

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	5
2.4	Regex转DFA	9
2.5	最小化DFA	11
3	语法分析	12
3.1	上下文无关法	12
3.2	NFA转CFG	13
3.3	递归下降	14
3.4	自顶向下分析	17
3.5	自底向上分析	20
3.6	语法制导翻译	24
4	语义分析与中间表示	24
5	运行时系统	26
5.1	存储管理	26
5.2	垃圾回收	26

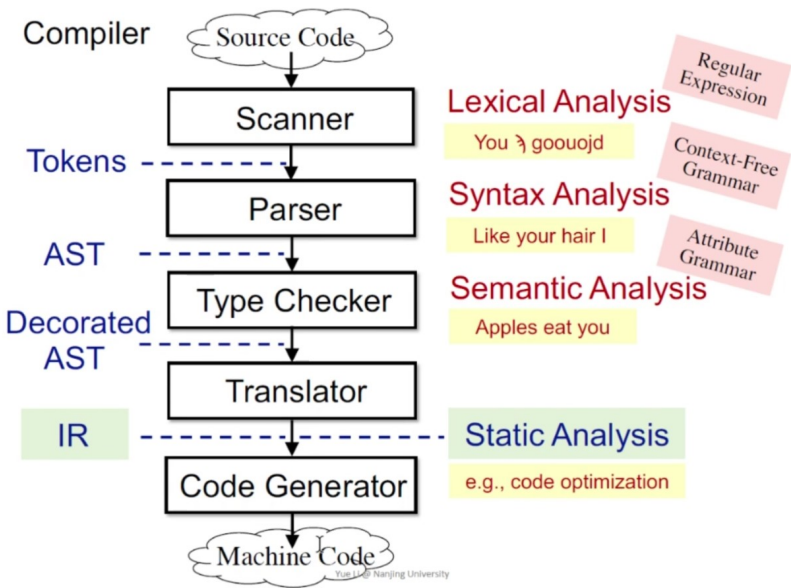
*Build 20200726

6	代码生成及优化	27
6.1	代码生成	27
6.2	代码优化	28

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。

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)	$L^* = \cup_{i=0}^{\infty} L^i$
正闭包(positive)	$L^+ = \cup_{i=1}^{\infty} L^i$

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 $L(a) = \{a\}$ (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。如果两个正则 r 和 s 定义了相同的正则集, 则记作 $r = s$ 。

正则表达式的拓展¹:

¹更多可参见[Regex101](#)

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

有以下运算规定：

- 一元运算符*有最高优先级，左结合（也包括+、?等扩展）
- 连接优先级次之，左结合
- |优先级最低，左结合

等价规则：

- 连接具有分配律： $r(st) = rs|rt$, $(s|t)r = sr|tr$
- ϵ 在闭包里被保证： $r^* = (r|\epsilon)^*$
- 闭包幂等(idempotent): $r^{**} = r^*$

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

$$\begin{aligned} \text{letter_} &\rightarrow A|B|\dots|Z|a|b|\dots|z|_ \\ \text{digit} &\rightarrow 0|1|\dots|9 \\ \text{id} &\rightarrow \text{letter_}(\text{letter_}|\text{digit})^* \end{aligned}$$

更简洁的写法

$$\begin{aligned} \text{letter_} &\rightarrow [A-Za-z] \\ \text{digit} &\rightarrow [0-9] \\ \text{id} &\rightarrow \text{letter_}(\text{letter_}|\text{digit})^* \end{aligned}$$

例 4. 下列正则表达式描述什么语言？

- $a(a|b)^*a$: 首尾是 a 中间任意个（可为0） a 或 b 的字符串
- $(a|b)^*a(a|b)(a|b)$: 倒数第三个字符为 a 仅含 a 或 b 的字符串
- $a^*ba^*ba^*ba^*$: 只含3个 b 且 a 在中间穿插（可没有）的字符串
- $((E|a)b^*)^*$: 空、全 a 全 b 、开头一个 a 紧接多个 b 的重复串
- $b^*(ab^*ab^*)^*$: 所有包含偶数个 a 的由 a 和 b 组成的字符串

注意考虑闭包为空的情况, ϵ 出现也可能导致空串!

例 5. 用正则表达式描述下列语言：

- 所有由按词典递增序排列的小写字母组成的字符串（如 add 、 low 都符合要求，而 zzg 则不符合）²:
 $a^+b^+c^+\dots z^+ \mid a^+b^+c^+\dots z^+ \mid a^+b^+c^+\dots z^+ \mid \dots$
- 不以 ab 开头的只含有字母 a 和 b 的字符串: $(ba|aa|bb)(a|b)^*$

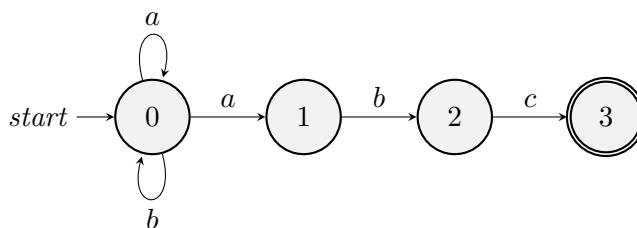
²参见<https://www.zhihu.com/question/28714623/answer/41865697>

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 6. 识别语言 $L((a|b)^*abb)$, 下面为一个 NFA



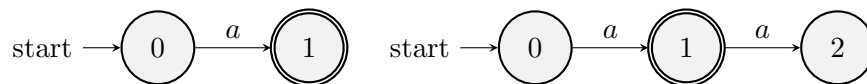
判别字符串能否被DFA识别很简单, 只需要读入字符按照状态转移表跳转, 判断末态是不是终态即可 (即模拟)。

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 $O(|str|)$ 。

对于DFA或NFA求反相当于将所有接受状态改为非接受状态, 非接受状态改为接受状态。注意可能出现DFA无对应符号出边的情况, 如a, 这时可以添加冗余结点来接受这些非法输入。



定理 1. 对任一正则表达式 R , 一定存在另一正则表达式 R' , 使得 $L(R')$ 是 $L(R)$ 的补集.

