第八章 代码生成

编译器的最后工作是用来生成目标机器的可执行代码，这个可执行代码是源代码语义的忠实体现。

代码生成是编译器最复杂的阶段，因为它不仅依赖于源语言的特征，而且还依赖于

- 目标结构
- 运行时环境的结构
- 运行在目标机器的操作系统的细节信息。

通过定制生成代码以便利用目标机器，如寄存器、寻址模式、管道和高速缓存的特殊性质，代码生成通常也涉及到了一些优化或改善的尝试。

由于代码生成较复杂，所以编译器一般将这一阶段分成几个涉及不同中间数据结构的步骤，其中包括了某种称做中间代码(intermediate code)的抽象代码。

三种常见的形式为：

- 后缀表达式
- 三地址码
- P-代码

中间代码和用于生成中间代码的数据结构

在翻译期间，中间表示(intermediate representation)或IR代表了源程序的数据结构。

抽象语法树作为主要的IR，但抽象语法树与目标代码并不相像（如控制流构造），因此需要一种新形式的 IR。这种与目标代码非常相似的中间表示形式被称为中间代码。然而并非所有中间表示都是中间代码。

所有中间代码都代表了语法树的某种线性化形式

中间代码可以是高水平的，它几乎和语法树一样可以抽象地表示各种操作。它或者还可以非常接近目标代码。它可以使用或不使用目标机器和运行时环境的细节信息，如数据类型的尺寸、变量的地址和寄存器。

