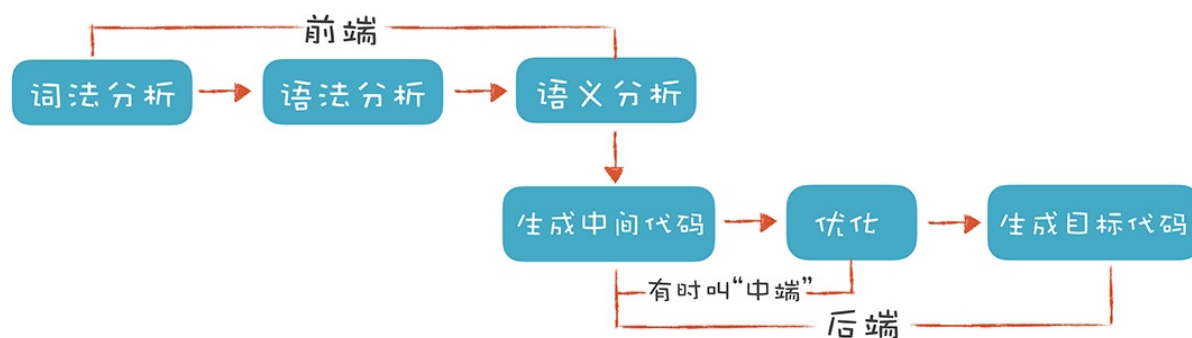


## 01-理解代码：编译器的前端技术

在开篇词里，我分享了一些使用编译技术的场景。其中有的场景，你只要掌握编译器的前端技术就能解决。比如文本分析场景，软件需要用户自定义功能的场景以及前端编程语言的翻译场景等。而且咱们大学讲的编译原理，也是侧重讲解前端技术，可见编译器的前端技术有多么重要。

当然了，这里的“前端”指的是编译器对程序代码的分析和理解过程。它通常只跟语言的语法有关，跟目标机器无关。而与之对应的“后端”则是生成目标代码的过程，跟目标机器有关。为了方便你理解，我用一张图直观地展现了编译器的整个编译过程。



你可以看到，编译器的“前端”技术分为**词法分析**、**语法分析**和**语义分析**三个部分。而它主要涉及自动机和形式语言方面的基础的计算理论。

这些抽象的理论也许会让你“撞墙”，不过不用担心，我今天会把难懂的理论放到一边，用你听得懂的大白话，联系实际使用的场景，带你直观地理解它们，**让你学完本节课之后，实现以下目标：**

- 对编译过程以及其中的技术点有个宏观、概要的了解。
- 能够在脑海里绘制一张清晰的知识地图，以应对工作需要。比如分析一个日志文件时，你能知道所对应的技术点，从而针对性地解决问题。

好了，接下来让我们正式进入今天的课程吧！

### 词法分析

通常，编译器的第一项工作叫做词法分析。就像阅读文章一样，文章是由一个个的中文单词组成的。程序处理也一样，只不过这里不叫单词，而是叫做“词法记号”，英文叫Token。我嫌“词法记号”这个词太长，后面直接将它称作Token吧。

举个例子，看看下面这段代码，如果我们要读懂它，首先要怎么做呢？

```
#include <stdio.h>
int main(int argc, char* argv[]){
    int age = 45;
    if (age >= 17+8+20) {
        printf("Hello old man!\\n");
    }
    else{
        printf("Hello young man!\\n");
    }
}
```

```
}  
    return 0;  
}
```

我们会识别出if、else、int这样的关键字，main、printf、age这样的标识符，+、-、=这样的操作符号，还有花括号、圆括号、分号这样的符号，以及数字字面量、字符串字面量等。这些都是Token。

那么，如何写一个程序来识别Token呢？可以看到，英文内容中通常用空格和标点把单词分开，方便读者阅读和理解。但在计算机程序中，仅仅用空格和标点分割是不行的。比如“age >= 45”应该分成“age” “>=”和“45”这三个Token，但在代码里它们可以是连在一起的，中间不用非得有空格。

这和汉语有点儿像，汉语里每个词之间也是没有空格的。但我们会下意识地把句子里的词语正确地拆解出来。比如把“我学习编程”这个句子拆解成“我” “学习” “编程”，这个过程叫做“分词”。如果你要研发一款支持中文的全文检索引擎，需要有分词的功能。

