LECTURE 11

INTRODUCTION TO SYNTAX ANALYSIS

SUBJECTS

Context free grammars

Derivations

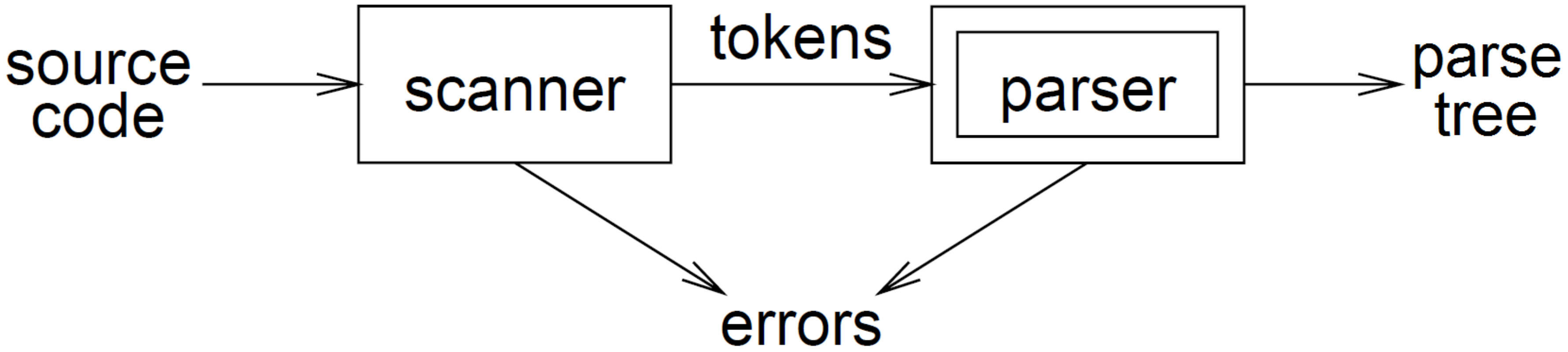
Parse Trees

Ambiguity

Top-down parsing

Left recursion

THE ROLE OF PARSER



CONTEXT FREE GRAMMARS

A Context Free Grammar (CFG) consists of

- Terminals
- Nonterminals
- Start symbol
- Productions

A language that can be generated by a grammar is said to be a context-free language

CONTEXT FREE GRAMMARS

Terminals: are the basic symbols from which strings are formed

- These are the tokens that were produced by the Lexical Analyser

Nonterminals: are syntactic variables that denote sets of strings

- One nonterminal is distinguished as the **start symbol**

The productions of a grammar specify the manner in which the terminal and nonterminals can be combined to form strings

EXAMPLE OF GRAMMAR

The grammar with the following productions defines simple arithmetic expressions

```
⟨expr⟩  ::=  ⟨expr⟩  ⟨op⟩  ⟨expr⟩
⟨expr⟩  ::= id
⟨expr⟩  ::= num
  ⟨op⟩  ::= +
  ⟨op⟩  ::= -
  ⟨op⟩  ::= *
  ⟨op⟩  ::= /
```

In this grammar, the terminal symbols are
***num, id* + - * /**

The nonterminal symbols are *⟨expr⟩* and *⟨op⟩*, and *⟨expr⟩* is the start symbol

DERIVATIONS

$\langle \texttt{expr} \rangle \Rightarrow \langle \texttt{expr} \rangle \langle \texttt{op} \rangle \langle \texttt{expr} \rangle$

is read *"expr derives expr op expr"*

$$
\begin{aligned}
\langle \texttt{expr} \rangle & \Rightarrow \langle \texttt{expr} \rangle \langle \texttt{op} \rangle \langle \texttt{expr} \rangle \\
& \Rightarrow \texttt{id} \ \langle \texttt{op} \rangle \langle \texttt{expr} \rangle \\
& \Rightarrow \texttt{id} * \langle \texttt{expr} \rangle \\
& \Rightarrow \texttt{id*id}
\end{aligned}
$$

is called a derivation of id*id from *expr*.

