第8章　　语法制导翻译法(1)

- 翻译程序：它将用某种源语言写的程序(源程序)转换成等价的用某种目标语言写的程序(目标程序)，其中的目标程序可以是某种中间语言程序。
- 中间语言:如汇编语言程序、四元式形式的程序等等;
- 语法制导翻译:就是先给文法中的每个产生式添加一个成分，这个成分常称为语义动作或翻译子程序，在执行语法分析的同时，执行相应产生式的语义动作。
- 这些语义动作不仅指明了该产生式所生成的符号串的意义，而且还根据这种意义规定了对应的加工动作。这些加工动作包括查填各类表格、改变编译程序的某些变量之值、打印各种错误信息以及生成中间语言程序等等。一旦某个产生式选用之后，接着就执行相应的语义动作，完成预定的翻译工作。

本课程介绍的语法分析方法

<1> 推导

<2> 语法树

<3> LL分析

<4> 优先法

<5> LR分析

第8章 语法制导翻译法(2)

- **编译程序—翻译程序：**源程序=>等价的目标程序。
- **目标程序—可以是中间代码：**汇编语言程序、四元式、三元式、逆波兰表示等。
- **语法制导翻译：**给文法中的每个产生式添加一个成分—语义动作或翻译子程序。

8.1 一般原理和树变换

- 语法制导翻译法(SDTS)

  一个源语言

  一个目标语言

  一组翻译规则
- SDTS是一个CFG(Context Free Grammar)

  上下文无关文法。

SDTS是一个上下文无关文法

- $T=(V_T, V_N, \Delta, R, S)$

  V_T：有穷输入字母表

  V_N：有穷非终结符号集合

  Δ：有穷输出字母表

  R：一组$A \rightarrow w, y$ 规则的有穷集合

  $A \in V_N$，$w \in (V_T \cup V_N)^*$，

  $y \in (\Delta \cup V_N)^*$

  S：开始符号

8.1一般原理和树变换

8.1.1 一般原理(1)

- 直观地说，语法制导翻译法(SDTS)由一个源语言、一个目标语言和一组翻译规则组成，这组规则可将任何源语言符号串翻译成对应的目标语言串。
- SDTS的翻译规则是文法中的产生式再添加上语义动作。作为一个概念模型，SDTS提供了一个极好的框架以理解翻译程序的某些基本原理。因此，我们首先给出SDTS的定义并讨论它的特性，然后，再介绍实现它的方法。
- SDTS也是一个CFG，其形式定义如下：

SDTS是一个五元组T=(V_T，V_N，Δ，R，S)

8.1一般原理和树变换

8.1.1 一般原理(2)

- SDTS是一个五元组 $T=(V_T, V_N, \Delta, R, S)$

V_T是一个有穷的输人字母表，包含源语言中的符号：

V_N是一个有穷的非终结符号集合：

Δ是一个有穷的输出字母表，Δ包含出现在翻译串或输出串中的那些符号；

R是形如$A\rightarrow w, y$的规则的有穷集合(这种规则的定义见后)：

$S\in V_N$是一个开始符号，其含义和用法如同CFG中的开始符号。

8.1一般原理和树变换

8.1.1　一般原理(3)

- R中的规则形如　　$A \rightarrow w, y$　　$A \in V_N$
  - w是由终结符和(或)非终结符组成的串：
  - y则是由V_N和(或)Δ中的符号组成的串。注意：出现在w和y中的非终结符必须是一一对应的。
  - 串w称为规则的源成分；串y称为规则的翻译成分。R中的规则有时也称为翻译规则。
- T的基础源文法是一个CFG：(V_N，V_T，P，S)，其中:
  - P是形如$A \rightarrow w$(即T中源成分)的产生式的集合;
  - $A \rightarrow w, y$是T中R的一个规则. 也就是说，从T中去掉输出字母表Δ，再从T的规则中移走翻译成分，就可得到T的基础源文法。
  - 类似地，也可以定义T的基础目标文法，即从T中去掉输入字母表V_T并从丁的规则中移去源成分。

8.1一般原理和树变换

8.1.1 一般原理(4)

- 例如，考虑SDTS T1=({a，b，c，+，-，[，]}，{E，T，A}，{ADD，SUB，NEG，x，y，z}，R，E)，其中R由下列翻译规则组成：

① E→E+T，T E ADD

② E→E-T，E T SUB

③ E→-T，T NEG

④ E→T，T

⑤ T→[E]，E

⑥ T→A，A

⑦ A→a，x

⑧ A→b，y

⑨ A→c，z

- T1的基础源文法是：

E→E+T|E-T|-T|T

T→[E]|A

A→a|b|c

- T1的基础目标文法则是：

E→T E ADD |

E T SUB |

T NEG | T

T→E | A

A→x|y|z

8.1一般原理和树变换

8.1.1 一般原理(5)

- 一个翻译模式是一个形如(u, v)的串对，其中：

➢ u是SDTS基础源文法的一个句型，它是由V_N和V_T中元素组成的串。

➢ 而v称为与其对应的翻译，它是由V_N和Δ中元素组成的串。

- 翻译模式的定义如下：

①(S, S)是一个翻译模式，且这两个S是相关的(S是SDTS的开始符号).

②(aAb，a′Ab′)是一个翻译模式，且两个A是相关的：此外，若A→g，g′是R中的一条规则，那么(agb，a′g′b′)也是一个翻译模式。规则中g和g′的非终结符之间的相关性也必须带进这种翻译模式之中。

8.1一般原理和树变换

8.1.1　一般原理(6)

- 表示法

  (aAb，a′Ab′) => (agb，a′g′b′)

  表示一种翻译模式到另一种翻译模式的变换。

- 不难看出，一个翻译模式的第一部分恰好是SDTS中基础源文法(是一个CFG)的一个句型：
- 而第二部分是其对应的翻译，即基础目标文法的一个句型。

由一个SDTS T所定义的翻译是下述对偶集：

$$\{(x,y) \mid (S,S) \Rightarrow^* (x,y), x \in V_T^* \;\&\; y \in \Delta^*\}$$

显然，它类似于CFG中“语言”的定义.

例如，考虑输入串　　–[a+c]–b

- 它是可从SDTS T1的基础源文法推出的，即

基础文法T 1(E):

E→E+T | E-T | -T | T

T→[E] | A

A→a | b | c

E => E–T

=> –T–T

=> –[E]–T

=> –[E+T]–T

=> –[T+T]–T

=> –[A+T]–T

=> –[a+T]–T

=> –[a+A]–T

=> –[a+c]–T

=> –[a+c]–A

=> –[a+c]–b

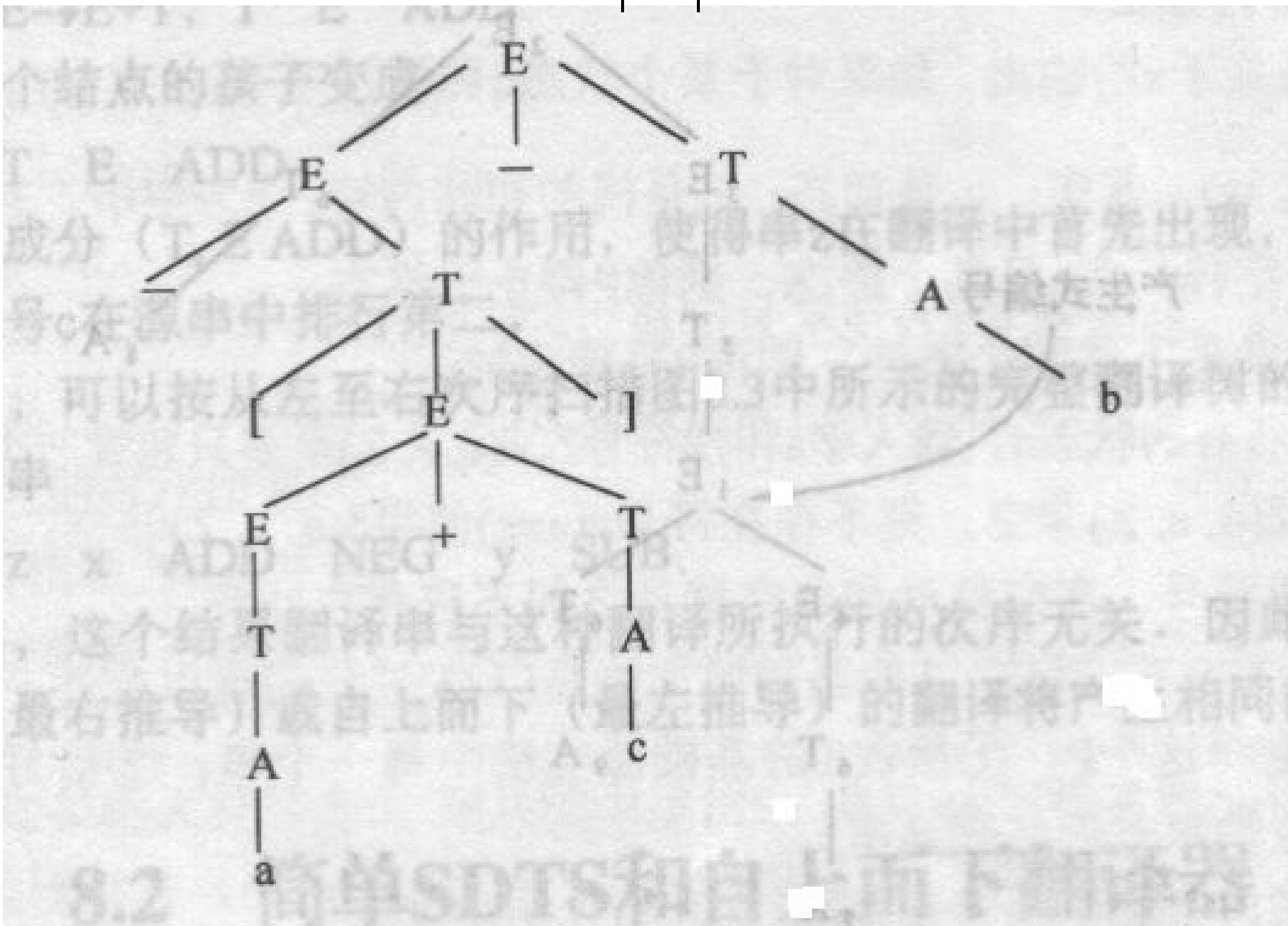


图8.1　–[a+c]–b的语法树

- 基础文法T 1(E):

