十分钟速成魔术回路

Chuigda Whitegive
作者
第七通用设计局

Cousin Ze
作者
第七通用设计局

前言

已经是 2025 年了。~~随着 TypeScript 和 Rust 等工具在 JavaScript Kiddies 圈子间的普及，~~现在基本上是个人就能在类型论方面胡吹两句。作为一名野心勃勃的计算机科学家[1]，你当然不甘心屈居人下，希望掌握一些普通人~~尤其是 JavaScript Kiddies~~ 绝对看不懂的东西，好让你在跟这些图样图森破的人谈笑风生的时候一眼看出他们的暴论，并从更高的理论层面把他们批判一番。然而，当你打开各种各样关于程序设计语言理论的书籍、博客和论文的时候，像这样的: $e := v \mid \lambda x \to e \mid e\, e'$，这样的: $\lambda f.\lambda x.f(fx)$ 还有这样的: $\frac{\Gamma \vdash \tau}{\Gamma \vdash x:\tau}$ 公式却属实让你猛打退堂鼓，怀疑自己不是这块料。同时你又实在不肯放下面子去看儿童读物。

如果你遇到了这种情况，这说明你对程序设计语言领域常见的行业黑话缺乏理解。用菌类的话来说，就是你缺少基本的魔术回路 *(Magic Circuit)*。别担心，本文正是要帮助你在十分钟内速成魔术回路，从而让你有能力阅读和理解程序设计语言理论领域的入门级资料（比如这一篇[2]）。

1. 巴科斯-瑙尔表示法

1.1. 这是啥

我们从最简单的东西——巴科斯-瑙尔表示法 *(Backus-Naur Form, BNF)*[3][4] 开始。如果这玩意你看不懂，那你编译原理肯定逃课了。不过国内编译原理课程在讲到 BNF 的时候大多都在死磕傻逼“四类文法”和“推导产生”的定义，听的人昏昏欲睡，你逃课也是可以理解的。

BNF 起初主要是用来表示程序设计语言的语法 *(syntax)*的，当然也可以用来表示其他的东西。它的基本形式是这样：

$$\begin{aligned} \text{一个东西} ::= & \text{ 另一个东西} \\ \mid & \text{ 再一个东西} \\ \mid & \text{ ...} \end{aligned}$$

或者有时候作者会把所有东西塞在一行里：

$$\text{一个东西} ::= \text{另一个东西} \mid \text{再一个东西} \mid \text{...}$$

这种形式很好理解，你只要把 ::= 读作“可以是”，把 | 读作“或者”就行了。举个例子：

$$\begin{aligned} \text{expr} ::= & \text{ value} \\ \mid & \text{ expr + value} \\ \mid & \text{ expr - value} \end{aligned}$$

你可以这么读：“一个 expr 可以是一个 value，或者是一个 expr 加上一个 value，或者是一个 expr 减去一个 value。”

---

[1] 甭管是不是，至少你应该这么称呼自己，而不是管自己叫程序员。
[2] 我们也翻译了这篇论文: https://chuigda.doki7.club/research/lambda-pi.pdf。
[3] 又称巴科斯范式 *(Backs Normal Form)*，但高德纳认为它不是一种范式 *(Normal form)*。我也这么认为。
[4] 顺带一提，我们把 *Normal Form* 一词翻译为 正规形式。

如你所见，::= 符号右边的东西可以包含 ::= 左边的那个符号，因此 BNF 可以通过这种方式来定义*递归的 (recursive)* 结构。如果你对递归没概念（那你翘的课可能有亿点多），那也没关系，接下来马上你就看到递归有什么用了。

1.2. 怎么用

这种记法最直接的用途就是以一种比较正式的方式介绍“一个 A 可以是 B，可以是 C……”这种规则。

比如说，你的 C 语言老师说：

> 一条 C 语言语句可以是一个 `if` 语句、可以是一个 `while` 语句、可以是一个语句块[5]……
> 一个 `if` 语句以关键字 `if` 开头，后面跟着一对圆括号，括号里是一个表达式，接着是另一条语句……
>
> — 你的 C 语言老师

显然，用语言来描述不但繁琐容易出错，而且逼格不够容易被人看不起。你写成 BNF，一下子就高大上了：

```
stmt       ::= if-stmt[6]
             | while-stmt
             | block-stmt
             | other-stmt

if-stmt    ::= if ( expr ) stmt
             | if ( expr ) stmt else stmt

while-stmt ::= while ( expr ) stmt

block-stmt ::= { stmt-list }

stmt-list  ::= stmt-list stmt
             | ε
```

