

编译原理笔记

陈鸿峥

2020.07*

目录

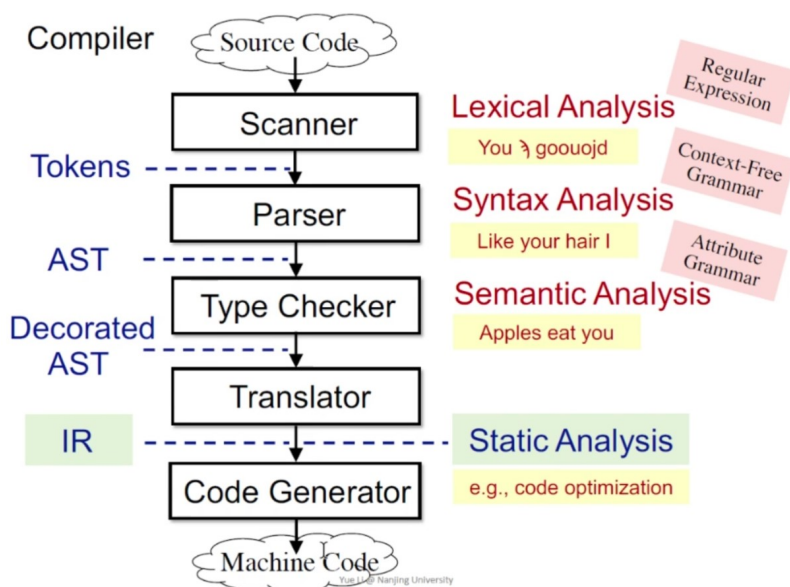
1	简介	2
2	词法分析	2
2.1	基本定义	2
2.2	正则表达式	3
2.3	有限自动机	4
2.4	Regex转DFA	8
2.5	最小化DFA	9
3	语法分析	10
3.1	上下文无关法	10
3.2	NFA转CFG	12
3.3	递归下降	12
3.4	预测分析	14
3.5	移进-规约	15
3.6	语法制导翻译	17
4	语义分析与中间表示	18

本课程采用书目Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques & Tools (2nd ed)*, 即大名鼎鼎的龙书。同时也参考了Stanford CS143: Compilers这门课程。

*Build 20200723

1 简介

编译器的几个阶段如下，前端包括词法(lexical)、语法(syntax)、语义(semantic)分析，中端IR生成、优化，后端代码生成。



编程语言设计的思想：

- 抽象(abstraction): 核心在于信息隐藏(infomation hiding)，只把必要的暴露出来
- 类型(types): 表达抽象、查出常见错误、使程序安全
- 重用(reuse): 开发软件系统中常见的模式（类型参数化、类与继承）

2 词法分析

分离词法分析和语法分析可以简化这两个任务，同时提升编译器的性能与兼容性。

2.1 基本定义

定义 1. 令牌(token)是一个令牌名字与可选属性值构成的对；模式(pattern)描述了每个词素(lexeme)要遵循什么规则；而词素（最小意义单位）则是源程序中一连串满足模式的字母，作为令牌的实例化。

例 1. 考虑C语句

```
printf("Total = %d\n", score);
```

其中printf和score是匹配(match)上令牌id模式的词素，而"Total = %d\n"是匹配上字面值literal的词素。

简单来讲，令牌是一个更大的概念，是同类词素的集合。比如一个令牌comparison的样例词素可以有<=和!=。

定义 2 (字母表与语言). 字母表(alphabet) Σ 是有限符号(symbol)的集合, 如ASCII就是一个字母表。字符串(string) s 是从字母表中抽取的有限符号的序列, $|s|$ 为字符串长度, ϵ 为空串。语言(language)是字符串的可数集合。

例 2. 字母表 $\Sigma = \{0, 1\}$, 则 $\{001, 1001\}$ 和 $\{\}$ 都是定义在 Σ 上的语言。

定义 3 (字符串术语). 前缀(prefix)和后缀(suffix)都可以包括 ϵ 。字串(substring)可通过删除任意前缀和任意后缀(包括零个)获得。真(proper)字串则不包含 ϵ 。子序列(subsequence)是删除零个或多个不一定连续的字母得到的字符串。

语言是一种集合, 故集合运算也适用于语言。

并集(union)	$L \cup M$
连接(concatenation)/交集	LM
柯林闭包(Kleene closure)	$L^* = \bigcup_{i=0}^{\infty} L^i$
正闭包(positive)	$L^+ = \bigcup_{i=1}^{\infty} L^i$

2.2 正则表达式

定义 4 (正则表达式(regular expression, regex)). 正则表达式 r 定义了语言 $L(r)$, 以递归形式定义:

1. 奠基:

- ϵ 是正则表达式, 即 $L(\epsilon) = \{\epsilon\}$
- $a \in \Sigma$ 是正则表达式, 即 $L(\mathbf{a}) = \{a\}$ (这里用斜体代表符号, 粗体代表符号对应的正则表达式)

2. 推论(induction): 若 r 和 s 都是正则表达式给出了语言 $L(r)$ 和 $L(s)$, 则

- $(r)|(s)$ 是正则表达式, 表示 $L(r) \cup L(s)$
- $(r)(s)$ 是正则表达式, 表示 $L(r)L(s)$
- $(r)^*$ 是正则表达式, 表示 $(L(r))^*$
- (r) 是正则表达式, 表示 $L(r)$

正则表达式表示的语言叫做正规集。

有以下运算规定:

- 一元运算符 $*$ 有最高优先级, 左结合
- 连接优先级次之, 左结合
- $|$ 优先级最低, 左结合

定义 5 (正则定义). $d_i \rightarrow r_i$, 其中 d_i 都是名字, 且各不相同。每个 r_i 是 $\Sigma \cup \{d_1, \dots, d_{i-1}\}$ 中符号上的正则表达式。

例 3. 比如C语言的标识符可记为

$$letter_ \rightarrow A|B|\cdots|Z|a|b|\cdots|z|_$$

$$digit \rightarrow 0|1|\cdots|9$$

$$id \rightarrow letter_ (letter_ | digit)^*$$

正则表达式的拓展¹:

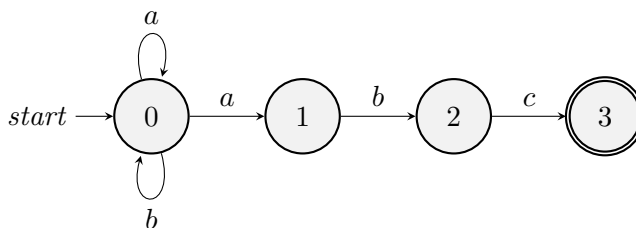
- r^+ 代表一个或多个
- $r^?$ 代表零或一个
- $[a - z]$ 字母类

