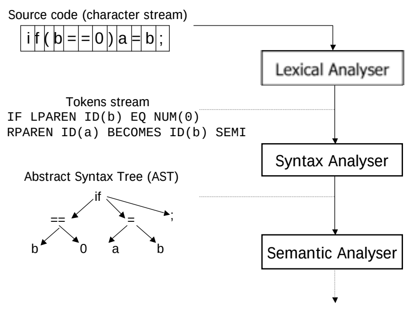
2010: Compilers

Syntax Analysis – Context Free Grammars

Dr. Licia Capra
UCL/CS

WHERE WE ARE

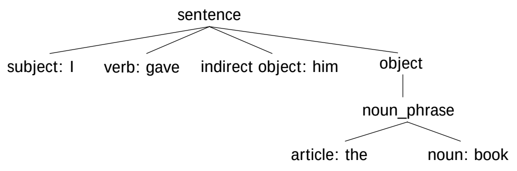


```
IF LPAREN ID(b) EQ NUM(0)
RPAREN ID(a) BECOMES ID(b) SEMI
```

SECOND STEP: SYNTAX ANALYSIS

- Goal: determine if the input token stream satisfies the syntax of the program

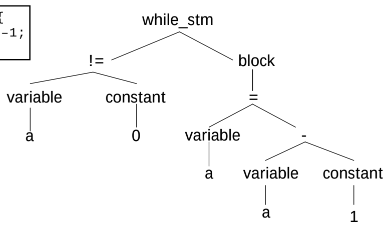
“I gave him the book”



SECOND STEP: SYNTAX ANALYSIS

- Goal: determine if the input token stream satisfies the syntax of the program

```
while (a!=0) {
      a = a -1;
}
```



SECOND STEP: SYNTAX ANALYSIS

- What we need for syntax analysis:
  - An expressive way to describe the syntax

    ... why not regular expressions?

  - An acceptor mechanism that determines if the input token stream satisfies the syntax description

    ... why not DFA?

SECOND STEP: SYNTAX ANALYSIS

- Example: nested constructs

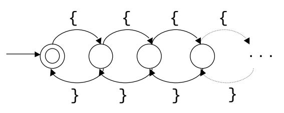
  `{{}} {} {{}{{{}}}} …`

SECOND STEP: SYNTAX ANALYSIS

- Example: nested constructs

  `{{}} {} {{}{{{}}}} …`

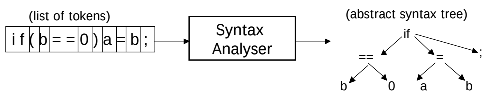
- RE are not powerful enough to express the syntax of a programming language



We need unbounded counting!

PHASE 2 - OUTLINE

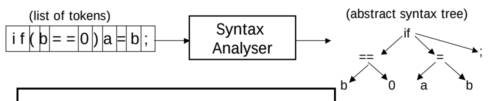
- Syntax Analyser



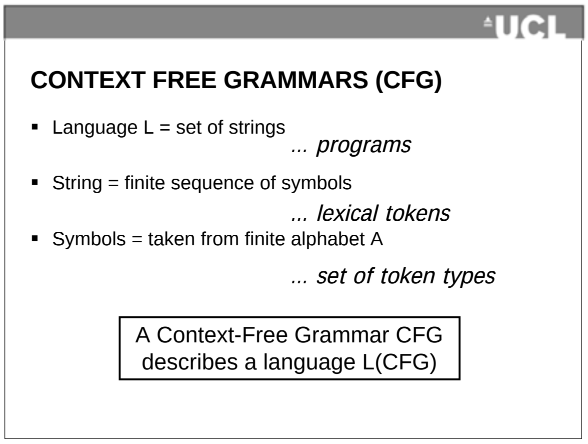
  - Context-Free Grammars (CGF)
  - Acceptors: LL(k), LR(K), SLR, LALR
  - Parser Generator (CUP)

