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 正则表达式、DFA与NFA的转换 5

2.5 最小化DFA 11

3 语法分析 **12**

3.1 上下文无关文法 12

3.2 基础概念与处理 14

3.3 自顶向下分析 16

3.4 自底向上分析 21

3.5 语法制导翻译 29

3.6 总结 31

4 语义分析与中间表示 **31**

5 运行时系统 **34**

5.1 存储管理 34

5.2 垃圾回收 34

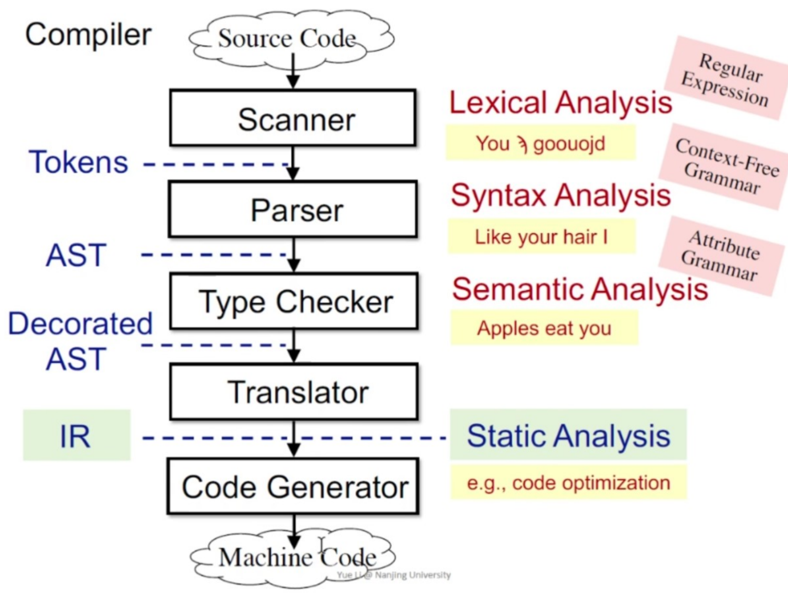
*Build 20200728

6 代码生成及优化 **35**
6.1 代码生成 35
6.2 代码优化 37

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$L(\mathbf{a}) = \{a\}$（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。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：$r(s|t)=rs|rt$，$(s|t)r=sr|tr$
- 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)**.** $d_i \to r_i$，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

$$\begin{aligned} letter\_ &\to A|B|\cdots|Z|a|b|\cdots|z|\_ \\ digit &\to 0|1|\cdots|9 \\ id &\to letter\_(letter\_|digit)^* \end{aligned}$$

更简洁的写法

$$\begin{aligned} letter\_ &\to [A-Za-z\_] \\ digit &\to [0-9] \\ id &\to letter\_(letter|digit)^* \end{aligned}$$

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：

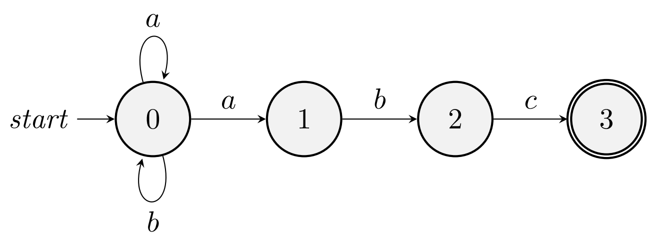
- 所有由按词典递增序排列的小写字母组成的字符串（如add、low都符合要求，而zzg则不符合）[2]：`a+b*c*...z* | a*b+c*...z* | a*b*c+...z* | ...`
- 不以ab开头的所有只含有字母a和b的字符串：`(ba|aa|bb)(a|b)*`

[2]参见https://www.zhihu.com/question/28714623/answer/41865697

2.3 有限自动机

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*



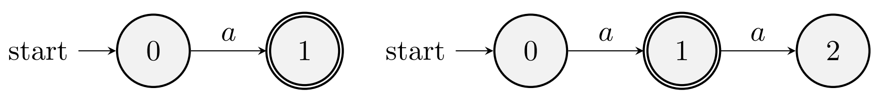
判别字符串能否被DFA识别很简单，只需要读入字符按照状态转移表跳转，判断末态是不是终态即可（即模拟）。

Algorithm 1 基于DFA的识别算法

$s = s_0$
$c = nextChar()$
while $(c != \textbf{eof})$ **do**
    $s = move(s, c)$
    $c = nextChar()$
if $s \in F$ **then**
    return “yes”
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.*

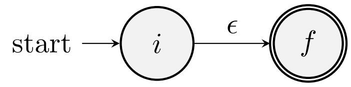
2.4 正则表达式、DFA与NFA的转换

2.4.1 正则表达式转NFA

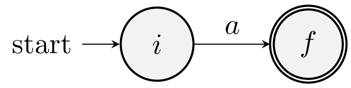
McNaughton-Yamada-Thompson算法

1. 奠基

- 对于表达式ϵ，构建NFA

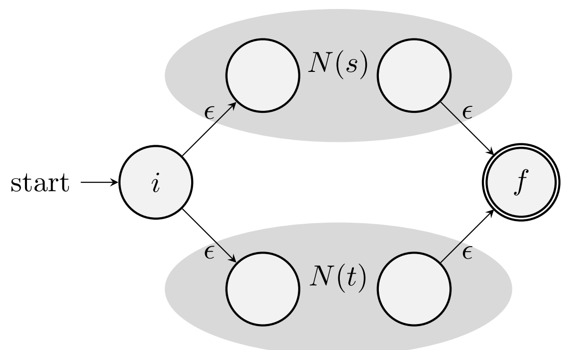


- 对于任意子表达式$a \in \Sigma$，构建NFA

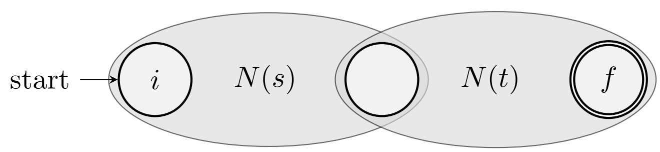


2. 推论

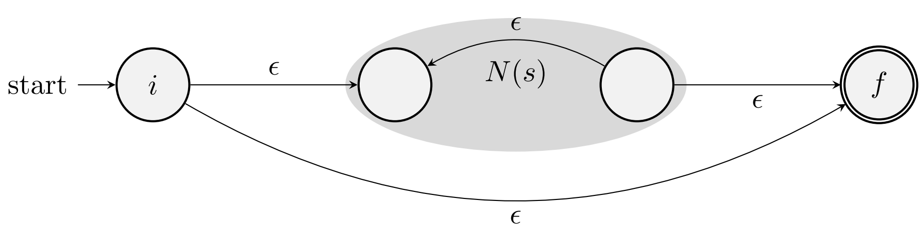
- $r = s|t$，取并集



- $r = st$，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$\epsilon|R$

2.4.2 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）

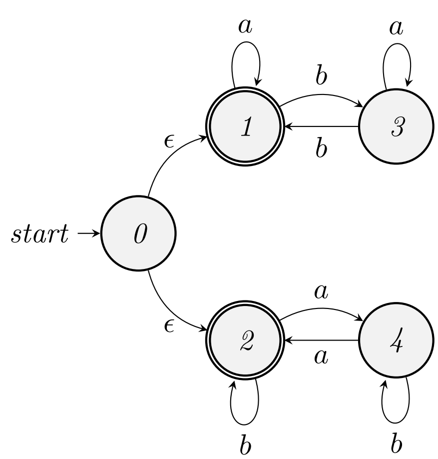
Require: NFA N

Ensure: DFA D（与N接受相同的语言）

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

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:*

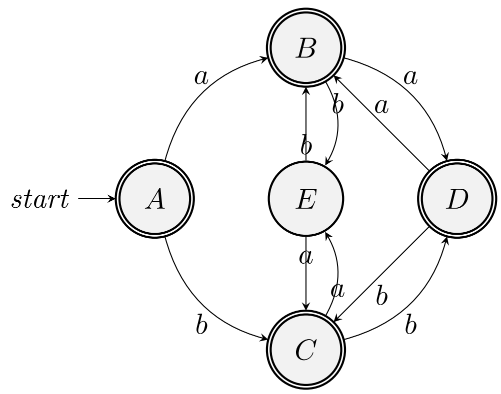


1. *这一NFA接受什么语言（用自然语言描述）?*
2. *构造接受同一语言的DFA.*

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

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

$S = \epsilon\text{-}closure(s_0)$
$c = nextChar()$
while c!=**eof** **do**
  $S = \epsilon\text{-}closure(move(S, c))$
  $c = nextChar()$
if $S \cap F != \varnothing$ **then**
  return “yes”
else
  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

