中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第6讲：语法分析(2)

张献伟

xianweiz.github.io

DCS290, 3/19/2024

Review Questions

- Q1: RE to describe L={$a^n c b^n$}, where n>0?

  NO.

- Q2: is RL applicable to syntax analysis? Why?

  No. RL is not powerful enough, e.g., no nested structure

- Q3: input and output of parser?

  Input: tokens from lexer; output: parse tree or AST

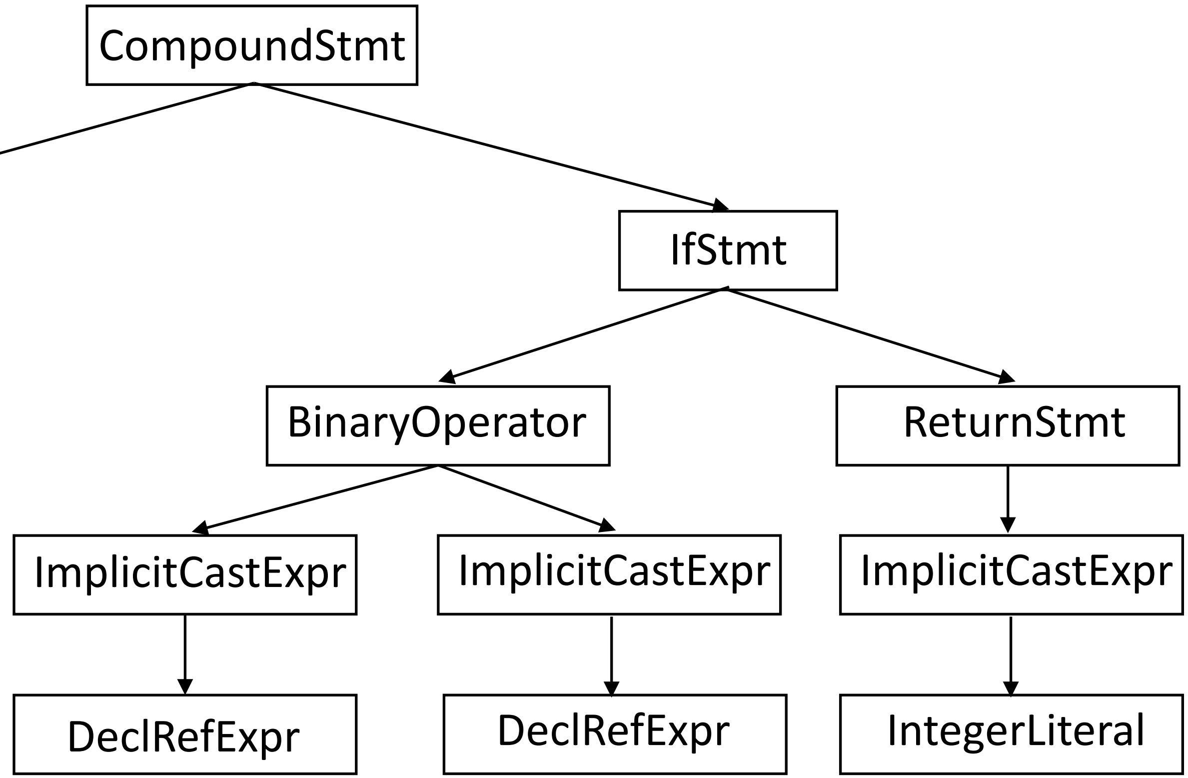
- Q4: how does grammar relate to syntax?

  Grammar is used to specify syntax.

- Q5: productions of grammar?

  LHS → *RHS*. e.g., E → E + E

Example (cont.)




```c
void main(){
  int a, b, c;
  if (b == c)
    return 1;
}
```

```
`-FunctionDecl 0x27999470 <parse.c:1:1, line:5:1> line:1:6 main 'void ()'
  `-CompoundStmt 0x27999800 <col:12, line:5:1>
    |-DeclStmt 0x279996f8 <line:2:3, col:14>
    | |-VarDecl 0x27999570 <col:3, col:7> col:7 a 'int'
    | |-VarDecl 0x279995f0 <col:3, col:10> col:10 used b 'int'
    | `-VarDecl 0x27999670 <col:3, col:13> col:13 used c 'int'
    `-IfStmt 0x279997e8 <line:3:3, line:4:12>
      |-BinaryOperator 0x27999780 <line:3:7, col:12> 'int' '=='
      | |-ImplicitCastExpr 0x27999750 <col:7> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x27999710 <col:7> 'int' lvalue Var 0x279995f0 'b' 'int'
      | `-ImplicitCastExpr 0x27999768 <col:12> 'int' <LValueToRValue>
      |   `-DeclRefExpr 0x27999730 <col:12> 'int' lvalue Var 0x27999670 'c' 'int'
      `-ReturnStmt 0x279997d8 <line:4:5, col:12>
        `-ImplicitCastExpr 0x279997c0 <col:12> 'void' <ToVoid>
          `-IntegerLiteral 0x279997a0 <col:12> 'int' 1
```


https://www.cnblogs.com/jourluohua/p/14524955.html

Syntax Analysis[语法分析]

- Informal description of variable declarations in C[变量声明]
  - Starts with *int* or *float* as the first token[类型]
  - Followed by one or more *identifier* tokens, separated by token *comma*[逗号分隔的标识符]
  - Followed by token *semicolon*[分号]
- To *check* *whether a program is well-formed* requires a specification of *what is a well-formed program*[语法定义]
  - The specification be precise[正确]
  - The specification be complete[完备]
    - Must cover all the syntactic details of the language
  - The specification must be convenient[便捷] to use by both language designer and the implementer
- A **context free grammar** meets these requirements

Example

- Grammar $G = \{T, N, s, \delta\}$
  - $T = \{0, 1\}$
  - $N = \{A, B\}$
  - $s = A$
  - $\delta = \{ A \rightarrow 0A \mid 1A \mid 0B,\ B \rightarrow 0 \}$

- Derivation: from grammar to language[文法到语言]
  - $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
  - $A \Rightarrow 1A \Rightarrow 10B \Rightarrow 100$
  - $A \Rightarrow 0A \Rightarrow 00A \Rightarrow 000B \Rightarrow 0000$
  - $A \Rightarrow 0A \Rightarrow 01A \Rightarrow \ldots$
  - … …

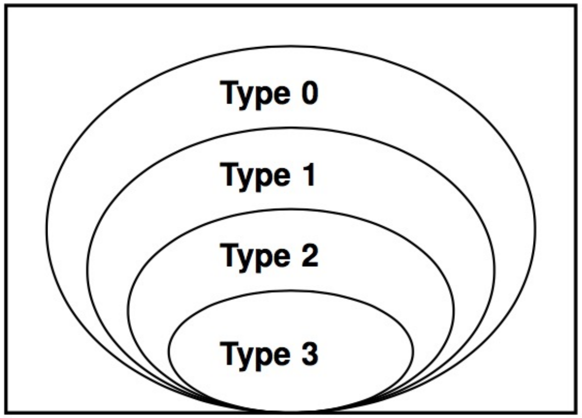
Sentence

Sentential form

Language Classification: Chomsky

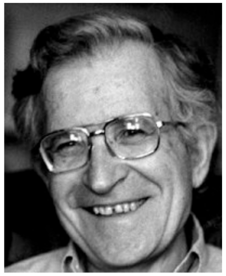
- **Language classification** based on form of grammar rules

- Four types of grammars:
  - Type 0 — unrestricted grammar
    - 0型文法 – 无限制文法
  - Type 1 — context sensitive grammar(CSG)
    - 1型文法 – 上下文有关文法
  - Type 2 — context free grammar (CFG)
    - 2型文法 – 上下文无关文法
  - Type 3 — regular grammar
    - 3型文法 – 正则文法



- Regular Grammar ⊆ CFG ⊆ CSG ⊆ Unrestricted Grammar

Chomsky



Chomsky hierarchy

In 1957, Noam Chomsky published *Syntactic Structures*, an landmark book that defined the so-called Chomsky hierarchy of languages

American linguist, philosopher, cognitive scientist, historian, and activist.