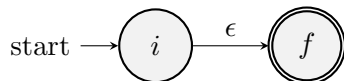
分析. 由正则表达式与DFA的等价性, 对于正则表达式 R , 必然存在DFA M 可以识别 $L(R)$, 那么将 M 中的接受状态改为非接受状态, 将非接受状态改为接收状态, 得到新的DFA M' 可以识别 $L(R)$ 的补集, 进而存在 M' 对应的正则表达式 R' , 使得 $L(R')$ 是 $L(R)$ 的补集.

2.3.2 正则表达式转NFA

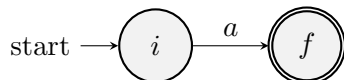
McNaughton-Yamada-Thompson算法

1. 奠基

- 对于表达式 ϵ ，构建NFA

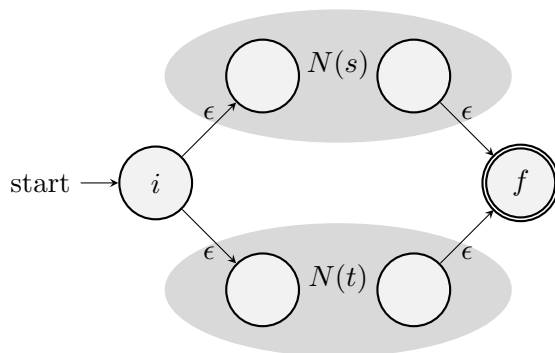


- 对于任意子表达式 $a \in \Sigma$ ，构建NFA

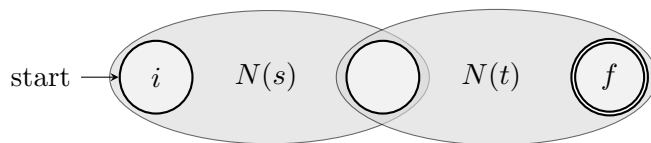


2. 推论

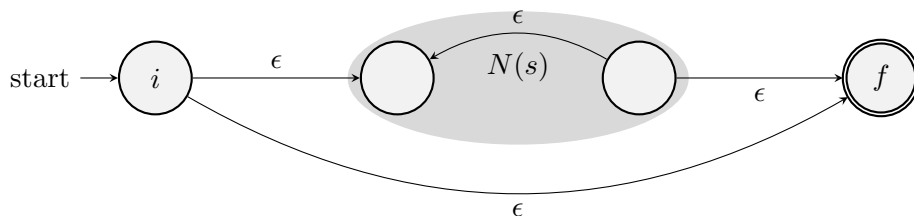
- $r = s|t$ ，取并集



- $r = st$ ，取连接



- $r = s^*$ ，Kleene闭包



其他拓展符号可通过上述基本符号得到，如

- R^+ 等价于 RR^*
- $R?$ 等价于 $\epsilon|R$

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 (包括自己). $move(T, a)$ 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)

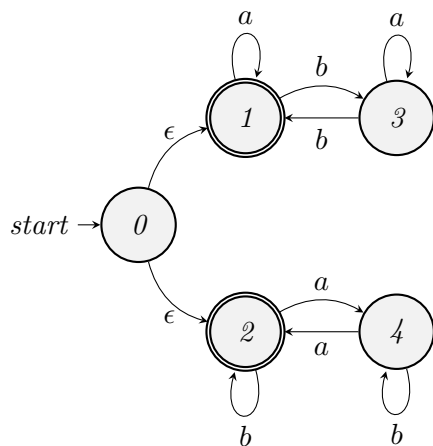
Require: NFA N

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: $Dtran[T, a] = U$
-

思路即先求出初态的 ϵ 闭包, 然后对每个输入符号做转移后再求 ϵ 闭包, 看是否产生新的子集状态。注意这里的输入符号转移一定得转, 即不能留在原状态。

例 7. 考虑以下NFA:

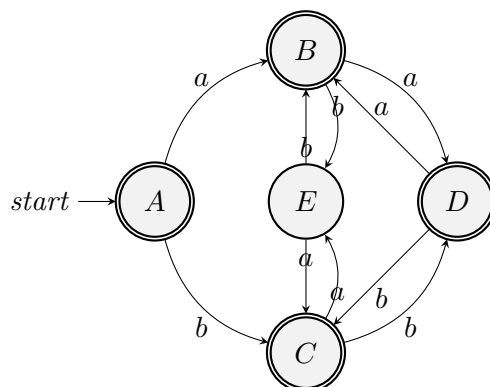


1. 这一NFA接受什么语言 (用自然语言描述)?
2. 构造接受同一语言的DFA.

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

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



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 2. *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 8. $L_n = (a \mid b)^* a (a \mid b)^{n-1}$ ，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

$bb \cdots a$

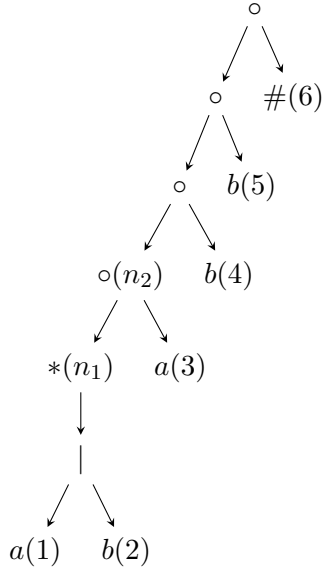
$bb \cdots b$

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

可以由正则表达式通过NFA转为DFA，本节则讲述直接由正则表达式转为DFA。

构造正则表达式的语法树，以 $\#$ 结尾。



*为star, |为or, o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 9. 结点 n_1 代表 $(a|b)^*$ ，结点 n_2 代表 $(a|b)^*a$

