Programming Languages
Lecture03 – Scanner and Parse Tree

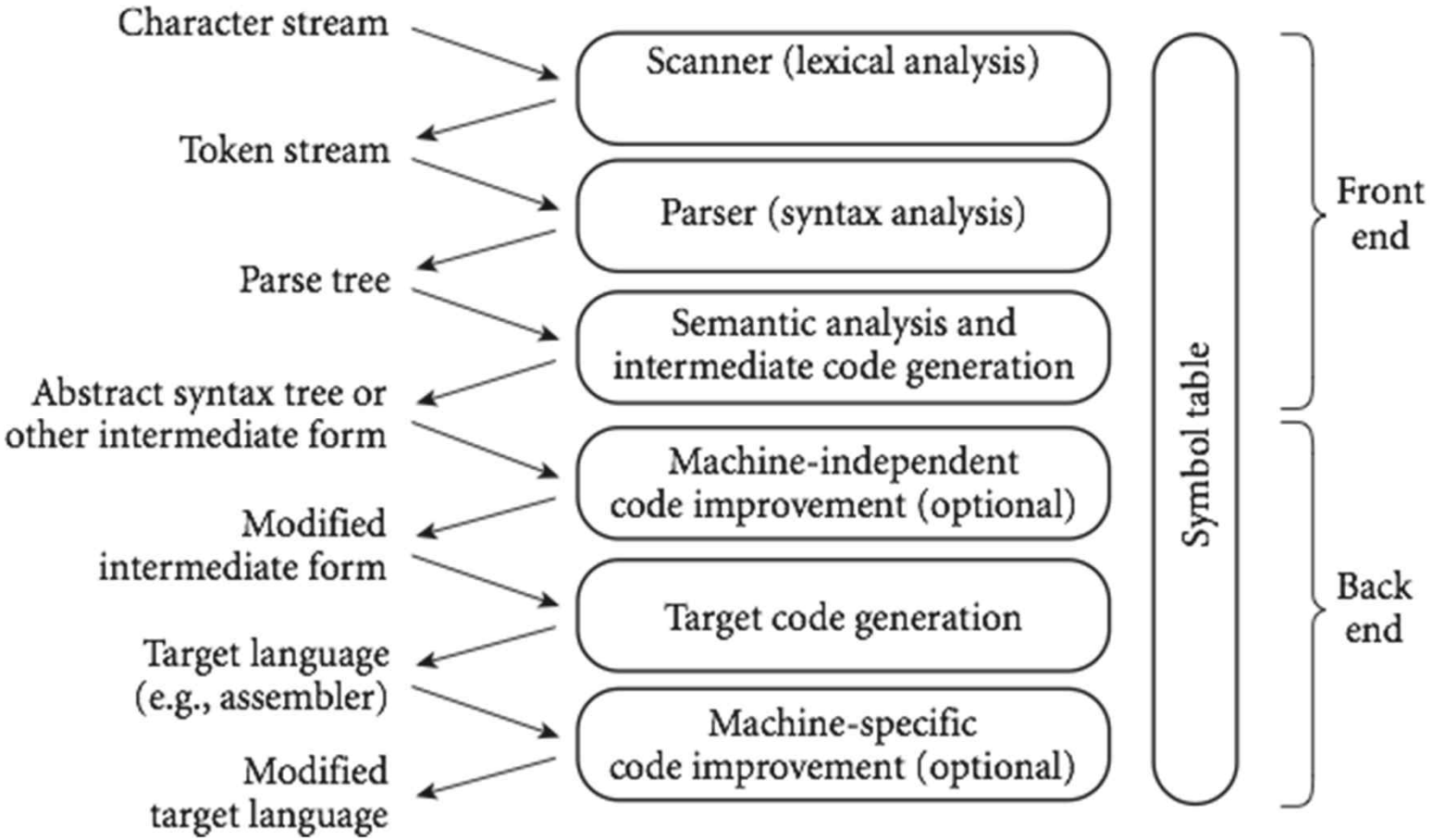
SUNG KYUN KWAN
UNIVERSITY

남 범 석

bnam@skku.edu

An Overview of Compilation

- Phases of Compilation



Scanning

- Recognition of a regular language
  - e.g., via DFA

  - Divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
  - We can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
  - You can design a parser to take characters instead of tokens as input, but it isn't pretty