His work has influenced fields such as computer science, mathmatics and psychology.

Linguistics [change | change source]

Chomsky created the theory of generative grammar. This is one of the most important contributions to the field of linguistics made in the 20th century. He also helped start the cognitive revolution in psychology through his review of B. F. Skinner's *Verbal Behavior*. He challenged the behaviorist way of looking at behavior and language. This was the main approach used in the 1950s. His natural approach to the study of language also changed the philosophy of language and mind. He also invented the Chomsky hierarchy, a way of looking at formal languages in terms of their power to explain language.

According to the Arts and Humanities Citation Index in 1992, Chomsky was cited as a source more often than any other living scholar during the 1980–1992 time period. He was the eighth-most cited scholar in any time period.[1][2][3]

Type 0: Unrestricted Grammar

- Form of rules $\alpha \to \beta$
  - where $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$
- Implied restrictions
  - LHS: no ε allowed
- Example:
  - aB → aCD: LHS is shorter than RHS
  - aAB → aB: LHS is longer than RHS
  - A → ε: ε-productions are allowed
- Derivations
  - Derivation strings may contract and expand repeatedly (since LHS may be longer or shorter than RHS)
  - Unbounded number of productions before target string

Type 1: Context Sensitive Grammar

- Form of rules: $\alpha A\beta \to \alpha\gamma\beta$
  - where $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $\gamma \in (N \cup T)^+$
- Replace A by γ only if found in the context of α and β
- Implied restrictions
  - LHS: shorter or equal to RHS
  - RHS: no ε allowed
- Example:
  - aAB→aCB: replace *A* with *C* when in between *a* and *B*
  - A → C: replace *A* with *C* regardless of context
- Derivations
  - Derivation strings may only expand
  - Bounded number of derivations before target string

SUN YAT-SEN UNIVERSITY

Type 2: Context Free Grammar

- Form of rules: $A \rightarrow \gamma$
  - where $A \in N$, $\gamma \in (N \cup T)^+$
- Replace A by γ (no context can be specified)
- Implied restrictions
  - LHS: a single non-terminal
  - RHS: no ε allowed
    - Sometimes relaxed to simplify grammar but rules can always be rewritten to exclude ε-productions
- Example:
  - A → aBc: replace *A* with *aBc* regardless of context

L = $\{a^n b^n \mid n \geq 0\}$ is **NOT regular** but **IS** a **context-free language**.

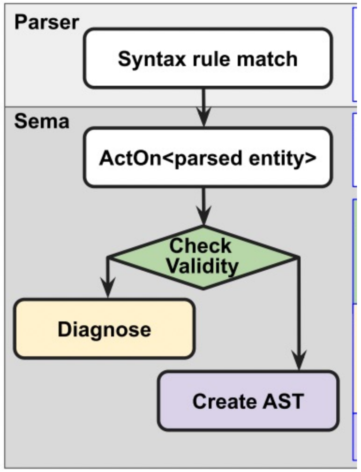
For the following CFG G = $< T, N, S, \delta >$ generates L:
$T = \{$ a, b $\}$, $N = \{$ S $\}$ and $\delta = \{$ S -> aSb , S -> ab $\}$

Type 3: Regular Grammar

- Form of rules $A \to \alpha$,or $A \to \alpha B$
  - where $A,B \in N$, $\alpha \in T$
- In terms of FA:
  - Move from state *A* to state *B* on input *α*
- Implied restrictions
  - LHS: a single non-terminal
  - RHS: a terminal or a terminal followed by a non-terminal
- Example: $A \to 1A \mid 0$
  - RE: **1*0**
- Derivation:
  - Derivation string length increases by 1 at each step

In Practice[实际中]

- Every regular language is a context-free language
  - Context-free languages are more general than regular languages
- If PLs are context-sensitive, why use CFGs for parsing?
  - Perfectly suited to describe recursive syntax of exprs & statements
  - CSG parsers are provably inefficient
  - Most PL constructs are context-free
    - if-stmt, declarations
  - The remaining context-sensitive constructs can be analyzed at the semantic analysis stage
    - e.g. def-before-use, matching formal/actual parameters
- In PLs
  - Regular language for lexical analysis
  - Context-free language for syntax analysis



Grammar and Derivation[文法与推导]

- **Grammar** is used to derive string or construct parser[文法]
- A **derivation** is a sequence of applications of rules[推导]
  - Starting from the start symbol
  - S⇒... ⇒... ⇒... ⇒ (sentence)
  - There are choices at each sentential form
    - choice of the nonterminal to be replaced[替换哪个？]
    - choice of a rule corresponding to the nonterminal[怎么替换？]
- Instead of choosing the nonterminal to be replaced, in an ***arbitrary*** fashion, it is possible to make an ***uniform*** choice at each step[统一化选择替换哪个]
- **Leftmost** and **Rightmost** derivations[最左和最右推导]
  - At each derivation step, leftmost derivation always replaces the leftmost non-terminal symbol
  - Rightmost derivation always replaces the rightmost one

Example

- Two derivations of string "id * id + id * id" using grammar:

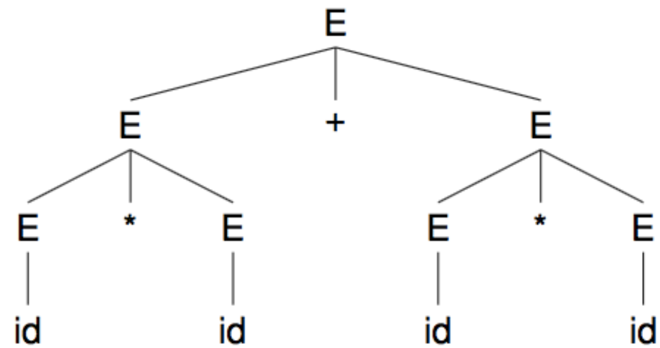
  $E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

- Leftmost derivation[最左推导]
  - $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow \text{id} * E + E \Rightarrow \text{id} * \text{id} + E \Rightarrow \ldots \Rightarrow \text{id} * \text{id} + \text{id} * \text{id}$
- Rightmost derivation[最右推导]
  - $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \text{id} \Rightarrow E + \text{id} * \text{id} \Rightarrow \ldots \Rightarrow \text{id} * \text{id} + \text{id} * \text{id}$

- Derivations can be summarized as a parse tree[分析树]

Parse Trees[分析树]

- Both previous derivations result in <u>the same</u> parse tree:



Two derivations of string
"id * id + id * id"
using grammar:
$E \rightarrow E*E \mid E+E \mid (E) \mid$ id

- A **parse tree** is a graphical representation of a derivation
  - But filters out the order in which productions are applied to replace non-terminals[过滤了推导顺序信息]
  - Each interior node represents the application of a production
    - Labeled with the non-terminal in the LHS of production
  - Leaves are labeled by terminals or non-terminals
    - Constitutes a sentential form (read from left to right)
    - Called the ***yield[产出]*** or ***frontier[边缘]*** of the tree

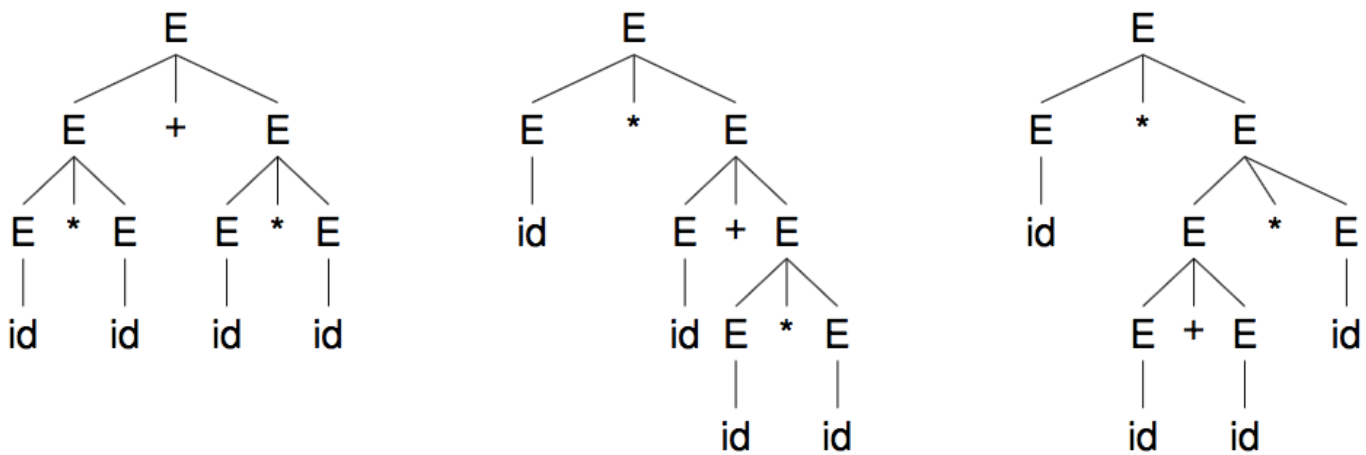
Parse Trees (cont.)

- Derivations and parse trees: many-to-one relationship
  - Leftmost derivation order: builds tree left to right
  - Rightmost derivation order: builds tree right to left
  - Different parser implementations choose different orders
  - One-to-one relationships between parse trees and either leftmost or rightmost derivations[最左或最右推导与分析树具有一对一对应关系]

- Program structure does not depend on order of rule application, instead it depends on what production rules are applied
  - Grammar must define unambiguously set of rules applied

Different Parse Trees

- Grammar *E→E*E* | *E+E* | (*E*) | id is ambiguous[二义的]
  - String id * id + id * id can result in 3 parse trees (and more)



The deepest sub-tree is traversed first, thus highest precedence

- Grammar can apply different rules to derive same string
  - Meaning of parse tree 1: (id * id) + (id * id)
  - Meaning of parse tree 2: id * (id + (id * id))
  - Meaning of parse tree 3: id * ((id + id) * id)

Preorder?
Inorder? ✓
Postorder?

Ambiguity[二义性]

- Grammar G is ambiguous if
  - It produces more than one parse tree for <u>some</u> sentence
  - i.e., there exist a string *str* ∈ L(G) such that
  - more than one parse tree derives *str*

    ≡ more than one leftmost derivation derives *str*

    ≡ more than one rightmost derivation derives *str*
- Unambiguous grammars are preferred for most parsers[文法最好没有歧义性]
  - Ambiguity of the grammar implies that at least some strings in its language have different structures (parse trees)
  - Thus, such a grammar is unlikely to be useful for a programming language, because two structures for the same string (program) implies two different meanings (executable equivalent programs) for this program

中山大學 SUN YAT-SEN UNIVERSITY

http://infolab.stanford.edu/~ullman/ialc/slides/slides7.pdf

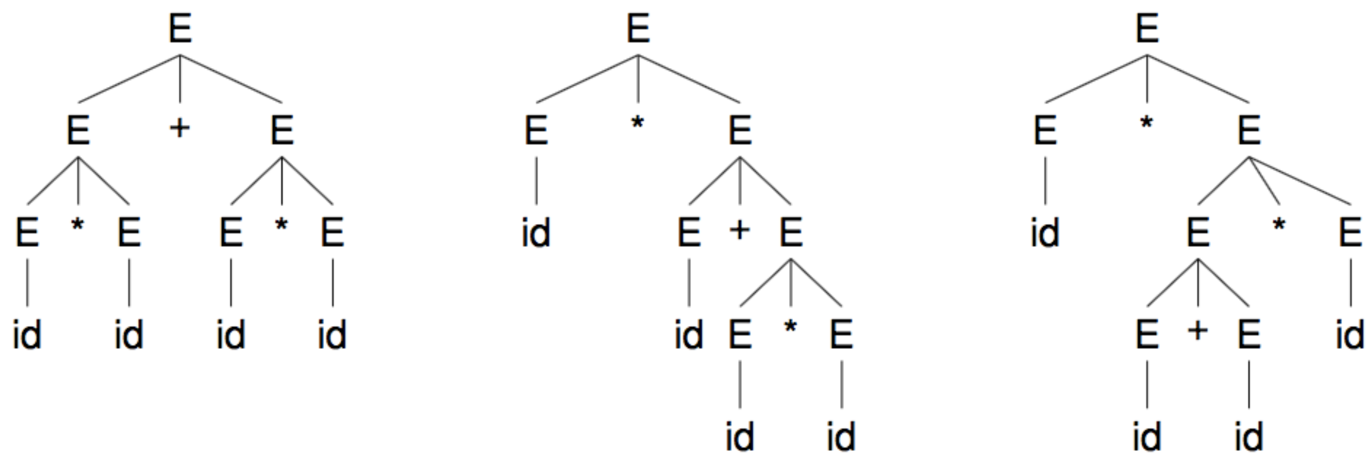
NSCC GZ

Ambiguity (cont.)

- Ambiguity is the property of the grammar, not the language
  - Just because G is ambiguous, does not mean L(G) is inherently ambiguous
  - A G′ can exist where G′ is unambiguous and L(G′) ≡ L(G)
- Impossible to convert ambiguous to unambiguous grammar automatically[歧义不能自动消除]
  - It is (often) possible to rewrite grammar to remove ambiguity
  - Or, use ambiguous grammar, along with disambiguating rules to "throw away" undesirable parse trees, leaving only one tree for each sentence (as in YACC)
    - A parse tree would be used subsequently for semantic analysis
    - Thus, more than one parse tree would imply several interpretations

Review Ambiguity Example

- Grammar $E \rightarrow E{*}E \mid E{+}E \mid (E) \mid \text{id}$ is ambiguous[二义的]
  - String id * id + id * id can result in 3 parse trees (and more)



The deepest sub-tree is traversed first, thus highest precedence

- Grammar can apply different rules to derive same string
  - Meaning of parse tree 1: (id * id) + (id * id)
  - Meaning of parse tree 2: id * (id + (id * id))
  - Meaning of parse tree 3: id * ((id + id) * id)

Remove Ambiguity[消除二义性]

a, b = 1, c = 2;
a = b = c;
b = a - b - c;
b = ???

