第五章：中间代码生成(2)

1. 表达式的中间代码生成

- 表达式的中间代码就是正确计算表达式值的四元式序列。
- 表达式中的变量可以是简单、复杂变量（数组变量，结构体成员变量等），还可以是函数调用，而表达式中的运算符也可以包括算术运算符，布尔运算符等。
- 本小节将给出简单算术表达式的中间代码生成的LL(1)语法制导方法。

基于LL(1)方法

【1】E→T
【2】E→E+T
【3】E→E-T
【4】T→P
【5】T→T*P
【6】T→T/P
【7】P→(E)
【8】P→id
【9】P→C

（1）E →T Es
（2）Es →ε
（3）Es →+T Es
（4）Es →-T Es
（5）T → P Ts
（6）Ts→ε
（7）Ts →*P Ts
（8）Ts →/P Ts
（9） P →C
（10）P →id
（11）P →(E)

- 简单表达式的带有动作符的LL(1)文法

(1) E →T Es
(2) Es →ε
(3) Es →+T#GenCode(+)#Es
(4) Es →-T#GenCode(-)#Es
(5) T → P Ts
(6) Ts→ε
(7) Ts →*P#GenCode(*)#Ts
(8) Ts →/P#GenCode(/)#Ts
(9) P →C#Push(C)#
(10) P →id#Push(id)#
(11) P →(E)

- 当遇到常量C和简单变量id时，把它们的语义信息压入语义栈；
- 当处理完一个运算符（+，-，*，/）的右分量时，该运算符的左、右运算分量已经分别存放在语义栈 Sem 的次栈顶和栈顶的位置，因此可以生成相应的运算符的四元式，并把运算结果的语义信息压入语义栈。

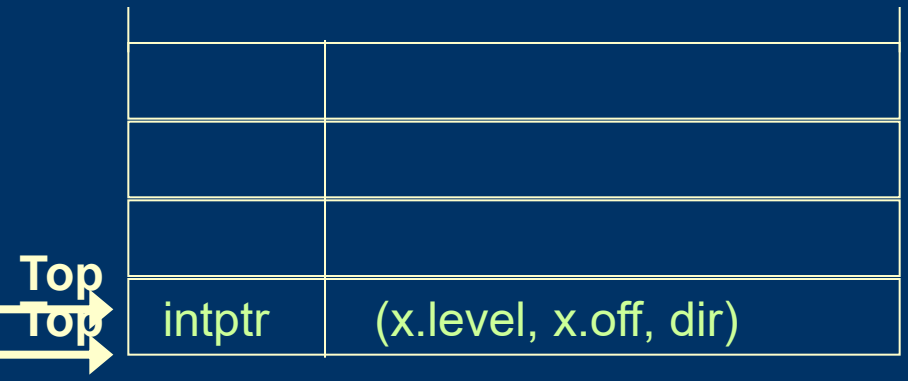
简单表达式的LL(1)分析表

	C	id	(	)	+	−	*	/	#
E	1	1	1						
Es				2	3	4			2
T	5	5	5						
Ts				6	6	6	7	8	6
P	9	10	11						

产生式	Predict集
(1) E → T Es	C, id, (
(2) Es → ε	#,)
(3) Es → + T Es	+
(4) Es → - T Es	−
(5) T → P Ts	C, id, (
(6) Ts→ε	#,),+,-
(7) Ts → * P Ts	*
(8) Ts → / P Ts	/
(9) P → C	C
(10) P → id	Id
(11) P → (E)	(

表达式 x + y * z 的中间代码生成过程(1)

分析栈S	输入流T	动作
# E	x + y * z #	LL[E ,id] = [1]
# EsT	x + y * z #	LL [T ,id] = [5]
# EsTs P	x + y * z #	LL [P ,id] = [10]
# EsTs #Push (id)# id	x + y * z #	Match
# EsTs #Push (id)#	+ y * z #	#Push (id)#
# EsTs	+ y * z #	

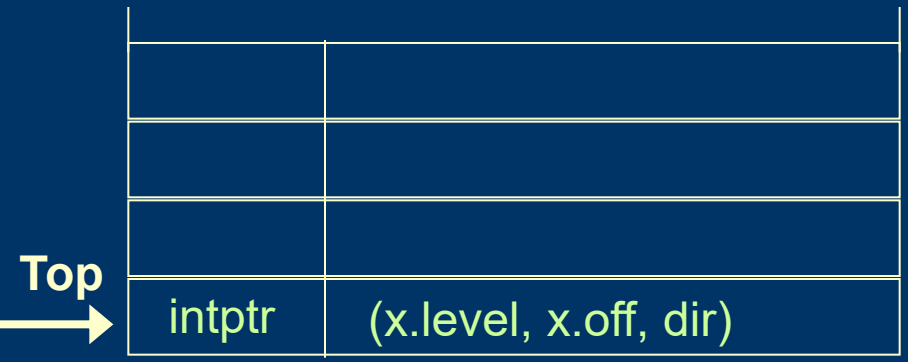


语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(2)