2.3 有限自动机

2.3.1 确定性/非确定性有限自动机

确定有限自动机(DFA)不可对 ϵ 进行移动, 而且对于每一状态 s , 输入符号 a , 只有唯一一条出边标记为 a ; 而非确定性有限自动机(NFA)可能有多种转换路径, 而且有 ϵ 移动。有限状态集 S , 状态 $s_0 \in S$ 为初始状态(start/initial), $F \subset S$ 为终止状态(accepting/final)。

例 4. 识别语言 $L((a|b)^*abb)$, 下面为一个NFA



判别字符串能否被DFA识别很简单, 只需要读入字符按照状态转移表跳转, 判断末态是不是终态即可。

Algorithm 1 基于DFA的识别算法

```

1:  $s = s_0$ 
2:  $c = nextChar()$ 
3: while ( $c \neq eof$ ) do
4:    $s = move(s, c)$ 
5:    $c = nextChar()$ 
6: if  $s \in F$  then
7:   return "yes"
8: elsereturn "no"

```

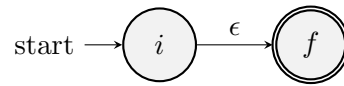
时间复杂度为 $O(|str|)$ 。

¹更多可参见[Regex101](#)

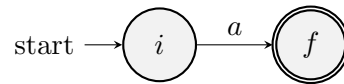
2.3.2 正则表达式转NFA

1. 奠基

- 对于表达式 ϵ ，构建NFA

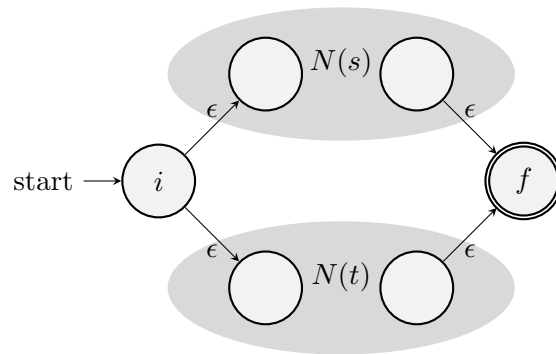


- 对于任意子表达式 $a \in \Sigma$ ，构建NFA

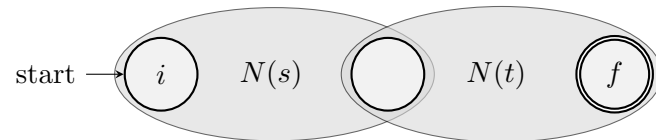


2. 推论

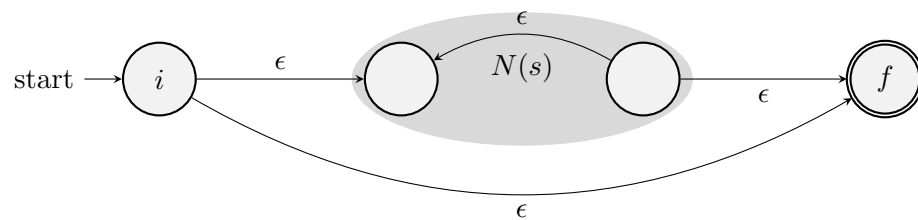
- $r = s|t$ ，取并集



- $r = st$ ，取连接



- $r = s^*$ ，Kleene闭包



2.3.3 NFA转DFA

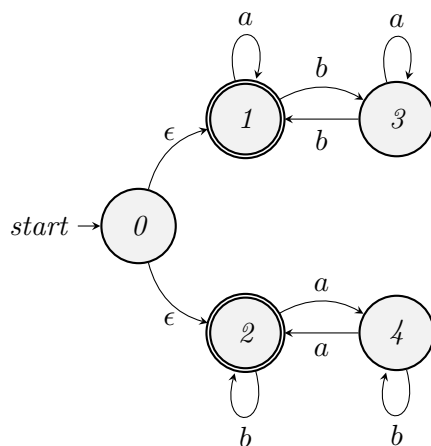
定义 6 (ϵ 闭包及 $move$). ϵ 闭包是可通过NFA的 ϵ 边转换的状态。 $move(T, a)$ 为状态 $s \in T$ 通过输入符号 a 可到达的新的状态。

Algorithm 2 子集构造 (NFA转DFA)

Require: NFA N **Ensure:** DFA D (与 N 接受相同的语言)

- 1: ϵ -closure(s_0)是 $Dstates$ 的唯一状态, 且未被标记(unmarked)
 - 2: **while** 在 $Dstates$ 中还有未被标记的状态 T **do**
 - 3: 标记 T
 - 4: **for** 每一个输入符号 a **do**
 - 5: $U = \epsilon$ -closure(move(T, a))
 - 6: **if** $U \notin Dstates$ **then**
 - 7: 将 U 作为未标记的状态加入 $Dstates$
 - 8: $Dtran[T, a] = U$
-

例 5. 考虑以下NFA:

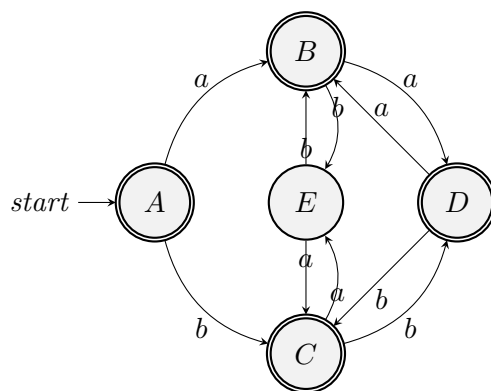


1. 这一NFA接受什么语言 (用自然语言描述)?
2. 构造接受同一语言的DFA.