$aa \cdots a$

$aa \cdots 1$

$\cdots$

$bb \cdots a$

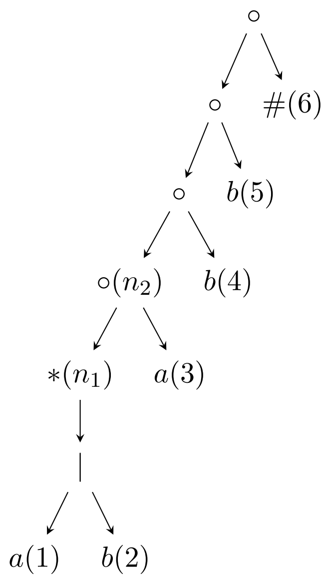
$bb \cdots b$

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。

2.4.3 正则表达式转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，$\circ$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 9. 结点n_1代表$(a|b)^*$，结点n_2代表$(a|b)^*a$

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

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**	$\varnothing$	$\varnothing$
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$L(r)$

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)

while 在$Dstates$中还有未被标记的状态S **do**

  标记S

  for 每一个输入符号a **do**

    $U = \bigcup_{\text{对应}a\text{的位置}p\in S} followpos(p)$

    if $U \notin Dstates$ **then**

      将U作为未标记的状态加入$Dstates$

    $Dtran[S,a] = U$

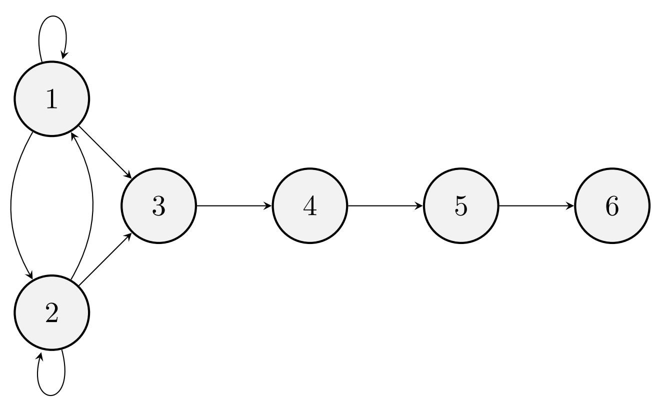
例 10. 考虑正则表达式

(	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

$position$	$followpos(i)$
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\varnothing$

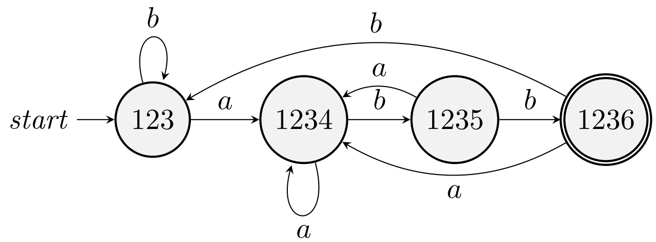
进而构造出一个有向图（加上标号可变成NFA）



$firstpos(n_0)$为$\{1,2,3\}$，由正则表达式转DFA的算法可得以下状态转移表。比如对于$S=\{1,2,3\}$，a的位置是1和3，那么取$followpos$的并为$\{1,2,3\}\cup\{4\}$。

	a	b
$\{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3\}$
$\{1,2,3,4\}$	$\{1,2,3,4\}$	$\{1,2,3,5\}$
$\{1,2,3,5\}$	$\{1,2,3,4\}$	$\{1,2,3,6\}$
$\{1,2,3,\underline{6}\}$	$\{1,2,3,4\}$	$\{1,2,3\}$

然后可得DFA



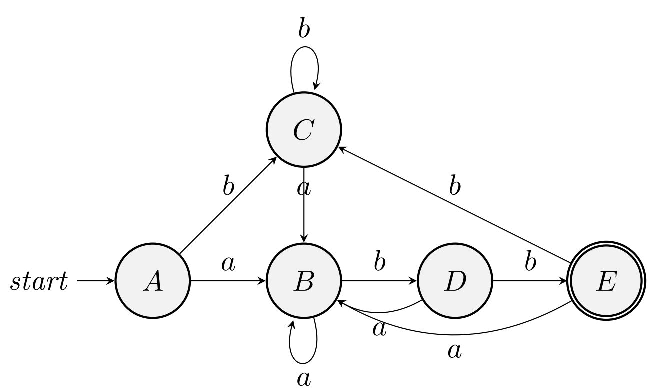
2.5 最小化DFA

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

[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。

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 上下文无关文法

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。

3.1.1 基本定义

定义 8 (上下文无关文法(context-free gramma, 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)*

$$X \to Y_1Y_2\dots Y_n$$

*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：*

- $L = \{0^n1^n \mid n \geq 1\}$：$E \to 0E1 \mid 01$
- *只含有0和1的回文串：*$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$
- *只含有(和)的匹配括号串：*$E \to (E) \mid EE \mid \epsilon$

- 最左推导(left-most)：每步推导都替换最左侧的非终端符号

$$E \overset{lm}{\Longrightarrow} -E \overset{lm}{\Longrightarrow} -(E) \overset{lm}{\Longrightarrow} -(E+E) \overset{lm}{\Longrightarrow} -(id+E) \overset{lm}{\Longrightarrow} -(id+id)$$

- 最右推导(right-most)：每步推导都替换最右侧的非终端符号

$$E \overset{rm}{\Longrightarrow} -E \overset{rm}{\Longrightarrow} -(E) \overset{rm}{\Longrightarrow} -(E+E) \overset{rm}{\Longrightarrow} -(E+id) \overset{rm}{\Longrightarrow} -(id+id)$$

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$E \to E+E \mid E*E \mid -E \mid (E) \mid id$及句子$id+id*id$，有以下两种推导：

$$\begin{aligned} E &\Longrightarrow E+E \\ &\Longrightarrow id+E \\ &\Longrightarrow id+E*E \\ &\Longrightarrow id+id*E \\ &\Longrightarrow id+id*id \end{aligned} \qquad \begin{aligned} E &\Longrightarrow E*E \\ &\Longrightarrow E+E*E \\ &\Longrightarrow id+E*E \\ &\Longrightarrow id+id*E \\ &\Longrightarrow id+id*id \end{aligned}$$

文法二义性可通过引入更多的产生式来消除。

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

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

其中E为$Expression$，T为$Term$，F为$Factor$，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`。

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

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。

3.1.2 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$A_i \to aA_j$；若状态i在输入ϵ上转换到状态j，则添加生成时$A_i \to A_j$
3. 若i是接受状态，则添加$A_i \to \epsilon$
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*

$$A \to aB \qquad A \to a \qquad A \to \epsilon$$

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*

$$A \to Ba \qquad A \to a \qquad A \to \epsilon$$

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。

3.2 基础概念与处理

对于CFG的预处理包括**消除左递归**和**提取左因子**。

3.2.1 消除左递归

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

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

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \beta_n \implies \begin{array}{l} A \to \beta_1 A' \mid \cdots \mid \beta_n A' \\ A' \to \alpha_1 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

例 16. *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。如下例*

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid \epsilon$$

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

$$S \to Aa \mid b$$
$$A \to bdA' \mid A'$$
$$A' \to cA' \mid adA' \mid \epsilon$$

3.2.2 提取左因子

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

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

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）

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \beta_n$$

直至没有生成式有相同前缀

3.2.3 FIRST集与FOLLOW集

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$\alpha \to \epsilon$，则$\epsilon \in FIRST(\alpha)$；若$A \to c\gamma$，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$X \to Y_1Y_2 \cdots Y_k$。
   - 若$Y_1 \cdots Y_{i-1} \to \epsilon$，则将$a \in FIRST(Y_i)$放入$FIRST(X)$。
   - 若$\epsilon \in FIRST(Y_j), j = 1, 2, \ldots, k$，则将$\epsilon$放入$FIRST(X)$中。
3. 若$X \to \epsilon$是生成式，将ϵ放入$FIRST(X)$中