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

  $T \rightarrow [E] \mid A$

  $A \rightarrow a \mid b \mid c$

- SDTS T1:

  $E \rightarrow E+T, T\ E\ \text{ADD}$

  $E \rightarrow E-T, E\ T\ \text{SUB}$

  $E \rightarrow -T, T\ \text{NEG}$

  $E \rightarrow T, T$

  $T \rightarrow [E], E$

  $T \rightarrow A, A$

  $A \rightarrow a, x$

  $A \rightarrow b, y$

  $A \rightarrow c, z$

- 翻译模式是一个形如(u,v)串对
- 例如对输入串 -[a+c]-b的翻译过程

$(E,E) \Rightarrow (E-T_2,\ E\ T_2\ \text{SUB})$

$\Rightarrow (-T_1-T_2,\ T_1\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[E]-T_2,\ E\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[E+T_1]-T_2,\ T_1\ E\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[T_3+T_1]-T_2,\ T_1\ T_3\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[A+T_1]-T_2,\ T_1 A\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[a+T_1]-T_2,\ T_1\ x\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[a+A]-T_2,\ A\ x\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[a+c]-T_2,\ z\ x\ \text{ADD}\ \text{NEG}\ T_2\ \text{SUB})$

$\Rightarrow (-[a+c]-A,\ z\ x\ \text{ADD}\ \text{NEG}\ A\ \text{SUB})$

$\Rightarrow (-[a+c]-b,\ z\ x\ \text{ADD}\ \text{NEG}\ y\ \text{SUB})$

- 为清楚起见，上述翻译模式中的某些非终结符写出了下标，以指明输入(源)串和输出(目标)串中所出现的非终结符之间的相关性。因此，出现在第二个翻译模式中的两个T是分别用T_1和T_2来区分的。
- 因此，输入串-[a+c]-b所对应的翻译是：

  z x ADD NEG y SUB
- 事实上，已经有了将一个(中缀)表示形式的算术表达式翻译成与其对应的后缀表达式的翻译器。
- 而且，通过该SDTS中最后的三条规则($A\rightarrow a,x|b,y|c,z$)，引进了某些词法操作以使这个翻译过程清晰化。
- 当然，在一个实际的编译程序中，标识符的集合是由某个终结符(例如id)代表的而且是通过词法分析程序识别和形成的。

