# 3 上下文无关文法

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 掌握上下文无关文法的基本概念；
- ⋆⋆⋆ 理解上下文无关文法的二义性问题及其消除方法；
- ⋆⋆⋆ 学会使用 EBNF 范式定义上下文无关语言的语法规则；
- ⋆⋆⋆ 理解 Tea 语言的语法规则；
- ⋆ 了解 Chomsky 文法的分类。

## 3.1 上下文无关文法

在上一节中，我们已经使用正则表达式定义了计算器中的标签类型。然而，正则表达式无法进一步刻画计算器表达式的结构，尤其难以处理括号匹配等嵌套结构问题。本节将介绍上下文无关文法（Context-Free Grammar, CFG）。与正则文法相比，CFG 具有更强的表达能力，能够描述更加复杂的语法结构。

**定义 1** (上下文无关文法)**.** 上下文无关文法由一组产生式（或规则）构成。每个产生式的形式为 $X \mapsto \gamma$，其中 $X$ 是一个非终结符，$\gamma$ 是由终结符和非终结符组成的字符串。

规则 3.1 给出了一个使用 CFG 描述合法计算器表达式的示例。该规则从非终结符 `E` 出发，通过不断应用产生式进行展开，能够生成所有合法的计算器表达式，同时不会生成任何非法表达式。

```
[1] E ↦ E '+' E
[2] E ↦ E '-' E
[3] E ↦ E '*' E
[4] E ↦ E '/' E
[5] E ↦ E '^' E                    (3.1)
[6] E ↦ '(' E ')'
[7] E ↦ NUM
[8] NUM ↦ <UNUM>
[9] NUM ↦ '-' <UNUM>
```

需要注意的是，在 CFG 中，每条产生式的左侧必须且只能是一个非终结符，不能包含额外的上下文约束。例如，$aX \mapsto ab$ 和 $bX \mapsto bc$ 的左侧对 $X$ 的展开施加了上下文条件，因此不符合上下文无关文法的定义。

## 3.2 二义性问题和消除

虽然语法规则 3.1 可以覆盖所有合法的计算器表达式，但对于某些表达式，可能存在多种解析方式，从而产生二义性。以算式 1+2*3 为例，图 3.1 展示了两种不同的解析方式。这两棵语法解析树对应的计算结果并不相同，其中仅解析树 1 是符合通常算术语义的正确结果。

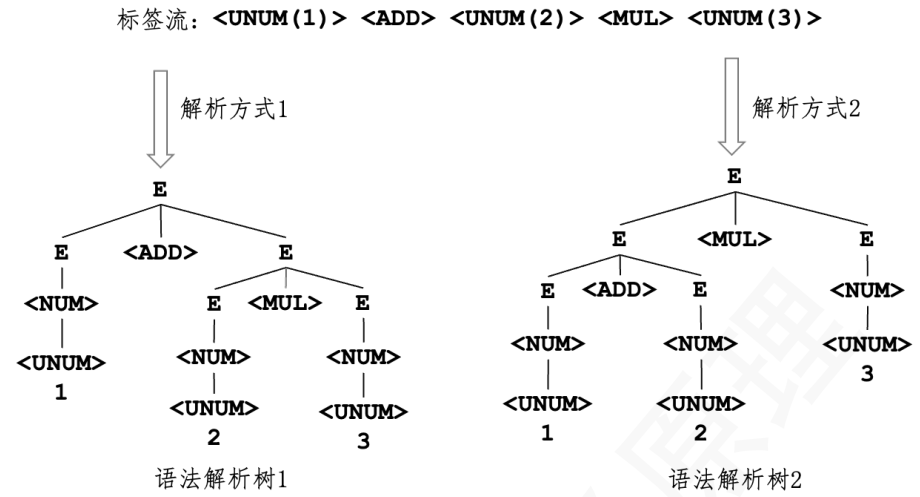


图 3.1: 算式 1+2*3 的两种解析方式

语法规则 3.1 存在二义性的主要原因有两点。首先，该规则未体现运算符的优先级关系；其次，未规定运算符的结合性，从而在处理诸如 “2^3^4” 这样的连续指数运算时可能产生错误的解析。

为消除由优先级引起的二义性，应在语法设计中显式引入运算符优先级，使高优先级运算的结果只能作为低优先级运算的操作数。为解决由结合性引起的二义性，则需要在文法中加以约束：对于左结合运算，递归应作用于左操作数；对于右结合运算，递归应作用于右操作数。

基于上述思路，对规则 3.1 进行改写，得到无二义性的文法规则 3.2 。

```
[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  ↦  '^
```