分析. 1. 含有偶数个 a 或偶数个 b 的由 a 、 b 构成的字符串, 或者全是 a 或全是 b

2. 由subset construction算法构造如下

NFA	DFA	a	b
$\{0, \underline{1}, 2\}$	A	$\{1, 4\}$	$\{2, 3\}$
$\{\underline{1}, 4\}$	B	$\{1, 2\}$	$\{3, 4\}$
$\{2, 3\}$	C	$\{3, 4\}$	$\{1, 2\}$
$\{\underline{1}, 2\}$	D	$\{1, 4\}$	$\{2, 3\}$
$\{3, 4\}$	E	$\{2, 3\}$	$\{1, 4\}$



直接用NFA识别语言算法如下，需要每次算所有当前可能状态执行动作 c 后的 ϵ 闭包。

Algorithm 3 用NFA识别语言

```

1:  $S = \epsilon\text{-closure}(s_0)$ 
2:  $c = \text{nextChar}()$ 
3: while  $c \neq \text{eof}$  do
4:    $S = \epsilon\text{-closure}(\text{move}(S, c))$ 
5:    $c = \text{nextChar}()$ 
6: if  $S \cap F \neq \emptyset$  then
7:   return “yes”
8: else
9:   return “no”

```

定理 1. DFA ， NFA 和正则表达式三者的描述能力是一样的。

但从NFA转为DFA可能导致状态数的指数增长。

例 6. $L_n = (a \mid b)^* a (a \mid b)^{n-1}$ ，与此NFA等价的DFA状态数必不少于 2^n 。

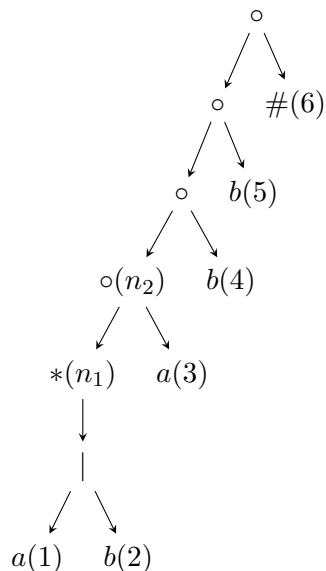
分析. 反证法。假设存在一个DFA D 接受语言 L_n ，且状态数少于 2^n 。构造 2^n 个长度为 n 的字符串

$aa \cdots a$
 $aa \cdots 1$
 \dots
 $bb \cdots a$
 $bb \cdots b$

由于 D 的状态数少于 2^n ，故上面必存在两个不同的字符串 s 和 t ，它们在DFA上会走到同一状态。因为 s 和 t 不等，因此总存在 i ，使得 $s[i] \neq t[i]$ 。不妨设 $s[i] = 0$ ， $t[i] = 1$ ，令 $s' = s + (n-1)$ 个 a ， $t' = t + (n-1)$ 个 a 。由 L_n 的表达式， s' 应该走到接受状态，而 t' 应该走到非接受状态。但由于 s 和 t 走到同一状态，那么它们再走 $(n-1)$ 个 a 也应该到达同一状态，但这个状态既是接受状态又是非接受状态，因此矛盾。

2.4 Regex转DFA

构造正则表达式的语法树，以#结尾



*为star, |为or, \circ 为cat

- $nullable(n)$: ϵ 包含在子树中则为真
- $firstpos(n)$: 符合regex子树的字符串中第一个字符可能出现的位置
- $lastpos(n)$: 符合要求字符串最后一个字符可能出现的位置
- $followpos(p)$: 紧跟 p 可能的位置

例 7. 结点 n_1 代表 $(a|b)^*$, 结点 n_2 代表 $(a|b)^*a$

- $nullable(n_1) = true$
- $firstpos(n_2) = \{1, 2, 3\}$
- $lastpos(n_2) = \{3\}$
- $followpos(1) = \{1, 2, 3\}$

计算 $followup$ 的两条法则:

- 若 n 为cat结点, 则对于左子树 c_1 的所有 $i \in lastpos(c_1)$, 有右子树 $firstpos(c_2) \in followpos(i)$
- 若 n 为star结点, 则对于所有 $i \in lastpos(n)$, 有 $firstpos(n) \in followpos(i)$

构建 $followpos$ 的过程实际上是深搜的过程, 可构造出一个有向图表示状态迁移。

Algorithm 4 Regex转DFA

Require: 正则表达式 r

Ensure: DFA D 可识别 $L(r)$

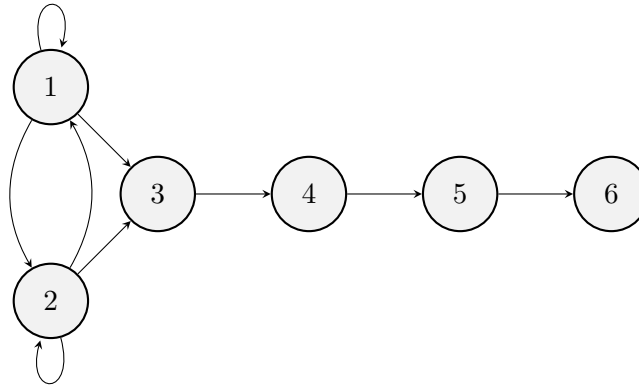
- 1: 语法树 T 的根节点为 n_0 , $firstpos(n_0)$ 是 $Dstates$ 的唯一状态, 且未被标记(unmarked)
 - 2: **while** 在 $Dstates$ 中还有未被标记的状态 S **do**
 - 3: 标记 S
 - 4: **for** 每一个输入符号 a **do**
 - 5: $U = \bigcup_{\text{对应}a\text{的位置}p \in S} followpos(p)$
 - 6: **if** $U \notin Dstates$ **then**
 - 7: 将 U 作为未标记的状态加入 $Dstates$
 - 8: $Dtran[S, a] = U$
-

例 8. 考虑正则表达式 $(a | b)^* a b b \#$
1 2 3 4 5 6

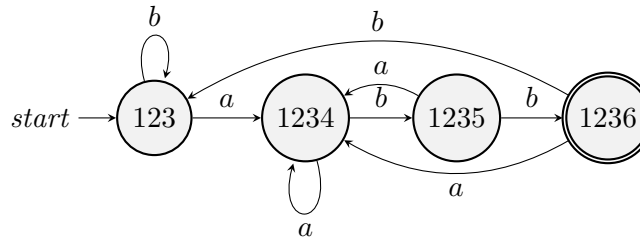
分析. 由表达式其实可以直接得到 $followpos$ 函数

<i>position</i>	<i>followpos(i)</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	\emptyset