8.1.2　树变换

- 语法制导的翻译过程也可用语法树来说明。

  图8.1给出了根据T_1的基础源文法推导输入串-[a+c]-b的语法树。它的翻译也可看做从一棵树到另一棵树的变换，其变换过程如下：

✓从树中剪掉终结符符号的结点

✓重新排列结点的孩子

✓添加对应输出符号Δ的终结符结点

例如，图8.2给出了去掉图8.1中终结符结点之后的语法树．在这种情形下，两个本来不同的产生式由于去掉了终结符可能会变得完全相同。例如，产生式

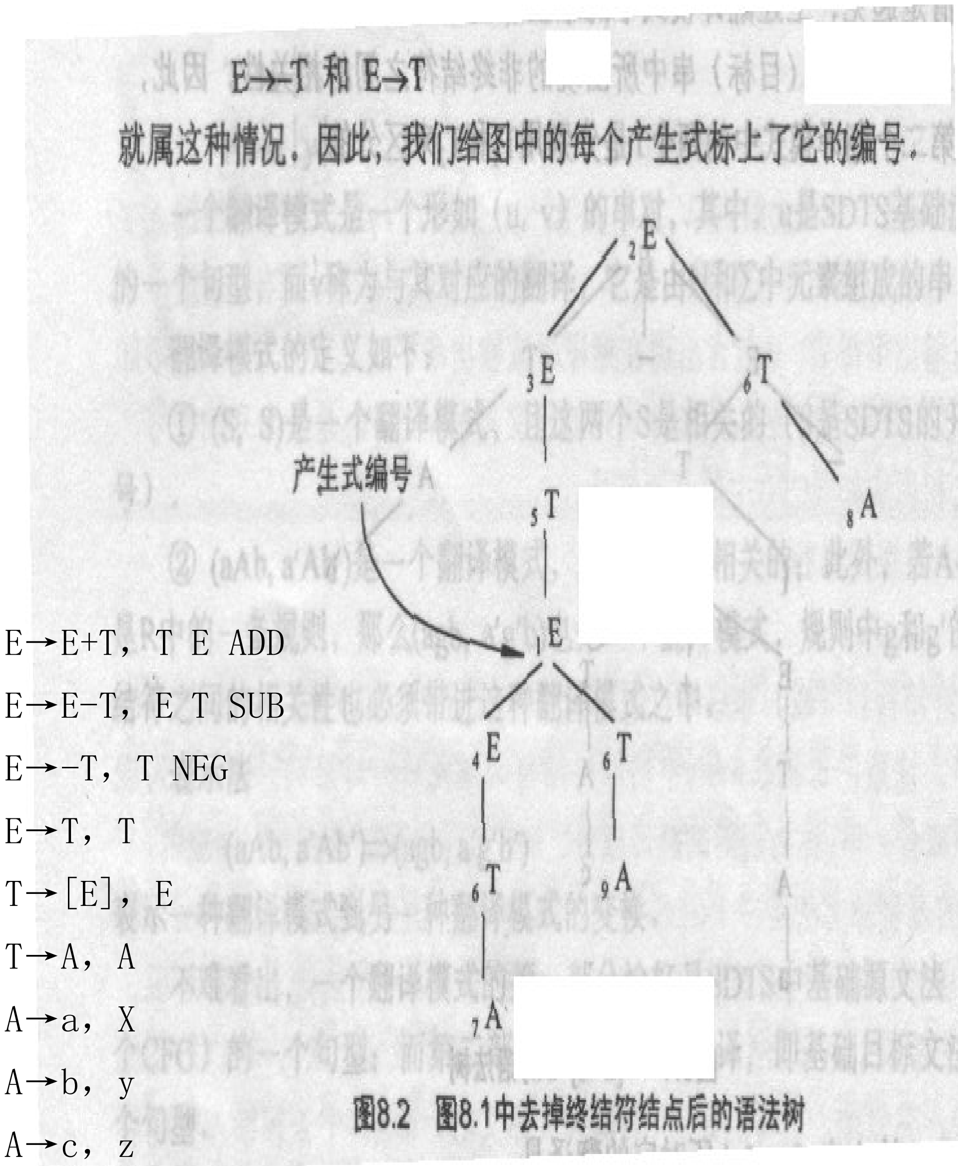

E→-T 和 E→T
就属这种情况，因此，我们给图中的每个产生式标上了它的编号．

图8.2 图8.1中去掉终结符结点后的语法树

① E→E+T，T E ADD

② E→E-T，E T SUB

③ E→-T，T NEG

④ E→T，T

⑤ T→[E]，E

⑥ T→A，A

⑦ A→a，X

⑧ A→b，y

⑨ A→c，z

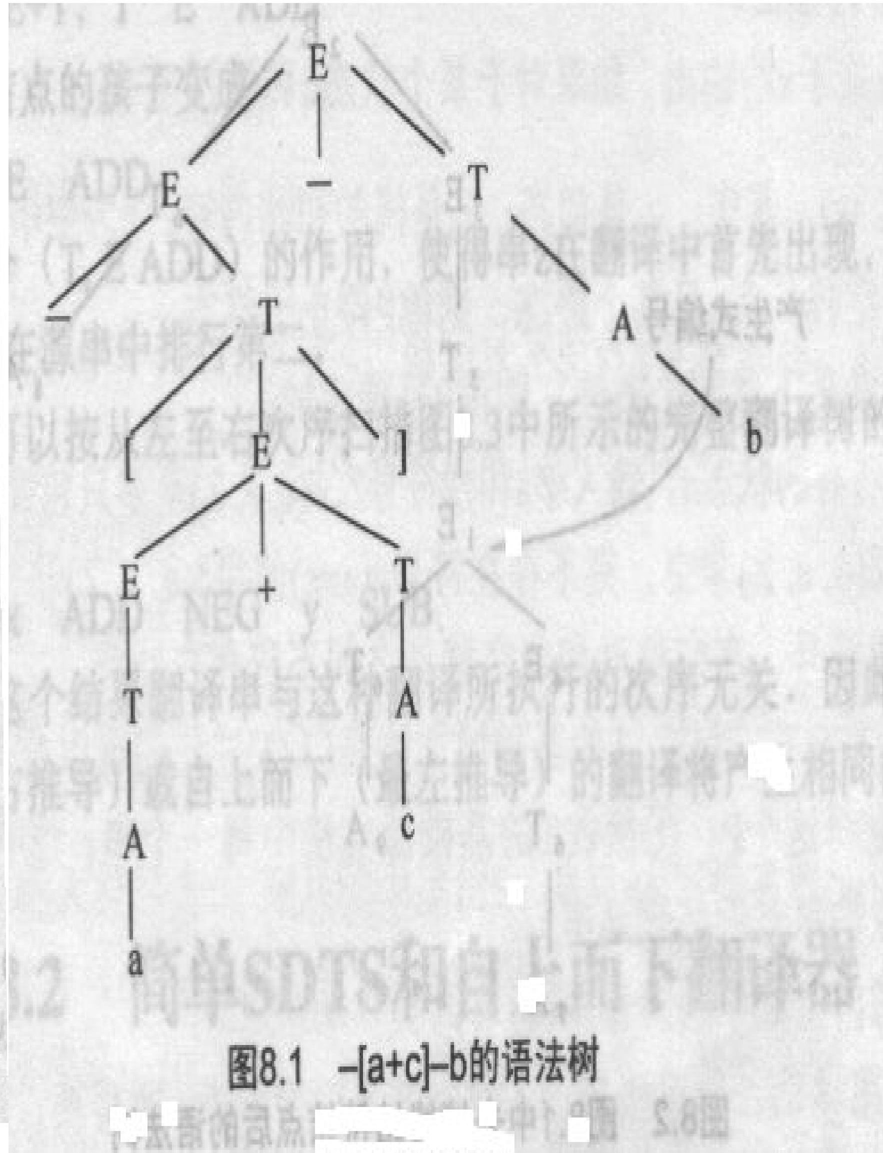


图8.1 -[a+c]-b的语法树

从树中剪掉终结符符号的结点
重新排列结点的孩子
添加对应输出符号Δ的终结符结点
因此，输入串-[a+c]-b所对应的翻译是：

z x ADD NEG y SUB

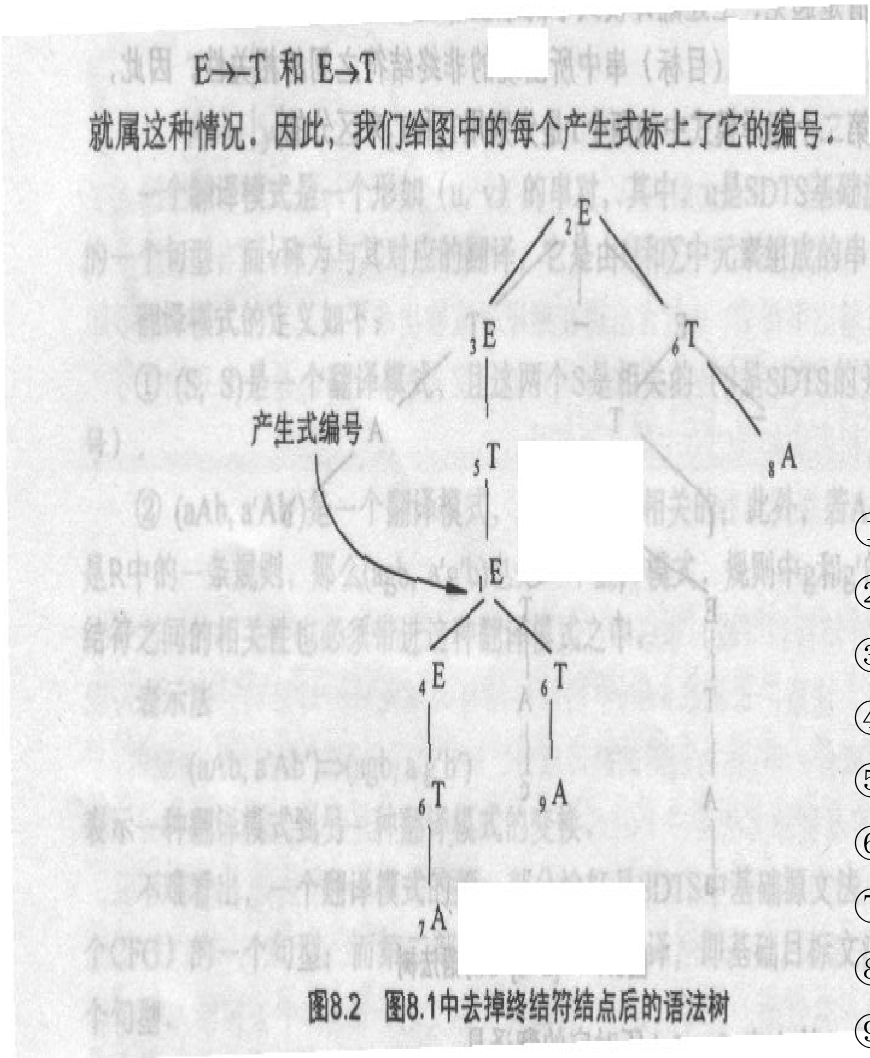


图8.2 图8.1中去掉终结符结点后的语法树

① E→E+T，T E ADD

② E→E-T，E T SUB

③ E→-T，T NEG

④ E→T，T

⑤ T→[E]，E

⑥ T→A，A

⑦ A→a，X

⑧ A→b，y

⑨ A→c，z

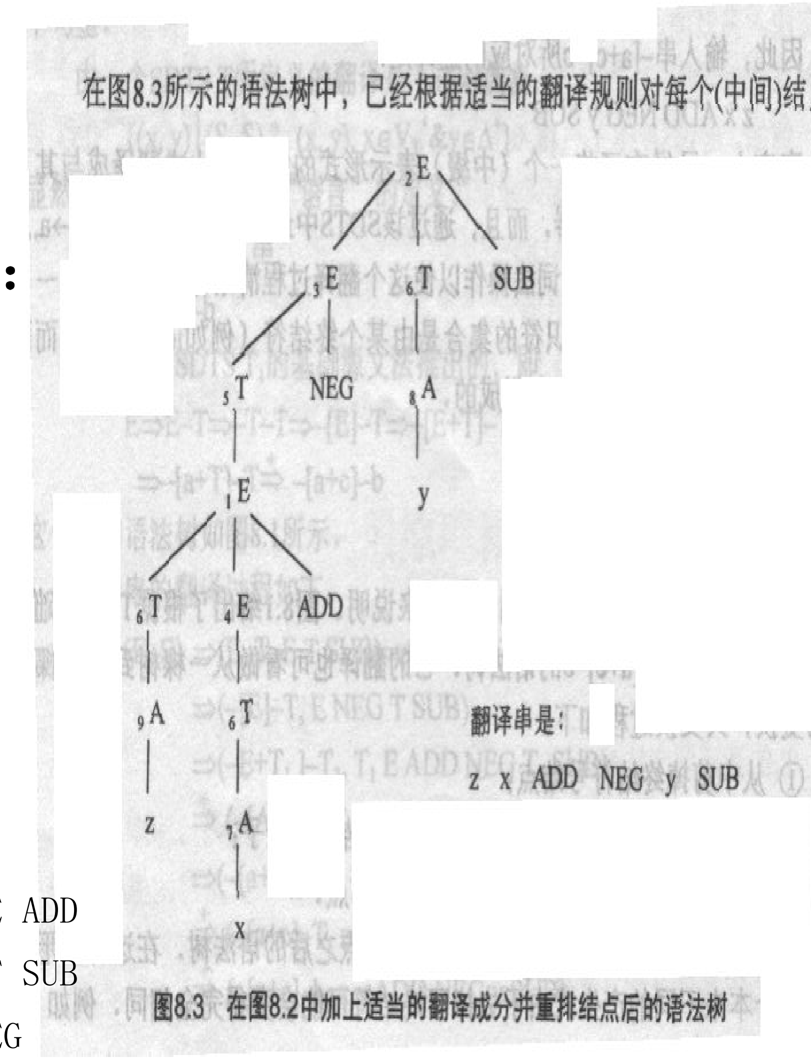


图8.3 在图8.2中加上适当的翻译成分并重排结点后的语法树

8.2 简单SDTS和自上而下翻译器

- 如果每一规则的翻译成分中，非终结符出现的次序与它们在源成分出现的次序相同，则称一个SDTS是简单的(simple)；
- 但是，如果 A→A1+A2， A2 A1 ADD

是SDTST的一个规则，那么，该SDTS T就不是简单的。

- 显然，利用简单的SDTS进行翻译不需要重排语法树的结点，只要去掉源成分中的终结符(结点)并插入相应的翻译成分中的终结符即可。

- 定理8.1　如果$T=(V_N, V_T, \Delta, R, S)$是其基础源文法为LL(K)的简单SDTS，那么，存在一个自上而下的确定的下推翻译器PDT(Push-Down Translater)，它接收T的输入语言中的任何符号串并产生对应的输出串。PDT P的定义如下：P的一个构形是一个四元组(q,x,y,z),其中，q是它的有穷控制器的状态，x是尚待扫描的输入串，y是下推栈，z是此时被打印出的输出符号串。
- 于是，在某次移动中，便有

$$(q, ax, Yy', z) \vdash (r, x, gy', zz')$$

其中，存在一条翻译器规则

$$\delta(q, a, Y)\text{包含}(r, g, z')$$

换言之，在状态q面临输入符号a且栈顶符号为Y时，P才允许移动到状态r，这一移动的结果是：输人符号a被去掉，栈顶符号Y被g所替换，而且串z′已作为输出串打印出。

- 称w是关于x的输出，如果对于某个状态q和栈符号串u，存在$(q_0, x, z_0, \varepsilon) \vdash^* (q, \varepsilon, u, w)$。
- 其中，q_0是初态；z_0是栈的初始内容；ε为空串。若u=ε，则称P停止于空栈；若q是某个终态，则称P停止于终态。
- 如果P满足下述两个条件，则称P是确定的：

① 对所有的状态q，输入串a和栈符号Z，$\delta(q, a, Z)$至多只包含一个元素。

② 若$\delta(q, \varepsilon, Z)$非空，则不存在符号a($a \neq \varepsilon$)使得$\delta(q, a, Z)$非空，即对于某个状态和栈符号，在空移动和非窑移动之间不应存在冲突。

- 给定SDTS T，其中，T的基础源文法是LL(1)，我们可以描述一个PDT P的构造，使得这个P：①接收T的基础源文法中每一个串；②恰好打印出T中与每个这种串对应的翻译串。