Parsing

- Recognition of a context-free language
  - e.g., via push-down automata (PDA)

  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams

Semantic Analysis

- Discovery of meaning in the program

  - Compiler actually does STATIC semantic analysis.
    - That's the meaning that can be figured out at compile time
  - Some things (e.g., array subscript out of bounds) can't be figured out until run time.
    - Things like that are part of the program's DYNAMIC semantics

Intermediate Form

- Generated after semantic analysis (if the program passes all checks)
  - Intermediate Form (IF) = Intermediate Representation (IR)
  - = Intermediate Code
  - IFs are often chosen for machine independence, ease of optimization, or compactness
  - They often resemble machine code for some imaginary idealized machine
    - e.g., a stack machine, or a machine with arbitrarily many registers
  - Many compilers actually move the code through more than one IF

Optimization & Code Generation

- Optimization takes an intermediate-code program and produces another one that does the same thing faster, or in less space
  - "Optimization" is a misnomer; we just improve code
  - Optimization is optional, but essential for performance
- Code generation produces assembly languages or relocatable machine languages
- Certain machine-specific optimizations may be performed during or after target code generation
  - E.g., use of special instructions or addressing modes, etc.

Symbol Table

- All phases rely on a symbol table
  - Symbol table keeps track of all the identifiers in the program and what the compiler knows about them

- This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

Lexical and Syntax Analysis by Example

- GCD Program (in C)

```
int main() {
  int i = getint(), j = getint();

  while (i != j) {
    if (i > j)  i = i - j;
    else  j = j - i;
  }

  putint(i);
}
```

Regular Expressions for Tokens

- Tokens are specified with regular expressions
- Numerical literals in Pascal may be generated by the following:

$$
\begin{aligned}
\mathit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{unsigned\_integer} &\longrightarrow \mathit{digit}\ \mathit{digit}\,\texttt{*} \\
\mathit{unsigned\_number} &\longrightarrow \mathit{unsigned\_integer}\ (\ (\ \texttt{.}\ \mathit{unsigned\_integer}\ ) \mid \epsilon\ ) \\
&\qquad (\ (\ (\ \texttt{e} \mid \texttt{E}\ )\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \mathit{unsigned\_integer}\ ) \mid \epsilon\ )
\end{aligned}
$$

Lexical Analysis

- GCD Program Tokens
  - Scanning (lexical analysis) groups characters into tokens, the smallest meaningful units of the program

```
int     main   (   )       {
int     i      =   getint   (   )   ,   j   =   getint   (   )   ;
while   (      i   !=       j   )   {
if      (      i   >        j   )   i   =   i   -       j   ;
else    j      =   j        -   i   ;
}
putint  (      i   )       ;
}
```

Syntax Analysis

- Parsing recognize the structure of the program

- Context-Free Grammar and Parsing
  - Parsing organizes tokens into a parse tree that represents higher-level constructs in terms of their constituents
  - Potentially recursive rules known as context-free grammar define the ways in which these constituents combine

Context-Free Grammars for While Loop

```
iteration-statement → while ( expression ) statement

statement → compound-statement

compound-statement → { block-item-list_opt }

block-item-list_opt → block-item-list
                    | ϵ
block-item-list → block-item
                | block-item-list block-item
block-item → declaration
           | statement
```