在 BNF 里，希腊字母 ε（读作“埃普西隆”）通常表示“空字符串”，也就是“什么都没有”。stmt-list ::= ε 这句话也就是再说“一个语句列表可以是空的，里面没有任何语句”。

这个 BNF 有几个有意思的地方：stmt 的右边有 if-stmt、while-stmt，而 if-stmt、while-stmt 的右边又都有 stmt。这样一来，你就可以用 stmt 来表示任意复杂的嵌套语句结构。比如说，对于这段程序：

```
while (x --> 0)
  if (x % 2 == 0)
    if (x % 3 == 0)
      do_something;
```

[5]实际上 C 语言的语法规则比这复杂得多，这里只是为了演示 BNF 的用法，所以对规则作了简化。
[6]再次强调，这是用于演示 BNF 用法的**简化规则**。如果你拿这玩意去跟人讨论 C 语言，必挨语言律师的骂。

你可以这样展开 stmt，让它匹配上面这段代码：

$$
\begin{aligned}
\text{stmt} &= \text{while-stmt} \\
&= \texttt{while (} \text{expr} \texttt{)} \ \text{stmt}_1 \\
&= \texttt{while (} \text{expr} \texttt{)} \ \text{if-stmt}_1 \\
&= \texttt{while (} \text{expr} \texttt{)} \ [\texttt{if (} \text{expr} \texttt{)} \ \text{stmt}_2]_{\text{if-stmt1}} \\
&= \texttt{while (} \text{expr} \texttt{)} \ [\texttt{if (} \text{expr} \texttt{)} \ \text{if-stmt}_2]_{\text{if-stmt1}} \\
&= \texttt{while (} \text{expr} \texttt{)} \ \left[\texttt{if (} \text{expr} \texttt{)} \ [\texttt{if (} \text{expr} \texttt{)} \ \text{stmt}_3]_{\text{if-stmt2}}\right]_{\text{if-stmt1}} \\
&= \texttt{while (} \text{expr} \texttt{)} \ \left[\texttt{if (} \text{expr} \texttt{)} \ [\texttt{if (} \text{expr} \texttt{)} \ \text{other-stmt}_3]_{\text{if-stmt2}}\right]_{\text{if-stmt1}}
\end{aligned}
$$

而 stmt-list 的右边有一条规则是 stmt-list stmt，用这种方式，我们可以用 stmt-list 表示任意多条语句的序列，比如说，对于这段程序：

```
// {
  while (x --> 0) y += 2 * x;
  while (y --> 0) z += 3 * y;
  if (z == magic_number) do_magic();
// }
```

你可以这样展开 stmt-list，让它匹配上面这段代码：

$$
\begin{aligned}
\text{stmt-list} &= \text{stmt-list}_1 \ \text{stmt}_1 \\
&= [\text{stmt-list}_2 \ \text{stmt}_2]_{\text{stmt-list1}} \ \text{stmt}_1 \\
&= \left[[\text{stmt-list}_3 \ \text{stmt}_3]_{\text{stmt-list2}} \ \text{stmt}_2\right]_{\text{stmt-list1}} \ \text{stmt}_1 \\
&= \left[[\varepsilon_{\text{stmt-list3}} \ \text{stmt}_3]_{\text{stmt-list2}} \ \text{stmt}_2\right]_{\text{stmt-list1}} \ \text{stmt}_1 \\
&= \varepsilon \ \text{stmt}_3 \ \text{stmt}_2 \ \text{stmt}_1 \\
&= \varepsilon \ \text{while-stmt}_3 \ \text{while-stmt}_2 \ \text{if-stmt}_1 \\
&= \text{while-stmt}_3 \ \text{while-stmt}_2 \ \text{if-stmt}_1
\end{aligned}
$$

除了用于表示语法以外，BNF 还可以用来表示一些其他的东西，比如类型：

$$
\begin{aligned}
\text{type} ::= \ & \texttt{int} \\
| \ & \texttt{bool} \\
| \ & \text{type} \texttt{ -> } \text{type}
\end{aligned}
$$

如果你又感觉头晕了，那就回到 1.1 节，重温一下规则，然后开始读：“一个 type 可以是 `int`，或者 `bool`，或者一个 ‘type `->` type’[7]”。

[7]这个 `->` 箭头是什么意思我们等会再解释。