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

计算上述函数的方式：

结点 n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
标记为 ϵ 的叶子	true	\emptyset	\emptyset
位置为 i 的叶子	false	$\{i\}$	$\{i\}$
or结点 $n = c_1 c_2$	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$
cat结点 $n = c_1c_2$	$nullable(c_1)$ and $nullable(c_2)$	if ($nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ else $firstpos(c_1)$	if ($nullable(c_2)$) $lastpos(c_1) \cup lastpos(c_2)$ else $lastpos(c_2)$
star结点 $n = c_1^*$	true	$firstpos(c_1)$	$followpos(c_1)$

- 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: $Dtran[S, a] = U$
-

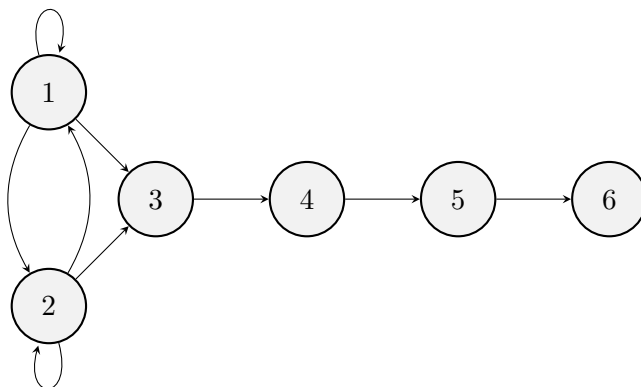
例 10. 考虑正则表达式 $(a \mid b)^* a b b \#$

$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix}$

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

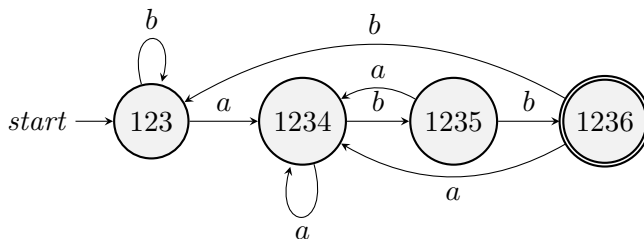
进而构造出一个有向图 (加上标号可变成 NFA)



由正则表达式转 DFA 的算法可得以下状态转移表

	a	b
{1, 2, 3}	{1, 2, 3, 4}	{1, 2, 3}
{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 5}
{1, 2, 3, 5}	{1, 2, 3, 4}	{1, 2, 3, 6}
{1, 2, 3, 6}	{1, 2, 3, 4}	{1, 2, 3}

然后可得DFA



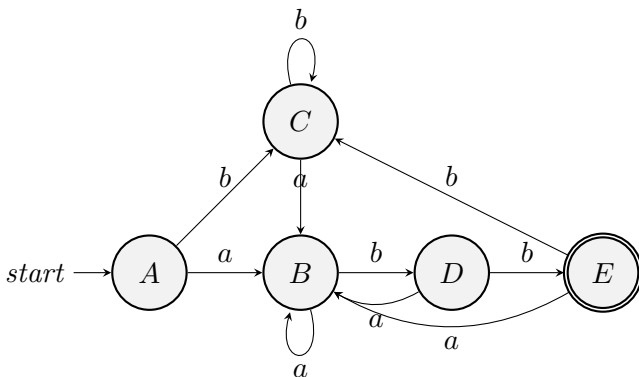
2.5 最小化DFA

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 3. 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 11. 如下状态转移图



分析. 初始划分 Π 包括两个组(group): 接受状态组(E)和非接受状态组($ABCD$)³。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。

³准确来说是接受状态的补, 如果接受状态组为全集, 那么另一组将为空。另外需要考虑转移到外部结点的情况, 即DFA中无对应跳转符号。

	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。

很多语言并非正则的，比如匹配的括号串 $\{(i)^i \mid i \geq 0\}$ ，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 (即*token*名字)
- 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 12. 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(left-most): 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(right-most): 每步推导都替换最右侧的非终端符号

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

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 13. 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:

$$\begin{array}{ll}
 E \Rightarrow E + E & E \Rightarrow E * E \\
 \Rightarrow id + E & \Rightarrow E + E * E \\
 \Rightarrow id + E * E & \Rightarrow id + E * E \\
 \Rightarrow id + id * E & \Rightarrow id + id * E \\
 \Rightarrow id + id * id & \Rightarrow id + id * id
 \end{array}$$

文法二义性可通过引入更多的产生式来消除。

例 14. $E \rightarrow E + E \mid E * E \mid (E) \mid id$ 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

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

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 , 强行添加了括号/优先级。

例 15. 悬挂的 *if-else*: `if E1 then if E2 then E3 else E4`, 可以令 `else` 匹配最近的 `then`。

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

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

定理 4. 正则表达式/NFA/DFA与左/右线性文法得表达能力是等价的

在处理程序时, 上下文无法文法存在局限性, 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, 不断回溯遍历, 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。

例 16. 消除左递归的方法: 先把单元素拎出来放左侧, 然后把所有递归移至右侧

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

如果是多级左递归, 则需要先将上级生成式代入到中间级生成式中, 再做消除

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

将下式改写为 $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$, 进而可消除左递归

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bdA' \mid A' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

定义 14 (提取左因子(left-factoring)). 将生成式右侧左部相同的因子部分提取出来, 找最长前缀

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \alpha\beta_n \mid \gamma$$

γ 代表所有不以 α 开始的生成式，提取左因子则得到（将 α 拿出来）

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \beta_n \end{aligned}$$

直至没有生成式有相同前缀

Algorithm 5 递归下降(top-down parser)

```

1: 选择 $A$ 生成式 $A \rightarrow X_1 X_2 \cdots X_k$ 
2: for  $i = 1$  to  $k$  do
3:   if  $X_i$ 是非终端符号 then
4:     调用 $X_i()$ 
5:   else
6:     if  $X_i$ 等于当前的输入符号 $a$  then
7:       读取下一输入符号
8:     else
9:        $error()$ 