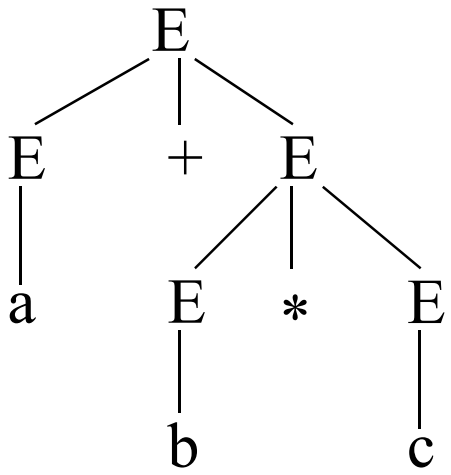
Context-Free Grammars for Expressions

- Expression grammar with precedence and associativity

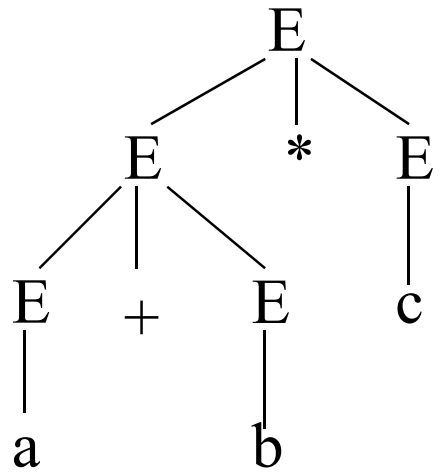
- ***expr → term | expr add_op term***
- ***term → factor | term mult_op factor***
- ***factor → id | number | - factor | (expr)***
- ***add_op → + | -***
- ***mult_op → * | /***

  - CFG for real programming languages are a lot more complicated

Parse Tree (Derivation Tree)



Q: Is this the only parse tree for a + b * c?



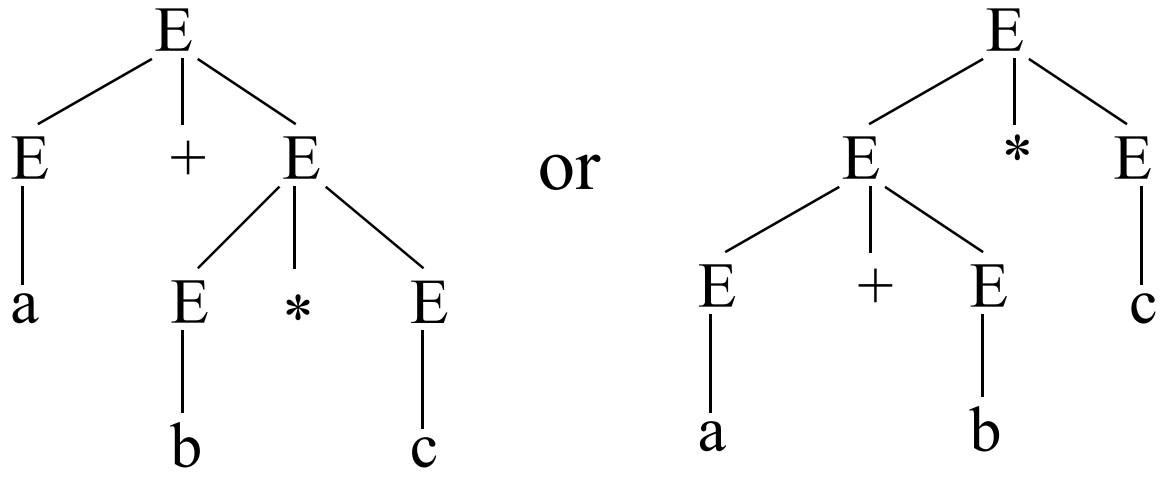
Practice - Draw parse trees for
(a * b) * c
a + (a)

Leftmost vs Rightmost Derivations

- Definition: A leftmost derivation is one in which the leftmost non-terminal is always expanded at each step (rightmost derivation is same, replacing leftmost with rightmost)
- Rightmost derivation for a + b * c
  - $E \Rightarrow E * E \Rightarrow E * c \Rightarrow E + E * c \Rightarrow E + b * c \Rightarrow a + b * c$
- Leftmost derivation for a + b * c
  - $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + b * E \Rightarrow a + b * c$

Ambiguity

- Definition: An ambiguous sentence is one that has 2 or more derivations



- A grammar is ambiguous if it generates ambiguous sentences.

Ambiguous Sentences

- Why do we care?
  - If the strings represent expressions in a programming language, do the 2 derivations compute the same value? NO
  - We should avoid ambiguous grammars if possible.