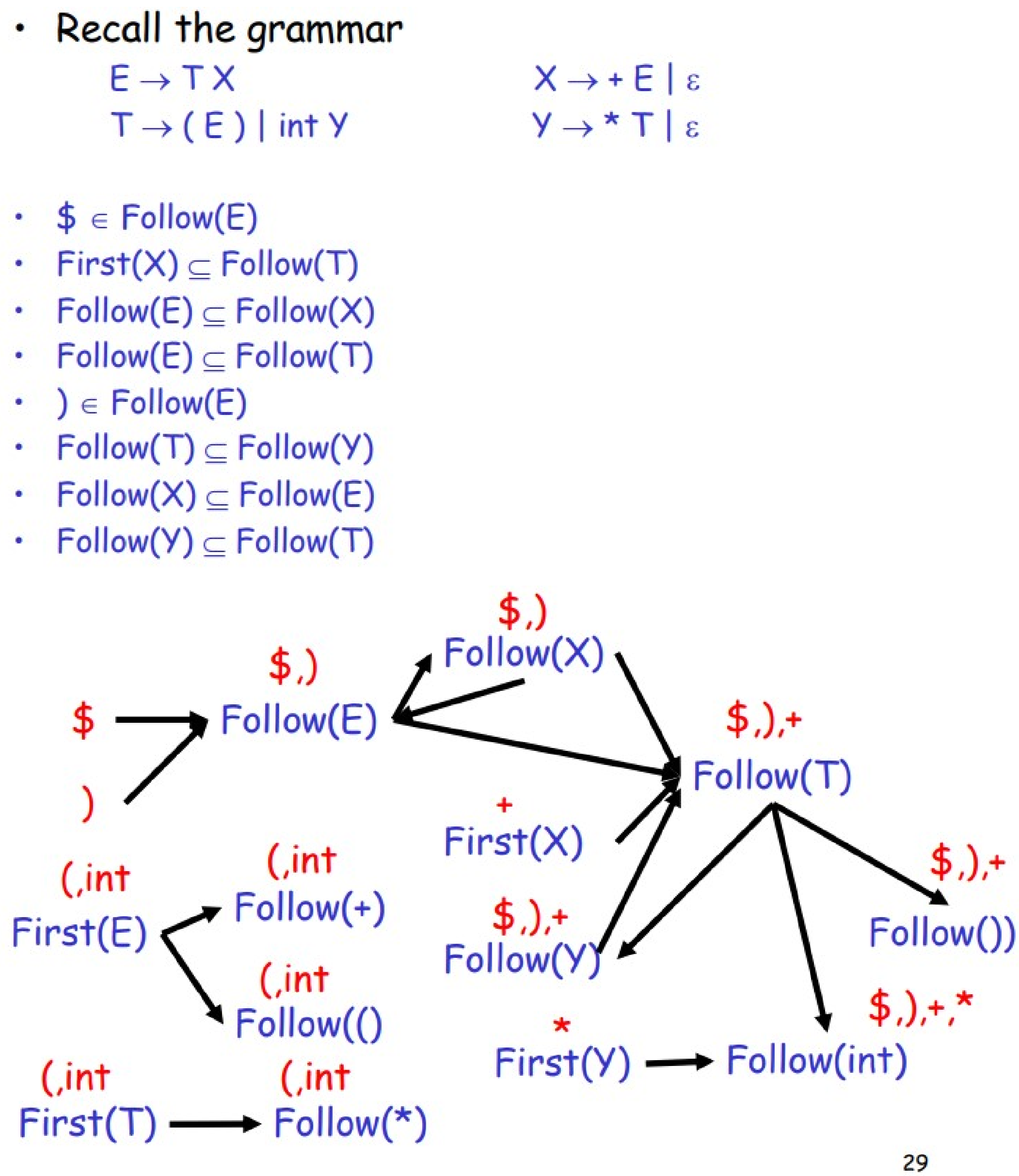
算法 2. 计算$FOLLOW(A)$集

1. 将\$放入$FOLLOW(S)$，其中S是开始符号
2. 如果有生成式$A \to \alpha B\beta$，那么$\forall a \in FIRST(\beta), a \neq \epsilon$： $a \in FOLLOW(B)$
3. 如果有生成式$A \to \alpha B$，或**生成式**$A \to \alpha B\beta$**且**$\epsilon \in FIRST(\beta)$，则$\forall a \in FOLLOW(A)$ ： $a \in FOLLOW(B)$（注意这里的因果关系）

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**。通常下面的生成式的$FOLLOW$集都会包含上面生成式的$FOLLOW$集。

另一种方式：

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



3.3 自顶向下分析

3.3.1 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。

Algorithm 5 递归下降(top-down parser)

1: 选择A生成式$A \to X_1X_2 \cdots X_k$
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:       $error()$

3.3.2 LL(1)文法

定义 16 (LL(1)文法[4]). *语法G是LL(1)文法当且仅当对于任意$A \to \alpha \mid \beta$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串*
3. *若$\beta \to \epsilon$，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$\alpha \to \epsilon$，则β不能推出任何以$FOLLOW(A)$中终端符号开始的字符串*

*前两个条件等价于$FIRST(\alpha) \cap FIRST(\beta) = \varnothing$，第三个条件等价于若$\epsilon \in FIRST(\beta)$，则$FIRST(\alpha) \cap FOLLOW(A) = \varnothing$，反之同理。这三个条件的作用都是**避免在输入符号上冲突**。*

通常没有左递归、无歧义的语法可以是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**。

例 17. *经典的二义if-else语法（已提取左因子）*

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

可以求得

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

[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）

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

算法 3 (构造LL(1)预测语法表). 对于每一生成式$A \to \alpha$，

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

注意是生成式右端。

例 18. 考虑以下文法

$$E \to T + E \mid T$$
$$T \to int \mid int * T \mid (E)$$

提取左因子后即得

$$E \to TX$$
$$X \to +E \mid \epsilon$$
$$T \to int\, Y \mid (E)$$
$$Y \to *T \mid \epsilon$$

$FIRST$集和$FOLLOW$集的推导可见上面的图，注意$FOLLOW(T)$和$FOLLOW(Y)$很容易推漏

$$\begin{aligned} FIRST(Y) &= \{*, \epsilon\} & FOLLOW(E) &= \{), \$\} \\ FIRST(T) &= \{int, (\} & FOLLOW(X) &= \{), \$\} \\ FIRST(X) &= \{+, \epsilon\} & FOLLOW(T) &= \{+,), \$\} \\ FIRST(E) &= \{int, (\} & FOLLOW(Y) &= \{+,), \$\} \end{aligned}$$

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。

	int	$*$	$+$	(	)	\$
E	TX			TX		
X			$+E$		ϵ	ϵ
T	$int\, Y$			(E)		
Y		$*T$	ϵ		ϵ	ϵ

基于表的预测语法分析，用栈实现。

Algorithm 6 Table-Driven Predictive Parsing

```
ip=0
X=stack.top()
while X ≠ $ do
    if X == w[ip] then
        stack.pop(); ip++;
    else
        if X is a terminal or M[X,a]=∅ then
            Error()
        else
            Output production M[X,a]= X → Y1Y2···Yk
            stack.pop()
            push Yk, Yk−1, ..., Y1 onto the stack
    X=stack.top()
if w[ip] != '$' then
    Error()
```

例 19. 考虑下面语法

$$
\begin{array}{lcl}
(1) & E & \to TE' \\
(2) & E' & \to +TE' \mid \epsilon \\
(3) & T & \to FT' \\
(4) & T' & \to *FT' \mid \epsilon \\
(5) & F & \to (E) \mid id
\end{array}
$$

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	$F \rightarrow id$			$F \rightarrow (E)$		

注意上述标红的项为通过第2条规则推导出来的项（$\epsilon \in FIRST(\alpha)$）。

例 20. 考虑以下文法：

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L, S \mid S$$

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

$$S \rightarrow (L) \mid a$$

$$L \rightarrow SL'$$

$$L' \rightarrow, SL' \mid \epsilon$$

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

Matched	*Stack*	*Input*	*Action*
	$S\$$	$(a,(a,a))\$$	
	$(L)\$$	$(a,(a,a))\$$	$output\ S \to (L)$
$($	$L)\$$	$a,(a,a))\$$	
$($	$SL')\$$	$a,(a,a))\$$	$output\ L \to SL'$
$($	$aL')\$$	$a,(a,a))\$$	$output\ S \to a$
$(a$	$L')\$$	$,(a,a))\$$	
$(a$	$,SL')\$$	$,(a,a))\$$	$output\ L' \to ,SL'$
$(a,$	$SL')\$$	$(a,a))\$$	
$(a,$	$(L)L')\$$	$(a,a))\$$	$output\ S \to (L)$
$(a,($	$L)L')\$$	$a,a))\$$	
$(a,($	$SL')L')\$$	$a,a))\$$	$output\ L \to SL'$
$(a,($	$aL')L')\$$	$a,a))\$$	$output\ S \to a$
$(a,(a$	$L')L')\$$	$,a))\$$	
$(a,(a$	$,SL')L')\$$	$,a))\$$	$output\ L' \to ,SL'$
$(a,(a,$	$SL')L')\$$	$a))\$$	
$(a,(a,$	$aL')L')\$$	$a))\$$	$output\ S \to a$
$(a,(a,a$	$L')L')\$$	$))\$$	
$(a,(a,a$	$)L')\$$	$))\$$	$output\ L' \to \epsilon$
$(a,(a,a)$	$L')\$$	$)\$$	
$(a,(a,a)$	$)\$$	$)\$$	$output\ L' \to \epsilon$
$(a,(a,a))$	$\$$	$\$$	