```

定义 15 (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：

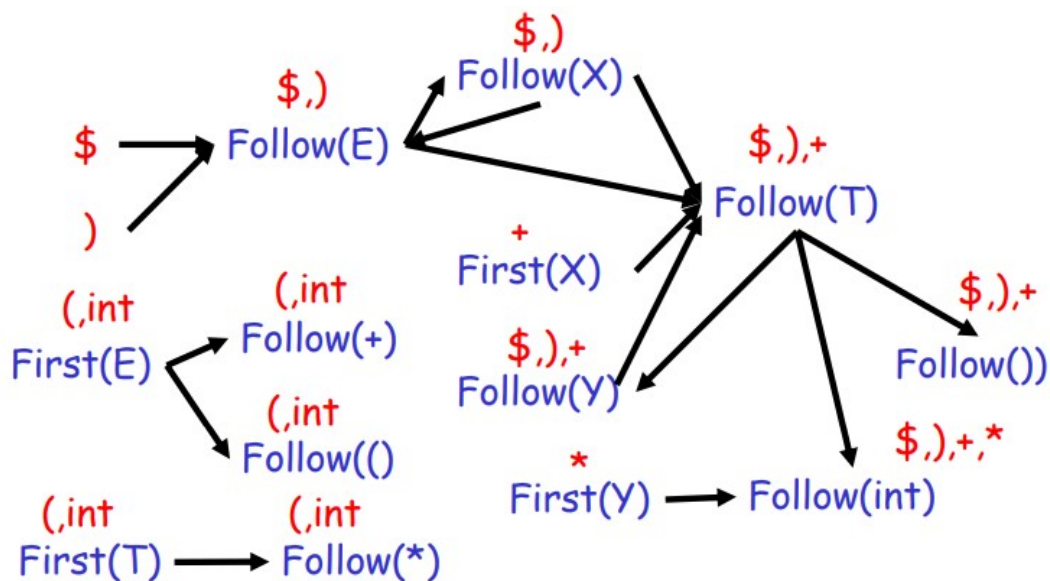
1. $\$ \in FOLLOW(S)$

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

$$\begin{array}{ll} E \rightarrow T X & X \rightarrow + E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow * T \mid \epsilon \end{array}$$

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



29

算法 3 (构造预测语法表). 对于每一生成式 $A \rightarrow \alpha$,

1. 对于每一终端符号 $a \in FIRST(\alpha)$, 将 $A \rightarrow \alpha$ 添加到 $M[A, a]$ 中。
2. 若 $\epsilon \in FIRST(\alpha)$, 则对 $b \in FOLLOW(A)$, 将 $A \rightarrow \alpha$ 添加到 $M[A, b]$ 中 (β 可以为 $\$$)。

注意是生成式右端。

例 17. 考虑下面语法

- (1) $E \rightarrow TE'$
- (2) $E' \rightarrow +TE' \mid \epsilon$
- (3) $T \rightarrow FT'$
- (4) $T' \rightarrow *FT' \mid \epsilon$
- (5) $F \rightarrow (E) \mid id$

计算FIRST集

- 从终端符号多的开始(5), $FIRST(F) = \{ (, id \}$
- 向上找含F的生成式(3), $FIRST(T) = FIRST(F)$
- 向上找含T的生成式(1), $FIRST(E) = FIRST(T)$
- $FIRST(E') = \{ +, \epsilon \}$
- $FIRST(T') = \{ *, \epsilon \}$

计算FOLLOW集

- 从起始符号开始(1), 注意(5)也有E, 故 $FOLLOW(E) = \{ \}, \$ \}$
- E' 只出现在E的生成式末尾, 因此 $FOLLOW(E') = FOLLOW(E) = \{ \}, \$ \}$
- $FOLLOW(T) \subset FIRST(E') = \{ +, \epsilon \}$, 由于 $\epsilon \in FIRST(E')$, 故 $FOLLOW(T) \subset FOLLOW(E) = \{ \}, \$ \}$, 即 $FOLLOW(T) = \{ +,), \$ \}$
- $FOLLOW(F) \subset FIRST(T') = \{ *, \epsilon \}$, 由于 $\epsilon \in FIRST(T')$, 故 $FOLLOW(F) \subset FOLLOW(T) = \{ +,), \$ \}$, 即 $FOLLOW(F) = \{ *, +,), \$ \}$
- T' 只出现在T的生成式末尾, 因此 $FOLLOW(T') = FOLLOW(T) = \{ *, +,), \$ \}$

对应有语法预测表

	id	$+$	$*$	$($	$)$	$\$$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

3.4 自顶向下分析

定义 16 (LL(1)文法⁴). 语法G是LL(1)文法当且仅当对于任意 $A \rightarrow \alpha \mid \beta$ 为G两个不同的生成式, 满足

1. α 和 β 不会同时推导出由同一终端符号a开始的字符串
2. α 和 β 中至多一个能获得空字符串
3. 若 $\beta \rightarrow \epsilon$, 则 α 不能推出任何以 $FOLLOW(A)$ 中终端符号开始的字符串; 同样地, 若 $\alpha \rightarrow \epsilon$, 则 β 不能推出任何以 $FOLLOW(A)$ 中终端符号开始的字符串

⁴第一个L代表输入字符串从左边开始扫描, 第二个L代表得到的推导是最左推导, (1)代表向前看1个输入符号(或单词)

前两个条件等价于 $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$ ，第三个条件等价于若 $\epsilon \in FIRST(\beta)$ ，则 $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$ ，反之同理

通常没有左递归、无歧义的文法可以是LL(1)。

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 18. 考虑以下文法

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

共同前缀分解后即得

$$\begin{aligned} E &\rightarrow TX \\ X &\rightarrow +E \mid \epsilon \\ T &\rightarrow int Y \mid (E) \\ Y &\rightarrow *T \mid \epsilon \end{aligned}$$