(3.2)

该改写主要体现在以下几个方面：

- **区分运算符优先级**：通过引入不同层级的运算符集合，明确优先级关系。其中，OP1 表示优先级最低的加减运算，OP2 表示乘除运算，OP3 表示优先级最高的指数运算。
- **分层表示表达式结构**：通过引入分层非终结符（如 E、E1、E2、E3），将不同优先级的表达式逐层刻画。例如，E 表示加减表达式，其操作数为更高优先级的乘除表达式 E1；同时，通过规则 E $\mapsto$ E1 保证文法的完备性与等价性。
- **显式刻画运算符结合性**：在文法中通过递归方向体现结合性。对于左结合运算（如 OP1 和 OP2），采用左递归形式（如 E $\mapsto$ E OP1 E1）；对于右结合运算（如指数运算 OP3），采用右递归形式（如 E2 $\mapsto$ E3 OP3 E2）。

## 3.3 扩展 BNF 范式

由于上下文无关文法的规则形式相对繁琐，在实际应用中通常采用扩展 BNF 范式（EBNF: Extended Backus–Naur Form）[2] 来描述具体的语言语法。EBNF 在基本 CFG 的基础上，引入了可选和闭包等构造，使语法规则更加紧凑且易于理解。

为了与上一节中使用的正则表达式符号保持一致，并提高书写的直观性与便捷性，本文采用表 3.1 所示的符号体系来构造 EBNF 表达式。该表示方法在一定程度上借鉴了 PEG 文法（Parsing Expression Grammar）[3] 中的符号设计，使语法描述更加统一和清晰。

表 3.1: 本文采用的 EBNF 文法构造符号

| 构造方式 | 符号 | 优先级 | 示例 | 含义 |
|---|---|---|---|---|
| 特定字符串 | `' '` | 5 | `'ab'` | 匹配字符串 “`ab`” |
| 可选匹配 | `?` | 4 | $\alpha?$ | 匹配 $\alpha$ 或空串 $\epsilon$ |
| 闭包 | `*` | 4 | $\alpha*$ | 匹配 $\alpha$ 的零次或多次重复（$\geq 0$） |
| 正闭包 | `+` | 4 | $\alpha+$ | 匹配 $\alpha$ 的一次或多次重复（$\geq 1$） |
| 排除 | `-` | 3 | $\alpha - \beta$ | 匹配 $\alpha$ 但不匹配 $\beta$（$\alpha, \beta$ 为正则表达式） |
| 连接 | | 2 | $\alpha\beta$ | 依次匹配 $\alpha$ 和 $\beta$ |
| 选择 | `\|` | 1 | $\alpha\|\beta$ | 匹配 $\alpha$ 或 $\beta$ |

基于上述表示方法，语法规则 3.3 使用 EBNF 对规则 3.2 进行了等价改写。相比原始 CFG，该表示形式更加简洁，并显著提升了语法规则的可读性与表达能力。因此，在实际编程语言的语法定义中，EBNF 是更为常用的描述方式。

```
    E ↦ Factor (('+'|'-') Factor)*
Factor ↦ Power (('*'|'/') Power)*
 Power ↦ Value ('^ Power)?
 Value ↦ <UNUM> | '-' <UNUM> | '(' E ')'
```
(3.3)

需要注意的是，上述改写不仅简化了规则形式，同时仍然保持了运算符的优先级与结合性：加减运算为最低优先级且左结合，乘除运算次之且左结合，指数运算优先级最高且为右结合。

上下文无关文法相关技术已经发展得较为成熟，存在许多实用的语法分析工具，例如 Bison[1]、ANTLR[2] 以及 Pest[3]。其中，Bison 通常与词法分析器（如 Flex）配合使用，ANTLR 提供了较为完善的语法分析框架，而 Pest 则是一种基于 PEG 文法的解析器生成工具，语法形式与本文所采用的 EBNF 表示方法较为接近。在这些工具中，用户通常可以通过单独声明运算符的优先级和结合性来消解二义性，而无需在文法规则中显式分层，从而降低语法描述的复杂性，使语法描述更加直观、紧凑。

[1] Bison: `https://www.gnu.org/software/bison/`

[2] ANTLR: `https://github.com/antlr/antlr4`

[3] Pest: `https://pest.rs/`

## 3.4 Tea 语法规则

Tea 语言（Teaching Programming Language）是一门面向编译原理课程设计的教学语言。其语法设计可视为 Rust 的子集，支持类型缺省推导，但不包含 Rust 的所有权和借用检查等高级语义特性。