PHASE 2 - OUTLINE

- Syntax Analyser



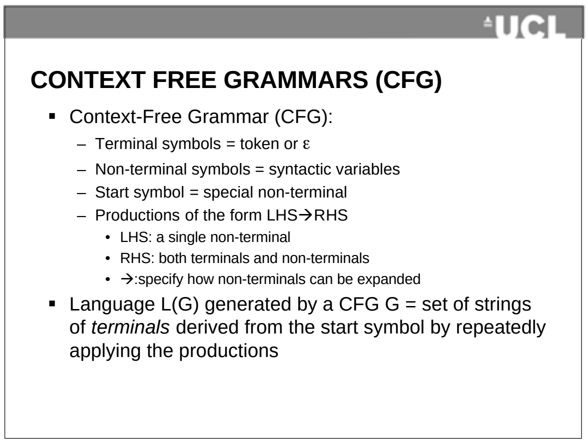
  - $\boxed{\text{Context-Free Grammars (CGF)}}$
  - Acceptors: LL(k), LR(k), SLR, LALR
  - Parser Generator (CUP)



CONTEXT FREE GRAMMARS (CFG)

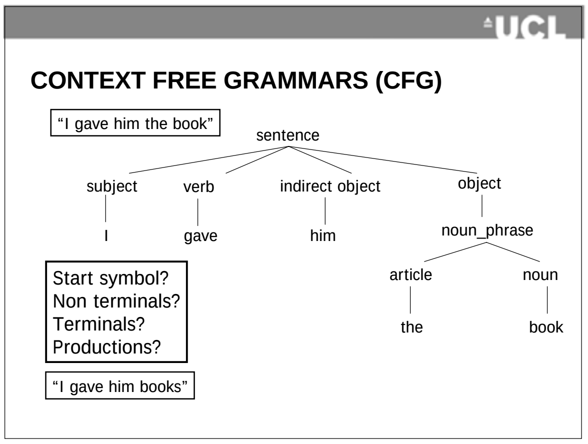
- Language L = set of strings
  ... programs
- String = finite sequence of symbols
  ... lexical tokens
- Symbols = taken from finite alphabet A
  ... set of token types

A Context-Free Grammar CFG describes a language L(CFG)



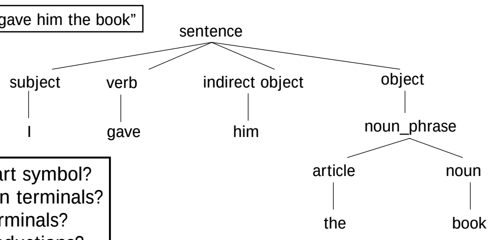
CONTEXT FREE GRAMMARS (CFG)

- Context-Free Grammar (CFG):
  - Terminal symbols = token or ε
  - Non-terminal symbols = syntactic variables
  - Start symbol = special non-terminal
  - Productions of the form LHS→RHS
    - LHS: a single non-terminal
    - RHS: both terminals and non-terminals
    - →:specify how non-terminals can be expanded
- Language L(G) generated by a CFG G = set of strings of *terminals* derived from the start symbol by repeatedly applying the productions



CONTEXT FREE GRAMMARS (CFG)

"I gave him the book"



Start symbol?
Non terminals?
Terminals?
Productions?

"I gave him books"

CONTEXT FREE GRAMMARS (CFG)

- Example: language of balanced parenthesis

Terminals	{ }
Non-terminals	S
Start Symbol	S
Productions	[illegible] S→{S}S [illegible] S→ ε

CONTEXT FREE GRAMMARS (CFG)

- A string is in L(G) if it exists a derivation of that string
- Example: is {{}}{} in L(G)?

[illegible] S→{S}S

[illegible] S→ ε

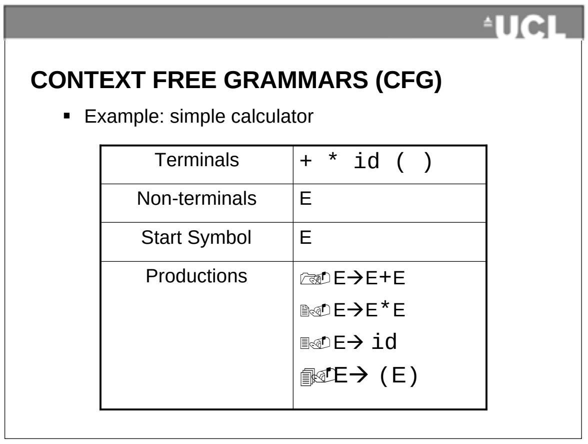
CONTEXT FREE GRAMMARS (CFG)

- A string is in L(G) if it exists a derivation of that string
- Example: is {{}}{} in L(G)?