分析栈S	输入流T	动作
# EsTs	+ y * z #	LL [Ts,+] = [6]
#Es	+ y * z #	LL [Es,+] = [3]
# Es #GenCode (+)# T +	+ y * z #	Match
# Es #GenCode (+)# T	y * z #	LL [T,id] =[5]
# Es #GenCode (+) #TsP	y * z #	LL [P,id] = [10]
# Es #GenCode (+)# Ts#Push (id)# id	y * z #	

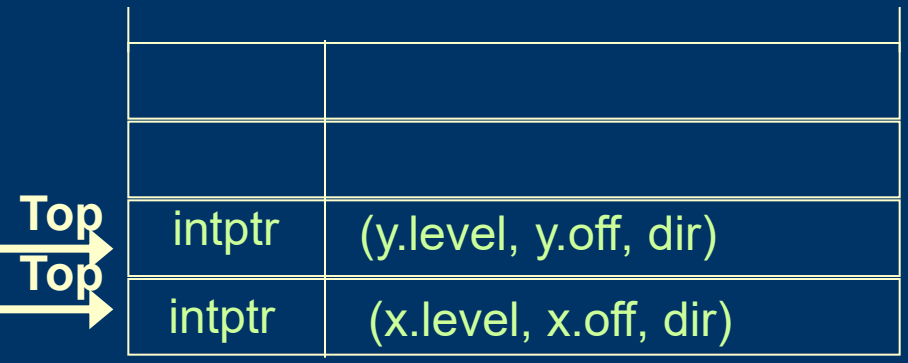


语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(3)

分析栈S	输入流T	动作
# Es# GenCode (+) # Ts # Push (id) # id	y * z #	Match
# Es# GenCode (+) # Ts # Push (id) #	* z #	# Push (id) #
# Es # GenCode (+) # Ts	* z #	

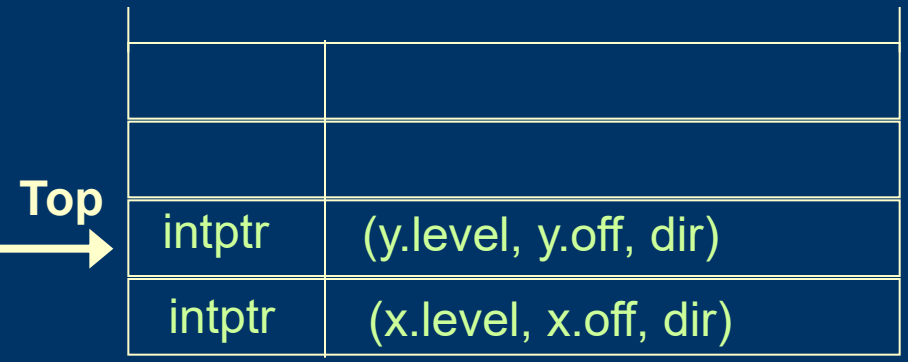


语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(4)

分析栈S	输入流T	动作
# Es # GenCode (+)# Ts	* z #	LL [Ts,*] = [7]
# Es # GenCode (+) # Ts # GenCode (*) # P*	* z #	Match
# Es # GenCode (+) # Ts # GenCode (*) # P	z #	

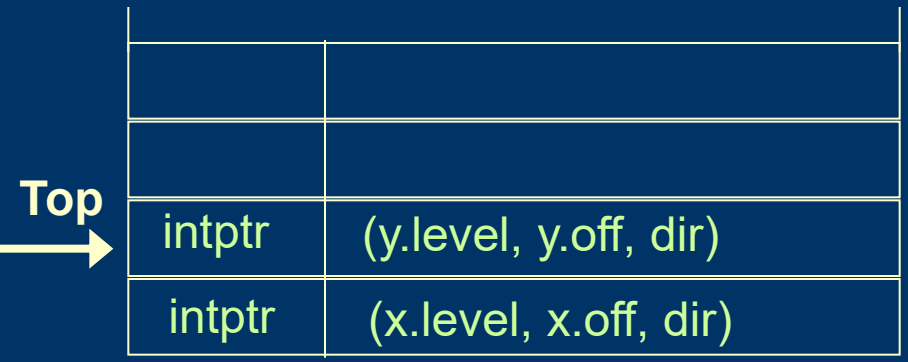


语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(5)

分析栈S	输入流T	动作
# Es # GenCode (+) # Ts # GenCode (*) # P	z #	LL[P,id] = [10]
# Es # GenCode (+) # Ts # GenCode (*) # # Push (id) # id	z #	Match
# Es # GenCode (+) # Ts # GenCode (*) # # Push (id) #	#	



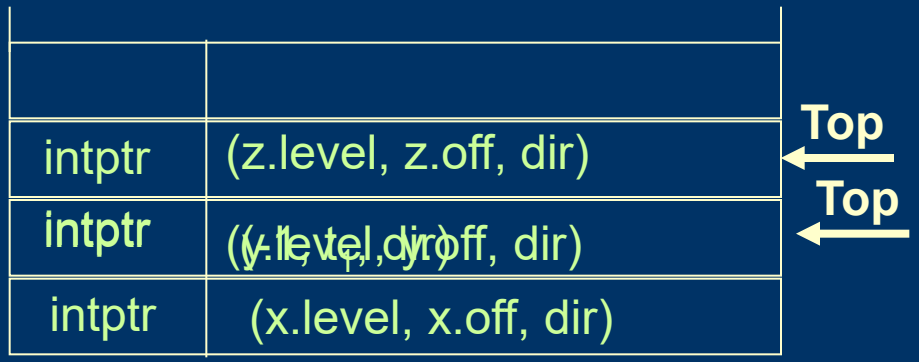
语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(6)

分析栈S	输入流T	动作
# Es # GenCode (+) # Ts # GenCode (*) # # Push (id) #	#	#Push (id)#
# Es # GenCode (+) # Ts # GenCode (*) #	#	#GenCode (*)#
# Es # GenCode (+) # Ts	#	

(MULTI, (y.level, y.off, dir), (z.level, z.off, dir), (-1, t_1, dir))



（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

表达式 x + y * z 的中间代码生成过程(7)

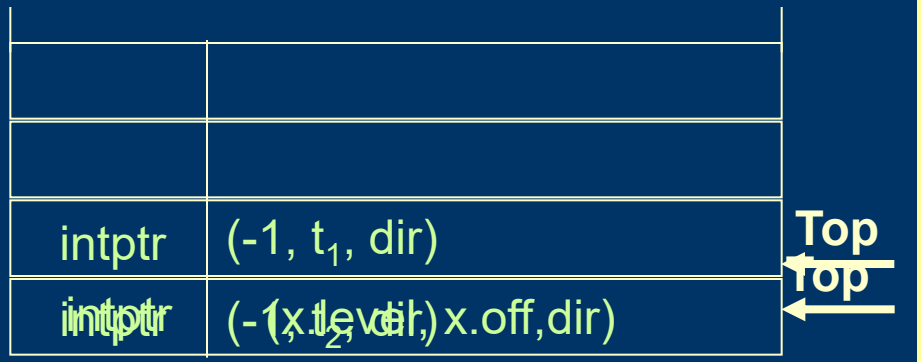
分析栈S	输入流T	动作
# Es # GenCode (+) # Ts	#	LL [Ts,#] = [6]
# Es # GenCode (+) #	#	#GenCode (+)#
# Es	#	LL [Es,#] = [2]
#	#	Success

(MULTI, y ,z, t_1)

(ADDI, x, t_1. t_2)

(MULTI, (y.level, y.off, dir*), (z.level, z.off,* dir*), (-1, t_1,* dir*))*

(ADDI, (x.level, x.off, dir), (-1, t_1, dir*) , (-1, t_2,* dir*))*



语义栈Sem

（1）E → T Es
（2）Es →ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → id # Push (id) #
（11）P → (E)

例：将下列表达式翻译成四元式序列。

X*2+A*(i+1)/(j+1)

其中i和j为整型变量，其它为实型变量。

1. (FLOAT, 2, _ , t_1)
2. (MULTF, X , t_1, t_2)
3. (ADDI, i, 1, t_3)
4. (FLOAT, t_3, _ , t_4)
5. (MULTF,A, t_4, t_5)
6. (ADDI, j, 1, t_6)
7. (FLOAT, t_6, _ , t_7)
8. (DIVF,t_5, t_7, t_8)
9. (ADDF, t_2, t_8, t_9)