DERIVATIONS

If $A ::= \gamma$ is a production and α and β are arbitrary strings of grammar symbols, we can say:

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow . . . \Rightarrow \alpha_n$, we say α_1 *derives* α_n.

DERIVATIONS

$\Rightarrow$ **means "derives in one step."**

$\overset{*}{\Rightarrow}$ **means "derives in zero or more steps."**

- *if* $\alpha \overset{*}{\Rightarrow} \beta$ *and* $\beta \Rightarrow \gamma$ *then* $\alpha \overset{*}{\Rightarrow} \gamma$

$\overset{+}{\Rightarrow}$ **means "derives in one or more steps."**

If $S \overset{*}{\Rightarrow} \alpha$, **where** α **may contain nonterminals, then we say that** α **is a sentential form**

- If α does no contains any nonterminals, we say that α is a sentence

DERIVATIONS

G: grammar

S: start symbol

L(G): the language generated by G

Strings in L(G) may contain only terminal symbols of G

A string of terminals *w* is said to be in L(G) if and only if $S \overset{+}{\Rightarrow} w$

- As we said, the string *w* is called a sentence of G

A language that can be generated by a grammar is said to be a context-free language

- If two grammars generate the same language, the grammars are said to be equivalent

DERIVATIONS

We have already seen the following production rules:

$\langle \mathtt{expr} \rangle$ ::= $\langle \mathtt{expr} \rangle$ $\langle \mathtt{op} \rangle$ $\langle \mathtt{expr} \rangle$ | `id` | `num`

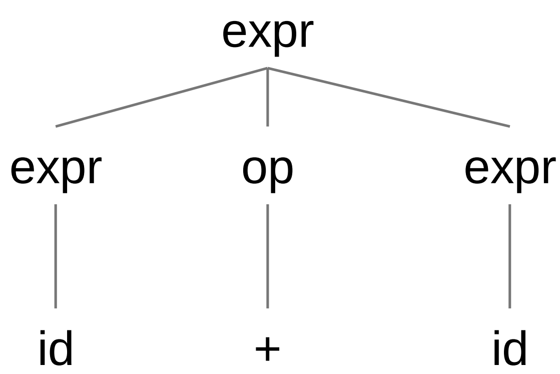
$\langle \mathtt{op} \rangle$ ::= + | - | * | /

The string `id+id` is a sentence of the above grammar because

$$
\begin{aligned}
\langle \mathtt{expr} \rangle \quad & \Rightarrow \langle \mathtt{expr} \rangle \ \langle \mathtt{op} \rangle \ \langle \mathtt{expr} \rangle \\
& \Rightarrow \mathtt{id} \ \langle \mathtt{op} \rangle \ \langle \mathtt{expr} \rangle \\
& \Rightarrow \mathtt{id} + \langle \mathtt{expr} \rangle \\
& \Rightarrow \mathtt{id} + \mathtt{id}
\end{aligned}
$$

We write $\langle \mathtt{expr} \rangle \stackrel{*}{\Rightarrow} \mathtt{id{+}id}$

PARSE TREE



This is called:
Leftmost derivation

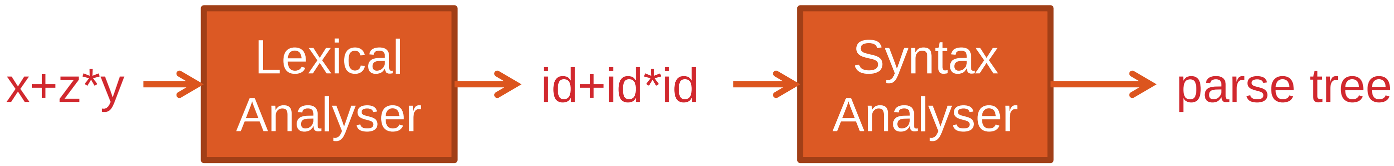
TWO PARSE TREES

Let us again consider the arithmetic expression grammar.