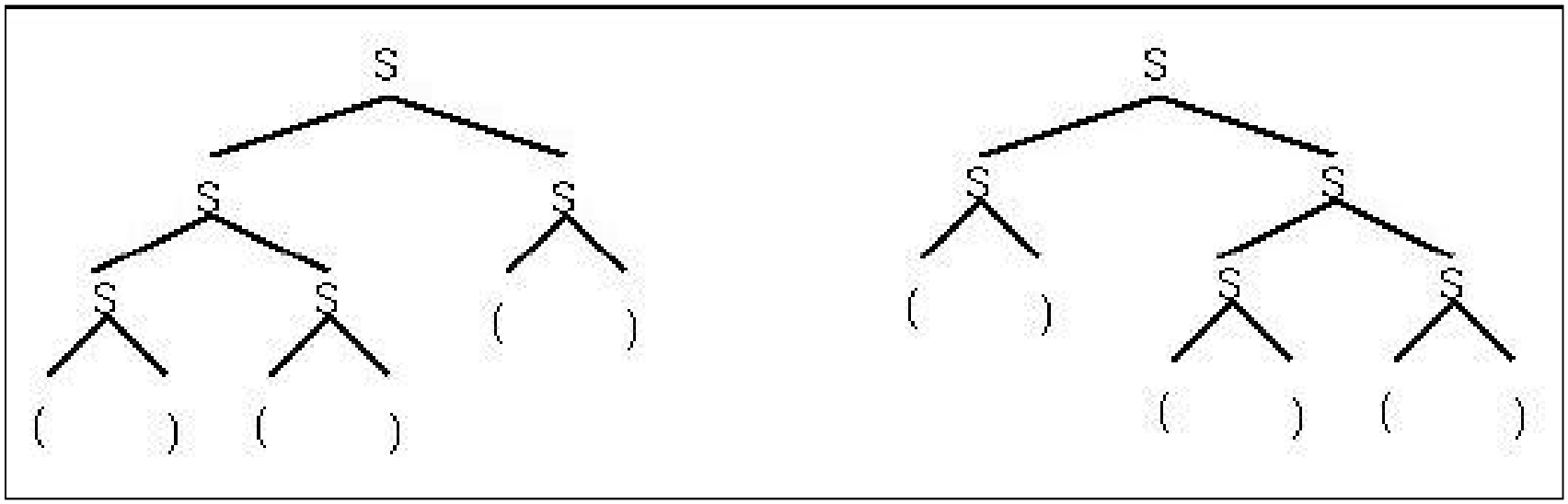
  - Q: Are these grammars ambiguous?
  - (1) $S \rightarrow aS \mid T$
    $T \rightarrow bT \mid U$
    $U \rightarrow cU \mid \varepsilon$

    There's at most one non-terminal in each sentential form, so there's no choice between left or right non-terminals to expand
  - (2) $S \rightarrow SS \mid () \mid (S)$

    How about ()()() ?

Ambiguous Sentences

- 2 different parse trees for the same string: ()()()
- 2 distinct leftmost derivations :
  - S ⇒ SS ⇒ SSS ⇒()SS ⇒()()S ⇒()()()
  - S ⇒ SS ⇒ ()S ⇒()SS ⇒()()S ⇒()()()



Avoiding Ambiguity

- How can we fix the problem?
  - 1. Precedence - which productions are applied first
  - 2. Associativity - which productions of equal precedence are applied first

- Enforcing precedence
  - the idea is that higher precedence productions should be made to occur later in a (leftmost) derivation