[illegible] S→{S}S

[illegible] S→ ε

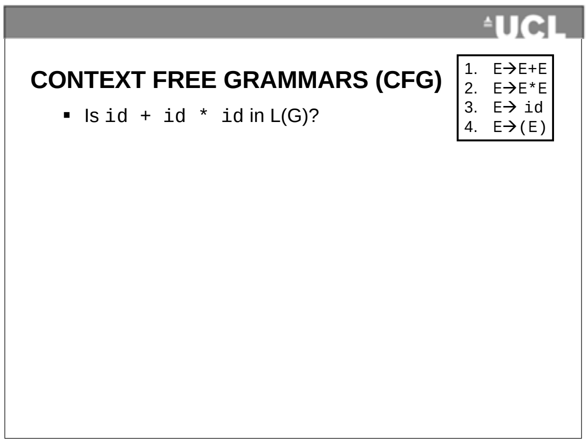
$$\underline{S} \rightarrow \{\underline{S}\}S \rightarrow \{\{\underline{S}\}S\}S \rightarrow \{\{\}\underline{S}\}S \rightarrow$$
$$\{\{\}\}\underline{S} \rightarrow \{\{\}\}\{\underline{S}\}S \rightarrow \{\{\}\}\{\}\underline{S} \rightarrow \{\{\}\}\{\}$$



CONTEXT FREE GRAMMARS (CFG)

- Example: simple calculator

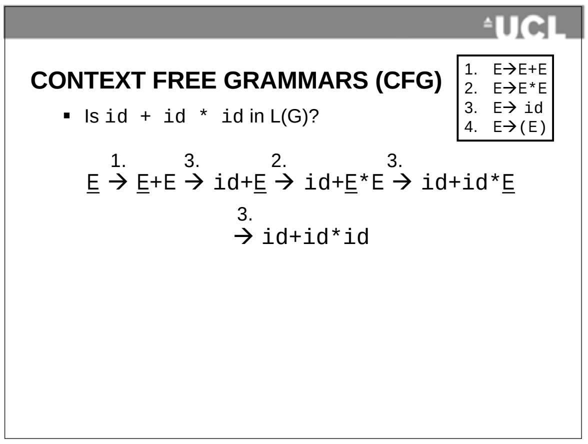
Terminals	`+ * id ( )`
Non-terminals	E
Start Symbol	E
Productions	E→E+E E→E*E E→ id E→ (E)



CONTEXT FREE GRAMMARS (CFG)

1. E→E+E
2. E→E*E
3. E→ id
4. E→(E)

- Is `id + id * id` in L(G)?



CONTEXT FREE GRAMMARS (CFG)

1. E→E+E
2. E→E*E
3. E→ id
4. E→(E)

- Is `id + id * id` in L(G)?

$$\underline{E} \xrightarrow{1.} \underline{E}+E \xrightarrow{3.} id+\underline{E} \xrightarrow{2.} id+\underline{E}*E \xrightarrow{3.} id+id*\underline{E}$$

$$\xrightarrow{3.} id+id*id$$

CONSTRUCTING A DERIVATION

- Start from start symbol S
- Use productions to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β, and γ, and for a production:

$$A \rightarrow \beta,$$

a single step of derivation is:

$$\alpha A \gamma \rightarrow \alpha\beta\gamma$$

CONSTRUCTING A DERIVATION

- Derivations:
  - Left-most=the left-most non-terminal symbols is always the one expanded
  - Right-most=the right-most non-terminal symbol is always the one expanded

$$\underline{E} \overset{\boxed{1}}{\rightarrow} \underline{E}+E \overset{\boxed{3}}{\rightarrow} id+\underline{E} \overset{\boxed{2}}{\rightarrow} id+\underline{E}*E \overset{\boxed{3}}{\rightarrow} id+id*\underline{E} \overset{\boxed{3}}{\rightarrow} id+id*id$$