中间代码及其有用：

- 产生非常高效的代码
- 使编译器更容易重定向

中间代码生成是编译器的中介部分，它是将源程序翻译成中间表示形式，然后再翻译成目标代码的桥梁。

使用中间代码有两个优点：

- 我们可以在中间代码生成部分之后，将不同的目标代码机器附加到同一前端部分上
- 与机器无关的代码优化器可应用于中间表示法。

中间代码是独立于机器的代码，但它们接近于机器指令。中间代码生成器将源语言中的给定程序转换为等效的中间语言程序。

中间语言可以是多种不同的语言，由编译器的设计者决定。后缀表达式、四地址码（Quadraples）、三地址码、可移植代码和汇编代码都可以用作中间语言

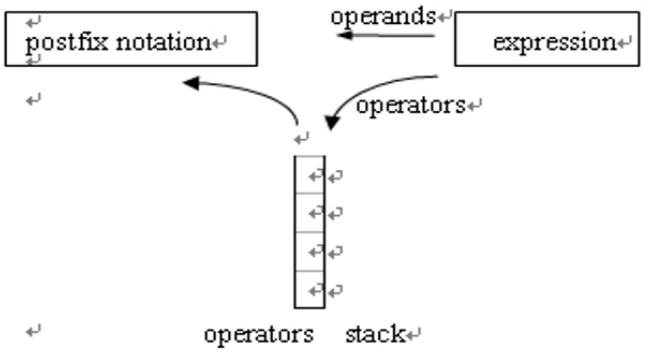
后缀表达式

如果我们能用后缀表达式表示源程序，那么它就很容易被翻译成目标代码，因为目标指令的顺序与后缀表达式中的运算符顺序相同。

表达式 a + b * c 的后缀表达式是 abc* +

- 操作数的顺序与原来是相同的
- 操作符跟在操作数后面，后缀表达式中没有括号
- 运算符按计算顺序依次出现。

迪杰斯特拉方法求后缀表达式（考点）

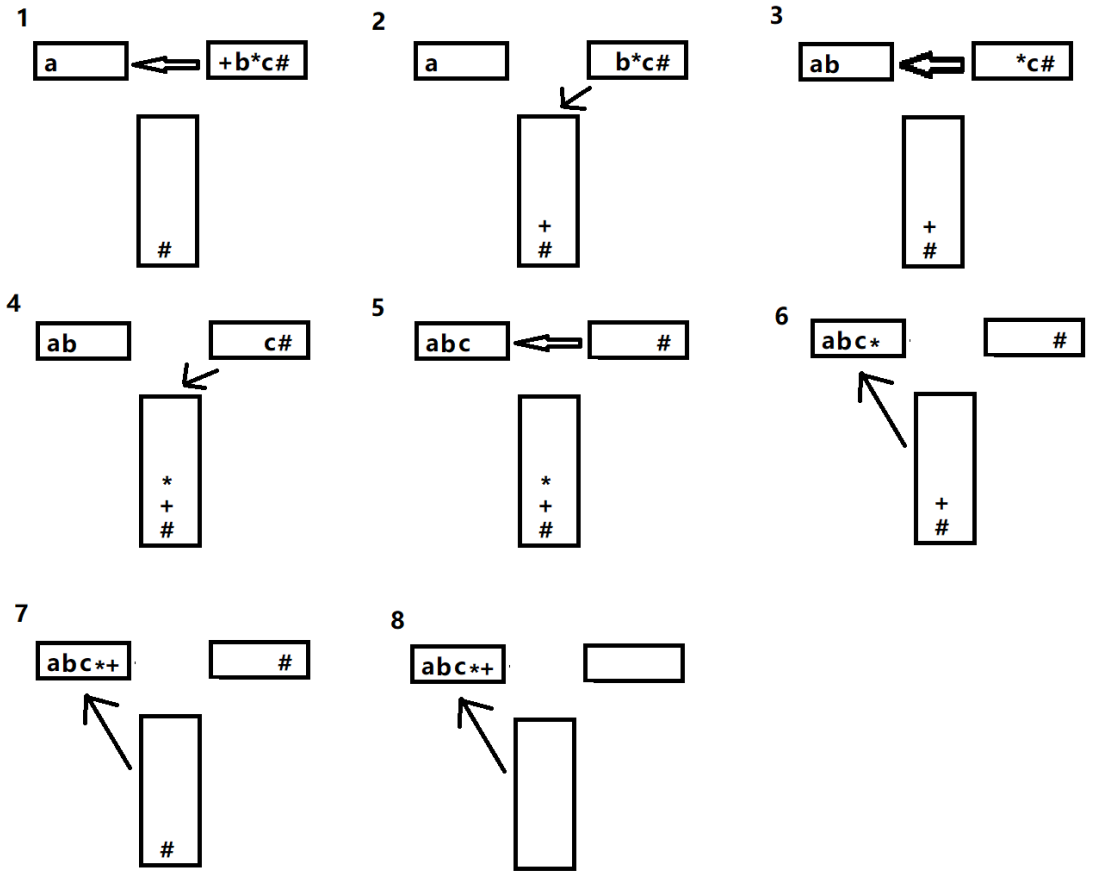


包括两个栈：操作数栈和运算符栈

对表达式的扫描是从左至右的。在扫描开始时，我们将标识符 `#` 压入到运算符栈的底部，同样，我们将标识符 `#` 添加到表达式的末尾，以标明它是表达式的终端。当两个标识符 `#` 相遇时，表示扫描结束，步骤如下：

- step1：如果是操作数，则转到操作数栈
- step2：如果是操作符，则应与操作符栈顶的操作符进行比较。当栈顶操作符的优先级大于或等于扫描操作符时，操作符栈顶的操作符将被弹出并移至左侧。另一方面，当栈顶操作符的优先级小于扫描操作符时，扫描操作符应被压入操作符栈。
- step3：如果是左括号，只需将其推入运算符栈，然后比较括号内的运算符。如果是右括号，括号内的所有运算符都会弹出，而且括号会消失，不会出现在后缀表达式中。
- step4：返回步骤 1，直到两个标识符 # 相遇

eg：a + b * c



三地址码

三个地址码中最基本的指令 `x = y op z`

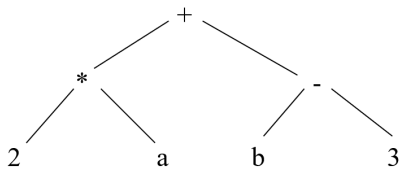
这个用法说明表示了对y和z的值的应用操作符op，并将值赋给 x 。这里的op可以是算术运算符，如+或-，也可以是其他能操作于 y、z 值的操作符。

x 的地址的使用不同于 y、z 的地址的使用。y、z(x不能)可以代表常量或没有运行时地址的字面常量。

eg:

Example

2*a+(b-3)




```
t1=2*a
t2=b-3
t3=t1+t2
```

Left-to-right linearization

```
t1=b-3
t2=2*a
t3=t2+t1
```

Right-to-left linearization

为适应标准程序语言的使用结构，必须为每个结构改变三地址码的形式。

三地址码没有标准形式

用于实现三地址码的数据结构

三地址码通常不被实现成我们所写的文本形式(虽然这是可能的)，相反是将其实现为包含几个域的记录结构。

将整个三地址指令序列实现成链表或数组，它能被保存在内存中并在需要时可以从临时文件中读写。

有4个域是必需的： 1个操作符和3个地址。对于那些少于3个地址的指令，将一个或更多的地域置成null或empty。必须有四个域的三地址码表示叫做四元式（quadruple）

三地址码另一个不同的实现是用自己的指令来代表临时变量，这样地址域从3个减少到了2个。因此在三地址指令中包含3个地址而目标地址总是一个临时变量。如此的三地址码实现称为三元式(triple)。

三元式是代表三地址码的有效方法，空间数量减少了且编译器不需要产生临时变量名；然而，三元式也有一个不利因素：用数组索引代表三元式使得三元式位置的移动变得很困难，而如用链表的话就不存在这个问题。

P-代码

在70年代和80年代早期，P -代码作为由许多Pascal编译器产生的标准目标汇编代码被设计成称作P-机器(P - machine)的假想栈机器的实际代码。

P -机器在不同的平台上由不同的解释器实现。

Pascal编译器容易移植，只需对新平台重写P-机器解释器即可。

P-代码的各种扩展和修改版在许多自然代码的编译器中得到了使用，其中大多数都是针对类Pascal语言的。

组成：

- 一个代码存储器
- 一个未指定的存放命名变量的数据存储器
- 一个存放临时数据的栈
- 一些保持栈和支持执行的寄存器

eg：表达式，代码结束的时候栈里面只有一个值，代表了运算的结果

Example 1

2*a+(b-3)

```
ldc 2    ; load constant 2
lod a    ; load value of variable a
mpi      ; integer multiplication
lod b    ; load value of variable b
ldc 3    ; load constant 3
sbi      ; integer subtraction
adi      ; integer addition
```

eg：赋值语句

Example 2

x:=y+1

```
lda x    ; load address of x
lod y    ; load value of y
ldc 1    ; load constant 1
adi      ; add
sto      ; store top to address
         ; bellow top & pop both
```

P-代码和三地址码的比较：

P -代码在许多方面比三地址码更接近于实际的机器码。P -代码指令也需要较少地址；我们已见过的都是一地址或零地址指令，另一方面，P -代码在指令数量方面不如三地址码紧凑，P -代码不是自包含的，指令操作隐含地依赖于栈，栈的好处是在代码的每一处都包含了所需的所有临时值，编译器不用如三地址码中那样为它们再分配名字

eg（考点）：the postfix notation for expression a*b+(c-d)/e is ab*cd-e/+.

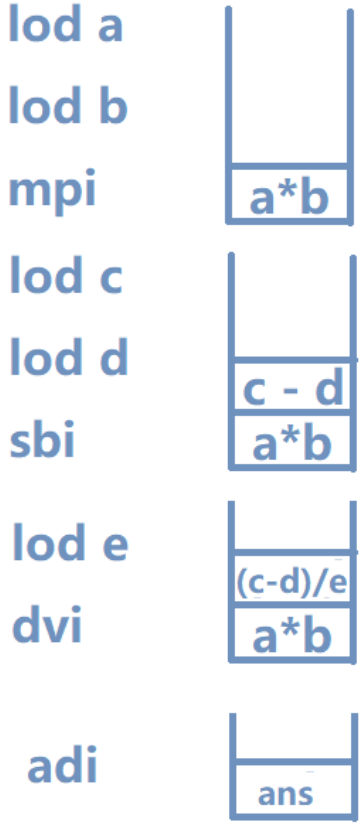
四元式、三元式：

a* b + (c - d) / e

	四元式	三元式
t1 = a * b	(* , a , b, t1)	(0) (* , a, b)
t2 = c - d	(- , c , d, t2)	(1) (- , c, d)
t3 = t2 / e	(/ , t2, e, t3)	(2) (/ , (1), e)
t4 = t1 + t3	(+ , t1, t3, t4)	(3) (+ , (0), (2))