进而构造出一个有向图



然后可得DFA



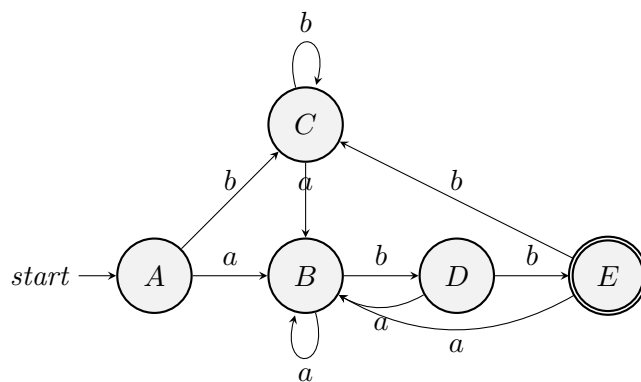
2.5 最小化DFA

定义 7 (区别(distinguish)). 字符串 w 区别状态 s 和 t , 如果DFA M 从状态 s 出发, 对输入串 w 进行状态转换, 最后停在某个接受状态; 从 t 出发, 对输入串 w 进行状态转换, 停在一个非接受状态; 反之亦然。

定理 2. 每一个正则集都可以唯一由一个状态数最少的DFA识别。

分析. 反证法。设算法得到的DFA为 D , 假设存在另一个DFA D' , D' 和 D 接受同一语言, 并且 D' 的状态数比 D 更少。设 D 的起始状态为 S , D' 的起始状态为 S' , 则 S 与 S' 不可区分。如果对于 S 和输入符号 a , 在 D 中迁移到状态 A ; 对于 S' 和输入符号 a , 在 D' 中迁移到状态 A' , 则 A 与 A' 不可区分。依此类推可知对于 D 中的任一状态 T , 在 D' 中都有一个状态 T' 与 T 不可区分。又由于 D 的状态数多于 D' 的状态数, 所以 D 中至少存在两个状态 T_1 和 T_2 , 使得 D' 中的一个状态 T 与它们均不可区分。因此 T_1 和 T_2 也不可区分, 于是矛盾。

例 9. 如下状态转移图



分析. 初始划分 Π 包括两个组: 接受状态组(E)和非接受状态组($ABCD$)。构造 Π_{new} , 先考虑(E), 仅一个状态, 不可划分, 仍将(E)放回 Π_{new} 。然后考虑($ABCD$), 对于输入 a , 这些状态都转换到 B , 分组($ABCD$)不变; 但对于输入 b , A 、 B 和 C 都转换到状态组($ABCD$)的一个成员, 而 D 转换到另一组成员 E 。因此, 在 Π_{new} 中, 状态组($ABCD$)需要分裂为两个新组(ABC)和 D , $\Pi_{new} = (ABC)(D)(E)$ 。继续执行下一轮操作, 最终得到 $\Pi_{final} = (AC)(B)(D)(E)$ 。因此选择 A 作为(AC)的代表, 其他不变, 可得到简化的自动机。

	a	b
A	B	A
B	B	D
D	B	E
E	B	A

3 语法分析

3.1 上下文无关法

语法分析需要解决: 从词法分析中获得的每个属性字(token)在语句中承担什么角色, 同时检查语句是否符合程序语言的语法。

很多语言并非是非正则的, 比如匹配的括号串 $\{()^i \mid i \geq 0\}$, 原因是FA不能记住其访问某一状态的次数, 因此需要有更加强大的语言。

定义 8 (上下文无关法(context-free grammar, CFG)). 包括四部分

- 终端符号(*terminal*)的集合 T
- 非终端符号的集合 N
- 唯一的开始符号 $S \in N$
- 若干以下形式的产生式(*production*)

$$X \rightarrow Y_1 Y_2 \dots Y_n$$

其中 $X \in N$ 且 $Y_i \in T \cup N \cup \{\epsilon\}$ 。多个左侧相同的产生式右侧可用 $|$ 合并。

定义 9 (推导(*derivation*)). 从开始符号开始, 每一步推导就是用一个产生式的右方取代左端的非终端符号。

CFG定义语言的能力比正则表达式强很大原因是它引入了递归的因素。

例 10. 用上下文无关文法定义下列语言：

- $L = \{0^n 1^n \mid n \geq 1\}$: $E \rightarrow 0E1 \mid 01$
- 只含有0和1的回文串: $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$
- 只含有(和)的匹配括号串: $E \rightarrow (E) \mid EE \mid \epsilon$
- 最左推导: 每步推导都替换最左侧的非终端符号

$$E \xRightarrow{lm} -E \xRightarrow{lm} -(E) \xRightarrow{lm} -(E + E) \xRightarrow{lm} -(id + E) \xRightarrow{lm} -(id + id)$$

- 最右推导: 每步推导都替换最右侧的非终端符号

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$$

定义 10 (二义性). 如果对于一个文法, 存在一个句子, 对这个句子可以构造两棵不同的分析树, 那么我们称这个文法为二义的。

看语法分析树的叶子结点能不能连成句子。

例 11. 对于文法 $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$ 及句子 $id + id * id$, 有以下两种推导:

$ \begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned} $	$ \begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned} $
---	---

文法二义性的消除可通过引入更多的产生式。

例 12. $E \rightarrow E + E \mid E * E \mid (E) \mid id$ 是有二义的, 因为不知道应该先算加法还是乘法。可将其改为

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid id
 \end{aligned}$$

其中 E 为 *Expression*, T 为 *Term*, F 为 *Facotr*, 即可消除二义性 (必然得先算乘法)。相当于先算 F , 再算 T , 最后算 E 。

例 13. 悬挂的 *if-else*: `if E1 then if E2 then E3 else E4`, 可以令 `else` 匹配最近的 `then`。

$$\begin{array}{ll}
 E & \rightarrow MIF \quad // \text{所有的 } then \text{ 都被匹配} \\
 & | UIF \quad // \text{仅有一些 } then \\
 MIF & \rightarrow \text{if } E \text{ then } MIF \text{ else } MIF \\
 & | OTHER \\
 UIF & \rightarrow \text{if } E \text{ then } E \\
 & | \text{if } E \text{ then } MIF \text{ else } UIF
 \end{array}$$

并不是所有上下文无关文法都可以做到无二义，也无法判断一个上下文无关文法是否是二义的。

3.2 NFA转CFG

1. 对于NFA的每一状态 i ，创建非终态 A_i
2. 若状态 i 在输入 a 上有转换边到状态 j ，则添加生成式 $A_i \rightarrow aA_j$ ；若状态 i 在输入 ϵ 上转换到状态 j ，则添加生成式 $A_i \rightarrow A_j$
3. 若 i 是接受状态，则添加 $A_i \rightarrow \epsilon$
4. 若 i 是初始状态，则令 A_i 为语法的初始符号

