COMP130014编译

第三讲：上下文无关文法

徐 辉

xuh@fudan.edu.cn

主要内容

一、上下文无关文法

二、扩展BNF范式

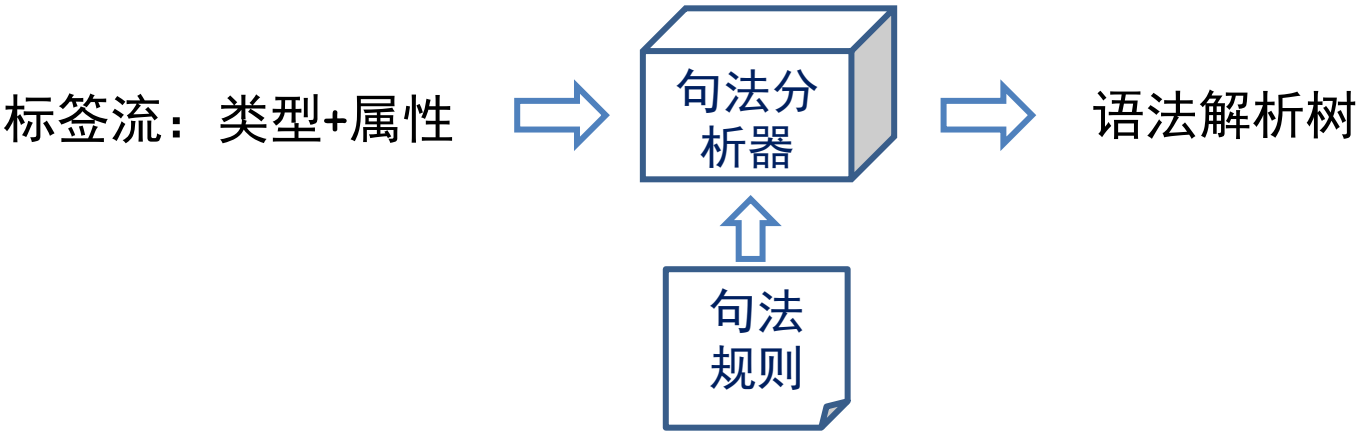
三、TeaPL文法定义

四、语言分析问题

一、上下文无关文法

句法解析问题

- 给定一个句子和句法规则，找到可生成该句子的一个推导
- 句法规则定义了句法分析器可接受的标签序列及其推导方式
- 文法/语法 = 词法 + 句法



语法推导举例

语法规则

[1] E → E + E
[2] E → E − E
[3] E → E × E
[4] E → E / E
[5] E → <UNUM>

目标句子： 1 + 2 x 3

<UNUM(1)> + <UNUM(2)> x <UNUM(3)>

推导步骤

[1] E → E + E
[5] E → <UNUM(1)> + E
[3] E → <UNUM(1)> + E × E
[5] E → <UNUM(1)> + <UNUM(2)> × E
[5] E → <UNUM(1)> + <UNUM(2)> × <UNUM(3)>

基本概念和符号

- 一门语言是多个句子的集合
- 句子是由终结符组成的序列
- 字符串是包含终结符和非终结符的序列
  - 非终结符：X、Y、Z
  - 终结符（标签）：<BINOP>、<NUM>
  - 字符串符号：α、β、γ
- 语法包含一个开始符号和多条推导规则
  - $S \rightarrow \beta$
- 语法G的语言L(G)是该语法可推导的所有句子的集合

上线文无关文法（Context-Free Grammar）

- 上下文无关语法是一个四元组(T, NT, S, P)
- T：终结符
- NT：非终结符
- S：起始符号
- P：推导规则集合：$\{X \rightarrow \gamma\}$
  - X是非终结符
  - γ 是字符串
  - 规则左侧只能有一个非终结符

括号匹配问题

- 用CFG语法设计一套括号匹配规则
- 验证：()(()())是该语法的一个推导吗?

