \texttt{*}\ expr \mid \texttt{(}expr\texttt{)} \mid \texttt{-}expr \mid \texttt{id}$$

Grammars contain **terminals** ($\equiv$ **tokens**) (e.g. `if`,`+`,`id`) and **non-terminals** (e.g., *expr*,*stmt*).

Grammars have a distinguished **start symbol** (ordinarily listed first, e.g., *stmt*).

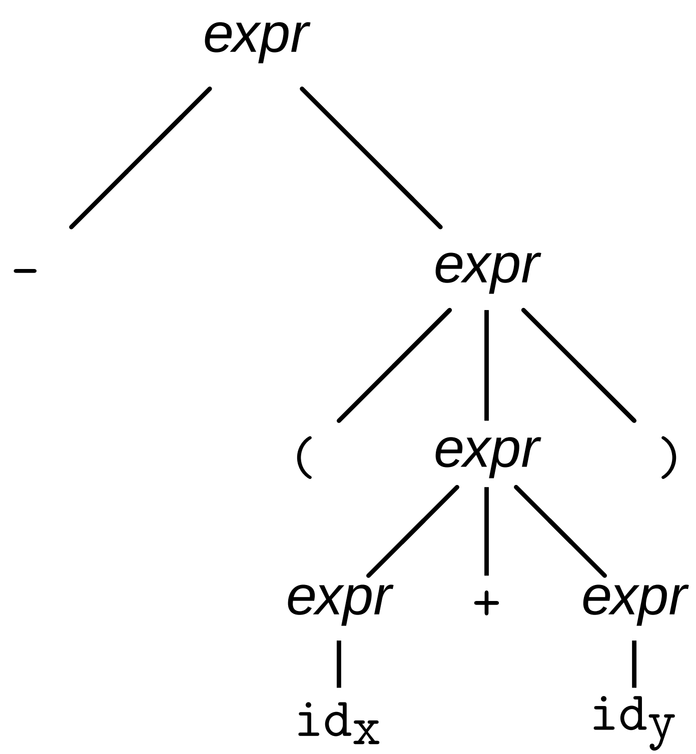
The language **generated** by a grammar is the set of **sentences** (strings) of terminals that can be **derived** by repeated application of productions, beginning with **start symbol**.

- We write $L(G)$ for the language generated by grammar G.

PARSE TREES

A **parse tree** is the graphical representation of a derivation.

Example tree for derivation of sentence `-(`$\mathtt{id}_x$` + `$\mathtt{id}_y$`)` :



Each application of a production corresponds to an **internal** node, labeled with a **non-terminal**.

Leaves are labeled with **terminals**, possibly with attributes.

The derived sentence is found by reading leaves left-to-right.

Derivations

Can "linearize" a parse tree into a sequence of one-step derivations.

Example:

$$
\begin{aligned}
\textit{expr} &\Rightarrow \texttt{-}\ \textit{expr} \\
&\Rightarrow \texttt{-(}\textit{expr}\texttt{)} \\
&\Rightarrow \texttt{-(}\textit{expr}\ \texttt{+}\ \textit{expr}\texttt{)} \\
&\Rightarrow \texttt{-(id}_{\texttt{x}}\ \texttt{+}\ \textit{expr}\texttt{)} \\
&\Rightarrow \texttt{-(id}_{\texttt{x}}\ \texttt{+}\ \texttt{id}_{\texttt{y}}\texttt{)}
\end{aligned}
$$

or

$$\textit{expr} \stackrel{*}{\Rightarrow} \texttt{-(id}_{\texttt{x}}\ \texttt{+}\ \texttt{id}_{\texttt{y}}\texttt{)}$$

Here $\Rightarrow$ is pronounced "derives" and $\stackrel{*}{\Rightarrow}$ is pronounced "derives in zero or more steps."

This example gives a **leftmost derivation**, i.e., at each step, the leftmost non-terminal is replaced.

DERIVATIONS (2)

We can define **rightmost derivation** analogously:

$$
\begin{aligned}
\textit{expr} &\Rightarrow \texttt{-}\ \textit{expr} \\
&\Rightarrow \texttt{-(}\textit{expr}\texttt{)} \\
&\Rightarrow \texttt{-(}\textit{expr}\ \texttt{+}\ \textit{expr}\texttt{)} \\
&\Rightarrow \texttt{-(}\textit{expr}\ \texttt{+}\ \texttt{id}_{\texttt{y}}\texttt{)} \\
&\Rightarrow \texttt{-(}\texttt{id}_{\texttt{x}}\ \texttt{+}\ \texttt{id}_{\texttt{y}}\texttt{)}
\end{aligned}
$$

Every parse tree has a **unique** leftmost derivation and a **unique** rightmost derivation.

