Context-free grammars

Context-free grammars can generate context-free languages.

They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules.

Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context — it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

- **Input:** sequence of tokens from lexer

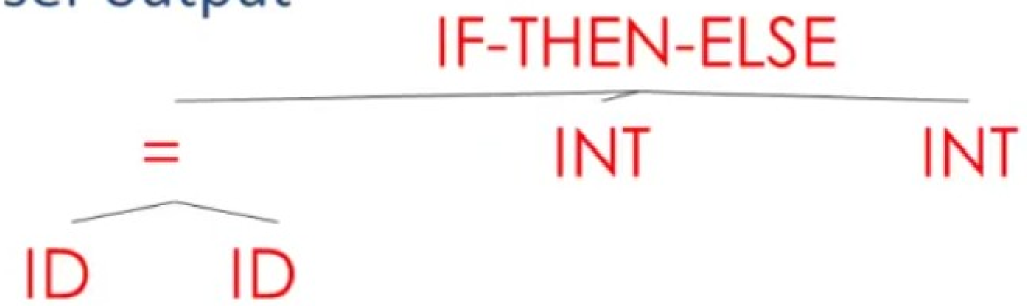
- **Output:** parse tree of the program

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Expressions

Conceptually, there are two types of expressions: those that assign a value to a variable and those that simply have a value.

The expression x = 7 is an example of the first type. This expression uses the = operator to assign the value seven to the variable x.
The expression itself evaluates to seven.

The code 3 + 4 is an example of the second expression type.
This expression uses the + operator to add three and four together without assigning the result, seven, to a variable.

Assignment in predicate can be useful for loops more than if statements.

```
while( var = GetNext() )
{
  ...do something with var
}
```

Which would otherwise have to be written

```
var = GetNext();
while( var )
{
 ...do something
 var = GetNext();
}
```

- Programming languages have recursive structure

- An EXPR is

  if EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  …

- Context-free grammars are a natural notation for this recursive structure

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S $(S \in N)$
  - A set of *productions*

$$X \rightarrow Y_1 \ldots Y_N$$

$$X \in N$$
$$Y_i \in N \cup T \cup \{\epsilon\}$$

$$\left\{ \begin{array}{l} S \to (S) \\ S \to \epsilon \end{array} \right\} \qquad \begin{array}{l} N = \{S\} \\ T = \{(,)\} \end{array}$$

ANY

1. Begin with a string with only the start symbol S

2. Replace any non-terminal X in the string by the right-hand side of some production $X \to Y_1 \ldots Y_n$

3. Repeat (2) until there are no non-terminals

$$X_1 \ldots X_i \underline{X} X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_i Y_1 \ldots Y_k X_{i+1} \ldots X_n$$

step in a derivation

$$\underbrace{X \rightarrow Y_1 \ldots Y_k}$$ production

$$S \rightarrow \_\_ \rightarrow \_\_\_ \;\; \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_n$$

$$\alpha_0 \xrightarrow{*} \alpha_n \quad (\text{in} \geq 0 \text{ steps})$$

Let G be a context-free grammar with start symbol S.
Then the language $L(G)$ of G is:

$$\{ a_1 \ldots a_n \mid \forall i \;\; a_i \in T \quad S \xrightarrow{*} a_1 \ldots a_n \}$$

- Terminals are so-called because there are no rules for replacing them

- Once generated, terminals are permanent

- Terminals ought to be tokens of the language

EXPR → if EXPR then EXPR else EXPR fi

EXPR → while EXPR loop EXPR pool

EXPR → id

⋮

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

⋮

Some elements of the language:

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Simple arithmetic expressions

$$
\begin{aligned}
E \;\rightarrow\; & E + E \\
\mid\; & E * E \\
\mid\; & (E) \\
\mid\; & \mathrm{id}
\end{aligned}
$$

id

$\mathrm{id} + \mathrm{id}$

$\mathrm{id} + \mathrm{id} * \mathrm{id}$

$(\mathrm{id} + \mathrm{id}) * \mathrm{id}$

The idea of a CFG is a big step. But:

- Membership in a language is “yes” or “no”; also need parse tree of the input

- Must handle errors gracefully

- Need an implementation of CFG's (e.g., bison)

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

A *derivation* is a sequence of productions

$$S \to \dots \to \dots \to \dots \to \dots \to \dots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \to Y_1 \dots Y_n$ add children $Y_1 \dots Y_n$ to node X

- Grammar

$$E \to E + E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

- Grammar

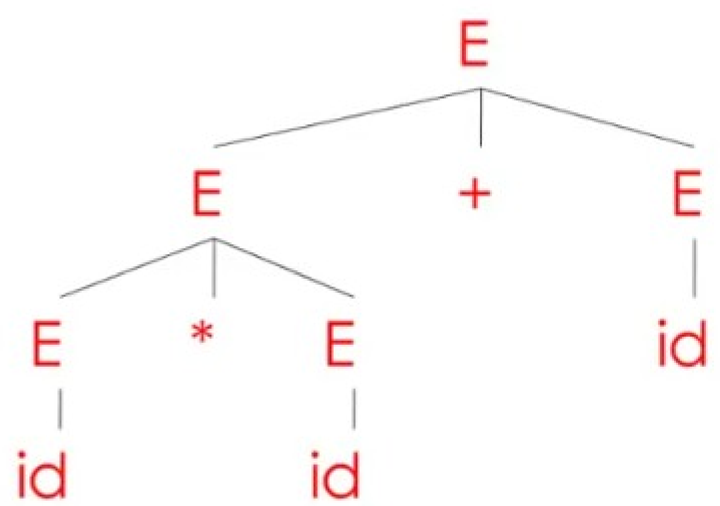
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- String

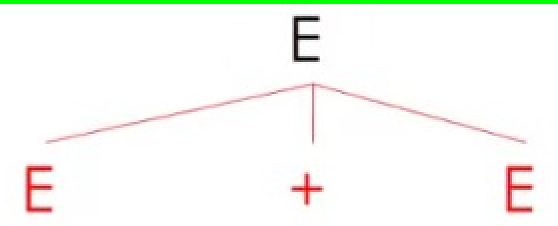
$$id * id + id$$

here we replace the leftmost non-terminal first, left most derivation

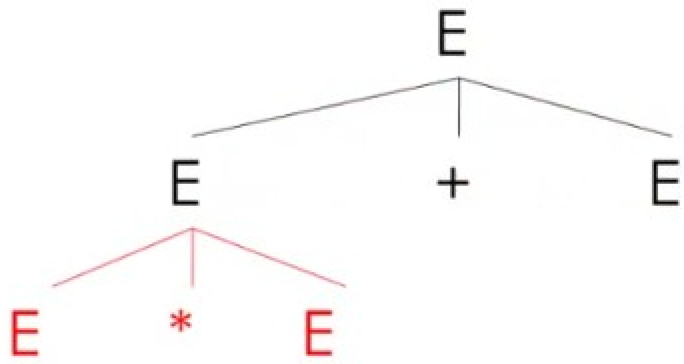
$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



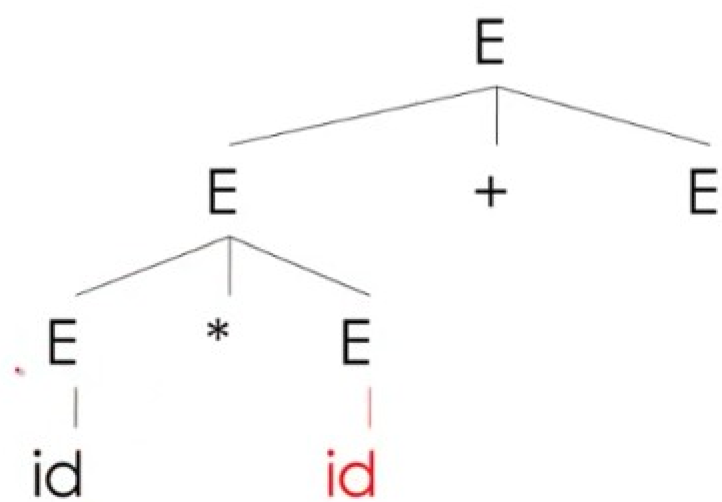
$$
\begin{aligned}
& E \\
\rightarrow\ & E + E
\end{aligned}
$$