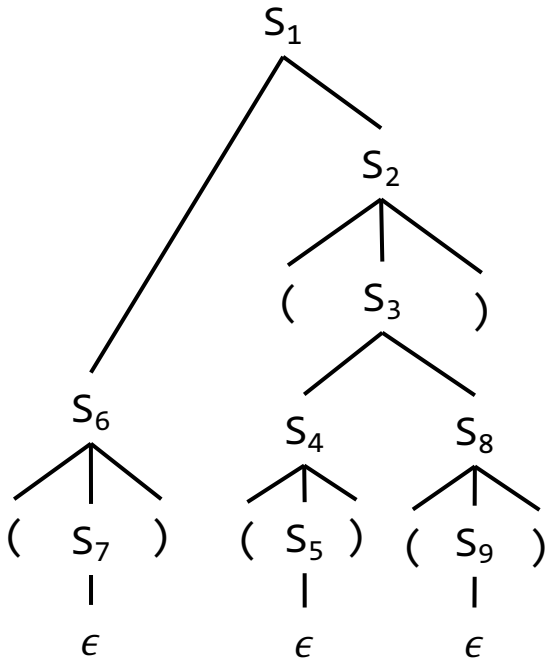
语法规则

[1] $S \rightarrow \epsilon$
[2] $S \rightarrow (S)$
[3] $S \rightarrow SS$

推导

[3] $S \rightarrow SS$
[2] $S \rightarrow S(S)$
[3] $S \rightarrow S(SS)$
[2] $S \rightarrow S(S(S))$
[1] $S \rightarrow S(S())$
[2] $S \rightarrow S((S)())$
[1] $S \rightarrow S(()())$
[2] $S \rightarrow (S)(()())$
[1] $S \rightarrow ()(()())$

语法解析树



写出计算器的CFG文法

```
[1] E → E <ADD> E
[2] E → E <SUB> E
[3] E → E <MUL> E
[4] E → E <DIV> E
[5] E → E <POW> E
[6] E → <LPAR> E <RPAR>
[7] E → NUM
[8] NUM → <UNUM>
[9] NUM → <SUB> <UNUM>
```

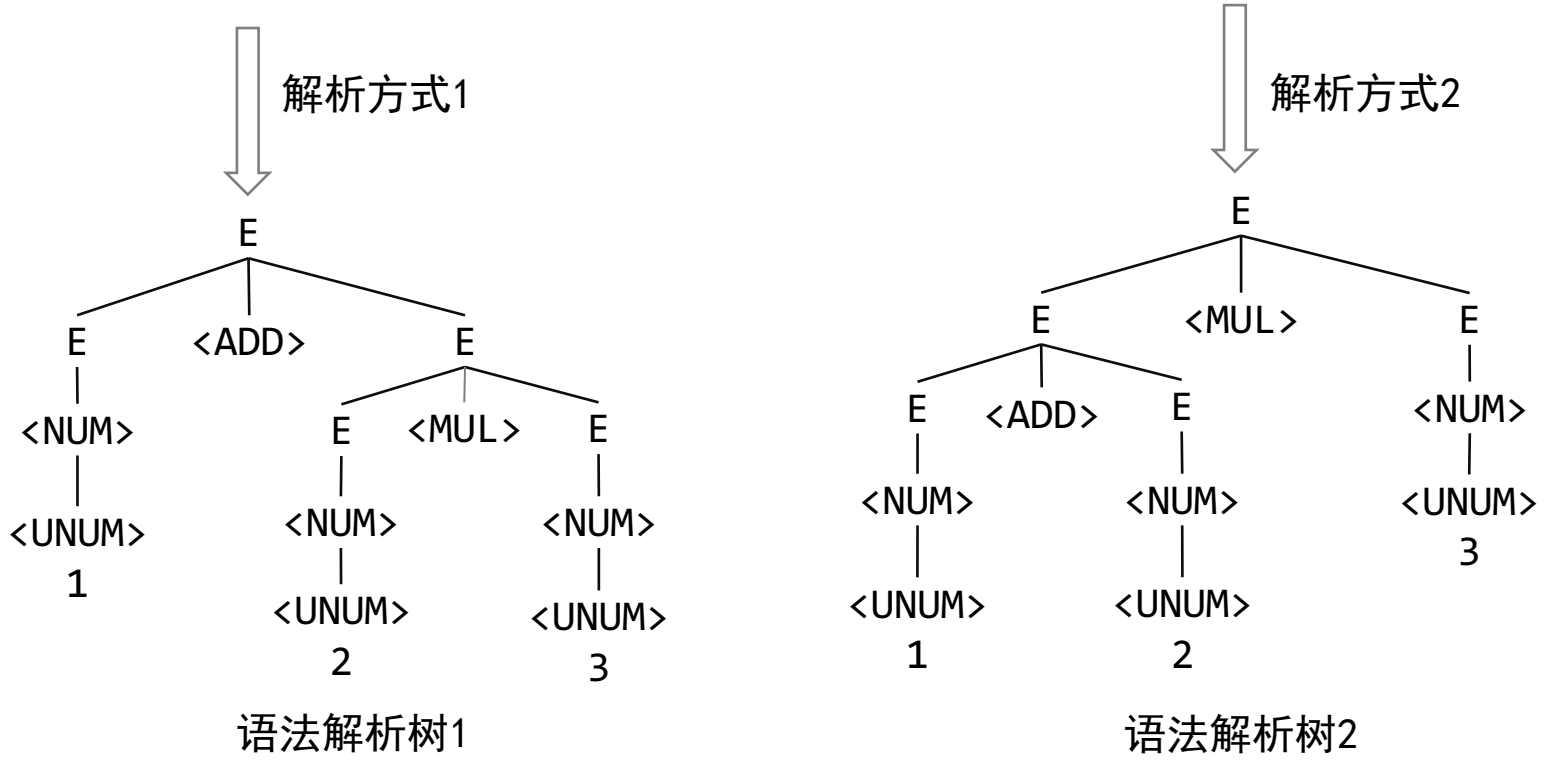
=

```
[1] E → E '+' E
[2] E → E '/' E
[3] E → E '*' E
[4] E → E '/' E
[5] E → E '^' E
[6] E → '(' E ')'
[7] E → NUM
[8] NUM → <UNUM>
[9] NUM → '-' <UNUM>
```