2. 下标变量的中间代码生成

2.1 下标变量的地址计算

假设有数组类型的变量声明如下：

A :array $[L_1..U_1]$ $[L_2 .. U_2]$…

$[L_n ..U_n]$ of T ; $(n \geq 1)$

- 数组A占单元大小：size(A)=D1*D2*...*Dn*α

其中$D_i=U_i-L_i+1$，α为类型T所占单元数。

- 下标变量取址：$a[i_1][i_2][i_3]...[i_n]$

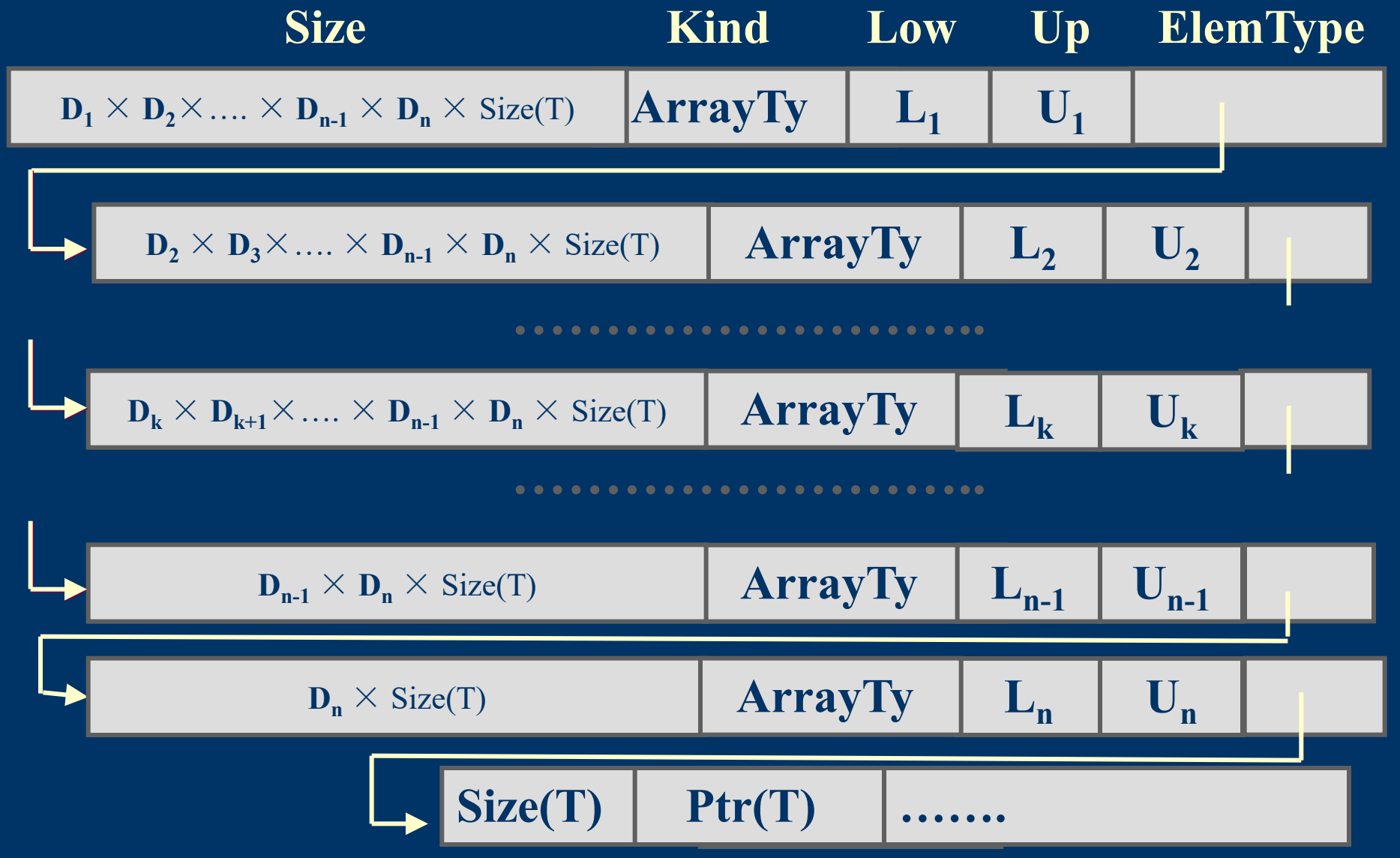
$=addr(a)+(i_1-L_1)*S_1+(i_2-L_2)*S_2+...+(i_n-L_n)*S_n$

//注：此处的S_i不同于书中P130给出的S_i，为第i层数组变量元素空间大小

假设有数组类型的变量声明如下：

A :array $[L_1..U_1]$ $[L_2 .. U_2]\cdots[L_n ..U_n]$ of T ; $(n \geq 1)$

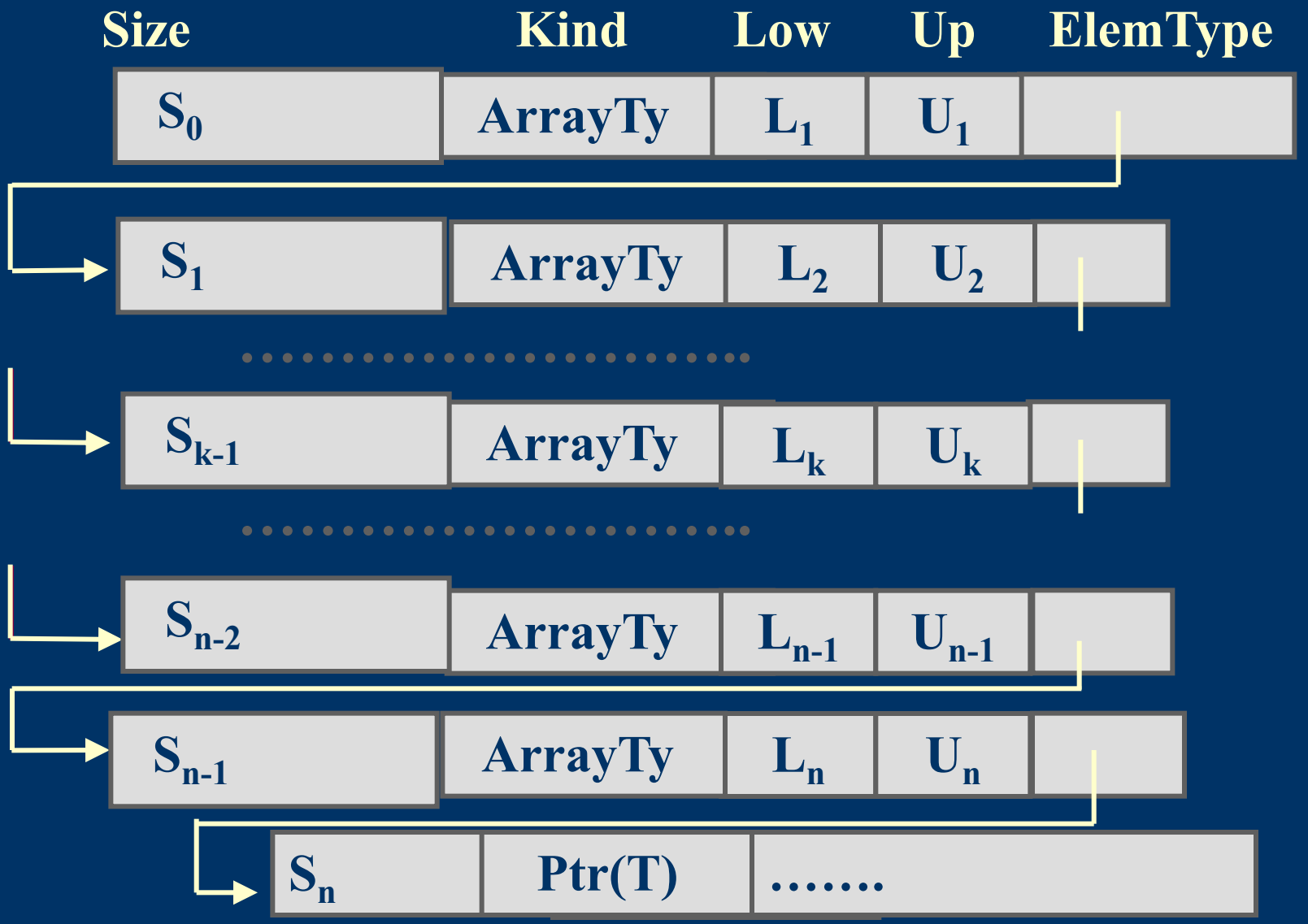
数组类型的内部表示　其中$D_i=U_i-L_i+1$

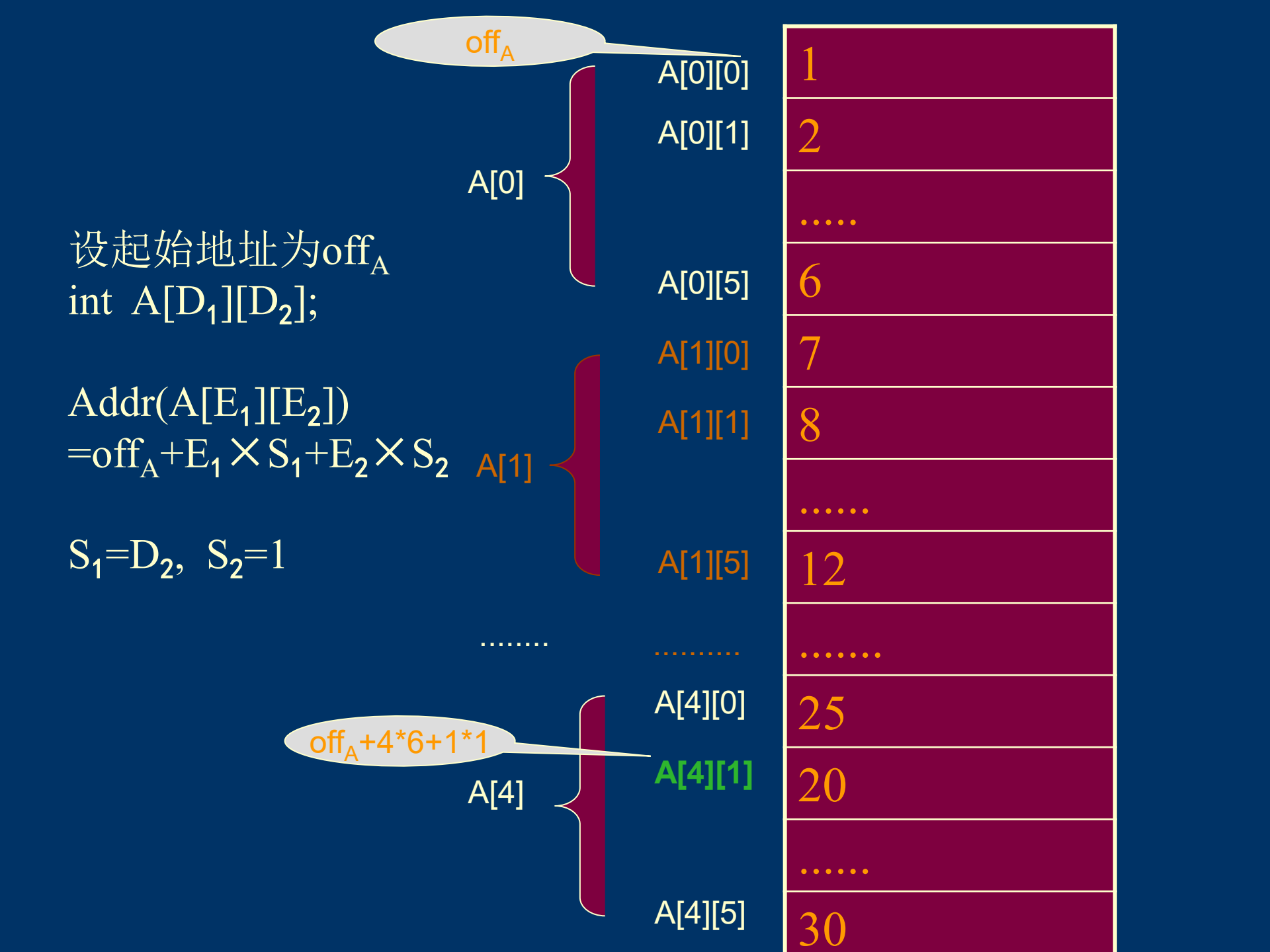


假设有数组类型的变量声明如下：

A :array [L_1..U_1] [L_2 .. U_2]…[L_n ..U_n] of T ；（n ≥ 1）

将size转化为S则有：




设起始地址为off_A
int A[D_1][D_2];
Addr(A[E_1][E_2])
=off_A+E_1×S_1+E_2×S_2
S_1=D_2, S_2=1
off_A
off_A+4*6+1*1
A[0]
A[1]
A[4]
A[0][0]
A[0][1]
A[0][5]
A[1][0]
A[1][1]
A[1][5]
A[4][0]
A[4][1]
A[4][5]
1
2
6
7
8
12
25
20
30

2.2下标变量的四元式结构

- 对于任意下标变量A[E_1] [E_2] …… [E_k] （n ≥ k），下面给出一种直接的中间代码结构：

E_1→ t_1

- (SUBI, t_1 , L_1, t_2)
- (MULTI , t_2 , S_1 , t_3)
- (AADD , A , t_3 , t_4)

A[E_1]地址

- $E_2 \rightarrow t_5$
- (SUBI, t_5 , L_2, t_6)
- (MULTI , t_6 , S_2 , t_7)
- (AADD , t_4 , t_7 , t_8)
- ……

A[E_1] [E_2]地址

- $E_k \rightarrow t_n$