- Specify precedence[指定优先级]
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity[指定结合性]
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$$
\begin{aligned}
E &\rightarrow E + E \mid T \\
T &\rightarrow T * T \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
$$

Possible to get id + (id + id) and (id + id) + id

// lowest precedence +
// middle precedence *
// highest precedence ()

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
$$

Now, can only have more ‘+’ on left

// + is left-associative
// * is left-associative

Remove Ambiguity (cont.)

$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$$E \rightarrow E + E \mid T$$
$$T \rightarrow T * T \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

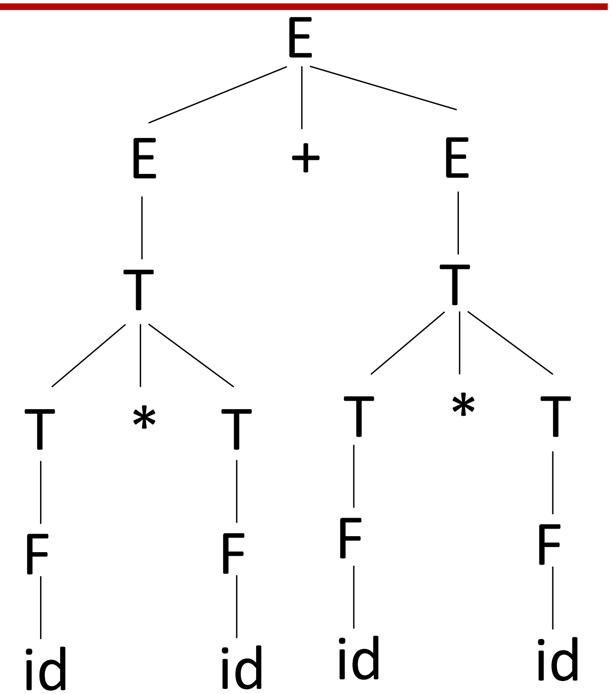
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



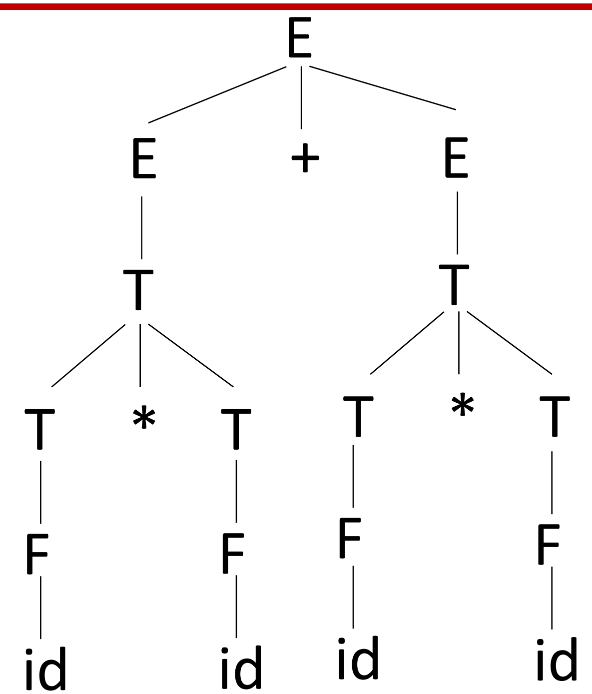
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E{*}E \mid E{+}E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



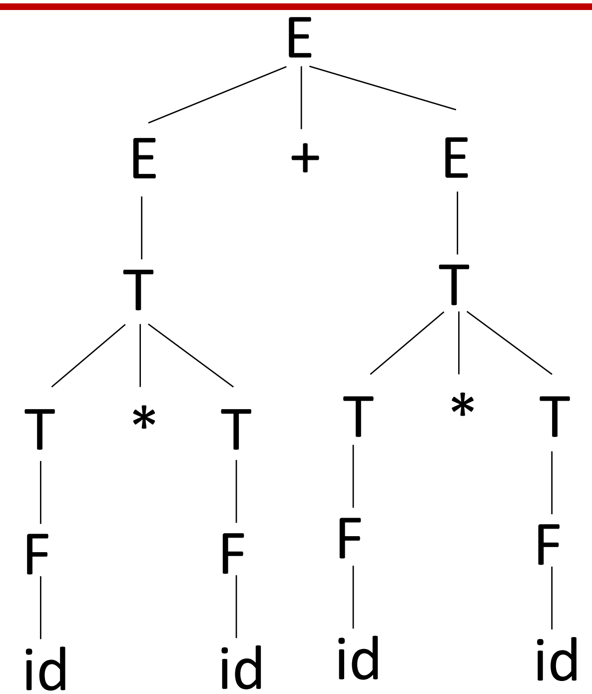
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



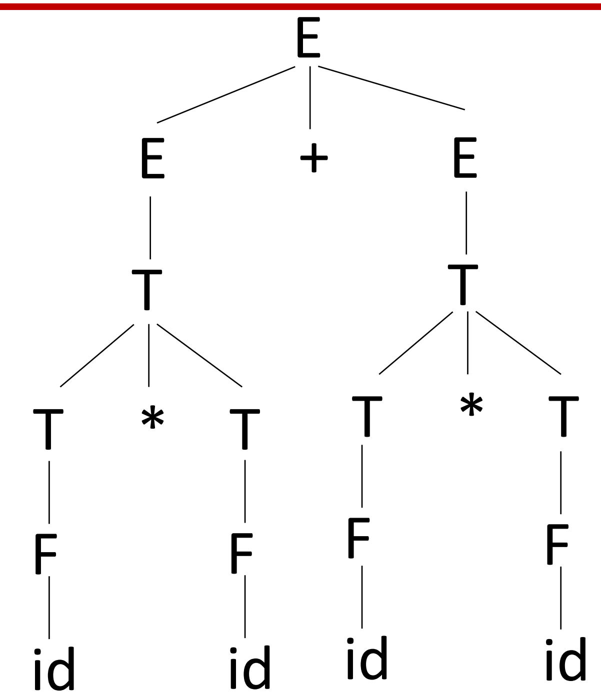
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id

Remove Ambiguity (cont.)