典型的自上而下LL(1)识别器有两类移动：

① 应用移动。位于栈顶的非终结符A被符号串w所替换。

其中，A→w是文法中的一个产生式；利用栈顶非终结符A和下一输入符号去查看对应的LL(1)分析表，可使这种操作确定化。

② 匹配移动。栈顶的终结符号a与下一输入符号匹配。

经过该操作之后，去掉了栈顶符号并使读头前进到下一位置。匹配失败即说明输入串有语法错。

通过上面的分析，现定义PDTP的操作为：

① 在一应用移动中，非终结符A已位于栈顶；借助分析表，就知道选用产生式A→w来进行归约．现假定R中的翻译规则是A→w, y

其中，$w=a_0B_1a_1B_2a_2…B_ka_K$，而$y=b_0B_1b_1B_2…B_kb_k$。这里；B_i是非终结符，a_i是输人符号串(或空)，b_i是输出符号串(或空)，且k≥0(注意：这个翻译规则是简单的)。

假定输入和输出符号是可区分的，那么，在一次应用移动中，位于栈顶的A就由下面的复合串$b_0a_0B_1b_1a_1B_2…B_kb_ka_K$ 所替代，而b_0变成栈顶符号。

② 若栈顶符号是输出字母表Δ的一个元素。则从栈中逐出它，并将它作为输出符号打印出。

③ 若栈顶符号是输入字母表VT的一个元素，则它应与下一个输入符号匹配(否则为语法错)。从栈中逐出它，并使读头前进一位置(即扫描下的输的输入符号)。

例如，考虑简单的SDTS：S→1S2S, xSySz　S→0, w

- 注意：此时,不必用下标来区分S,因为这个SDTS是简单的。它的基本源文法显然是LL(1)。以一个输入串为例,假定输入串为1102020，可以定义与(8.1)对应的PDT如下：

① 若栈顶为S，并且

1） 如果输入符号为1，则从栈中逐出S，并将x1Sy2Sz下推进栈(x位于栈顶)；

2） 如果输人符号为0，则从栈中逐出S，并将w0下推进栈(w位于栈顶)。

② 若栈顶符号t∈{x, y, z, w}，则从栈中逐出t，并输出t。

③ 若栈顶符号t∈{0, 1, 2}，则让t继续与下一输入符号匹配。

- 在翻译过程中，可以根据情况选用上述规则直至该PDT停止或发现语法错误。

- 规则变换：

$A \to a_0 B_1 a_1 B_2 a_2 \ldots,$

$\quad b_0 B_1 b_1 B_2 b_2 \ldots$

$\Rightarrow A \to b_0 a_0 B_1 b_1 a_1 B_2 b_2 a_2 \ldots$

- 例如：

$S \to 1S2S,\ xSySz$

$S \to 0,\ w$

$\Rightarrow S \to x1Sy2Sz$

$\quad S \to w0$

	0	1	2	#
S	w0	x1Sy2Sz		

操作	栈	输出	输入串
	S#		1102020#
	x1Sy2Sz#		1102020#
	1Sy2Sz#	x	1102020#
	Sy2Sz#		102020#
	x1Sy2Szy2Sz#		102020#
	1Sy2Szy2Sz#	x	102020#
	Sy2Szy2Sz#		02020#
	w0y2Szy2Sz#		02020#
	0y2Szy2Sz#	w	02020#
	y2Szy2Sz#		2020#
	2Szy2Sz#	y	2020#
	Szy2Sz#		020#
	w0zy2Sz#		020#
	0zy2Sz#	w	020#
	zy2Sz#		20#
	2Sz#	zy	20#
	Sz#		0#
	w0z#		0#
	0z#	w	0#
	#	z	#

结论：

- 如果SDTS的基础源文法是二义性的，那么就不存在确定的PDT。
- 若SDTS的基础源文法是LL(1)，那么，它的PDT是确定的，而且对每一个可接收的输入串恰好存在一个翻译。
- 虽然我们只是讨论了符号串到符号串的翻译器，但如果在输出串中再添加一些语义操作，例如查填符号表、修改编译程序的变量等等，就可能构造一个实用的翻译程序。

8.3 简单后缀SDTS和自下而上翻译器

- 如果SDTS是简单的，且规则有如下形式
  $A \rightarrow a_0B_1a_1B_2a_2\ldots$，$B_1B_2\ldots w$
  则称SDTS是简单后缀的SDTS。
- 定理 8.2 对于基础文法为LR(k)简单后缀的SDTS，则存在一确定的LR(k) PDT，它接受输入语句并产生对应输出串。
- 在选择产生归约时，输出相应的输出终结符符号。
- 通过在LR(k)分析器的归约动作中加入下述操作，很容易将一个LR(k)分析器改造成一个后缀翻译器。
- 如果选定产生式i进行归约，则在执行归约操作之后，再打印出与产生式i相关的翻译成分中的终结符部分。

8.3.1 后缀翻译

- 有算术表达式，考虑其R部分由下述翻译规则组成的SDTS：

  E→E+T, E T ADD　　E→T, T　　T→T*F, T F MPY

  T→F, F　　F→A, A LOAD　　F→(E)，E

  A→Aa, Aa　　A→a, a

该SDTS显然是简单后缀的，表达式 aa*(aaa+a) 的翻译是

  aa LOAD aaa LOAD a LOAD ADD MPY

其中，操作符LOAD操作在它前面的标识符上。如果希望LOAD操作在它后面的标识符上，可用下面两条规则替代F→A，A LOAD：

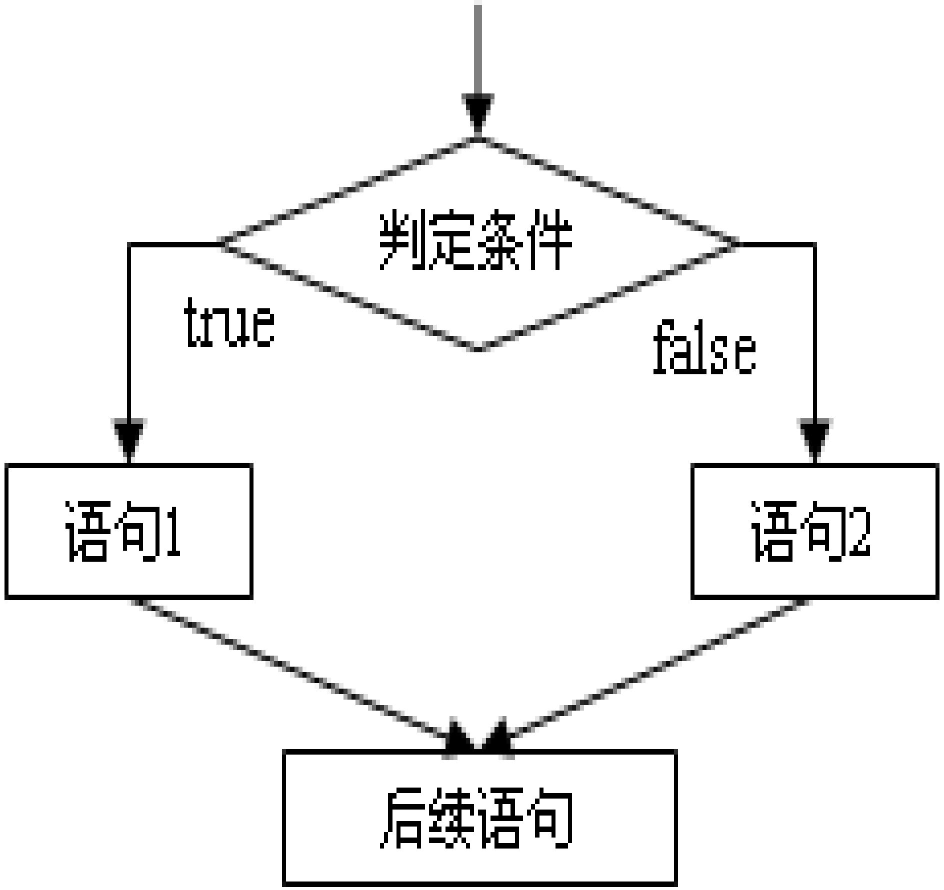
  F→L A，L A

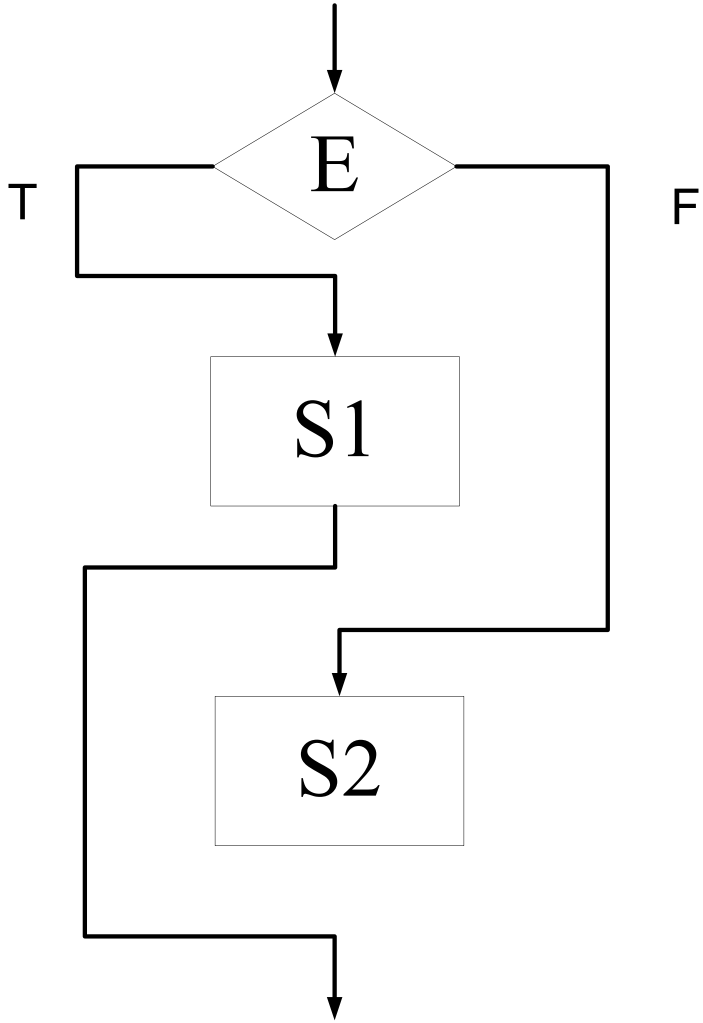
  L→ε，LOAD

这样修改后的文法仍然是LR(1)。产生式L→ε的归约操作可通过向前查看后面标识符的一个符号所确定。于是，表达式aa*(aaa+a)的翻译是

LOAD aa LOAD aaa LOAD a ADD MPY

S→if E then S1 else S2





E; FJP L1; S1;UJF L2;LOC L1; S2; LOC L2;