$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E
\end{aligned}
$$



$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E * E + E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- An in-order traversal of the leaves is the original input

- The parse tree shows the association of operations, the input string does not

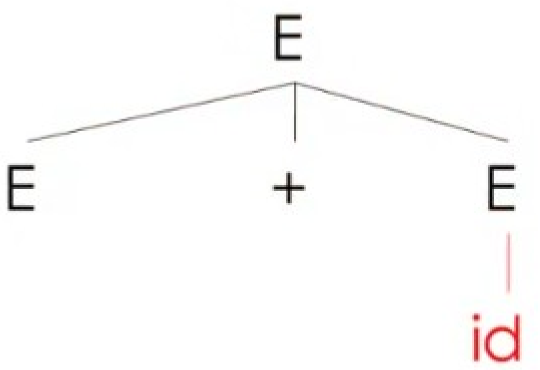
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

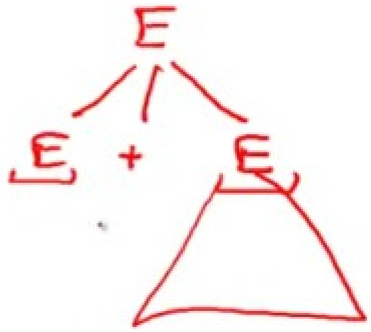
$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

replace rightmost non-terminal first,
right most derivation

Note that right-most and left-most derivations have the same parse tree

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + id
\end{aligned}
$$





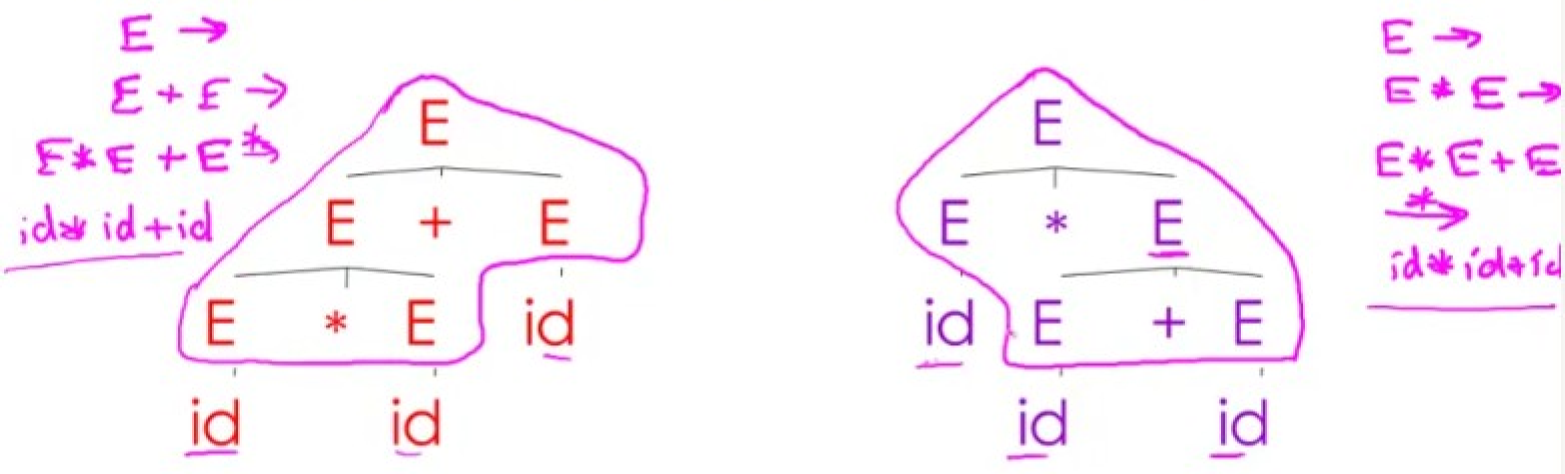
- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s