3.4 自底向上分析

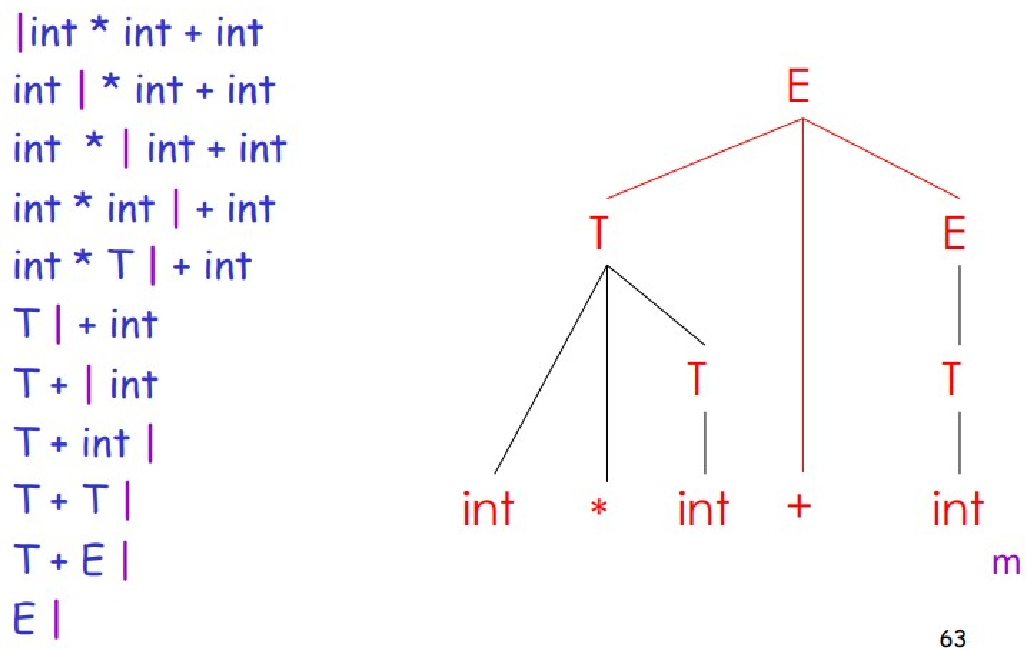
自底向上的语法分析采用两种动作：

- 移进(shift)：将 | 向右移动一格

$$ABC \mid xyz \implies ABCx \mid yz$$

- 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 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**（生成式的右端）而不是移进。

定义 18 (活前缀(viable prefix)). α是活前缀若存在ω使得$\alpha \mid \omega$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。

3.4.1 LR(0)语法

为构建规范(canoical)LR(0)项，**定义增量语法G'有生成式**$S' \to S$，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$S' \to S$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$A \to \alpha \cdot B\beta$在$CLOSURE(I)$中且$B \to \beta$是一个生成式，则将项$B \to \cdot\gamma$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$[A \to \alpha X \cdot \beta]$的闭包使得$[A \to \alpha \cdot X\beta]$在$I$中

定义 21 (核项(kernel item)). 初始项$S' \to \cdot S$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）

定义 22 (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$X \to \beta$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$Y \rightarrow \omega.t\delta$

3.4.2 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。

定义 23 (SLR(1)语法). 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$\mathcal{C} = \{I_0, I_1, \ldots, I_n\}$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：

$$
\begin{array}{llll}
(1) & E & \rightarrow & E + T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & TF \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & F^* \\
(6) & F & \rightarrow & a \\
(7) & F & \rightarrow & b
\end{array}
$$

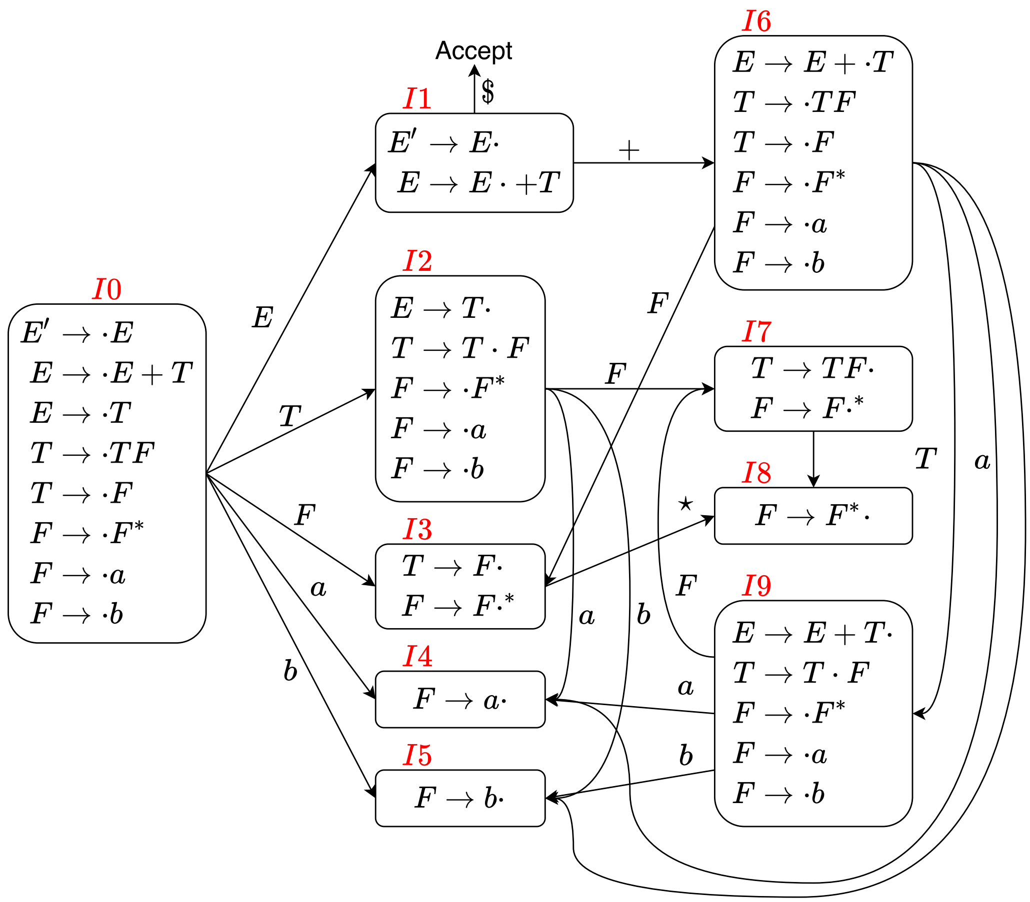
1. 写出每个非终端符号的$FIRST$集和$FOLLOW$集.
2. 构造识别这一文法所有活前缀*(viable prefixes)*的$LR(0)$自动机（参照课本*4.6.2*节图*4.31*）.
3. 构造这一文法的SLR分析表（参照课本*4.6.3*节图*4.37*）.
4. 给出SLR分析器识别输入串$a + ab^*$的过程（参照课本*4.6.4*节图*4.38*）