图 3.2 展示了一段使用 Tea 编写的阶乘函数示例代码。可以看到，该语言在函数定义、条件语句以及表达式结构等方面具有较强的直观性，同时保持了良好的结构化特征，便于进行语法分析。

```
fn factorial(n:i32) -> i32 {
    if n == 0 || n == 1 {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

fn main() -> i32 {
    let r = factorial(n);
    return r;
}
```

图 3.2: Tea 代码样例：阶乘函数

接下来，我们将在上述示例的基础上，使用扩展 BNF 范式（EBNF）对 Tea 语言的核心语法规则进行形式化定义，从而为后续的语法分析与编译实现奠定基础。

### 3.4.1 运算符与优先级

Tea 语言中的运算符及其特性如表 3.2 所示。为了保持与常见语言的兼容性，这些运算符的优先级和结合性设置与 Rust 官方规范一致[4]，同时与 C 语言标准一致[5]。

表 3.2: Tea 中的运算符及优先级

| 优先级（C） | 运算符 | 描述 | 结合性（C） | Tea 使用限制 |
|---|---|---|---|---|
| 8 | `-`, `!` | 单目运算符：负号、逻辑非 | 右 | `-` 后仅允许跟数字，`!` 后仅允许跟括号 |
| 7 | `*`, `/` | 双目运算符：乘除法 | 左 | — |
| 6 | `+`, `-` | 双目运算符：加减法 | 左 | — |
| 5 | `>`, `>=`, `<`, `<=` | 比较运算符：大小比较 | 左 | 不支持连续比较 |
| 4 | `==`, `!=` | 比较运算符：等值判断 | 左 | 不支持连续比较 |
| 3 | `&&` | 逻辑与运算 | 左 | — |
| 2 | `\|\|` | 逻辑或运算 | 左 | — |
| 1 | `=` | 赋值运算 | 右 | 不支持连续赋值 |

为了实现简化与便于解析，Tea 在运算符支持和使用方式上相比 Rust 做了一些限制。例如，Tea 不支持位运算，也不允许连续赋值（如 `a = b = c`）。这些限制将在后续的具体语法规则定义中体现。

[4]Rust 运算符优先级与结合性：https://doc.rust-lang.org/reference/expressions.html#expression-precedence

[5]C 语言运算符优先级：https://c-cpp.com/c/language/operator_precedence

### 3.4.2 代码基本组成

Tea 程序由若干语句和定义组成，包括模块引用、变量声明、函数声明与定义、结构体定义以及注释等。语法规则如下：

$$\texttt{program} \ \mapsto \ \texttt{(useStmt | varDeclStmt | fnDeclStmt | fnDef | structDef | comment | ';')}^{*} \tag{3.4}$$

### 3.4.3 代码模块引用

模块引用用于导入其它模块的定义，语法规则如下：

$$\begin{aligned} \texttt{useStmt} &\ \mapsto \ \texttt{'use' modPath ';'} \\ \texttt{modPath} &\ \mapsto \ \texttt{id ('::' id)}^{*} \end{aligned} \tag{3.5}$$

### 3.4.4 变量声明和定义

变量声明与定义是程序中存储数据的基础。Tea 语言支持标量和数组类型的声明，类型可以缺省或显式指定，并且可以在声明时进行初始化。但不允许在同一语句中声明多个变量或数组对象。

$$\begin{aligned} \texttt{varDeclStmt} &\ \mapsto \ \texttt{'let' (varDecl | varDef) ';'} \\ \texttt{varDecl} &\ \mapsto \ \texttt{scalarDecl | typedScalarDecl | arrayDecl | typedArrayDecl} \\ \texttt{scalarDecl} &\ \mapsto \ \texttt{id} \\ \texttt{typedScalarDecl} &\ \mapsto \ \texttt{id ':' type} \\ \texttt{arrayDecl} &\ \mapsto \ \texttt{id ':' '[' num ']'} \\ \texttt{typedArrayDecl} &\ \mapsto \ \texttt{id ':' '[' type ';' num ']'} \\ \texttt{varDef} &\ \mapsto \ \texttt{scalarDecl '=' rValue} \\ &\ \ \texttt{|} \ \ \texttt{typedScalarDecl '=' rValue} \\ &\ \ \texttt{|} \ \ \texttt{arrayDecl '=' arrayInit} \\ &\ \ \texttt{|} \ \ \texttt{typedArrayDecl '=' arrayInit} \\ \texttt{arrayInit} &\ \mapsto \ \texttt{'[' rValue (',' rValue)}^{*}\texttt{ ']'} \\ &\ \ \texttt{|} \ \ \texttt{'[' rValue ';' num ']'} \end{aligned} \tag{3.6}$$

