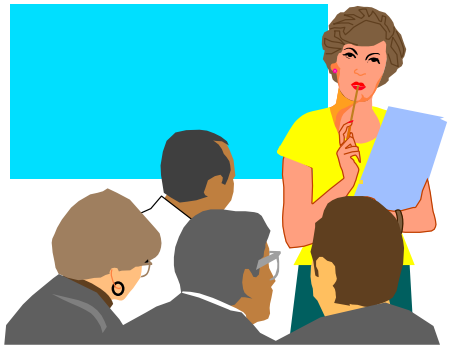
编译技术



胡春明
hucm@buaa.edu.cn

2019.9-2019.12

		一	二	三	四	五	六	日
第12周	上午	18	19上课	20	21上课	22	23	24
	下午		线上答疑					
第13周	上午	25	26上课	27	28上课	29	30	1
	下午		F332 上机答疑		不上机			
第14周	上午	2	3上课	4	5上课	6	7	8
	下午				不上机			
第15周	上午	9	10	11	12	13	14	15
	下午				不上机			
第16周	上午	16	17	18	19	20	21	22
	下午		上机		F332 上机答疑			
第17周	上午	23	24	25	26	27	28	29
	下午		上机考试					

编译过程是指将高级语言程序翻译为等价的目标程序的过程。

习惯上是将编译过程划分为5个基本阶段：

自顶向下（Top-Down）分析：推导（Derivations）

若 $Z \overset{+}{\underset{G[Z]}{\Rightarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

自底向上（Bottom-Up）分析：规约（Reductions）

若 $Z \overset{+}{\underset{G[Z]}{\Leftarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

自顶向下（Top-Down）分析：推导（Derivations）

若 $Z \overset{+}{\underset{G[Z]}{\Rightarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

（1）推导顺序：有多个“非终结符”，优先用哪个?

（2）避免二义性：避免文法有多个可用规则

? 主要问题:
- 左递归问题
- 回溯问题

▪ 主要方法:
- 递归子程序法
- LL分析法

自底向上（Bottom-Up）分析：

若 $Z \overset{+}{\underset{G[Z]}{\Leftarrow}} S$ 则 $S \in L(G[Z])$ 否则 $S \notin L(G[Z])$

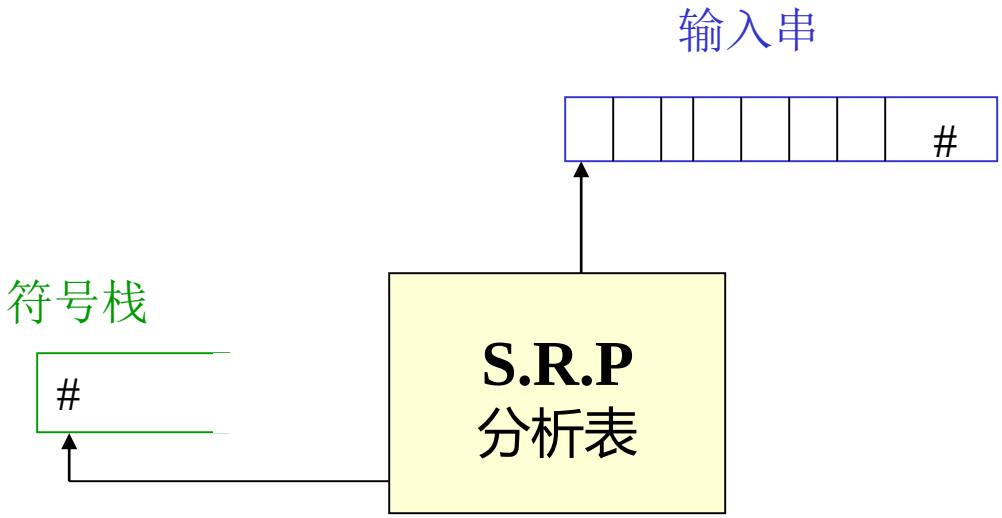
? 主要问题：
- 句柄的识别问题

▪ 主要方法：
- 算符优先分析法
- LR分析法

自底向上（移进-规约）分析方法

移进—归约分析（Shift-Reduce Parsing)

要点：建立符号栈，用来记录分析的历史和现状，并根据所面临的状态，确定下一步动作是移进还是归约。



算符优先分析

算符优先分析(Operator-Precedence Parsing)

1) 这是一种经典的**自底向上分析法**，简单直观，并被广泛使用，开始主要是对表达式的分析，现在已不限于此。可以用于一大类上下无关的文法。

2) 称为**算符优先分析**是因为这种方法是**仿效算术式的四则运算**而建立起来的，作算术式的四则运算时，为了保证计算结果和过程的唯一性，规定了一个统一的四则运算法则，规定运算符之间的优先关系。

运算法则:

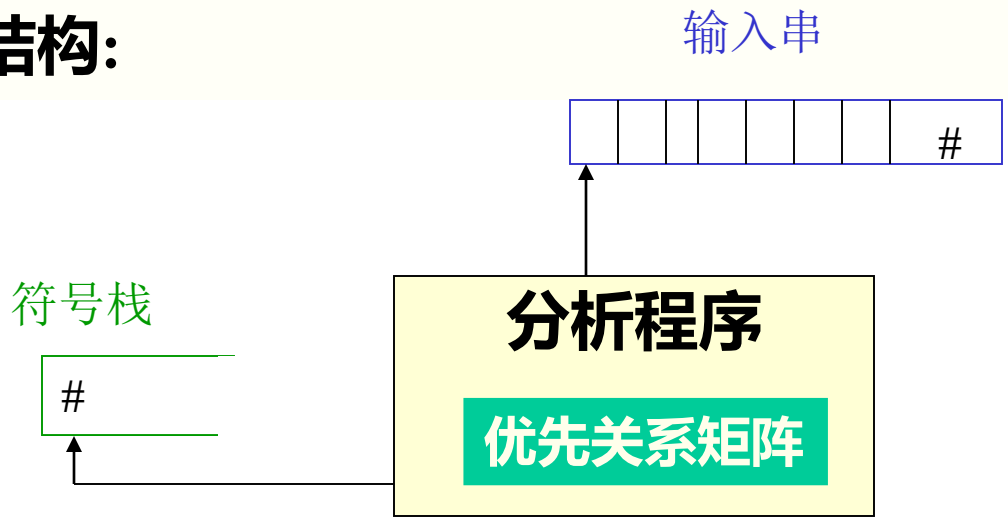
1.乘除的优先级大于加减
2.同优先级的运算符左大于右
3.括号内的优先级大于括号外

例如：4+8-6/2*3 运算过程和结果唯一

3) 算符优先分析的特点:

仿效四则运算过程，预先规定**相邻终结符**之间的优先关系，然后利用这种优先关系来确定句型的“**句柄**”，并进行归约。

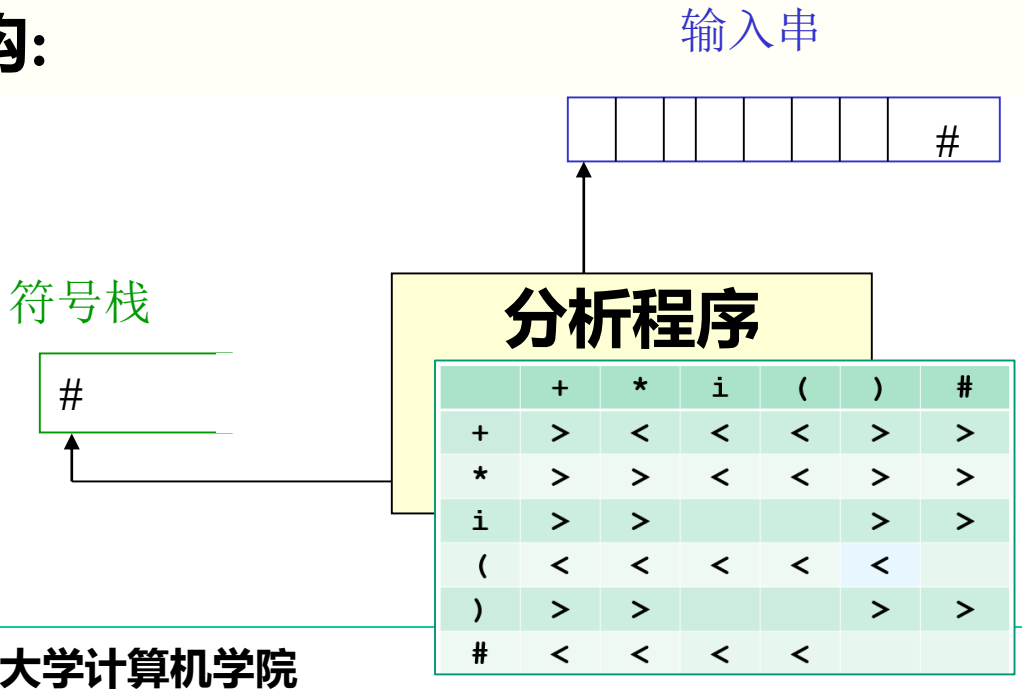
4) 分析器结构:



回顾：(1) 确定相邻终结符的相邻关系 （2） 确定优先级

产生分析表，利用这种优先关系来确定句型的“**句柄**”，
如果栈内优先级大于栈外，移进
如果栈内优先级小于栈外，规约

分析器结构:



	+	*	i	(	)	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(	<	<	<	<	<	
)	>	>			>	>
#	<	<	<	<		

问题

- 二义性?
- 从文法构造优先级的方法?
- 是否所有文法都可以用算符优先分析法呢?
- 如何从给定文法构造算符优先关系矩阵呢?