[5]或SLR(1)分析，通常省略(1)。用的是LR(0)项，但是在语法分析时才前看1个输入符号。

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$E' \to E$，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	$+ab^*\$$	$[4,a]r6\ F\to a,\ [0,F]s3$
(3)	03	F	$+ab^*\$$	$[3,+]r4\ T\to F,\ [0,T]s2$
(4)	02	T	$+ab^*\$$	$[2,+]r2\ E\to T,\ [0,E]s1$
(5)	01	E	$+ab^*\$$	$[1,+]s6$
(6)	016	$E+$	$ab^*\$$	$[6,a]s4$
(7)	0164	$E+a$	$b^*\$$	$[4,b]r6\ F\to a,\ [6,F]s3$
(8)	0163	$E+F$	$b^*\$$	$[3,b]r4\ T\to F,\ [6,T]s9$
(9)	0169	$E+T$	$b^*\$$	$[9,b]s5$
(10)	01695	$E+Tb$	$^*\$$	$[5,*]r7\ F\to b,\ [9,F]s7$
(11)	01697	$E+TF$	$^*\$$	$[7,*]s8$
(12)	016978	$E+TF^*$	$\$$	$[8,\$]r5\ F\to F^*,\ [9,F]s7$
(13)	01697	$E+TF$	$\$$	$[7,\$]r3\ T\to TF,\ [6,T]s9$
(14)	0169	$E+T$	$\$$	$[9,\$]r1\ E\to E+T,\ [0,E]s1$
(15)	01	E	$\$$	$[1,\$]ACC$

例 22. 下面的文法并非*SLR(1)*

$$S\to L=R\mid R$$
$$L\to *R\mid id$$
$$R\to L$$

对于I_2项：

$$S\to L\cdot=R$$
$$R\to L\cdot$$

有$ACTION[2,=]$是移进，但$FOLLOW(R)=\{\$,=\}$又会导致在$=$上进行规约，故移进-规约冲突，该文法不是*SLR(1)*的

3.4.3 LR(1)语法

定义 24 (LR(1)项). $[A\to\alpha\cdot\beta,a]$，其中$A\to\alpha\cdot\beta$为该项的核*(core)*，$A\to\alpha\beta$为生成式，a是终端符号或*\$*，*1*指项中第二个元素的长度，a也被称为前看*(lookahead)*。只有当*LR(1)*项有$[A\to\alpha\cdot,a]$的形式，**且下一输入符号为a时，才会用$A\to\alpha$进行规约**（这在构造解析表时会用到）。a总会是$FOLLOW(A)$的子集，但往往是真子集。

算法 5 (计算$CLOSURE(I)$)**.** 对于每一$[A\to\alpha\cdot B\beta,a]\in I$，每一$G'$中的生成式$B\to\gamma$，$\forall b\in FIRST(\beta a)$，将$[B\to\cdot\gamma,b]$加入$I$中。

初始化$\mathcal{C}=\{CLOSURE(\{[S'\to\cdot S,\$]\})\}$。

例 23. 考虑下面的增量语法

$$\begin{array}{lll}(1) & S' & \to S\\ (2) & S & \to CC\\ (3) & C & \to cC \mid d\end{array}$$

有$FIRST(C)=\{c,d\}$，可以构造得下面的*LR(1)*自动机。以$[S\to\cdot CC,\$]$为例，考虑$FIRST(C\$)=\{c,d\}$，故闭包会新增四项$[C\to\cdot cC,c]$、$[C\to\cdot cC,d]$、$[C\to\cdot d,c]$、$[C\to\cdot d,d]$。

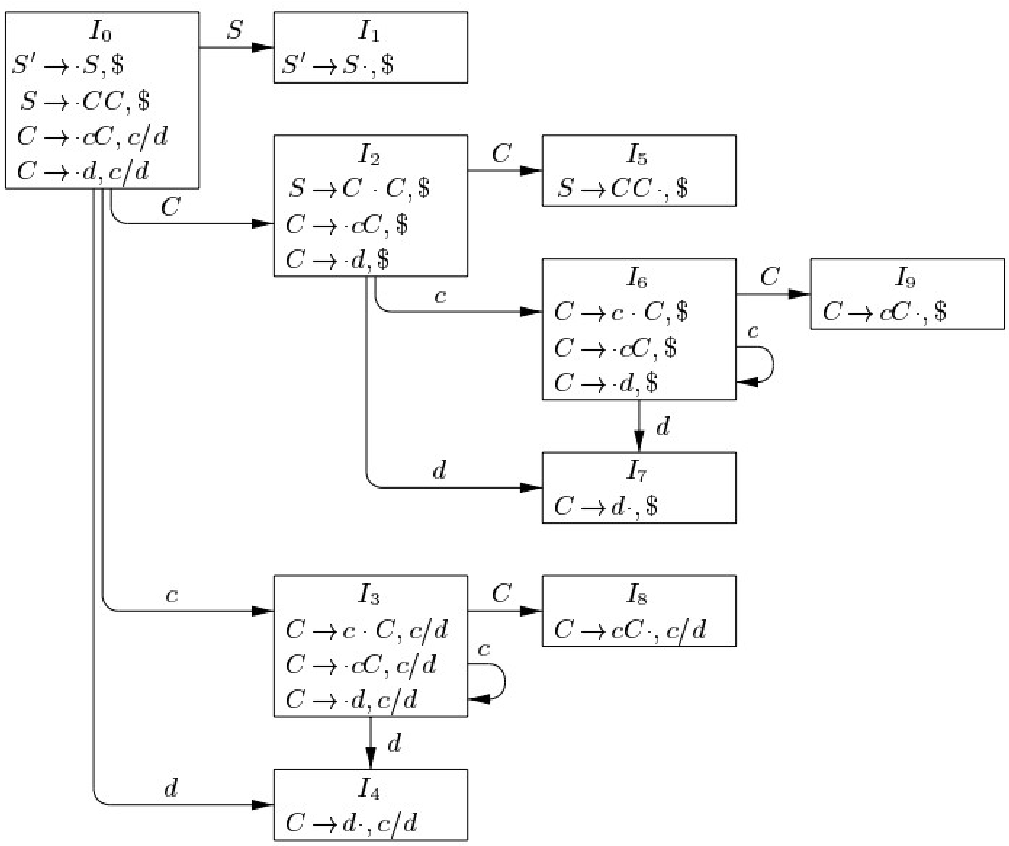


Figure 4.41: The GOTO graph for grammar (4.55)

类似地有规范*LR(1)*解析表

	c	d	$\$$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(0)=SLR(1)的表项较少，但LR(1)的表项就指数级上涨。

3.4.4 LALR分析

LALR(Lookahead LR)在实践中很常用，表大小通常远小于规范LR表。核心思想是将LR(1)中具有**相同核**的表项合并。

例 24. 将上面例子中的I_3和I_6合并得到

$$C \to c \cdot C, c/d/\$$$
$$C \to \cdot cC, c/d/\$$$
$$C \to \cdot d, c/d/\$$$

I_4和I_7合并得到

$$C \to d\cdot, c/d/\$$$

I_8和I_9合并得到

$$C \to cC\cdot, c/d/\$$$

进而有*LALR*解析表

	c	d	$\$$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

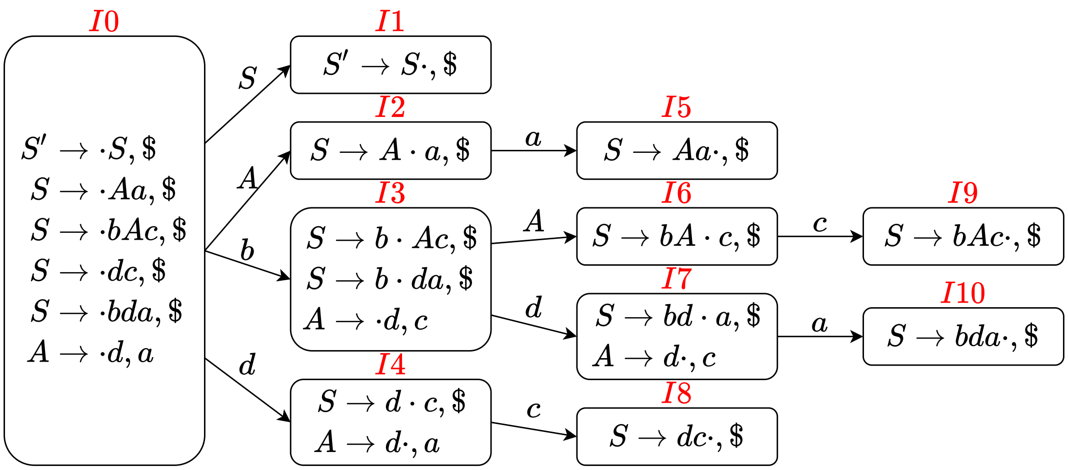
合并LR(1)项不会导致新的移进-规约冲突，但可能会产生新的**规约-规约冲突**。

例 25. 证明下列文法

$$S \to Aa \mid bAc \mid dc \mid bda$$
$$A \to d$$

是*LALR(1)*文法但不是*SLR(1)*文法.