由于标量类型和结构体类型在 Tea 语言中的语法规则相似，这里统一使用 `typedScalarDecl` 来表示标量或结构体类型的变量声明。

### 3.4.5 类型

Tea 语言支持原始类型和结构体类型，结构体可包含标量或数组字段：

$$\begin{aligned} \texttt{type} &\ \mapsto \ \texttt{primitiveType | structType | sliceType} \\ \texttt{primitiveType} &\ \mapsto \ \texttt{i32} \\ \texttt{structType} &\ \mapsto \ \texttt{id} \\ \texttt{sliceType} &\ \mapsto \ \texttt{'\&' '[' type ']'} \\ \texttt{structDef} &\ \mapsto \ \texttt{'struct' id '\{' varDeclList '\}'} \\ \texttt{varDeclList} &\ \mapsto \ \texttt{varDecl (',' varDecl)}^{*} \end{aligned} \tag{3.7}$$

由于指针和引用的复杂性，Tea 语言编译器暂未考虑在变量声明时直接使用引用类型。上述定义中的 `sliceType` 仅用于函数参数传递。另外，当前语法未考虑结构体构造器的设计，因此在使用结构体变量时，不能通过整体初始化的方式赋值，只能对各个字段分别进行赋值。

### 3.4.6 左右值

左值表示可以被赋值的对象或可寻址实体，例如变量、数组元素或结构体字段；右值表示表达式计算得到的值，可以用于运算、函数调用参数或布尔判断。所有左值都是右值的一种特殊情况，但并非所有右值都是左值。形式化定义如下：

```
    lValue ↦ id exprSuffix*
exprSuffix ↦ '.' id | id '[' (id | num) ']'
    rValue ↦ arithExpr | boolExpr
 arithExpr ↦ (arithExpr ('+' | '-'))? factor
    factor ↦ (factor ('*' | '/'))? exprUnit
  exprUnit ↦ num | lValue | fnCall | '&' id | '(' arithExpr ')'
  boolExpr ↦ (boolExpr '||')? andExpr
   andExpr ↦ (andExpr '&&')? boolUnit
  boolUnit ↦ cmpExpr | '!'? '(' boolExpr ')'
   cmpExpr ↦ exprUnit ('==' | '!=' | '>' | '>=' | '<' | '<=') exprUnit
```
(3.8)

### 3.4.7 函数声明和调用

函数声明用于标明函数签名，函数调用用于执行函数逻辑。在 Tea 语言中，函数参数支持数组引用类型的传递。规则定义如下：

```
   fnDeclStmt ↦ 'fn' fnSign ';'
       fnSign ↦ id '(' paramList? ')'
              |  id '(' paramList? ')' '->' type
    paramList ↦ varDeclList
       fnCall ↦ localCall | modPrefixCall
    localCall ↦ id '(' argList? ')'
modPrefixCall ↦ modPath '::' id '(' argList? ')'
      argList ↦ rValue (',' rValue)*
```
(3.9)

### 3.4.8 函数定义

函数定义包含函数体及语句列表：

$$
\begin{aligned}
\texttt{fnDef} &\mapsto \texttt{fn fnSign '\{' stmt}^{*}\texttt{ '\}'} \\
\texttt{stmt} &\mapsto \texttt{varDeclStmt | assignStmt | callStmt | retStmt | ifStmt} \\
&\quad \texttt{| whileStmt | breakStmt | continueStmt | ';'} \\
\texttt{assignStmt} &\mapsto \texttt{lValue '=' rValue ';'} \\
\texttt{callStmt} &\mapsto \texttt{fnCall ';'} \\
\texttt{retStmt} &\mapsto \texttt{'return' rValue? ';'} \\
\texttt{ifStmt} &\mapsto \texttt{'if' boolExpr '\{' stmt}^{*}\texttt{ '\}' ('else' '\{' stmt}^{*}\texttt{ '\}')?} \\
\texttt{whileStmt} &\mapsto \texttt{'while' boolExpr '\{' stmt}^{*}\texttt{ '\}'} \\
\texttt{breakStmt} &\mapsto \texttt{'break' ';'} \\
\texttt{continueStmt} &\mapsto \texttt{'continue' ';'}
\end{aligned}
\tag{3.10}
$$