For the line of code:

x+2*y

(we are not considering the semi colon for now)



Grammar:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

TWO PARSE TREES

Let us again consider the arithmetic expression grammar.

The sentence id + id * id has two distinct leftmost derivations:

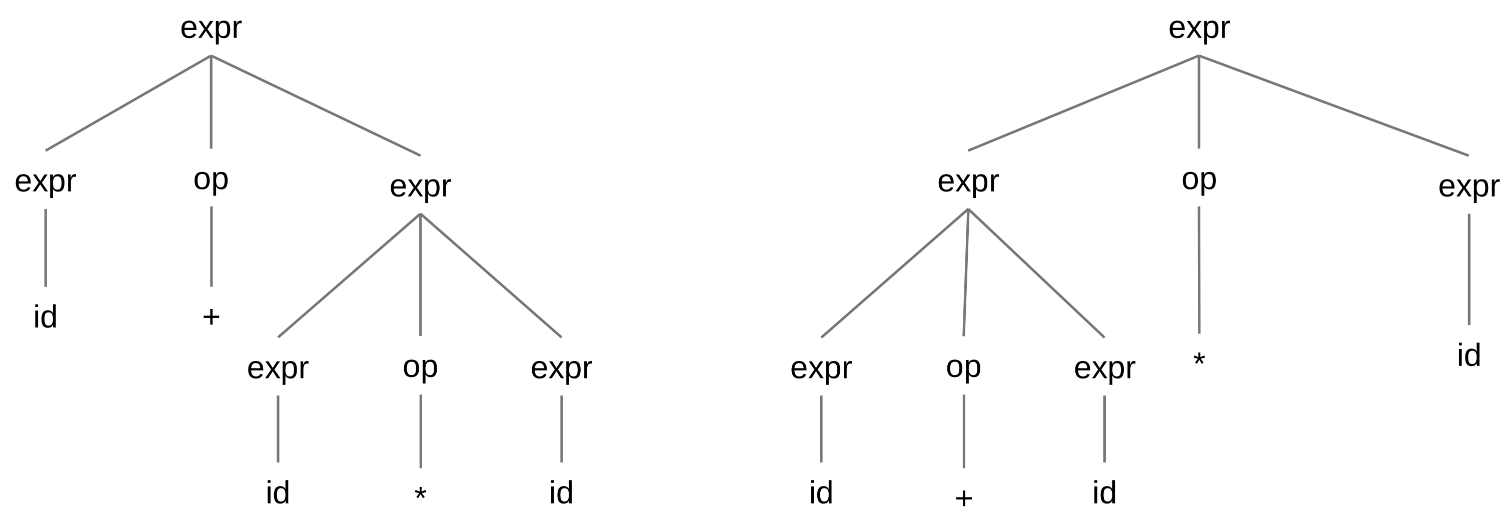
```
⟨expr⟩  ⇒  ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ id  ⟨op⟩ ⟨expr⟩
        ⇒ id + ⟨expr⟩
        ⇒ id + ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ id + id ⟨op⟩ ⟨expr⟩
        ⇒ id + id * ⟨expr⟩
        ⇒ id + id * id
```

```
⟨expr⟩  ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ ⟨expr⟩ ⟨op⟩ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ id ⟨op⟩ ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ id + ⟨expr⟩ ⟨op⟩ ⟨expr⟩
        ⇒ id + id ⟨op⟩ ⟨expr⟩
        ⇒ id + id * ⟨expr⟩
        ⇒ id + id * id
```

Grammar:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

TWO PARSE TREES



Equivalent to:
id+(id*id)

Equivalent to:
(id+id)*id

Grammar:

```
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | id | num
⟨op⟩ ::= + | - | * | /
```

PRECEDENCE

The previous example highlights a problem in the grammar:

- It does not enforce precedence
- It has not implied an order of evaluation

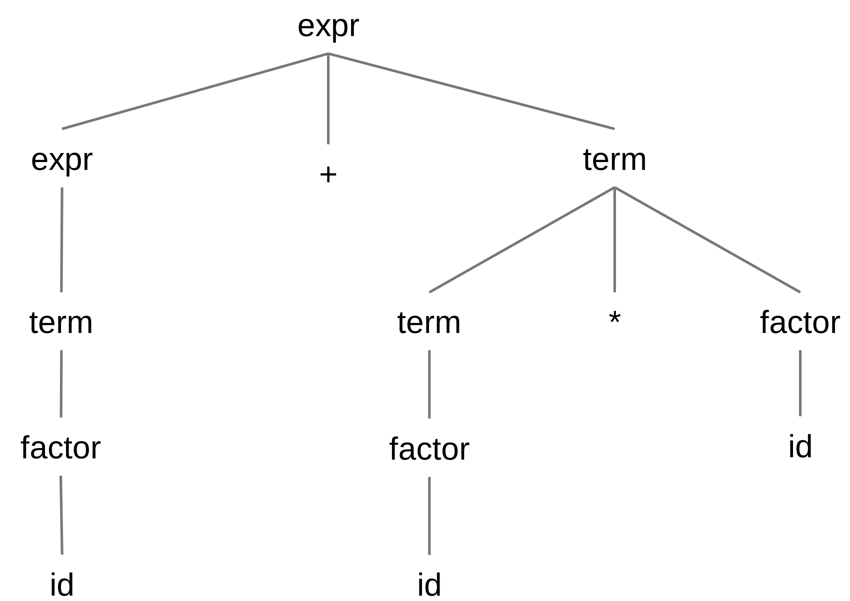
We can expand the production rules to add precedence

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

APPLYING PRECEDENCE UPDATE

The sentence id + id * id has only one leftmost derivation now:

```
⟨expr⟩  ⇒  ⟨expr⟩  + ⟨term⟩
        ⇒  ⟨term⟩  + ⟨term⟩
        ⇒  ⟨factor⟩ + ⟨term⟩
        ⇒ id + ⟨term⟩
        ⇒ id + ⟨term⟩ * ⟨factor⟩
        ⇒ id + ⟨factor⟩ * ⟨factor⟩
        ⇒ id + id * ⟨factor⟩
        ⇒ id + id * id
```



Grammar:

```
⟨expr⟩     ::= ⟨expr⟩ +  ⟨term⟩ | ⟨expr⟩ -  ⟨term⟩ |  ⟨term⟩
⟨term⟩     ::= ⟨term⟩ * ⟨factor⟩ |  ⟨term⟩ / ⟨factor⟩ | ⟨factor⟩
⟨factor⟩   ::= num | id
```

AMBIGUITY

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.

Example:

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

Consider the following statement:

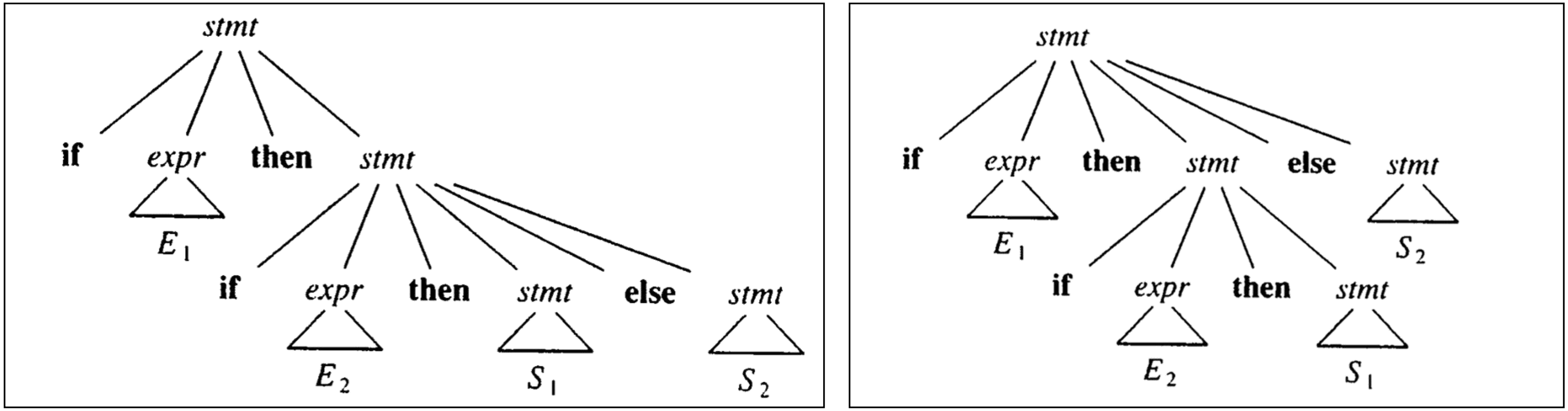
$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