P-code:

a * b + (c - d) / e



代码生成

从中间代码生成目标代码，涉及两个标准技术：

- 宏扩展：用一系列等效的目标代码指令代替每一种中间代码指令

- **静态模拟**：直接模拟中间代码的效果，并生成与这些效果相匹配的目标代码。

eg:

Example

$$exp \rightarrow \mathbf{id} = exp \mid aexp$$
$$aexp \rightarrow aexp + factor \mid factor$$
$$factor \rightarrow (exp) \mid \mathbf{num} \mid \mathbf{id}$$

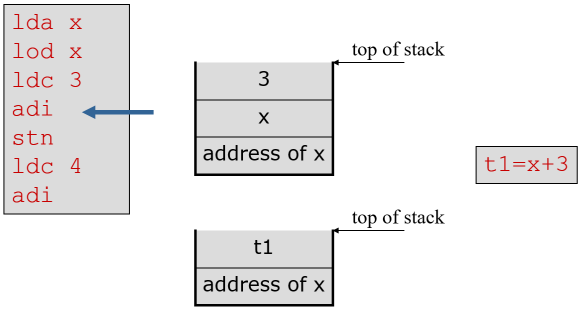
```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

(x=x+3)+4

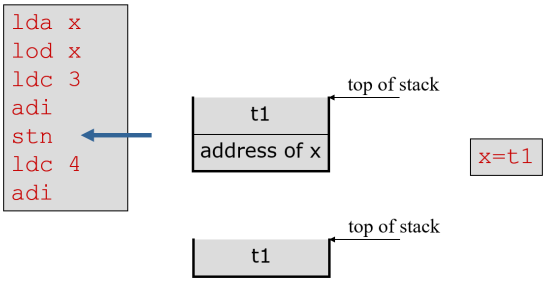
```
t1 = x+3
x = t1
t2 = t1+4
```

代码生成过程：

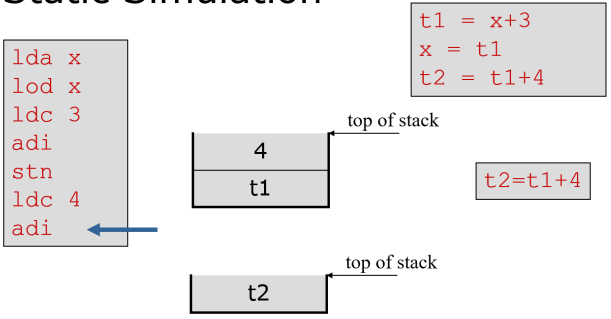
Static Simulation



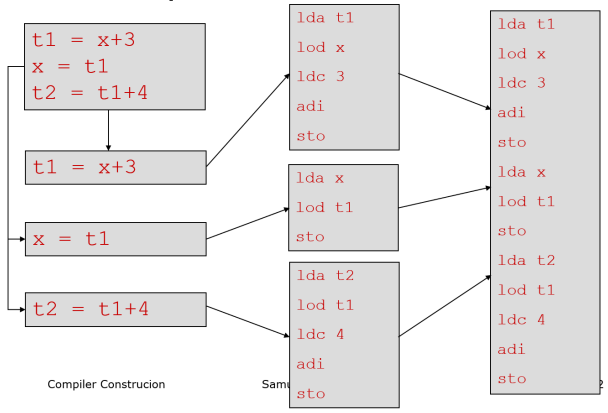
Static Simulation



Static Simulation



Macro Expansion



可以看到最终生成出的代码和一开始写的代码并不一样，这是正常的，代码生成只是按照静态模拟和宏扩展来生成对应的代码，后续还有代码优化的步骤。