Compilers - Principles, Techniques and Tools (2nd)

Chapter 1: Introduction

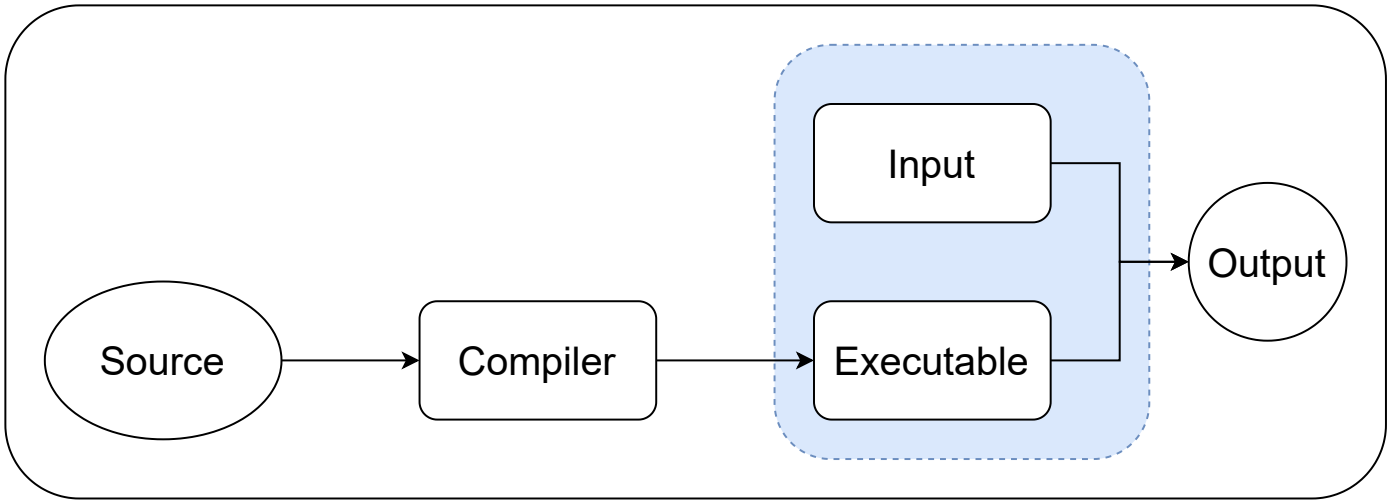
2023-01-30

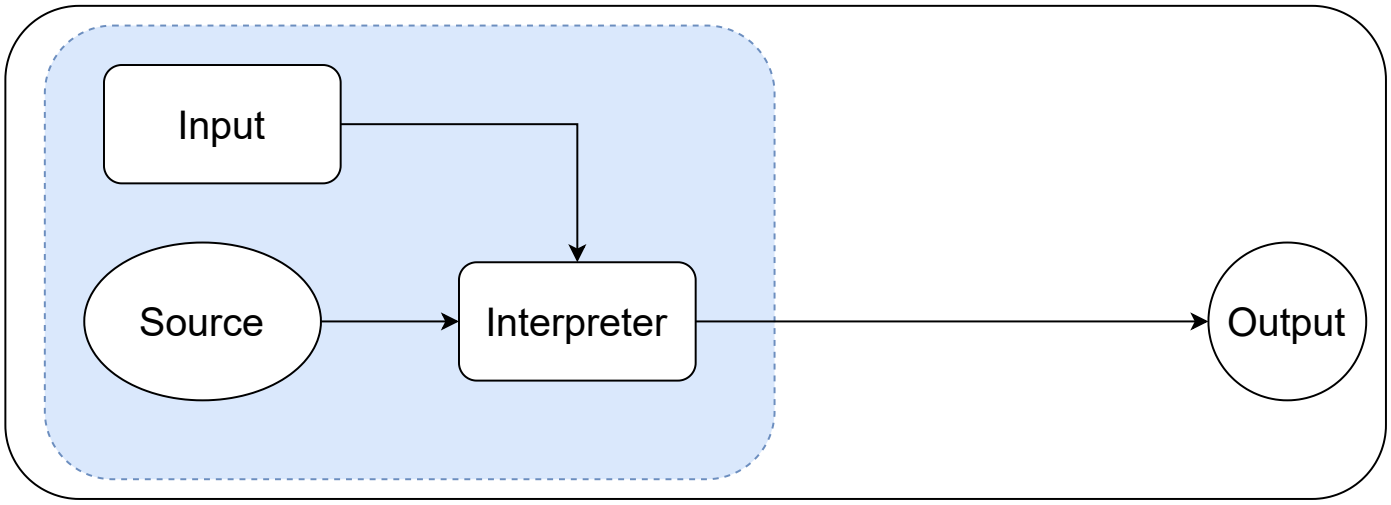