It has two derivations

- It is a context free ambiguity

AMBIGUITY

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*



ELIMINATING AMBIGUITY

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

- E.g. “match each else with the closest unmatched then”
- This is most likely the intention of the programmer

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

MAPPING THIS TO A JAVA EXAMPLE

uOttawa
L'Université canadienne
Canada's university

In Java, the grammar rules are slightly different then the previous example

Below is a (very simplified) version of these rules

```
<stmnt>        ::= <matched>
               | <unmatched>
<matched>      ::= if ( <expr> ) <matched> else <matched>
               | other stmnts
<unmatched>    ::= if ( <expr> ) < stmnt >
               | if ( <expr> ) <matched> else <unmatched>
```

MAPPING THIS TO A JAVA EXAMPLE

For the following piece of code

```
if (x==0)
   if (y==0)
      z = 0;
   else
      z = 1;
```

After running the lexical analyser, we get the following list of tokens:

```
if ( id == num ) if (id == num) id = num ; else id = num ;
```

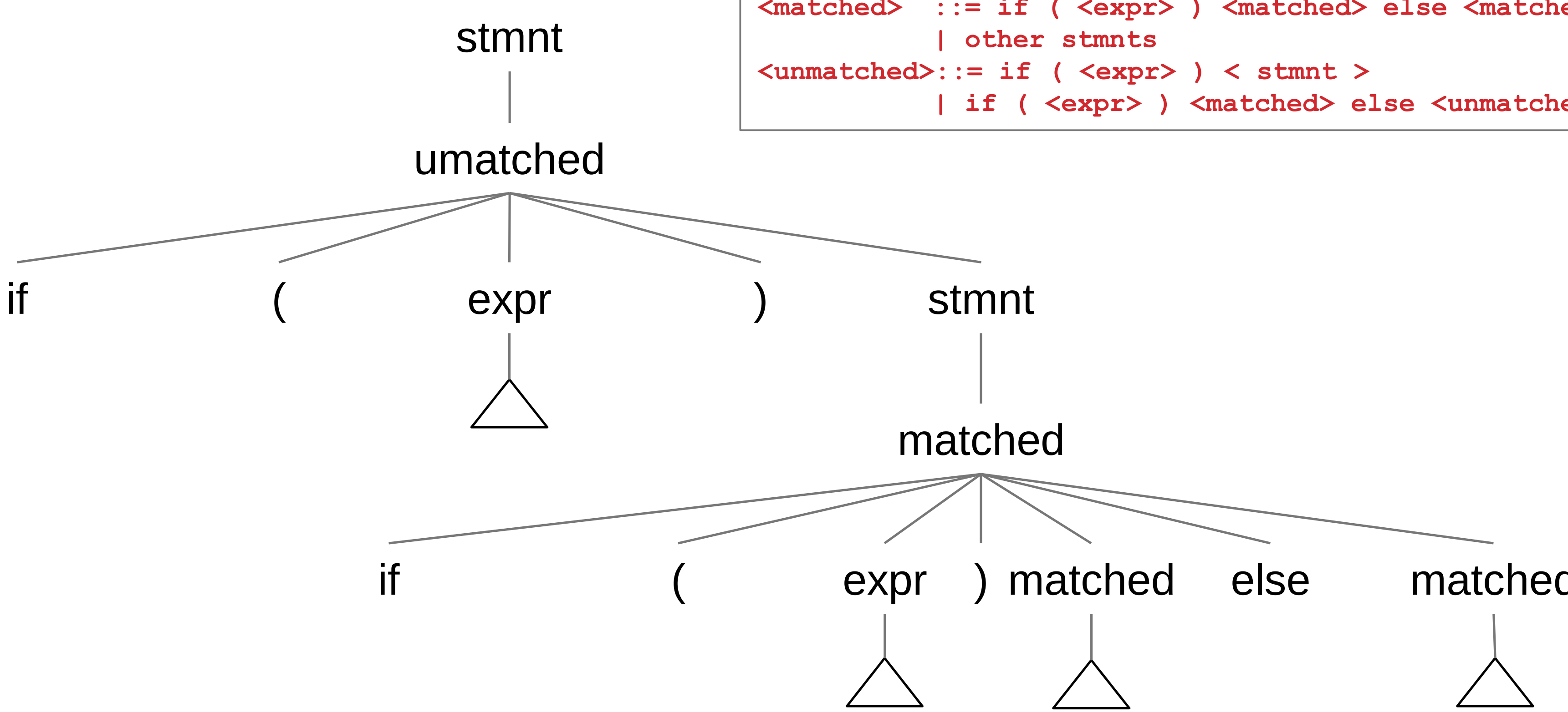
MAPPING THIS TO A JAVA EXAMPLE

Token input string

```
if ( id == num ) if (id == num) id = num ; else id = num ;
```

Grammar

```
<stmnt>      ::= <matched>
               | <unmatched>
<matched>    ::= if ( <expr> ) <matched> else <matched>
               | other stmnts
<unmatched>::= if ( <expr> ) < stmnt >
               | if ( <expr> ) <matched> else <unmatched>
```



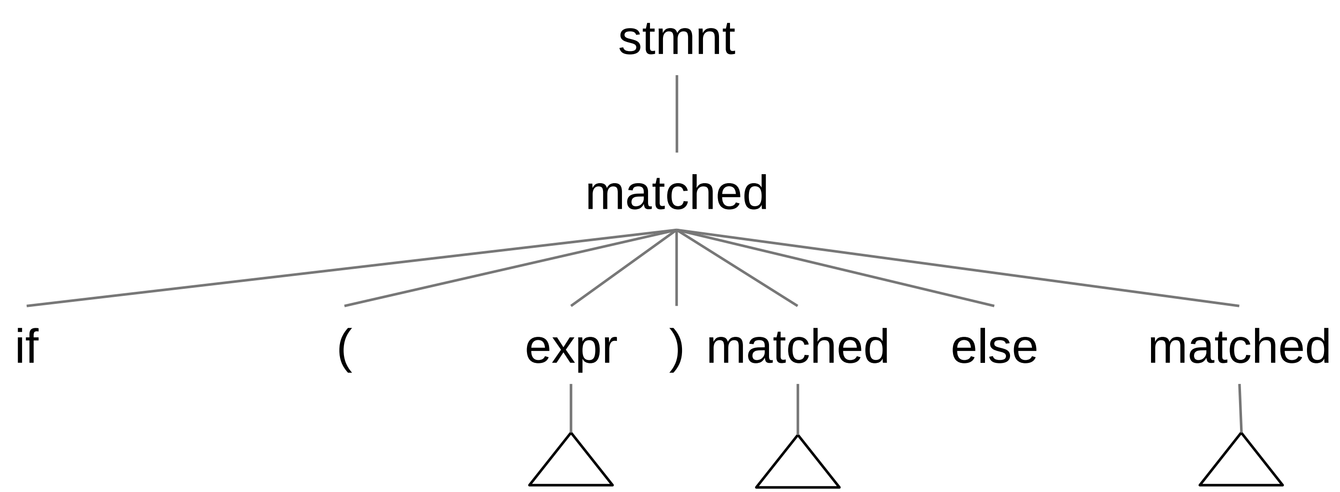
MAPPING THIS TO A JAVA (ANOTHER) EXAMPLE