分析. 构造增广文法$S' \to \cdot S$，$FIRST(\epsilon\$) = \$$，$FIRST(a\$) = a$，可以得到$I_0$。



由上图知，没有相同核心*(core)*的状态，因此不需要合并，从而*LALR*分析表不冲突，该文法是*LALR(1)*文法。

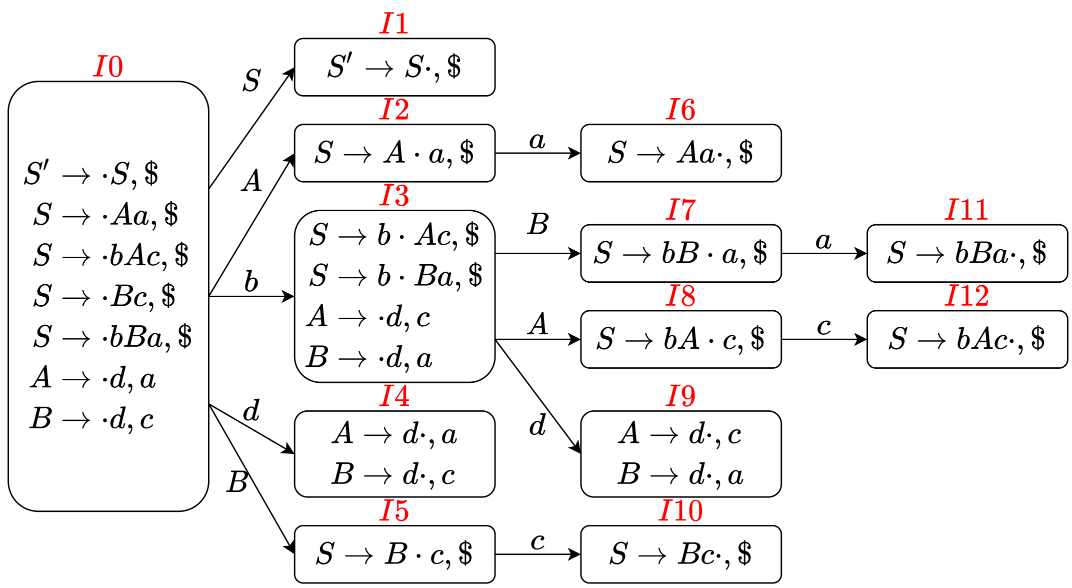
又有$FOLLOW(A) = \{a, c\}$，考虑图中的状态$I4$，当输入符号为c时，$c \in FOLLOW(A)$，既有移进$S \to d \cdot c$，又有归约$S \to d\cdot$，因此*SLR*分析表有冲突，该文法不是*SLR(1)*文法。

例 26. 证明下列文法

$$S \to Aa \mid bAc \mid Bc \mid bBa$$
$$A \to d$$
$$B \to d$$

是*LR(1)*文法但不是*LALR(1)*文法.

分析. 构造增广文法$S' \to \cdot S$，$FIRST(a\$) = a$，$FIRST(c\$) = c$，可以得到状态$I_0$。



由上面的*DFA*知*LR*分析表没有冲突，因此该文法是*LR(1)*文法。

但如果将图中相同核心的状态$I4$和$I9$合并，会有

$$A \to d\cdot, a/c$$

$$B \to d\cdot, a/c$$

即出现了规约-规约冲突，因此该文法不是*LALR(1)*文法。

可以在解析表(parsing table)层面来解决语法的二义性，即选择移进/规约的特定操作。

3.5 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 27. 对下列语法进行求值

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

有语法制导定义

$$\begin{array}{ll} E \to int & E.val = int.val \\ E \to E_1 + E_2 & E.val = E_1.val + E_2.val \\ E \to (E_1) & E.val = E_1.val \end{array}$$

可以在AST上标注(annotate)两种属性：

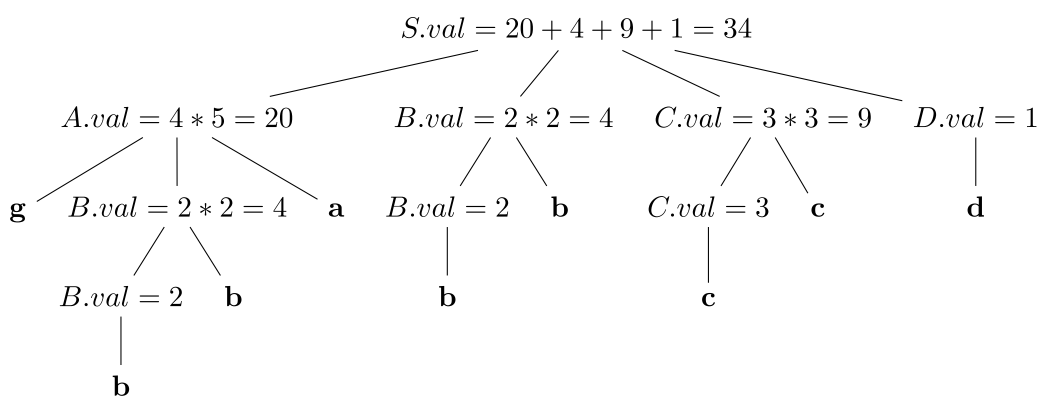
- 继承属性(inherited)：从语法树的父亲或兄弟中计算得到
- 综合属性(synthesized)：从后代计算得到

例 28. 考虑以下语法制导定义：

语法规则	语义规则
$S \to ABCD$	$S.val = A.val + B.val + C.val + D.val$
$A \to gBa$	$A.val = B.val * 5$
$B \to B_1 b$	$B.val = B_1.val * 2$
$B \to b$	$B.val = 2$
$C \to C_1 c$	$C.val = C_1.val * 3$
$C \to c$	$C.val = 3$
$D \to d$	$D.val = 1$

对于输入串$gbbabbccd$构造带注释的分析树*(annotated parse tree)*.

分析. 带注释的分析树如下



例 29. 下列文法定义了二进制整数的语法规则

$$N \to SL$$
$$L \to LB \mid B$$
$$S \to + \mid -$$
$$B \to 0 \mid 1$$

给出语法制导定义，求出该二进制整数的十进制值，存在综合属性`val`中

序号	生成式	语义规则
1	$N \to SL$	$N.val = S.sign * L.val$
2	$L \to L_1 B$	$L.val = L_1.val * 2 + B.val$
3	$L \to B$	$L.val = B.val$
4	$S \to +$	$S.sign = 1$
5	$S \to -$	$S.sign = -1$
6	$B \to 0$	$B.val = 0$
7	$B \to 1$	$B.val = 1$

扩展文法：

$$
\begin{aligned}
expr &\to expr_1 + term && \{print('+')\}\\
expr &\to expr_1 - term && \{print('-')\}\\
expr &\to term &&\\
term &\to 0 && \{print('0')\}\\
term &\to 1 && \{print('1')\}\\
&\vdots &&\\
term &\to 9 && \{print('9')\}
\end{aligned}
$$

大括号中的语句称为动作(action)，这一系列产生式称为翻译模式(translation scheme)。

将动作视为产生式右端的一部分，则可得到扩展的语法树。对语法分析树做先序遍历，则可以得到后缀表达式。

3.6 总结

语法分析分为自顶向下和自底向上两大类，自顶向下是从开始符号出发，根据产生式规则**推导**给定的句子；自底向上则是由给定的句子**规约**到文法的开始符号。

其中LL(1)是自顶向下的分析法，LR(0)、SLR(1)、LR(1)、LALR(1)是自底向上的分析法。

文法	文法要求	构造分析表
LL(1)	生成式首符号不能相同	将生成式加到首符号项上
LR(0)	LR(0)自动机无冲突	
SLR(1)	下一输入$a \in FOLLOW(X)$时进行规约	ACTION-GOTO表同左
LR(1)	用$FIRST(\beta a)$确定前看项无冲突	下一输入为a才规约$[A \to \alpha\cdot, a]$
LALR(1)	将LR(1)相同核项合并无冲突	同上

文法处理能力：LR(0) $<$ SLR(1) $<$ LALR(1) $<$ LR(1)

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。

$S \to id = E$	`S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)`
$E \to E_1 + E_2$	`E.addr = new Temp()` `E.code = E1.code || E2.code || gen(E.addr '=' E1.addr '+' E2.addr)`
$E \to -E_1$	`E.addr = new Temp()` `E.code = E1.code || gen(E.addr '=' minus E1.addr)`
$E \to (E_1)$	`E.addr = E1.addr` `E.code = E1.code`