8.3.2 IF-THEN-ElSE控制语句(1)

- 考虑高级语言中的条件语句

  S→if E then S1 else S2

其中，E是表达式；S是一语句。

- 对于这种形式的输入串，简单后缀翻译器先生成关于E的计值代码，然后翻译那两个语句。
- 但这样一种语句要求在E和第一个S之间产生一个条件分支指令，在两个语句之间产生一个无条件分支指令。如何来产生这些指令呢?
- 解决办法之一是将原来的单—产生式拆成三个产生式：

  S→T else S2

  T→I then S1

  I→if E

这三个产生式合起来与原来的那个产生式等价。

8.3.2　IF-THEN-ElSE控制语句(2)

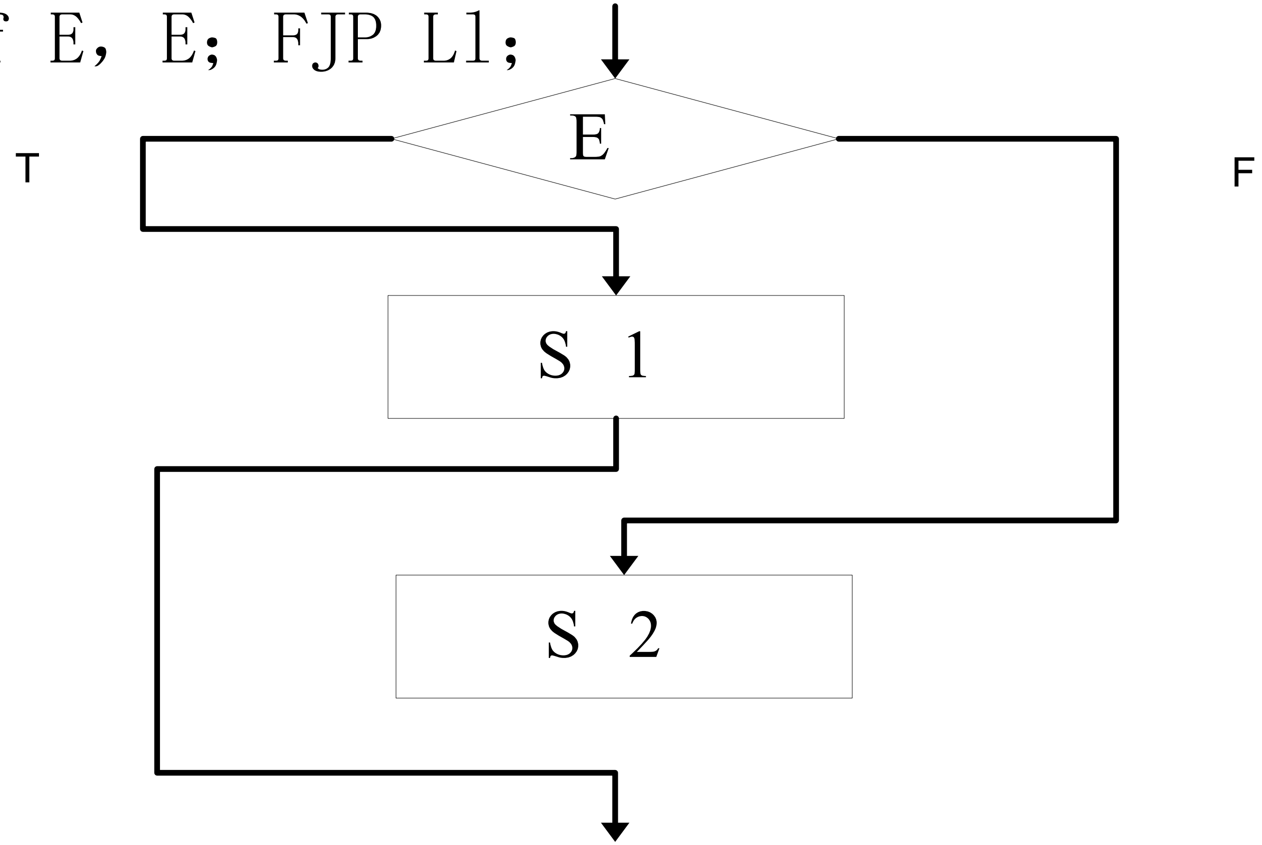
E；　FJP　L1；S1;UJF L2;LOC L1；S2；LOC L2;

- 经这样变形后，可以构造一个简单的后缀SDTS：

  S→T　else　S2，T　S2；　LOC L2;

  T→I　then　S1，I　S1；　UJP L2；LOC L1;

  I→if E，E；FJP L1；



8.3.2 IF-THEN-ElSE控制语句(3)

- 另一种解决办法是引进含“then′和“else″键字的产生式：

S→if E H S L S，E H S L S；LOC L2;

H→then，；FJP L1;

L→else，；UJP L2；LOC L1;

这些附加的产生式的作用在于：在分析该语句的过程中，能在适当的位置插入所需要的翻译串。

- 当然，也可以使用空产生式来实现同样的目的：

S→if E then H S else L S，E H S L S；LOC L2;

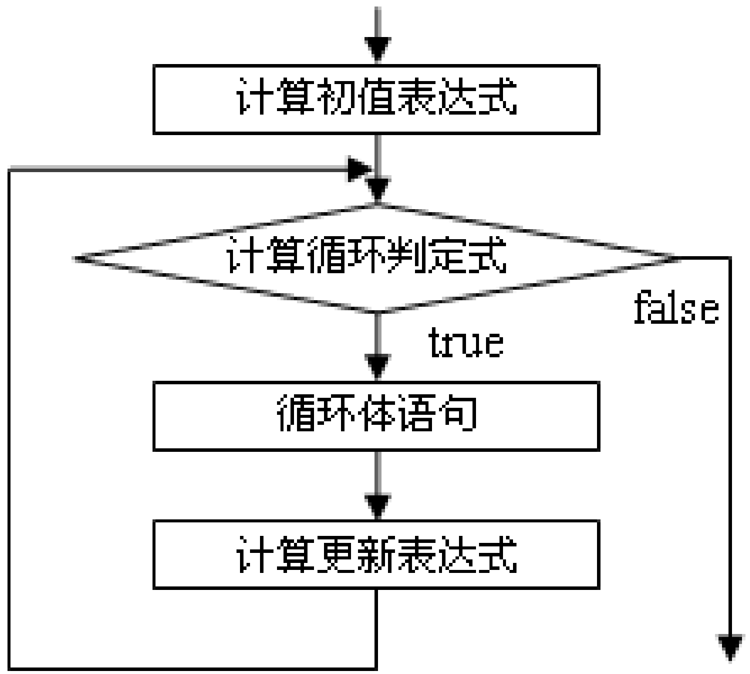
H→ε，；FJP L1;

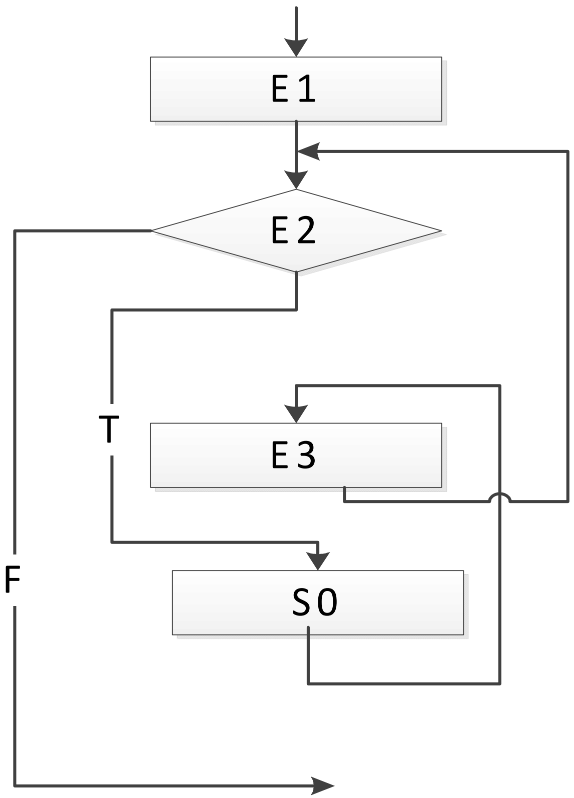
L→ε，；UJP L2；LOC L1;

- 这三种解决办法产生同样的效果。

for(初值表达式; 循环判定式; 更新表达式) 循环体语句;

for(E_1;E_2;E_3)S_0;





S→C S;,　C S; Ujp L2; LOC L4

C→B; E_3),　B; E_3 Ujp L1; LOC L3;

B→A;E_2 ,　A E_2 Fjp L4; Ujp L3 ;LOC L2;

A→for(E_1 ,　E_1; LOC L1;

E_1; LOC L1;

E_2; Fjp L4; Ujp L3 ;LOC L2;

E_3; Ujp L1; LOC L3;

S; Ujp L2; LOC L4

8.3.3 函数调用

- 考虑函数调用的情况，假定有关的基础源文法是

F→A(L)　　L→L；E　　L→E　A→Aa　　A→a

- 注意：这里实参表列由分号隔开的若干个E组成。
- 如果允许CALL和过程名都出现在实在参数代码之后，就需要一个非简单的翻译器。
- 下面的简单后缀的SDTS可用来产生过程的后缀调用形式：

F→A(L),A L;CALL　　L→L;E，L E

L→E,E　　　　　　　A→Aa,Aa　　　　A→a，a ；LOAD

那么，函数调用：aa(a;a+aaa)　将翻译成

aa；LOAD a；LOAD a；LOAD aaa；ADD；CALL

而且CALL将操作在第一个标识符aa上。

8.4　抽象语法树的构造(1)

- 抽象语法树AST(Abstract-Syntax Tree)是某个语言结构的一种简洁的树形表示形式，它只需包含该结构尚须转换或归约的信息。一般说来，任何语法结构(例如表达式、控制结构和说明)都可以用AST表示。
- AST也可作为一个多遍编译程序的中间语言结构。采用AST表示法有助于代码的产生和优化。

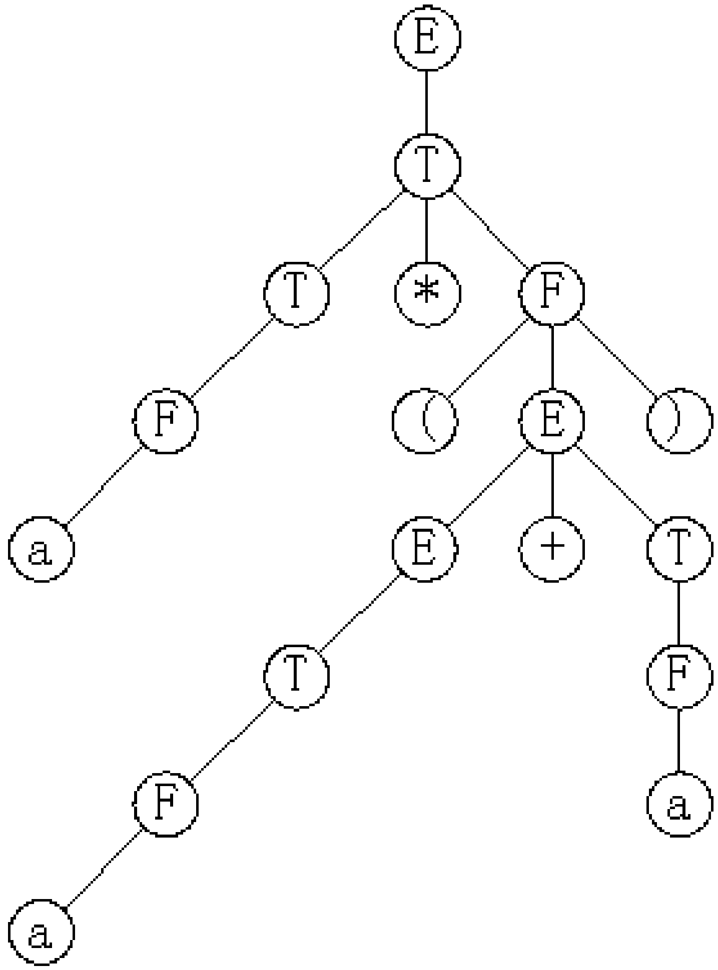
8.4 抽象语法树的构造(2)

- 考虑文法G_0：

  E→E+T　　　E→T

  T→T*F　　　T→F

  F→（E）　　F→a
- 利用该文法构造的每个简单表达式的语法树都比较大。
- 例如，简单表达式a*(a+a)的语法树如图8.4所示。这棵树有7个末结点和11个中间结点，但所给表达式本身却只有两个运算符和三个操作数．为什么差别如此之大呢?

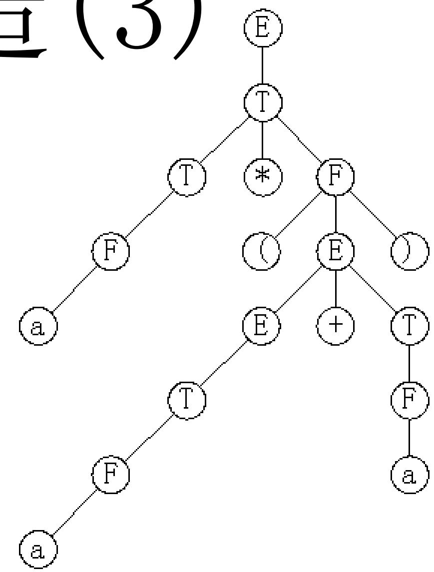


a*(a+a)的语法树

8.4　抽象语法树的构造(3)

可按如下方式将这棵语法树简化成一个AST。

- 首先，去掉与单非产生式,上提终结符结点。
- 第二，上提运算符并让它取代其父结点。
- 第三，去掉括号，并上提运算符。
- 最后得到的语法树便是一个AST。



a*(a+a)的语法树

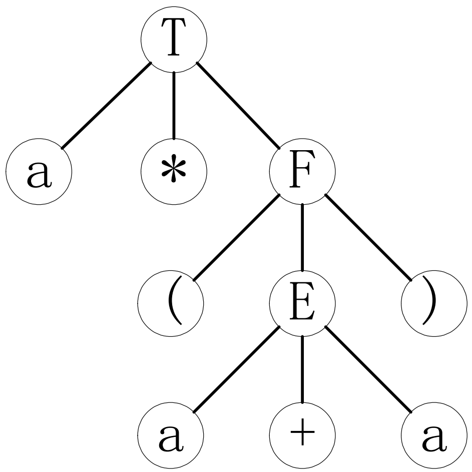


图8.5　图8.4去掉单非产生式

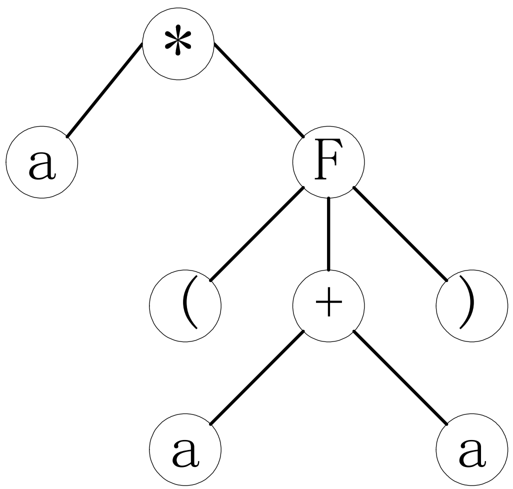


图8.6 运算符上提

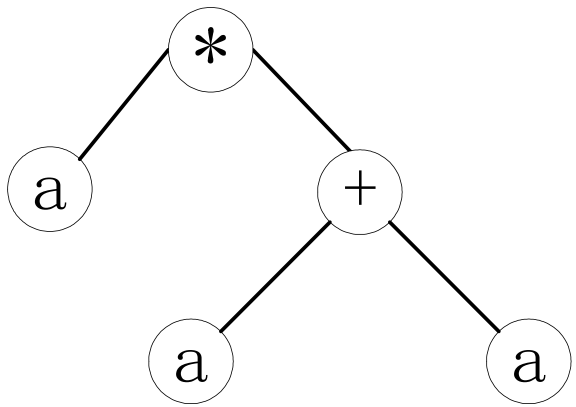


图8.7 去掉括号，运算符上提

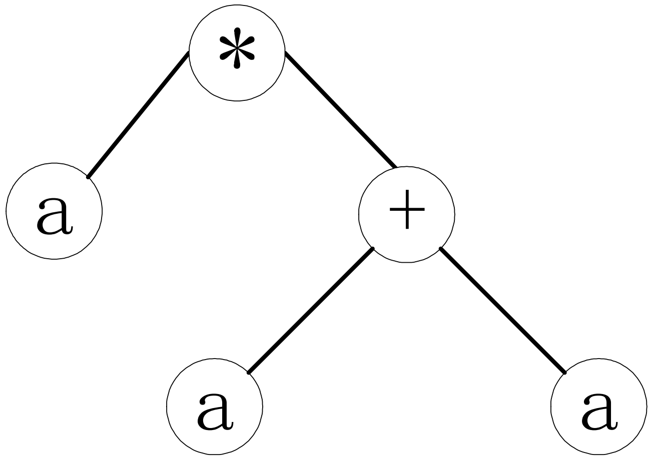


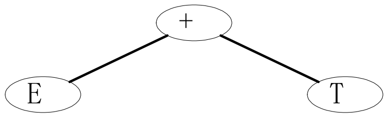
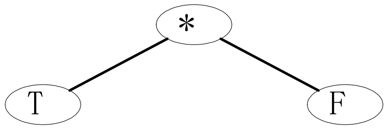
图8.7 去掉括号，运算符上提

- 这棵树也是下述表达式的抽象语法树：

  a*((a)+a)　　a*(a+(a))　　(a*(a+a))

- 注意：AST并不是一种规范形式，它并没有指明运算符的交换律和结合律．例如，a*b+c、c+b*a、c+a*b等表达式在数学上是等价的，但可生成不同的AST.
- 只要稍微改变一下SDTS的实现方式，就可不必生成一个完整的语法树而直接构造一个AST.

8.4.1 自下而上构造AST

表8.2　根据文法G_0及其分析器直接产生一个AST	
文法中的产生式	AST的成分
E→E+T	
E→T	T
T→T*F	
T→F	F
F→(E)	E
F→a	a

- 对于一个自下而上分析器.
- 考虑产生式E→E+T，当E+T作为句柄出现在栈顶时，已存在两棵子树E和T，但并无以“+”为根的子树。这条规则是说：构造以“+”为根的一棵子树，并将E和T子树作为它的两个孩子(左孩子和右孩子)。于是，得到以“+”为根的一棵新子树，将它连到E以替代栈顶的E和T。
- 对于规则E→T，当T是句柄时，它本身已连在某棵子树上，将这棵子树连到E以替代栈顶上的T。
- 最后，规则F→(E)告诉我们将E子树连到F以取代栈顶的(E)，而F→a则指明将由终结符a构成的单一结点的树连到F并用F取代a。
- 当到达接收状态时，已构造好一棵AST，且它已连到位于栈顶的开始符号上。

8.4.2 AST的拓广

- AST的一般形式是：其中间结点是运算符，而它的叶子为简单变量或常数。事实上，运算符号可以是任何单目运算符、二目运算符或n目运算符。
- 例如，一个类似于X(e1,e2,…,en) 的数组元素可表示成图8.8所示形式的AST，其中，X是多维数组的名字，e1，e2，…，en为其下标。
- AST也可表示控制结构和说明(略)。

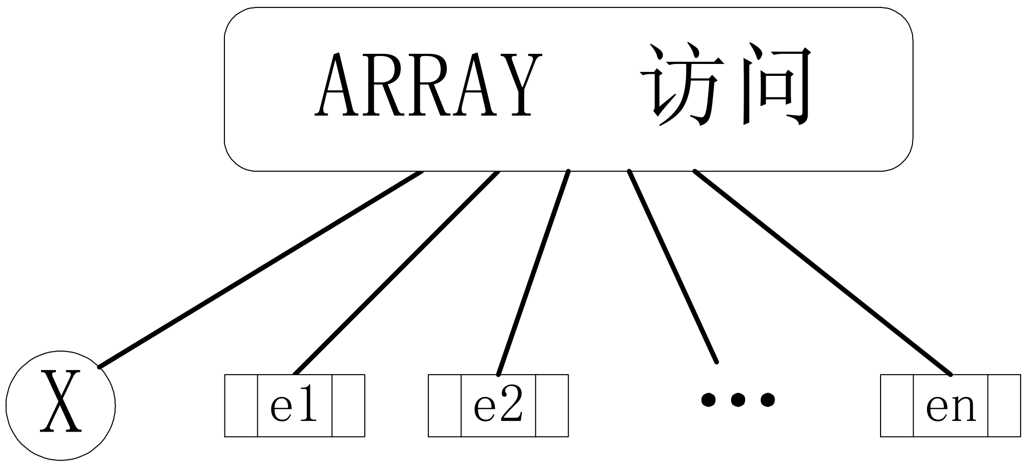


图8.8 数组访问(或过程调用)的AST

8.6 中间代码形式

- 在讨论属性翻译文法的具体用法之前，先介绍编译程序中的中间代码形式.

- 所谓中间代码形式是指用来表述源程序并与之等效的一种编码方式，可根据具体情况将它设计成各种形式。
- 例如，汇编语言程序就可看做一种中间代码形式。一般说来，对于一个多遍扫描的编译程序，越到后面阶段，所产生的中间代码也就越接近于机器代码，
- 于是，编译程序首先将源程序翻译成中间代码形式，然后，再把中间代码翻译成目标代码，也就是把语义分析和代码生成分开处理.
- 比较常用的中间代码形式有逆波兰表示、四元式、三元式和树形表示等等. 下面，我们讨论这几种典型的中间代码形式。

8.6.1 逆波兰表示法（1）

- 逆波兰表示法是由波兰逻辑学家J. Lukasiewicz首先提出来的一种表示表达式的方法. 在这种表示法中，运算符直接跟在其运算量(操作数)的后面。因此，逆波兰表示法有时也称为后缀(post fix)表示法. 下面是一些逆波兰表示法的例子：

AB*	表示A*B
AB*C+	表示A*B+C
ABCD / +*	表示A*(B+C / D)
AB*CD*+	表示A*B+C*D

8.6.1　逆波兰表示法（2）

- 与习惯的中缀表示法相比，逆波兰表示法有两个明显的特点：
- ✓ 第一，不再有括号，而且既简明又确切地规定了运算的计算顺序。
- ✓ 第二，运算处理极为方便，只需从左至右扫描表达式中的符号.
  - 当遇到运算量时，就把它存到运算量栈中，
  - 当遇到双目(单目)运算符时，就取出最近存入运算量栈中的两个(一个)运算对象进行运算处理，并把计算的结果作为一个新的运算量存入运算量栈，再继续向右扫描表达式中的符号，直至整个表达式处理完毕.
- ➢ 只要遵守在运算量之后直接紧跟它们的运算符这样的规则，就能很容易地将逆波兰表示推广到其它非表达式结构。

8.6.2　逆波兰表示法的推广(1)

- 考虑赋值语句，其一般形式为　〈左部〉:=〈表达式〉
- ✓ 如果把“:=”看做一个进行赋值运算的双目运算符，那么，赋值语句的逆波兰表示式为　〈左部〉〈表达式〉:=
- ✓ 而多重赋值可表示为　〈左部〉〈赋值语句〉 :=
- ✓ 例如，赋值语句A:=B*C+D可写做　ABC*D+:=
- ✓ 注意：当处理到“:=”这个运算符时，在表达式赋值执行完毕后，必须把〈左部〉和〈表达式〉从栈中消除，
- ✓ 因为，此时不产生结果值。对于多重赋值的情形，则需要把〈表达式〉这个量保存下来，直到整个多重赋值语句处理完毕后才能退掉。
- ➢ 此外，因为要把〈表达式〉之值送到该〈左部〉中去，所以在栈中只需〈左部〉的地址而不需其值。

8.6.2 逆波兰表示法的推广(2)

- 下面，讨论如何用逆波兰表示法表示转向语句、条件语句和条件表达式。
- 为此，先引进几个运算：

✓ Jump 是一个单目运算符，“〈标号〉jump”表示无条件转到〈标号〉去，“〈序号〉jump”表示无条件转到〈序号〉去。

✓ jumpf是一个双目运算符，“〈布尔表达式〉〈序号i〉 jumpf” 表示当〈布尔表达式〉为假时，转移到〈序号i〉处；否则，按原顺序执行。

- 那么，转向语句的逆波兰表示就是〈标号〉jump;

8.6.2　逆波兰表示法的推广(3)

- 条件语句的逆波兰表示是

〈布尔表达式〉〈序号1〉jumpf〈语句1〉〈序号2〉jump〈语句2〉，其中,〈序号1〉指的是〈语句2〉开始符号的序号,〈序号2〉指的是紧接在〈语句2〉之后的那个符号的序号。

- 高级语言中的数组说明array $A[l_1, \ldots u_1, \ldots, l_n \ldots u_n]$ 可用 $l_1 u_1 \ldots l_n u_n$A ADEC来表示，其中，只有ADEC是运算符，它的运算对象的个数是可变的，由其下标的个数决定。类似地，下标变量A[〈表达式〉，…，〈表达式〉]可用〈表达式〉…〈表达式〉A SUBS表示。
- 高级语言中的其它成分也可用类似的表示法表示，不在此一一讨论。

一个用逆波兰表示法表示程序段的例子(例8.1)

```
begin integer k ;
  k:=100 ;
  h:
  if k>i+j then
      begin  k:=k-1 ;
       goto h
      end
   else
     k:=i*2-j*2 ;
   i:=j:=0
end
```

- 该程序段的逆波兰表示是

```
(1)  Block
(2)  k 100:=
(5)  h :
(7)  kij+>(23)jumpf
(14) kk1-:=h jump
       (32)jump
(23) ki2*j2*-:=
(32) ij0:=:=
(37) Blockend
```