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

例 19. 经典的二义if-else语法（已提取左因子）

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

可以求得

$$\begin{aligned} FIRST(S) &= \{i, a\} \\ FIRST(S') &= \{\epsilon, e\} \\ FIRST(E) &= b \\ FOLLOW(S) &= \{\$ \} + FIRST(S') - \{\epsilon\} = \{\$, e\} \\ FOLLOW(S') &= \{\$ \} + FOLLOW(S) = \{\$, e\} \\ FOLLOW(E) &= \{\$, t\} \end{aligned}$$

因为 $\epsilon \in FIRST(S' \rightarrow \epsilon)$ 里，而 $FIRST(S' \rightarrow eS) \cap FOLLOW(S') = \{e\} \neq \emptyset$ ，所以不是LL(1)文法。

Algorithm 6 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]=  $X \rightarrow Y_1Y_2 \cdots Y_k$ 
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()
```

基于表的预测语法分析，用栈实现。

例 20. 考虑以下文法：

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow L, S \mid S \end{aligned}$$

1. 消除文法的左递归.
2. 构造文法的 $LL(1)$ 分析表.
3. 对于句子 $(a, (a, a))$ ，给出语法分析的详细过程（参照课本228页的图4.21）.

分析. 1. 如下

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow SL' \\ L' &\rightarrow, SL' \mid \epsilon \end{aligned}$$

2. 先求出 $FIRST$ 集和 $FOLLOW$ 集（由于文法中存在逗号，故将字符用引号括起来以示区分）

$$\begin{aligned} FIRST(S) &= \{(' , a')\} & FOLLOW(S) &= \{('$, ')'\} \\ FIRST(L) &= \{(' , a')\} & FOLLOW(L) &= \{')'\} \\ FIRST(L') &= \{(' , \epsilon)\} & FOLLOW(L') &= \{')'\} \end{aligned}$$

$LL(1)$ 分析表如下，其中第一列为非终端符号，第一行为输入符号.

	()	a	,	\$
S	$S \rightarrow (L)$		$S \rightarrow a$		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L'		$L' \rightarrow \epsilon$		$L' \rightarrow, SL'$	

3. 语法分析过程如下

<i>Matched</i>	<i>Stack</i>	<i>Input</i>	<i>Action</i>
	$S\$$	$(a, (a, a))\$$	
	$(L)\$$	$(a, (a, a))\$$	<i>output</i> $S \rightarrow (L)$
($L)\$$	$a, (a, a))\$$	
($SL')\$$	$a, (a, a))\$$	<i>output</i> $L \rightarrow SL'$
($aL')\$$	$a, (a, a))\$$	<i>output</i> $S \rightarrow a$
(a	$L')\$$	$, (a, a))\$$	
(a	$, SL')\$$	$, (a, a))\$$	<i>output</i> $L' \rightarrow, SL'$
(a,	$SL')\$$	$(a, a))\$$	
(a,	$(L)L')\$$	$(a, a))\$$	<i>output</i> $S \rightarrow (L)$
(a, ($L)L')\$$	$a, a))\$$	
(a, ($SL')L')\$$	$a, a))\$$	<i>output</i> $L \rightarrow SL'$
(a, ($aL')L')\$$	$a, a))\$$	<i>output</i> $S \rightarrow a$
(a, (a	$L')L')\$$	$, a))\$$	
(a, (a	$, SL')L')\$$	$, a))\$$	<i>output</i> $L' \rightarrow, SL'$
(a, (a,	$SL')L')\$$	$a))\$$	
(a, (a,	$aL')L')\$$	$a))\$$	<i>output</i> $S \rightarrow a$
(a, (a, a	$L')L')\$$	$)\$$	
(a, (a, a	$)L')\$$	$)\$$	<i>output</i> $L' \rightarrow \epsilon$
(a, (a, a)	$L')\$$	$)\$$	
(a, (a, a)	$)\$$	$)\$$	<i>output</i> $L' \rightarrow \epsilon$
(a, (a, a))	$\$$	$\$$	

3.5 自底向上分析

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

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

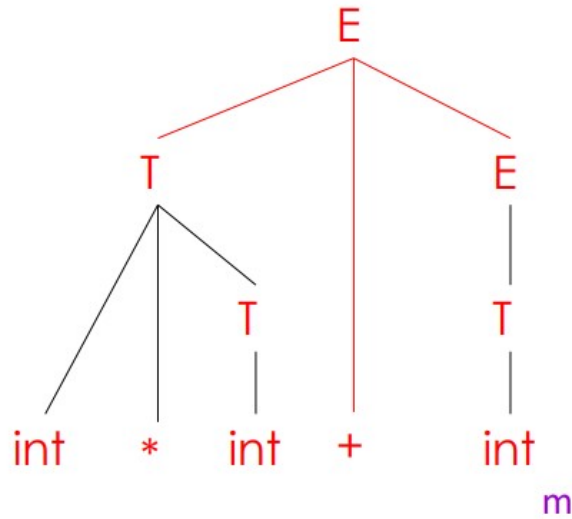
- 规约(reduce)：在字符串右侧逆向应用生成式

$$Cbxy \mid ijk \Longrightarrow CbA \mid ijk$$