$L \to L_1[E]$	`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)`

S→if (B) S1 else S2	`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`

三地址码与对应的DAG如下图所示，注意这里将相同终端符号结点都给合并了。

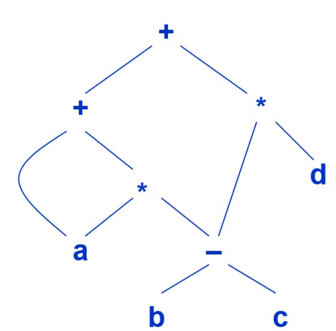
$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

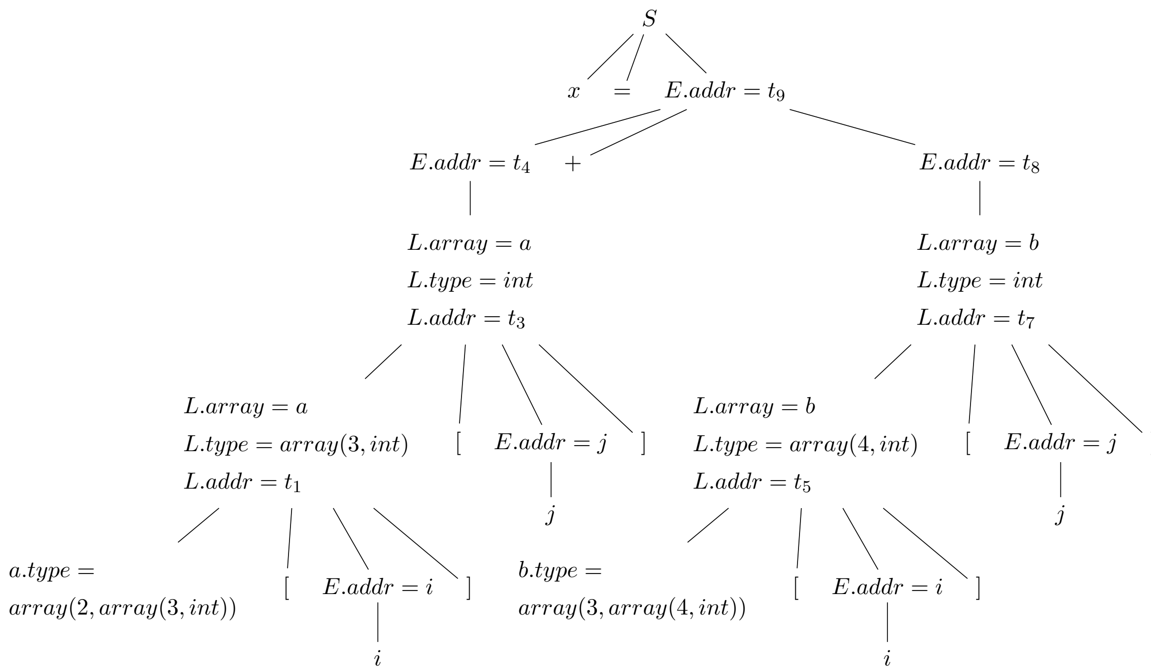
$t_4 = t_1 * d$

$t_5 = t_3 + t_4$



例 30. 令a表示一个2×3的整型数组，b表示一个3×4的整型数组. 假定一个整数的宽度为4. 试使用课本图*6.22*的翻译模式，翻译赋值语句`x=a[i][j]+b[i][j]`. 提示：参考课本例*6.12*.

分析. 带注释的分析树如下

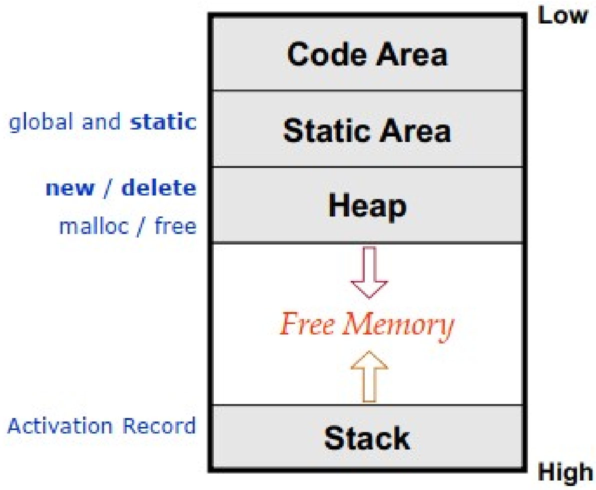


生成的三地址码如下

$$
\begin{aligned}
&t_1 = i * 12\\
&t_2 = j * 4\\
&t_3 = t_1 + t_2\\
&t_4 = a[t_3]\\
&t_5 = i * 16\\
&t_6 = j * 4\\
&t_7 = t_5 + t_6\\
&t_8 = b[t_7]\\
&t_9 = t_4 + t_8\\
&x = t_9
\end{aligned}
$$

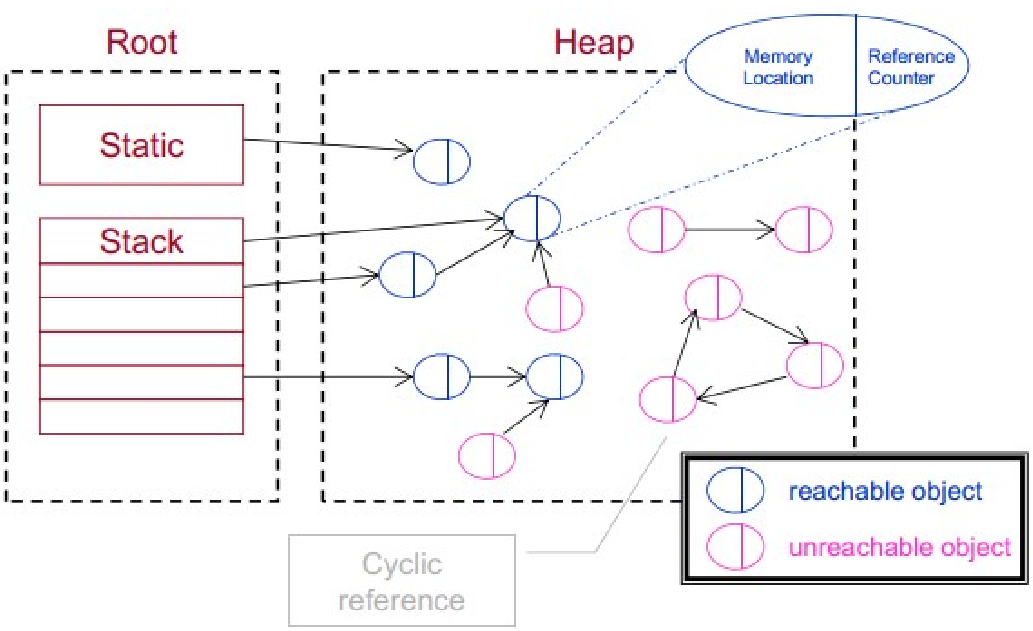
5 运行时系统

5.1 存储管理



5.2 垃圾回收

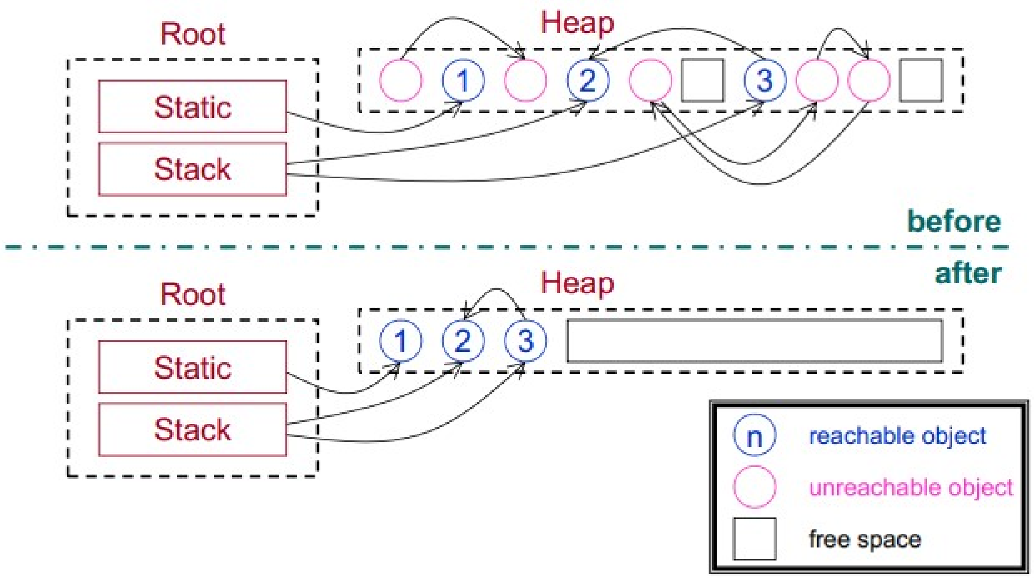
- 引用计数(reference counting)：创建加1，删除减1
  - 简单、立即增量式回收
  - 不能回收循环引用的示例