其实，我们可以通过制定一些规则来区分每个不同的Token，我举了几个例子，你可以看一下。

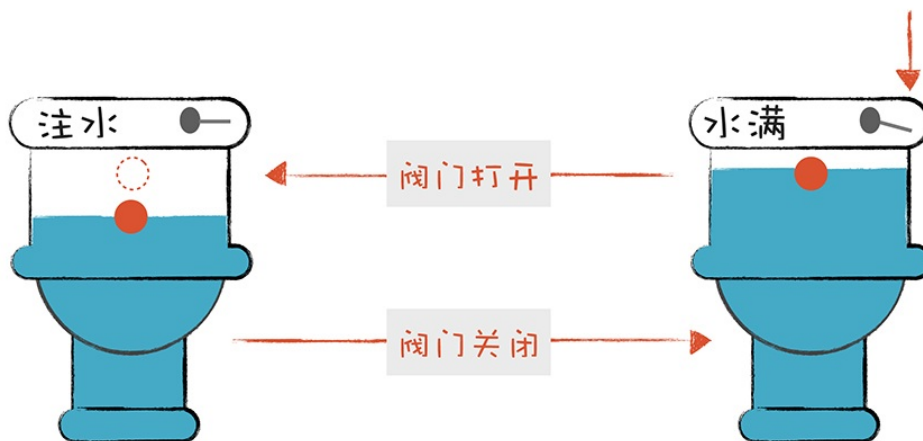
- **识别age这样的标识符。**它以字母开头，后面可以是字母或数字，直到遇到第一个既不是字母又不是数字的字符时结束。
- **识别>=这样的操作符。**当扫描到一个>字符的时候，就要注意，它可能是一个GT（Greater Than，大于）操作符。但由于GE（Greater Equal，大于等于）也是以>开头的，所以再往下再看一位，如果是=，那么这个Token就是GE，否则就是GT。
- **识别45这样的数字字面量。**当扫描到一个数字字符的时候，就开始把它看做数字，直到遇到非数字的字符。

这些规则可以通过手写程序来实现。事实上，很多编译器的词法分析器都是手写实现的，例如GNU的C语言编译器。

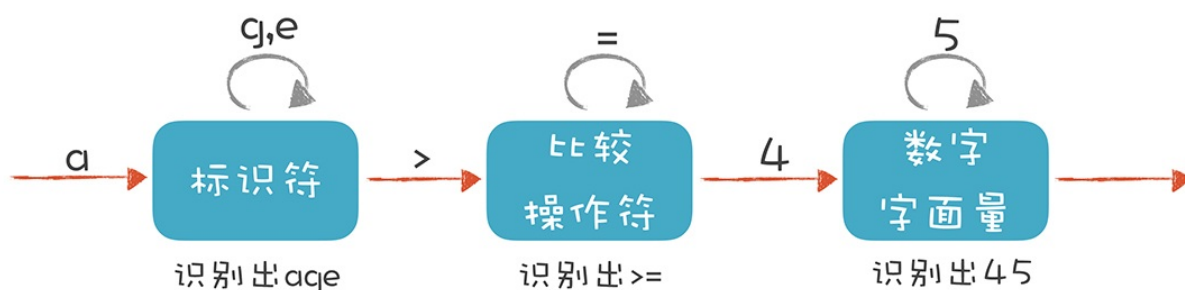
如果嫌手写麻烦，或者你想花更多时间陪恋人或家人，也可以偷点儿懒，用词法分析器的生成工具来生成，比如Lex（或其GNU版本，Flex）。这些生成工具是基于一些规则来工作的，这些规则用“正则文法”表达，符合正则文法的表达式称为“正则表达式”。生成工具可以读入正则表达式，生成一种叫“有限自动机”的算法，来完成具体的词法分析工作。

不要被“正则文法”和“有限自动机”吓到。正则文法是一种最普通、最常见的规则，写正则表达式的时候用的就是正则文法。我们前面描述的几个规则，都可以看成口语化的正则文法。

有限自动机是有限个状态的自动机器。我们可以拿抽水马桶举例，它分为两个状态：“注水”和“水满”。摁下冲马桶的按钮，它转到“注水”的状态，而浮球上升到一定高度，就会把注水阀门关闭，它转到“水满”状态。



词法分析器也是一样，它分析整个程序的字符串，当遇到不同的字符时，会驱使它迁移到不同的状态。例如，词法分析程序在扫描age的时候，处于“标识符”状态，等它遇到一个>符号，就切换到“比较操作符”的状态。词法分析过程，就是这样一个个状态迁移的过程。



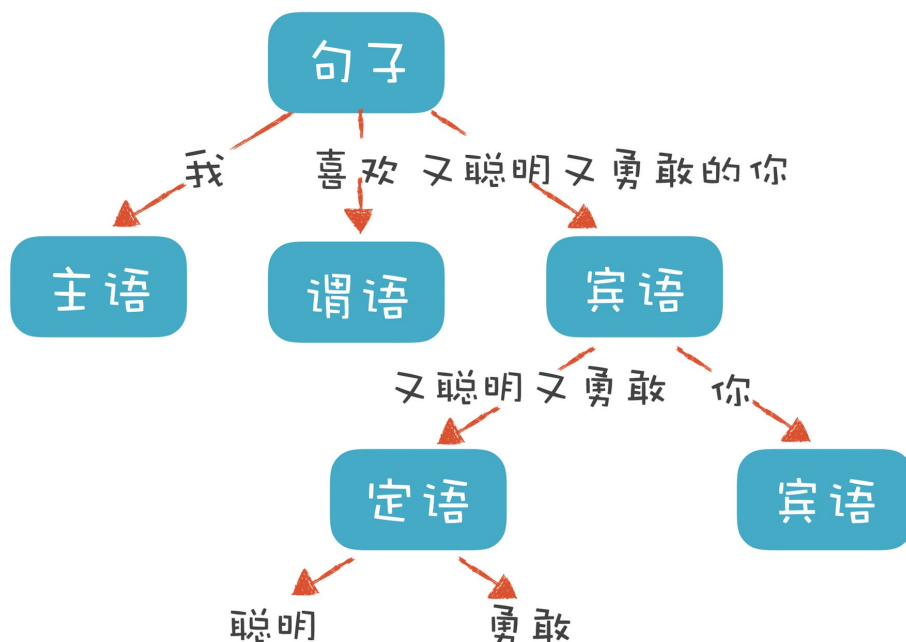
你也许熟悉正则表达式，因为我们在编程过程中经常用正则表达式来做用户输入的校验，例如是否输入了一个正确的电子邮件地址，这其实就是在做词法分析，你应该用过。

## 语法分析

编译器下一个阶段的工作是语法分析。词法分析是识别一个个的单词，而语法分析就是在词法分析的基础上识别出程序的语法结构。这个结构是一个树状结构，是计算机容易理解和执行的。

以自然语言为例。自然语言有定义良好的语法结构，比如，“我喜欢又聪明又勇敢的你”这个句子包含了“主、谓、宾”三个部分。主语是“我”，谓语是“喜欢”，宾语部分是“又聪明又勇敢的你”。其中宾语部分又可以拆成两部分，“又聪明又勇敢”是定语部分，用来修饰“你”。定语部分又可以分成“聪明”和“勇敢”两个最小的单位。

这样拆下来，会构造一棵树，里面的每个子树都有一定的结构，而这个结构要符合语法。比如，汉语是用“主谓宾”的结构，日语是用“主宾谓”的结构。这时，我们说汉语和日语的语法规则是不同的。



程序也有定义良好的语法结构，它的语法分析过程，就是构造这么一棵树。一个程序就是一棵树，这棵树叫做**抽象语法树**（Abstract Syntax Tree, AST）。树的每个节点（子树）是一个语法单元，这个单元的构成规则就叫“语法”。每个节点还可以有下级节点。

层层嵌套的树状结构，是我们对计算机程序的直观理解。计算机语言总是一个结构套着另一个结构，大的程序套着子程序，子程序又可以包含子程序。

接下来，我们直观地看一下这棵树长什么样子。我在Mac电脑上打下这个命令：

```
clang -cc1 -ast-dump hello.c
```

这个命令是运行苹果公司的C语言编译器来编译hello.c，-ast-dump参数使它输出AST，而不是做常规的编译。我截取了一部分输出结果给你看，从中你可以看到这棵树的结构。试着修改程序，添加不同的语句，你会看到不同的语法树。

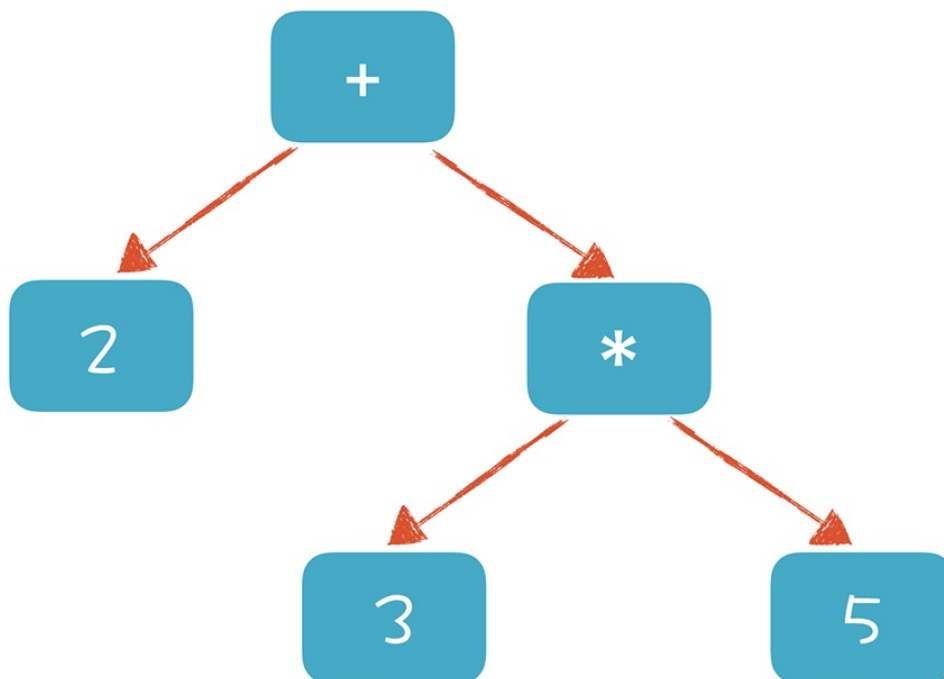
```

-CompoundStmt 0x7fad85855838 <line:3:1, line:12:1>
|-DeclStmt 0x7fad85855280 <line:4:2, col:14>
|`-VarDecl 0x7fad85855200 <col:2, col:12> col:6 used age 'int' cinit
|`-IntegerLiteral 0x7fad85855260 <col:12> 'int' 45
|-IfStmt 0x7fad858557c8 <line:5:2, line:10:2>
|`-<<NULL>>>
|`-<<NULL>>>
|-BinaryOperator 0x7fad85855388 <line:5:6, col:18> 'int' '>='
|`-ImplicitCastExpr 0x7fad85855370 <col:6> 'int' <LValueToRValue>
|`-DeclRefExpr 0x7fad85855298 <col:6> 'int' lvalue Var 0x7fad85855200 'age' 'int'
|-BinaryOperator 0x7fad85855348 <col:13, col:18> 'int' '+'
|`-BinaryOperator 0x7fad85855300 <col:13, col:16> 'int' '+'
|`-IntegerLiteral 0x7fad858552c0 <col:13> 'int' 17
|`-IntegerLiteral 0x7fad858552e0 <col:16> 'int' 8
|`-IntegerLiteral 0x7fad85855328 <col:18> 'int' 20
|-CompoundStmt 0x7fad85855680 <col:22, line:7:2>
|`-CallExpr 0x7fad85855620 <line:6:3, col:28> 'int'
|`-ImplicitCastExpr 0x7fad85855608 <col:3> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
|`-DeclRefExpr 0x7fad85855540 <col:3> 'int (const char *, ...)' Function 0x7fad858553e8 'printf' 'int (const char *, ...)'
|`-ImplicitCastExpr 0x7fad85855668 <col:10> 'const char *' <BitCast>
|`-ImplicitCastExpr 0x7fad85855650 <col:10> 'char *' <ArrayToPointerDecay>
|`-StringLiteral 0x7fad858555a8 <col:10> 'char [16]' lvalue "Hello old man!\n"
|-CompoundStmt 0x7fad858557b0 <line:8:6, line:10:2>
|`-CallExpr 0x7fad85855750 <line:9:3, col:30> 'int'
|`-ImplicitCastExpr 0x7fad85855738 <col:3> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
|`-DeclRefExpr 0x7fad85855698 <col:3> 'int (const char *, ...)' Function 0x7fad858553e8 'printf' 'int (const char *, ...)'
|`-ImplicitCastExpr 0x7fad85855798 <col:10> 'const char *' <BitCast>
|`-ImplicitCastExpr 0x7fad85855780 <col:10> 'char *' <ArrayToPointerDecay>
|`-StringLiteral 0x7fad858556f8 <col:10> 'char [18]' lvalue "Hello young man!\n"
|-ReturnStmt 0x7fad85855820 <line:11:2, col:9>
|`-IntegerLiteral 0x7fad85855800 <col:9> 'int' 0

```

如果你觉得这棵树还不够直观，可以参考我提供的[网址](#)，它能够生成JavaScript语言的AST，并以更加直观的方式呈现。

在这个网址里输入一个可以计算的表达式，例如“2+3\*5”，你会得到一棵类似下图的AST。



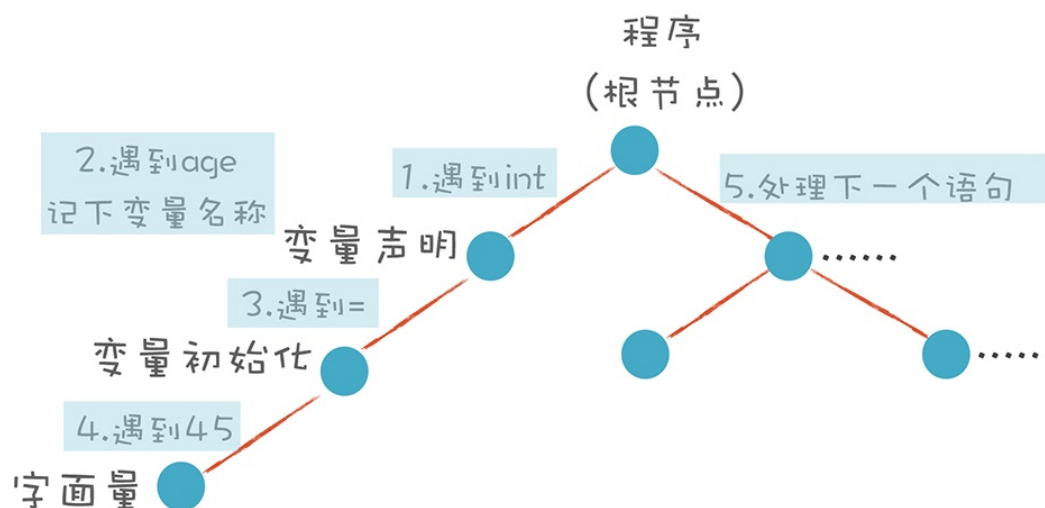
**形成AST以后有什么好处呢？就是计算机很容易去处理。**比如，针对表达式形成的这棵树，从根节点遍历整棵树就可以获得表达式的值。基于这个原理，我在后面的课程中会带你实现一个计算器，并实现自定义公式功能。

如果再把循环语句、判断语句、赋值语句等节点加到AST上，并解释执行它，那么你实际上就实现了一个脚本语言。而执行脚本语言的过程，就是遍历AST的过程。当然，在后面的课程中，我也会带你实际实现一个脚本语言。

**好了，你已经知道了AST的作用，那么怎样写程序构造它呢？**

一种非常直观的构造思路是自上而下进行分析。首先构造根节点，代表整个程序，之后向下扫描Token串，构建它的子节点。当它看到一个int类型的Token时，知道这儿遇到了一个变量声明语句，于是建立一个“变量声明”节点；接着遇到age，建立一个子节点，这是第一个变量；之后遇到=，意味着这个变量有初始化值，那么建立一个初始化的子节点；最后，遇到“字面量”，其值是45。

这样，一棵子树就扫描完毕了。程序退回到根节点，开始构建根节点的第二个子节点。这样递归地扫描，直到构建起一棵完整的树。



这个算法就是非常常用的递归下降算法。是不是很简单？你完全可以动手写出来。

递归下降算法是一种自顶向下的算法，与之对应的，还有自底向上的算法。这个算法会先将最下面的叶子节点识别出来，然后再组装上一级节点。有点儿像搭积木，我们总是先构造出小的单元，然后再组装成更大的单元。原理就是这么简单。

也许你会想，除了手写，有没有偷懒的、更省事的方法呢？多一些时间去陪家人总不是坏事。

你现在已经有了一定的经验，大可以去找找看有没有现成的工具，比如Yacc（或GNU的版本，Bison）、Antlr、JavaCC等。实际上，你可以在维基百科里找到一个挺大的清单，我把它放到了CSDN的[博客](#)上，其中对各种工具的特性做了比较。