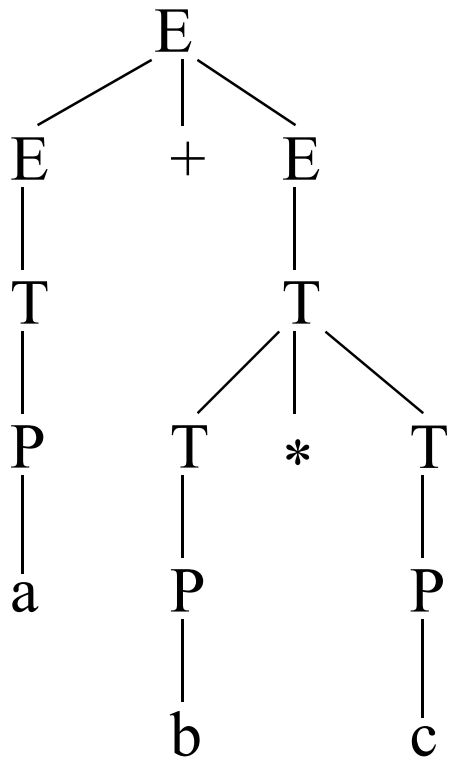
Precedence

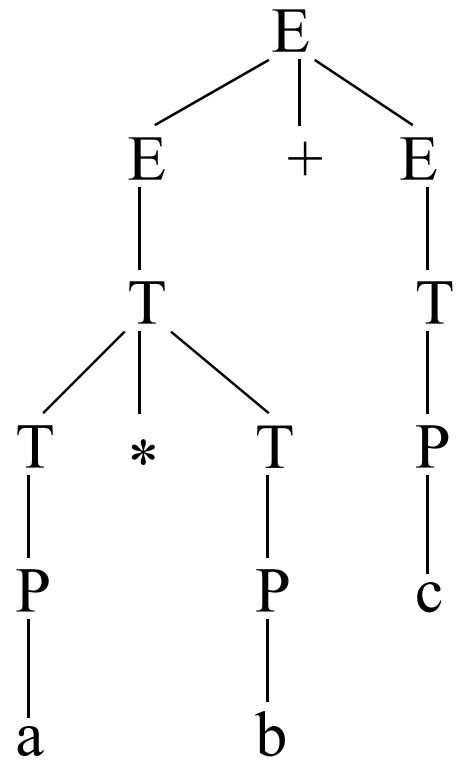
- Example: another grammar for arithmetic expressions, with precedence
  (* higher than +), to remove ambiguity
  - $E \rightarrow E + E \mid T$
  - $T \rightarrow T * T \mid P$
  - $P \rightarrow id \mid (E)$
- The grammar is not ambiguous for the example string a + b * c

Precedence (cont.)