(These are usually, but not necessarily, different.)

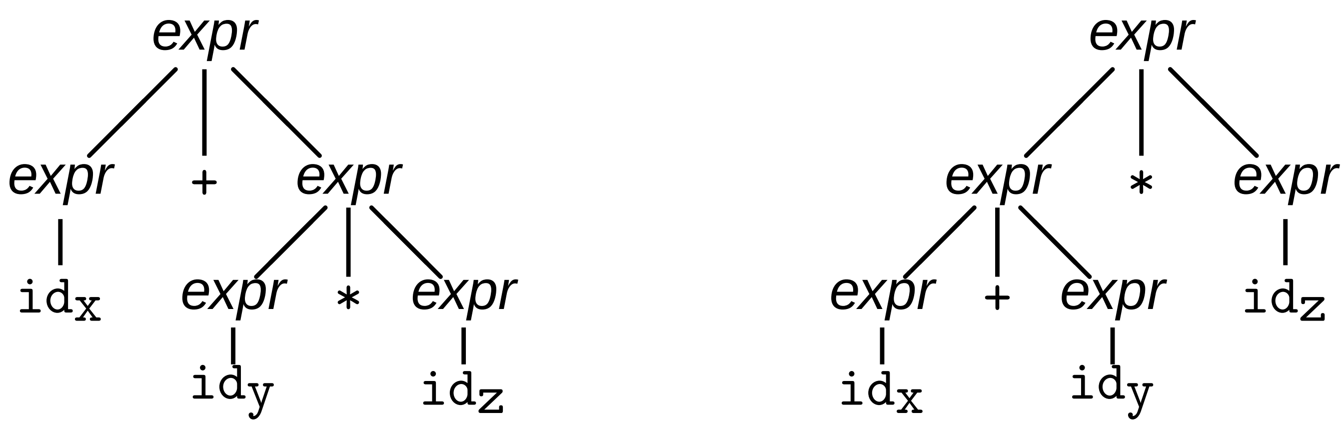
Ambiguity

BUT a given **sentence** in $L(G)$ can have more than one parse tree. Grammars G for which this is true are called **ambiguous**.

Example: with our grammar, the sentence

$\mathtt{id}_{\mathtt{x}}$ + $\mathtt{id}_{\mathtt{y}}$ * $\mathtt{id}_{\mathtt{z}}$

has two parse trees:



We may think of the left tree as being the "correct" one, but nothing in the grammar says this.

RESOLVING AMBIGUITIES

Ambiguous grammars can be a significant problem in practice, because we rely on the parse tree to capture the basic structure of a program.

To avoid the problems of ambiguity, we can:

- Rewrite grammar
- Use “disambiguating rules” when we implement parser for grammar.

A CLASSIC AMBIGUITY: THE "DANGLING ELSE"