$$\underline{E} \overset{\boxed{2}}{\rightarrow} E*\underline{E} \overset{\boxed{3}}{\rightarrow} \underline{E}*id \overset{\boxed{1}}{\rightarrow} E+\underline{E}*id \overset{\boxed{3}}{\rightarrow} \underline{E}+id*id \overset{\boxed{3}}{\rightarrow} id+id*id$$

$$\underline{E} \overset{\boxed{1}}{\rightarrow} E+\underline{E} \overset{\boxed{2}}{\rightarrow} E+\underline{E}*E \overset{\boxed{3}}{\rightarrow} \underline{E}+id*E \overset{\boxed{3}}{\rightarrow} id+id*\underline{E} \overset{\boxed{3}}{\rightarrow} id+id*id$$

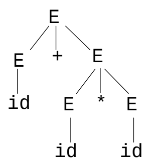
DERIVATIONS AND PARSE TREES

- Parse Tree = graphical (tree) representation of a derivation
  - Leaves = terminals
  - Intermediate nodes = non-terminals
  - Root = start symbol

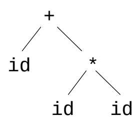
DERIVATIONS AND PARSE TREES

```
E→E+E→E+E*E→id+E*E→id+E*id
→id+id*id
```

Parse Tree
(concrete syntax tree)



Abstract Syntax Tree



ASTs contain only terminals

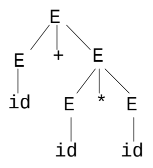
EXERCISE

- Given the grammar

```
S → ( L ) | a
L → L , S | S
```

  - Construct a left-most derivation for (a, (a,a))
  - Construct a right-most derivation for (a, (a,a))
  - Build the parse tree and the abstract syntax tree for the above derivations

DERIVATIONS AND PARSE TREES



```
E→E+E→E+E*E→id+E*E→id+E*id
→id+id*id

E→E+E→id+E→id+E*E→id+id*E
→id+id*id
```

- Note: parse trees contain no information about the order of the derivation steps
- Different derivations may have the same parse tree

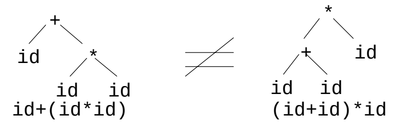
RECAP …

- Context-Free Grammar (CFG) describes the language of syntactically correct program
  - Correctness: it exists a derivation that produces the input token list
    - Left-most vs. right-most derivation
  - Parse-tree: graphical representation of a derivation
    - Concrete vs. abstract syntax tree

DERIVATIONS AND PARSE TREES

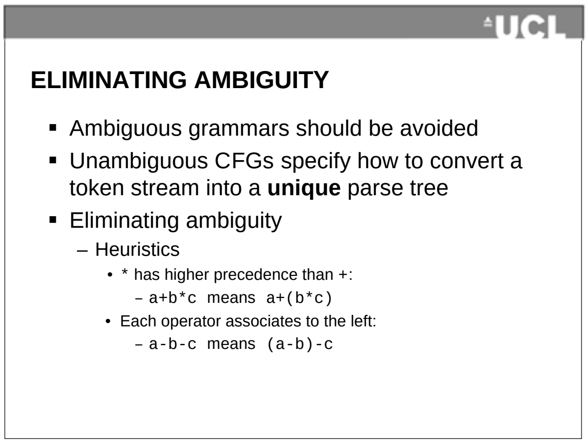
- Ambiguous grammar = there exist different left-most (or right-most) derivations for the same string
  - These derivations have different parse trees (thus different meaning)

```
E→E+E→id+E→id+E*E           E→E*E→E+E*E→id+E*E
→id+id*E →id+id*id           →id+id*E →id+id*id
```



