

对 Parser 的误解

一直很了解人们对于 parser 的误解，可是一直都提不起兴趣来阐述对它的观点。然而我觉得是有必要解释一下这个问题的时候了。我感觉得到大部分人对于 parser 的误解之深，再不澄清一下，恐怕这些谬误就要写进歪曲的历史教科书，到时候就没有人知道真相了。

什么是 Parser

首先来科普一下。所谓 parser，一般是指把某种格式的文本（字符串）转换成某种数据结构的过程。最常见的 parser，是把程序文本转换成编译器内部的一种叫做“抽象语法树”（AST）的数据结构。也有简单一些的 parser，用于处理 CSV，JSON，XML 之类的格式。

举个例子，一个处理算数表达式的 parser，可以把“1+2”这样的，含有 1，+，2 三个字符的字符串，转换成一个对象（object）。这个对象就像 `new BinaryExpression(ADD, new Number(1), new Number(2))` 这样的 Java 构造函数调用生成出来的那样。

之所以需要做这种从字符串到数据结构的转换，是因为编译器是无法直接操作“1+2”这样的字符串的。实际上，代码的本质根本就不是字符串，它本来就是一个具有复杂拓扑的数据结构，就像电路一样。“1+2”这个字符串只是对这种数据结

构的一种“编码”，就像 ZIP 或者 JPEG 只是对它们压缩的数据的编码一样。

这种编码可以方便你把代码存到磁盘上，方便你用文本编辑器来修改它们，然而你必须知道，文本并不是代码本身。所以从磁盘读取了文本之后，你必须先“解码”，才能方便地操作代码的数据结构。比如，如果上面的 Java 代码生成的 AST 节点叫 `node`，你就可以用 `node.operator` 来访问 `ADD`，用 `node.left` 来访问 `1`，`node.right` 来访问 `2`。这是很方便的。

对于程序语言，这种解码的动作就叫做 `parsing`，用于解码的那段代码就叫做 `parser`。

Parser 在编译器中的地位

那么貌似这样说来，`parser` 是编译器里面很关键的一个部分了？显然，`parser` 是必不可少的，然而它并不像很多人想象的那么重要。`Parser` 的重要性和技术难度，被很多人严重的夸大了。一些人提到“编译器”，就跟你提 `LEX`，`YACC`，`ANTLR` 等用于构造 `parser` 的工具，仿佛编译器跟 `parser` 是等价的似的。还有些人，只要听说别人写了个 `parser`，就觉得这人编程水平很高，开始膜拜了。这些都是肤浅的表现。

我喜欢把 `parser` 称为“万里长征的第0步”，因为等你 `parse` 完毕得到了 `AST`，真正的编译技术才算开始。一个编译器包含许多的步骤：语义分析，类型检查/推导，代码优化，机器代码生成，…… 这每个步骤都是在对某种中间数据结构（比如 `AST`）进行分析或者转化，它们完全不需

要知道代码的字符串形式。也就是说，一旦代码通过了 parser，在后面的编译过程里，你就可以完全忘记 parser 的存在。所以 parser 对于编译器的地位，就像 ZIP 之于 JVM，就像 JPEG 之于 PhotoShop。Parser 虽然必不可少，然而它比起编译器里面最重要的过程，是处于一种辅助性的地位。

鉴于这个原因，好一点的大学里的程序语言（PL）课程，都完全没有关于 parser 的内容。学生们往往直接用 Scheme 这样代码数据同形的语言，或者直接使用 AST 数据结构来构造程序。在 Kent Dybvig 这样编译器大师的课程上，学生直接跳过 parser 的构造，开始学习最精华的语义转换和优化技术。实际上，Kent Dybvig 根本不认为 parser 算是编译器的一部分。因为 AST 数据结构才是程序本身，而程序的文本只是这种数据结构的一种编码形式。

Parser 技术发展的误区

既然 parser 在编译器中处于次要的地位，可是为什么还有人花那么大功夫研究各种炫酷的 parser 技术呢。LL, LR, GLR, LEX, YACC, Bison, parser combinator, ANTLR, PEG, 制造 parser 的工具似乎层出不穷，每出现一个新的工具都号称可以处理更加复杂的语法。

很多人盲目地设计复杂的语法，然后用越来越复杂的 parser 技术去 parse 它们，这就是 parser 技术仍然在发展的原因。向往复杂的语法，是程序语言领域流传非常广，危害非常大的错误倾向。在人类历史的长河中，留下了许多难以磨灭

的历史性糟粕，它们固化了人类对于语言设计的理念。很多人设计语言似乎不是为了拿来好用的，而是为了让用它的人迷惑或者害怕。

有些人假定了数学是美好的语言，所以他们盲目的希望程序语言看起来更加像数学。于是他们模仿数学，制造了各种奇怪的操作符，制定它们的优先级，这样你就可以写出 $2 << 7 - 2 * 3$ 这样的代码，而不需要给子表达式加上括号。还有很多人喜欢让语法变得“简练”，就为了少打几个括号，分号，花括号，……可是由此带来的结果是复杂，不一致，有多义性，难扩展的语法，以及障眼难读，模棱两可的代码。

更有甚者，对数学的愚蠢做法执迷不悟的人，设计了像 Haskell 和 Coq 那样的语言。在 Haskell 里面，你可以在代码里定义新的操作符，指定它的“结合律” (associativity) 和“优先级”

(precedence)。这样的语法设计，要求 parser 必须能够在 parse 过程中途读入并且加入新的 parse 规则。Coq 试图更加“强大”一些，它让你可以定义“mixfix 操作符”，也就是说你的操作符可以连接超过两个表达式。这样你就可以定义像 `if...then...else...` 这样的“操作符”。

制造这样复杂难懂的语法，没有什么真正的好处。不但给程序员的学习造成了不必要的困难，让代码难以理解，而且也给 parser 的作者带来了严重的挑战。可是有些人就是喜欢制造问题，就像一句玩笑话说的：有困难要上，没有困难，制造困难也要上！

如果你的语言语法很简单（像 Scheme 那样），你是不需要任何高深的 parser 理论的。说白了，

你只需要知道如何 parse 匹配的括号。最多一个小时，几百行 Java 代码，我就能写出一个类似 Scheme 语言的 parser。

可是很多人总是嫌问题不够有难度，于是他们不停地制造更加复杂的语法，甚至会故意让自己的语言看起来跟其它的不一样，以示“创新”。当然了，这样的语言就得用更加复杂的 parser 技术，这正好让那些喜欢折腾复杂 parser 技术的人洋洋得意。

编译原理课程的误导

程序员们对于 parser 的误解，很大程度上来自于大学编译原理课程照本宣科的教育。很多老师自己都不理解编译器的精髓，所以就只有按部就班的讲一些“死知识”，灌输“业界做法”。一般大学里上编译原理课，都是捧着一本大部头的“[龙书](#)”或者“[虎书](#)”，花掉一个学期 1/3 甚至 2/3 的时间来学写 parser。由于 parser 占据了大量时间，以至于很多真正精华的内容都被一笔带过：语义分析，代码优化，类型推导，静态检查，机器代码生成，……以至于很多人上完了编译原理课程，记忆中只留下写 parser 的痛苦回忆。

“龙书”之类的教材在很多人心目中地位是如此之高，被誉为“经典”，然而除了开头很大篇幅来讲 parser 理论，这书其它部分的水准一般般。大部分学生的反映是“看不懂”，然而由于一直以来没有更好的选择，它经典的地位是难以动摇。“龙书”后来的新版我浏览过一下，新加入了类型检查/推导的部分，可是我看得出来，作者们自己对

于类型理论都是一知半解，所以也就没法写清楚，关键部分几乎是一笔带过。

虎书作者 Appel 水平稍高些，但还是免不了 parser 这个坑。我在 Cornell 的时候上过一门用虎书做教材的编译器课，也是痛苦不堪，一个月都在折腾 parser，我不知道自己为什么要做那些无聊的事情。再加上极其不合理的评分制度，导致我最后不得不退掉这门课程。后来跟虎书作者一个学生合作搞研究，也有一段奇葩的经历，扯远了；)

所以我从来就不认为自己是“编译器”专业的，我认为自己是“PL 专业”。编译器领域照本宣科成分更多一些，PL 专业更加注重本质的东西。

如果你想真的深入理解编译理论，最好是从 PL 课程的读物，比如 [EOPL](#) 开始。我可以说 PL 这个领域，真的和编译器的领域很不一样。请不要指望编译器的作者（比如 LLVM 的作者）能够设计出好的语言，因为他们可能根本不理解很多语言设计的东西，他们只是会实现某些别人设计的语言。可是反过来，理解了 PL 的理论，编译器的东西只不过是把一种语言转换成另外一种语言（机器语言）而已。工程的细枝末节很麻烦，可是当你掌握了精髓的原理，那些都容易摸索出来。

我写 parser 的心得和秘诀

虽然我已经告诉你，给过度复杂的语言写 parser 是很苦逼，没有意思的工作，然而有些历史性的错误已经造成了深远的影响，所以很多时候虽然

心知肚明，你也不得不妥协一下。由于像 C++，Java，JavaScript，Python 之类语言的流行，有时候你是被迫要给它们写 parser。在这一节，我告诉你一些秘诀，也许可以帮助你更加容易的写出这些语言的 parser。

很多人都觉得写 parser 很难，一方面是由于语言设计的错误思想导致了复杂的语法，另外一方面是由于人们对于 parser 构造过程的思维误区。很多人不理解 parser 的本质和真正的用途，所以他们总是试图让 parser 干一些它们本来不应该干的事情，或者对 parser 有一些不切实际的标准。当然，他们就会觉得 parser 非常难写，非常容易出错。

1. 尽量拿别人写的 parser 来用。维护一个 parser 是相当繁琐耗时，回报很低的事情。一旦语言有所改动，你的 parser 就得跟着改。所以如果你能找到免费的 parser，那就最好不要自己写。现在的趋势是越来越多的语言在标准库里提供可以 parse 它自己的 parser，比如 Python 和 Ruby。这样你就可以用那语言写一小段代码调用标准的 parser，然后把它转换成一种常用的数据交换格式，比如 JSON。然后你就可以用通用的 JSON parser 解析出你想要的数据结构了。

如果你直接使用别人的 parser，最好不要使用它原来的数据结构。因为一旦 parser 的作者在新版本改变了他的数据结构，你所有的代码都会需要修改。我的秘诀是做一个“AST 转换器”，先把别人的 AST 结构转换成自己的 AST 结构，然后在自己的

AST 结构之上写其它的代码，这样如果别人的 parser 修改了，你可以只改动 AST 转换器，其它的代码基本不需要修改。

用别人的 parser 也会有一些小麻烦。比如 Python 之类语言自带的 parser，丢掉了很多我需要的信息，比如函数名的位置，等等。我需要进行一些 hack，找回我需要的数据。相对来说，这样小的修补还是比从头写一个 parser 要划得来。但是如果你实在找不到一个好的 parser，那就只好自己写一个。

2. 很多人写 parser，很在乎所谓的“one-pass parser”。他们试图扫描一遍代码文本就构造出最终的 AST 结构。可是如果你放松这个条件，允许用多 pass 的 parser，就会容易很多。你可以在第一遍用很容易的办法构造一个粗略的树结构，然后再写一个递归树遍历过程，把某些在第一遍的时候没法确定的结构进行小规模转换，最后得到正确的 AST。

想要一遍就 parse 出最终的 AST，可以说是一种过早优化 (premature optimization)。有些人盲目地认为只扫描一遍代码，会比扫描两遍要快一些。然而由于你必须在这一遍扫描里进行多度复杂的操作，最终的性能也许还不如很快的扫完第一遍，然后再很快的遍历转换由此生成的树结构。