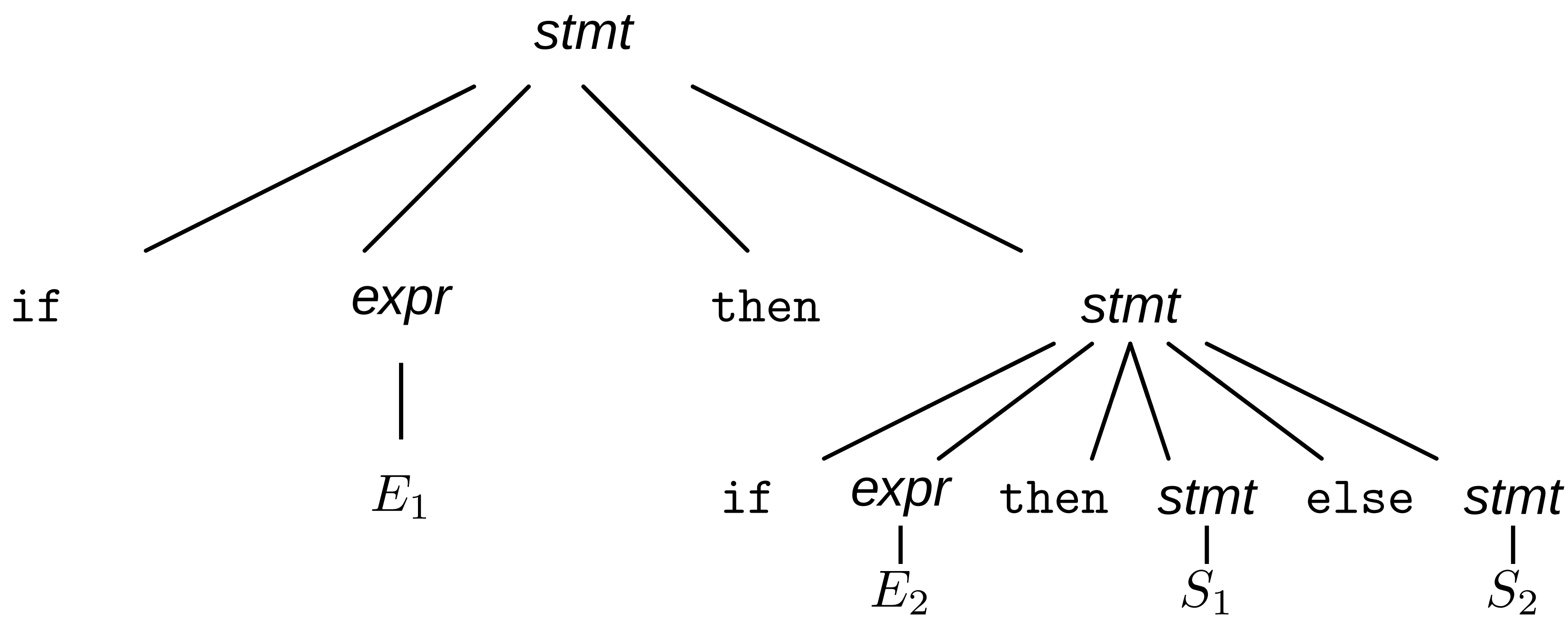
Suppose we want `else` clauses to be optional in `if` statements. Here's a possible grammar:

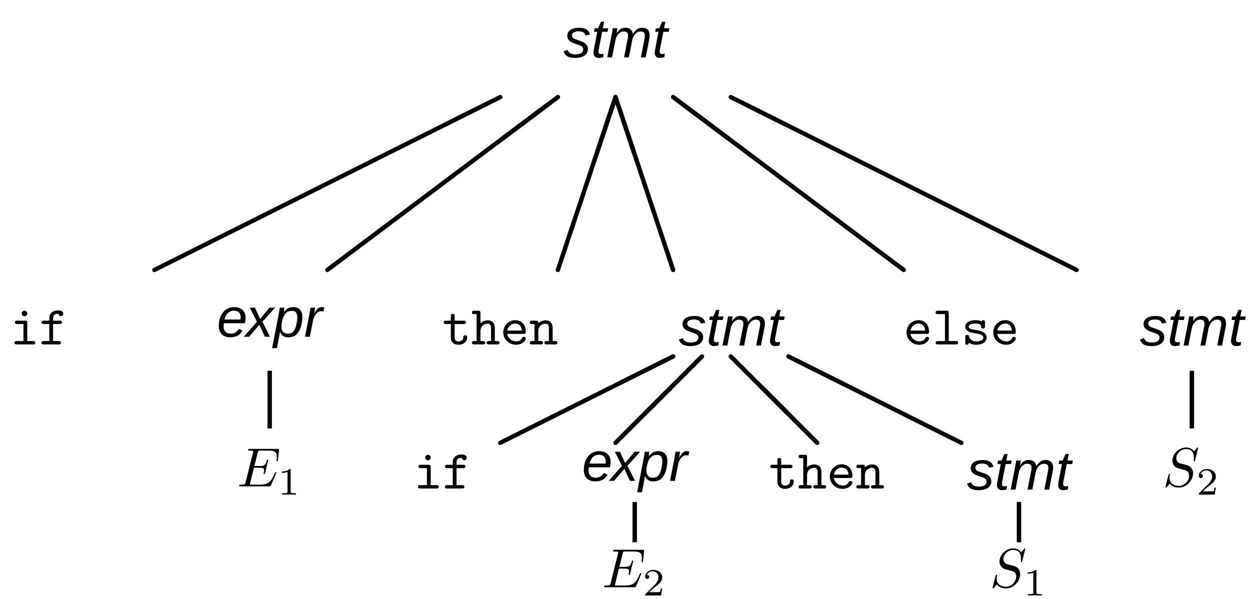
$$\textit{stmt} \rightarrow \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}$$
$$\mid\ \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}\ \texttt{else}\ \textit{stmt}\ \mid\ \ldots$$

