Compilation Principle
编 译 原 理

第6讲：语法分析(3)

张献伟

xianweiz.github.io

DCS290, 3/10/2022

Review Questions

- Formal definition of Grammar?

  (T, N, s, σ): T – terminals; N – non-terminals, s – start, σ – productions

- Grammar G: *stmt* → **if** (*expr*) *stmt* **else** *stmt*
  | **while** (*expr*) *stmt* | *v*
  expr → true | false

  N = { *stmt expr* }

- Is **if** (true) *stmt* **else** *v* an sentence of grammar G?

  NO. It is a sentential form (句型), as *stmt* is non-terminal symbol.

- Is **while** (false) **if** (true) *v* **else** *v* an sentence of G?

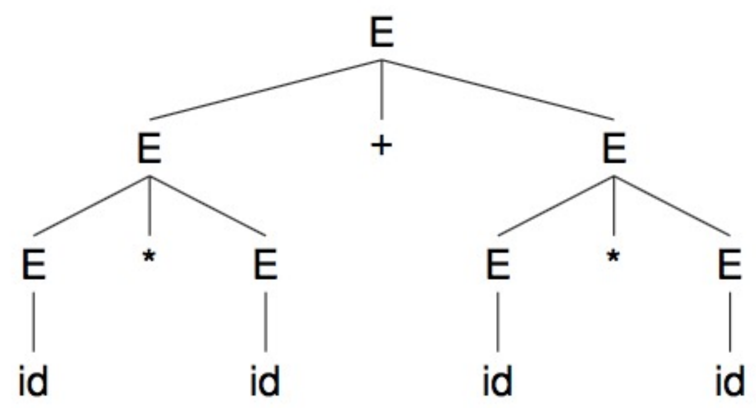
  YES. It can be derived using the production rules.

- Describe the languages generated by G: *list* → *list*, **id** | **id**?

  A list of one or more ids separated by commas.

Parse Trees[分析树]

- Both previous derivations result in the same 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

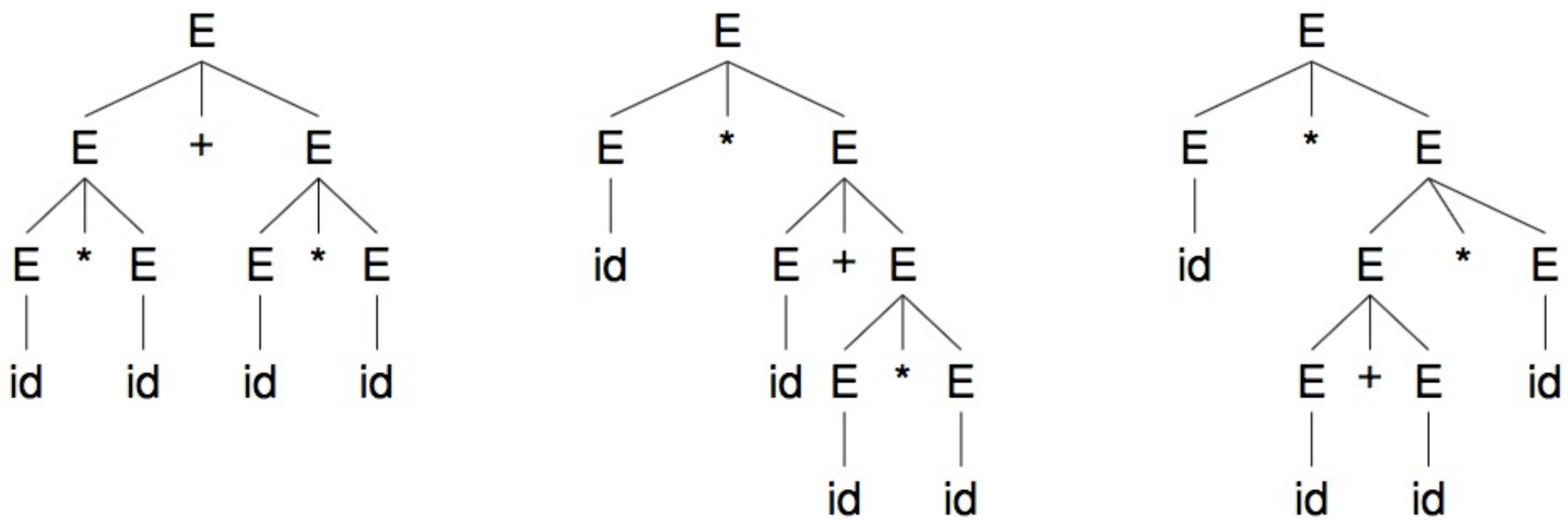
Parse Trees (cont.)

- 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 <u>order</u> of rule application, instead it depends on <u>what</u> 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 higher 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?

SUN YAT-SEN UNIVERSITY

Ambiguity[二义性]

- Grammar G is ambiguous if
  - It produces more than one parse tree for some 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

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; more than one parse tree would imply several interpretations

Remove Ambiguity[消除二义性]

- 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

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

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

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

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

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

Now, can only have more ‘+’ on left

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

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
    - build a representation of derivation and pass to next phase
- 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

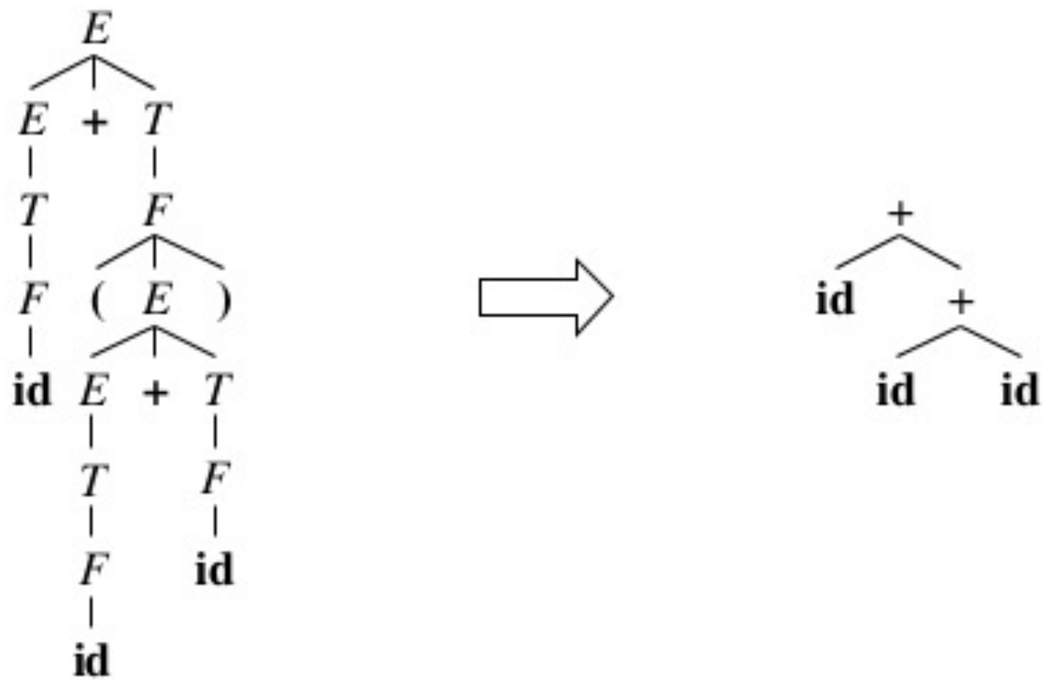
- AST: condensed form of parse tree
  - Operators and keywords do not appear as leaves (e.g., +)
  - Chains of single productions are collapsed (e.g., E -> T)

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

SUN YAT-SEN UNIVERSITY

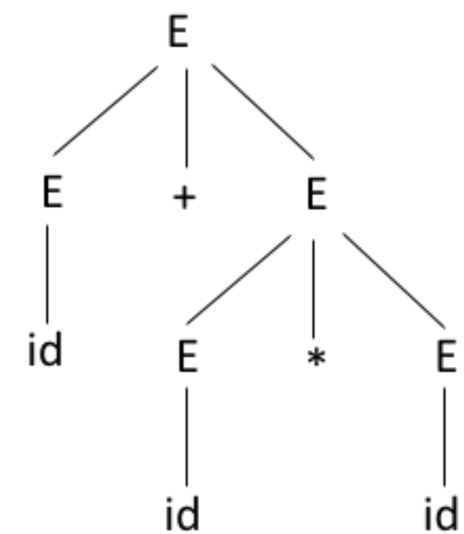
Summary of CFG[小结]

- 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
  - ... 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 Types[分析器类型]

- **Grammar** is used to derive string or construct **parser**
- Most compilers use either **top-down** or **bottom-up** parsers
- Top-down parsing[自顶向下分析]
  - Starts from root and expands into leaves
    - Tries to expand start symbol to input string
    - Finds leftmost derivation[最左推导]
  - In each step
    - Which non-terminal to replace?
    - Which production of the non-terminal to use?
  - Parser code structure closely mimics grammar
    - Amenable to implementation by hand
    - Automated tools exist to convert to code (e.g. ANTLR)



Parser Types (cont.)

- Bottom-up parser[自底向上分析]
  - Starts at leaves and builds up to root
    - Tries to reduce the input string to the start symbol
    - Finds reverse order of the rightmost derivation[最右推导的逆 → 最左归约, 也称为**规范归约**]
  - Parser code structure nothing like grammar
    - Very difficult to implement by hand
    - Automated tools exist to convert to code (e.g. Yacc, Bison)
    - LL ⊂ LR (Bottom-up works for a larger class of grammars)
- Top-down vs. bottom-up[对比]
  - Top-down: easier to understand and implement manually
    - E.g., ANTLR
  - Bottom-up: more powerful, can be implemented automatically
    - E.g., YACC/Bison

Example

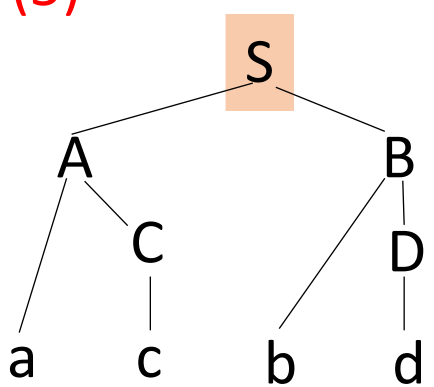
- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

- This language has only one sentence: L(G) = {acbd}

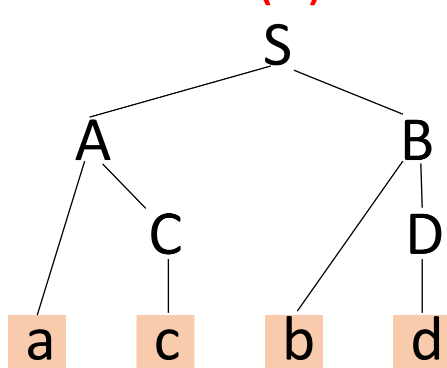
Top-down (leftmost derivation)

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)



Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbD (2)
⇒ acbd (1)



Preview: Bottom-up Parsing[自低向上]

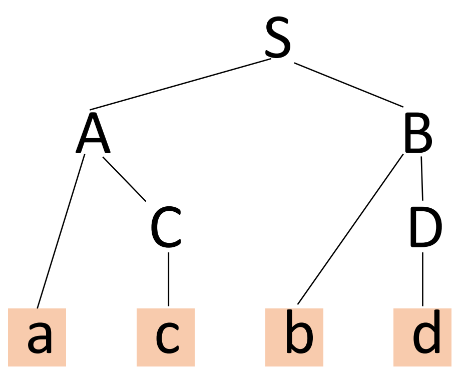
- Consider a CFG grammar G

S→AB A→aC B→bD D→d C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	<u>SUCCESS!</u>

Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbD (2)
⇒ acbd (1)



Top-down Parsers[自顶向下]

- **Recursive descent parser** (RDP, 递归下降分析) with backtracking[回溯]
  - Implemented using recursive calls to functions that implement the expansion of each non-terminal
  - Goes through all possible expansions by **trial-and-error** until match with input; backtracks when mismatch detected
  - Simple to implement, but may take exponential time

- **Predictive parser**[预测分析]
  - Recursive descent parser with prediction (no backtracking)
  - Predict next rule by looking ahead *k* number of symbols
  - Restrictions on the grammar to avoid backtracking
    - Only works for a class of grammars called LL(k)

RDP with Backtracking[回溯]

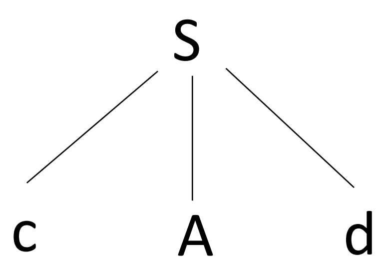
- **Approach**: for a non-terminal in the derivation, productions are tried in some order until
  - A production is found that generates a portion of the input, or
  - No production is found that generates a portion of the input, in which case backtrack to previous non-terminal
- Terminals of the derivation are compared against input
  - Match: advance input, continue parsing
  - Mismatch: backtrack, or fail
- Parsing fails if no derivation generates the entire input

Recursive Decent Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*
  - The input pointer pointing to *c*, the first symbol of *w*
  - *S* has only one production, so we use it to expand *S* and obtain the tree

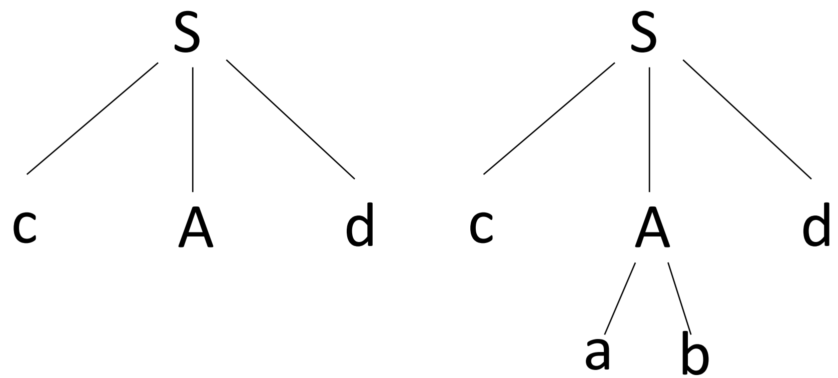


Recursive Decent Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*
  - Next, expand *A* using *A* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

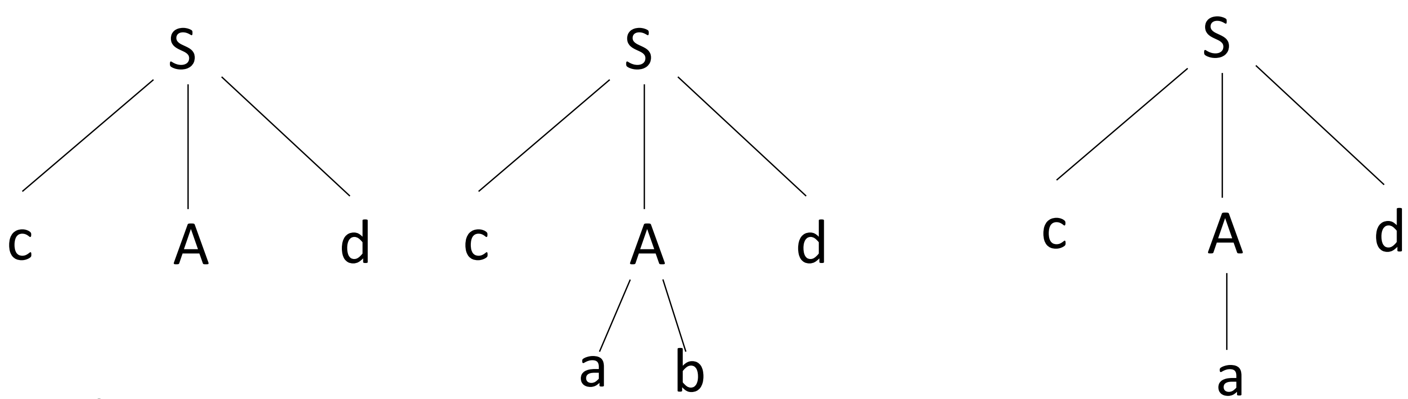


Recursive Decent Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well
  - Leaf a matches the 2nd symbol of *w*, and leaf *d* matches the 3rd
  - We have produced a parse tree for *w*, we halt and announce successful completion of parsing



Left Recursion Problem[左递归问题]

- Recursive descent doesn't work with left recursion
  - Right recursion is OK
- Why is left recursion[左递归] a problem?
  - For left recursive grammar

    $A \rightarrow Ab \mid c$
  - We may repeatedly choose to apply A b

    $A \Rightarrow A\ b \Rightarrow A\ b\ b\ \ldots$
  - Sentence can grow indefinitely w/o consuming input
    - Non-terminal: expand, terminal: match
  - How do you know when to stop recursion and choose *c*?

- Rewrite the grammar so that it is right recursive[改为右递归]
  - Which expresses the same language

SUN YAT-SEN UNIVERSITY 中山大學

NSCC GZ

Left Recursion[左递归]

- A grammar is left recursive if
  - It has a nonterminal *A* such that there is a derivation $A \Rightarrow^+ A\alpha$ for some string α
- Recursion types [直接和间接左递归]
  - **Immediate left recursion**, where there is a production $A \rightarrow A\alpha$
  - Non-immediate: left recursion involving derivation of 2+ steps

    $S \rightarrow Aa \mid b$

    $A \rightarrow Sd \mid \varepsilon$
  - $S \Rightarrow Aa \Rightarrow Sda$

- Algorithm to systematically eliminates left recursion from a grammar

SUN YAT-SEN UNIVERSITY

Remove Left Recursion[消除左递归]

- Grammar: $A \rightarrow A\alpha \mid \beta$ ($\alpha \neq \beta$, β doesn't start with A)

  $A \Rightarrow A\alpha$

  $\Rightarrow A\alpha\alpha$

  …

  $\Rightarrow A\alpha\ldots\alpha\alpha$

  $\Rightarrow \beta\alpha\ldots\alpha\alpha$

  $r = \beta\alpha^*$

- Rewrite to:

  $A \rightarrow \beta A'$ // begins with β (A' is a new non-terminal)

  $A' \rightarrow \alpha A' \mid \varepsilon$ // A' is to produce a sequence of α

  $\Rightarrow \alpha\alpha A'$

  …

  $\Rightarrow \alpha\ldots\alpha A' \Rightarrow \alpha\ldots\alpha$

Remove Left Recursion (cont.)

- Grammar:

  A → Aα | β

  to

  A → βA'

  A' → αA' | ε

- E → E + T | T (α = + T, β = T) ⟹ E → TE'

  E' → +TE' | ε

- T → T * F | F (α = * F, β = F) ⟹ T → FT'

  T' → *FT' | ε

- F → (E) | id ⟹ F → (E) | id

Summary of Recursive Descent[小结]

- **Recursive descent** is a simple and general parsing strategy
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
  - L(Recursive_descent) ≡ L(CFG) ≡ CFL
- However it is not popular because of **backtracking**
  - Backtracking requires re-parsing the same string
  - Which is inefficient (can take exponential time)
  - Also undoing semantic actions may be difficult
    - E.g. removing already added nodes in parse tree