3. 另外一些人试图在 parse 的过程中做一些本来不属于 parser 职责的事情，比如进行

一些基本的语义检查。有些人会让 parser 检查“使用未定义的变量”等语义错误，一旦发现就在当时报错，终止。这种做法混淆了 parser 的作用，造成了不必要的复杂性。

就像我说的，parser 只是一个解码器。parser 要做的事情，应该是从无结构的字符串里面，解码产生有结构的数据结构。而像“使用未定义的变量”这样的语义检查，应该是在生成了 AST 之后，使用单独的树遍历来进行的。人们常常混淆“解码”，“语法”和“语义”三者的不同，导致他们写出过度复杂，效率低下，难以维护的 parser。

4. 另一种常见的误区是盲目的相信 YACC，ANTLR 之类所谓“parser generator”。实际上 parser generator 的概念看起来虽然美好，可是实际用起来几乎全都是噩梦。事实上最好的 parser，比如 EDG C++ parser，几乎全都是直接用普通的程序语言手写而成的，而不是自动生成的。

这是因为 parser generator 都要求你使用某种特殊的描述语言来表示出语法，然后自动把它们转换成 parser 的程序代码。在这个转换过程中，这种特殊的描述语言和生成的 parser 代码之间，并没有很强的语义连接关系。如果生成的 parser 有 bug，你很难从生成的 parser 代码回溯到语法描述，找到错误的位置和原因。你没法对语法描述进行 debug，因为它只是一个文本文件，根本不能运行。

所以如果你真的要写 parser，我建议你直接用某种程序语言手写代码，使用普通的递归下降（recursive descent）写法，或者 parser combinator 的写法。只有手写的 parser 才可以方便的 debug，而且可以输出清晰，人类可理解的出错信息。

5. 有些人喜欢死扣 BNF 范式，盲目的相信“LL”，“LR”等语法的区别，所以他们经常落入误区，说“哎呀，这个语法不是LL的”，于是采用一些像 YACC 那样的 LR parser generator，结果落入非常大的麻烦。虽然有些语法看起来不是 LL 的，它们的 parser 却仍然可以用普通的 recursive descent 的方式来写。

这里的秘诀在于，语言规范里给出的 BNF 范式，并不是唯一的可以写出 parser 的做法。BNF 只是一个基本的参照物，它让你可以对语法有个清晰的概念，可是实际的 parser 却不一定非得按照 BNF 的格式来写。有时候你可以把语法的格式稍微改一改，变通一下，却照样可以正确地 parse 原来的语言。由于很多语言的语法都类似于 C，所以很多时候你写 parser 只需要看一些样例程序，然后根据自己的经验来写，而不需要依据 BNF。

Recursive descent 和 parser combinator 写出来的 parser 可以非常强大，甚至可以超越所谓“上下文无关文法”，因为在递归函数里面你可以做几乎任意的东西，所以你甚至可以把上下文传递到递归函数里，然后根据上下文来决定对当前

的节点做什么事情。而且由于代码可以得到很多的上下文信息，如果输入的代码有语法错误，你可以根据这些信息生成非常人性化的出错信息。

总结

所以你看到了，parser 并不是编译器，它甚至不属于编译里很重要的东西。程序语言和编译器里面有比 parser 重要很多，有趣很多的东西。

Parser 的研究其实是在解决一些根本不存在或者人为制造的问题。复杂的语法导致了复杂的 parser 技术，它们仍然在给计算机世界带来不必要的困扰和麻烦。对 parser 写法的很多误解，过度工程和过早优化，造成了很多人错误的高估写 parser 的难度。

能写 parser 并不是什么了不起的事情，它是个苦差事。所以如果你会写 parser，请不要以为是什么了不起的事情，如果你看到有人写了某种语言的 parser，也不要表现出让人哭笑不得的膜拜之情。