Token input string

```
if ( id == num ) else id = num ;
```

Grammar

```
<stmnt>     ::= <matched>
              | <unmatched>
<matched>   ::= if ( <expr> ) <matched> else <matched>
              | other stmnts
<unmatched>::= if ( <expr> ) < stmnt >
              | if ( <expr> ) <matched> else <unmatched>
```



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 tree, we repeat the following steps until the leaves of the parse tree match the input string

1. At a node labelled *A*, select a production A::=α and construct the appropriate child for each symbol of α
2. When a terminal is added to the parse tree that does not match the input string, backtrack
3. Find the next nonterminal to be expanded

TOP DOWN PARSING

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string

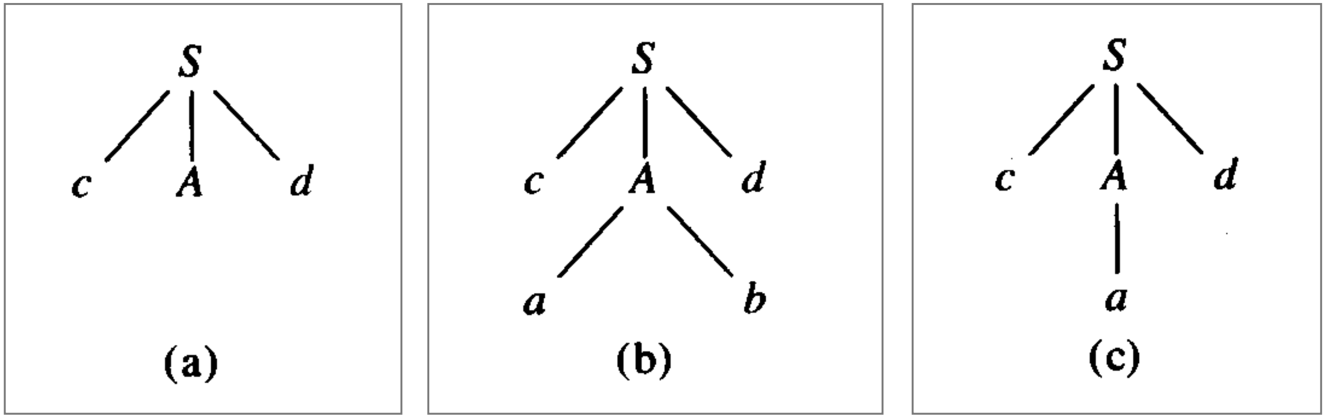
Example:

Grammar:

```
<S> ::= c<A>d
<A> ::= ab | a
```

Input string

`cad`



We need to backtrack!

EXPRESSION GRAMMAR

Recall our grammar for simple expressions:

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

Consider the input string:

`id - num * id`

EXAMPLE

Prod'n	Sentential form	Input
1	$\langle\text{expr}\rangle$	$\uparrow$ `id` `-` `num` `*` `id`

Reference Grammar

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
 & \mid & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
 & \mid & \langle\text{term}\rangle \\
\langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
 & \mid & \langle\text{term}\rangle \,/\, \langle\text{factor}\rangle \\
 & \mid & \langle\text{factor}\rangle \\
\langle\text{factor}\rangle & ::= & \texttt{num} \\
 & \mid & \texttt{id}
\end{array}
$$

EXAMPLE

Reference Grammar

⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
	|	⟨expr⟩ − ⟨term⟩
	|	⟨term⟩
⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
	|	⟨term⟩ / ⟨factor⟩
	|	⟨factor⟩
⟨factor⟩	::=	num
	|	id