算符优先文法

（1）算符优先文法（OPG - Operator Precedence Grammar）

算符文法（OG）的定义

若文法中无形如U∷= …VW…的规则，这里V,W∈V_n
则称G为OG文法，也就是算符文法。

1) a=b iff 文法中有形如 U∷= …ab…或U∷= …aVb…的规则。

2) a<b iff 文法中有形如 U∷=…aW…的规则，其中 W⇒ b…或W⇒ Vb… 。

3) a>b iff 文法中有形如 U∷=…Wb…的规则，其中 W⇒ …a或W⇒ …aV 。

算符优先文法（OPG）的定义

设有一OG文法，如果在任意两个终结符之间，至多只有上述关系中的一种，则称该文法为**算符优先文法(OPG)**

对于OG文法的几点说明:

(1) 运算是以中缀形式出现的

(2) 可以证明，若文法为OG文法，则不会出现两个非终结符相邻的句型。

(3) 算法语言中的表达式以及大部分语言成分的文法均是OG文法

构造FIRSTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	
E	{+} U FIRSTVT(E) U FIRSTVT(T)	E::=E+T E::=T
T	{*} U FIRSTVT(T) U FIRSTVT(F)	T::=T*F T::=F
F	{(, i}	F::=(E)|i

构造FIRSTVT(U)的算法

E::= E + T | T
T::= T * F | F
F::= (E) | i

	FIRSTVT	
E	{+} U FIRSTVT(E) U FIRSTVT(T)	E::=E+T E::=T
T	{*,(,i} U FIRSTVT(T) U FIRSTVT(F)	T::=T*F T::=F
F	{(, i}	F::=(E)|i

构造FIRSTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	
E	{+,*,(,i}UFIRSTVT(E) U FIRSTVT(T)	E::=E+T E::=T
T	{*,(,i} U FIRSTVT(T) U FIRSTVT(F)	T::=T*F T::=F
F	{(, i}	F::=(E)|i

构造FIRSTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	
E	{+,*,(,i}	E::=E+T E::=T
T	{*,(,i}	T::=T*F T::=F
F	{(, i}	F::=(E)|i

构造LASTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	LASTVT	
E	{+,*,(,i}	{+} U LASTVT(T)	E::=E+T E::=T
T	{*,(,i}	{*} U LASTVT(F)	T::=T*F T::=F
F	{(, i}	{), i}	F::=(E)|i

构造LASTVT(U)的算法

E::= E + T | T
T::= T * F | F
F::= (E) | i

	FIRSTVT	LASTVT	
E	{+,*,(,i}	{+} U LASTVT(T)	E::=E+T E::=T
T	{*,(,i}	{*,),i} U LASTVT(F)	T::=T*F T::=F
F	{(, i}	{), i}	F::=(E)|i

构造LASTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	LASTVT	
E	{+,*,(,i}	{+,*,),i} U LASTVT(T)	E::=E+T E::=T
T	{*,(,i}	{*,),i} U LASTVT(F)	T::=T*F T::=F
F	{(, i}	{), i}	F::=(E)|i

构造LASTVT(U)的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	LASTVT	
E	{+,*,(,i}	{+,*,),i}	E::=E+T E::=T
T	{*,(,i}	{*,),i}	T::=T*F T::=F
F	{(, i}	{), i}	F::=(E)|i

构造优先表的算法

E::= E + T | T
T::= T * F | F
F::= (E) | i

	FIRSTVT	LASTVT
E	{+,*,(,i}	{+,*,),i}
T	{*,(,i}	{*,),i}
F	{(, i}	{), i}

(1) 扫描每一条规则，如果有 U::=..ab.. |..aVb.. 设置 a=b

	+	*	i	(	)	#
+						
*						
i						
(					=	
)						
#						

构造优先表的算法

E::= E + T | T
T::= T * F | F
F::= (E) | i

	FIRSTVT	LASTVT
E	{+,*,(,i}	{+,*,),i}
T	{*,(,i}	{*,),i}
F	{(, i}	{), i}

(2) 扫描，如有 U::=..aV , 设置a<b, 对 b∈ FIRSTVT{V}

	+	*	i	(	)	#
+		<	<	<		
*			<	<		
i						
(	<	<	<	<	=	
)						
#						

构造优先表的算法

```
E::= E + T | T
T::= T * F | F
F::= ( E ) | i
```

	FIRSTVT	LASTVT
E	{+,*,(,i}	{+,*,),i}
T	{*,(,i}	{*,),i}
F	{(, i}	{), i}

(3) 扫描，如有 U::=..Va , 设置b>a, 对 b∈ LASTVT{V}

	+	*	i	(	)	#
+	>	<	<	<	>	
*	>	>	<	<	>	
i	>	>			>	
(	<	<	<	<	=	
)	>	>			>	
#						

构造优先表的算法

E::= E + T | T
T::= T * F | F
F::= (E) | i

	FIRSTVT	LASTVT
E	{+,*,(,i}	{+,*,),i}
T	{*,(,i}	{*,),i}
F	{(, i}	{), i}

(4) 对#特殊处理： E::=#E+T | #T, E::=E+T# | T#

	+	*	i	(	)	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(	<	<	<	<	=	
)	>	>			>	>
#	<	<	<	<		

问题

- **二义性？** （靠改写文法，或文法+优先关系）
- **从文法构造优先级的方法？**
  （FIRSTVT, LASTVT, 加三种优先规则的计算，加#的特殊处理）
- **是否所有文法都可以用算符优先分析法呢？**
  （适用范围：有算符/OG+“不纠结”/OPG）
- **句柄在哪里？长度是多少？**
  （句柄在栈顶，长度嘛……）

(3) 算符优先分析算法的实现

先定义优先级，在分析过程中通过比较相邻运算符之间的优先级来确定句型的“句柄”并进行归约。

? -- 最左素短语

[定义] **素短语**：文法G的句型的素短语是一个短语，它至少包含有一个终结符号，并且除它自身以外不再包含其他素短语。

例:文法G[E]

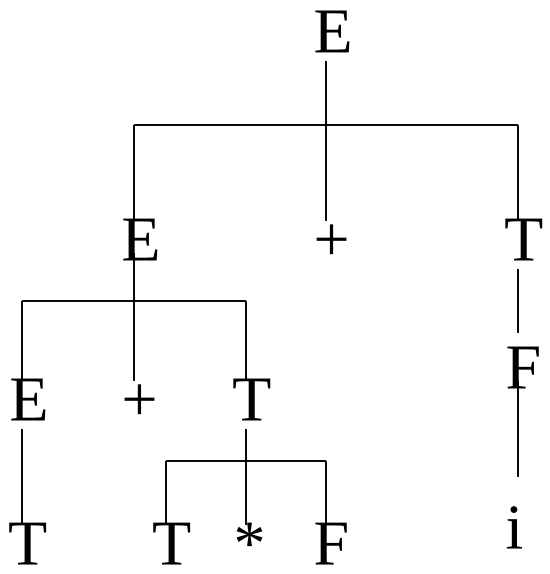
E::=E+T|T

T::=T*F|F

F::=(E)|i

求句型 T+T*F+i 的素短语

文法的语法树:



例:文法G[E]

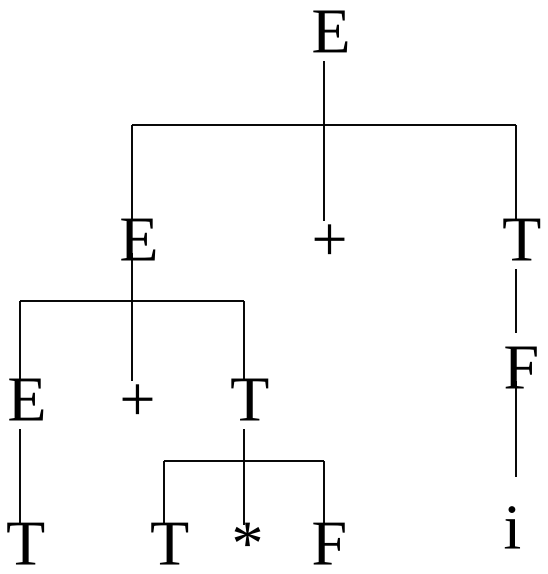
E::=E+T|T

T::=T*F|F

F::=(E)|i

求句型 T+T*F+i 的素短语

文法的语法树:



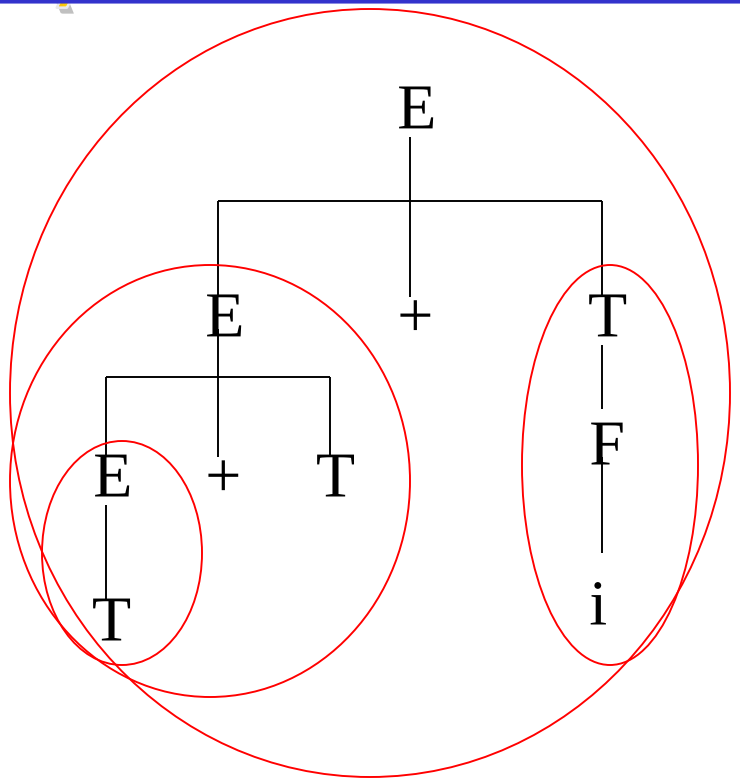
短语:T+T*F+i, T+T*F

T(最左), T*F, i

其中T不包含终结符,T是句柄

而T+T*F+i和T+T*F包含其他素短语。

只有T*F和i为素短语,其中T*F为最左素短语,而该句型句柄为T。



句型：T + T + i

短语：T+T+i

T +T

T

i

句柄：T

素短语：T+T, i

算符优先分析法如何确定当前句型的最左素短语?

设有OPG文法句型为:

$$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$$

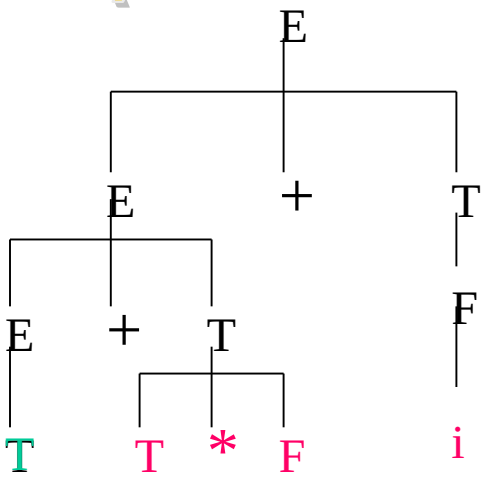
其中N_i为非终结符(可以为空), a_i为终结符

定理：一个OPG句型的最左素短语是满足下列条件的最左子串：$a_{j-1}N_ja_j\ldots N_ia_iN_{i+1}a_{i+1}$

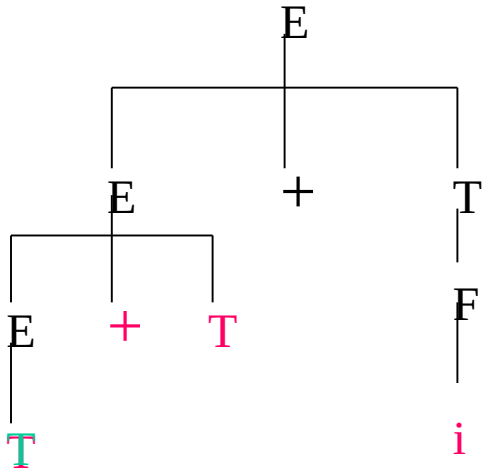
其中 $a_{j-1}<a_j$

$a_j=a_{j+1}$, $a_{j+1}=a_{j+2}$,…, $a_{i-2}=a_{i-1}$, $a_{i-1}=a_i$

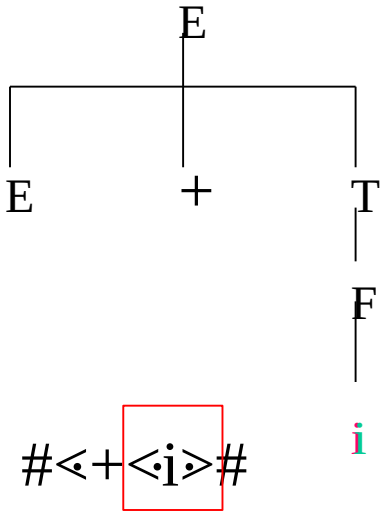
$a_i>a_{i+1}$



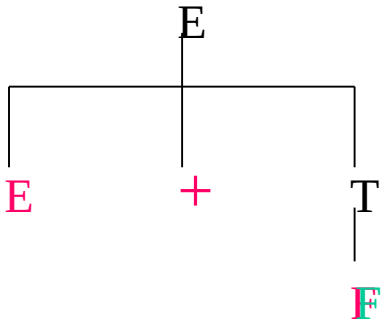
#<+<*>+<i>#



#<+>+<i>#



#<+<i>#



#<+>#

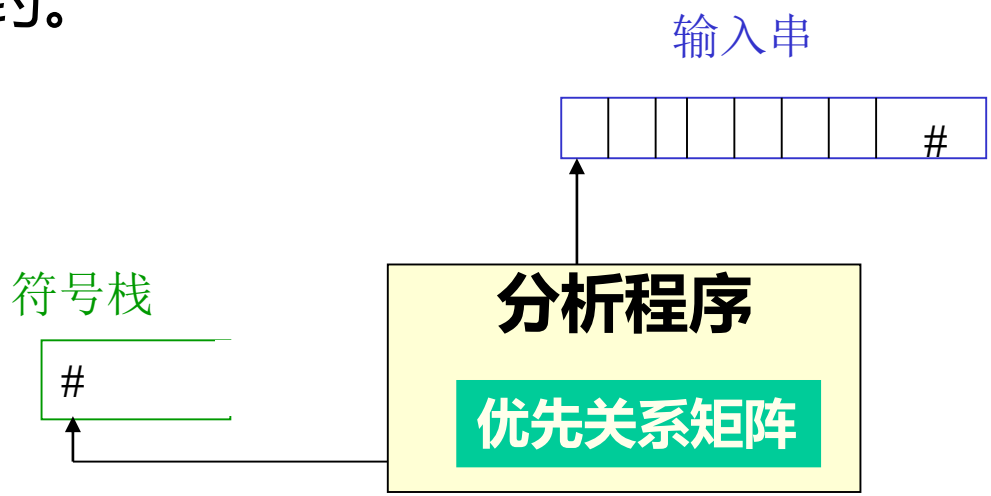
步骤	句型	关系	最左子串	归约符号
1	#T+T*F+i#	#<+<*>+<i>>#	T*F	T
2	#T+T+i#	#<+>+<i>>#	T+T	E
3	#E+i#	#<+<i>>#	i	F
4	#E+F#	#<+>#	E+F	E

可以看出:

1. **每次归约最左子串,确实是当前句型的最左素短语(语法树)**
2. **归约的不都是真句柄（仅i归约为F是句柄，但它是最左素短语)**
3. **没有完全按规则进行归约,因为素短语不一定是简单短语**

算符优先分析法的实现：

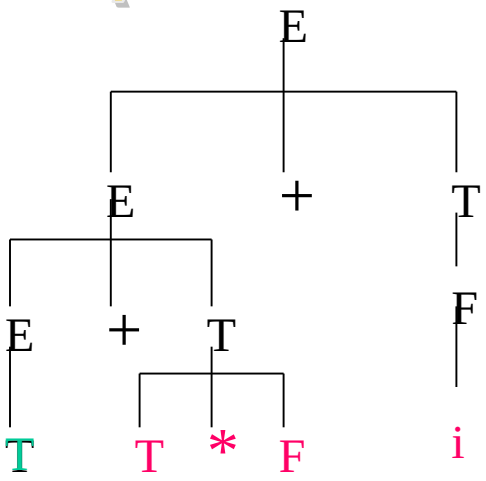
基本部分是找句型的最左子串（最左素短语）并进行归约。



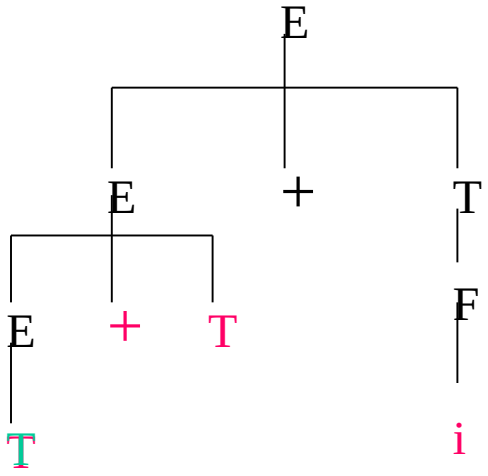
栈内符号优先级 < = 栈外符号优先级：移进

栈内符号优先级 > 栈外符号优先级：找到了**素短语的尾**，再往前找素短语的头，确定“句柄”长度进行归约

LR分析方法



#<+<*>+<i>#



#<+>+<i>#

这样产生的是“最左素短语”，然后规约它所对应的“推导”是（ ）推导?

（A）最左推导 （B）最右推导 （C）基本是最右推导

#<+<i>#

#<+>#

LR 文法

是一种自底向上的分析方法（1965年 D.Knuth 提出）

L：从左向右分析 (left to right)

R：产生“最右推导”(right-most derivation)

k=1：向前查看“k=1”个符号

LR（k）

从左到右扫描(L)自底向上进行归约(Right-most Derivation)

(一定是规范归约)，是自底向上分析方法的高度概括和集中

历史 + 展望 + 现状 => 句柄

能否追踪当前的分析状态?

S

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

int	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E

文法:

S → E

E → F

E → E + F

F → F*T

F → T

T → int

T → (E)

int	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→.F
E→.E+F

文法:

S → E

E → F

E → E + F

F → F*T

F → T

T → int

T → (E)

int	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→.F
E→.E+F
F→.F*T
F→.T

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

int	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→.F
E→.E+F
F→.F*T
F→.T
T→.int
T→.(E)

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

int	+	int	*	int	+	int

能否追踪当前的分析状态?

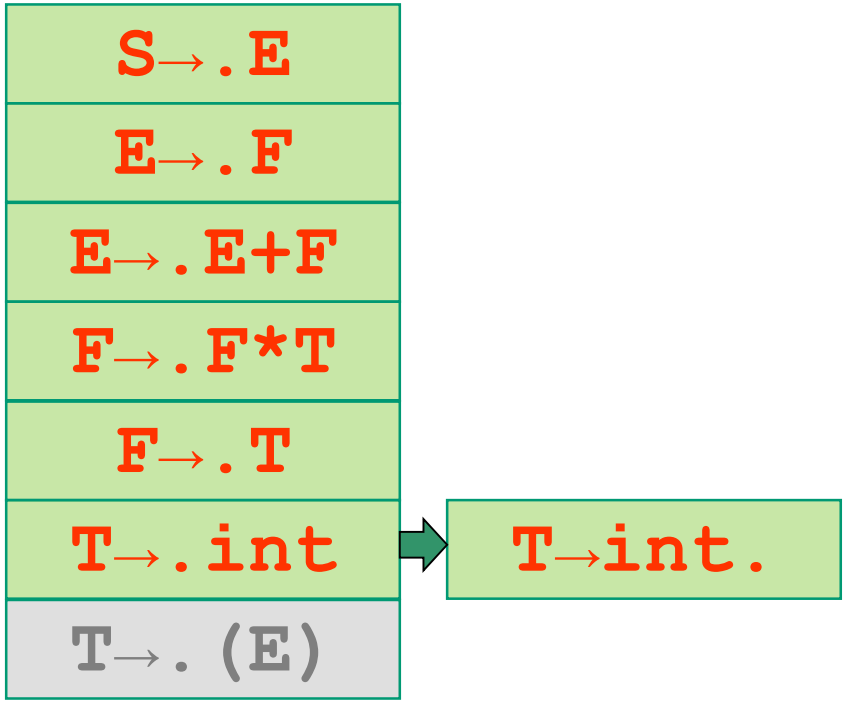
S→.E
E→.F
E→.E+F
F→.F*T
F→.T
T→.int
T→.(E)

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

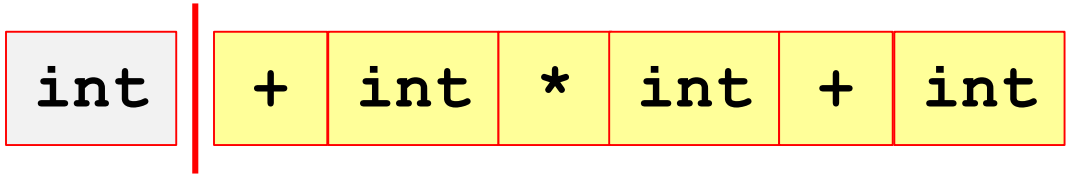
int	+	int	*	int	+	int

能否追踪当前的分析状态?

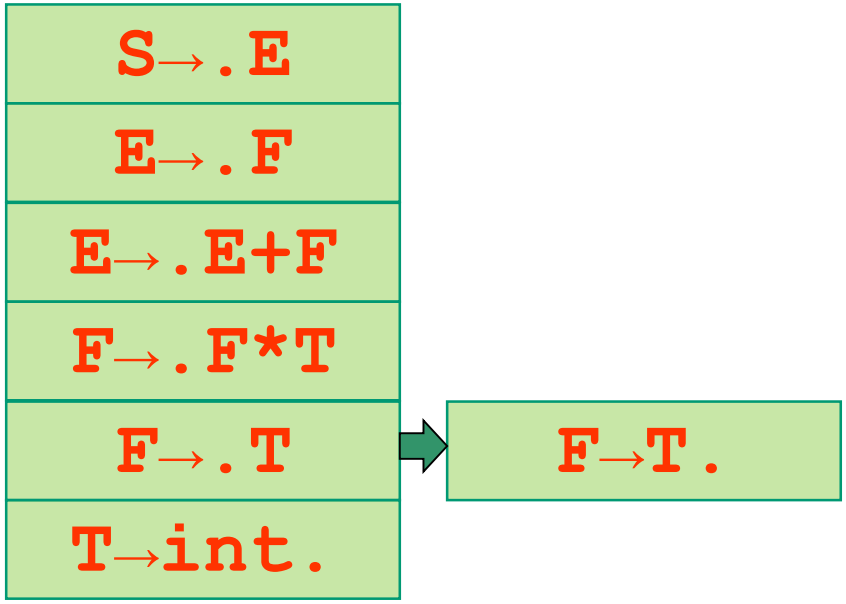


文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

T	+	int	*	int	+	int

能否追踪当前的分析状态?