```

| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |

```



63

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 17 (句柄(handle)). *A handle is a string that can be **reduced** and also allows further reductions back to the start symbol.* 可以理解为当前正在处理的token，用于**规约**而不是移进。

定义 18 (活前缀(viable prefix)). α 是活前缀若存在 ω 使得 $\alpha | \omega$ 是移进-规约语法分析器的状态。

LR分析是最通用的**非回溯**移进-规约语法解析方法，难点在于构建分析表太过麻烦，但Yacc等工具可辅助构建。

3.5.1 LR(0)语法

为构建规范(canoical)LR(0)项，定义增量语法 G' 有生成式 $S' \rightarrow S$ ，其中 S 为原语法 G 的开始符号。这个生成式用于告知parser接受(accept)输入并停止解析，即接受仅发生在要对 $S' \rightarrow S$ 进行规约的时候。

定义 19 (CLOSURE). 若 I 为语法 G 项的集合，则 $CLOSURE(I)$ 为从 I 中构造出项的集合：

1. 初始时，在 I 中的每一项都会被加到 $CLOSURE(I)$ 中
2. 若 $A \rightarrow \alpha \cdot B\beta$ 在 $CLOSURE(I)$ 中且 $B \rightarrow \beta$ 是一个生成式，则将项 $B \rightarrow \cdot\gamma$ 加入到 $CLOSURE(I)$ 中；重复使用此规则，直至没有新项可以被加入

定义 20 (GOTO). I 是项的集合， X 为输入符号， $GOTO(I, X)$ 为所有项 $[A \rightarrow \alpha X \cdot \beta]$ 的闭包使得 $[A \rightarrow \alpha \cdot X\beta]$ 在 I 中

定义 21 (核项(kernel item)). 初始项 $S' \rightarrow \cdot S$ 及所有 \cdot 不在左端的项称为核项，除了初始项外所有 \cdot 在左端的项称为非核项（即那些新加入闭包的项）

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（没有得移进就规约，自动机无对应符号出边）
- 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 $Y \rightarrow \omega.t\delta$

3.5.2 SLR分析

SLR(simple left-to-right scan)⁵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 上停在状态 s
- 当 s 包含 $X \rightarrow \beta$ 的项且 $t \in FOLLOW(X)$ 时进行 $X \rightarrow \beta$ 规约
- 当 s 包含 $X \rightarrow \beta.t\omega$ 的项时移进

算法 4 (SLR(1)分析表). 构造 $C = \{I_0, I_1, \dots, I_n\}$ 为LR(0)项的集合 G'

1. 若 $[A \rightarrow \alpha \cdot a\beta] \in I_i$ 且 $GOTO(I_i, a) = I_j$ ，则设 $ACTION[i, a]$ 为移进 j ，
2. 若 $[A \rightarrow \alpha \cdot] \in I_i$ ，则 $\forall a \in FOLLOW(A)$ ，设 $ACTION[i, a]$ 为规约 $A \rightarrow \alpha$
3. 若 $[S' \rightarrow S \cdot] \in I_i$ ，则设 $ACTION[i, \$]$ 为接受(ACC)

若上述有冲突的动作，则该文法不是SLR(1)的。

依照分析表可以得到语法分析的算法

- 若 $ACTION[s, a]$ 为移进 t ，则将 t 推入栈中
- 若 $ACTION[s, a]$ 为规约 $A \rightarrow \beta$ ，则将 $|\beta|$ 个符号从栈顶弹出，令 t 为栈顶符号，将 $GOTO[t, A]$ 推入栈中，输出规约 $A \rightarrow \beta$
- 若 $ACTION[s, a] = acc$ ，则语法解析结束

例 21. 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) $F \rightarrow b$