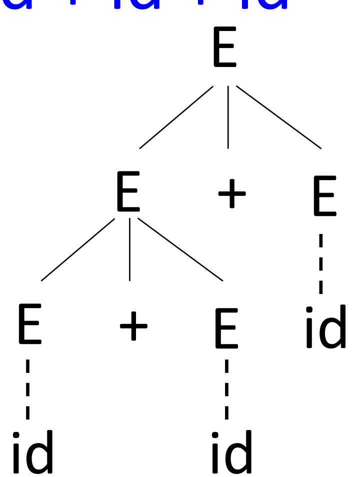
$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

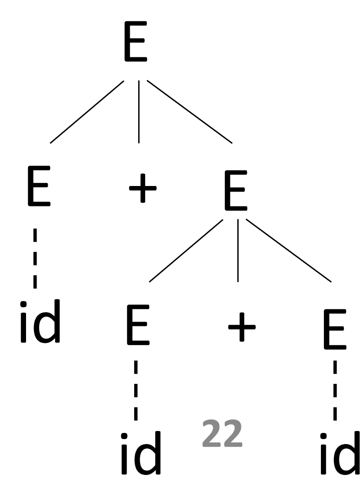
$$\boxed{\begin{aligned} E &\rightarrow E + E \mid T \\ T &\rightarrow T * T \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}}$$

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id



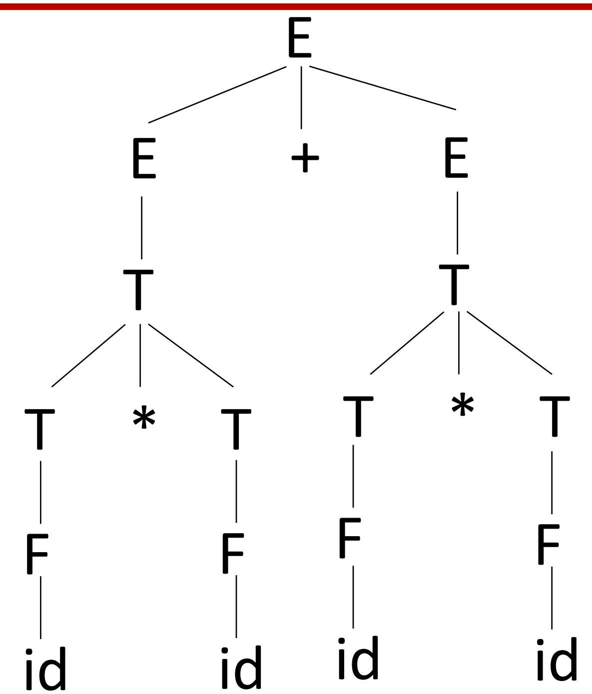


Remove Ambiguity (cont.)

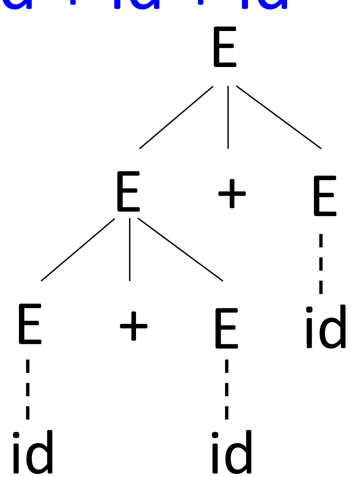
E→*E***E* | *E*+*E* | (*E*) | id

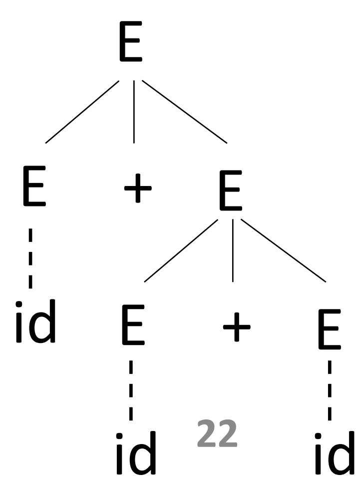
E → *E* + *E* | *T*
T → *T* * *T* | *F*
F → (*E*) | id

E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → (*E*) | id



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id



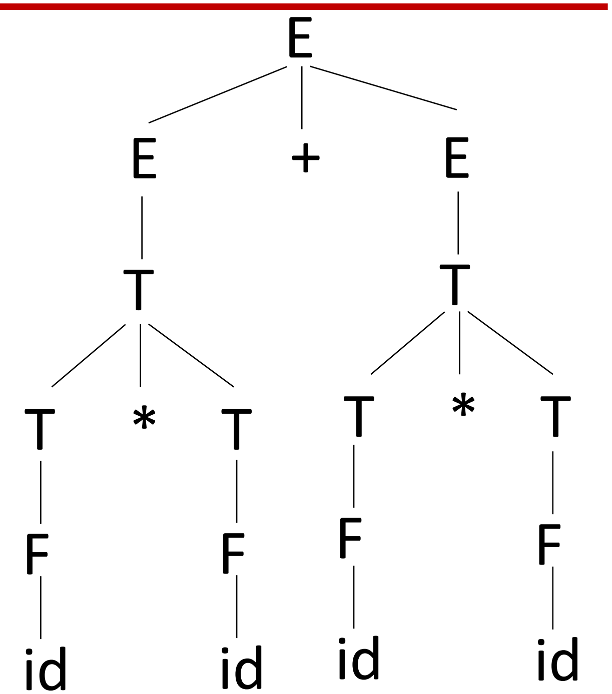


Remove Ambiguity (cont.)

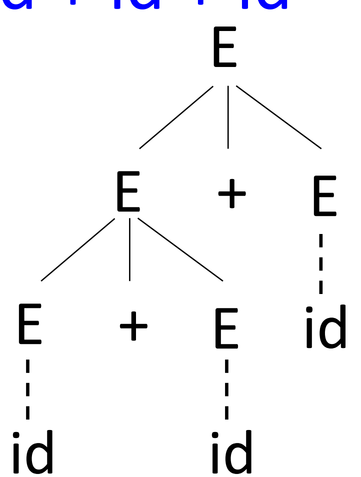
$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

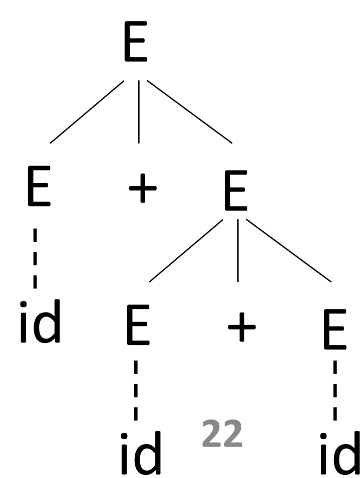
$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

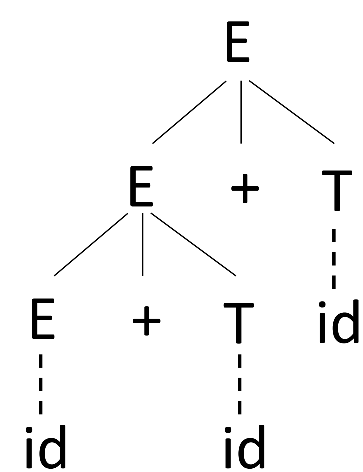
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id







Example

$E \rightarrow E*E \mid E\text{-}E \mid \mathrm{id}$

Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \text{id}$

Example

//precedence: * - same

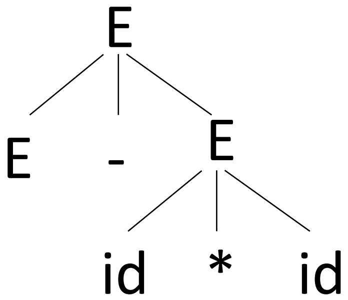
$E \rightarrow E*E \mid E\text{-}E \mid \mathrm{id}$ id - id * id

Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \text{id}$

id - id * id

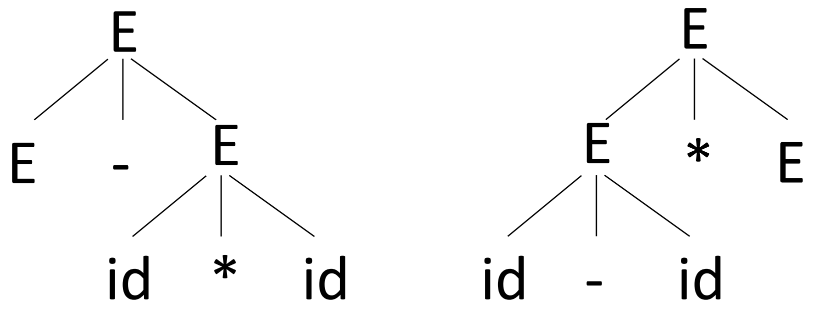


Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \mathrm{id}$

id - id * id

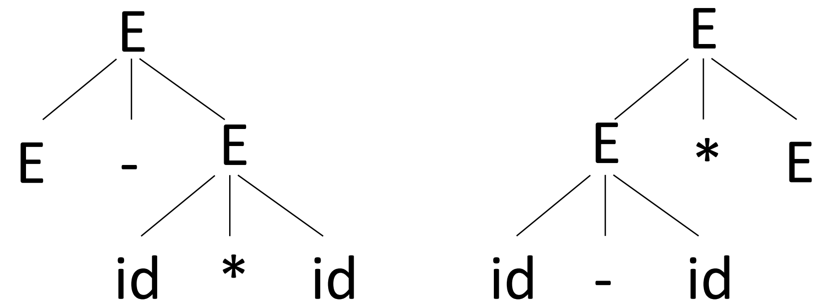


Example

//precedence: * - same

$E \rightarrow E * E \mid E - E \mid \mathrm{id}$

id - id * id



$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

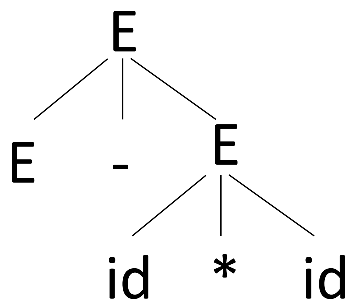
$F \rightarrow \mathrm{id}$

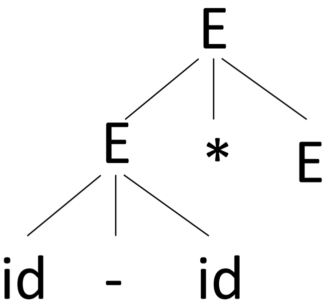
Example

//precedence: * - same

$E \rightarrow E * E \mid E - E \mid id$

id - id * id





//precedence: * is high

$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

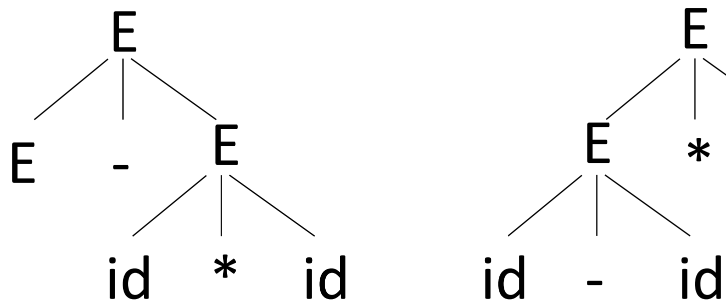
$F \rightarrow id$

Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \text{id}$

id - id * id



//precedence: * is high

$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

$F \rightarrow \text{id}$

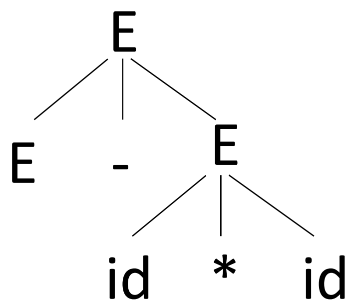
id - id - id

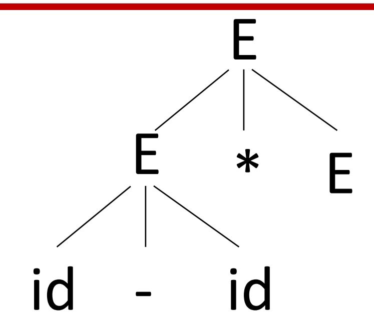
Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \text{id}$