- A derivation defines a parse tree
  - But one parse tree may have many derivations

- Left-most and right-most derivations are important in parser implementation

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

This string has two parse trees id * id + id



- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

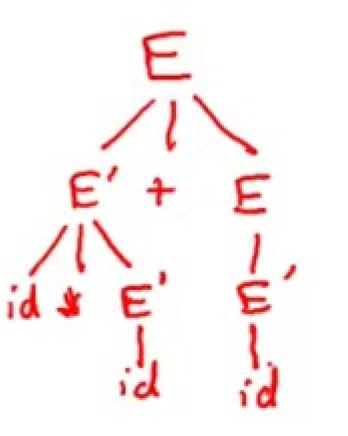
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

- There are several ways to handle ambiguity

- Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E' + E \mid E'$$
$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

- Enforces precedence of $*$ over $+$



$id * \underline{id} + \underline{id}$

- Enforces precedence of $*$ over $+$

$$E \rightarrow E' + \underline{E} \rightarrow E' + E' + \underline{E} \rightarrow E' + E' + E' + \underline{E} \rightarrow \ldots \rightarrow E' + \ldots + E'$$
$$E' \rightarrow id * \underline{E'} \rightarrow id * id * \underline{E'} \rightarrow id * id * id * \underline{E'} \rightarrow \ldots \rightarrow id * \ldots * id$$

The Dangling Else

- Consider the grammar

  $E \rightarrow$ if E then E

  | if E then E else E

  | OTHER

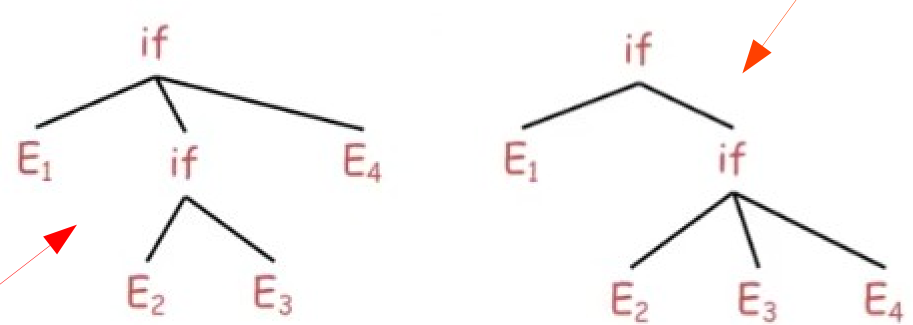
if E_1 then (if E_2 then E_3 else E_4)

we want elses to associate to the closest unmatched then

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



if E_1 then (if E_2 then E_3) else E_4

The Dangling Else

- else matches the closest unmatched then

$E \rightarrow$ MIF /* all then are matched */
$\quad$ | UIF /* some then is unmatched */

MIF $\rightarrow$ if E then MIF else MIF
$\quad$ | OTHER

UIF $\rightarrow$ if E then E
$\quad$ | if E then MIF else UIF

the only possibility for a
UIF which is itself an
if-then-else
is the unmatched
if-then-else
is in the else branch

the property we are looking
for is each else matches
the closest then

- The expression if E_1 then if E_2 then E_3 else E_4

- Impossible to convert automatically an ambiguous grammar to an unambiguous one

- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations

- Most tools allow precedence and associativity declarations to disambiguate grammars

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int

- Left associativity declaration: %left +

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int

- Precedence declarations: %left +
  %left *