语言处理系统

编译器基本介绍

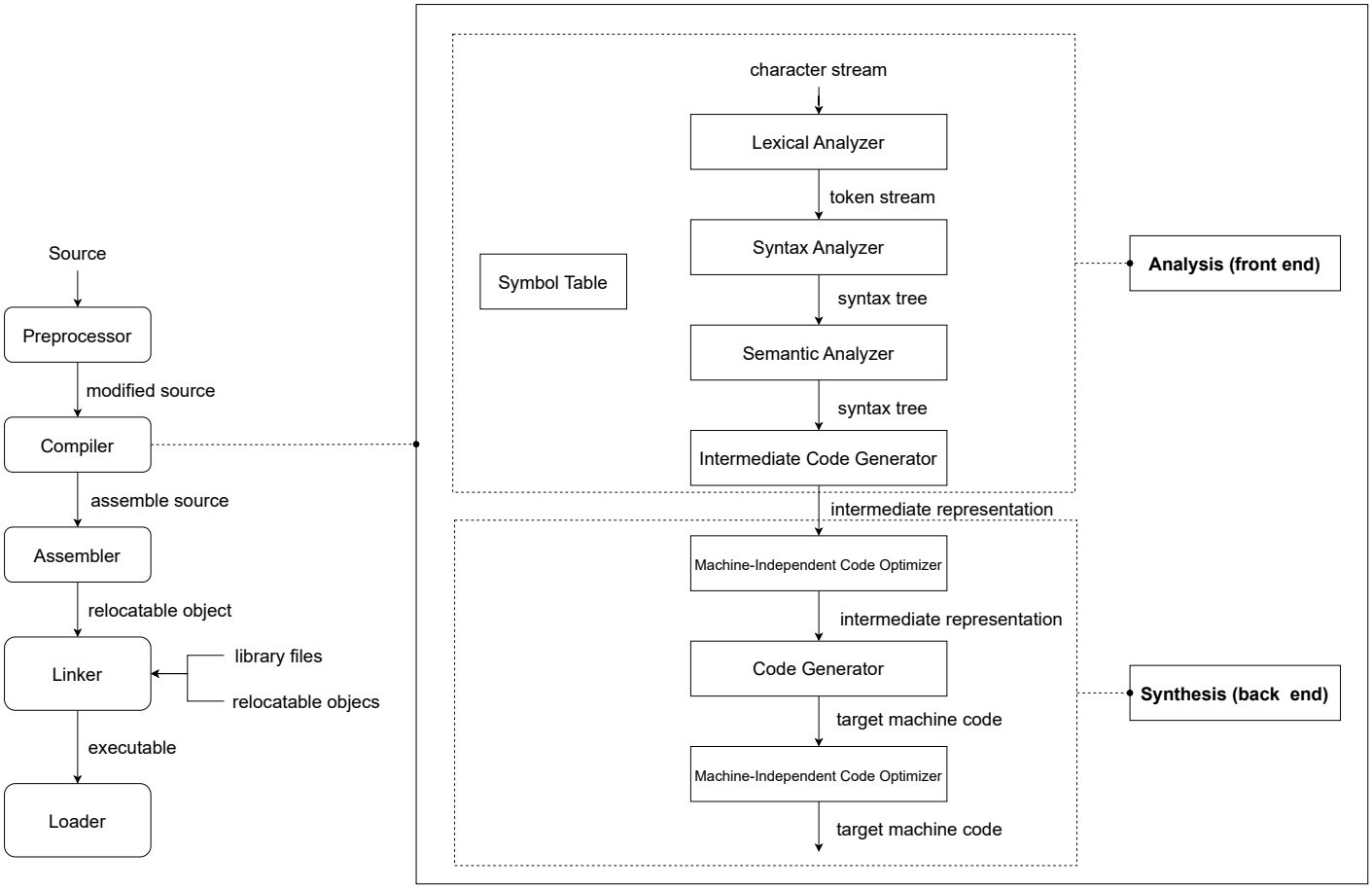
人和机器通过程序语言进行沟通, 虽然人可以直接理解程序语言, 但是机器并不能, 为了解决这个问题, 就需要语言处理器 (language processor) 来将程序语言转换为机器可以理解的语言.