二义性问题

- L(G)中的某个句子存在一个以上的最左（或最右）推导
- 语法解析树不同

标签流：<UNUM(1)> <ADD> <UNUM(2)> <MUL> <UNUM(3)>




If your DOG
does a POO
Please put it
in a litter bin.
Please help keep our
open spaces clean.

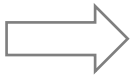
A programmer’s wife asks him to go to the grocery. She says "Get a gallon of milk. If they have eggs, get 12."

The programmer returns with 12 gallons of milk.

消除二义性：将运算符特性融入语法规则

- 优先级：^>×/÷>+/−
- 结合性：×/÷/+/−左结合，^右结合

```
[1] E → E '+' E
[2] E → E '/' E
[3] E → E '*' E
[4] E → E '/' E
[5] E → E '^' E
[6] E → '(' E ')'
[7] E → NUM
[8] NUM → <UNUM>
[9] NUM → '-' <UNUM>
```



```
[1] E → E OP1 E1
[2] E → E1
[3] E1 → E1 OP2 E2
[4] E1 → E2
[5] E2 → E3 OP3 E2
[6] E2 → E3
[7] E3 → NUM
[8] E3 → '(' E ')'
[9] NUM → <UNUM>
[10] NUM → '-' <UNUM>
[11] OP1 → '+'
[12] OP1 → '-'
[13] OP2 → '*'
[14] OP2 → '/'
[15] OP3 → '^'
```

练习：为下列语言设计语法规则：

1) 所有0和1组成的字符串，每个0后面紧跟若干个1
2) 所有0和1组成的字符串，0和1的个数相同
3) 所有0和1组成的字符串，0和1的个数不相同

练习：语法设计

- 为正则语言设计CFG（用于解析正则表达式）
  - 支持字符 [A-Za-z0-9]
  - 支持连接、选择|、闭包*
  - 支持()
- 检查语法是否有二义性？