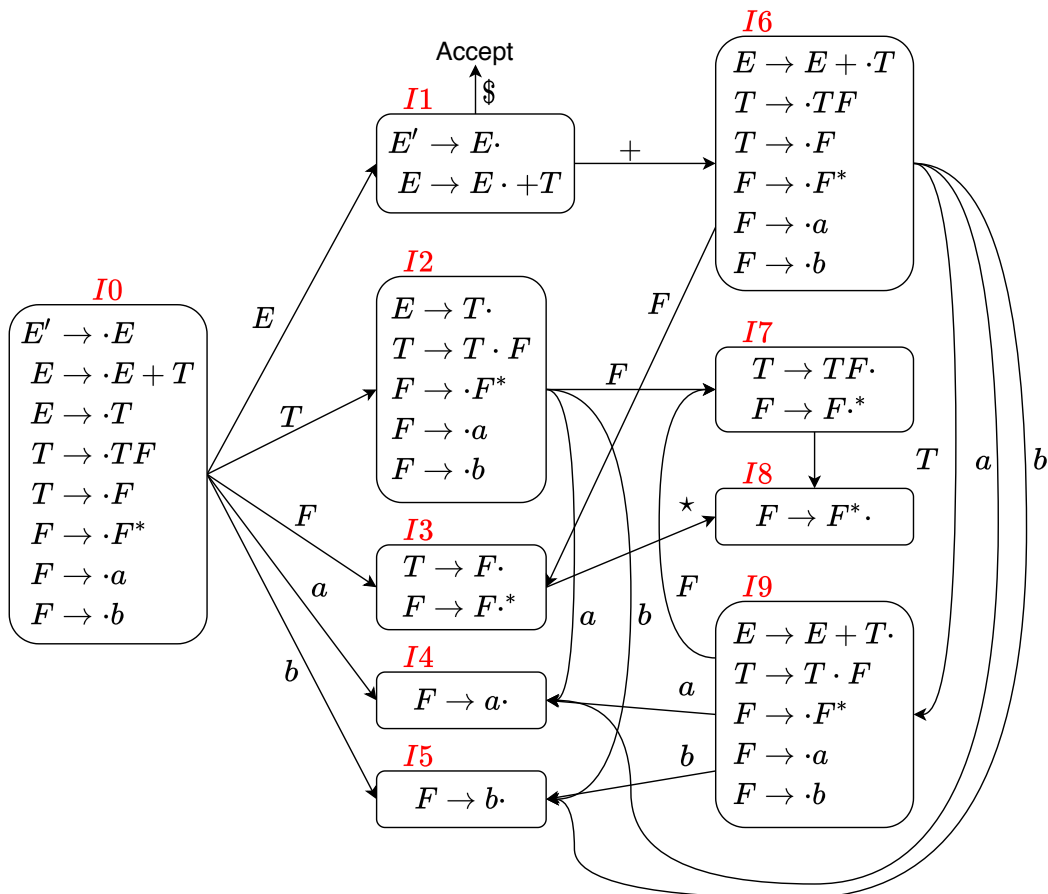
1. 写出每个非终端符号的FIRST集和FOLLOW集.
2. 构造识别这一文法所有活前缀(viable prefixes)的LR(0)自动机（参照课本4.6.2节图4.31）.
3. 构造这一文法的SLR分析表（参照课本4.6.3节图4.37）.
4. 给出SLR分析器识别输入串 $a + ab^*$ 的过程（参照课本4.6.4节图4.38）

⁵或SLR(1)分析，通常省略(1)

分析. 1. *FIRST*集和*FOLLOW*集如下

$$\begin{aligned} FIRST(E) &= \{a, b\} & FOLLOW(E) &= \{\$, +\} \\ FIRST(T) &= \{a, b\} & FOLLOW(T) &= \{\$, +, a, b\} \\ FIRST(F) &= \{a, b\} & FOLLOW(F) &= \{\$, +, *, a, b\} \end{aligned}$$

2. 构造增广语法 $E' \rightarrow E$ ，并得到 $LR(0)$ 自动机如下



3. 依据上述两问结果，可构造*SLR*分析表如下（ s 后面的数字为*DFA*状态编号， r 后面的数字为生成式的编号）

STATE	ACTION					GOTO		
	a	b	+	*	\$	E	T	F
0	s4	s5				1	2	3
1			s6		ACC			
2	s4	s5	r2		r2			7
3	r4	r4	r4	s8	r4			
4	r6	r6	r6	r6	r6			
5	r7	r7	r7	r7	r7			
6	s4	s5					9	3
7	r3	r3	r3	s8	r3			
8	r5	r5	r5	r5	r5			
9	s4	s5	r1		r1			7

4. 依上述ACTION-GOTO表，可得以下过程

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$a + ab^* \$$	$[0, a]s4$
(2)	04	a	$+ab^* \$$	$[4, a]r6F \rightarrow a, [0, F]s3$
(3)	03	F	$+ab^* \$$	$[3, +]r4T \rightarrow F, [0, T]s2$
(4)	02	T	$+ab^* \$$	$[2, +]r2E \rightarrow T, [0, E]s1$
(5)	01	E	$+ab^* \$$	$[1, +]s6$
(6)	016	E+	$ab^* \$$	$[6, a]s4$
(7)	0164	E + a	$b^* \$$	$[4, b]r6F \rightarrow a, [6, F]s3$
(8)	0163	E + F	$b^* \$$	$[3, b]r4T \rightarrow F, [6, T]s9$
(9)	0169	E + T	$b^* \$$	$[9, b]s5$
(10)	01695	E + Tb	$* \$$	$[5, *]r7F \rightarrow b, [9, F]s7$
(11)	01697	E + TF	$* \$$	$[7, *]s8$
(12)	016978	E + TF*	$\$$	$[8, \$]r5F \rightarrow F^*, [9, F]s7$
(13)	01697	E + TF	$\$$	$[7, \$]r3T \rightarrow TF, [6, T]s9$
(14)	0169	E + T	$\$$	$[9, \$]r1E \rightarrow E + T, [0, E]s1$
(15)	01	E	$\$$	$[1, \$]ACC$

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 22. 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)

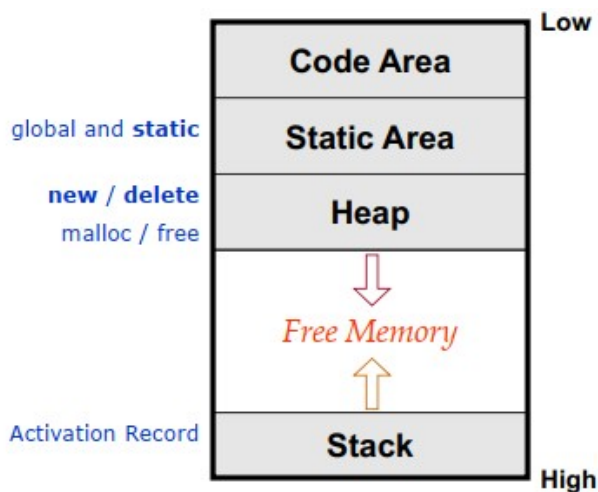
```
x = y op z // arithmetic and logical
x = op y // negation and conversion
x = y // copy
goto L // unconditional jump
if x goto L // conditional jump
if False x goto L // conditional jump
if x op y goto L // relational operation
param x1 // parameter passing
param x2
...
param xn
call p, n // procedure 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, ll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