语言处理器通常有两种, 一种是编译器 (compiler), 另外一种是解释器 (interpreter). 编译器将程序语言转换为一个可执行程序, 用户可以在机器上直接执行这个程序, 而解释器则是直接读入程序语言, 然后将其转换为机器可以理解的语言在机器上进行执行, 如下图所示, 其中浅蓝色框中的表示用户看到的部分.





注意, 通常使用编译器的语言处理器会借助其他工具生成最终的可执行程序, 比如如汇编器, 链接器, 加载器等, 一种主流的编译器架构如下图所示

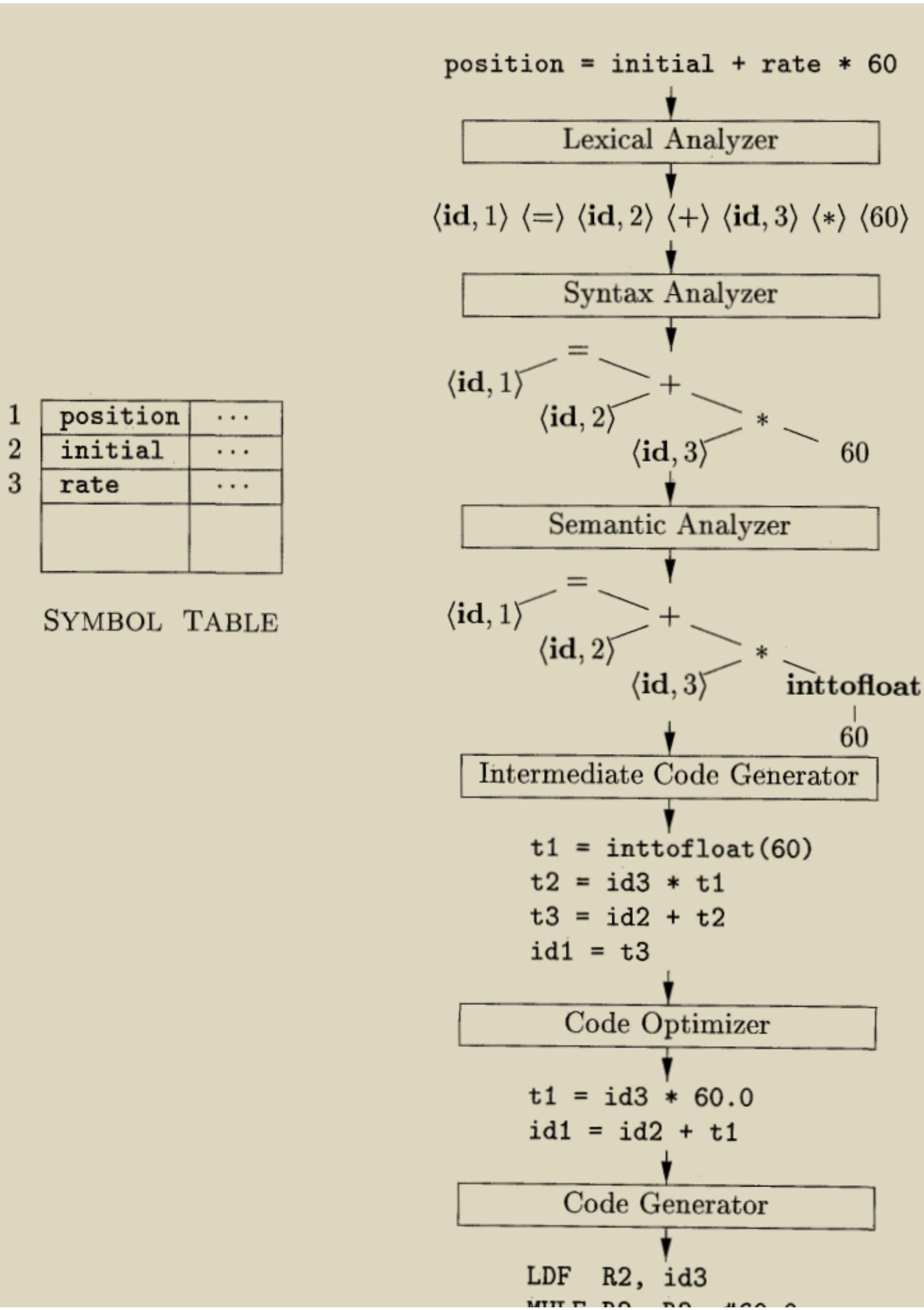


本书主要讲述编译器相关理论技术, 此图也给出了编译器内部的组成部分, 主要分为前端和后端两部分. 前端会对源码进行处理, 生成语法树以及符号表, 通过语法树和符号表生成机器无关的中间代码, 然后将这些内容传递给后端, 最终生成可执行程序.