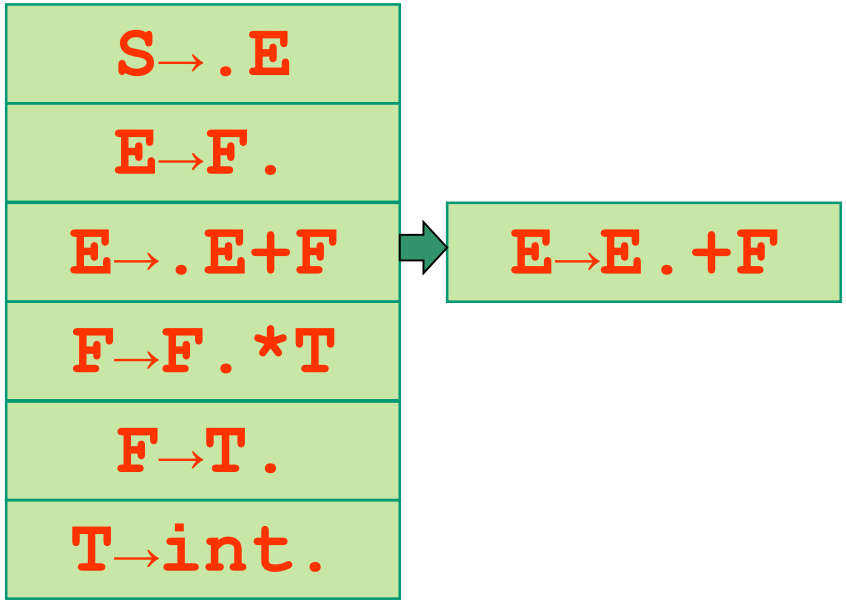
S→.E
E→.F
E→.E+F
F→.F*T
F→T.
T→int.

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

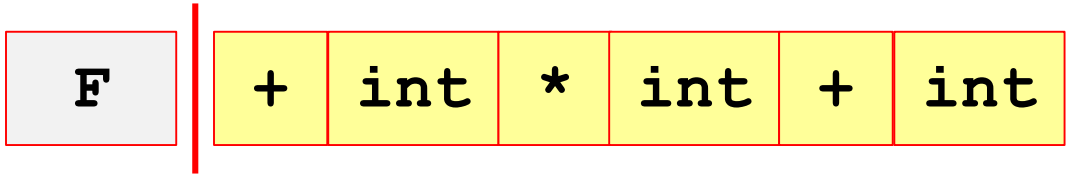
F	+	int	*	int	+	int

能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



能否追踪当前的分析状态?

S→.E
E→F.
E→E.+F
F→F.*T
F→T.
T→int.

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

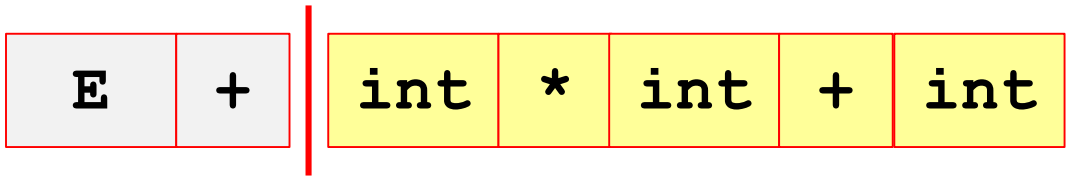
E	+	int	*	int	+	int

能否追踪当前的分析状态?

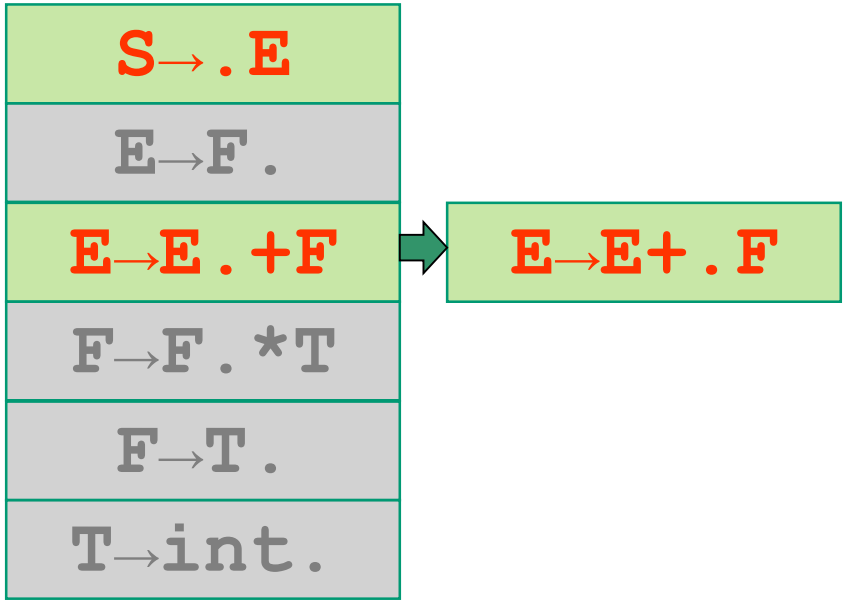
S→.E
E→F.
E→E.+F
F→F.*T
F→T.
T→int.

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

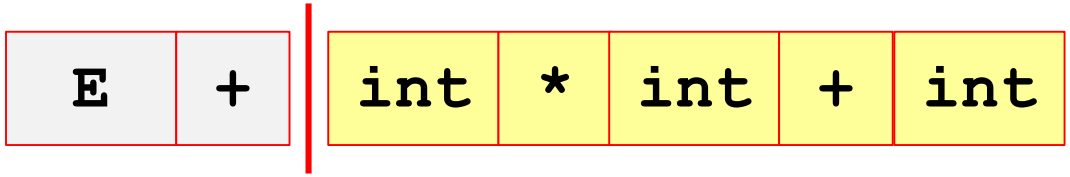


能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



能否追踪当前的分析状态?

S→.E
E→E+.F

文法:

S → E

E → F

E → E + F

F → F*T

F → T

T → int

T → (E)

E	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→E+.F
F→.F*T
F→.T

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

E	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→E+.F
F→.F*T
F→.T
T→.int
T→.(E)

文法:

S → E

E → F

E → E + F

F → F*T

F → T

T → int

T → (E)

E	+	int	*	int	+	int

能否追踪当前的分析状态?

S→.E
E→E+.F
F→.F*T
F→.T
T→.int
T→.(E)

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

E	+	int	*	int	+	int

能否追踪当前的分析状态?

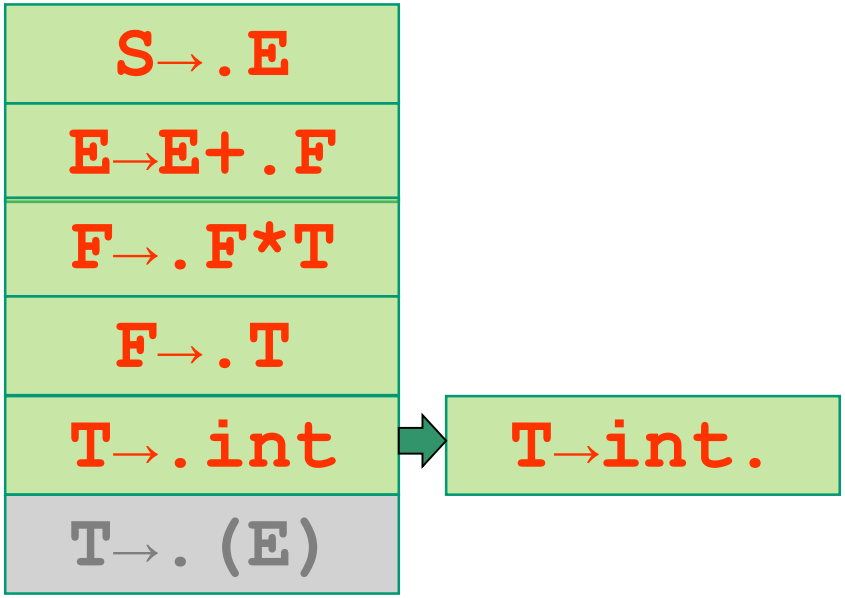
S→.E
E→E+.F
F→.F*T
F→.T
T→.int
T→.(E)

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

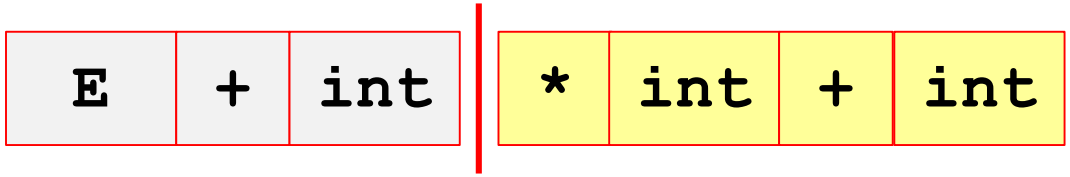
E	+	int	*	int	+	int	

能否追踪当前的分析状态?

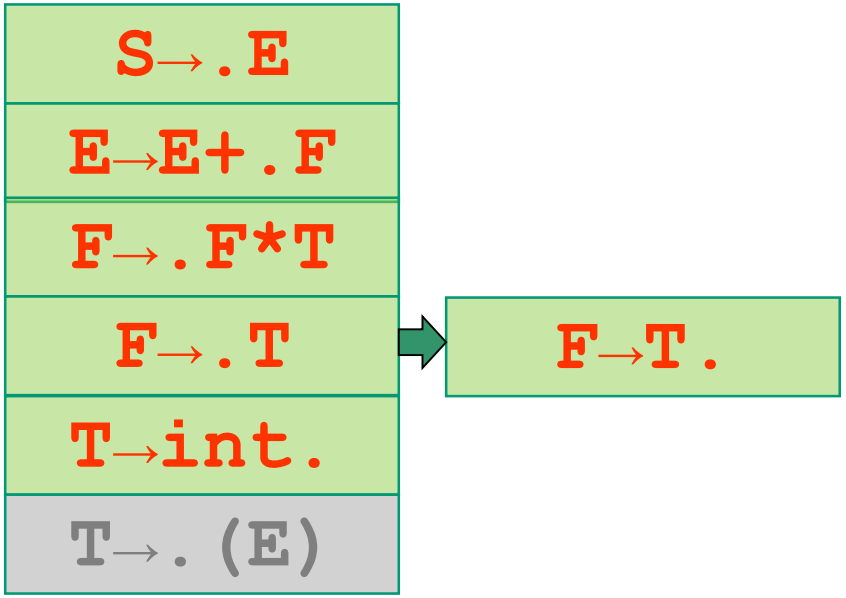


文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

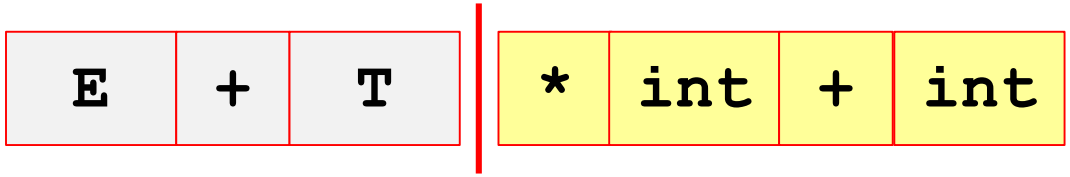


能否追踪当前的分析状态?

