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
- 掌握上下文无关文法的二义性问题和消除方法
- 学会使用扩展 BNF 范式定义上下文无关语言
- 了解 Chomsky 文法分类

3.1 上下文无关文法

在上一节课中，我们已经使用正则表达式定义了计算器中的标签类型，但无法使用正则表达式进一步定义计算器表达式。其中一个主要原因是正则表达式无法解决表示括号的匹配问题。本节学习的上下文无关文法是一种比正则文法表达能力更强的工具。

定义 1 (上下文无关文法（CFG: Context-free Gramar）)**.** 由一系列形如 $X \mapsto \gamma$ 的规则或产生式组成，其中 X 是一个非终结符，γ 是由终结符或非终结符组成的字符串。

规则 3.1尝试使用 CFG 定义合法的计算器表达式，从 `E` 开始应用不同的语法规则组合层层展开应该可以得到所有合法的计算器表达式。

$$
\begin{aligned}
\texttt{E} &\mapsto \texttt{E}\ \ \texttt{'+'}\ \ \texttt{E} \\
\texttt{E} &\mapsto \texttt{E}\ \ \texttt{'-'}\ \ \texttt{E} \\
\texttt{E} &\mapsto \texttt{E}\ \ \texttt{'*'}\ \ \texttt{E} \\
\texttt{E} &\mapsto \texttt{E}\ \ \texttt{'/'}\ \ \texttt{E} \\
\texttt{E} &\mapsto \texttt{E}\ \ \texttt{'\^{}'}\ \ \texttt{E} \\
\texttt{E} &\mapsto \texttt{'('}\ \ \texttt{E}\ \ \texttt{')'} \\
\texttt{E} &\mapsto \texttt{NUM} \\
\texttt{NUM} &\mapsto \texttt{<UNUM>} \\
\texttt{NUM} &\mapsto \texttt{'-'}\ \ \texttt{<UNUM>}
\end{aligned}
\tag{3.1}
$$

注意，CFG 要求每一条文法规则的左侧只能有一个非终结符，不应含其它任何限制条件。例如，$aX \mapsto ab$，$bX \mapsto bc$ 的左侧对于如何展开 X 有条件限制，因此不属于 CFG。

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同时存在多种解析方式，引起歧义。以算式 $1+2*3$ 为例，存在图 3.1中的两种解析方式。这两棵语法解析树对应的计算结果不同，只有解析树 1 是正确的。

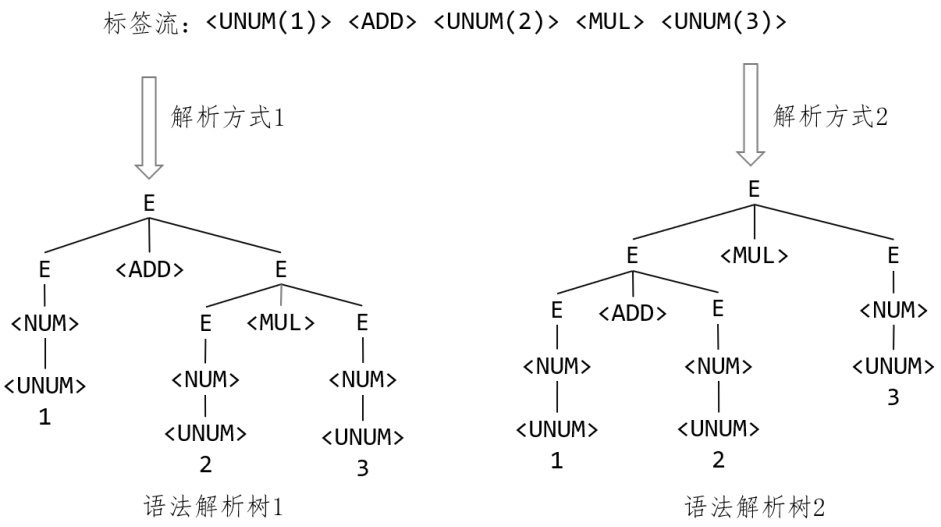

图 3.1: 语法解析：算式 1+2*3

语法规则 3.1存在二义性的主要原因有两点：一是未考虑运算符优先级；二是未考虑结合性，解析 2^3^4 这种连续的指数运算时亦可能会出错。为消除优先级带来的二义性问题，应在语法规则设计时引入优先级关系，即在不考虑括号的情况下使得低优先级运算的结果只能作为高优先级运算的操作数。为消除结合性带来的二义性问题，应在语法规则中强制左结合运算只允许其左侧操作数递归展开，右结合运算只允许其右侧操作数递归展开。