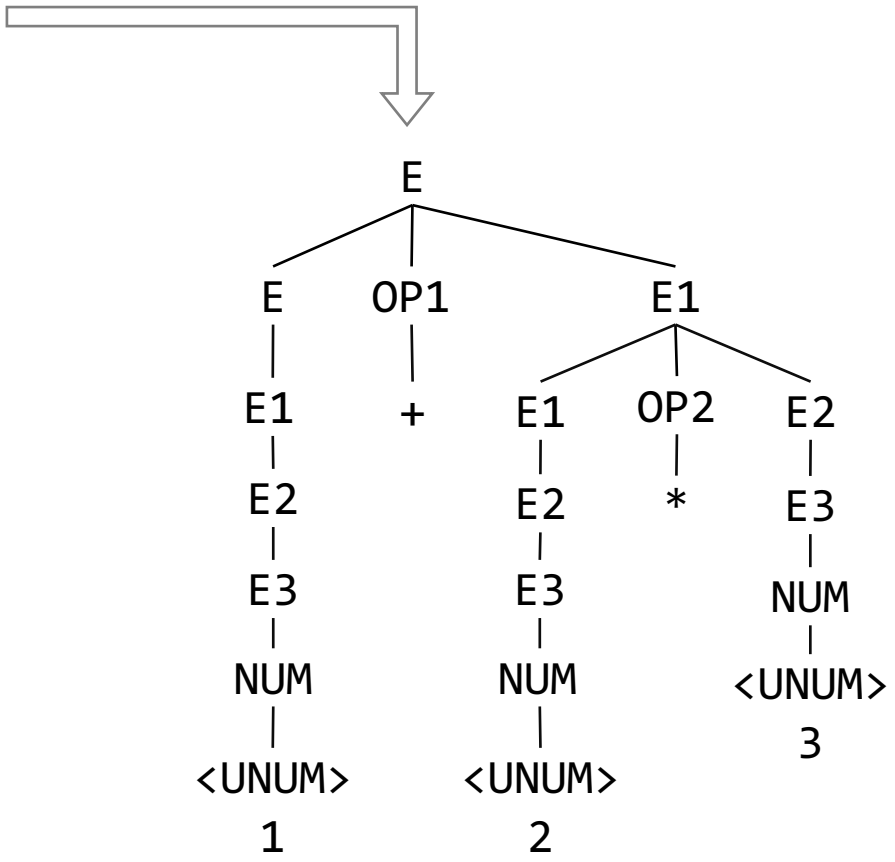
二、扩展BNF范式

CFG的问题

- 规则条目多且复杂，易写易读性差
- 语法解析树复杂

```
[1] E → E OP1 E1
[2] E → E1
[3] E1 → E1 OP2 E2
[4] E1 → E2
[5] E2 → E3 OP3 E2
[6] E2 → E3
[7] E3 → NUM
[8] E3 → '(' E ')'
[9] NUM → <UNUM>
[10] NUM → '-' <UNUM>
[11] OP1 → '+'
[12] OP1 → '-'
[13] OP2 → '*'
[14] OP2 → '/'
[15] OP3 → '^'
```

应用：1+2*3的语法解析树



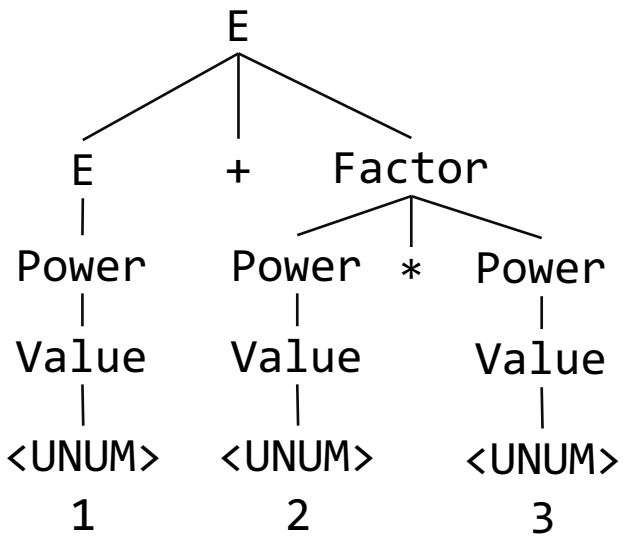
扩展BNF范式（Extended Backus-Naur Form）

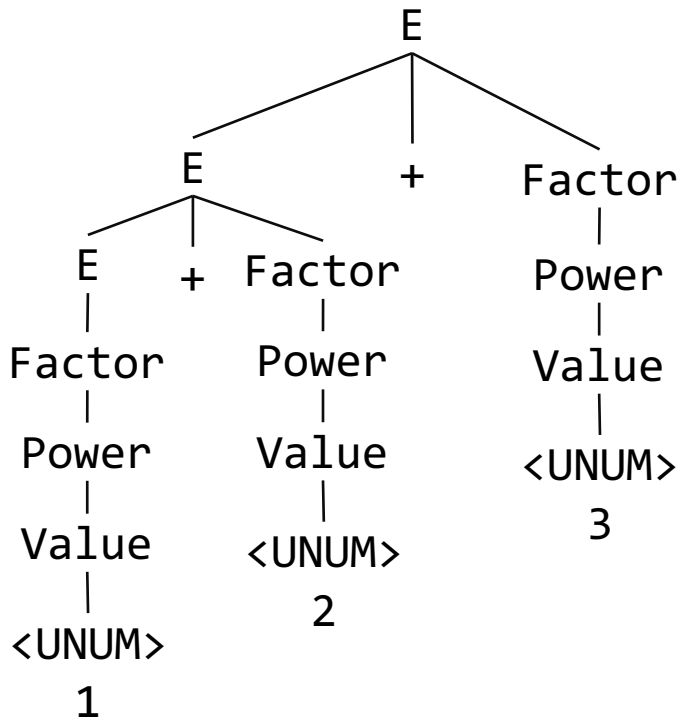
- 引入更多运算符提升规则描述效率
- EBNF运算符有多种表示方法，我们沿用正则文法符号