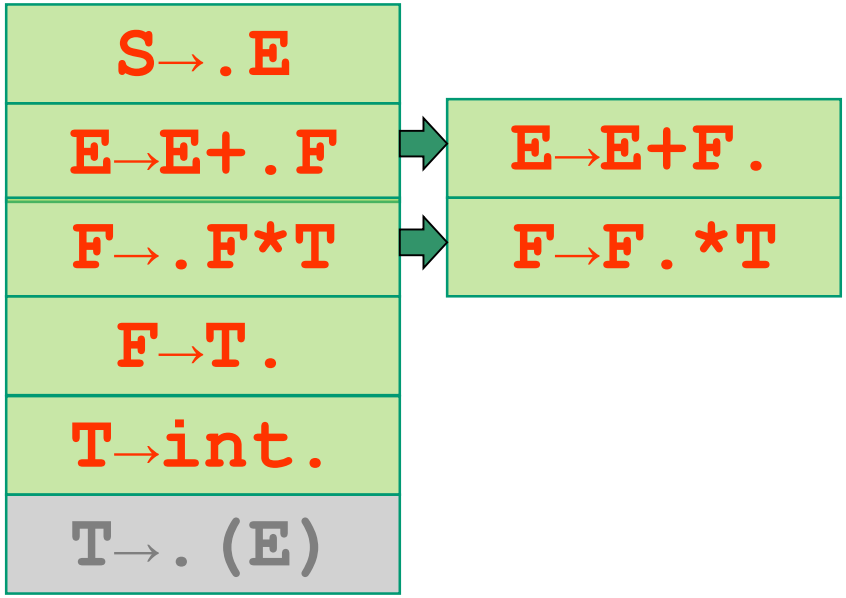


文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

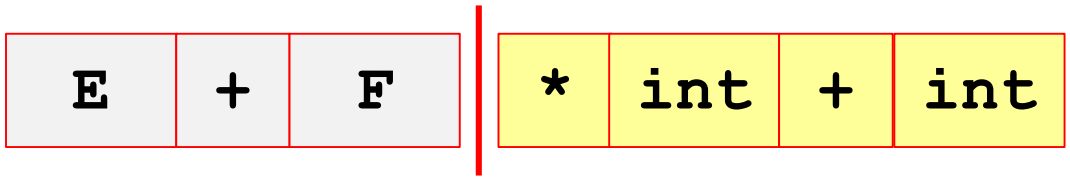


能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



能否追踪当前的分析状态?

S→.E
E→E+F.
F→F.*T
F→T.
T→int.

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

E	+	F	*	int	+	int	

能否追踪当前的分析状态?

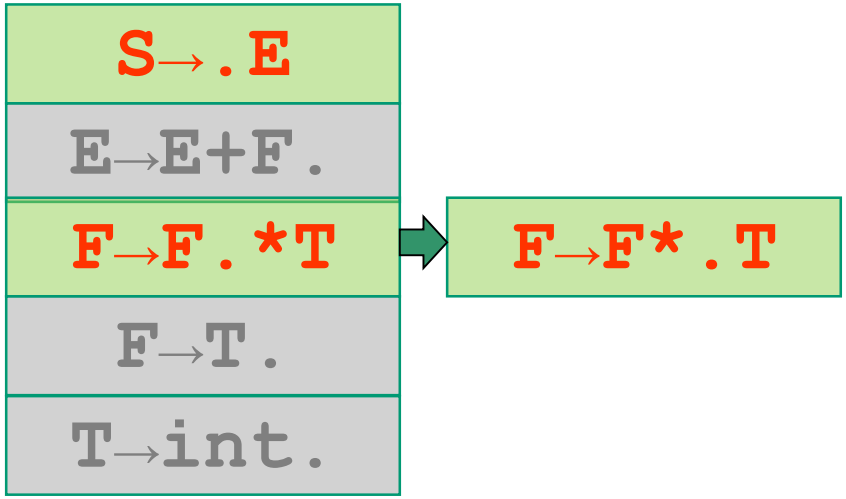
S→.E
E→E+F.
F→F.*T
F→T.
T→int.

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

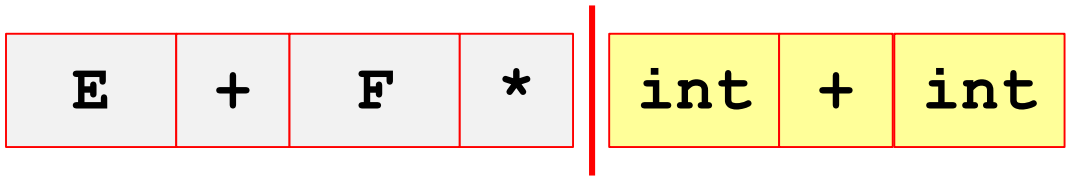
E	+	F	*	int	+	int

能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



能否追踪当前的分析状态?

S→.E
F→F*.T

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

E	+	F	*	int	+	int

能否追踪当前的分析状态?

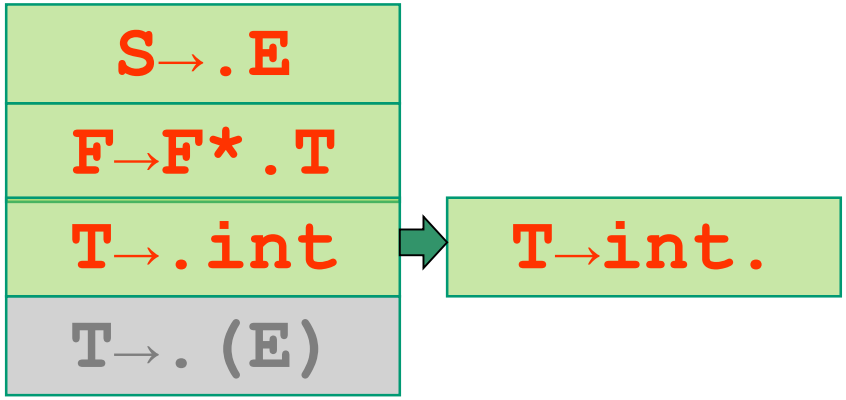
S→.E
F→F*.T
T→.int
T→.(E)

文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

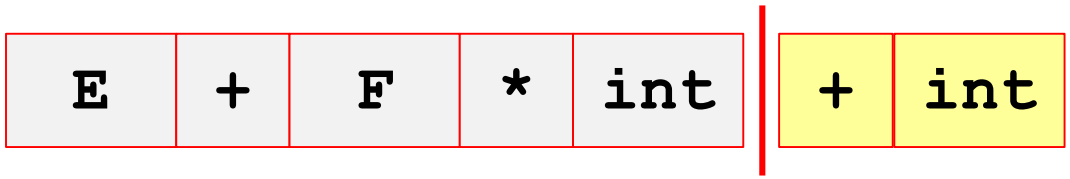
E	+	F	*	int	+	int

能否追踪当前的分析状态?

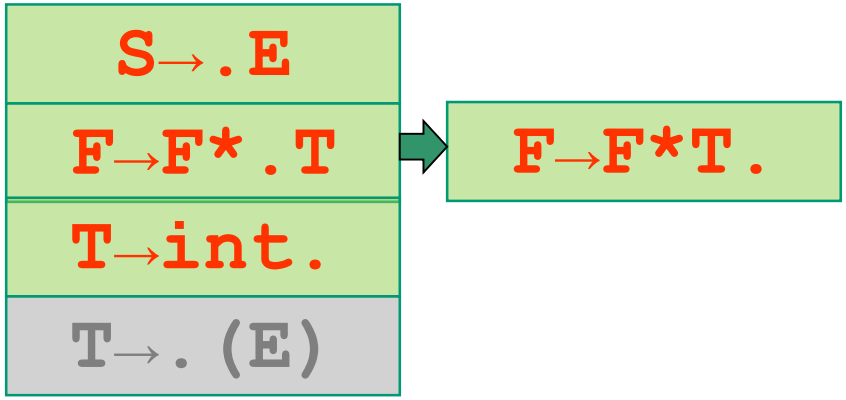


文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

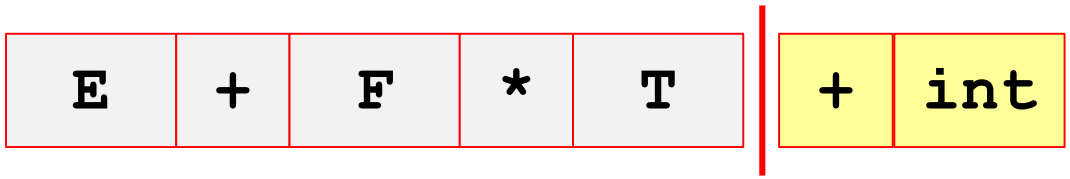


能否追踪当前的分析状态?



文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)