$$
\begin{aligned}
\texttt{E} &\mapsto \texttt{E OP1 E1} \\
\texttt{E} &\mapsto \texttt{E1} \\
\texttt{E1} &\mapsto \texttt{E1 OP2 E2} \\
\texttt{E1} &\mapsto \texttt{E2} \\
\texttt{E2} &\mapsto \texttt{E3 OP3 E2} \\
\texttt{E2} &\mapsto \texttt{E3} \\
\texttt{E3} &\mapsto \texttt{NUM} \\
\texttt{E3} &\mapsto \texttt{'('\ \ E\ \ ')'} \\
\texttt{NUM} &\mapsto \texttt{<UNUM>} \\
\texttt{NUM} &\mapsto \texttt{'-'\ \ <UNUM>} \\
\texttt{OP1} &\mapsto \texttt{'+'} \\
\texttt{OP1} &\mapsto \texttt{'-'} \\
\texttt{OP2} &\mapsto \texttt{'*'} \\
\texttt{OP2} &\mapsto \texttt{'/'} \\
\texttt{OP3} &\mapsto \texttt{'\^{}'}
\end{aligned}
\tag{3.2}
$$

基于上述思路对规则 3.1进行改写，可得到无二义性的规则 3.2。具体改写过程主要包括以下几点：

- **区分运算符优先级**：使用 `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。
- **区分操作数和结果优先级**：如使用 `E` 表示加减运算的运算结果，它的操作数可以是优先级更高的乘

除运算的结果 E1，也可以是同等优先级的加减运算结果 E；为保证规则等价性，E 也可以直接仅对应乘除运算 E1。

- **使语法规则满足结合性**：如运算符 OP1 是左结合的，则其规则应为左递归形式（E ↦ E OP1 E1）；运算符 OP3 是右结合的，其规则应为右递归形式（E2 ↦ E3 OP3 E2）。

3.3 扩展 BNF 范式

由于 CFG 写起来比较复杂，我们一般使用 EBNF 范式（EBNF: Extended Backus-Naur Form）[2] 来描述具体的上下文无关语言规则，即增加闭包和选择等构造方式。为了与上一节学习的正则表达式符号兼容且易于书写，我们使用表 3.1中的符号构造 EBNF。这种表示方法参考了 PEG 文法 [3] 中的构造符号设计。

表 3.1: 扩展 BNF 范式：本文采用的文法构造符号。

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'
字符通配符	.	5	.	匹配单个任意字符
字符集合	[]	5	[a-z]	匹配任意 a-z 之间的字符
可选匹配	?	4	$\alpha?$	匹配任意非终结符或字符串 α，或为 ϵ
闭包	*	4	$\alpha*$	匹配连续若干个（≥ 0）α
正闭包	+	4	$\alpha+$	匹配连续若干个（≥ 1）α
非	!/^	3	$!\alpha$	匹配 α 之外的任意符号
连接		2	$\alpha\beta$	连续匹配 α 和 β
选择	|	1	$\alpha\|\beta$	匹配 α 或 β

语法规则 3.3使用 EBNF 对规则 3.2进行了改写，新规则更为简洁，且易读性强。因此我们在定义编程语言语法规则时一般采用这种表示形式。

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

在 Bison[1]等实际语法解析工具中，用户可以不必在规则中区分运算优先级，而是通过另外声明优先级的方式筛选错误的解析方式，降低规则描述的复杂性。

3.4 TeaPL 语法规则

TeaPL 语言（Teaching Programming Language）是为编译课程教学设计的一门语言。该语言在使用方式上与 C 语言相似，但在一些语法设计上采用了 Rust 等新兴语言的形式，主要是便于语法解析和缺省类型实现等考虑。图 3.2展示了一段用 TeaPL 编写的阶乘函数代码样例。下面我们采用 EBNF 对其语法标准进行定义。

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

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

fn main() -> int {
    let r = factorial(n);
    ret r;
}
```

图 3.2: TeaPL 代码样例

3.4.1 运算符和优先级

TeaPL 中采用的运算符以及每种运算符的特性如表 3.2所示，该运算符优先级和结合性设置与 C 语言标准保持兼容[2]。

表 3.2: TeaPL 中的运算符和优先级。

优先级（C）	运算符	描述	结合性（C）	TeaPL 使用限制
8	-, !	单目运算符：负号、逻辑非	右	-后只允许跟数字, ! 后只允许跟括号
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：比大小	左	比较对象不支持其它比较或逻辑运算
4	==, !=	比较运算符：等价性	左	比较对象不支持其它比较或逻辑运算
3	&&	逻辑运算符：与	左	
2	||	逻辑运算符：或	左	
1	=	赋值	右	不支持连续赋值

3.4.2 代码基本组成

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

3.4.3 标识符和数字

$$\texttt{id} \mapsto \texttt{[a-z\_A-Z][a-z\_A-Z0-9]*} \tag{3.5}$$

$$\texttt{num} \mapsto \texttt{unum} \mid (\texttt{'-'}\ \texttt{unum}) \tag{3.6}$$

$$\texttt{unum} \mapsto \texttt{[1-9][0-9]+} \mid \texttt{0} \tag{3.7}$$

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

3.4.4 变量声明