顺理成章地，你还能找到很多开源的语法规则文件，改一改，就能用工具生成你的语法分析器。

很多同学其实已经做过语法解析的工作，比如编写一个自定义公式的功能，对公式的解析就是语法分析过程。另一个例子是分析日志文件等文本文件，对每行日志的解析，本质上也是语法分析过程。解析用XML、JSON写的各种配置文件、模型定义文件的过程，其实本质也是语法分析过程，甚至还包含了语义分析工作。

## 语义分析

好了，讲完了词法分析、语法分析，编译器接下来做的工作是语义分析。说白了，语义分析就是要让计算机理解我们的真实意图，把一些模棱两可的地方消除掉。



以 “You can never drink too much water.” 这句话为例。它的确切含义是什么？是 “你不能喝太多水”，还是 “你喝多少水都不嫌多”？实际上，这两种解释都是可以的，我们只有联系上下文才能知道它的准确含义。

你可能会觉得理解自然语言的含义已经很难了，所以计算机语言的语义分析也一定很难。其实语义分析没那么复杂，因为计算机语言的语义一般可以表达为一些规则，你只要检查是否符合这些规则就行了。比如：

- 某个表达式的计算结果是什么数据类型？如果有数据类型不匹配的情况，是否要做自动转换？
- 如果在一个代码块的内部和外部有相同名称的变量，我在执行的时候到底用哪个？就像 “我喜欢又聪明又勇敢的你” 中的 “你”，到底指的是谁，需要明确。
- 在同一个作用域内，不允许有两个名称相同的变量，这是唯一性检查。你不能刚声明一个变量a，紧接着又声明同样名称的一个变量a，这就不允许了。

语义分析基本上就是做这样的事情，也就是根据语义规则进行分析判断。

语义分析工作的某些成果，会作为属性标注在抽象语法树上，比如在age这个标识符节点和45这个字面量节点上，都会标识它的数据类型是int型的。

在这个树上还可以标记很多属性，有些属性是在之前的两个阶段就被标注上了，比如所处的源代码行号，这一行的第几个字符。这样，在编译程序报错的时候，就可以比较清楚地了解出错的位置。

做了这些属性标注以后，编译器在后面就可以依据这些信息生成目标代码了，我们在编译技术的后端部分会去讲。

## 课程小结

讲完语义分析，本节课也就告一段落了，我来总结一下本节课的重点内容：

- 词法分析是把程序分割成一个个Token的过程，可以通过构造有限自动机来实现。
- 语法分析是把程序的结构识别出来，并形成一棵便于由计算机处理的抽象语法树。可以用递归下降的算法来实现。
- 语义分析是消除语义模糊，生成一些属性信息，让计算机能够依据这些信息生成目标代码。

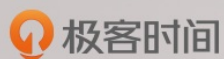
我想让你知道，上述编译过程其实跟你的实际工作息息相关。比如，词法分析就是你工作中使用正则表达式的过程。而语法分析在你解析文本文件、配置文件、模型定义文件，或者做自定义公式功能的时候都会用到。

我还想让你知道，编译技术并没有那么难，它的核心原理是很容易理解的。学习之后，你能很快上手，如果善用一些辅助生成工具会更省事。所以，我希望你通过学习这篇文章，已经破除了一些心理障碍，并跃跃欲试，想要动手做点儿什么了！

## 一课一思

你有没有觉得，刚开始学编译原理中的某些知识点时特别难，一旦学通了以后，就会发出类似的感慨：“啊！原来就是这么回事！” 欢迎在留言区与我分享你的感慨时刻。另外，你是否尝试实现过一个编译器，还颇有一些心得？可以在留言区与大家一起交流。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。



# 编译原理之美

## 手把手教你实现一个编译器

宫文学

北京物演科技CEO



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

### 精选留言：

• Simon 2019-08-14 18:29:17

之前基于ANTLR给游戏策划做了一套dsl 战斗技能解析器，策划只要在文件中配置技能的效果就行，战斗的时候根据配置的表达式去解析并计算伤害啥的。感觉策划们用的很爽，新增加技能或者修改技能效果再也不用找开发了。开发感觉更爽。😄😄 [12赞]