- Derivation trees for a + b * c and a * b + c

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



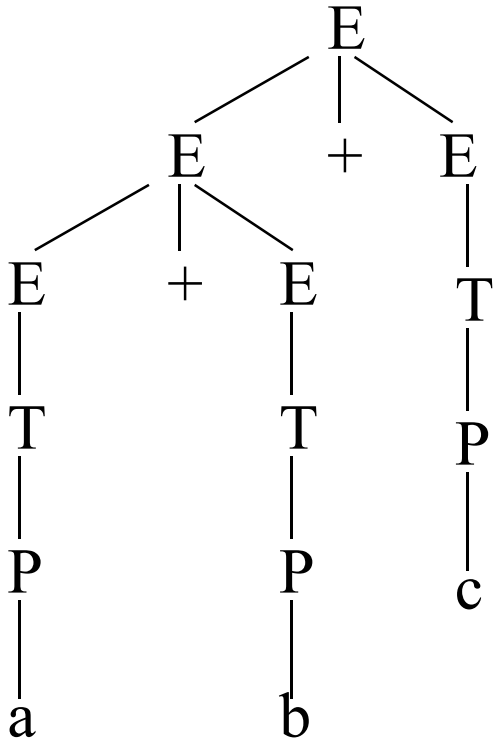


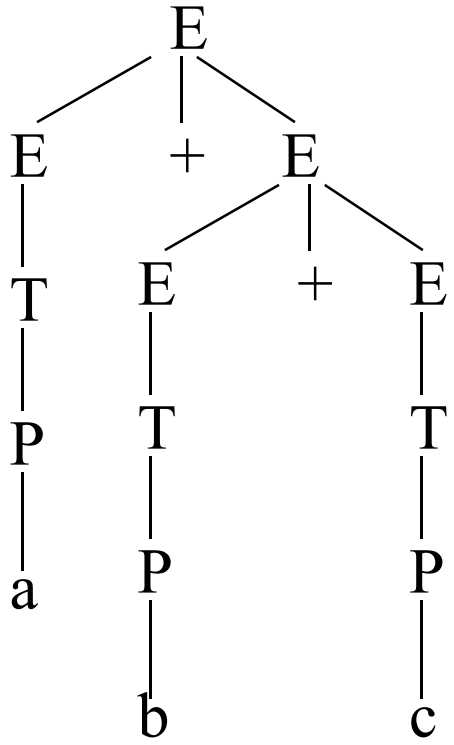
Associativity

- Precedence fixed the ambiguity problem for the example, but what about the string
  a + b + c?
- There are 2 leftmost derivations

Associativity (cont.)

$$a + b + c$$





Associativity (cont.)

- Again, why does the ambiguity matter?
  - For the string a + b + c it doesn't, because the + operator is associative, but what about subtraction or division (- or /), which aren't?
- Example: Can interpret a - b - c as (a - b) - c or as a - (b - c), which evaluates to a - b + c, if evaluate from left to right

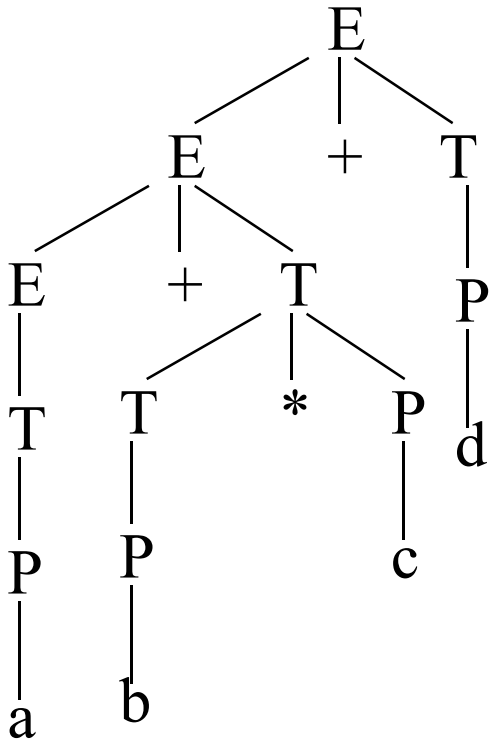
Associativity (cont.)

- Usually want left associativity, but how do you get that in a grammar?
  - for left associativity use a left recursive rule
  - for right associativity use a right recursive rule
- Grammar
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * P \mid P$
  - $P \rightarrow id \mid (E)$
- Can build one leftmost derivation for a string with both associativity and precedence required - e.g., a + b * c + d

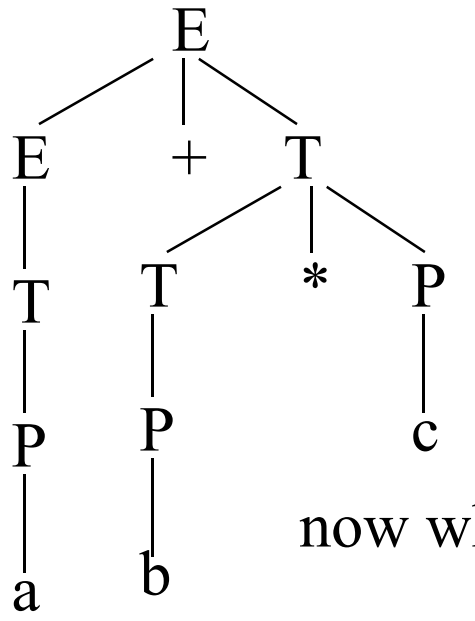
Parse Tree

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

$a + b * c + d$



and an alternate failed derivation



now what?

Summary

- Scanning
  - Lexical analysis
  - Group characters to a word (token)
  - Regular Expression
- Parsing
  - Syntax analysis using Context-Free Grammar
  - Build parse tree according to grammar
  - Although syntactically correct, there can be semantic problems
    - use variables before their definitions
    - no declarations/types for variables