Prod'n	Sentential form	Input
1	⟨expr⟩	↑id − num * id
2	⟨expr⟩ + ⟨term⟩	↑id − num * id
4	⟨term⟩ + ⟨term⟩	↑id − num * id
7	⟨factor⟩ + ⟨term⟩	↑id − num * id
9	id + ⟨term⟩	↑id − num * id
–	id + ⟨term⟩	id ↑− num * id
–	⟨expr⟩	↑id − num * id
3	⟨expr⟩ − ⟨term⟩	↑id − num * id
4	⟨term⟩ − ⟨term⟩	↑id − num * id
7	⟨factor⟩ − ⟨term⟩	↑id − num * id
9	id − ⟨term⟩	↑id − num * id
–	id − ⟨term⟩	id ↑− num * id
–	id − ⟨term⟩	id − ↑num * id
7	id − ⟨factor⟩	id − ↑num * id
8	id − num	id − ↑num * id
–	id − num	id − num ↑* id
–	id − ⟨term⟩	id − ↑num * id
5	id − ⟨term⟩ ∗ ⟨factor⟩	id − ↑num * id
7	id − ⟨factor⟩ ∗ ⟨factor⟩	id − ↑num * id
8	id − num ∗ ⟨factor⟩	id − ↑num * id
–	id − num ∗ ⟨factor⟩	id − num ↑* id
–	id − num ∗ ⟨factor⟩	id − num * ↑id
9	id − num ∗ id	id − num * ↑id
–	id − num ∗ id	id − num * id↑

EXAMPLE

Another possible parse for id – num * id

Prod'n	Sentential form	Input
1	$\langle\text{expr}\rangle$	$\uparrow$ `id - num * id`
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow$ `id - num * id`
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow$ `id - num * id`
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow$ `id - num * id`
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow$ `id - num * id`
2	$\cdots$	$\uparrow$ `id - num * id`

Reference Grammar

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

If the parser makes the wrong choices, expansion does not terminate

- This is not a good property for a parser to have
- Parsers should terminate, eventually…

LEFT RECURSION

A grammar is left recursive if:

“It has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α”

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

ELIMINATING LEFT RECURSION

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 $\langle$`foo`$\rangle$

We can re-write 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 $\langle$bar$\rangle$ is a new non-terminal

This Fragment contains no left recursion

ELIMINATING LEFT RECURSION

Similarly,

$$
\begin{aligned}
\langle \text{foo} \rangle \;::=\; & \langle \text{foo} \rangle\ \alpha_1 \\
| \; & \langle \text{foo} \rangle\ \alpha_2 \\
\ldots \\
| \; & \langle \text{foo} \rangle\ \alpha_n \\
| \; & \beta_1 \\
| \; & \beta_2 \\
\ldots \\
| \; & \beta_n
\end{aligned}
$$

Can be re-written as:

$$
\begin{aligned}
\langle \text{foo} \rangle \;::=\; & \beta_1\ \langle \text{bar} \rangle \\
| \; & \beta_2\ \langle \text{bar} \rangle \\
\ldots \\
| \; & \beta_n\ \langle \text{bar} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{bar} \rangle \;::=\; & \alpha_1\ \langle \text{bar} \rangle \\
| \; & \alpha_2\ \langle \text{bar} \rangle \\
\ldots \\
| \; & \alpha_n\ \langle \text{bar} \rangle \\
| \; & \varepsilon
\end{aligned}
$$

EXAMPLE

Our expression grammar contains two cases of left-recursion

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

Applying the transformation gives

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

With this grammar, a top-down parser will

- Terminate (for sure)
- Backtrack on some inputs

LOOK AHEAD...

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

Therefore, we might need to look ahead in order to avoid backtracking

This is where predictive parsers come in handy

- LL(1): left to right scan, left-most derivation, 1-token look ahead
- LR(1): left to right scan, right most derivation, 1-token look ahead

THANK YOU!

QUESTIONS?