- (SUBI, t_n , L_k, t_{n+1})
- (MULTI , t_{n+1} , S_k , t_{n+2})
- (AADD , $t_?$, t_{n+2} , t_{n+3})

A[E_1] [E_2] …… [E_k]地址

练习题：

```
i,j:integer;
a :  array[1..10][1..5] of  integer ;
```

下标变量a [i +1] [j*i-2]的 中间代码：

1. (ADDI，i，1，t_1)
2. (SUBI, t_1 ,1, t_2)
3. (MULTI , t_2 , 5 , t_3)
4. (AADD , a, t_3 ， t_4)
5. (MULI，j，i，t_5)
6. (SUBI，t_5，2，t_6)
7. (SUBI, t_6,1, t_7)
8. (MULTI , t_7 , 1 , t_8)
9. (AADD , t_4 , t_8 ， t_9)

2.3下标变量的中间代码生成

下标变量的LL(1)动作文法可以写成：

(1)V → id#PUSH(id)#A

(2)A → #Init#[E]#AddNext#B

(3)B →ε

(4)B → [E]#AddNext#B

其中:

- Init用来初始化下标计数器 k = 0;
- 遇到变量标识符，则调用PUSH(id)把其语义信息（一般为类型和地址）入栈中;
- 在下标表达式之后，调用AddNext，用于生成地址累加的四元式。它的算法是:

AddNext算法：

- 下标计数器加1， k := k+1；
- 从语义栈中取出下标表达式计算结果result；
- 生成计算第k维下标需要累加的地址偏移的四元式组：

  （ SUBI, result, L_k, t_{n+1}）

  （ MULTI , t_{n+1} , S_k , t_{n+2} ）
- 从语义栈中取出地址累加结果$t_?$；
- 生成地址累加四元式（ AADD , $t_?$, t_{n+2} , t_{n+3}）