id - id * id





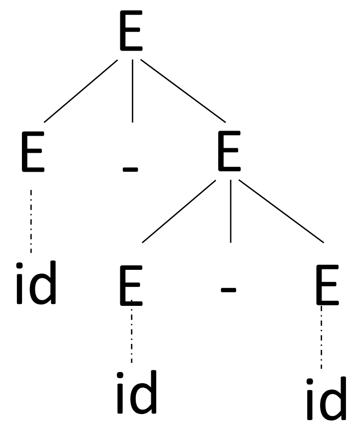
//precedence: * is high

$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

$F \rightarrow \text{id}$

id - id - id

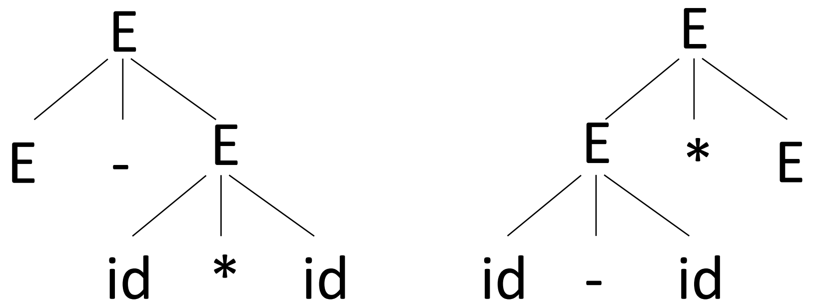


Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid \mathrm{id}$

id - id * id



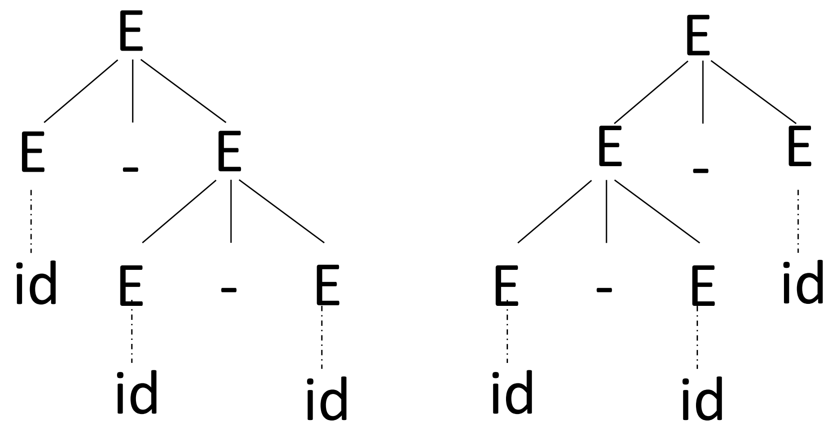
//precedence: * is high

$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

$F \rightarrow \mathrm{id}$

id - id - id

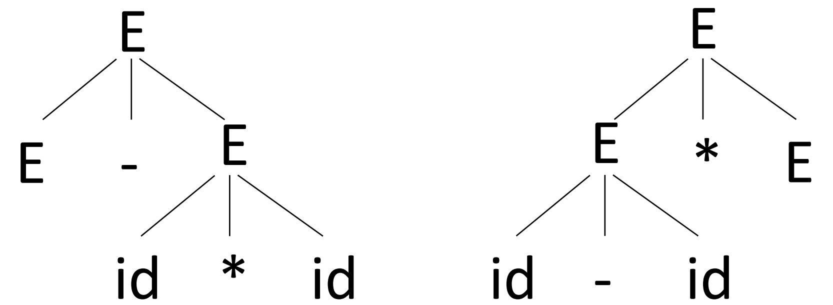


Example

$E \rightarrow E{*}E \mid E\text{-}E \mid \text{id}$

//precedence: * - same

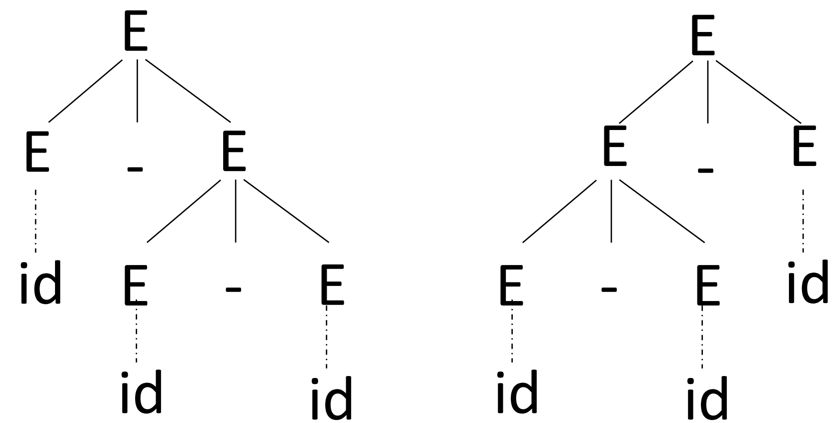
id - id * id



$E \rightarrow E - E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow \text{id}$

//precedence: * is high

id - id - id



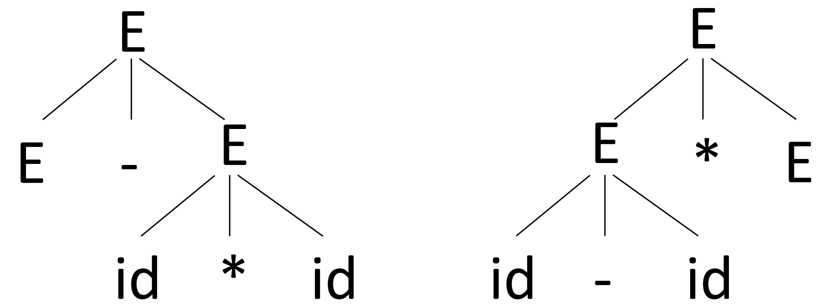
$E \rightarrow E - T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \text{id}$

Example

$E \rightarrow E*E \mid E\text{-}E \mid \text{id}$

//precedence: * - same

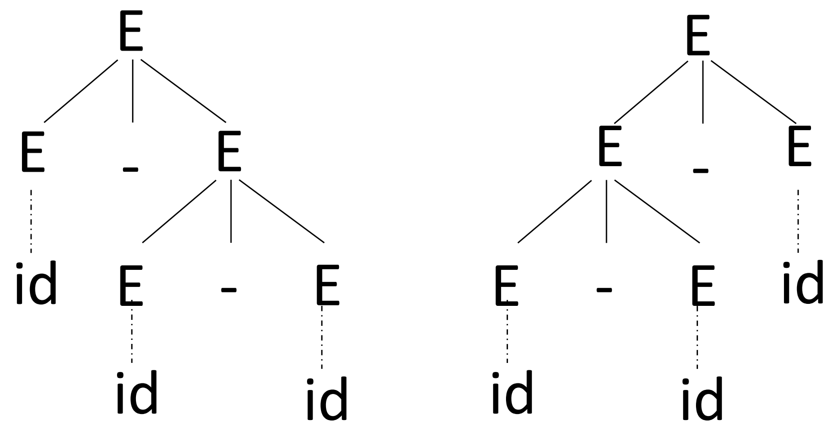
id - id * id



$E \rightarrow E - E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow \text{id}$

//precedence: * is high

id - id - id



$E \rightarrow E - T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \text{id}$

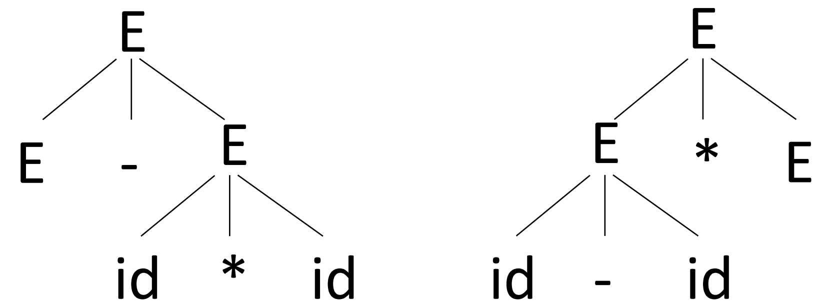
//precedence: * is high
//associativity: - is left

