CS30017 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、上下文无关文法工具

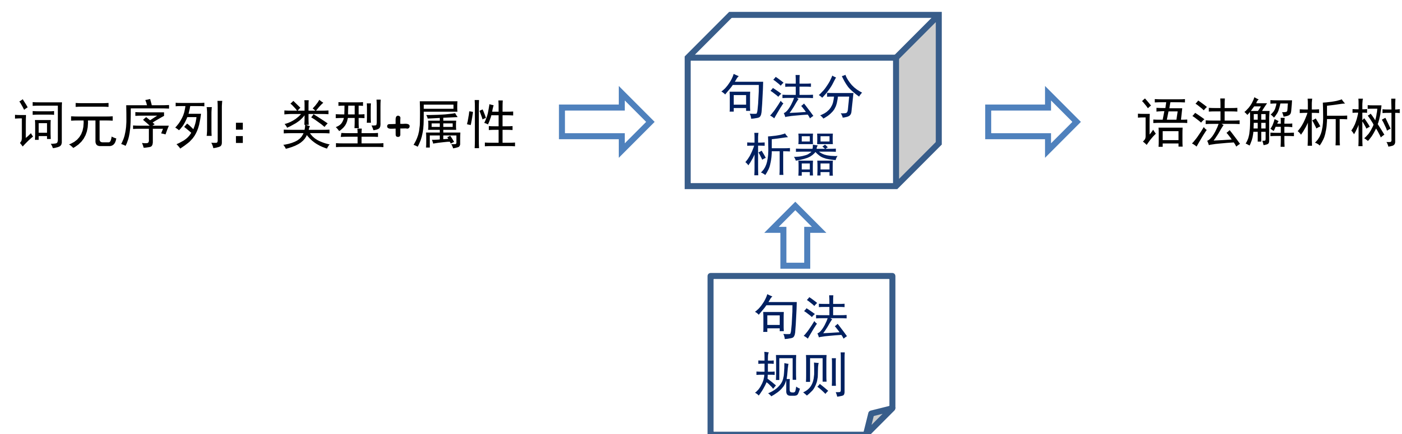
三、TeaLang语法规则

四、文法能力分类

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词元序列及其推导方式
- 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

- **句子**：由词元组成的序列
- **语言**：多个句子的集合
- 语法：包含一个开始符号和多条推导规则
  - 规则形式：$S \to \beta$
  - 规则符号使用约定：
    - 非终结符：X、Y、Z
    - 终结符（词元）：<BINOP>、<NUM>
    - 终结符和非终结符组成的序列：α、β、γ
- 语法G的语言L(G)：语法L(G)可推导的所有句子的集合

上线文无关文法（Context-Free Grammar）

- 上下文无关文法是一个四元组(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
  - CFG要求规则左侧只能有一个非终结符，否则上下文相关
  - γ 是终结符和非终结符组成的符号序列

括号匹配问题

- 设计一套CFG语法规则，解决括号匹配问题
- 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] $S \rightarrow SS$

推导过程

[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] $S \rightarrow ()(()())$

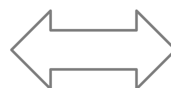
语法解析树

S
S
(S)
S S S
(S) (S) (S)
ϵ ϵ ϵ

如何为计算器设计CFG语法规则？

```
[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>
```

语法规则



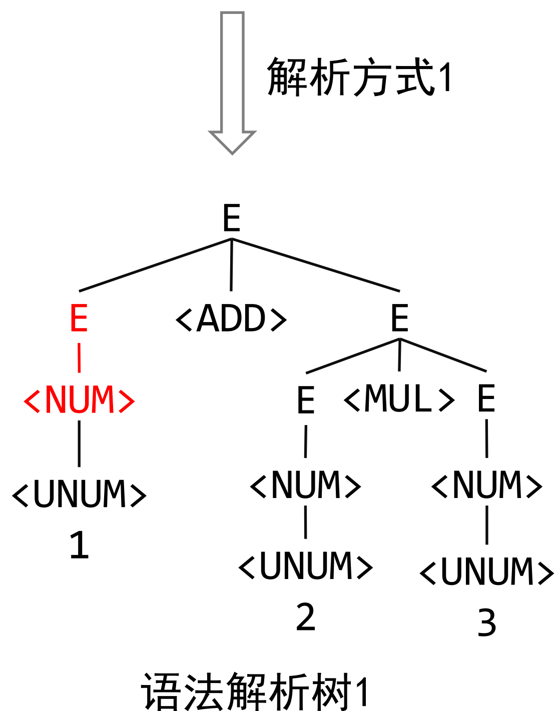
```
[1] E → E '+' E
[2] E → E '-' E
[3] E → E '*' E
[4] E → E '/' E
[5] E → E '^' E
[6] E → '(' E ')'
[7] E → NUM
[8] NUM → <UNUM>
[9] NUM → '-' <UNUM>
```