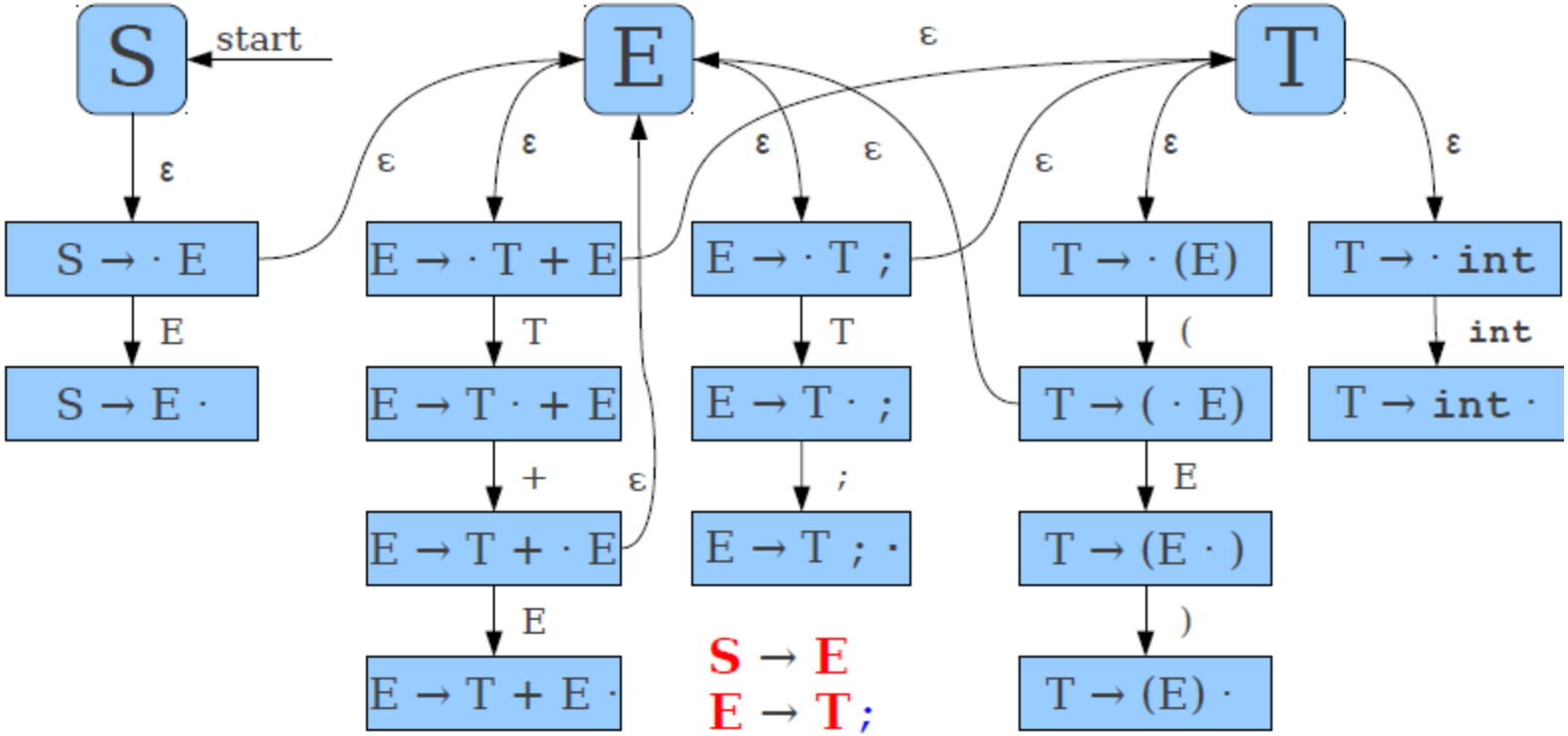
能否追踪当前的分析状态?

S→.E
F→F*T.
T→int.
T→.(E)
E→F.

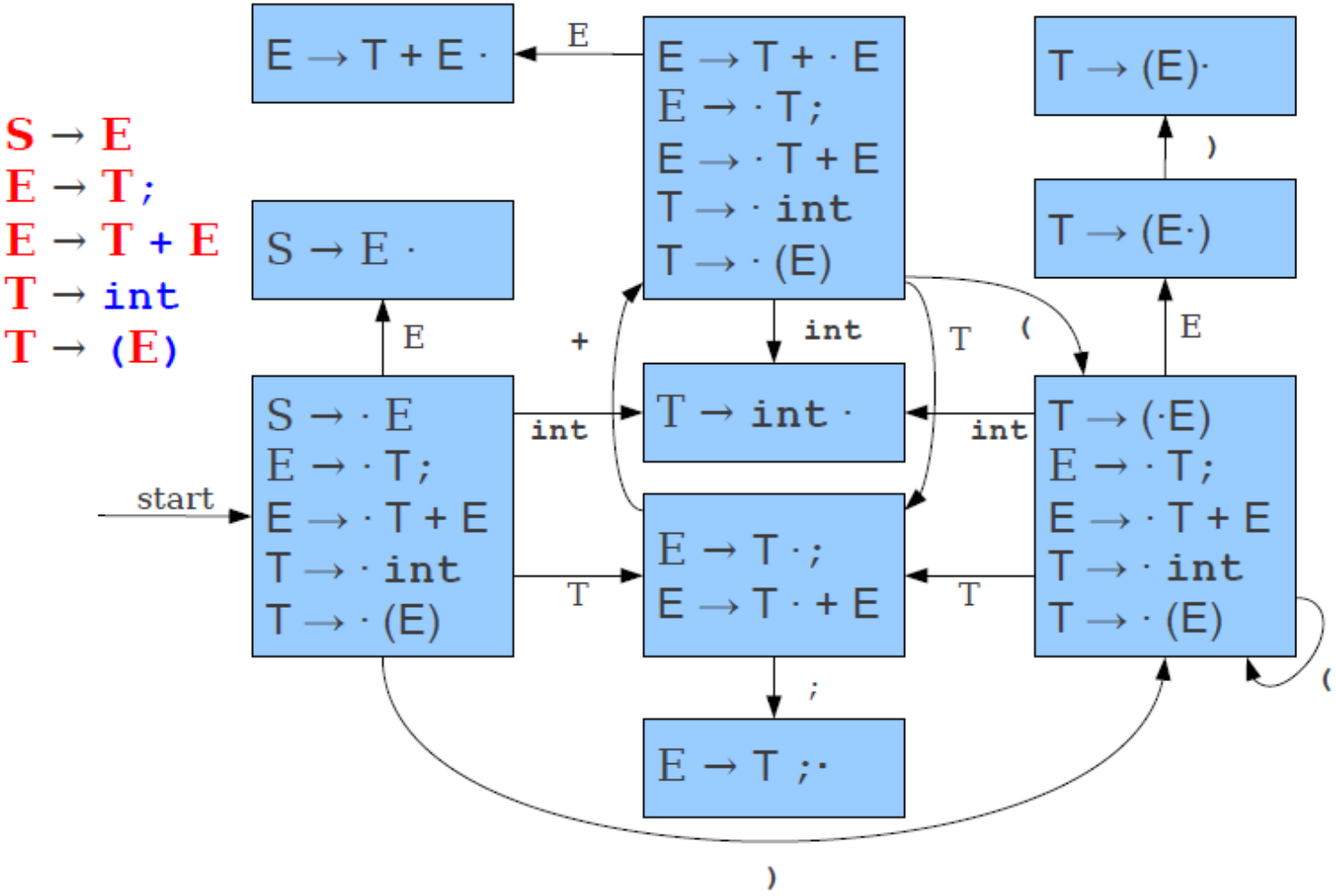
文法:

S → E
E → F
E → E + F
F → F*T
F → T
T → int
T → (E)