作者回复2019-08-14 19:15:25

对的，做DSL能够给具体的领域带来很大的帮助！感谢你分享游戏领域的经验！

我知道游戏领域有人在做DSL，我对这个领域不太了解。马云也曾经参观国外的一个游戏公司，发现他们把平台做的很好，能够支持很多游戏的快速研发。他回来以后，在阿里提了中台的概念，目前在企业应用领域很流行。中台呀，游戏平台呀，编译原理都能发挥作用。

• coder 2019-08-14 21:39:55

我做过一款已经投入商用的编译器，从编译器的前端到中端的优化，再到后端针对某个特定architecture的代码生成以及优化，完整地趟过一遍，编译器的实现是基于clang和llvm的，目前的感受是，工程经验确实积累了不少。antlr, flex, bison等这些工具也都用过。所以在看到课程目录时，感觉比较熟悉，目录中提到的东西都是知道的。

但是我目前的感受是，自己在理论方面的积累还是有些欠缺的，那些PL相关的理论也同样重要，包括图灵机，状态机，计算模型，lambda演算，类型和类型系统等，所以希望老师也能够也讲一下这部分理论 😊😊😊 [7赞]

作者回复2019-08-14 22:29:49

看来我要跟你学习才对:)

这门课是实战和原理并重。但在设计上，会先让同学们建立对编译的直观感受，然后再引导到对算法的兴趣上。所以，在前端部分，我把算法的总结和提升放到了最后部分。因为这个时候，大家都已经没有陌生感、恐惧感了，学算法也就顺理成章了。



你的经验已经比大部分同学多。我也争取把理论这部分也在课程里搞好！

- Fan 2019-08-14 19:46:51  
希望后面有具体的一些demo [7赞]

作者回复2019-08-14 20:06:26

有的，有很多demo。这门课不是纯讲理论，而是拿实操带动我们学理论。

第一节课是前端技术的概述。第二节就开始做例子了！示例代码到时候也释放出来。

- 旺旺 2019-08-14 18:08:08  
讲的还是很易懂的，👍！ [5赞]

作者回复2019-08-14 19:19:38

谢谢鼓励。讲的易懂，是我准备这门课程最大的挑战。我会继续努力！

- X中倪 2019-08-14 17:06:04  
盘他 [5赞]

作者回复2019-08-14 19:21:25

对头！

- 公众号：业余草 2019-08-15 09:08:21  
BAT为什么没有造出一门语言的轮子？ [3赞]

作者回复2019-08-15 10:20:21

首先呢，编译原理不仅仅是造大家通用的语言。有很多是用来解决领域问题的。比如，好像阿里内部就有一个语言，叫cava，是跟它的搜索引擎配合的。

其次，我们做这些底层平台的时候，都要涉及编译。还是拿阿里的druid开源项目说事，它就手写了一个高速的sql parser。其他几家大的互联网公司肯定也有不少这种应用。

华为的方舟编译器，不是也很争气吗？有人说它没有多少创新，但它就是效果显著呀。

我跟大家一样，都期盼带有中国血统的、全球人都用的语言诞生出来。我相信时间不远了。这种事情需要技术底蕴，更重要的是需要产业发言权。这方面，我们是越来越强了！大家一起期盼！

- William 2019-08-14 23:45:16  
分析一下自己做得一些摘录。

trick:

mac 下的 clang命令可以编译C语言代码。

术语:

正则文法 最普通、最常见的规则

有限自动机 有限个状态的自动机器

词法分析 lexical analysis -> 分词

\* 实现原理：有限自动机

\* 现成的词法分析工具：Lex、GNU Flex

语法分析 parser -> 根据语法规则识别出程序的语法结构（抽象语法树AST）

\* 需要考虑优先级等等

\* 递归下降方法

\* 现成的语法分析工具：Yacc、GNU Bison、Antlr、JavaCC

语义分析 semantic analysis -> 上下文分析、消除歧义

- \* 变量引用消解、作用域
- \* 合法性检查
- \* 数据类型标识
- \* 语义分析的某些结果，会作为属性标注在AST上

[3赞]

作者回复2019-08-15 08:16:57

记得这么认真！

Great!