  ① pop（2）；

  ②把累加后的地址结果t_{n+3}压入语义栈；

扩展表达式的带有动作符的LL(1)文法

（1） E → TEs
（2） Es → ε
（3） Es → +TEs
（4） Es → -TEs
（5） T → PTs
（6） Ts→ ε
（7） Ts → *PTs
（8） Ts → /PTs
（9） P → C
（10）P → V
（11）P → (E)
（12） V →idA
（13） A → ε
（14） A → [E]B
（15） B → ε
（16） B→ [E]B

（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

•简单算术表达式的LL(1)文法定义

（1）E → TEs
（2）Es → ε
（3）Es → +TEs
（4）Es → -TEs
（5）T → PTs
（6）Ts→ ε
（7）Ts → *PTs
（8）Ts → /PTs
（9）P → C
（10）P → V
（11）P → (E)
（12） V →idA
（13） A → ε
（14） A → [E]B
（15） B → ε
（16） B→ [E]B

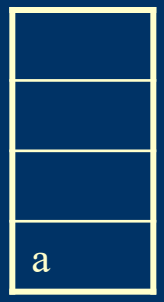
	First集	Follow集
E	C, id , (	# ,) ,]
E s	ε, +, -	# ,),]
T	C, id , (	+,- ,# ,),]
Ts	ε, *, /	+,-,# ,),]
P	C, id , (	*,/ ,+,-,# ,),]
V	id	*,/ ,+,-,# ,),]
A	ε,[	*,/ ,+,-,# ,),]
B	ε,[	*,/,+,-,# ,),]

•简单表达式的带有动作符的LL(1)分析表

	C	id	(	)	+	−	*	/	[	]	#
E	1	1	1								
Es				2	3	4				2	2
T	5	5	5								
Ts				6	6	6	7	8		6	6
P	9	10	11								
V		12									
A				13	13	13	13	13	14	13	13
B				15	15	15	15	15	16	15	15

产生式	Predict集
(1) E → T Es	C, id , (
(2) Es → ε	#,),]
(3) Es → + T Es	+
(4) Es → - T Es	−
(5) T → P Ts	C, id , (
(6) Ts→ε	#,),+,-,]
(7) Ts → * P Ts	*
(8) Ts → / P Ts	/
(9) P → C	C
(10) P →V	id
(11) P → (E)	(
(12) V →idA	id
(13) A → ε	#,),+,-,*,/,]
(14) A → [E]B	[
(15) B → ε	#,),+,-,*,/,]
(16) B→ [E]B	[

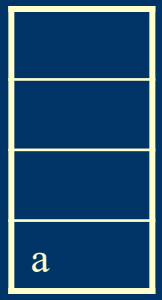
分析栈S	输入流T	动作
#E	a[i][j]#	LL[E,id]=1
#E_ST	a[i][j]#	LL[T,id]=5
#$E_S$$T_S$P	a[i][j]#	LL[P,id]=10
#$E_S$$T_S$V	a[i][j]#	LL[V,id]=12
#$E_S$$T_S$A#PUSH(id)#id	a[i][j]#	MATCH
#$E_S$$T_S$A#PUSH(id)#	[i][j]#	PUSH(id)
#$E_S$$T_S$A	[i][j]#	LL[A,[ ]=14



（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

分析栈S	输入流T	动作
#$E_S T_S$A	[i][j]#	LL[A,[ ]=14
#$E_S T_S$ B #AddNext#] E [#init#	[i][j]#	init
#$E_S T_S$B #AddNext#] E[	[i][j]#	match
#$E_S T_S$B #AddNext#]E	i][j]#	LL[E,id]=1
#$E_S T_S$B #AddNext#]EsT	i][j]#	LL[T,id]=5
#$E_S T_S$B #AddNext#]EsT_SP	i][j]#	LL[P,id]=10
#$E_S T_S$B #AddNext#]EsT_SV	i][j]#	LL[V,id]=12
#$E_S T_S$B #AddNext#]EsT_SA#PUSH(id)#id	i][j]#	match

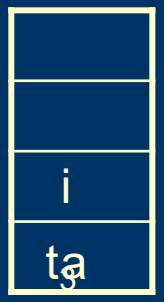
k=0



（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

分析栈S	输入流T	动作
#$E_S T_S B$ #AddNext#]$Es T_S A$#PUSH(id)#id	i][j]#	match
#$E_S T_S B$ #AddNext#]$Es T_S A$#PUSH(id)#	][j]#	PUSH(id)
#$E_S T_S B$ #AddNext#]$Es T_S A$	][j]#	LL[A,]]=13
#$E_S T_S B$ #AddNext#] $Es T_S$	][j]#	LL[T_S ,]]=6
#$E_S T_S B$ #AddNext#]Es	][j]#	LL[E_S ,]]=2
#$E_S T_S B$ #AddNext#]	][j]#	match
#$E_S T_S B$ #AddNext#	[j]#	AddNext