构造方式	符号	优先级	示例
字符串	' '	5	'a'
通配符	.	5	
字符集	[]	5	[a-z]
可选匹配	?	4	β?
闭包	*	4	β*
正闭包	+	4	β+
非	!	3	!β
连接		2	$\alpha\beta$
选择	|	1	$\alpha\|\beta$

EBNF及其对应的语法解析树

```
[1] E → (E ('+'|'-'))? Factor
[2] Factor → (Factor ('*'|'/'))? Power
[3] Power → Value ('^' Power)?
[4] Value → <UNUM> | ('-' <UNUM>) | ('(' E ')')
```

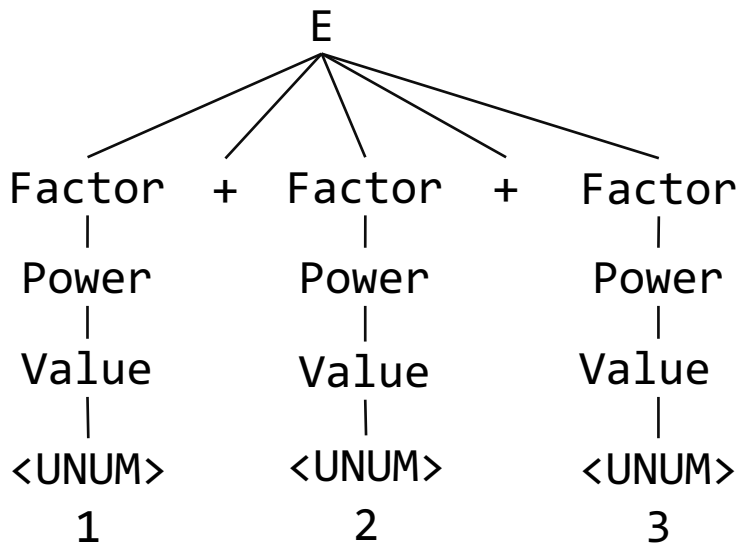




EBNF对应的语法解析树

```
[1] E → Factor (('+'|'-') Factor)*
[2] Factor → Power (('*'|'/') Power)*
[3] Power → Value ('^' Power)?
[4] Value → <UNUM> | ('-' <UNUM>) | ('(' E ')')
```

?



练习：使用EBNF设计/改写正则语言文法

- 为正则语言设计CFG（用于解析正则表达式）
  - 支持字符 [A-Za-z0-9]
  - 支持连接、或|、闭包*运算
  - 支持()
- 检查语法是否有二义性？

思考

- 扩展EBNF和CFG表达能力是否等价？

三、TeaPL文法定义

使用EBNF定于TeaPL：程序组成

```
program ⟼  (varDeclStmt | structDef | fnDeclStmt
            | fnDef | macro | comment)*
```

varDeclStmt	全局变量声明
structDef	数据结构定义
fnDeclStmt	函数声明
fnDef	函数定义
macro	宏
comment	注释

变量声明形式

```
let a:int;
let a:int = 0;
let a;
let a = 0;
let a[5]:int;
let a[n]:int;
let a[n];
let a[2]:int = {0};
let a[2]:int = {1, 2};
```

变量声明

声明时初始化

类型可省略

支持数组类型

数组声明时初始化

长度应保持一致（类型检查）

不支持：

- 二维数组：`let a[m][n];`
- 一条语句同时声明多个变量：`let i,j;`

变量声明

```
varDeclStmt ⟼ 'let' (varDecl | varDef) ';'

 varDecl ⟼ id (':' type)?
         | id '[' (id | num) ']' (':' type)?

  varDef ⟼ id (':' type)? '=' rightVal
         | id '[' (id | num) ']' (':' type)? '=' '{' num '}'
```