Given this grammar, a statement of the form

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

has two possible parse trees...


stmt
if
expr
then
stmt
E_1
if
expr
then
stmt
else
stmt
E_2
S_1
S_2


stmt
if
expr
then
stmt
else
stmt
E_1
if
expr
then
stmt
S_2
E_2
S_1

RESOLVING AMBIGUITY BY REWRITING GRAMMAR

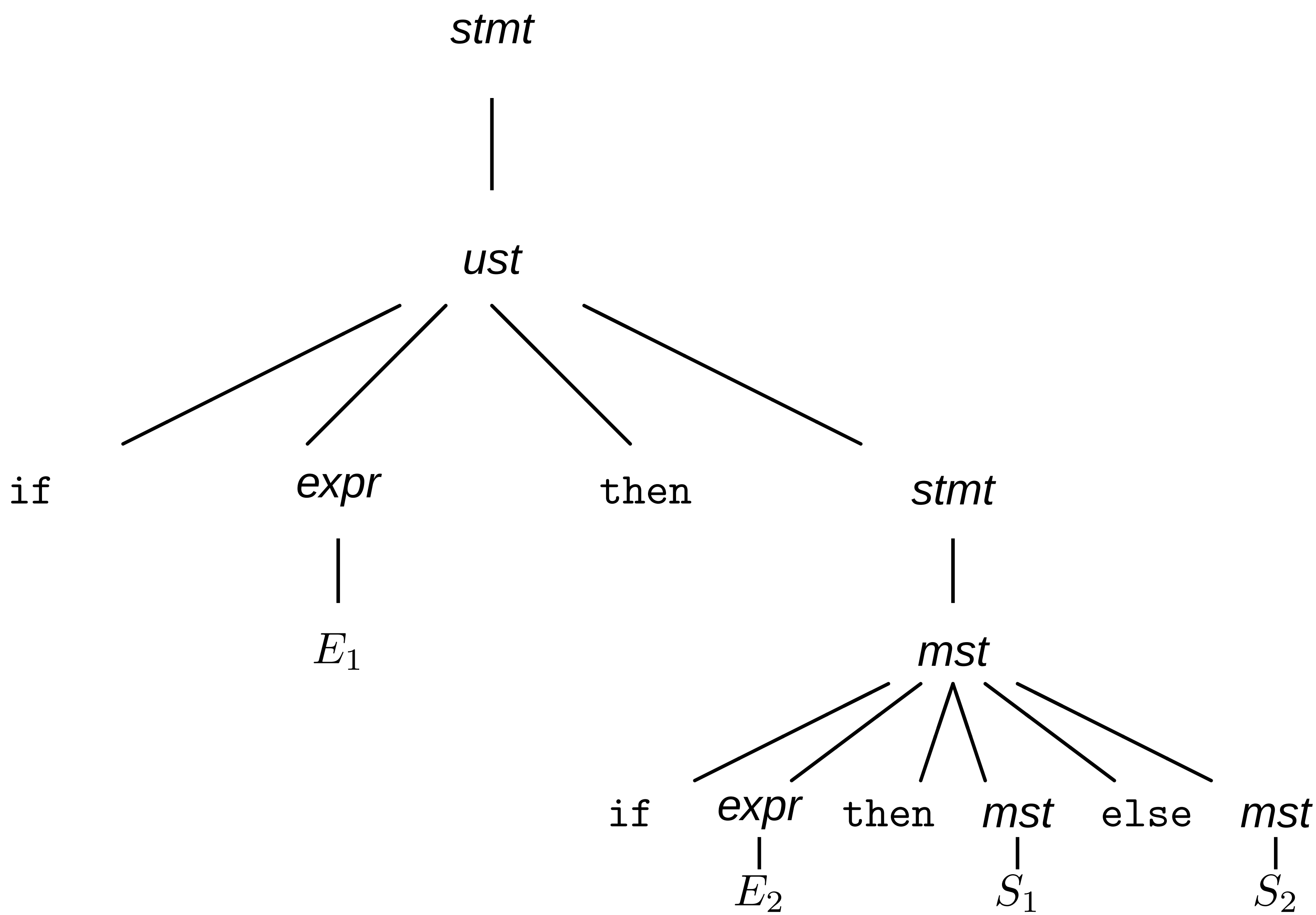
Usually want the first tree (`else` goes with most recent `then`), but grammar is ambiguous.