现在让我们看一个例子, 假设程序的源代码为

```
position = initial + rate * 60
```

那么编译器编译该代码的中间详细过程如下图所示


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
Semantic Analyzer
inttofloat
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
1 position ...
2 initial ...
3 rate ...
SYMBOL TABLE

```
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

- Lexical Analysis

  lexical 处理源码后, 会生成一个 token 列表, 每个 token 的形式都是 `<token-name, attribute-value>`. `token-name` 是一个抽象符号, 在 syntax analyzer 中会用到, 这里的 `id` 表示 identifier, 因为 position, initial, rate 都是一个标识符, 所以都用 `id` 来表示. `<attribute-value>` 是符号表的索引, 可以通过该值从符号表获取 token 的附加信息, 比如 token 的名称和类型 (注意图中简化了 `=, +, *, 60` 的表示).

- Syntax Analyzer

  语法分析的主要作用是将 lexical 生成的 token 列表转换为一个语法树, 后续阶段都会以该语法树为基础进行程序分析并生成目标程序.

- Semantic Analysis

  语义分析主要有如下几个作用

  - 语义一致性检查 (semantic consistency)

    检查编写的源码是否符合语言规范, 比如类型检查 (type checking) 会查看要求的类型和给定的类型是否匹配, 比如数组索引要求是一个整数, 但是用户传入的是一个浮点数, 那么这个索引就是非法的.

  - 类型协变 (coercion)

    即类型的隐式转换, 假设 position, initial, rate 都是浮点数, 而语义分析器发现乘法两侧的操作数类型不一致, 就会默认将整数 60 转换为浮点数, 如上图中的 inttofloat 节点所示.

- 中间代码生成

  在生成机器代码之前, 编译器一般会生成一种机器无关的但是接近机器语言的中间代码, 这种中间代码通常具有两个重要的属性

  - 容易从源码生成
  - 容易转换为目标机器语言

  比如上图就是生成了一个三地址 (three-address) 格式的中间代码, 每条指令含有的操作数不超过 3 个, 中间运算结果使用临时变量表示.

- 代码优化

  为了各种各样的目的, 比如提高程序运行速度, 减少产物的大小等, 会对中间代码进行优化. 比如这里的优化就是直接在编译时将 60 变成 60.0, 并将 4 条中间指令变为 2 条.

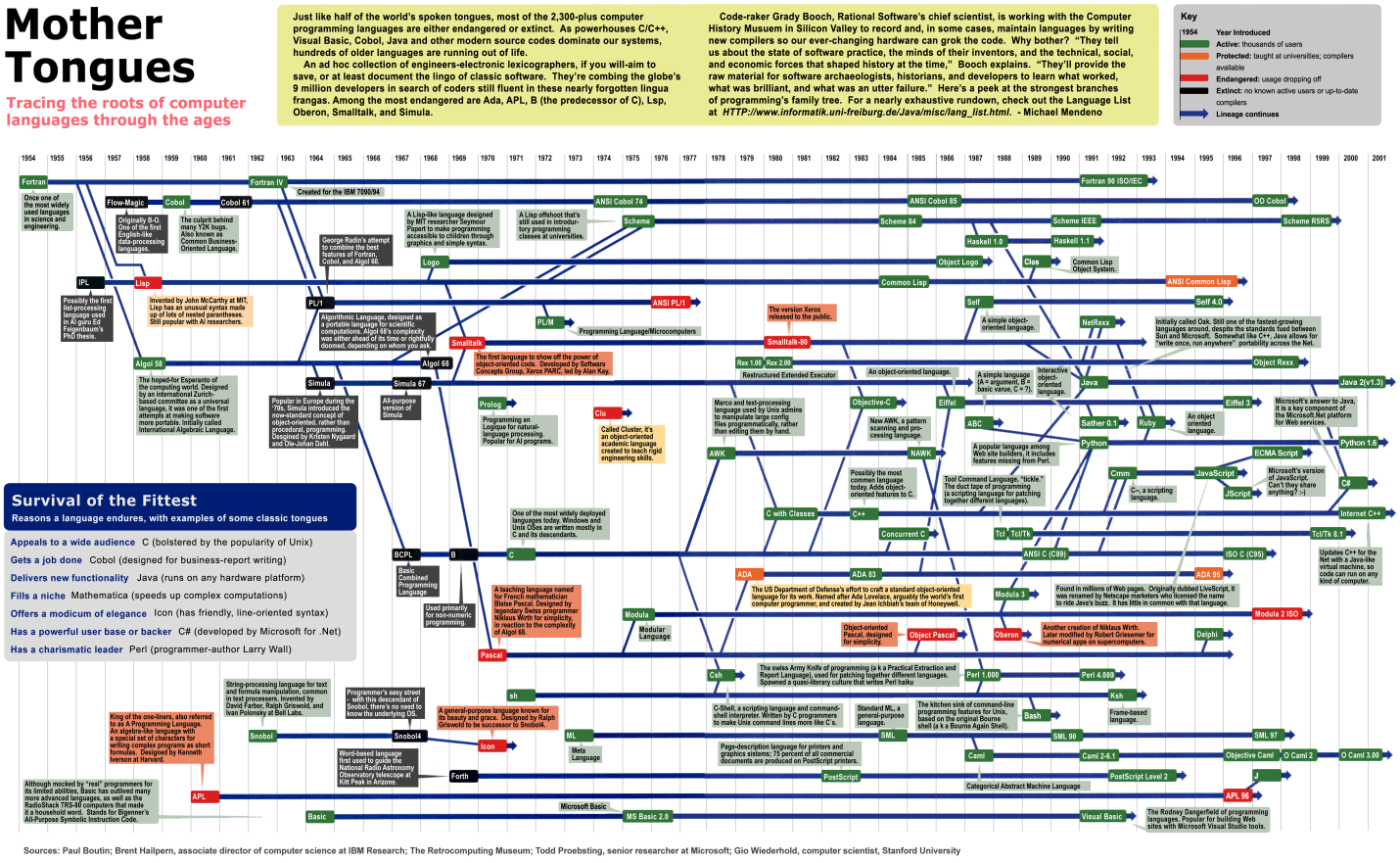
- 代码生成

  这一步主要的工作是将中间代码转换为目标机器码, 比较关键的部分是为变量分配寄存器.

除了以上步骤, 编译器还涉及到符号表的管理, 符号表的主要作用时存储源码中各个变量的相关信息. 比如名称, 存储类型, 作用域, 函数名, 函数参数传递类型, 函数返回值等.

编程语言的进化

在高级语言出现之前是汇编语言的天下, 虽然汇编语言有助记词 (mnemonic), 但是编程仍然是十分麻烦的. 20 世纪 50 年代, 第一个高级编程语言是 Fortran 被创建出来, 自此开启了大约 50 年的编程语言黄金时代, 如下图所示 (源自 这里).

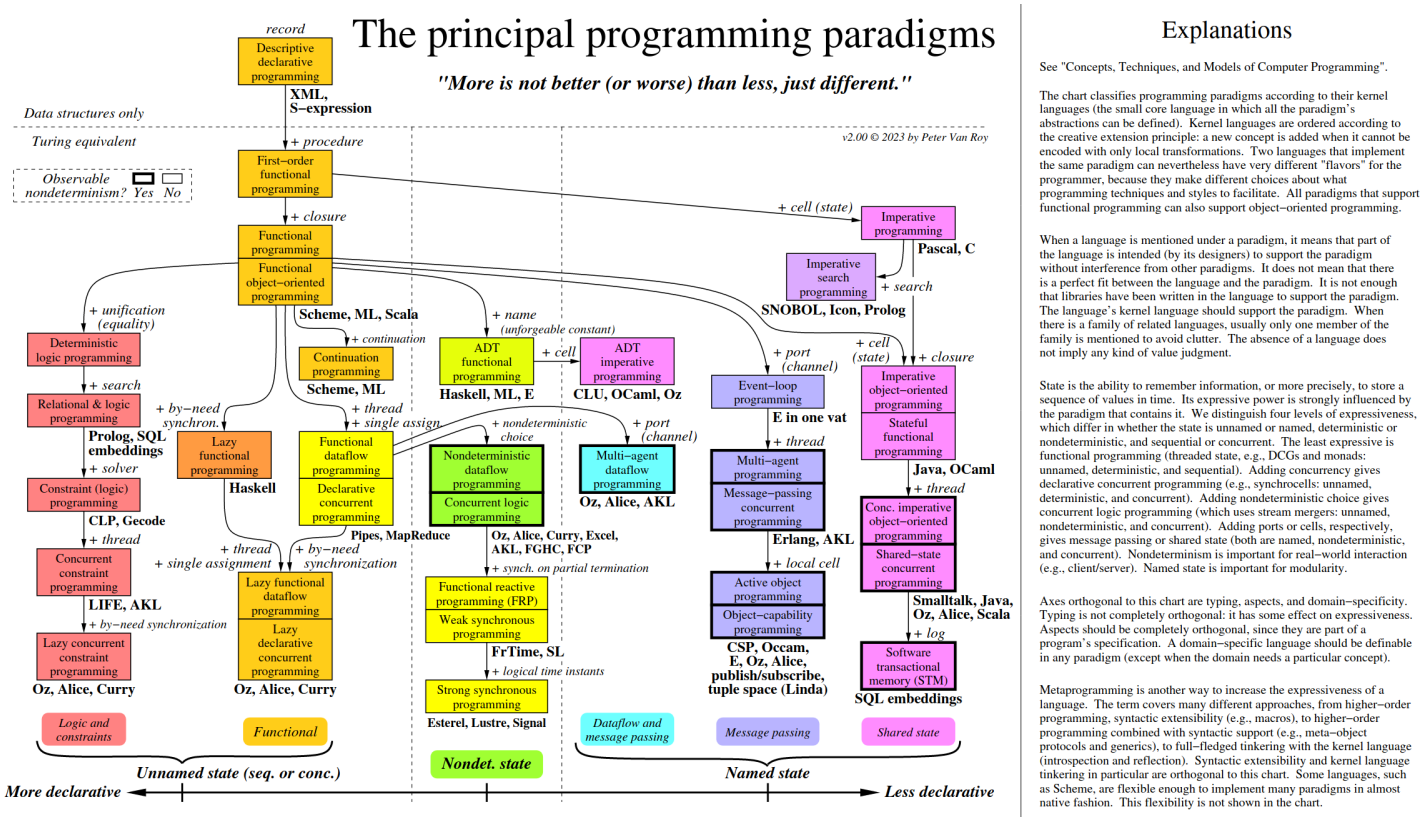


额外补充一下 2000 年之后一些新兴语言的出现时间轴

- 2002: Scratch
- 2003: Grovy, Scala

- 2007: Clojure
- 2009: Go
- 2010: Rust
- 2011: Dart, Kotlin
- 2014: Swift

关于编程语言的特性进化, 可以参考下图 (原始 PDF 可以从 这里 下载)



编译技术的应用

主要包括如下几方面

- 实现高级编程语言
- 针对特定计算机架构进行优化, 主要包括并行 (parallelism) 和内存层级 (memory hierarchis) 两方面
- 设计新的计算机架构
- 程序翻译

  主要包含如下几个方面

  - 二进制码翻译 (Binary Translation), 比如将 ARM 机器码翻译为 x86 代码
  - Hardware Synthesis, 使用高级语言描述硬件, 使用编译器将其转换为硬件布局

    - 数据库查询语句, 常见的就是 SQL 语句
- 编写软件开发的生产力工具
    - 类型检查
    - 边界校验
    - 内存管理

编程语言的要素

- 动态和静态策略

  设计语言编译器时的策略

    - static policy

      编译器可以在编译时做出决策, 解决问题.

    - dynamic policy

      编译器只能在运行时作出决策, 解决问题.

  设计编程语言时的策略

    - static scope

      在编译时通过分析程序能得出 declaration 的作用域.

    - dynmaic scope

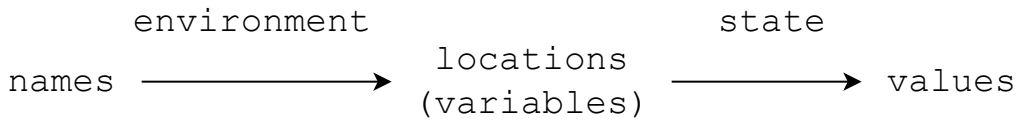
      在运行时才能知道 declaration 所对应的变量 x, 因为一个 declaration 可能指向多个不同的 x .

- 环境和状态

  在讲述环境 (environments) 和状态 (state) 之前, 我们先要了解名称 (names), 标识符 (identifiers) 和变量 (variables) 三者的区别.

  标识符通常是一个字符串, 可以用来指代一个实体 (entity), 比如一个类, 一个类型, 一个过程 (procedure) 或者一个数据对象. 所有的标识符都是名称, 但是并不是所有的名称都是标识符, 因为名称也可以是表达式. 比如说 `x.y` 是一个名称, 该名称用来表示 `y` 是结构体 `x` 的一个字段, 这里 `x` 和 `y` 都是标识符, `x.y` 是一个名称, 但是 `x.y` 不是一个标识符. 像 `x.y` 这样的复合名字 (composite names) 被称之为 限定名称 (qualified names).

  编程语言使用环境来关联 names 和 locations (也可以说是变量), 用状态来关联 locations 和值, 如下图所示



- 作用域 (scope)

  和作用域相关的有两个重要概念: 声明 (declaration) 和定义 (definition). declaration 给出了某个东西的类型, 而 definition 则给出了该东西的值. 比如 `int i;` 给出了 `i` 的 declaration, 而 `i = 1` 给出了 `i` 的值.

  我们以下图的 C++ 程序为例来说明如何寻找名称 `x` 的作用域或者说 `x` 的声明 (declaration).