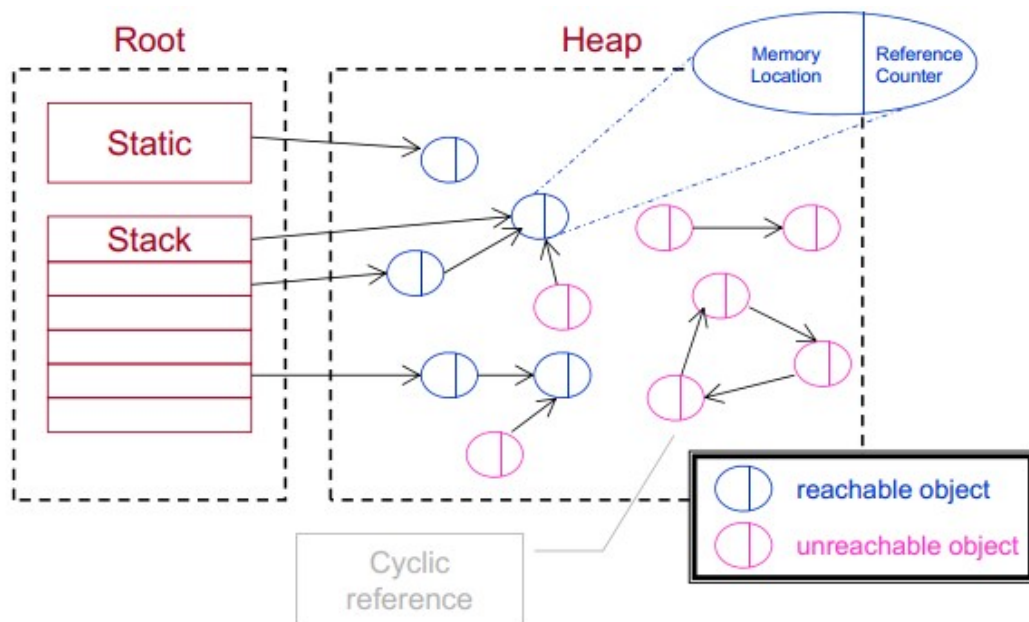
5.1 存储管理



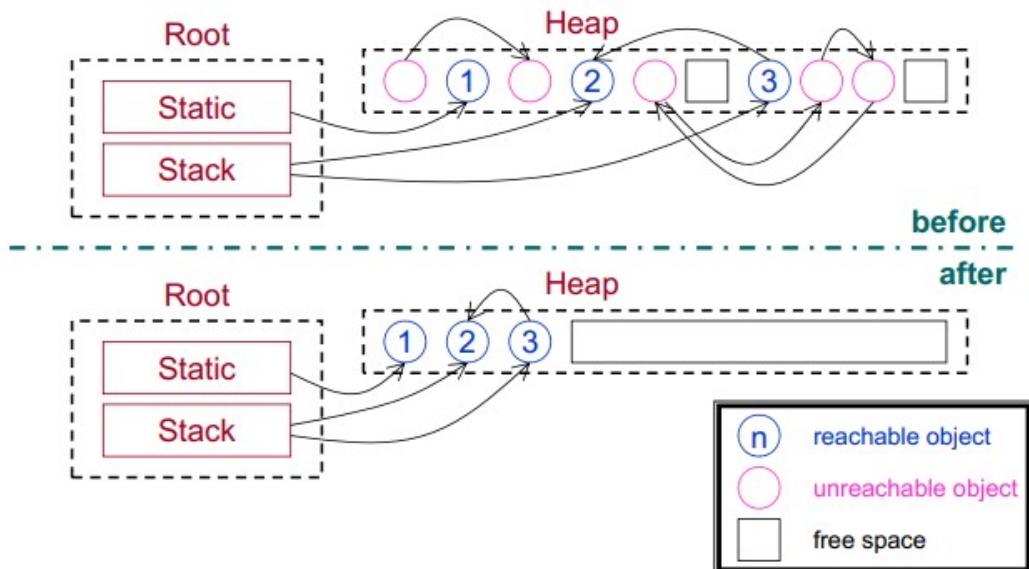
5.2 垃圾回收

- 引用计数(reference counting): 创建加1, 删除减1
 - 简单、立即增量式回收

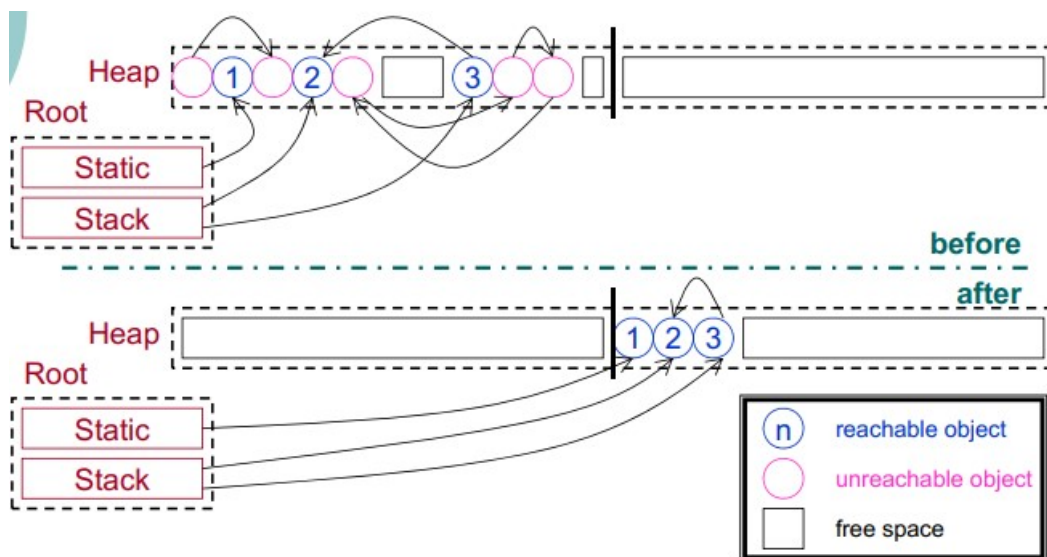
- 不能回收循环引用的示例



- 标记扫除(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。

6 代码生成及优化

6.1 代码生成

- 指令选择：选择最适合目标机器的指令来实现IR
- 寄存器分配和指派
- 指令调度

定义 22 (基本块(basic block)). 单一入口单一出口。成为 $leader$ 的指令：

1. 第一条三地址指令
2. 条件或无条件跳转指令的目标
3. 条件或无条件跳转指令的下一指令

6.2 代码优化

- 窥孔优化(peephole): 基于滑动窗口，最小粒度

```
x = x + 0 // eliminated
x = x * 1 // eliminated
y = x * 2 // y = x << 1
LD R0, a
ST a, R0 // eliminated
```

- 局部优化：在基本块内的优化
 - 公共子表达式删除

- 常量/拷贝传递
- 荣誉操作消除
- 循环优化：在循环内的优化
- 全局优化：最粗粒度的优化

定义 23 (循环(loop)). 只有唯一入口/头的强连通子图