Solution: rewrite grammar using new non-terminals *mst* ("matched statement") and *ust* ("unmatched statement").

stmt	$\rightarrow$	*mst* | *ust*
mst	$\rightarrow$	`if` *expr* `then` *mst* `else` *mst*
	|	...
ust	$\rightarrow$	`if` *expr* `then` *stmt*
	|	`if` *expr* `then` *mst* `else` *ust*

Now only one parse is possible.

Assuming S_1,S_2 are not unmatched `if` statements, we get...


stmt
ust
if
expr
then
stmt
E_1
mst
if
expr
then
mst
else
mst
E_2
S_1
S_2

Ambiguity in Arithmetic Expressions

A grammar such as

$$
\begin{array}{lll}
E & \rightarrow & E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \\
 & & E \uparrow E \mid (E) \mid \texttt{-} E \mid \texttt{id}
\end{array}
$$

is ambiguous about order of operations.

Want to define

- **Precedence** - which operation is done first (“binds more tightly”) ?
- **Associativity** - is

$\texttt{X}\ op_1\ \texttt{Y}\ op_2\ \texttt{Z}$

equivalent to

$\texttt{(X}\ op_1\ \texttt{Y)}\ op_2\ \texttt{Z}$ (left-associativity)

or to

$\texttt{X}\ op_1\ \texttt{(Y}\ op_2\ \texttt{Z)}$ (right-associativity)

assuming op_1 and op_2 have same precedence?

Standard Precedence and Associativity

The "usual" rules (based on common usage in written math) give the following precedences, highest first:

$-$ (unary minus)

$\uparrow$ (exponentiation)

$*$ $/$

$+$ $-$

All the binary operators are left-associative except exponentiation ($\uparrow$).

Note that these rules are just a matter of **convention**; a programming language designer might choose different ones.

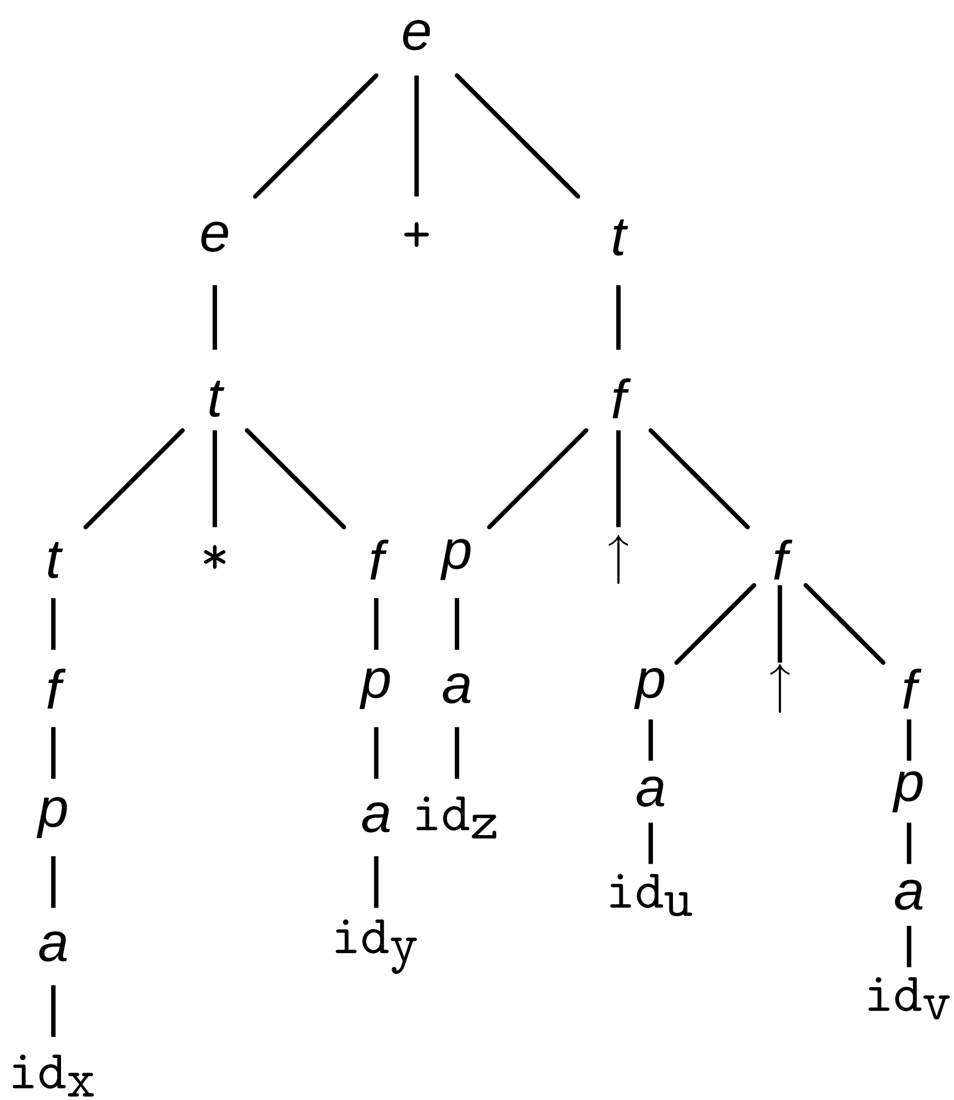
We can handle precedence/associativity information as "side-conditions" to ambiguous grammar when building a parser (by hand or via a parser generator).