定义 11 (右线性文法). 如果每个产生式都属于下列形式之一

$$A \rightarrow aB \quad A \rightarrow a \quad A \rightarrow \epsilon$$

则这样的文法称为右线性文法

定义 12 (左线性文法). 如果每个产生式都属于下列形式之一

$$A \rightarrow Ba \quad A \rightarrow a \quad A \rightarrow \epsilon$$

则这样的文法称为左线性文法

在处理程序时，上下文无关文法存在局限性，无法解决诸如以下问题：

- 变量先声明，再使用
- 调用函数时，实参个数和形参个数一致

都得留到语义分析阶段才解决。

3.3 递归下降

递归下降语法翻译即从顶层的非终端符号 E 开始，顺序尝试 E 的所有规则，完整例子可见[此文档](#)，但需要先消除左递归。

定义 13 (左递归). 对于非终端符号 A 有生成式 $A \rightarrow A\alpha$ ，则该文法是左递归的。

消除左递归的方法：

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \beta_n \implies \begin{aligned} A &\rightarrow \beta_1 A' \mid \cdots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \cdots \mid \alpha_m A' \epsilon \end{aligned}$$

定义 14 (FIRST集与FOLLOW集). $FIRST(\alpha)$ 集为从 α 中推导出来的字符串第一个终端符号的集合, 若 $\alpha \rightarrow \epsilon$, 则 $\epsilon \in FIRST(\alpha)$; 若 $A \rightarrow c\gamma$, 则 $c \in FIRST(A)$ 。 $FOLLOW(A)$ 集为可以出现在 A 右侧的终端符号的集合。若 A 是最右端的符号, 则字符串结束符号 $\$ \in FOLLOW(A)$ 。

算法 1. 计算 $FIRST(X)$ 集

1. 如果 X 是终端符号, 则 $FIRST(X) = \{X\}$
2. 如果 X 是非终端符号, 且 $X \rightarrow Y_1 Y_2 \cdots Y_k$ 。
 - 若 $Y_1 \cdots Y_{i-1} \rightarrow \epsilon$, 则将 $a \in FIRST(Y_i)$ 放入 $FIRST(X)$ 。
 - 若 $\epsilon \in FIRST(Y_j), j = 1, 2, \dots, k$, 则将 ϵ 放入 $FIRST(X)$ 中。
3. 若 $X \rightarrow \epsilon$ 是生成式, 将 ϵ 放入 $FIRST(X)$ 中

算法 2. 计算 $FOLLOW(A)$ 集

1. 将 $\$$ 放入 $FOLLOW(S)$, 其中 S 是开始符号
2. 如果有生成式 $A \rightarrow \alpha B \beta$, 那么 $\forall a \in FIRST(\beta), a \neq \epsilon: a \in FOLLOW(B)$
3. 如果有生成式 $A \rightarrow \alpha B$, 或生成式 $A \rightarrow \alpha B \beta$, 且 $\epsilon \in FIRST(\beta)$, 则 $\forall a \in FOLLOW(A): a \in FOLLOW(B)$

简而言之, $FOLLOW$ 集看下一符号的 $FIRST$, 如果 ϵ 在下一符号的 $FIRST$ 集中, 则看生成式左端的 $FOLLOW$ 集。

另一种方式:

1. $\$ \in FOLLOW(S)$
2. $\forall A \rightarrow \alpha X \beta: FIRST(\beta) - \{\epsilon\} \subset FOLLOW(X)$
3. $\forall A \rightarrow \alpha X \beta, \epsilon \in FIRST(\beta): FOLLOW(A) \subset FOLLOW(X)$

- Recall the grammar

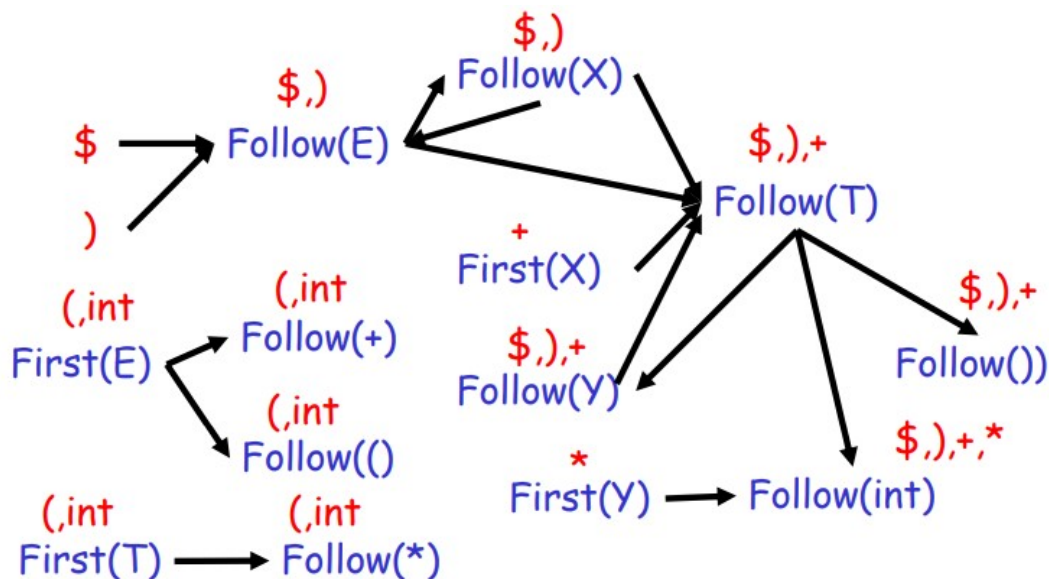
$$E \rightarrow T X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$
- $) \in \text{Follow}(E)$
- $\text{Follow}(T) \subseteq \text{Follow}(Y)$
- $\text{Follow}(X) \subseteq \text{Follow}(E)$
- $\text{Follow}(Y) \subseteq \text{Follow}(T)$



29

3.4 预测分析

LL(1)文法

- 第一个L: 输入字符串从左边开始扫描
- 第二个L: 得到的推导是最左推导
- (1): 向前看1个输入符号 (或单词)