```
varDeclStmt ↦ 'let' (varDecl | varDef) ';'                                    (3.8)
    varDecl ↦ id (':' type)? | id '[' (id | num) ']' (':' type)?              (3.9)
     varDef ↦ id (':' type)? '=' rightVal                                     (3.10)
            | id '[' (id | num) ']' (':' type)? '=' '{' num '}'               (3.11)
```

3.4.5 类型

```
        type ↦ primitiveType | structType                                     (3.12)
primitiveType ↦ int                                                           (3.13)
   structType ↦ id                                                            (3.14)
    structDef ↦ 'struct' id '{' fieldDecl (',' fieldDecl)* '}'                (3.15)
    fieldDecl ↦ id ':' type | id '[' (id | num) ']' ':' type                  (3.16)
                                                                              (3.17)
```

3.4.6 右值

```
 rightVal ↦ arithExpr                                                         (3.18)
arithExpr ↦ (arithExpr ('+' | '-'))? factor                                   (3.19)
   factor ↦ (factor ('*' | '/'))? exprUnit                                    (3.20)
 exprUnit ↦ num | id | fnCall | '(' rightVal ')' | id '.' id | id '[' (id | num) ']'
                                                                              (3.21)
```

3.4.7 函数声明

```
fnDeclStmt ↦ 'fn' fnSign ';'                                                  (3.22)
    fnSign ↦ id '(' params? ')' '->' type?                                    (3.23)
    params ↦ id ':' type (',' id ':' type)*                                   (3.24)
```

3.4.8 函数定义

```
      fnDef ↦ fn fnSign codeBlock                                          (3.25)
  codeBlock ↦ '{' stmt* '}'                                                (3.26)
       stmt ↦ varDeclStmt | assignStmt | callStmt | retStmt | ifStmt       (3.27)
            | whileStmt | breakStmt | continueStmt | ';'
 assignStmt ↦ leftVal '=' rightVal ';'                                     (3.28)
    leftVal ↦ id | id '[' (num | id) ']' | id '.' id                       (3.29)
   callStmt ↦ fnCall ';'                                                   (3.30)
     fnCall ↦ id '(' (rightVal (, rightVal)*)? ')'                         (3.31)
    retStmt ↦ 'ret' rightVal? ';'                                          (3.32)
     ifStmt ↦ 'if' '(' boolExpr ')' codeBlock (else codeBlock)?            (3.33)
  whileStmt ↦ 'while' '(' boolExpr ')' codeBlock                           (3.34)
  breakStmt ↦ 'break' ';'                                                  (3.35)
continueStmt ↦ 'continue' ';'                                              (3.36)
```

3.4.9 布尔表达式

```
boolExpr ↦ (boolExpr '||')? andExpr                                        (3.37)
 andExpr ↦ (andExpr '&&')? boolUnit                                        (3.38)
boolUnit ↦ cmpExpr | '!(' cmpExpr ')' | '!'? '(' boolExpr ')'              (3.39)
 cmpExpr ↦ exprUnit ('==' | '!=' | '>' | '>=' | '<' | '<=') exprUnit       (3.40)
```

3.4.10 代码注释

```
comment ↦ '//' (!newline)* newline | '/*' (!'*/')*' */'                   (3.41)
newline ↦ '\n'                                                             (3.42)
```

3.5 文法能力分类

根据表示能力不同，Chomsky 将文法分为 4 个等级 [1]。如表 3.3所示，正则文法是表示能力最弱的文法，无法表示 $a^n b^n$（$n \in N$）这种要求两个字符出现任意相同次数的语言；所有正则文法都可以采用 CFG 表示，即产生式右侧不含非终结符的情况。由于不考虑上下文，CFG 应用于编程语言语法规则设计时无法满足变量定义和使用时的类型一致性要求。因此，类型检查规则设计需要使用上下文敏感文法。我们经常使用的 C、Python 等通用编程语言一般是 0 型文法。

理论上，每一级文法都可以用于定义下一级文法的描述规则。如采用 CFG 可以定义正则表达式描述规则，并进一步对任意正则表达式进行解析。同理，0 型文法可以定义 1 型文法的描述规则并用于类型推导或类型检查这类任务。

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	$S \to a \vert b$	$a^n, n \in N$

练习

1. 以开发正则表达式工具（输入任意的正则表达式，生成对应的正则匹配器）为目标，设计用于解析正则表达式的 CFG 规则。
2. 使用 EBNF 对上述语法规则进行改写。
3. 下列两组 CFG 规则是否属于正则文法?
   (a) `S` $\mapsto$ `0S1S|1S0S|`ϵ
   (b) `S` $\mapsto$ `aT|b`, `T` $\mapsto$ `c|`ϵ

Bibliography

[1] Noam Chomsky. "On certain formal properties of grammars." Information and control 2, no. 2 (1959): 137-167.

[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, pp. 111-122. 2004.