简化终结符表示

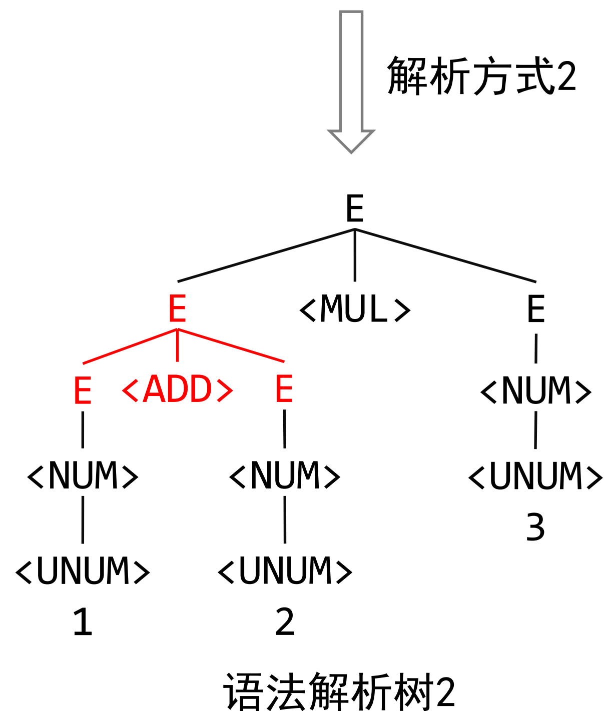
二义性问题

- 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

词元序列： <UNUM(1)> <ADD> <UNUM(2)> <MUL> <UNUM(3)>



语法解析树1



语法解析树2

RANDWICK CITY COUNCIL
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：语法规则设计