Example

$E \to E*E \mid E\text{-}E \mid \text{id}$

//precedence: * - same

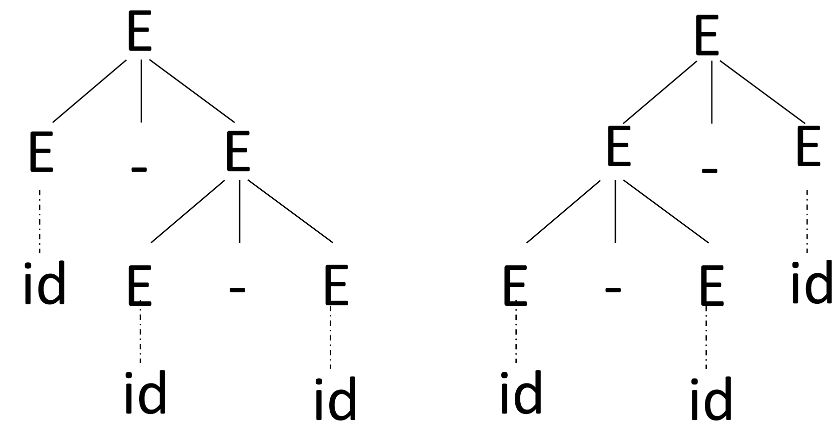
id - id * id



$E \to E - E \mid T$
$T \to T * T \mid F$
$F \to \text{id}$

//precedence: * is high

id - id - id



$E \to E - T \mid T$
$T \to T * F \mid F$
$F \to \text{id}$

//precedence: * is high
//associativity: - is left

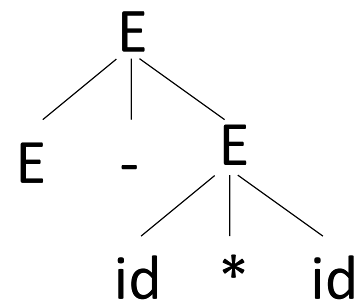
id - id - id

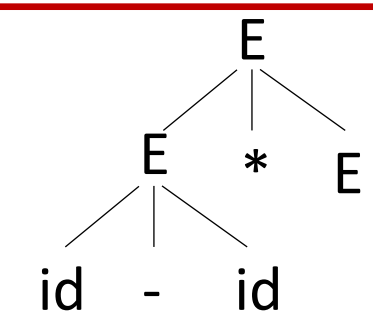
Example

//precedence: * - same

$E \rightarrow E*E \mid E\text{-}E \mid id$

id - id * id





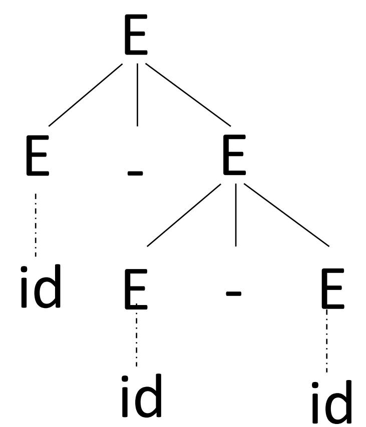
//precedence: * is high

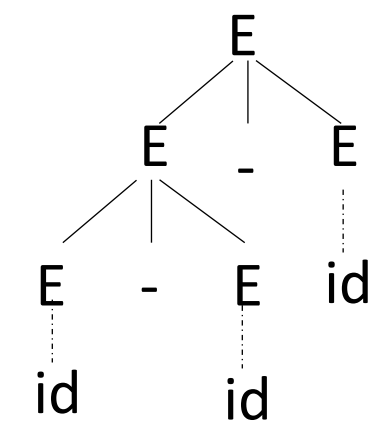
$E \rightarrow E - E \mid T$

$T \rightarrow T * T \mid F$

$F \rightarrow id$

id - id - id





//precedence: * is high

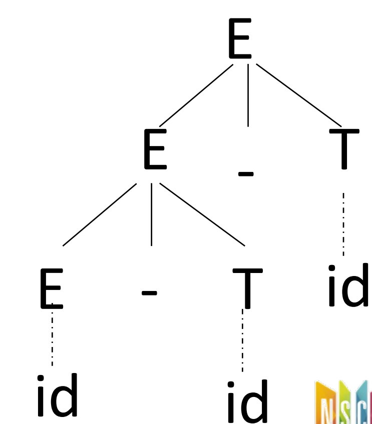
//associativity: - is left

$E \rightarrow E - T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

id - id - id



Grammar → Parser[文法到分析器]

- What exactly is **parsing**, or syntax analysis?[语法分析]
  - To process an input string for a given grammar,
  - and compose the derivation if the string is in the language
  - Two subtasks
    - determine if string can be derived from grammar or not[是否能推导？]
    - build a representation of derivation and pass to next phase[表示出来]
- What is the best representation of derivation?[推导表示]
  - Can be a parse tree or an abstract syntax tree
- An abstract syntax tree is[抽象语法树]
  - Abbreviated representation of a parse tree
  - Drops some details without compromising meaning
    - some terminal symbols that no longer contribute to semantics are dropped (e.g. parentheses)
    - internal nodes may contain terminal symbols

Example: Abstract Syntax Tree

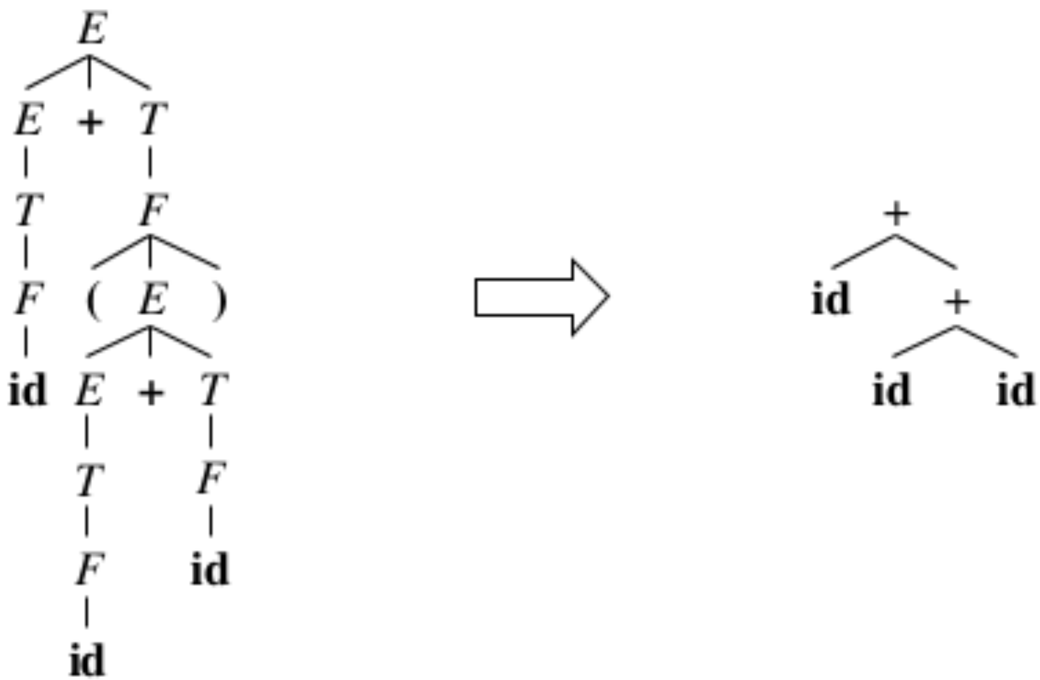
- AST: condensed form of parse tree
  - Operators and keywords do not appear as leaves (e.g., +)
  - Chains of single productions are collapsed (e.g., E -> T)

G:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

Input:

id + (id + id)



parse tree AST

Summary of CFG[小结]

- Compilers specify program structure using CFG
  - Most programming languages are not context free
  - Context sensitive analysis can easily be separated out to semantic analysis phase

- A parser uses CFG to
  - ... group lexical tokens to form expressions, statements, etc
  - ... answer if an input *str* ∈ L(G)
  - ... and build a parse tree
    - ... or build an AST instead
  - ... and pass it to the rest of compiler
    - ... or give an error message stating that *str* is invalid

Parser Implementation: Yacc + lex

parser.y

```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'

%%

lines : lines expr '\n' { printf("= %g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | NUMBER
     ;

%%

/*
int yylex() {
    int c;
    while ((c = getchar()) == ' ') ;
    if ((c == '.') || isdigit(c)) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
*/

int main() {
    if (yyparse() != 0)
        fprintf(stderr, "Abnormal exit\n");
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid num$

lexer.l

```
%{
 #define YYSTYPE double
 #include "y.tab.h"
 extern double yylval;
%}
number [0-9]+\.?|[0-9]*\.[0-9]+

%%

[ ]          { /* skip blanks */ }
{number}     { sscanf(yytext, "%lf", &yylval);
                 return NUMBER; }
\n|.         { return yytext[0]; }

%%

int yywrap(void) {
  return 1;
}
```

Generated by Yacc

Defined in y.tab.c