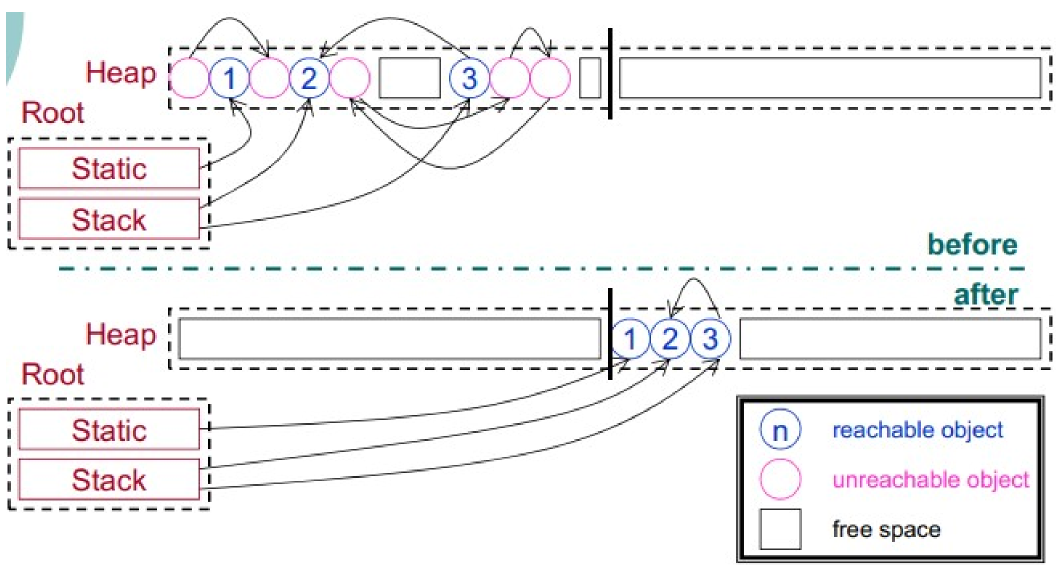
- 标记扫除(mark and sweep)：做图深搜找连通块
  - 有办法清除循环引用

– 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 25 (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

例 31. 考虑以下基本块:

$$t_0 = 5$$

$$t_1 = 3 * t_0$$

$$t_2 = R + r$$

$$t_3 = t_1 * t_2$$

$$t_4 = t_2$$

$$t_5 = t_3 - t_4$$

$$t_6 = t_1 * t_2$$

$$A = t_6 + t_5$$

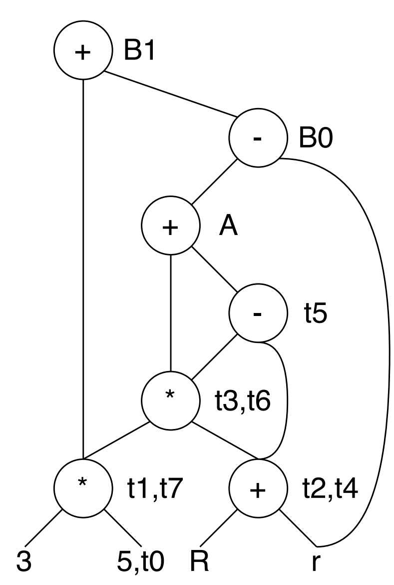
$$B = A - r$$

$$t_7 = t_1$$

$$B = t_7 + B$$

1. 构造这一基本块的DAG.
2. 假设只有A和B在基本块后面还要被引用，产生优化后的三地址代码.

分析. 1. 基本块的DAG如下图所示



2. 由于只有A和B在基本块后面还要被引用，因此最终只需保留A和B的结果即可。无依赖关系的代码都可以作为死代码优化删除。最终优化后的三地址代码如下

$$
\begin{aligned}
t_2 &= R + r \\
t_3 &= 15 * t_2 \\
t_5 &= t_3 - t_2 \\
A &= t_3 + t_5 \\
B &= A - r \\
B &= 15 + B
\end{aligned}
$$

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
  - 冗余操作消除
- 循环优化：在循环内的优化
- 全局优化：最粗粒度的优化

定义 26 (循环(loop)). 只有唯一入口/头的强连通子图

例 32. 考虑下列代码片段：

(1) `m := 0`

(2) `v := 0`

(3) `if v >= n goto (19)`

(4) `r := v`

(5) `s := 0`

(6) `if r < n goto (9)`

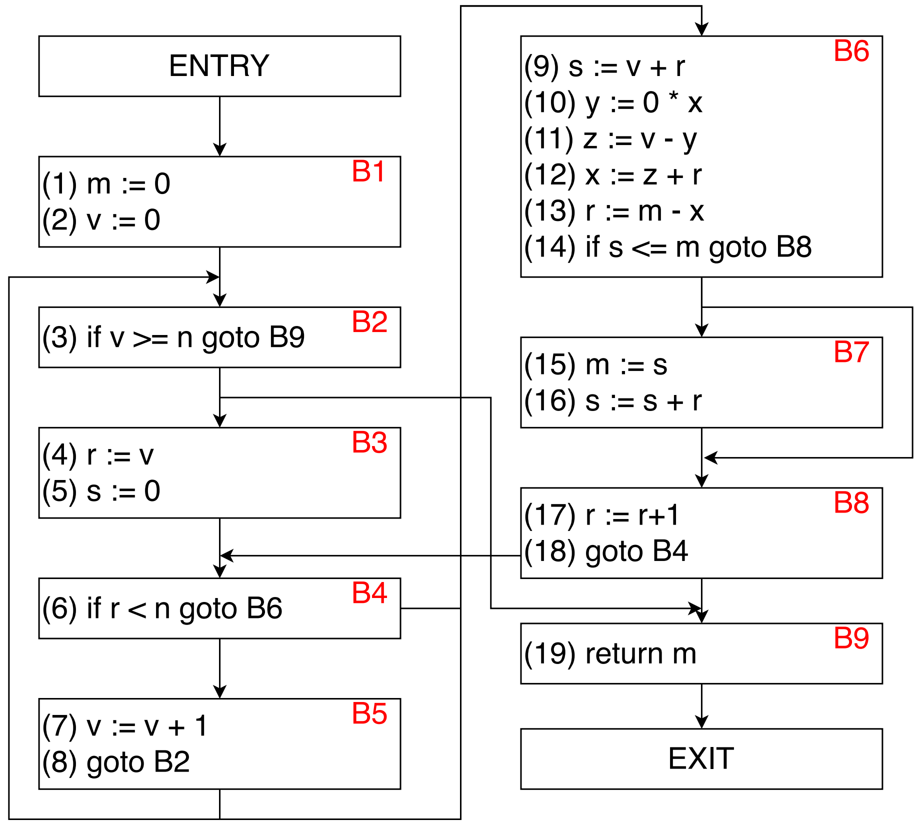
(7) `v := v + 1`

(8) `goto (3)`

```
(9)  s := v + r
(10) y := 0 * x
(11) z := v - y
(12) x := z + r
(13) r := m - x
(14) if s <= m goto (17)
(15) m := s
(16) s := s + r
(17) r := r+1
(18) goto (6)
(19) return m
```

为这段代码划分基本块(*Basic Block*)，并画出控制流图(*Control Flow Graph*). 在答案中你可以直接画出控制流图，但对图中的每个结点，请用$m \sim n$表示相应的基本块由第m至第n条语句组成.

分析. 如下图所示，共9个基本块，每个基本块包含的语句已在图中标出。


ENTRY
B1
(1) m := 0
(2) v := 0
B2
(3) if v >= n goto B9
B3
(4) r := v
(5) s := 0
B4
(6) if r < n goto B6
B5
(7) v := v + 1
(8) goto B2
B6
(9) s := v + r
(10) y := 0 * x
(11) z := v - y
(12) x := z + r
(13) r := m - x
(14) if s <= m goto B8
B7
(15) m := s
(16) s := s + r
B8
(17) r := r+1
(18) goto B4
B9
(19) return m
EXIT