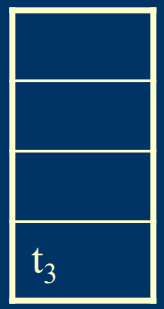
(SUBI, i, 1,t_1)
(MULTI,t_1,5, t_2)
(AADD,a,t_2, t_3)
Pop(2)
Push(t_3)



（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

分析栈S	输入流T	动作
#$E_S T_S B$ #AddNext#	[j]#	AddNext
#$E_S T_S B$	[j]#	LL[B,[ ]=16
#$E_S T_S B$ #AddNext#] E[	[j]#	match
#$E_S T_S B$ #AddNext#] E	j]#	LL[E,id]=1
#$E_S T_S B$ #AddNext#] EsT	j]#	LL[T,id]=5
#$E_S T_S B$ #AddNext#] EsT_SP	j]#	LL[P,id]=10
#$E_S T_S B$ #AddNext#] EsT_SV	j]#	LL[V,id]=12

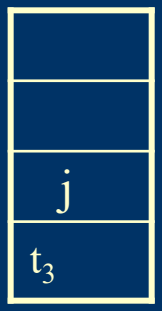
(SUBI, i, 1,t_1)
(MULTI,t_1,5, t_2)
(AADD,a,t_2, t_3)
Pop(2)
Push(t_3)



（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

分析栈S	输入流T	动作
#$E_S T_S$B #AddNext#] EsT_SV	j]#	LL[V,id]=12
#$E_S T_S$B #AddNext#] EsT_SA#PUSH(id)#id	j]#	match
#$E_S T_S$B #AddNext#] EsT_SA#PUSH(id)#	]#	PUSH(id)
#$E_S T_S$B #AddNext#] EsT_SA	]#	LL[A,]]=13
#$E_S T_S$B #AddNext#] EsT_S	]#	LL[T_S ,]]=6
#$E_S T_S$B #AddNext#] Es	]#	LL[E_S ,]]=2
#$E_S T_S$B #AddNext#]	]#	LL[E_S ,]]=2

(SUBI, i, 1,t_1)
(MULTI,t_1,5, t_2)
(AADD,a,t_2, t_3)
Pop(2)
Push(t_3)

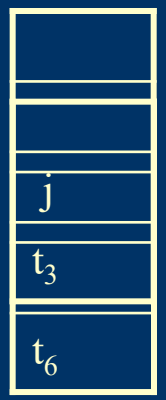


（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

分析栈S	输入流T	动作
#$E_S T_S B$ #AddNext#]	]#	match
#$E_S T_S B$ #AddNext#	#	AddNext
#$E_S T_S B$	#	LL[B,#]=15
#$E_S T_S$	#	LL[T_S ,#]=6
#E_S	#	LL[E_S ,#]=2
#	#	OK

(SUBI, i, 1,t_1)
(MULTI,t_1,5, t_2)
(AADD,a,t2, t_3)
Pop(2)
Push(t_3)

(SUBI, j, 1,t_4)
(MULTI,t_4,1, t_5)
(AADD,t_3,t_5, t_6)
Pop(2)
Push(t_6)



（1）E → T Es
（2）Es → ε
（3）Es → + T # GenCode (+) # Es
（4）Es → - T # GenCode (-) # Es
（5）T → P Ts
（6）Ts→ ε
（7）Ts → * P # GenCode (*) # Ts
（8）Ts → / P # GenCode (/) # Ts
（9）P → C # Push (C) #
（10）P → V
（11）P → (E)
（12） V →id # Push (id) # A
（13） A → ε
（14） A →#init# [E] #AddNext# B
（15） B → ε
（16） B→ [E] #AddNext# B

3. 赋值语句的中间代码

- 赋值语句的形式为：Left := Right,
- 赋值语句的四元式结构为：
  - Left 的中间代码
  - Right 的中间代码
  - (FLOAT ,Right , —,t)
  - (ASSIG ,Right(t),- ,Left)

3. 赋值语句的中间代码

- 赋值语句中间代码生成动作文法如下：

  S → V:= E #Assig#

- Assig需要做如下处理：

1. 从语义栈中取出赋值号左右分量的语义信息；
2. 比较类型是否相同，如果不同，则生成类型转换中间代码；
3. 生成赋值四元式：

   (ASSIG , Right(t), - , Left)

赋值语句的中间代码的例子

- **i,j,x,y:integer;**
- **a : array[1..10][1..5] of integer ;**

赋值语句**a [i] [j]:=x+y*3**的中间代码(其中变量均为整型）：

a [i] [j]

1. (SUBI ,i , 1 , t_1)
2. (MULTI , t_1 , 5 , t_2)
3. (AADD, a , t_2 , t_3)
4. (SUBI , j, 1 , t_4)
5. (MULTI, t_4 ,1 , t_5)
6. (AADD, t_3 , t_5 , t_6)

x+y*3

7.（ MULTI,y,3,t_7)
8. (ADDI,x, t_7,t_8)

9.(ASSIGN ,t_8, _ ,t_6)