递归下降在每一步都会有多种生成式的选择, 这会导致大量的回溯。而在LL(1)文法中, 每一步都只有一种生成式的选择, 避免了回溯。

左因子分解(left-factoring)将生成式的共同前缀分解出来。

例 14. 考虑以下文法

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int \mid int * T \mid (E)$$

共同前缀分解后即得

$$E \rightarrow TX$$

$$X \rightarrow +E \mid \epsilon$$

$$T \rightarrow int Y \mid (E)$$

$$Y \rightarrow *T \mid \epsilon$$

有 $LL(1)$ 语法表，其中最左列为最左非终端符号，最上行为下一输入符号，表格内容为使用的右端生成式。

	<i>int</i>	*	+	()	\$
<i>E</i>	<i>TX</i>			<i>TX</i>		
<i>X</i>			<i>+E</i>		ϵ	ϵ
<i>T</i>	<i>int Y</i>			<i>(E)</i>		
<i>Y</i>		<i>*T</i>	ϵ		ϵ	ϵ

基于表的预测语法分析，用栈实现。

Algorithm 5 Table-Driven Predictive Parsing

```

1: ip=0
2: X=stack.top()
3: while X ≠ $ do
4:   if X == w[ip] then
5:     stack.pop(); ip++;
6:   else
7:     if X is a terminal or M[X,a]=∅ then
8:       Error()
9:     else
10:      Output production M[X,a]=  $X \rightarrow Y_1 Y_2 \cdots Y_k$ 
11:      stack.pop()
12:      push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack
13:   X=stack.top()
14: if w[ip] != '$' then
15:   Error()

```

3.5 移进-规约

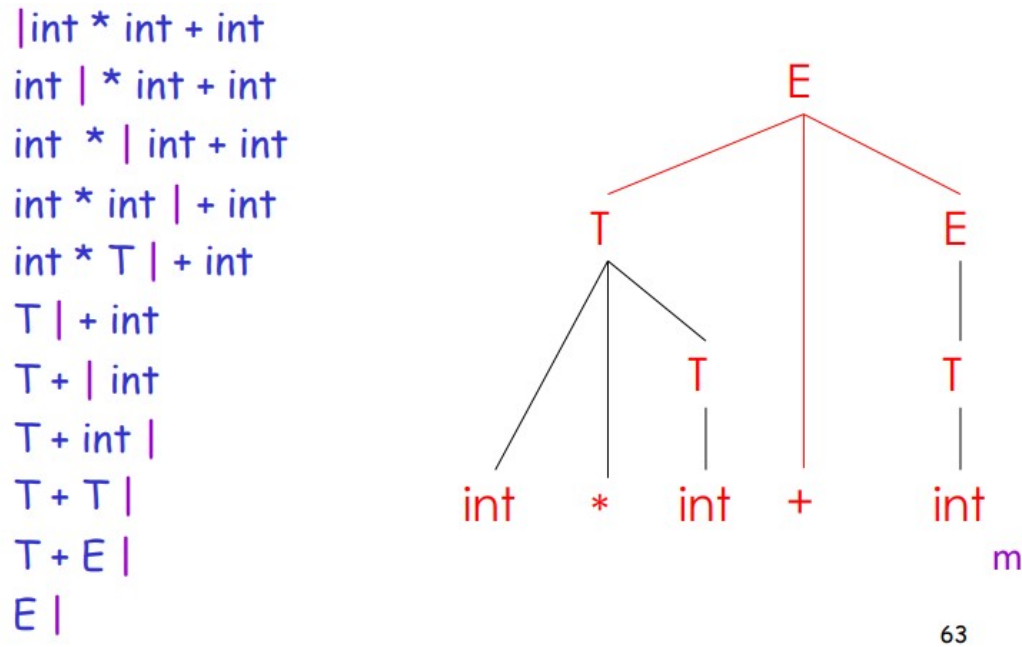
自底向上的语法分析采用两种动作：

- 移进(shift)：将|向右移动一格

$$ABC \mid xyz \Longrightarrow ABCx \mid yz$$

- 规约(reduce): 在字符串右侧逆向应用生成式

$$Cbxy \mid ijk \implies CbA \mid ijk$$



移进将终端符号移入栈中，规约将生成式的右端符号弹出，将生成式的左端非终端符号推入。

定义 15 (句柄(handle)). *A handle is a string that can be reduced and also allows further reductions back to the start symbol.*

定义 16 (活前缀(viable prefix)). α 是活前缀若存在 ω 使得 $\alpha \mid \omega$ 是移进-规约语法分析器的状态。

LR(0)语法:

- 栈包含 α ，下一输入是 t ，DFA 在输入 α 上终止在状态 s
- 当 s 包含 $X \rightarrow \beta$ 的项时进行规约
- 当 s 包含 $X \rightarrow \beta.t\omega$ 的项时移进

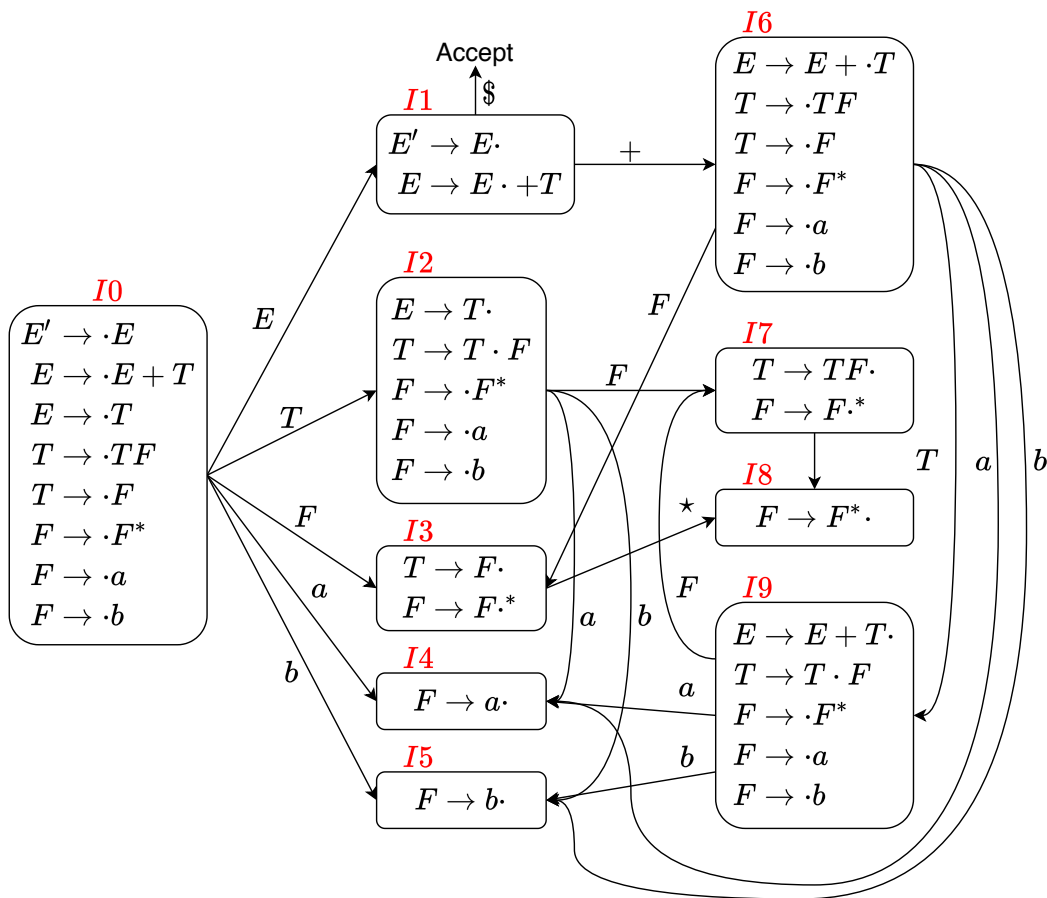
LR(0)可能存在以下两种冲突:

- 规约-规约冲突: $X \rightarrow \beta$. 且 $Y \rightarrow \omega$.
- 移进-规约冲突: $X \rightarrow \beta$. 且 $Y \rightarrow \omega.t\delta$

例 15 (用NFA识别活前缀). 考虑以下文法:

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow TF$
- (4) $T \rightarrow F$
- (5) $F \rightarrow F^*$
- (6) $F \rightarrow a$
- (7) $F \rightarrow b$

构造识别这一文法所有活前缀(*viable prefixes*)的LR(0)自动机.



SLR(simple left-to-right scan)用启发式算法提升了LR(0)移进规约的效率, 减少冲突。

- 栈包含 α , 下一输入是 t , DFA在输入 α 上终止在状态 s
- 当 s 包含 $X \rightarrow \beta$ 的项且 $t \in FOLLOW(X)$ 时进行规约
- 当 s 包含 $X \rightarrow \beta.t\omega$ 的项时移进

3.6 语法制导翻译

抽象语法树(Abstract Syntax Trees, AST)是将原本语法树中冗余的成分给去除, 比如左右括号原本

都是各自一个结点，但在AST中不会呈现。

语法制导翻译(syntax-directed translation)给语法符号提供了属性(attribute)，给生成式提供了动作(action)。

例 16. 对下列语法进行求值

$$E \rightarrow int \mid E + E \mid (E)$$

有语法制导定义

$$\begin{aligned} E \rightarrow int & \quad E.val = int.val \\ E \rightarrow E_1 + E_2 & \quad E.val = E_1.val + E_2.val \\ E \rightarrow (E_1) & \quad E.val = E_1.val \end{aligned}$$

- 综合属性(synthesized): 从后代计算得到
- 继承属性(inherited): 从语法树的父亲或兄弟中计算得到

4 语义分析与中间表示

- 高层中间表示: 语法树、有向无环图(DAG)，用于静态类型检查
- 低层中间表示: 三地址码，适合机器相关的任务（寄存器分配、指令选择）

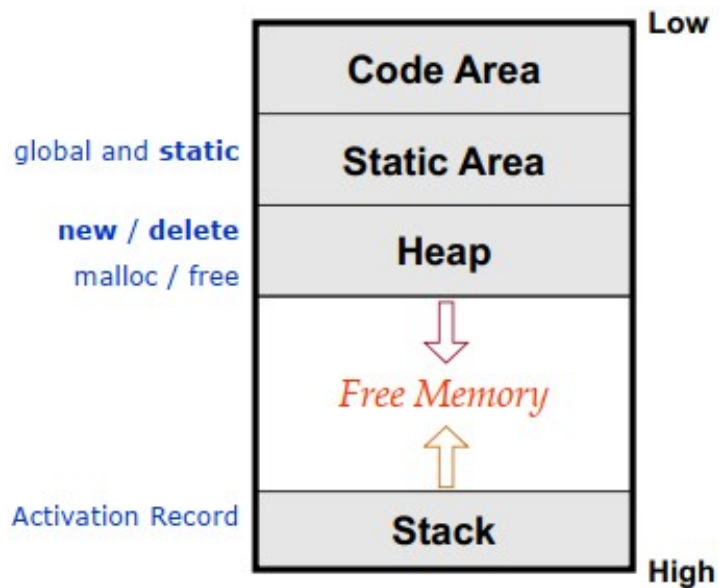
```
x = y op z // arithmetic and logical
x = op y // negation and conversion
x = y // copy
goto L // unconditional jump
if x goto L // conditional jump
if False x goto L // conditional jump
if x op y goto L // relational operation
param x1 // parameter passing
param x2
...
param xn
call p, n // procedure call
y = call p, n // function call
return y // return a value
x = y[i] // indexed copy, i is the offset
x[i] = y
x = &y // address and pointer assignment
x = *y
*x = y
```

top指代当前的符号表，gen代表生成中间代码，||代表代码的连接。

$S \rightarrow id = E$	<code>S.code = E.code gen(top.get(id.lexeme) '=' E.addr)</code>
$E \rightarrow E_1 + E_2$	<code>E.addr = new Temp() E.code = E1.code E2.code gen(E.addr '=' E1.addr '+' E2.addr)</code>
$E \rightarrow -E_1$	<code>E.addr = new Temp() E.code = E1.code gen(E.addr '=' minus E1.addr)</code>
$E \rightarrow (E_1)$	<code>E.addr = E1.addr E.code = E1.code</code>
$L \rightarrow L_1[E]$	<code>L.array = L1.array L.type = L1.type.element t = new Temp() L.addr = new Temp() gen(t '=' E.addr '*' L.type.width) gen(L.addr '=' L1.addr '+' t)</code>
$S \rightarrow \text{if } (B) S1 \text{ else } S2$	<code>B.true = new Label() B.false = new Label() S1.next = S2.next = S.next S.code = B.code label(B.true) S1.code gen('goto' S.next) label(B.false) S2.code</code>