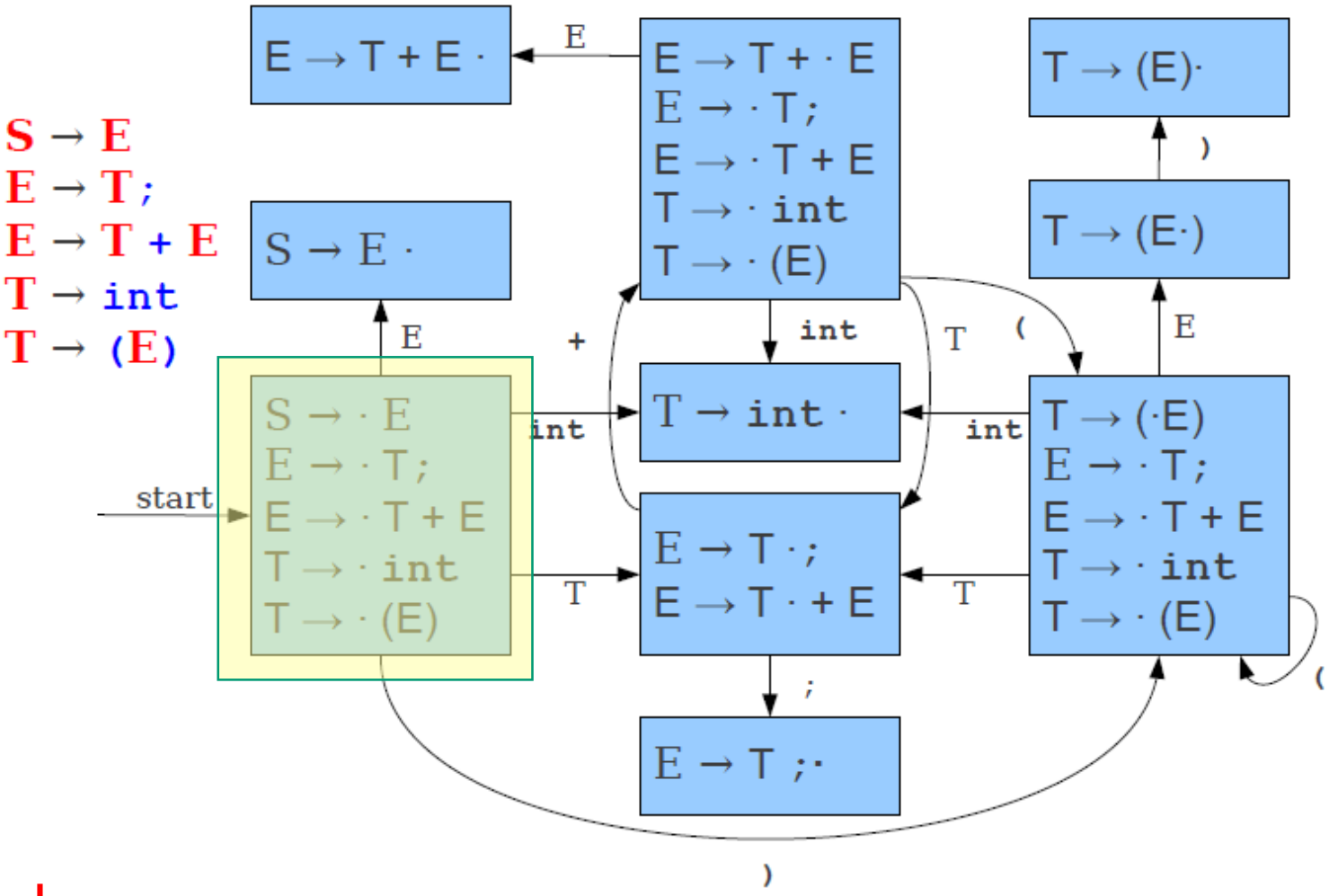
E	+	F	*	F	+	int



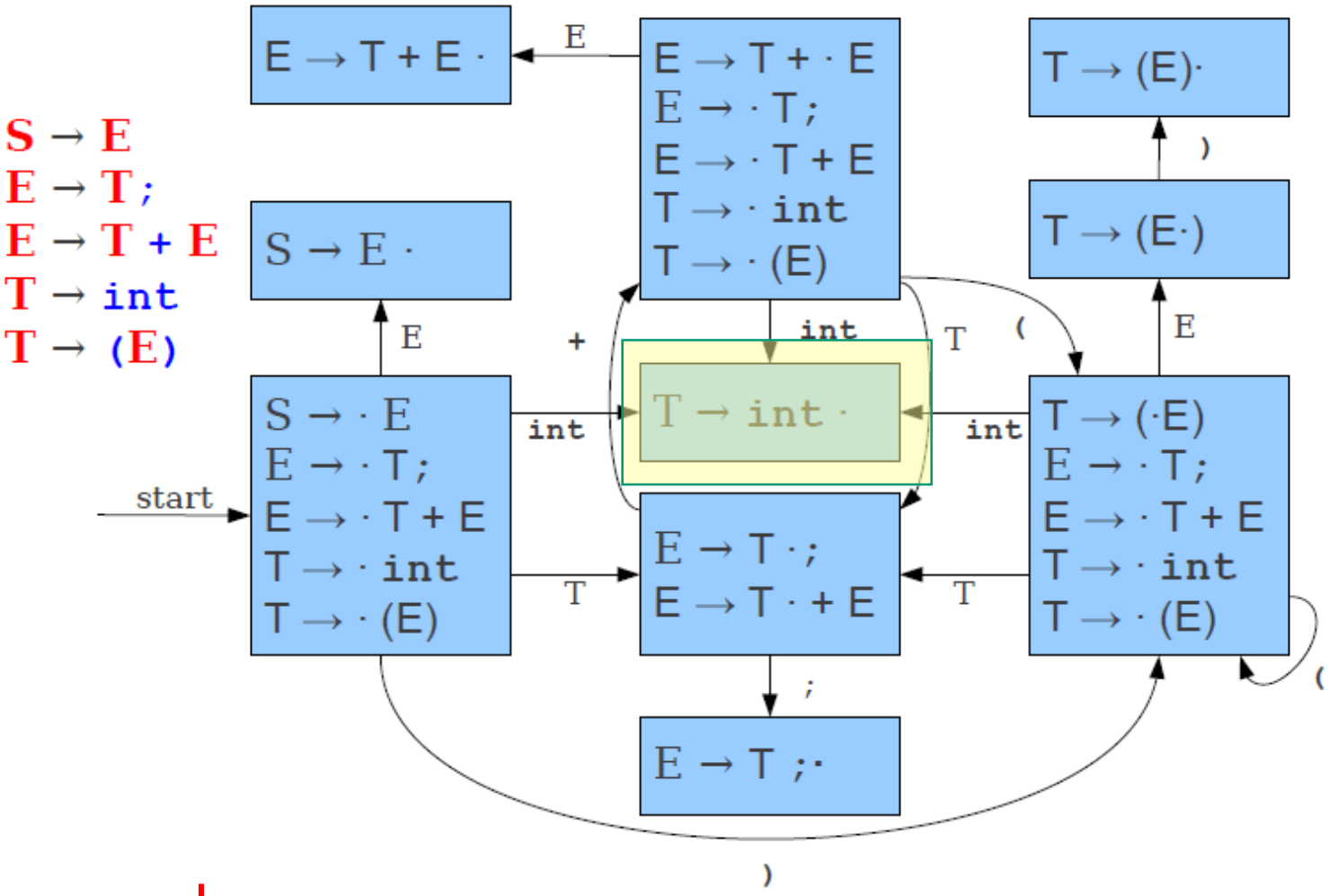
Source: Stanford CS143 (2012)



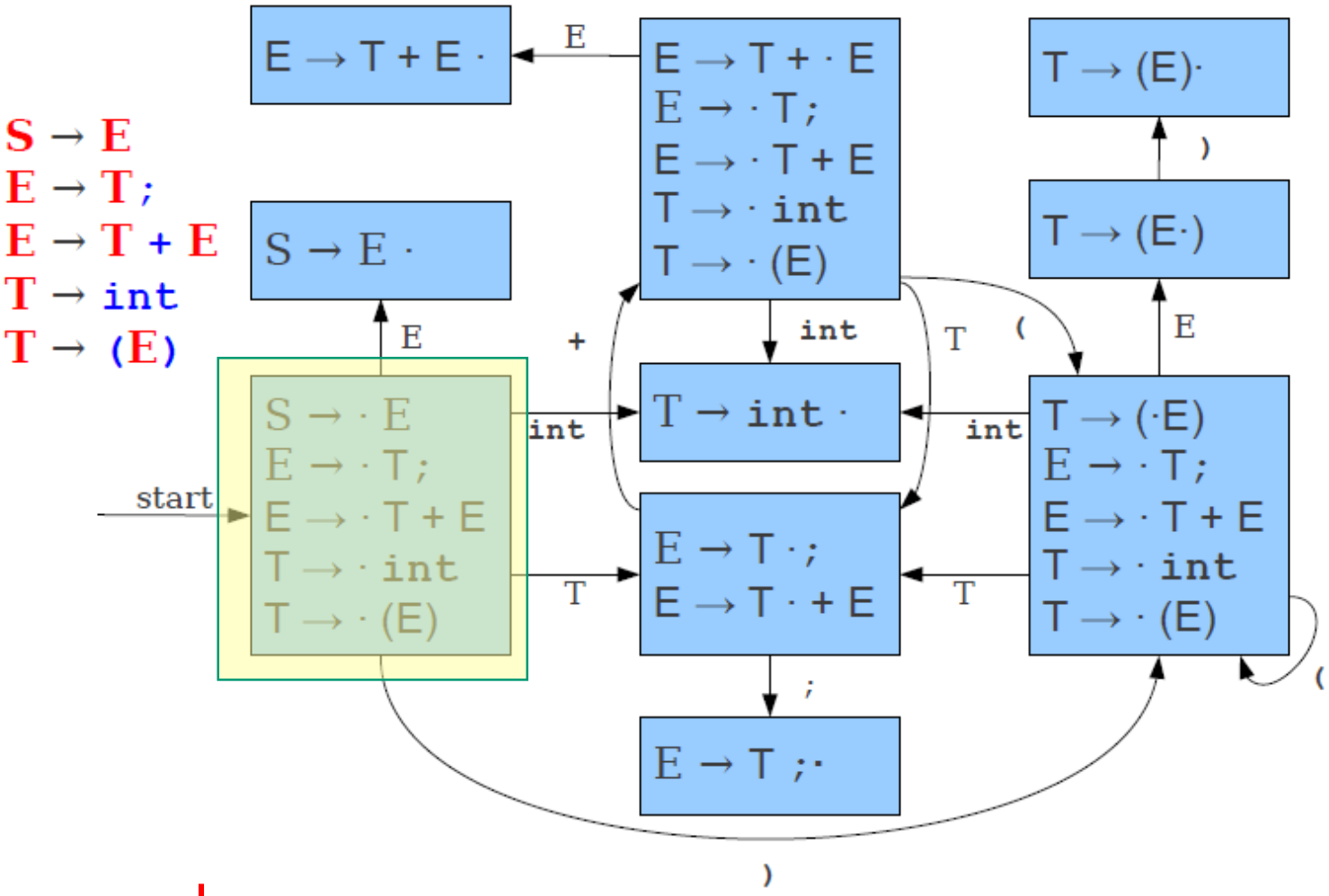
Source: Stanford CS143 (2012)


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
S → E ·
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
E → T ·;
E → T · + E
E → T ;·
T → (E)·
T → (E·)
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E
+
int
T
(
)
;

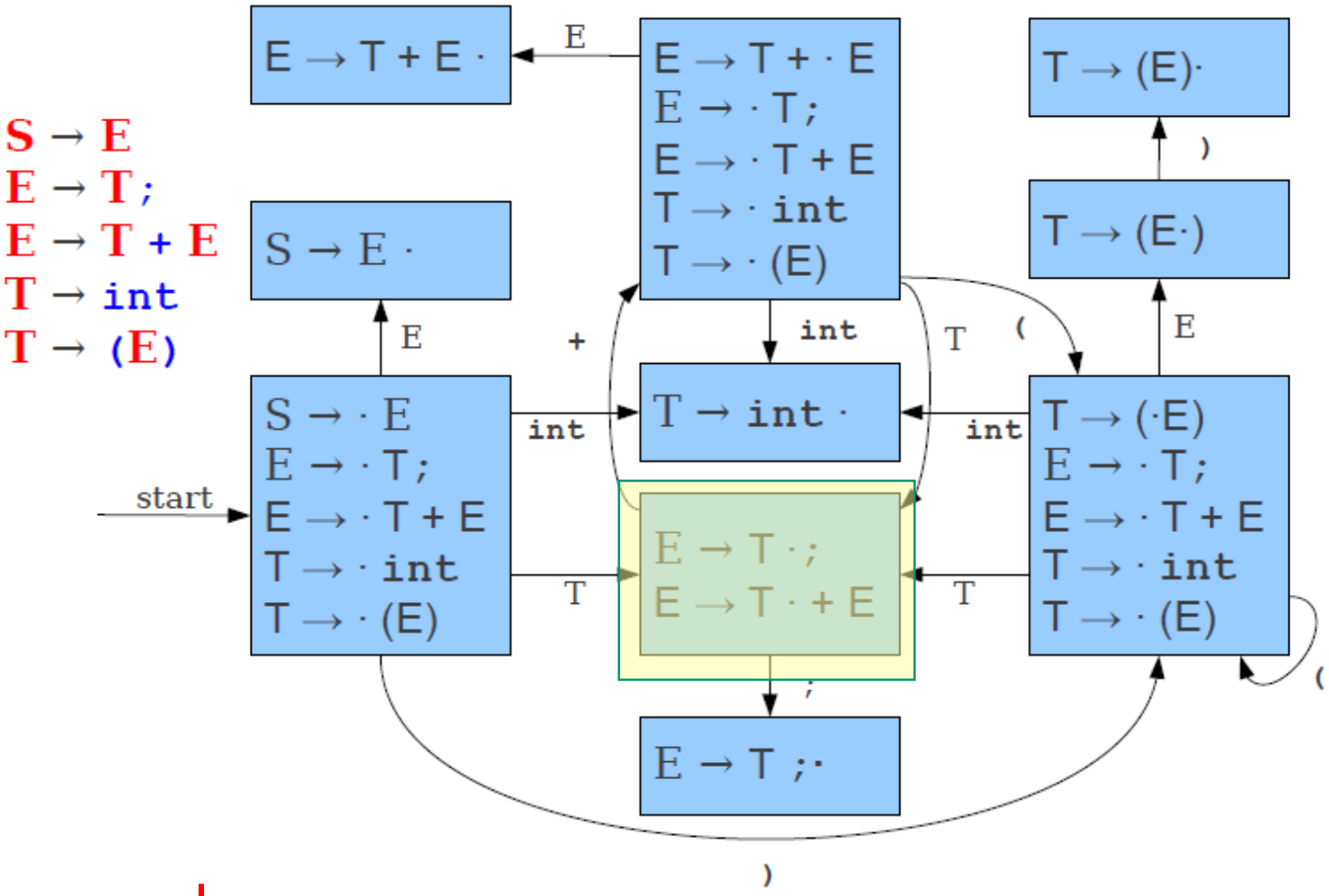
int	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(