if E then S1 else S2

block

E

<1> jumpf

S1

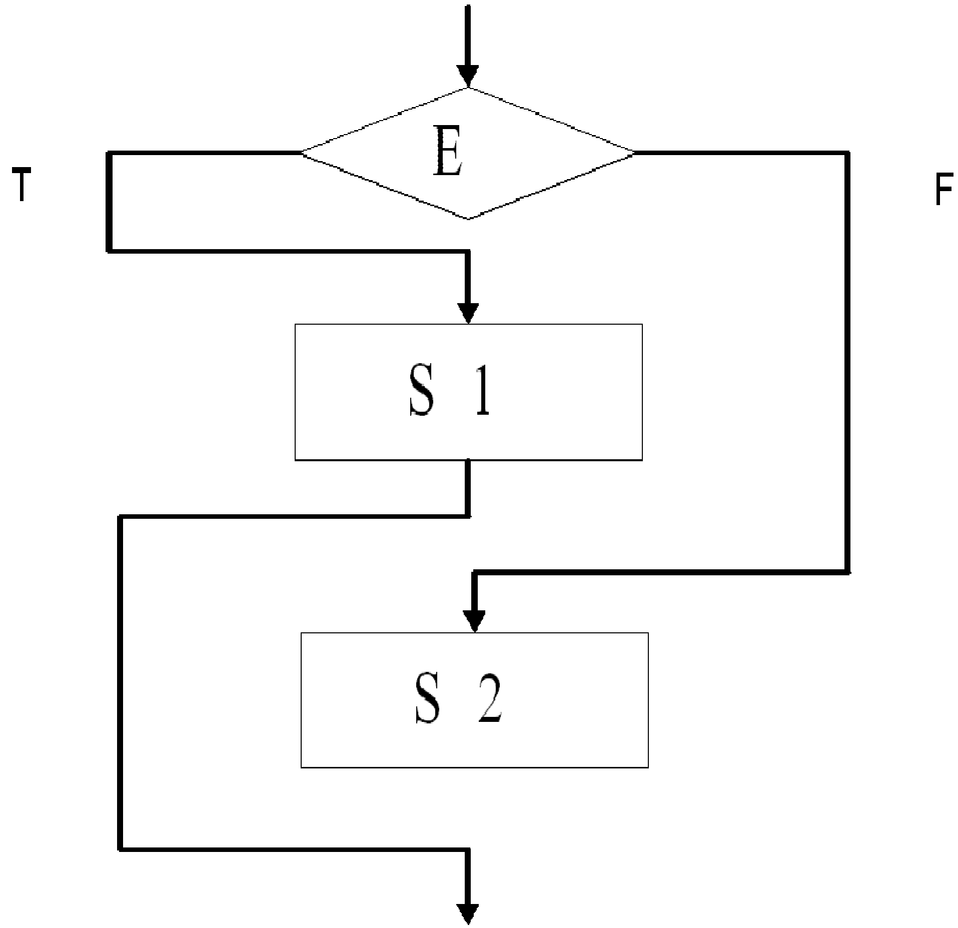
<2> jump

<1>

S2

<2>

blockend



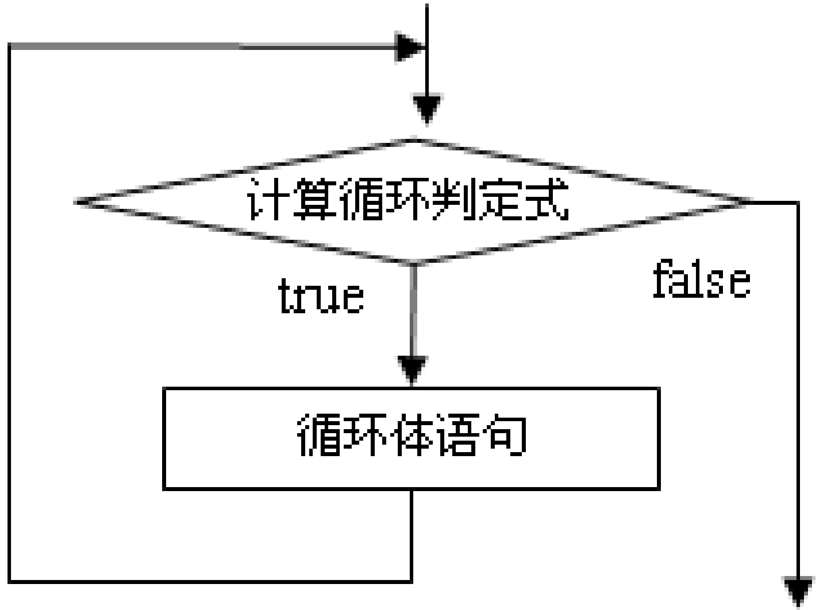
while E do S

<1> E <2>jumpf

S

<1> jump

<2>



$$sum = \sum_{n=1}^{100} n$$

$$\exp = \sum_{\frac{1}{n^2} < 10^{-5}} \frac{1}{n^2}$$

```
for  k:= k1  step  k3  until  k2  do  S
```

$$sum = \sum_{n=1}^{100} n$$

```
kk1:=
<1>
kk2-sign(k3)*0<<2> jumpf
S
kkk3+:=
<1> jump
<2>
```

$$\exp = \sum_{\frac{1}{n^2}<10^{-5}} \frac{1}{n^2}$$

8.6.3 四元式（1）

- 四元式的一般形式是

  〈运算符〉 〈运算量1〉 〈运算量2〉 〈结果〉

  当〈运算符〉为单目运算时，总认为〈运算量2〉为空，即此时施运算于〈运算对象1〉，运算结果放在〈结果〉中。

  例如，A×B的四元式是

  * A B T

➢ 这里T是暂存A*B之计算结果的临时变量按此方式。

- 表达式A*B+C*D的四元式为

*	A	B	T1
*	C	D	T2
+	T1	T2	T3

➢ 其中T1,T2,T3都是计算结果的临时变量。和逆波兰表示相仿，四元式也是按照表达式的执行顺序组合起来的。

8.6.3 四元式（2）

- -(A/B-C) 的四元式为

/	A	B	T1
-	T1	C	T2
⊖	T2	_	T3

➢ ⊖表示单目运算符-，对于无运算结果的运算符，其四元式中的<结果>处规定为空。

运算符	运算对象1	运算对象2	结果	含　　义
θ	p1		T	θp1 => T
ω	p1	p2	T	p1ωp2 => T
>	p1		T	>p1 => T
jump	A			无条件地转至地址(或序号)为A的四元式
jump	h			无条件地转至标号h所指的那个四元式
jumpf	A	B		当B为假时，转至地址(或序号)为A的四元式
:=	p1		p3	p1 => p3
Block				程序段的开始
Blockcnd				程序段的结束

例8.1中程序的四元式为

```
begin integer k ;
    k:=100 ;
h:  if k>i+j then
begin  k:=k-1 ;
      goto h
end  else  k:=i*2-j*2 ;
i:=j:=0
end
```

(1)	Block			
(2)	:=	100	_	k
(3)	+	i	j	T1
(4)	>	k	T1	T2
(5)	jumpf	(9)	T2	_
(6)	-	k	1	k
(7)	jump	(3)	_	_
(8)	jump	(12)	_	_
(9)	*	i	2	T3
(10)	*	j	2	T4
(11)	-	T3	T4	k
(12)	:=	0	_	j
(13)	:=	j	_	i
(14)	Blockend			

if E then S1 else S2

block

E

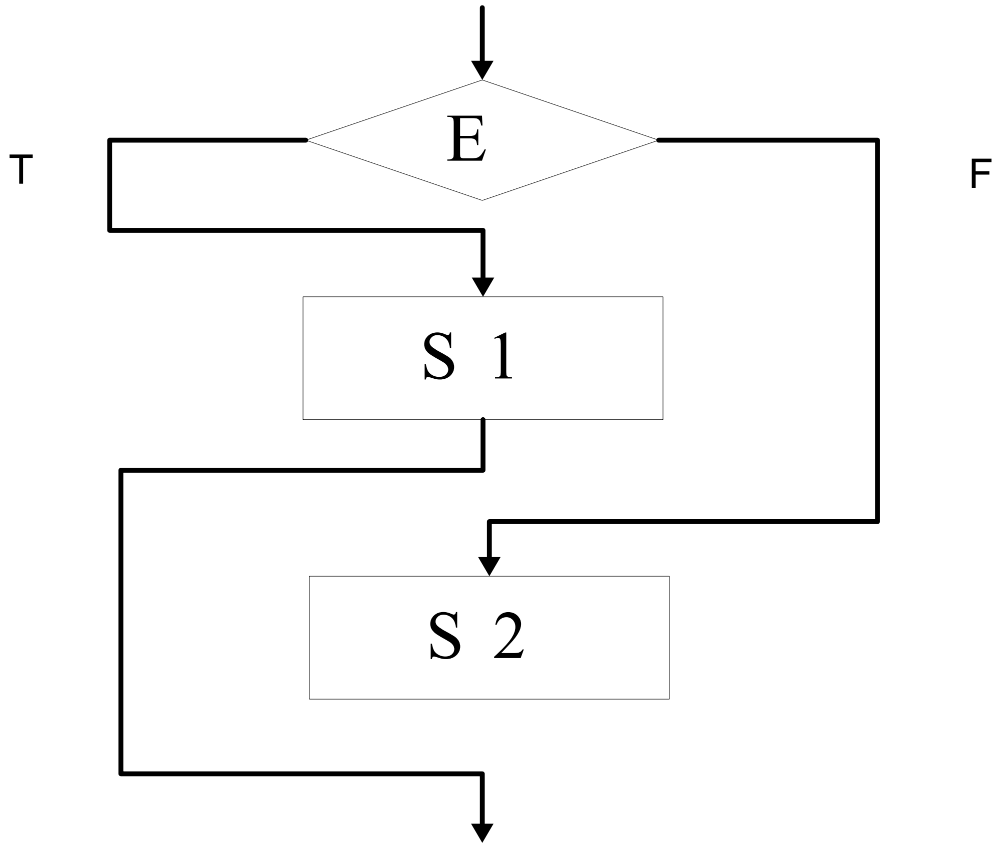
Jumpf (1) T -

S1

Jump (2) - -

(1) S2

(2)



while E do S

(1) E

jumpf (2) T -

S

jump (1) - -

(2)

$$sum = \sum_{n=1}^{100} n$$

$$\exp = \sum_{\frac{1}{n^2} < 10^{-5}} \frac{1}{n^2}$$

for k:= k1 step k3 until k2 do S

```
k:=k1;
<1>
(k-k2)*sign(k3)>0
jumpf <2>
S
k:=k+k3
jump<1>
<2>
```

$$sum = \sum_{n=1}^{100} n$$

$$\exp = \sum_{\frac{1}{n^2}<10^{-5}} \frac{1}{n^2}$$

8.6.4 三元式

- 三元式与四元式基本相同，所不同的只是没有表示运算结果的部分，凡涉及运算结果的，均用相应三元式的地址或序号来代替。

➢ 三元式的一般形式为 〈运算符〉 〈运算量1〉 〈运算量2〉

例如，A+B*C可表示为

① * B C

② + A ①

➢ 注意：在这种表示法中，①指的是第一个三元式的结果，而不是常数1，

而把1+B*C表示为

① * B C

② + 1 ①

➢ 当然，在具体处理中，必须把这种新的运算量(即它引用另一个三元式)和别的运算量加以区别，这可在运算对象的相应处附以某种标记来鉴别。

❑ 与四元式相比三元式有两个优点，一是它无须引进(四元式中的)那些临时变量，再就是它占用的存储空间比四元式少．但它也有不足之处，即当要实现代码优化时，通常需要从程序中删去某些运算：或者把另外一运算移到程序中的不同地方。采用四元式时，这项工作是容易完成的；但若采用三元式，由于三元式相互引用太多，所以，这项工作较难完成。

例8.1中程序的三元式为

```
begin integer k；
    k:=100；
h:  if k>i+j then
begin  k:=k-1；
      goto h
end  else  k:=i*2-j*2；
j:=j:=0
end
```

if E then S1 else S2

block

E <1> jumpf

S1

<2> jump

<1>S2

<2>

while E do S

<1> E <2>jumpf

S

<1> jump

<2>

$$sum = \sum_{n=1}^{100} n$$

$$\exp = \sum_{\frac{1}{n^2} < 10^{-5}} \frac{1}{n^2}$$

```
for  k:= k1 step  k3  until  k2  do  S

k:=k1;
<1>
kk2-sign(k3)*<2> jumpf
S
kk3+:=
<1> jump
<2>
```

$$sum = \sum_{n=1}^{100} n$$

$$\exp = \sum_{\frac{1}{n^2} < 10^{-5}} \frac{1}{n^2}$$