- 为正则语言设计语法规则，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？

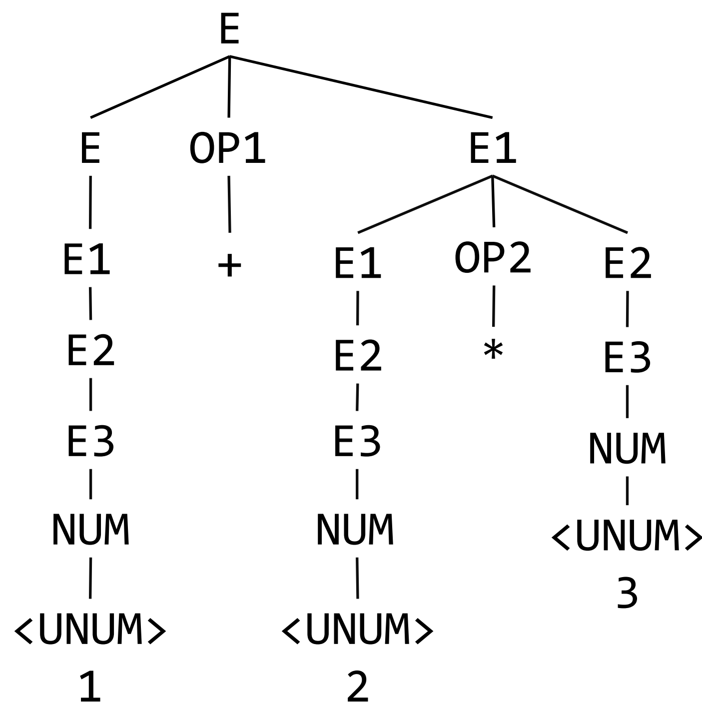
二、上下文无关文法工具

CFG的问题

- 规则条目多且繁琐，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

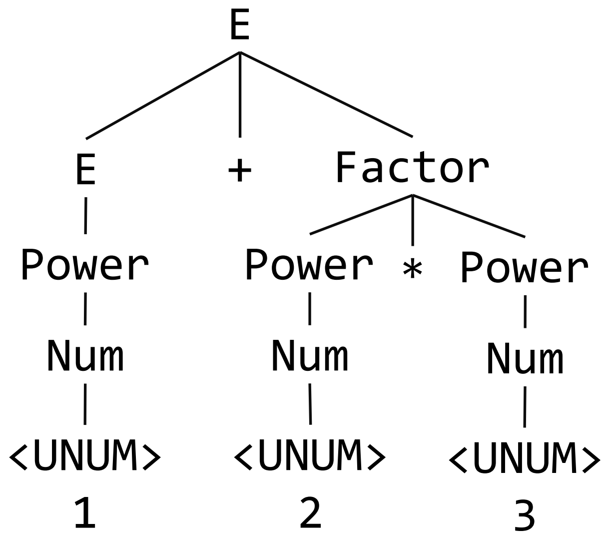
EBNF范式（Extended Backus-Naur Form）

- 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	含义
字符串	' '	5	'ab'	
可选匹配	?	4	β?	
闭包	*	4	β*	
正闭包	+	4	β+	
排除	-	3	α-β	符合α，但非β；α和β都是正则表达式
连接		2	$\alpha\beta$	
选择	|	1	$\alpha \| \beta$	

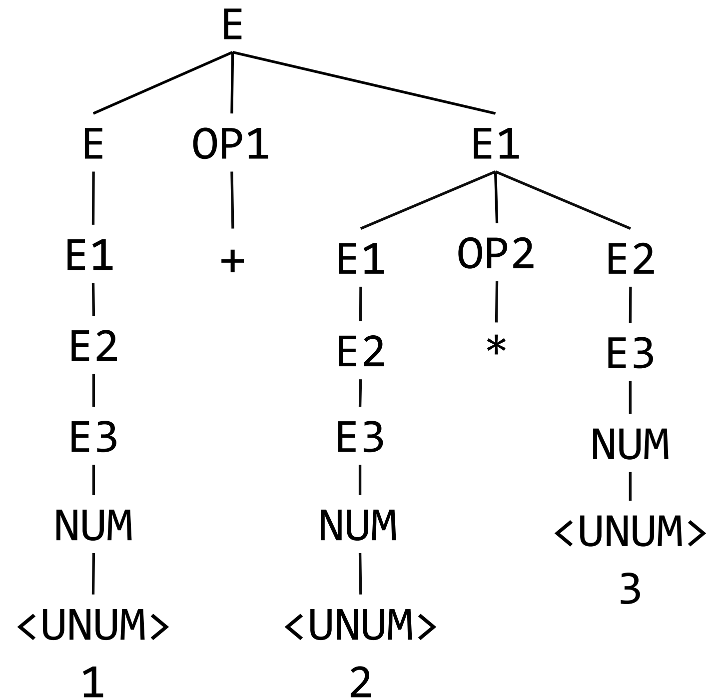
EBNF及其语法解析树实例

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



1+2*3的语法解析树

对比之前



PEG文法

- 带优先级的解析规则集合
- Pest工具使用，不展开讨论

构造方式	符号	优先级	示例	含义
字符串	' '	5	'ab'	
可选匹配	?	4	β?	
闭包	*	4	β*	
正闭包	+	4	β+	
排除	-	3	α-β	符合α，但非β；α和β都是正则表达式
连接		2	$\alpha\beta$	
选择	~~|~~	~~1~~	~~$\alpha\|\beta$~~	
顺序选择	/		α/β	如果同时满足α和β, 按照从左至右选择

练习：使用EBNF改写正则语言语法规则

- 为正则语言设计语法规则，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？

三、Tea语言语法规则

Tea语言介绍

- Rust语言的子集
- 支持i32、数组、struct类型
- 没有所有权等高级语义

使用EBNF定义Tea语言语法：程序组成

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

useStmt	模块引用
varDeclStmt	全局变量声明
structDef	数据结构定义
fnDeclStmt	函数声明
fnDef	函数定义
comment	注释

模块引用

```
useStmt ⟼  'use' modPath ';'
modPath ⟼  id ('::' id)*
```

```
// 标准库，提供常用函数
fn getint()->i32;
fn getch()->i32;
fn timer_start(lineno:i32);
fn timer_stop(lineno:i32);
fn putint(a:i32);
fn putch(a:i32);
```

std.teah头文件

```
// 引用std模块
use std;

//使用std声明的函数
ch = std::getch();
```

变量声明和定义

```
let a:i32;
let a:i32 = 0;
let a;
let a = 0;
let a:[i32;5];
let a:[i32;2] = [0;2];
let a:[i32;2] = [1,2];
```

- `let a:i32;` → 变量声明
- `let a:i32 = 0;` → 声明时初始化
- `let a;` / `let a = 0;` → 类型可省略
- `let a:[i32;5];` → 支持数组类型
- `let a:[i32;2] = [0;2];` → 数组声明时初始化
- `let a:[i32;2] = [1,2];` → 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;`