```
main() {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout << a << b;
        }
        {
            int b = 4;                           B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

  以上图 B_4 中 a, b 为例进行说明如何寻找它们的 declaration.

  首先我们从 a, b 所在的作用域出发, 从小到大列出其所在的作用域可得 B_4, B_2, B_1 (这里没有列出 B_3 是因为 B_3 不包含 a, b). 由于 B_4 包含了 b 的 declaration, 所以这就是 b 所对应的 declaration. 但是 B_4 不包含 a 的 declaration, 所以我们继续在大的作用域 B_2 中找, B_2 也没有, 我们继续往上去 B_1 中找, 在 B_1 中找到了 a 的 declaration, 所以 B_4 中 a 的 declaraion 位于 B_1. 如果我们找到了 B_1 还没有找到 a 的 declaration, 那就说明程序源码有错.

- 参数传递机制

  主要分为三种: call-by-value, call-by-reference, call-by-name.

  学过 C++ 的应该都比较理解前两个, 即值传递和引用传递, 前者是把 value 复制一份传递到函数中, 而后者是把 location 传递到函数中. 那 call-by-name 是什么意思? 这种机制实质上是一种懒加载思想, 它的作用有点像宏展开, 直接把函数体中的形参替换为实参, 只有当代码执行到替换的位置时, 才会去计算实参的值, 类比下面的例子

  ```
  int call_by_name(int x, int y) {
      return x > 0 ? y : 0;
  }
  ```

  如果 `x <= 0` , 那么 y 就不会被计算, 而直接返回 0, 关于 call-by-name 更多可以参考文章 Pass-By-Name Parameter Passing, 由于现代流行的编程语言几乎不用这种方法传递参数, 所以就不多介绍了.