REWRITING ARITHMETIC GRAMMARS

Can build precedence/associativity into grammar using extra non-terminals, each corresponding to a separate level of precedence:

$$
\begin{array}{lcl}
\textit{atom} & \longrightarrow & \texttt{(}\textit{expr}\texttt{)} \mid \texttt{id} \\
\textit{primary} & \longrightarrow & \texttt{-}\textit{primary} \mid \textit{atom} \\
\textit{factor} & \longrightarrow & \textit{primary} \uparrow \textit{factor} \mid \textit{primary} \\
\textit{term} & \longrightarrow & \textit{term} \ \texttt{*} \ \textit{factor} \mid \textit{term} \ \texttt{/} \ \textit{factor} \mid \textit{factor} \\
\textit{expr} & \longrightarrow & \textit{expr} \ \texttt{+} \ \textit{term} \mid \textit{expr} \ \texttt{-} \ \textit{term} \mid \textit{term}
\end{array}
$$

Example: $\texttt{id}_\texttt{x} * \texttt{id}_\texttt{y} + \texttt{id}_\texttt{z} \uparrow \texttt{id}_\texttt{u} \uparrow \texttt{id}_\texttt{v}$



Extended BNF

Various semi-standard extensions to BNF are often used in language manuals.

They allow grammar specifications to avoid explicit **recursion** and ϵ -productions, by adding **optional** symbols, **repetition**, and **grouping**.

$[a]$	means	$a \mid \epsilon$
$\{a\}$	means	$\epsilon \mid a \mid aa \mid \ldots$
$(a \mid b)c$	means	$ac \mid bc$

To avoid confusion between new meta-symbols and terminals, we often enclose the latter in single quotes:

Example:

atom	$\rightarrow$	`'('` *expr* `')'` $\mid$ `id`
primary	$\rightarrow$	*-primary* $\mid$ *atom*
factor	$\rightarrow$	*primary* $\{ \uparrow$ *primary* $\}$
term	$\rightarrow$	$\{$ *factor* $($ `*` $\mid$ `/` $)$ $\}$ *factor*
expr	$\rightarrow$	$\{$ *term* $($ `+` $\mid$ `-` $)$ $\}$ *term*

Issues with EBNF

Must be careful not to **change** the generated language accidentally when going between EBNF and BNF.

Also, note that EBNF grammar given in example is ambiguous because we've lost the associativity information present in the BNF version (even though both generate same language).

Often, a language's grammar will be given in ambiguous EBNF tegether with separate informal specifications that resolve the ambiguities.

PROPERTIES OF CFG'S

Any **regular** language can be described by a CFG.

Example: `(a|b)*abb`

$$\begin{array}{lcl} A_0 & \rightarrow & \mathtt{a}A_0 \mid \mathtt{b}A_0 \mid \mathtt{a}A_1 \\ A_1 & \rightarrow & \mathtt{b}A_2 \\ A_2 & \rightarrow & \mathtt{b}A_3 \\ A_3 & \rightarrow & \epsilon \end{array}$$

But we don't use CFG's for lexical analysis, because it's overkill.

Regular expressions are:

- easier to understand
- shorter
- always lead to efficient analyzers

Any CFL can be parsed by a computer program, but only **some** CFL's can be parsed **efficiently**.

We'll study both "bottom-up" and "top-down" parsing methods.

CFG'S CAN'T DO EVERYTHING

Not every language is a CFL.

Example: $L = \{wcw \mid w \in (a \mid b)^*\}$ is not CF.

- L abstracts idea of variable declaration before use.
- So "semantic" analysis (type-checking) uses additional (mostly ad-hoc) techniques.