变量声明规则

$$
\begin{aligned}
\texttt{varDeclStmt} &\mapsto \texttt{'let' (varDecl | varDef) ';'} \\
\texttt{varDecl} &\mapsto \texttt{scalarDecl | typedScalarDecl} \\
&\quad\ \texttt{arrayDecl | typedArrayDecl} \\
\texttt{scalarDecl} &\mapsto \texttt{id} \\
\texttt{typedScalarDecl} &\mapsto \texttt{id ':' type} \\
\texttt{arrayDecl} &\mapsto \texttt{id ':' '[' num ']'} \\
\texttt{typedArrayDecl} &\mapsto \texttt{id ':' '[' type ';' num ']'}
\end{aligned}
$$

变量定义规则

```
varDef ⟼ scalarDecl '=' rValue
        | typedScalarDecl '=' rValue
        | arrayDecl '=' arrayInit
        | typedArrayDecl '=' arrayInit

arrayInit ⟼ '=' '[' rValue (',' rValue)* ']'
          | '=' '[' rValue ';' num ']'
```

类型

- 主要支持i32和结构体
- 不支持指针和一般引用
- slice类型只用于函数调用参数传递

$$
\begin{aligned}
\texttt{type} &\mapsto \texttt{primitiveType} \mid \texttt{structType} \mid \texttt{sliceType} \\
\texttt{primitiveType} &\mapsto \texttt{i32} \mid \texttt{bool} \\
\texttt{structType} &\mapsto \texttt{id} \\
\texttt{sliceType} &\mapsto \texttt{'\&'}\ \texttt{'['}\ \texttt{type}\ \texttt{']'}
\end{aligned}
$$

结构体定义

```
  structDef ⟼ 'struct' id '{' varDeclList '}'
varDeclList ⟼ varDecl (',' varDecl)*
```

```
struct MyStruct {
    x: i32,
    x: [i32;10]
}
struct MyStruct2 {
    x: MyStruct
}
let foo = MyStruct {...}
```

- `x: [i32;10]` → 结构体内可以包含数组字段
- `x: MyStruct` → 结构体内可以包含结构体字段
- `let foo = MyStruct {...}` → 不用考虑struct构造器的问题

左值表达式

- 左值：可以被赋值的对象或可寻址实体

$$\text{lValue} \mapsto \text{id exprSuffix*}$$

$$\text{exprSuffix} \mapsto \text{'.' id} \mid \text{'[' (id} \mid \text{num) ']'}$$

```
x = ...
x.a = ...
x[1] = ...
x.a[1] = ...
x[1].a = ...
```

右值表达式

- 右值：表达式计算得到的值，可以出现在赋值语句右侧
- 左值都是右值的特例，但并非所有右值都是左值。

```
    rValue ⟼ arithExpr | boolExpr
 arithExpr ⟼ (arithExpr ('+' | '-'))? factor
    factor ⟼ (factor ('*' | '/'))? exprUnit
  exprUnit ⟼ num | lValue | fnCall | '&' id
             | '(' arithExpr ')'
       num ⟼ unum | ('-' unum)
```

运算符优先级：与Rust/C语言标准兼容

<table>
<tr><th>优先级</th><th>运算符</th><th>含义</th><th>结合性</th><th>Tea语言限制</th></tr>
<tr><td>8</td><td>-, !</td><td>单目运算符</td><td>右</td><td>'-'后只跟数字，'!'只能跟'('</td></tr>
<tr><td>7</td><td>*, /</td><td>乘除号</td><td>左</td><td></td></tr>
<tr><td>6</td><td>+, -</td><td>加减号</td><td>左</td><td></td></tr>
<tr><td>5</td><td>>, >=, <, <=</td><td>比大小</td><td>左</td><td rowspan="2">不支持连续比较</td></tr>
<tr><td>4</td><td>==, !=</td><td>等价性</td><td>左</td></tr>
<tr><td>3</td><td>&&</td><td>逻辑与</td><td>左</td><td></td></tr>
<tr><td>2</td><td>||</td><td>逻辑或</td><td>左</td><td></td></tr>
<tr><td>1</td><td>=</td><td>赋值</td><td>右</td><td>不支持连续赋值</td></tr>
</table>

https://doc.rust-lang.org/reference/expressions.html\#expression-precedence
https://c-cpp.com/c/language/operator_precedence

布尔表达式

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

```
a>b
(a>b)
!(a>=b)
// a>b>c
// a+b>c+d+e
(a+b)>(c+d+e)
```