- halweg 2019-08-14 18:15:56

很棒！

之前看了一本从逻辑门构建一台带操作系统的计算机的书，卡在了那本书里jack的语言的编译上，这次希望可以搞懂 [3赞]

作者回复2019-08-14 19:18:07

如果我的哪一讲让你卡壳了，你要告诉我！我单独给你讲：)

- frazer 2019-08-15 09:20:56

老师，不涉及 逻辑 分析么？ [2赞]

作者回复2019-08-15 10:14:37

当然要涉及。

但我重视帮大家建立直觉的理解。一旦你直觉上理解那件事了，对它有感觉了，去做逻辑推理就很顺利了。

我觉得，这种直接认知，是学好一门课的关键。

- Mr.J 2019-08-15 01:43:33

老师好，词法分析好理解一些，即程序识别出每个词汇，语法分析这个，拆分AST，打个比方，一个java代码：

```
class A {  
  int a = 2;  
  boolean = false;  
}
```

语法分析时，把这个拆分AST，比如，遇到class，定义为根节点，两个关键字int 和boolean为两个子节点，每个子节点下面继续拆分，可以这样理解不，一条完整的java代码，就是从最后一个自己点一直向上到某个节点？

语义分析这个是在整个上下文中去进行的，定了全局变量a，在方法中使用a时，能够知道这个是在全居中定义过的那个a？ [2赞]

作者回复2019-08-15 08:08:53

你大的概念理解完全正确。

这个类语法分析后会变成一棵树。像这样的语法，解析起来是很简单的。反倒是像表达式这样看似很简单的语法，解析起来反倒有难度。在后面的课程里会深入到细节。

语义分析的关键点你也抓住了，就是上下文。语法阶段是上下文无关的，语义阶段则专门处理上下文。加油！

- William 2019-08-14 23:44:53

请问老师，JavaScript解释器 V8引擎或者Node.js解释器里内置了Antlr这些语法分析器吗，还是自己实现了一套？ [2赞]

作者回复2019-08-15 08:21:09

这些级别的项目，还是自己写的。因为会最契合项目的需求。比如速度、资源消耗、解析过程等方面的需求。因为对V8的编写者来说，语法分析这个层面的事情真的不是事情。你如果把前端技术学透，也不会觉得这是个事。

另一方面，对于大多数项目来说，借助工具就足够了。比如，Hibernate，比如mysql workbench。后者就是用的antlr。

- 雲至 2019-08-14 17:45:58

老师能讲一个具体树最好了 [2赞]

作者回复2019-08-14 19:20:41

后面的课程中，有好多棵具体的树，慢慢来！

- 胖胖胖 2019-08-14 21:39:23

感觉大概明白啥意思了，但想总结出笔记，发现像词法分析中正则这些概念大概理解了但没法清晰的定义出来，老师阔以在这些讲解之后给一个清晰的定义，方便我们验证自己的理解吗。最好有一些好的和当堂课相关，更深入一点的读物推荐。就像计算机组成原理专栏的徐老师的那种方式。纯属个人建议。然后确实讲的很清楚，感谢老师！ [1赞]

作者回复2019-08-14 22:32:18

好的。第一讲是前端技术的概述，我的目标是把全貌展现一下。接着，就进入每个具体知识点的讲解。下一讲就开始把词法分析的细节展开了。

好的，我会在课程里注意推荐读物。课程体系中，我们专门安排了答疑的部分，估计到时候会放到那里面的。

- 信 2019-08-14 20:44:44

babel的编译器大概看过看不太懂，我是做前端的，学了老师的课程能不能做一个页面设计器呢 [1赞]

作者回复2019-08-14 21:43:19

我自己做过表单设计器、报表设计器，跟你说的页面设计器应该有相似之处。

界面模板引擎啦、前端的DSL啦，也都与你的兴趣有关联吧？这些都与编译技术有关。

我后面在课程里还提供了一个报表系统的原型系统。估计也会对你有启发。也就是说，做前端方面的工具，真的经常用到编译技术。

- kirogiyi 2019-08-15 23:12:52

有一种回归计算机本质的感觉，向宫老师多学习。

- 重 2019-08-15 23:10:31

不错

- 傲娇的小宝 2019-08-15 23:03:49

感觉就是类似XML的实现，节点类似xml的节点，语义分析的成果就像xml上的属性。似乎很多东西其实都是树形结构，通过属性记录一些东西，节点保存东西，运行的时候两相结合。这些操作的根本目的就是保存住有效信息并且有效读取。

- Rockbean 2019-08-15 23:02:11

请教老师，AST可不可以转成可视化UML

- Rockbean 2019-08-15 22:50:34

想起标识符的命名为什么不能用数字开头，扫描到了数字，立马认为这个token是数字，不作标识符处理

了

- Vivian 2019-08-15 22:20:42  
看到了AST