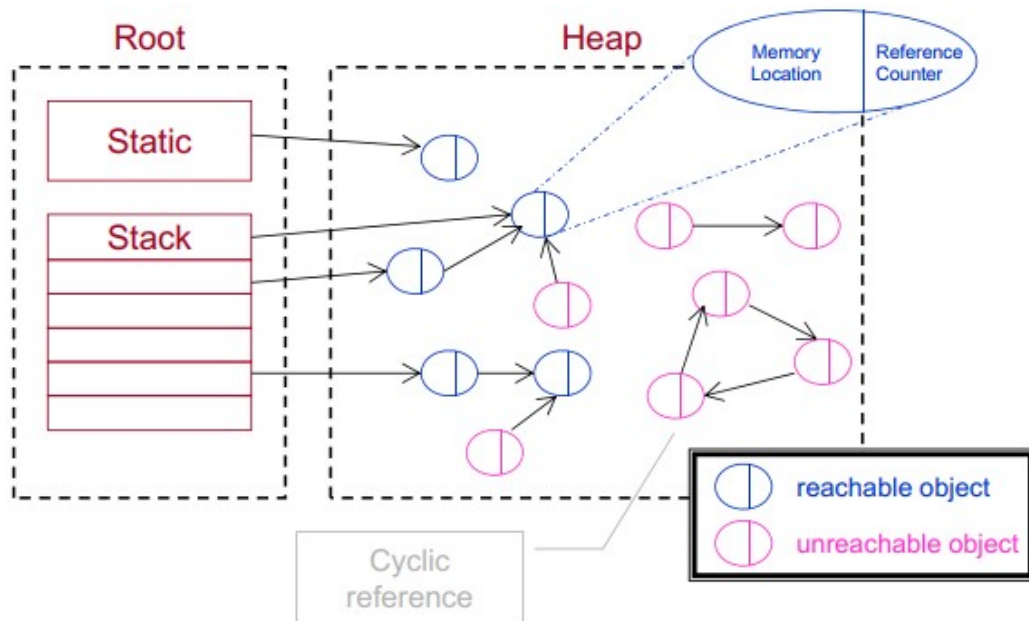
5 运行时系统

5.1 存储管理

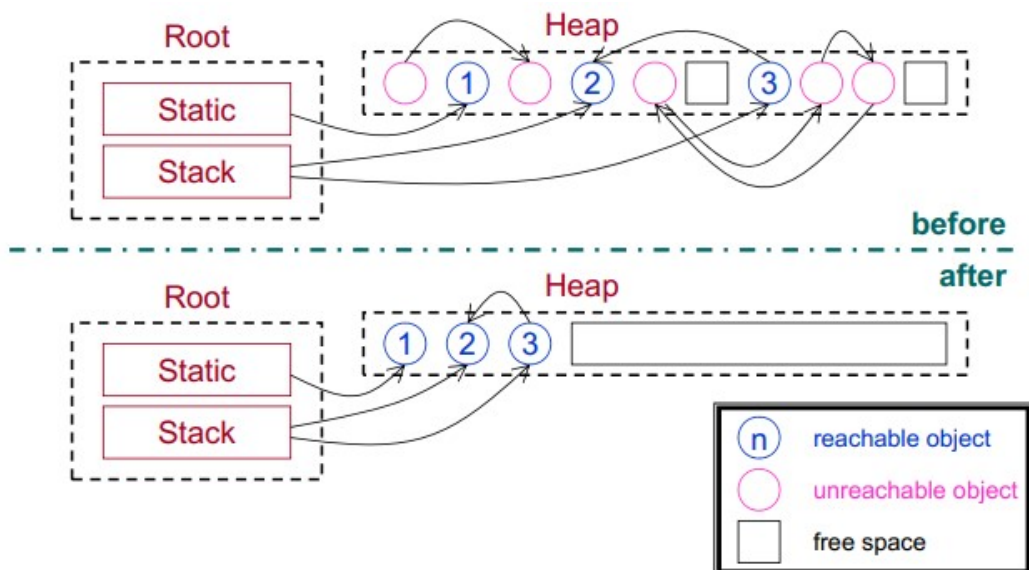


5.2 垃圾回收

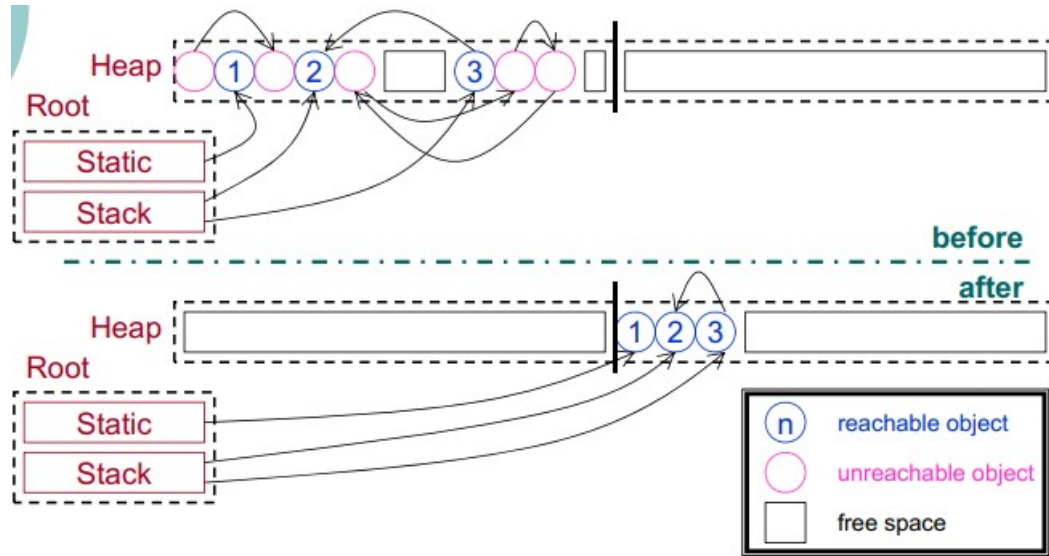
- 引用计数(reference counting): 创建加1, 删除减1
 - 简单、立即增量式回收
 - 不能回收循环引用的示例



- 标记扫描(mark and sweep): 做图深搜找连通块
 - 有办法清除循环引用
 - 大量垃圾时效率低, 无法满足实时应用
- 标记压缩(mark and compact): 标记, 计算新地址, 拷贝对象到新地址并更新引用



- 拷贝收集(copying collector): 堆被划分为两个区域, 可达对象一旦被发现就会立即被移动, 但不可达对象不做改动



JVM采用了两代(young & old)的方式, 对于年轻的对象采用拷贝收集, 对于老的对象则采用标记压缩。