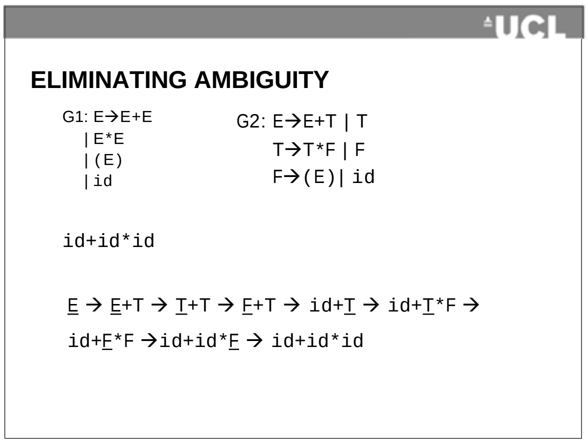
ELIMINATING AMBIGUITY

- Ambiguous grammars should be avoided
- Unambiguous CFGs specify how to convert a token stream into a **unique** parse tree
- Eliminating ambiguity
  - Heuristics
    - Adding non-terminals to enforce precedence
    - Allowing only left or right recursion (for associativity)



ELIMINATING AMBIGUITY

- Ambiguous grammars should be avoided
- Unambiguous CFGs specify how to convert a token stream into a **unique** parse tree
- Eliminating ambiguity
  - Heuristics
    - * has higher precedence than +:
      - `a+b*c means a+(b*c)`
    - Each operator associates to the left:
      - `a-b-c means (a-b)-c`

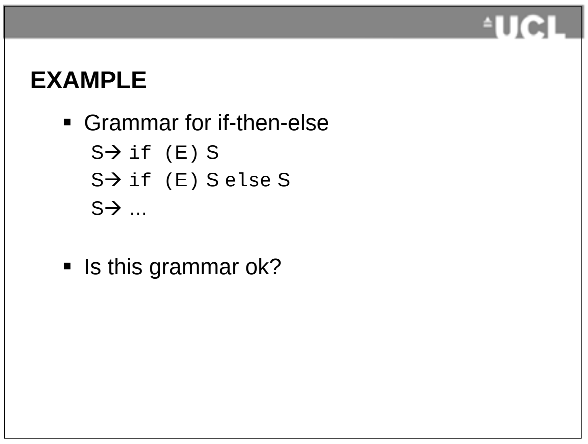


ELIMINATING AMBIGUITY

```
G1: E→E+E            G2: E→E+T | T
     |E*E                 T→T*F | F
     |(E)                 F→(E)| id
     |id
```

`id+id*id`

$\underline{E} \rightarrow \underline{E}+T \rightarrow \underline{T}+T \rightarrow \underline{F}+T \rightarrow id+\underline{T} \rightarrow id+\underline{T}*F \rightarrow$
$id+\underline{F}*F \rightarrow id+id*\underline{F} \rightarrow id+id*id$



EXAMPLE

- Grammar for if-then-else
  ```
  S→ if (E) S
  S→ if (E) S else S
  S→ …
  ```
- Is this grammar ok?

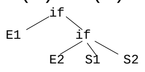
EXAMPLE

S→ `if (E)` S
S→ `if (E)` S `else` S

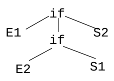
- How to parse:

  `if (E1) if(E2)` S1 `else` S2

<u>S</u>→`if(E)` <u>S</u> → `if (E) if (E)` S `else` S



<u>S</u>→ `if (E)` <u>S</u> `else` S → `if (E) if (E)` S `else` S

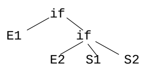


EXAMPLE

- Closest-if rule

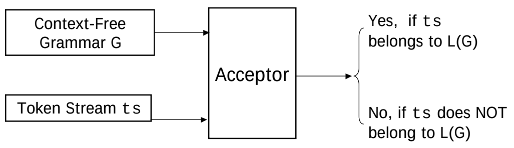
  statement→ matched | unmatched
  matched → `if (E)` matched `else` matched | S
  unmatched → `if (E)` statement |
  `if (E)` matched `else` unmatched

<u>statement</u>→<u>unmatched</u> → `if(E)` <u>statement</u> →
`if (E)` <u>matched</u> →`if (E)if(E)` <u>matched</u> `else` matched→
`if (E) if(E)` S `else` <u>matched</u>→
`if (E) if(E)` S `else` S



WHAT'S NEXT?

- Acceptors for context-free grammars



- Syntax Analysers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted