编译原理
3. 语法分析-CFG及其解析

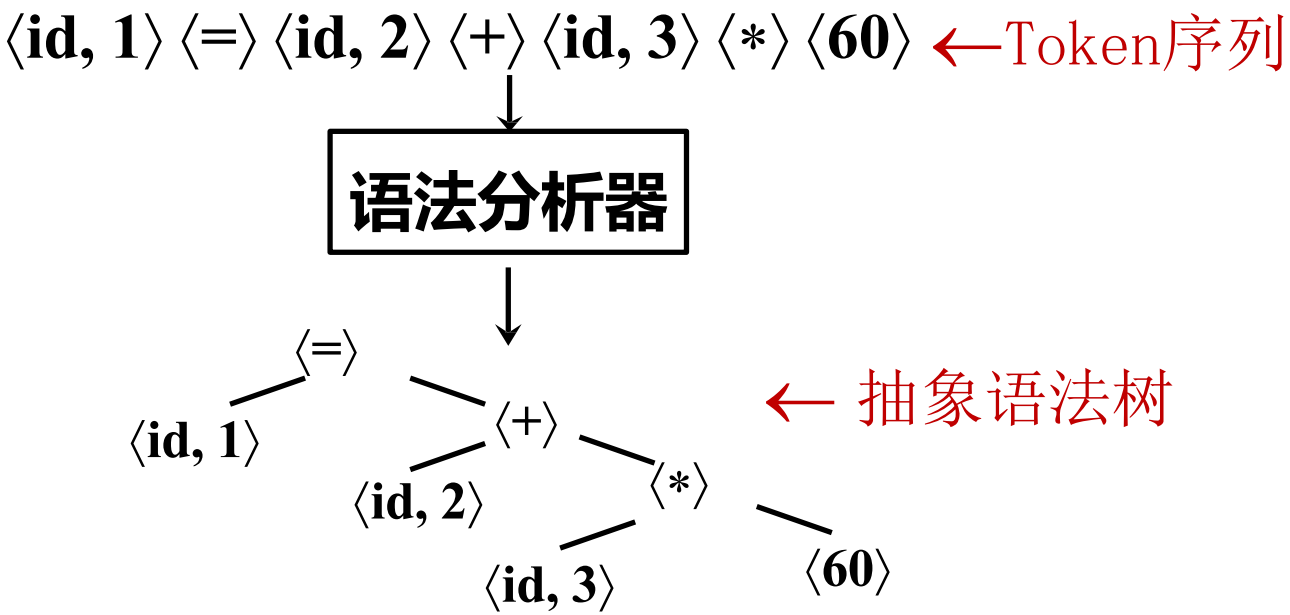
rainoftime.github.io
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
2. Lexical Analysis
3. **Parsing**
4. Abstract Syntax
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

回顾: 语法分析器的作用

- **基本作用:** 从词法分析器获得Token序列，确认该序列是否可以由语言的文法生成
  - 对于语法错误的程序，报告错误信息
  - 对于语法正确的程序，生成语法分析树 (简称语法树)
    - 通常产生的是抽象语法树（ AST ）



语法分析器的实现

- **Option 1: By-hand (recursive descent)**
  - Clang, gcc(Since3.4)
  - Libraries can make this easier (e.g.,parser combinators–parsec)

- **Option 2: Use a parser generator**
  - Much easier to get right (“Specification is the implementation”)
  - Gcc (Before 3.4), Glasgow Haskell Compiler, OCaml
  - Parser generator: Yacc, Bison, ANTLR, menhir

本讲内容

1 上下文无关文法

2 语法分析概述

1. 上下文无关文法

- CFG简介
- 推导和归约
- RE和CFG

问题: 如何形式化定义编程语言的语法？

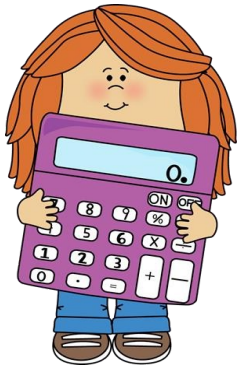
例: 简单计算器程序

$1 + (2 - 3)$ 合法

$1 + 2 - 3 +$ 非法

$1\ 2 + - 3$ 非法

$1 + 2 - a$ 非法



语法分析的目的是教会计算机判断输入合法性

例: C语言程序

- 以下程序没有词法错误，但是有多个语法错误

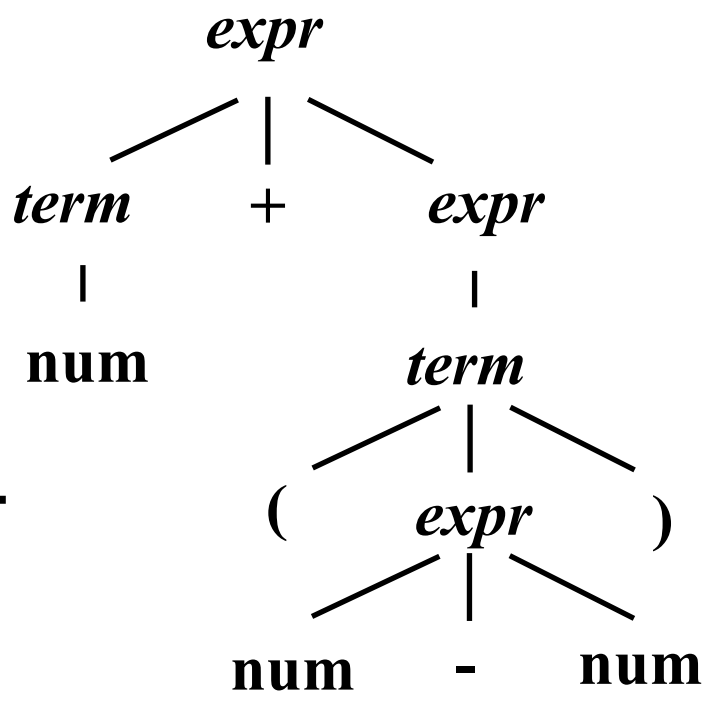
```
int foo(int x){
  int y;
  if ((x > 0)
    y = 1;
  else
    y = 0
  return y;
}
```

```
foo.c:4:9: error: expected ')'
        y = 1;
        ^
foo.c:3:8: note: to match this '('
    if ((x > 0)
       ^
foo.c:6:14: error: expected ';' after expression
        y = 0
             ^
             ;
2 errors generated._
```

如何判定输入合法性 & 生成语法树

- **首先规定好合法的基本单元—词法分析**
  - Token：如由0-9组成的数字(num)和符号+、-、(、)
- **其次要理解算术表达式的构成**
  - 大表达式可拆为子表达式
  - 拆解过程是递归的，直至看到基本单元

语法树
Parse Tree



如何构造编程语言的语法分析器

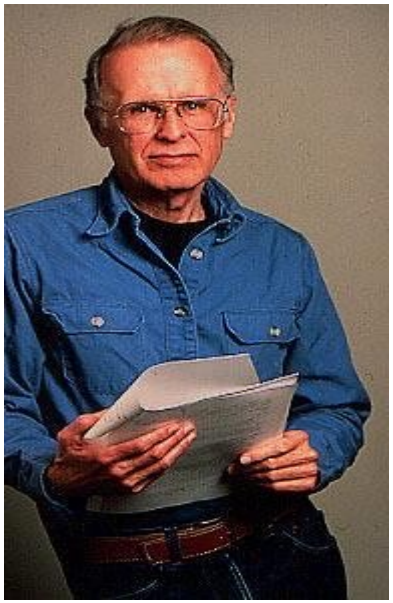
- **Specifying the syntax of a programming language with Context-Free Grammars (CFG)**

  如何描述编程语言的语法结构？

- **Build the parser based on the CFG:**
  - Top-Down Parsing (自顶向下分析)
  - Bottom-Up Parsing (自底向上分析)
- **More about parsing:**
  - Automatic parser generation
  - Error Recovery

上下文无关文法 (Context-Free Grammar)

- **John Backus(1977图灵奖)提出**
  - **提出了多种高级编程语言**
    - Speedcoding -> FORTRAN -
      >ALGOL 58 -> ALGOL 60
  - **提出了编译技术的理论基础**
    - 巴科斯范式（Backus–Naur Form）
    - **上下文无关文法**
  - **对计算机科学影响巨大**
    - 诞生了许多理论研究成果
    - 现代编译器还保留了FORTRAN I的大概架构



弗吉尼亚大学化学专业，哥伦比亚大学数学专业，曾服务于阿波罗登月计划

上下文无关文法: 描述语言的语法结构

$$G = (T, N, P, S)$$

T：终结符集合 (Terminals)

N：非终结符集合 (Non-terminals)

P：产生式集合 (Productions) $A \rightarrow \alpha, A \in N, \alpha \in (T \cup N)^*$

S：开始符号 (Start symbol): $S \in N$

上下文无关文法: 描述语言的语法结构

$$G = (T, N, P, S)$$

T：终结符集合 (Terminals): 组成串的基本符号(*token*)

- 例: T = { num, +, -, (,)}

上下文无关文法: 描述语言的语法结构

$$G = (T, N, P, S)$$

T：**终结符集合 (Terminals)**

N：**非终结符集合 (Nonterminal)**: 表示串的集合的语法变量

- 在程序语言中通常对应于某个程序构造
  - 例: N = {expr, term, stmt}

$T \cap N = \Phi$

$T \cup N$：**文法符号集**

上下文无关文法: 描述语言的语法结构

$$G = (T, N, P, S)$$

T：**终结符集合** (Terminals)

N：**非终结符集合** (Nonterminals)

P：**产生式$A \to \alpha$ 集合** (Productions): $A \in N, \alpha \in (T \cup N)^*$

- 头(左)部A是一个非终结符号，右部α是一个符号串
- 描述将终结符和非终结符组合成串的方法

例: $E \to E + E, E \to E * E, E \to (E), E \to \mathbf{id}$

上下文无关文法: 描述语言的语法结构

$$G = (T, N, P, S)$$

T：**终结符集合** (Terminals)

N：**非终结符集合** (Non-terminals)

P：**产生式集合** (Productions) $A \rightarrow \alpha, A \in N, \alpha \in (T \cup N)^*$

S：**开始符号** (Start symbol): $S \in N$ 唯一一个开始符号

- 某个被指定的非终结符号
- 它对应的串的集合就是文法的语言

例: 上下文无关文法

$expr \rightarrow expr\ op\ expr$

$expr \rightarrow (expr)$

$expr \rightarrow - expr$

$expr \rightarrow \mathbf{id}$

$op \rightarrow + \mid - \mid * \mid /$

- 非终结符集合$N = \{expr,\ op\}$
- 终结符集合$T = \{+,-,*,/,\ (,\ ),\mathrm{id}\}$
- $expr$是开始符号

operator运算符　operand 操作数

例: 上下文无关文法

1. S → S; S
2. S → id := E
3. S → print (L)
4. E → id
5. E → num
6. E → E + E
7. E → (S , E)
8. L → E
9. L → L , E

Grammar 3.1. A syntax for straight-line programs. (From Tiger Book)

- T= {id, print, num, +, (,), }, N = {S, E, L}
- Start symbol: S

Strings/sentences in the language of this CFG:

- id := num
- id := num + num
- print (num)
- id := id + (id := num + num, id)
- ……

上下文无关文法 – EOF Marker

- $: end of file (EOF)
- To indicate that $ must come after a complete S-phrase
  - Add a new start symbol S' and a new production S' → S$

S -> E$
E -> E + T
E -> E – T
E -> T

T → T * F
T → T / F
T → F

F -> id
F -> num
F -> (E)

Grammar 3.10. (Tiger book)

产生式的缩写

- 对一组有**相同左部**的α产生式

$$\alpha \rightarrow \beta_1 , \alpha \rightarrow \beta_2 , \dots , \alpha \rightarrow \beta_n$$

- 可以简记为

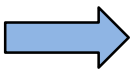
$$\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

- 读作：α**定义为**β_1，或者β_2，…，或者β_n

β_1，β_2，…，β_n称为α的**候选式**(*Candidate*)

➢ 例

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



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

1. 上下文无关文法

- CFG简介
- **推导和归约**
- RE和CFG

问题：给定文法，如何判定输入串属于文法规定的语言?

推导 (Derivations) 和归约(Reductions)

给定文法$G=(T,N,P,S)$

- 直接推导: 把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替
  - 如果$A \to \gamma \in P$，且$\alpha, \beta \in (T \cup N)^*$，称串$\alpha A \beta$直接推导出$\alpha\gamma\beta$，并记作$\alpha A \beta \Rightarrow \alpha\gamma\beta$。

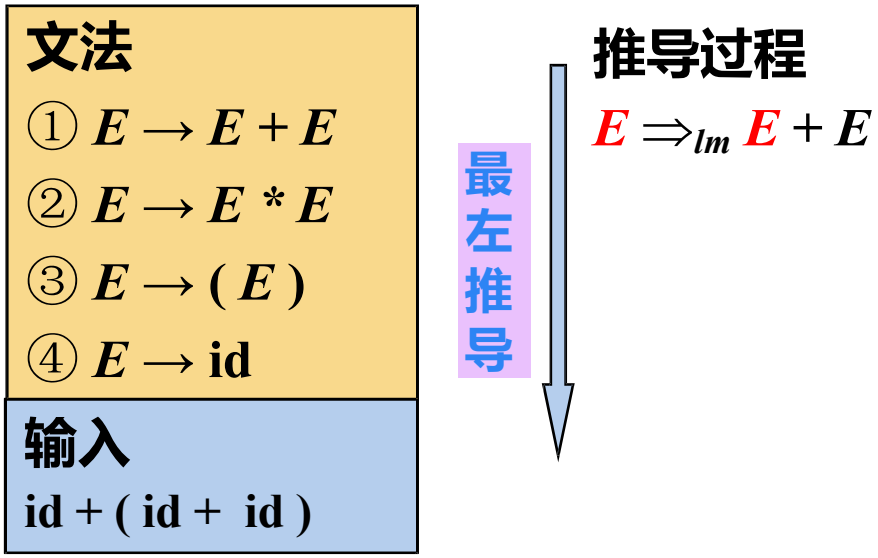
- 直接规约: 如果$\alpha A \beta \Rightarrow \alpha\gamma\beta$，则$\alpha\gamma\beta$直接规约到$\alpha A \beta$

推导 (Derivations) 和归约(Reductions)

- 多步推导: 如果$\alpha_0 \Rightarrow \alpha_1$，$\alpha_1 \Rightarrow \alpha_2$，$\alpha_2 \Rightarrow \alpha_3$，…，$\alpha_{n-1} \Rightarrow \alpha_n$，则可以记作$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$，称符号串 α_0经过n步推导出α_n，可简记为$\alpha_0 \Rightarrow^n \alpha_n$
  - $\Rightarrow^+$表示" 经过正数步推导"
  - $\Rightarrow^*$表示" 经过若干（可以是0）步推导"

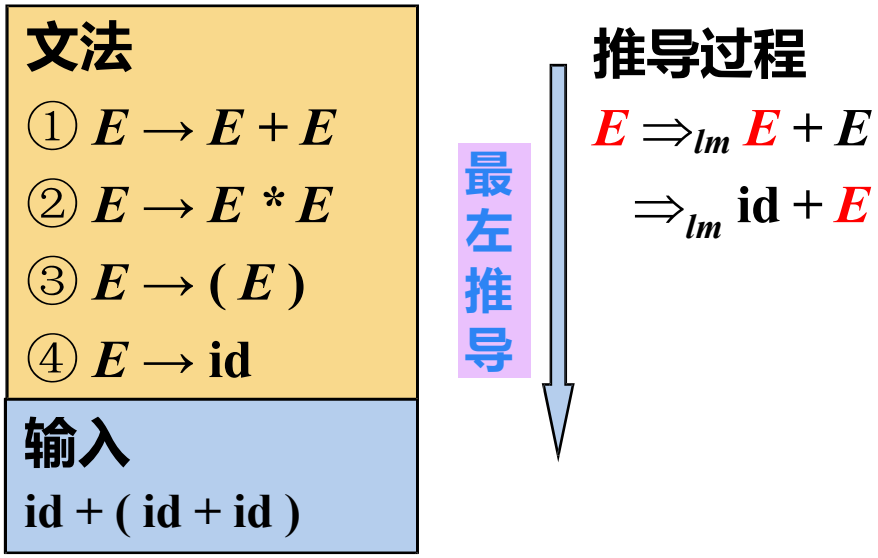
最左推导 (Left-most Derivation)

- **最左推导: 每步代换最左边的非终结符**



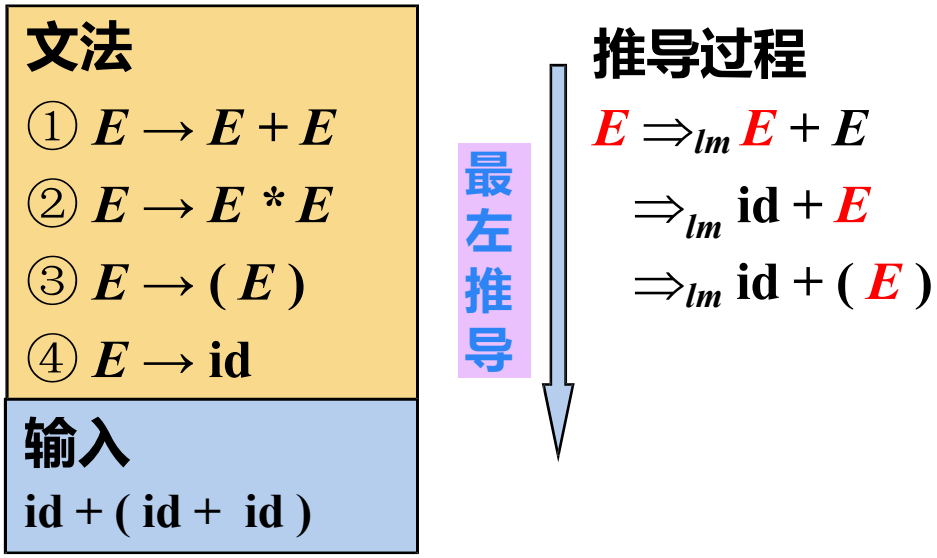
最左推导 (Left-most Derivation)

- **最左推导: 每步代换最左边的非终结符**



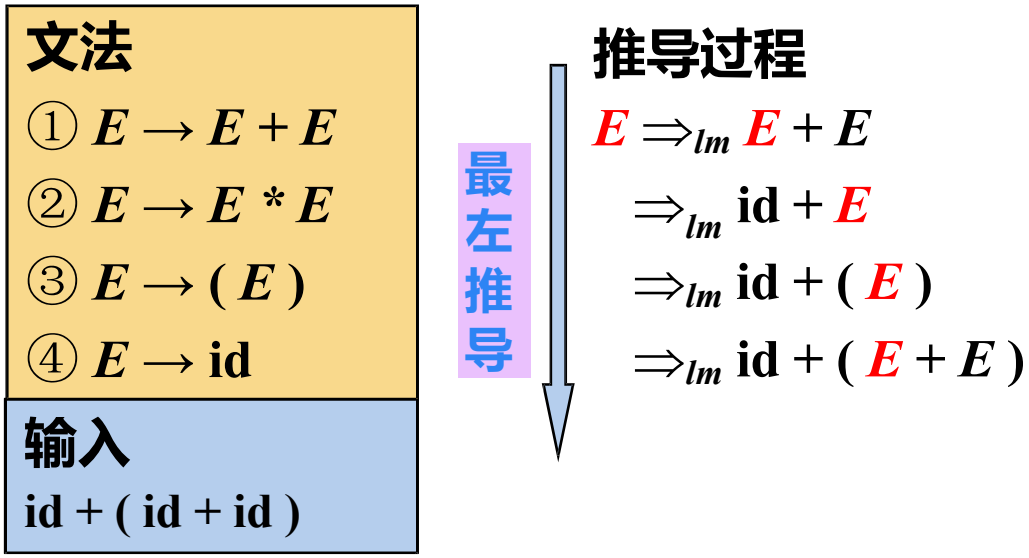
最左推导 (Left-most Derivation)

- **最左推导: 每步代换最左边的非终结符**



最左推导 (Left-most Derivation)

- **最左推导: 每步代换最左边的非终结符**



最左推导 (Left-most Derivation)

- **最左推导: 每步代换最左边的非终结符**

文法

① $E \rightarrow E + E$

② $E \rightarrow E * E$

③ $E \rightarrow (E)$

④ $E \rightarrow \mathrm{id}$

输入

id + (id + id)

最左推导

推导过程

$E \Rightarrow_{lm} E + E$

$\Rightarrow_{lm} \mathrm{id} + E$

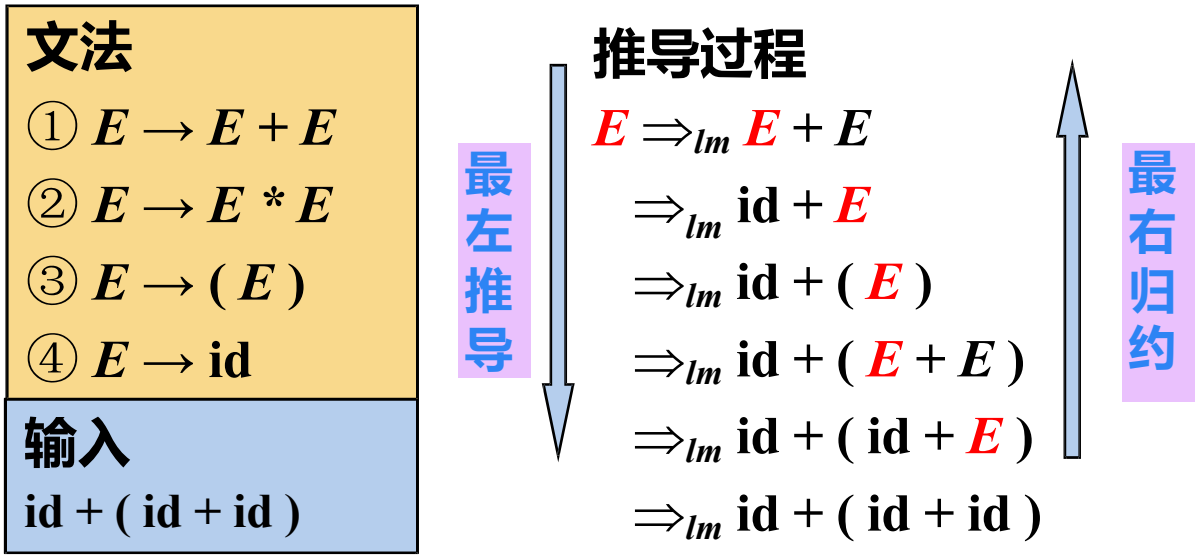
$\Rightarrow_{lm} \mathrm{id} + (E)$

$\Rightarrow_{lm} \mathrm{id} + (E + E)$

$\Rightarrow_{lm} \mathrm{id} + (\mathrm{id} + E)$

最左推导 (Left-most Derivation)

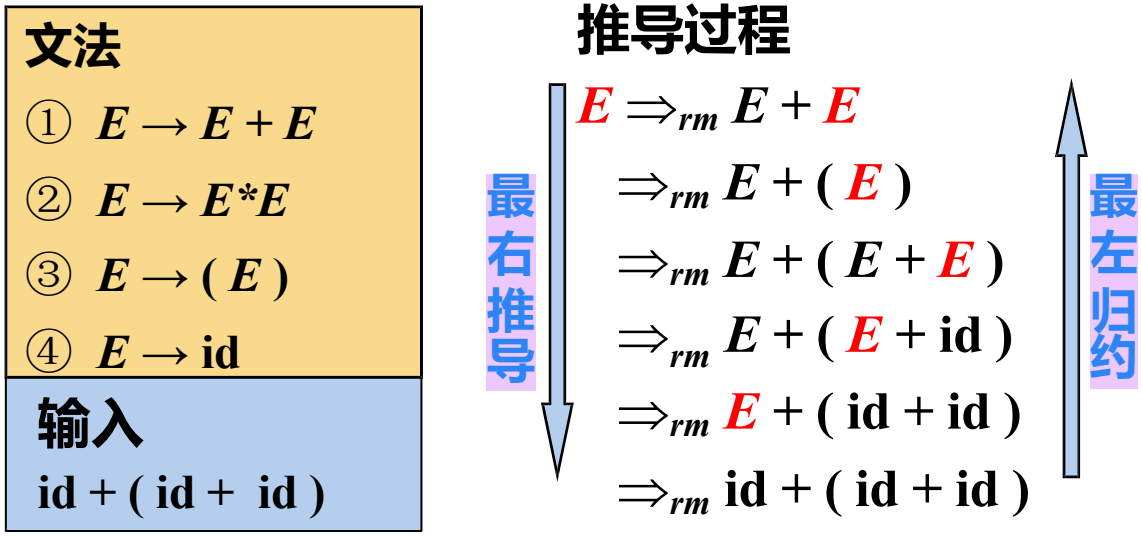
- **最左推导: 每步代换最左边的非终结符**



  - 如果 $S \Rightarrow^*_{lm} \alpha_0$, 则称α是当前文法的最左句型(*left-sentential form*)
  - 在自顶向下的分析中，总是采用**最左推导**的方式

最右推导 (Right-most Derivation)

- **最右推导: 每步代换最右边的非终结符**



  - 在自底向上的分析中，总是采用**最左归约**的方式

- **练习: 尝试理解以上推导过程**

句型、句子和语言

- **句型** (Sentential form): **对开始符号为S的文法G, 如果$S \Rightarrow^* \alpha$，$\alpha \in (T \cup N)^*$，则称α是G的一个句型**
  - 句型中既可以包含终结符，又可以包含非终结符，也可能是空串

- **句子**(Sentence): **如果$S \Rightarrow^* w$，$w \in T^*$则称w是G的一个句子**
  - 句子是不含非终结符的句型
  - 仅含终结符号的句型是一个句子

- **语言**: **由文法G推导出的所有句子构成的集合，记为$L(G)$。**

$$L(G) = \{w \mid S \Rightarrow^* w, w \in T^*\}$$

语言$L(G)$是由文法G产生的所有句子的集合

例: 文法定义的句型和句子

- **考虑文法**: $E \to E + E \mid E * E \mid (E) \mid -E \mid \text{id}$

- **存在以下推导序列**

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

1. –(id + id)是文法的句子(Sentence)
2. –(id + E)是文法的句型 (Sentential form)

输入串的判定

问题：给定文法，如何判定输入串属于文法规定的语言?

从开始符号能推导出该词串

- 句子的推导（派生） -从生成语言的角度
- 句子的归约 -从识别语言的角度

} 根据文法规则

从词串能归约出开始符号

文法：

① <句子> → <名词短语> <动词短语>

② <名词短语> → <形容词> <名词短语>

③ <名词短语> → <名词>

④ <动词短语> → <动词> <名词短语>

⑤ <形容词> → *little*

⑥ <名词> → *boy*

⑦ <名词> → *apple*

⑧ <动词> → *eat*

推导 ↓

<句子> ⇒ <名词短语> <动词短语>

⇒ <形容词> <名词短语> <动词短语>

⇒ *little* <名词短语> <动词短语>

⇒ *little* <名词> <动词短语>

⇒ *little boy* <动词短语>

⇒ *little boy* <动词><名词短语>

⇒ *little boy eats* <名词短语>

⇒ *little boy eats* <名词>

⇒ *little boy eats apple*

↑ 归约

思考题

- **上下文无关是什么意思？**

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

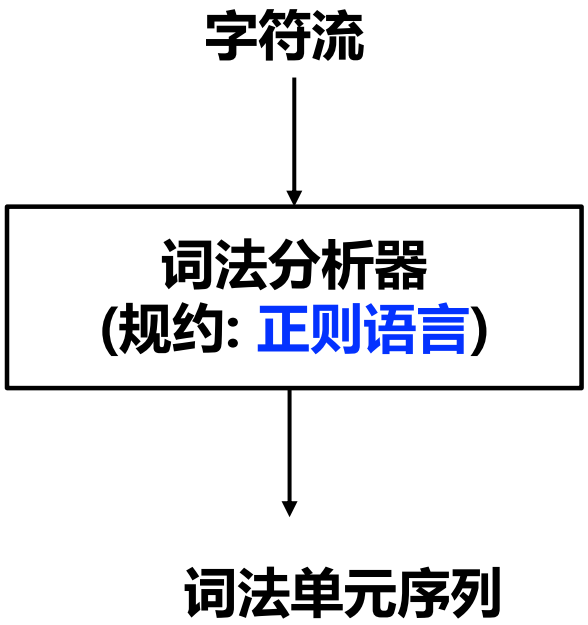
在文法推导的每一步，符号串γ仅依据A的产生式推导，无需依赖A的上下文α和β

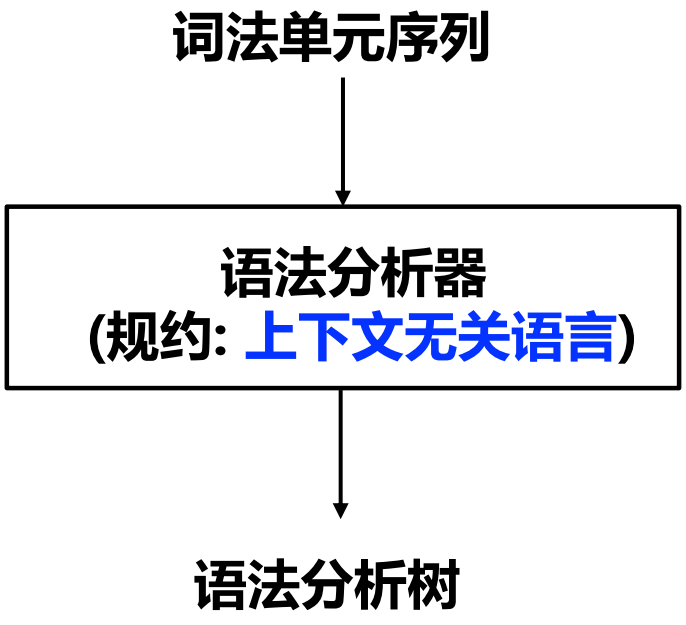
1. 上下文无关文法

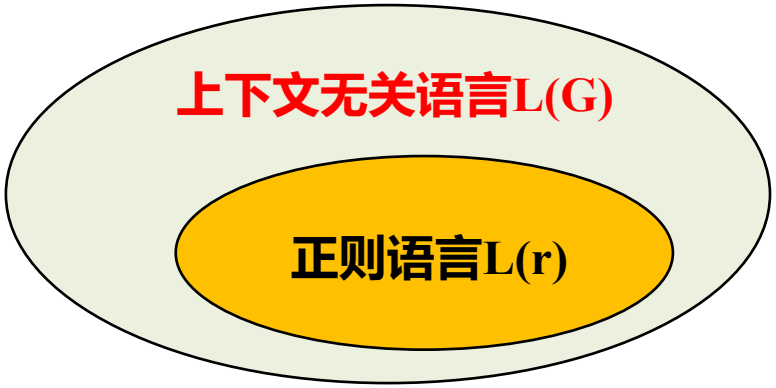
- CFG简介
- 推导和归约
- RE和CFG

问题：为什么词法和语法分析用不同形式语言?

语法分析和词法分析的比较







回顾: 形式语言

- **语言：字母表Σ上的一个串集**
  - 例: {ε, 0, 00, 000, …}, {ε}, ∅
  - 句子：属于语言的串
- **语言的运算**

运算	定义和表示
L和M的并	$L \cup M = \{s \mid s$属于L或者s属于$M\}$
L和M的连接	$LM = \{st \mid s$属于L且t属于$M\}$
L的幂	$\begin{cases} L^0 = \{\varepsilon\} \\ L^n = L^{n-1}L, \ n \geqslant 1 \end{cases}$
L的Kleene闭包	$L^* = \cup_{i=0}^{\infty} L^i$
L的正闭包	$L^+ = \cup_{i=1}^{\infty} L^i$

回顾: 正则语言--由正则表达式刻画

- **正则表达式 *r*定义正则语言，记为*L*(*r*)**

1. ε是一个RE，$L(\varepsilon) = \{\varepsilon\}$
2. 如果 $a \in \sum$，则a是一个RE，$L(a) = \{a\}$
3. 假设 r和 s都是 RE，分别表示语言 $L(r)$和$L(s)$
   - $r|s$ 是一个RE，$L(r|s) = L(r) \cup L(s)$
   - rs 是一个RE，$L(rs) = L(r)\,L(s)$
   - r^* 是一个RE，$L(r^*) = (L(r))^*$ Kleene闭包
   - (r) 是一个RE，$L((r)) = L(r)$

回顾: 上下文无关语言

- **上下文无关文法(CFG)的定义**

$$G = (T, N, P, S)$$

T：终结符集合

N：非终结符集合

P：产生式(Production) $A \rightarrow \alpha$ 集合

– 约束: 左部A是一个非终结符号，右部α是符号串

S：开始符号 (Start symbol): 某个非终结符号

上下文无关语言$L(G)$: 由文法G产生的所有句子的集合

正则语言的形式文法刻画!

- **正则文法的定义**

$$G = (T, N, P, S)$$

T：终结符集合

N：非终结符集合

P：产生式(Production) $\alpha \rightarrow \beta$ 集合

- 右线性文法: $\alpha \rightarrow \beta$形如$A \rightarrow aB$ 或$A \rightarrow a$ 其中$A, B \in N, a \in \mathrm{T} \cup \{\varepsilon\}$
- 左线性文法: $\alpha \rightarrow \beta$形如$A \rightarrow Ba$或 $A \rightarrow a$

S：开始符号 (start symbol): 某个非终结符号

正则语言: 右线性文法/左线性文法产生的所有句子的集合

为什么词法分析用正则表达式，不用正则文法？

- **正则表达式描述简洁(刻画Token)， 且易于理解**

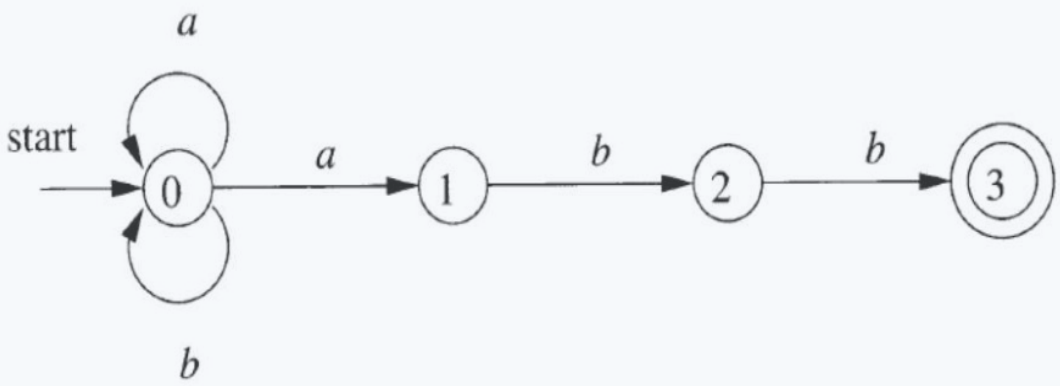
$A_0 \rightarrow aA_0 \mid aA_1 \mid bA_0$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

正则文法定义



NFA状态转换图

(a|b)* abb

正则表达式定义

正则语言 vs 上下文无关语言

- **从文法角度，对产生式$\alpha \rightarrow \beta$形式的限制不同**
  - **上下文无关文法:**

    左部α是一个非终结符号，右部是一个**符号串**
  - **正则文法:**

    右线性$A \rightarrow aB$ 或$A \rightarrow a$, 左线性$A \rightarrow Ba$或 $A \rightarrow a$

每个正则语言都是一个上下文无关语言，反之不成立

正则语言 vs 上下文无关语言

- **从实用角度，正则语言表达能力有限，难以刻画编程语言的语法**
  - 如：不能用于描述配对或嵌套的结构
    - 例：配对括号串的集合，如不能表达 $(^n)^n$, $n \geq 1$
    - 原因：有穷自动机无法记录访问同一状态的次数

```
int foo(int x){
  int y;
  if ((x > 0)
    y = 1;
  else
    y = 0
  return y;
}
```

```
foo.c:4:9: error: expected ')'
        y = 1;
        ^
foo.c:3:8: note: to match this '('
    if ((x > 0)
       ^
foo.c:6:14: error: expected ';' after expression
        y = 0
             ^
             ;
2 errors generated.
```

分离词法分析和语法分析

- **为什么用正则语言定义词法**
  - 词法规则非常简单，不必用上下文无关文法
  - 对于Token，正则表达式描述简洁且易于理解
  - 从正则表达式构造出的词法分析器效率高

- **分离词法分析和语法分析的好处**
  - 简化设计、提升性能
  - 编译器的可移植性加强
  - 便于编译器前端的模块划分

引申: 形式文法的分类

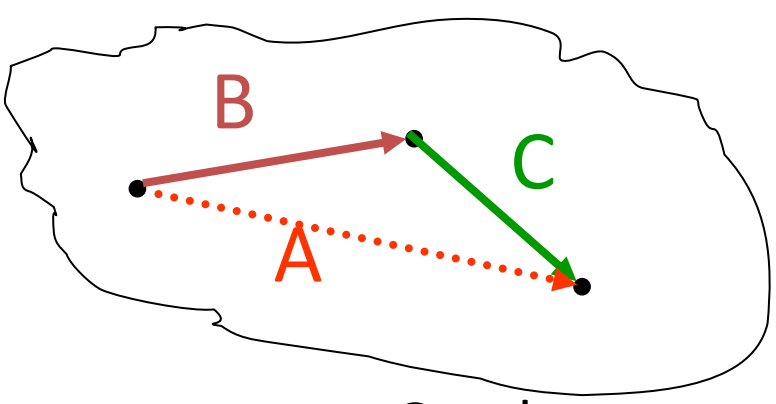
Chomsky在1956创立了形式语言学，并将形式语言的文法分为四类：

文法 $G = (T, N, P, S)$

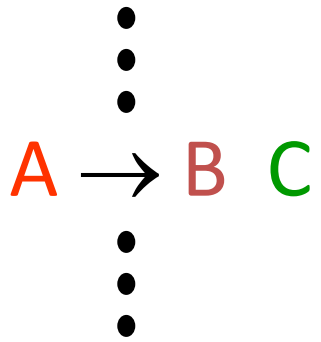
- **0型文法(短语结构文法): 递归可枚举语言**
  - $\alpha \to \beta$，$\alpha, \beta \in (N \cup T)^*, |\alpha| \geq 1$
- **1型文法(上下文有关文法)**
  - $|\alpha| \leq |\beta|$，但$S \to \varepsilon$可以例外
- **2型文法(上下文无关文法)**
  - $A \to \beta$，$A \in N, \beta \in (N \cup T)^*$
- **3型文法(正则文法)**
  - $A \to aB$(左线性$A \to Ba$)，或$A \to a$，$A, B \in N, a \in T$

引申: CFG只能表达“语法”吗？

- **CFL-Reachability (结合CFG和图可达性)**
  - $\boldsymbol{G}$: Graph (N nodes, E edges); $\boldsymbol{L}$: A context-free language
  - L-path from s to t iff $s \xrightarrow{\alpha}{}^{*} t, \alpha \in L$
  - Running time: $O(N^3) \rightarrow O(N^3/\log N)$ [POPL'08]
- **CFL-Reachability用于分析程序语义**
  - 类型推导、指针分析、数据流分析、etc.
  - E.g, LLVM内部的cfl-aa



Graph



Grammar

2. 语法分析概述

- CFG的Parse Tree
- 设计编程语言的文法

回顾: 判定输入串属于CFG规定的语言

从开始符号能推导出该词串

- 句子的推导（派生） -从生成语言的角度
- 句子的归约 -从识别语言的角度

} 根据文法规则

从词串能归约出开始符号

文法：

① <句子> → <名词短语> <动词短语>

② <名词短语> → <形容词> <名词短语>

③ <名词短语> → <名词>

④ <动词短语> → <动词> <名词短语>

⑤ <形容词> → *little*

⑥ <名词> → *boy*

⑦ <名词> → *apple*

⑧ <动词> → *eat*

推导 ↓

<句子> ⇒ <名词短语> <动词短语>

⇒ <形容词> <名词短语> <动词短语>

⇒ *little* <名词短语> <动词短语>

⇒ *little* <名词> <动词短语>

⇒ *little boy* <动词短语>

⇒ *little boy* <动词><名词短语>

⇒ *little boy eats* <名词短语>

⇒ *little boy eats* <名词>

⇒ *little boy eats apple*

↑ 归约

分析树(Parse Tree): 推导的图形化表示

- **分析树具有下面性质**
  - **根节点**为文法的**初始符号**
  - 每个**叶子节点**是一个**终结符**
  - 每个**内部节点**是一个**非终结符**
  - 每一个父节点和他的子节点构成一条**产生式**

推导过程：$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\text{id}+E) \Rightarrow -(\text{id}+\text{id})$

文法:

① $E \rightarrow E+E$

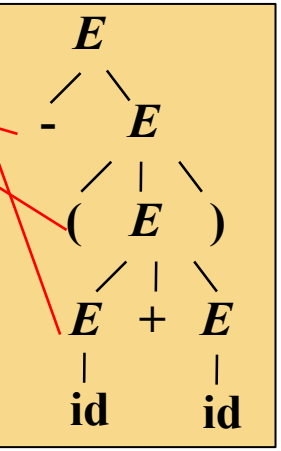
② $E \rightarrow E*E$

③ $E \rightarrow -E$

④ $E \rightarrow (E)$

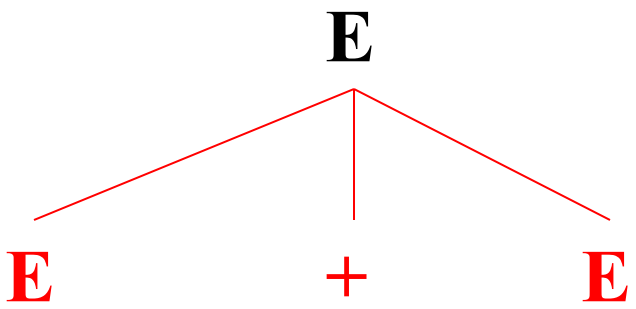
⑤ $E \rightarrow \text{id}$

分析树：



例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** $id * id + id$



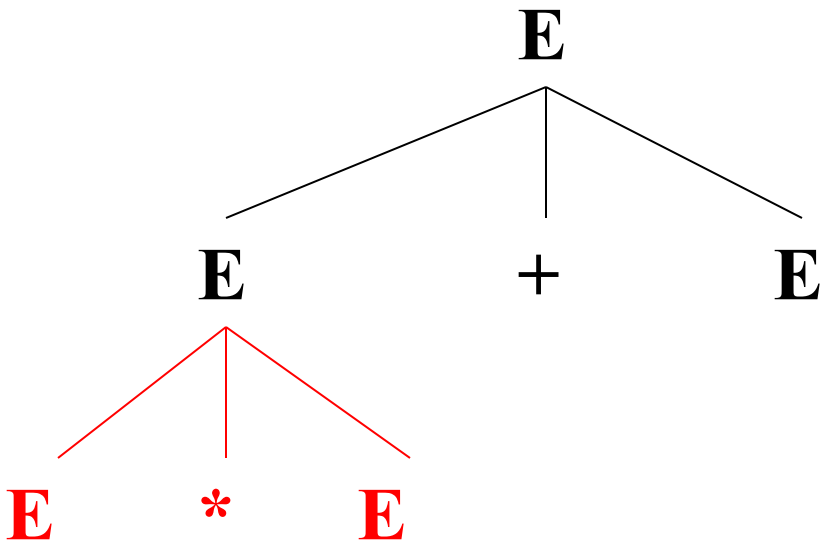
E

$\rightarrow$ E+E

例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** id * id + id

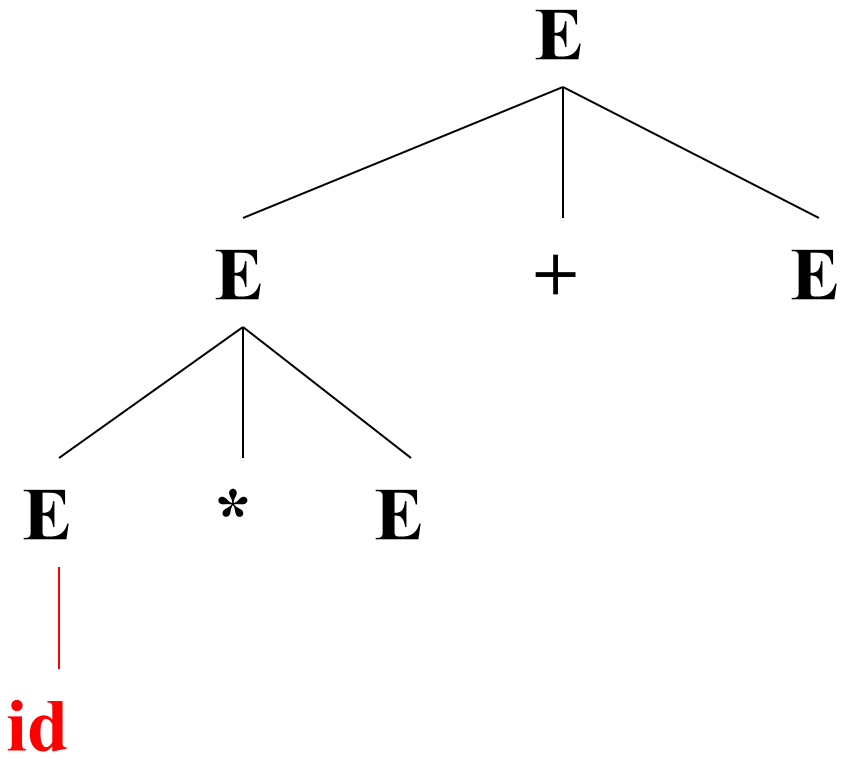
$$E$$
$$\rightarrow E+E$$
$$\rightarrow E*E+E$$



例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** id * id + id

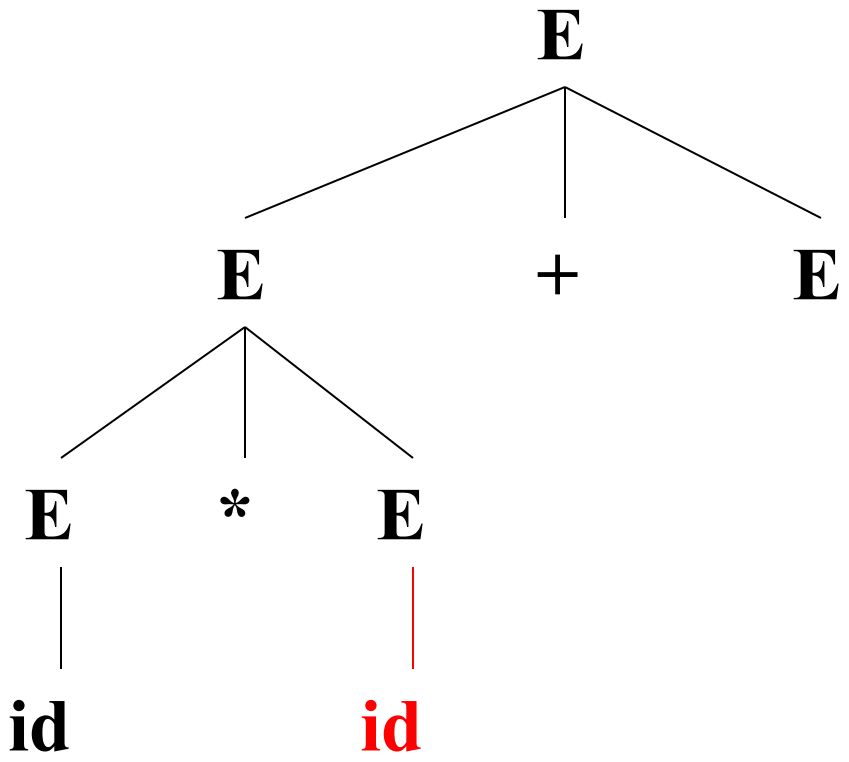
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E
\end{aligned}
$$



例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** id * id + id

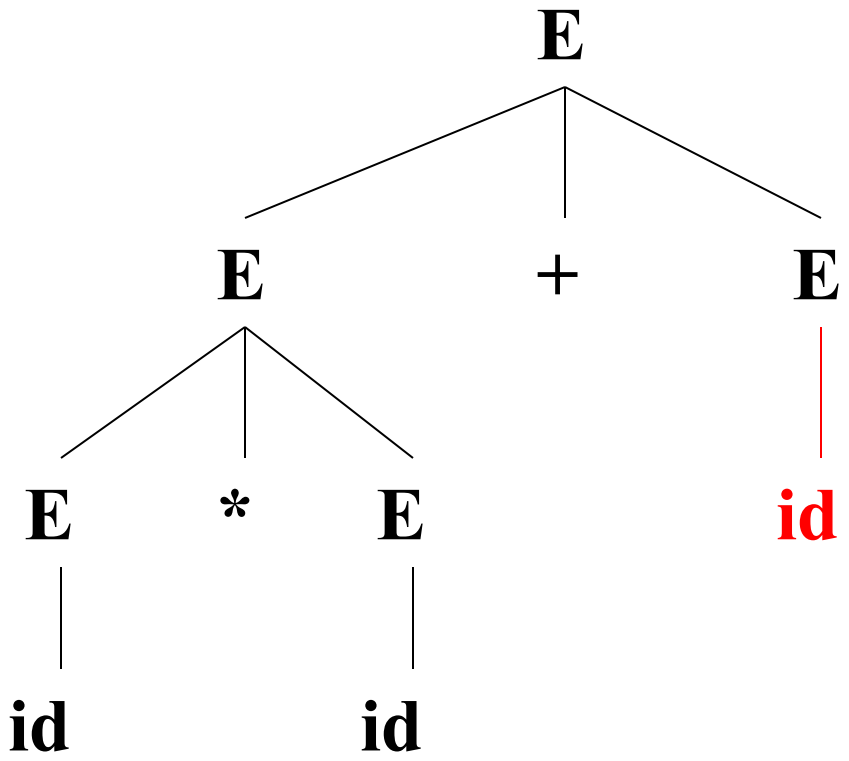
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E
\end{aligned}
$$



例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** id * id + id

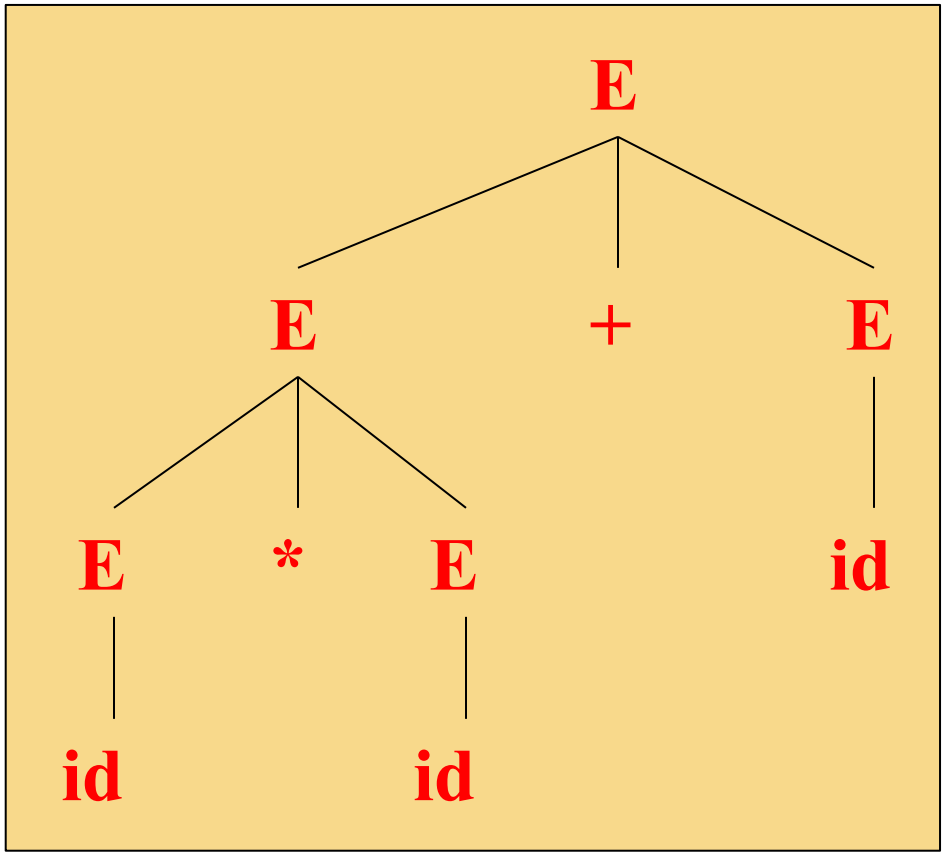
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



例: Parse Tree构造

- **文法** $E \rightarrow E+E \mid E*E \mid (E) \mid id$
- **串** id * id + id

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



练习: Parse Tree构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

expr

expr

语法分析作为搜索问题

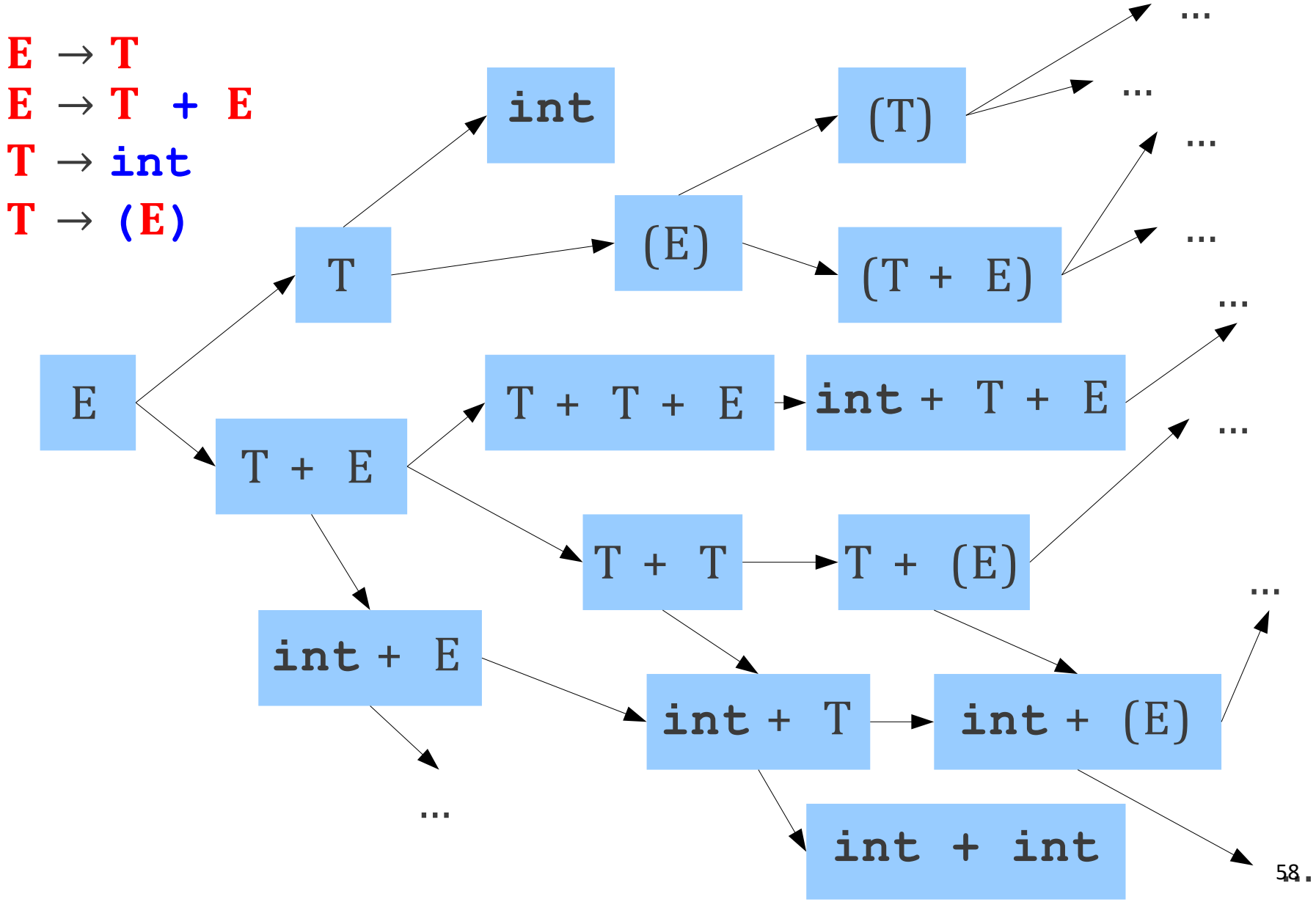
- **语法分析的核心问题: 对于一个终结符号串x**
  - 设法从S推导出x
  - 或者反过来，设法将x归约为S

➢ 句子的推导 -从生成语言的角度

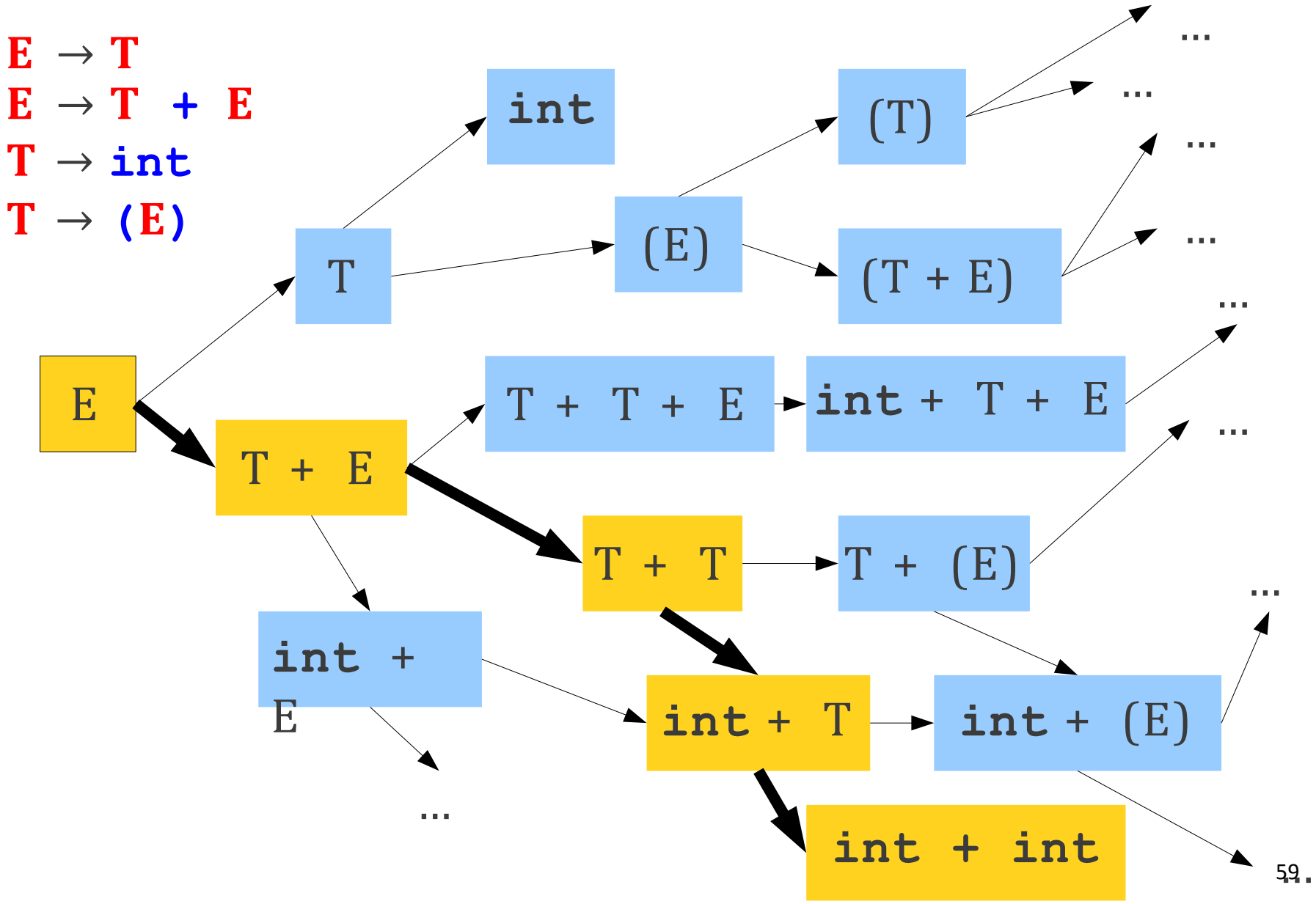
➢ 句子的归约 -从识别语言的角度

} 根据**文法规则**

例: 语法分析作为搜索问题



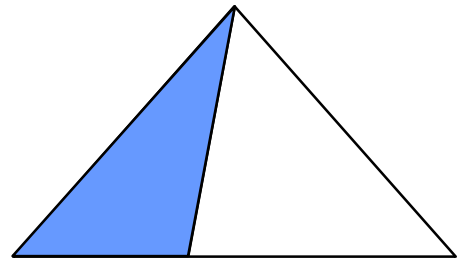
例: 语法分析作为搜索问题



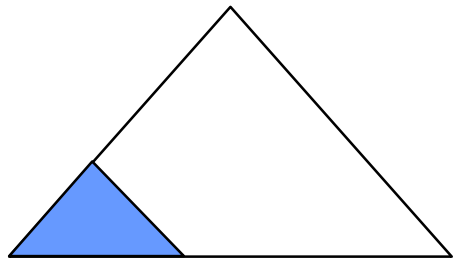
语法分析作为搜索问题

对于一个终结符号串x, 从S推导出x或者将x归约为S

- **自顶向下(Top-down)分析**
  - 从开始符号S出发，尝试根据产生式规则推导 (derive)出x
  - Parse Tree的构造方法：从根部开始
- **自底向上(Bottom-up) 分析**
  - 尝试根据产生式规则归约(reduce)到文法的开始符号S
  - Parse Tree的构造方法: 从叶子开始



Top-down



Bottom-up

语法分析作为搜索问题

对于一个终结符号串x, 从S推导出x或者将x归约为S

- **自顶向下 vs. 自底向上**
- **为了高效搜索**
  - 控制搜索空间: 文法产生式的限制?
  - 搜索空间影响搜索方式: 如正则文法 vs. CFG

语法分析作为搜索问题

对于一个终结符号串*x*, 从*S*推导出*x*或者将*x*归约为*S*

- **If there are no restrictions on the form of grammar used, parsing CFL requires $O(n^3)$ time**
  - E.g., Cocke-Younger-Kasami's algorithm (CYK)

- **Subsets of CFLs typically require $O(n)$ time**
  - Predictive parsing using LL(1) grammars
  - Shift-Reduce parsing using LR(1) grammars

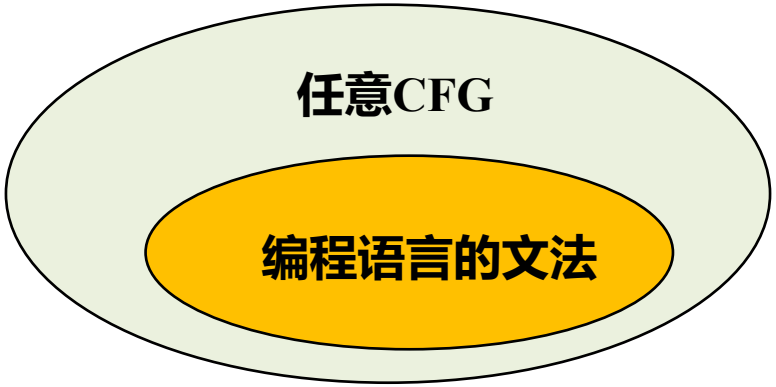
2. 语法分析概述

- CFG的Parse Tree
- 设计编程语言的文法

设计编程语言的文法

- **为了高效语法分析，可能对文法做处理/限制**
  - **消除二义性** (Resolving ambiguities)
    - 二义性：可以为一个句子生成多颗不同的分析树
  - **消除左递归(Elimination of left recursions)**
    - Avoid infinite loop in top-down parsing
  - **提左公因子(Left-factoring)**
    - Avoid backtracking in top-down parsing

(通常限于自顶向下分析)



二义性文法 (Ambiguous Grammar)

- 如果文法的某些句子存在不止一棵分析树，则该文法是二义的
- “给定CFG是否无二义性?”是不可判定问题 [1]
- 但能给出一组充分条件， 满足这组充分条件的文法是无二义性的
  - 满足，肯定无二义性
  - 不满足，也未必就是有二义性的

[1] David G. Cantor. On The Ambiguity Problem of Backus Systems. JACM’62

例: 二义性文法

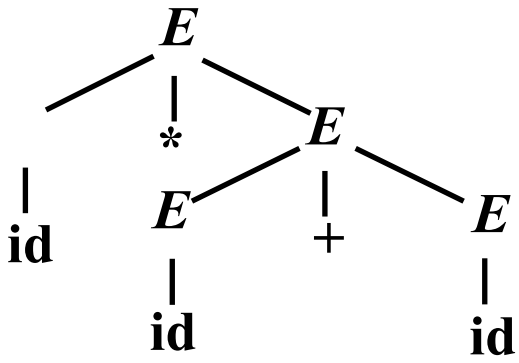
- **文法** $E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$
  - id * id + id有两个不同的最左推导

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow \mathbf{id} * E \\
&\Rightarrow \mathbf{id} * E + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E * E + E \\
&\Rightarrow \mathbf{id} * E + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}
\end{aligned}
$$

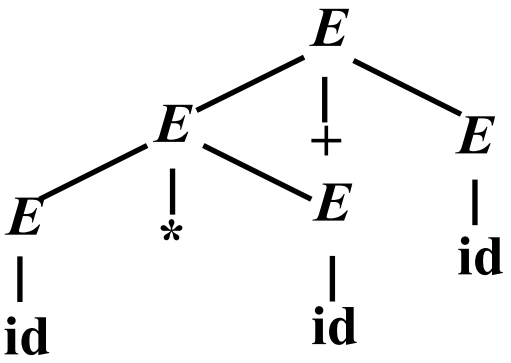
例: 二义性文法

- **文法** $E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$
  - id * id + id有两棵不同的分析树

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



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



二义性的影响

- **编程语言的文法通常是无二义的**
  - 否则就会导致一个程序有多种“正确”的**解释**

文法 $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid \mathbf{id}$

$$E \Rightarrow E*E \Rightarrow \mathbf{id}*E \Rightarrow \mathbf{id}*E+E \Rightarrow \mathbf{id}*\mathbf{id}+E \Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}$$

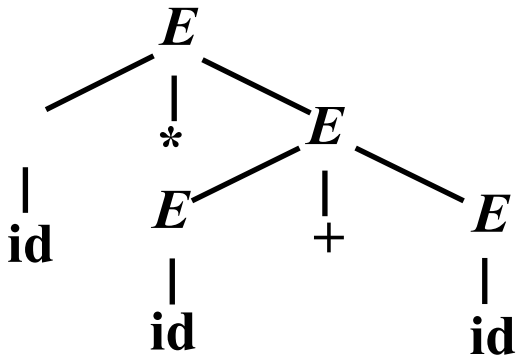
3*4+5
->3*9
->27

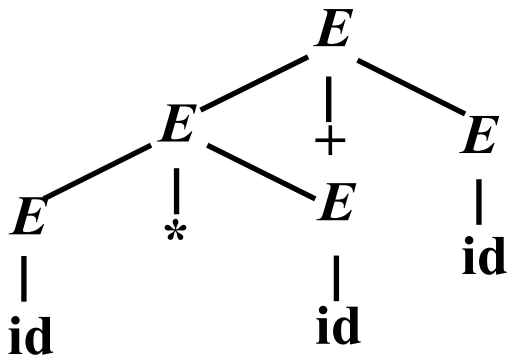
Wrong!

$$E \Rightarrow E+E \Rightarrow E*E+E \Rightarrow \mathbf{id}*E+E \Rightarrow \mathbf{id}*\mathbf{id}+E \Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}$$

3*4+5
->12+5
->17

Right





消除二义性

- **二义性的根源**
  - 多种“正确”推导处于文法同一层

- **消除二义性的惯用技术：分层**
  - 改造文法，对于引发二义性的多种推导处于文法同一层的情况，将真正想要的推导提取出来，放到更深的层次
  - 最左推导中，更深层的非终结符总是会被优先替换
  - 确保只有一种最左推导，消除二义性

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

+，*操作都是**左结合**的，并且在运算中有不同的**优先级**，但是在这个文法中没有得到体现

消除二义性

- **二义性的根源**
  - 多种“正确”推导处于文法同一层
- **消除二义性的惯用技术：分层**
  - 规定符号的优先级
  - 规定符号的结合性

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$$

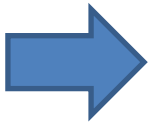
$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathrm{id}$$

+，*操作都是**左结合**的，并且*比+有更高的**优先级**

例: 消除二义性

- **运算优先级**
  - 根据算符不同的优先级，引入新的非终结符
  - 越接近开始符号 S 的文法符号优先级越低
- **运算结合性**
  - 递归非终结符在终结符左边，运算就左结合
  - 如A → Aβ，A 在终结符(如*)左侧出现(即终结符在 β 中)

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$



$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

+，*操作都是**左结合**的，并且*比+有更高的**优先级**

例: 消除二义性

- 在 Yacc (Parser自动生成器)等工具中，我们可以直接指定优先级、结合性而无需自己重写文法。

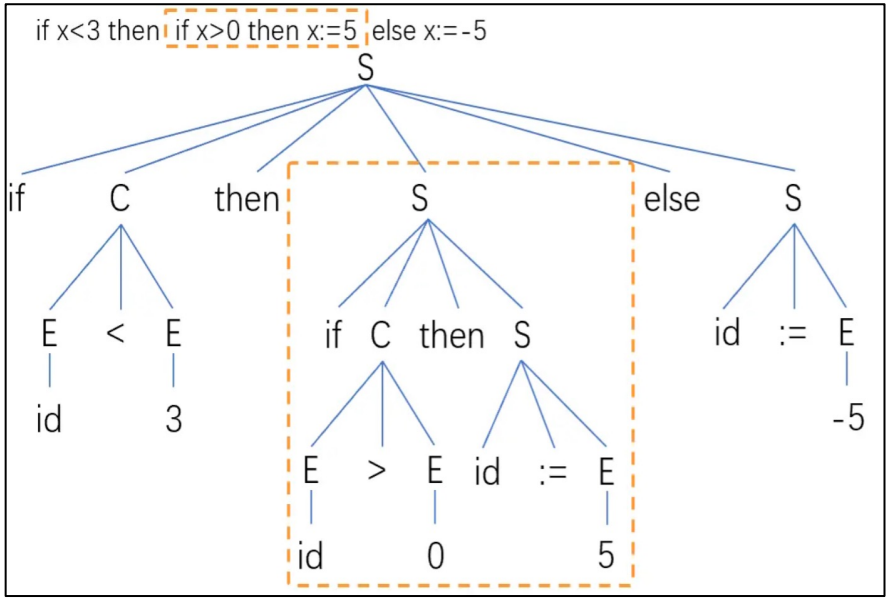
```
1    %left '+'
2    %left '*'
3    %right '-'
```

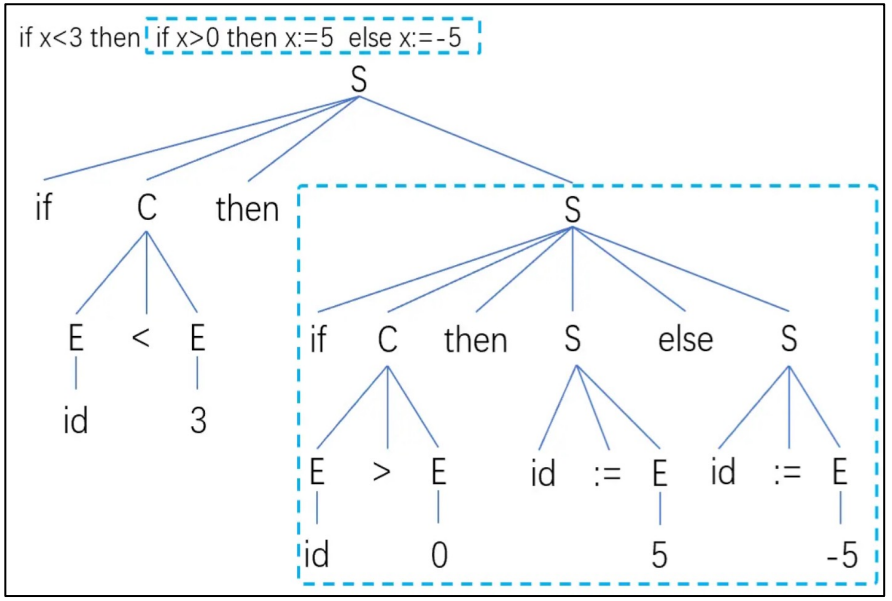
练习: 文法的二义性

- 右图的文法有无二义性?
- 如果存在二义性，如何消除

$$
\begin{aligned}
S &\rightarrow \text{if } C \text{ then } S \\
&\mid \text{if } C \text{ then } S \text{ else } S \\
&\mid \text{id} := E \\
C &\rightarrow E = E \mid E < E \mid E > E \\
E &\rightarrow E + E \mid - E \mid \text{id} \mid \text{n}
\end{aligned}
$$

if x < 3 then if x > 0 then x := 5 else x := -5

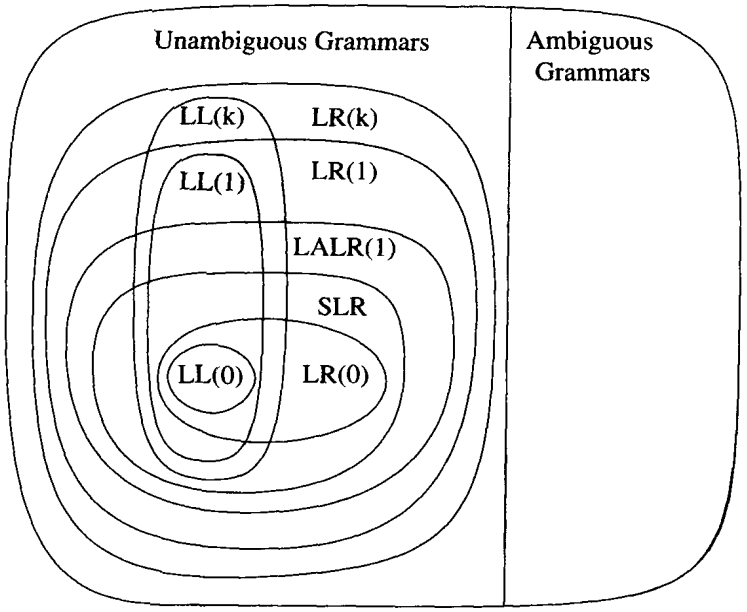




本讲小结

- **上下文无关文法**
  - CFG及其定义的语言
  - 推导和归约、最左/右推导
  - RE和CFG和联系和区别

- **语法分析概述**
  - CFG的Parse Tree
  - 语法分析作为搜索问题
    - 自顶向下、自底向上
  - 设计编程语言的文法
    - 二义性



Thank you all for your attention

通常的符号约定(供参考，不用记忆)

- **终结符** (Terminals, T)
  - (*a*) 字母表中**排在前面的小写字母**，如 *a*、*b*、*c*
  - (*b*) **运算符**，如 +、*等
  - (*c*) **标点符号**，如括号、逗号等
  - (*d*) **数字**0、1、…、9
  - (*e*) **粗体字符串**，如id、if等

通常的符号约定

- 终结符(Terminals, T)
- 非终结符 (Nonterminals， N)
  - (*a*) 字母表中排在前面的大写字母，如*A*、*B*、 *C*
  - (*b*) 字母*S*。通常表示开始符号
  - (*c*) 小写、斜体的名字，如 *expr*、*stmt*等
  - (*d*) 代表程序构造的大写字母。如*E*(表达式)、*T*(项)

通常的符号约定

- **终结符**
- **非终结符**

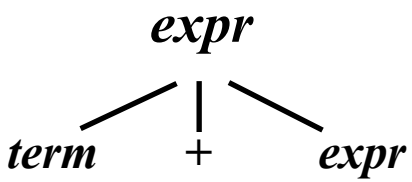
终结符	a, b, c	终结符号串	$u, v, \ldots, z$
非终结符	A, B, C		
文法符号	X, Y, Z	文法符号串	α, β, γ

- 字母表中排在后面的大写字母（如X、Y、Z）表示文法符号（即终结符或非终结符）
- 字母表中排在后面的小写字母（主要是u、v、$\ldots$、z）表示终结符号串（包括空串）
- 小写希腊字母，如α、β、γ，表示文法符号串（包括空串）
- 除非特别说明，第一个产生式的左部就是开始符号

练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

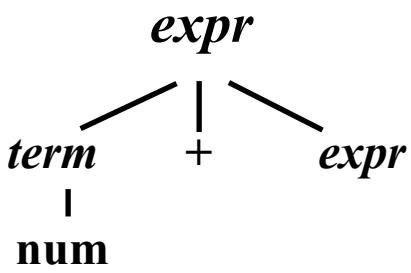
$expr \Rightarrow term + expr$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

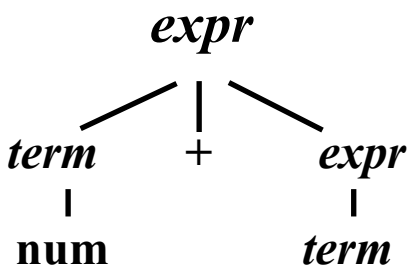
$$\begin{aligned} expr &\Rightarrow term + expr \\ &\Rightarrow \mathbf{num} + expr \end{aligned}$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

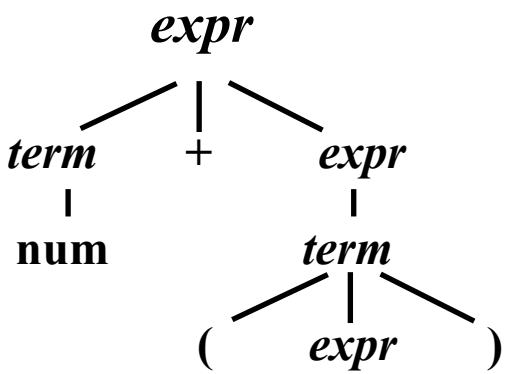
$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term
\end{aligned}
$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

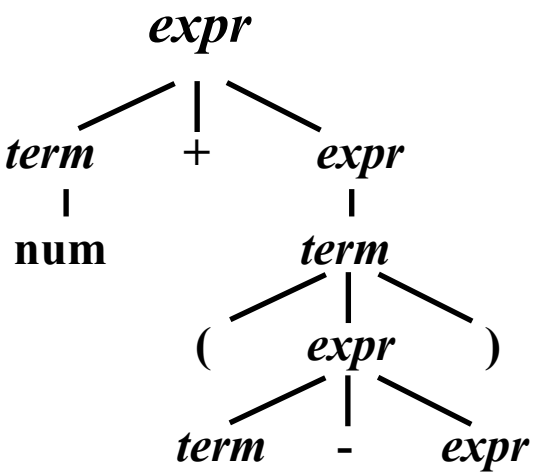
$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term \\
&\Rightarrow \mathbf{num} + (expr)
\end{aligned}
$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

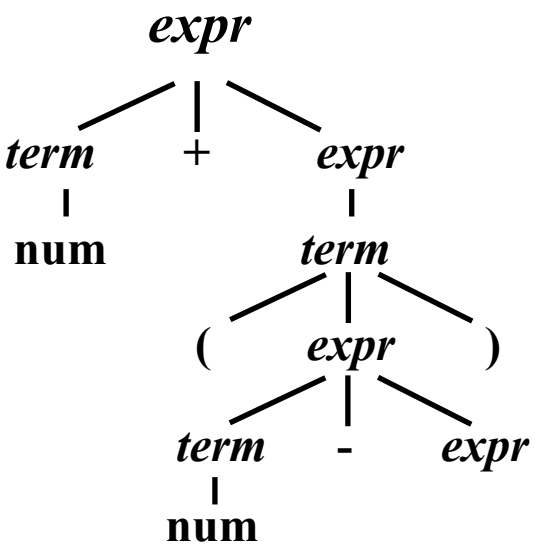
$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term \\
&\Rightarrow \mathbf{num} + (expr) \\
&\Rightarrow \mathbf{num} + (term - expr)
\end{aligned}
$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

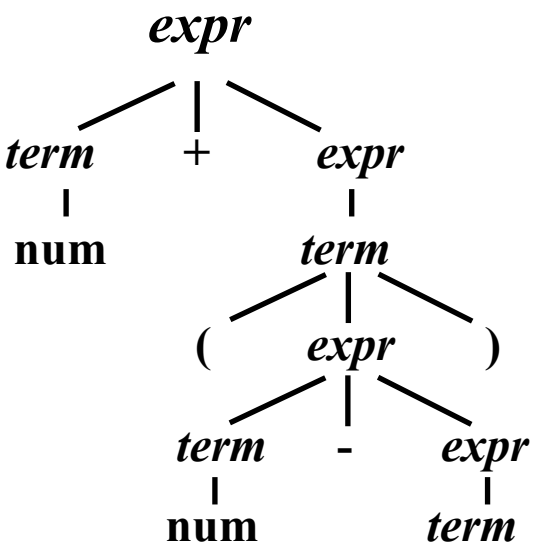
$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term \\
&\Rightarrow \mathbf{num} + (expr) \\
&\Rightarrow \mathbf{num} + (term - expr) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - expr)
\end{aligned}
$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

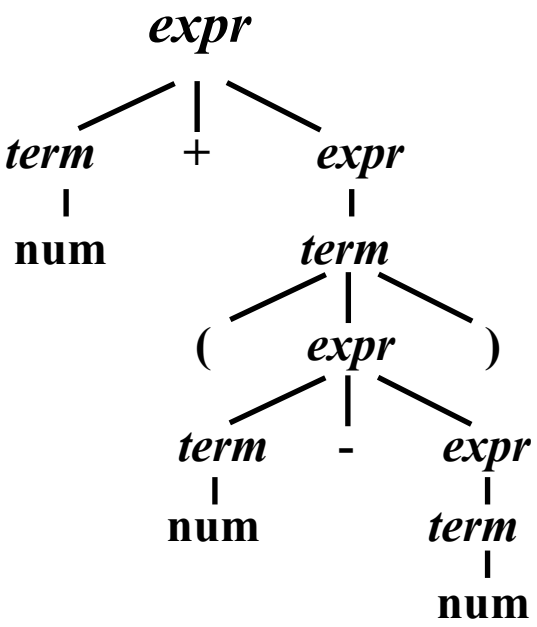
$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term \\
&\Rightarrow \mathbf{num} + (expr) \\
&\Rightarrow \mathbf{num} + (term - expr) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - expr) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - term)
\end{aligned}
$$



练习: 分析树构造

- **对于文法** $expr \rightarrow term \mid term + expr \mid term - expr$
  $term \rightarrow \mathbf{num} \mid (expr)$
- 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow \mathbf{num} + expr \\
&\Rightarrow \mathbf{num} + term \\
&\Rightarrow \mathbf{num} + (expr) \\
&\Rightarrow \mathbf{num} + (term - expr) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - expr) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - term) \\
&\Rightarrow \mathbf{num} + (\mathbf{num} - \mathbf{num})
\end{aligned}
$$



PDA(Pushdown Automaton)的形式定义

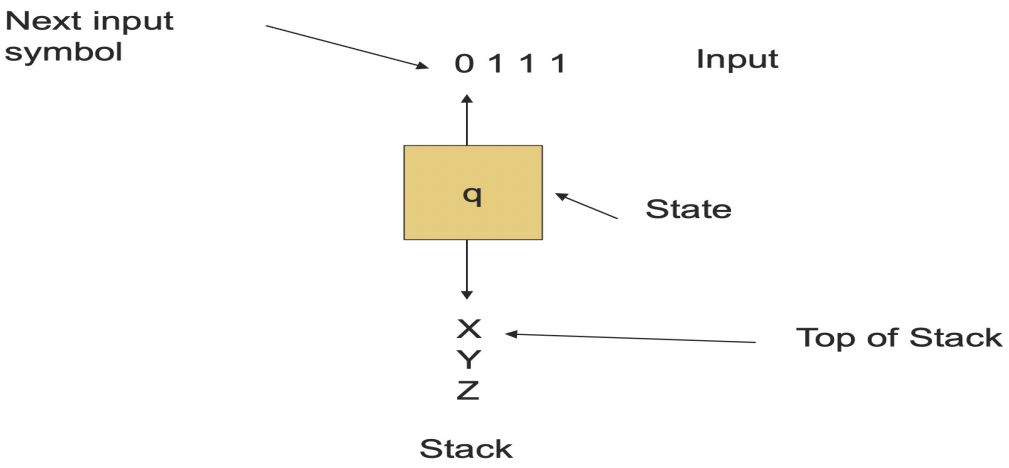
$$M=(Q,\Sigma,\Gamma,\delta,q_0,z_0,F)$$

1) Q: A finite set of states
2) Σ: A finite input alphabet
3) Γ: A finite stack alphabet
4) $q_0 \in Q$: The initial/starting styate
5) $z_0 \in \Gamma$: A starting stack symbol
6) $F \subseteq Q$: A set of final/accepting states
7) $\delta: Q\times(\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q\times\Gamma^*)$, transition function

(非“必修”，可能有助于理解LL(1)分析的非递归实现)

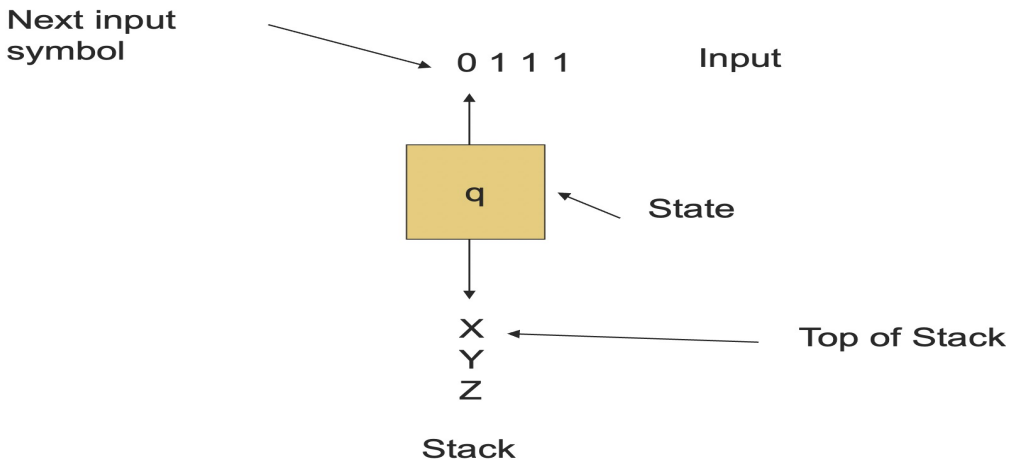
PDA的Move

- PDA上的移动δ: $Q\times(\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q\times\Gamma^*)$由下面三个因素定义：
  1. 它的当前状态；
  2. 当前输入符号（或者 ε ）；
  3. 当前的栈顶元素。



PDA的Move

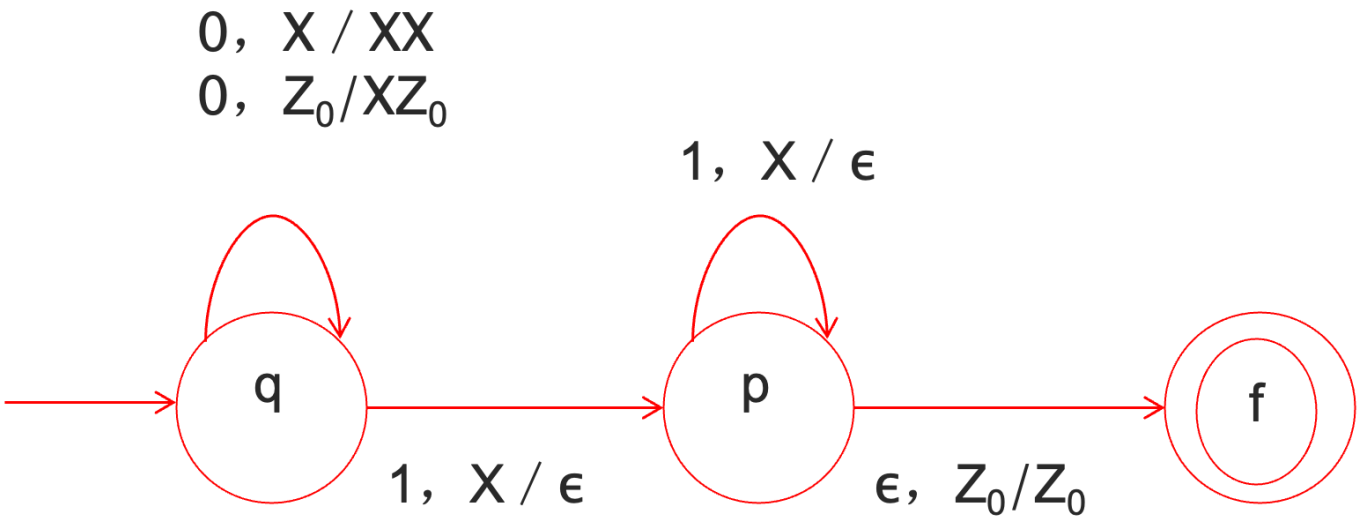
- PDA通常是非确定性的，也就是说它的下一个移动可以有不止一个选择，在每一个选择中，PDA可以：
  1. 改变状态；
  2. 将栈顶元素替换成0个或者多个符号。
     - 替换成0个符号就相当于是出栈(pop)操作；
     - 替换成多个符号就相当于一个出栈(pop)加上一系列入栈(push)操作。



例: CFL对应的PDA

- $L(M_1)=\{ 0^n1^n \mid n\geq 1 \}$

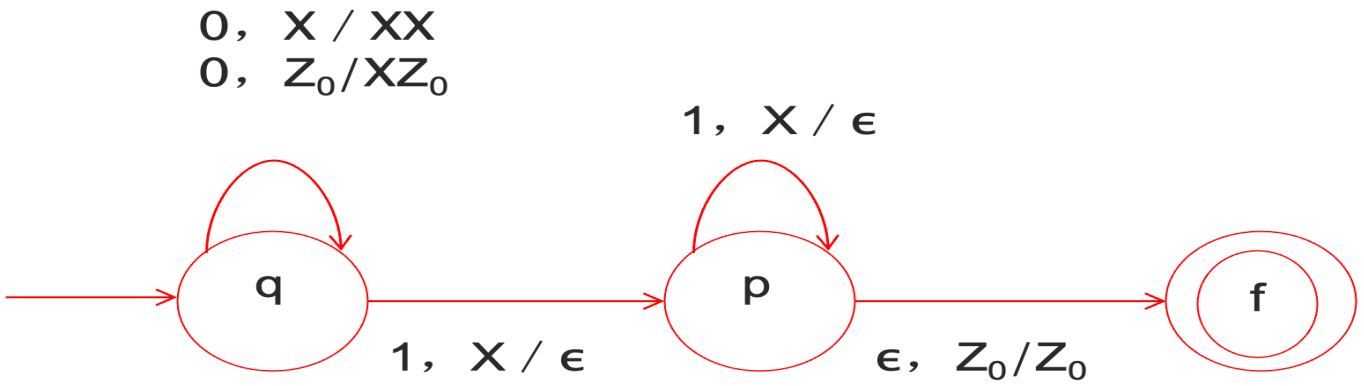
- 假定Z0 是初始符号，也标识了栈的底部
- X 是标记符，用于对输入中见到的0计数。

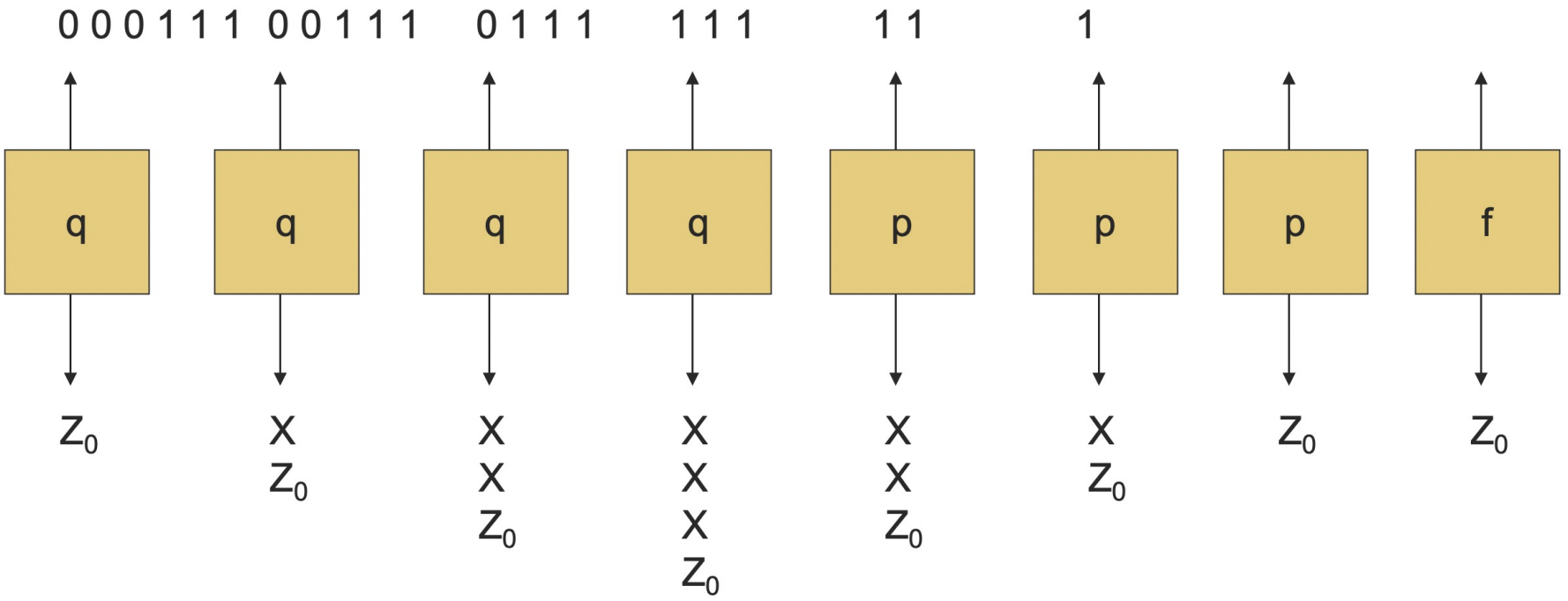


(注：不同ppt/课本上，转换边的标注方式可能不一样)

- q 是起始状态，表示我们到目前为止只看到了0；
- p 表示已经至少看到了一个1，并且只有在后续输入为1的时候才会前进；
- f 是终止状态，表示接受。

例: CFL对应的PDA


0，X / XX
0，Z0/XZ0
1，X / ϵ
q
p
f
1，X / ϵ
ϵ，Z0/Z0


0 0 0 1 1 1
0 0 1 1 1
0 1 1 1
1 1 1
1 1
1
q
q
q
q
p
p
p
f
Z0
X Z0
X X Z0
X X X Z0
X X Z0
X Z0
Z0
Z0