- `!(a>=b)` → !后强制使用括号，避免歧义
- `// a>b>c` → 非法，易产生歧义
- `// a+b>c+d+e` → 非法，易产生歧义

函数声明

```
fn foo(a:i32, b:i32);
fn foo(a:&[i32]) -> i32;
```

多参数，无返回值

slice参数，有返回值

```
fnDeclStmt ⟼ 'fn' fnSign ';'
    fnSign ⟼ id '(' paramList? ')'
            |  id '(' paramList? ')' '->' type
 paramList ⟼ varDeclList
```

函数调用

```
std::getch();
let a:[i32;2] = [0;2];
foo(&a);
```



```
      fnCall ⟼ localCall | modPrefixCall
   localCall ⟼ id '(' argList? ')'
modPrefixCall ⟼ modPath '::' id '(' argList? ')'
     argList ⟼ rValue (',' rValue)*
```

函数定义

```
fn foo(a:i32, b:i32) -> i32 {
    return a + b;
}
```

```
fnDef ⟼ 'fn' fnSign '{' stmt* '}'

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

基本语句

$$
\begin{aligned}
\texttt{assignStmt} &\mapsto \texttt{lValue '=' rValue ';'} \\
\texttt{callStmt} &\mapsto \texttt{fnCall ';'} \\
\texttt{retStmt} &\mapsto \texttt{'return' rValue? ';'} \\
\texttt{ifStmt} &\mapsto \texttt{'if' boolExpr '\{' stmt* '\}' ('else' '\{' stmt* '\}')?} \\
\texttt{whileStmt} &\mapsto \texttt{'while' boolExpr '\{' stmt* '\}'} \\
\texttt{breakStmt} &\mapsto \texttt{'break' ';'} \\
\texttt{continueStmt} &\mapsto \texttt{'continue' ';'}
\end{aligned}
$$

强制使用括号，避免歧义

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

```
while a>=b {
    ...
}
```

数字、标识符和注释（正则表达式）

```
       unum  ⟼  [1-9][0-9]*|0
         id  ⟼  [a-z_A-Z] ([a-z_A-Z0-9])*

    comment  ⟼  lineComment | blockComment
lineComment  ⟼  '//' [^\n]* '\n'
blockComment ⟼  '/*' ([^*] | '*' [^/])* '*/'
```

下列代码是否符合Tea语言语法规则？

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

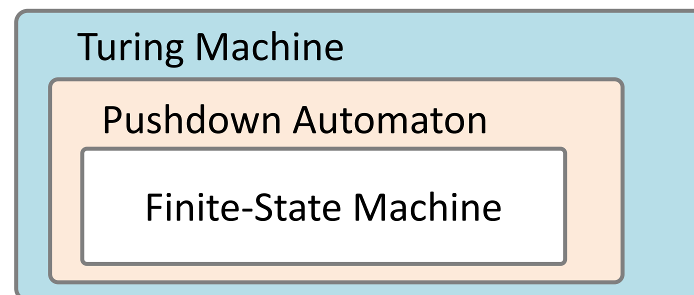
```
fn foo() -> i32 {
    let x:[2];
    return x[0];
}
```

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：
  - $X \rightarrow \gamma$
  - $\gamma \rightarrow \gamma_1$
  - ...
- 特点：右侧的非终结符均可替换为终结符

[1] $S \rightarrow \mathrm{A}|B$
[2] $A \rightarrow (0?1)^*$
[3] $B \rightarrow (1?0)^*$

⇨ $S \rightarrow (0?1)^*|(1?0)^*$

上下文敏感语言

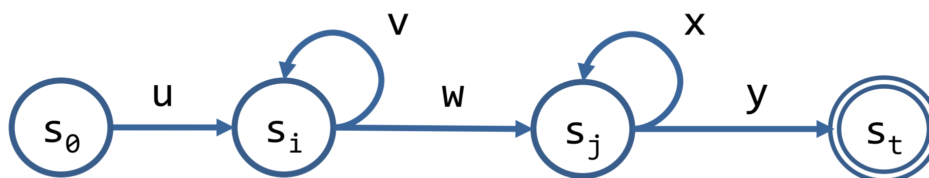
- $L=\{a^n b^n c^n, n>0\}$不是上下文无关语言
- 上下文敏感文法规则形式：$aS \rightarrow \beta$

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

$L=\{a^n b^n c^n, n>0\}$的语法规则

上下文无关语言的泵引理

- 任意长度超过p的句子可以被拆分为$uvwxy$的形式
- v和x重复任意次后得到的句子仍属于该语言：uv^nwx^ny
- 正则属于CFG：$uv^nw\epsilon^n\epsilon$



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) $L = \{a^p | \text{ p is prime}\}$
- 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语法规则吗？
7) 用通用图灵机可以解析任意CFG语法规则吗？
8) 用通用图灵机可以解析任意程序吗？