类型

```
        type ⟼ primitiveType | structType | ptrType
primitiveType ⟼ int | bool | char | long | float | double
   structType ⟼ id
     ptrType ⟼ '*' type

   structDef ⟼ 'struct' id '{' fieldDecl (, fieldDecl)* '}'
   fieldDecl ⟼ id ':' type
             | id '[' (id | num) ']' ':' type
```

```
struct MyStruct {
    x: int,
    y[10]: int
}
```

结构体内可以包含数组

右值表达式

```
 rightVal ⟼ arithExpr | boolExpr
arithExpr ⟼ (arithExpr ('+' | '-'))? factor
   factor ⟼ (factor ('*' | '/'))? exprUnit
 exprUnit ⟼ num | id | fnCall | '(' rightVal ')'
          |  id '.' id | id '[' (id | num) ']'
          |  deref | addr | string
      num ⟼ unum | ('-' unum)
    deref ⟼ '*' id
     addr ⟼ '&' id
   string ⟼ '"' (!'"')* '"'
```

函数声明和定义

```
fn foo(a:int, b:int) -> int;

fn foo(a:int, b:int) -> int {
    return a + b;
}
```

函数声明

函数定义

```
fnDeclStmt ⟼ 'fn' fnSign ';'
    fnSign ⟼ id '(' params? ')' '->' type?
    params ⟼ id ':' type (',' id ':' type)*

     fnDef ⟼ 'fn' fnSign codeBlock
 codeBlock ⟼ '{' (stmt | codeBlock)* '}'
```

基本语句

```
     stmt ⟼ varDeclStmt | assignStmt | callStmt
            | retStmt | ifStmt | whileStmt
            | breakStmt | continueStmt | ';'
            | forStmt | matchStmt

assignStmt ⟼ leftVal '=' rightVal ';'
   leftVal ⟼ id | id '[' (num | id) ']' | id '.' id
             | deref
```

基本语句

```
 callStmt ⟼ fnCall ';'
   fnCall ⟼ id '(' (rightVal (, rightVal)*)? ')'
  retStmt ⟼ 'ret' rightVal? ';'

   ifStmt ⟼ 'if' '(' boolExpr ')' codeBlock (else codeBlock)?
whileStmt ⟼ 'while' '(' boolExpr ')' codeBlock

  breakStmt ⟼ 'break' ';'
continueStmt ⟼ 'continue' ';'
```

```
if(a>=b) {
    ...
} else {
    ...
}
```

强制使用括号，避免歧义

条件表达式

```
boolExpr  ⟼ (boolExpr '||')? andExpr
 andExpr  ⟼ (andExpr '&&') boolUnit
boolUnit  ⟼ cmpExpr | '!(' cmpExpr ')' | '!'?'(' boolExpr ')'
 cmpExpr  ⟼ exprUnit ('>' | '>=' | '<' | '<=' | '==' | '!=')
              exprUnit
```

```
if(!(a>=b)) {...}
if((a>b) && (b>c)){...}
if(a>b>c) {...}
if(a+b>c+d){...}
if((a+b)>(c+d)){...}
if(a>b==c){...}
```

!后强制使用括号，避免歧义

不允许，易产生歧义

不允许，易产生歧义

不允许，易产生歧义

运算符优先级总结：与C语言标准兼容

优先级	运算符	含义	结合性	TeaPL限制
8	-,!	单目运算符	右	'-'后只跟数字，'!'只能跟'('
7	*,/	乘除号	左	
6	+,-	加减号	左	
5	>,>=,<,<=	比大小	左	不支持连续比较 比较对象不支持逻辑运算
4	==,!=	等价性	左	
3	&&	逻辑与	左	
2	||	逻辑非	左	
1	=	赋值	右	不支持连续赋值

https://c-cpp.com/c/language/operator_precedence

数字、标识符和注释

```
unum  ⟼  [1-9][0-9]*|0
      |  ([1-9][0-9]*|0) '.' [0-9]+

  id  ⟼  [a-z_A-Z] ([a-z_A-Z0-9])*


comment  ⟼  '//' (!newline)* newline
comment  ⟼  '/*' (!'*/')* '*/'
newline  ⟼  '\n'
```