### 3.4.9 标识符和数字

我们用正则表达式定义标识符和数字。

$$
\begin{aligned}
\texttt{id} &\mapsto \texttt{[a-z\_A-Z][a-z\_A-Z0-9]*} \\
\texttt{num} &\mapsto \texttt{unum | ('-' unum)} \\
\texttt{unum} &\mapsto \texttt{[1-9][0-9]* | 0}
\end{aligned}
\tag{3.11}
$$

### 3.4.10 代码注释

Tea 支持单行和多行注释，形式化定义如下：

$$
\begin{aligned}
\texttt{comment} &\mapsto \texttt{lineComment | blockComment} \\
\texttt{lineComment} &\mapsto \texttt{'//' [\^{}\textbackslash n]}^{*}\texttt{ '\textbackslash n'} \\
\texttt{blockComment} &\mapsto \texttt{'/*' ([\^{}*] | '*' [\^{}/])}^{*}\texttt{ '*/'}
\end{aligned}
\tag{3.12}
$$

## 3.5 文法能力分类

根据表示能力的不同，[1] 将文法分为四个等级。如表 3.3 所示，正则文法（3 型）的表示能力最弱，无法表示像 $a^n b^n$ $(n \in N)$ 这类要求字符出现次数相同的语言。所有正则语言都可以通过上下文无关文法表示，即产生式右侧允许包含非终结符。由于 CFG 不考虑上下文，它在编程语言语法设计中无法直接保证变量定义与使用的类型一致性。因此，类型检查等语义规则至少需要借助上下文敏感文法（1 型文法）来描述。而常见通用编程语言，如 C、Rust 等，其完整语法与语义规则属于 0 型文法（递归可枚举文法）。

表 3.3: Chomsky 文法分类

| 类型 | 计算模型 | 规则形式 | 语言示例 |
|---|---|---|---|
| 0 型：递归枚举 | 图灵机 | - | 普通程序 |
| 1 型：上下文敏感 | 线性有界图灵机 | $\alpha S \to \beta$ | $a^n b^n c^n,\ n \in N$ |
| 2 型：上下文无关 | 下推自动机 | $S \to \beta$ | $a^n b^n,\ n \in N$ |
| 3 型：正则 | 有穷自动机 | $S \to a \mid b$ | $a^n,\ n \in N$ |

理论上，每一级文法都可以用来描述下一级文法的规则。例如，CFG 可以用来定义正则表达式的语法，并进一步解析任意正则表达式；同理，0 型文法可以用于描述 1 型文法的规则，从而应用于类型推导或类型检查等任务。

## 参考文献


[1] Noam Chomsky. “On certain formal properties of grammars.” *Information and Control 2*, 1959.

[2] ISO/IEC 14977:1996 Information Technology-Syntactic Metalanguage-Extended BNF.

[3] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation.” *In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

# 练习

1) 判断下列两组 CFG 规则是否属于正则文法：

   (a) S $\mapsto$ 0S1S|1S0S|$\epsilon$

   (b) S $\mapsto$ aT|b, T $\mapsto$ c|$\epsilon$

2) 以开发一个能够接收任意正则表达式并生成对应正则匹配器的工具为目标，设计用于解析正则表达式的语法规则。

3) Scheme 是一种函数式编程语言，属于 Lisp 语言家族的分支。部分语法规则如下表所示。请以编写一个 Scheme 语法解析器为目标，设计“函数定义”的上下文无关语法规则。只需覆盖 factorial 函数示例，并保证规则逻辑自洽，无需全面实现全部 Scheme 特性。

表 3.4: Scheme 部分语法规则

| 规则 | Scheme 代码示例 | 规则描述 | 含义 |
|---|---|---|---|
| 算数运算 | `(+ x 1)` | 支持 +、-、*、/ | x + 1 |
| 变量赋值 | `(define y (+ x 1))` | `define` 变量名 右值表达式 | y = x+1 |
| 函数定义 | `(define [foo x y]`<br>`  (define z 2)`<br>`  (- (+ x y) z))` | `define` [函数名 参数] 函数体 | 定义函数 foo(x,y)<br>返回值：x + y - z |
| 函数调用 | `(foo 1 2)` | 函数名 参数 1 参数 2 .. | foo(1,2) |
| 条件语句 | `(if (= n 1)`<br>`  1`<br>`  0)` | `if` 条件 分支 1 分支 2 | if(n==1) 1 else 0 |

```
(define [factorial n]
    (if (= n 1)
        n
        (* n (factorial (- n 1)))
    )
)
```

代码 3.1: Scheme 代码示例：阶乘函数