问题：下列哪些代码符合TeaPL语法？

```
fn foo() {
    ret 1;
}
```

```
fn foo() {
    ret 2>1;
}
```

```
fn foo() -> int {
    ret 1;
}
```

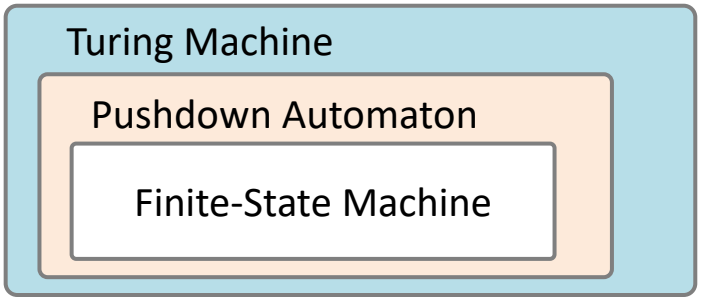
```
fn foo() -> bool {
    ret 2>1;
}
```

四、文法能力分类

按表达能力分类

Chomsky Hierarchy

类型	文法名称	计算模型	规则形式	语言示例
0 型	递归枚举	图灵机	无限制	
1 型	上下文敏感	线性有界图灵机	左侧可以多个符号 $\alpha S \rightarrow \beta$	$a^n b^n c^n$
2 型	上下文无关	下推自动机	左侧仅一个符号$S \rightarrow \beta$	$a^n b^n$
3 型	正则	有穷自动机	右侧全部为终结符$S \rightarrow$ab	a^n



正则语言 VS 上下文无关语言

- 正则语言也可以用CFG规则形式表示：
  - $X \to \gamma$
  - $\gamma \to \gamma_1$
  - ...
- 特点：右侧的非终结符均可替换为终结符

$$
\begin{array}{l}
[1]\ S \to \mathrm{A}|B \\
[2]\ A \to (0?\,1)^* \\
[3]\ B \to (1?\,0)^*
\end{array}
\quad \Longrightarrow \quad S \to (0?\,1)^*|(1?\,0)^*
$$

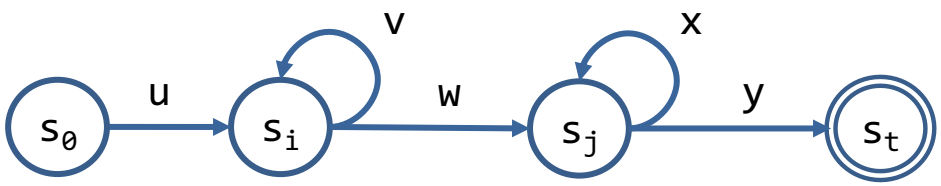
非CFG语言：上下文敏感语法

- $L = \{a^n b^n c^n, n > 0\}$不是CFG语言
- 上下文敏感文法规则形式：$aS \to \beta$

[1] $S \to aBC$
[2] $S \to aSBC$
[3] $CB \to BC$
[4] $aB \to ab$
[5] $bB \to bb$
[6] $bC \to bc$
[7] $cC \to cc$

非CFG语言的泵引理

- CFG语言的泵引理（必要条件）：
  - 任意长度超过p的句子可以被拆分为$uvwxy$的形式
  - v和x被重复任意次后得到的新句子（如$uvvwxxy$）仍属于该语言
- 正则属于CFG：$uv^n w\epsilon^n\epsilon$



练习：下列语言是否为正则语言？

- 集合表示
  1) $L = \{a^n b^n | n \leq 100\}$
  2) $L = \{a^n | n \geq 1\}$
  3) $L = \{a^{2n} | n \geq 1\}$
  4) $L = \{a^p |$ p is prime$\}$
- Regex/CFG语法表示
  1) $S \rightarrow (0?\,1)^*$
  2) $S \rightarrow aT|\epsilon, T \rightarrow Sb$
  3) $S \rightarrow 0S1S|1S0S|\epsilon$

思考

1) 用正则表达式可以定义任意正则语言吗？
2) 有穷自动机可以解析任意正则表达式吗？
3) 用CFG可以定义任意正则语言吗？
4) 用CFG可以定义任意上下文无关语言吗？
5) 用下推自动机可以解析任意正则表达式吗？
6) 用下推自动机可以解析任意CFG规则吗？
7) 用通用图灵机可以解析任意CFG规则吗？
8) 用通用图灵机可以解析任意程序吗？