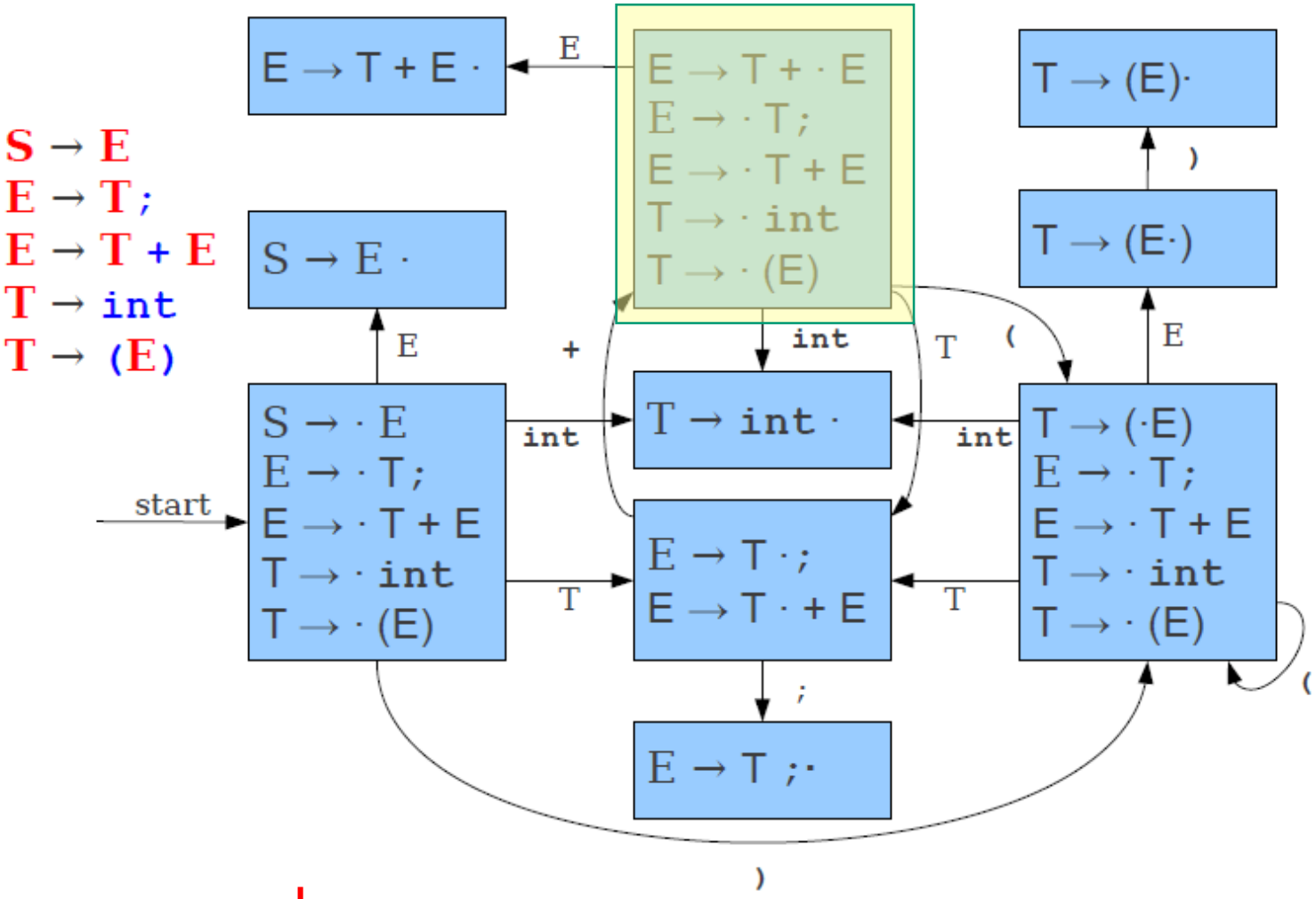
int	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
S → E ·
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
T → (E·)
T → int ·
T → (·E)
E → T ·;
E → T · + E
E → T ;·
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
E
+
int
T
(
)
;

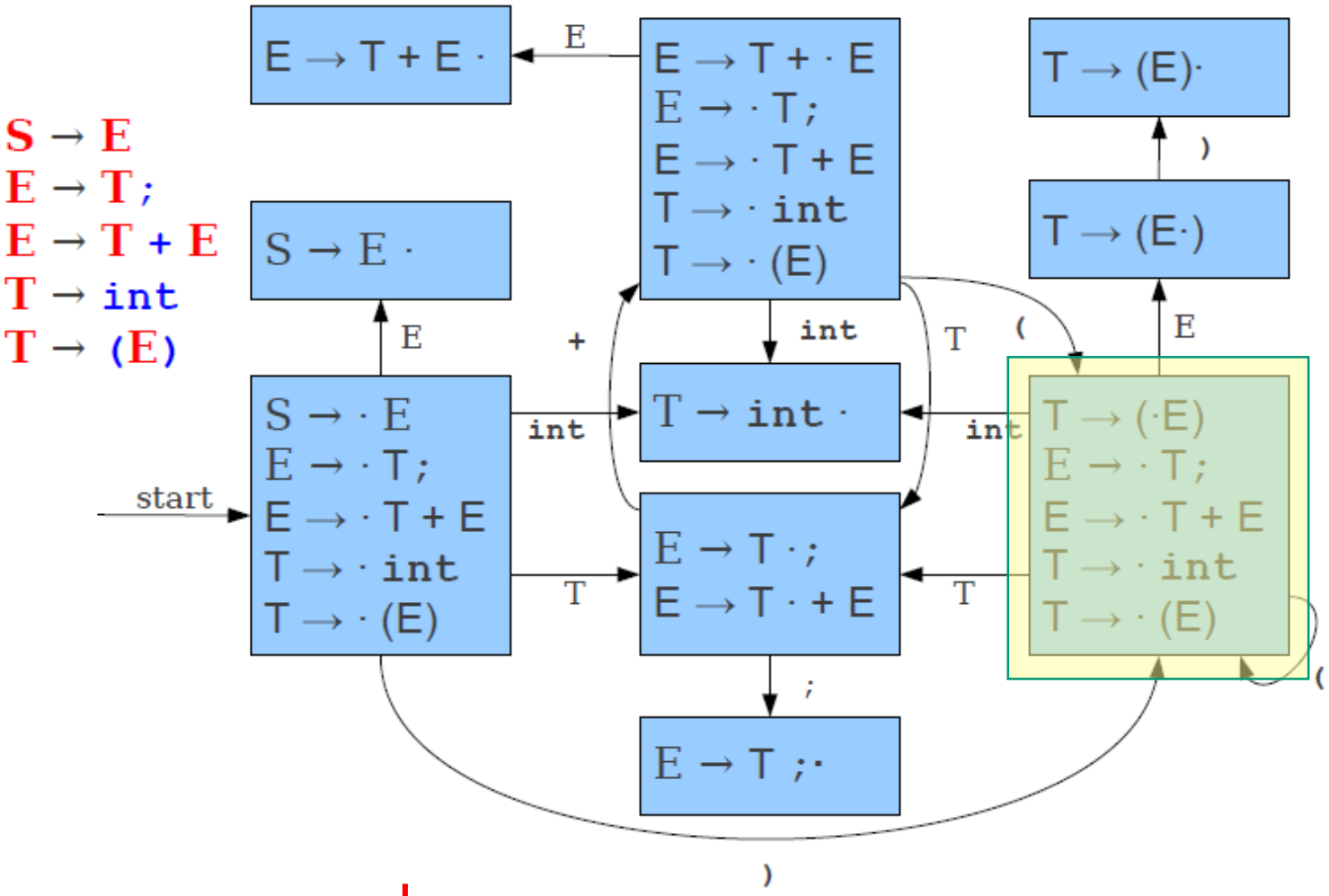
T	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
int
E → T ·;
E → T · + E
T
T
;
E → T ;·
)
(

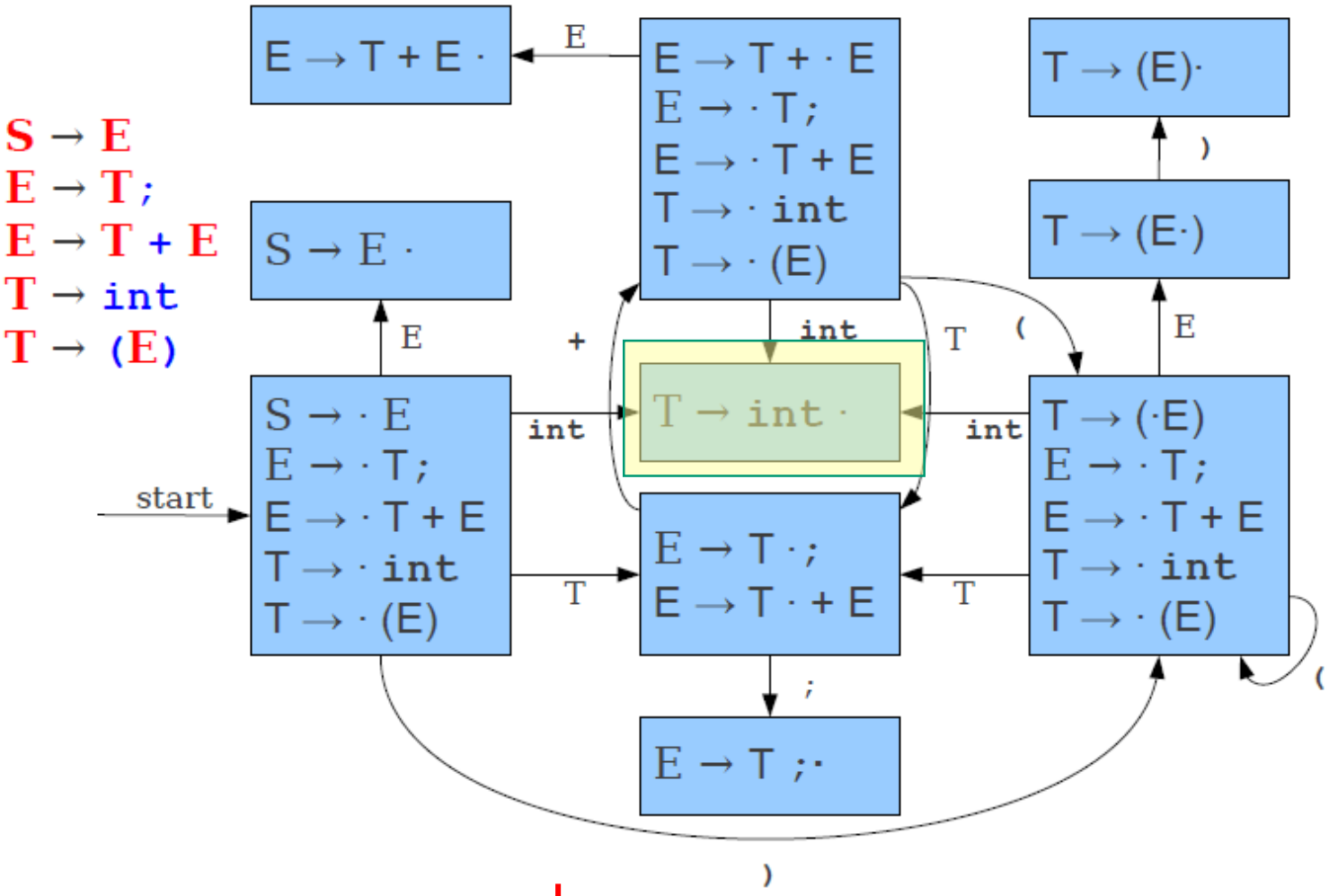
T	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(

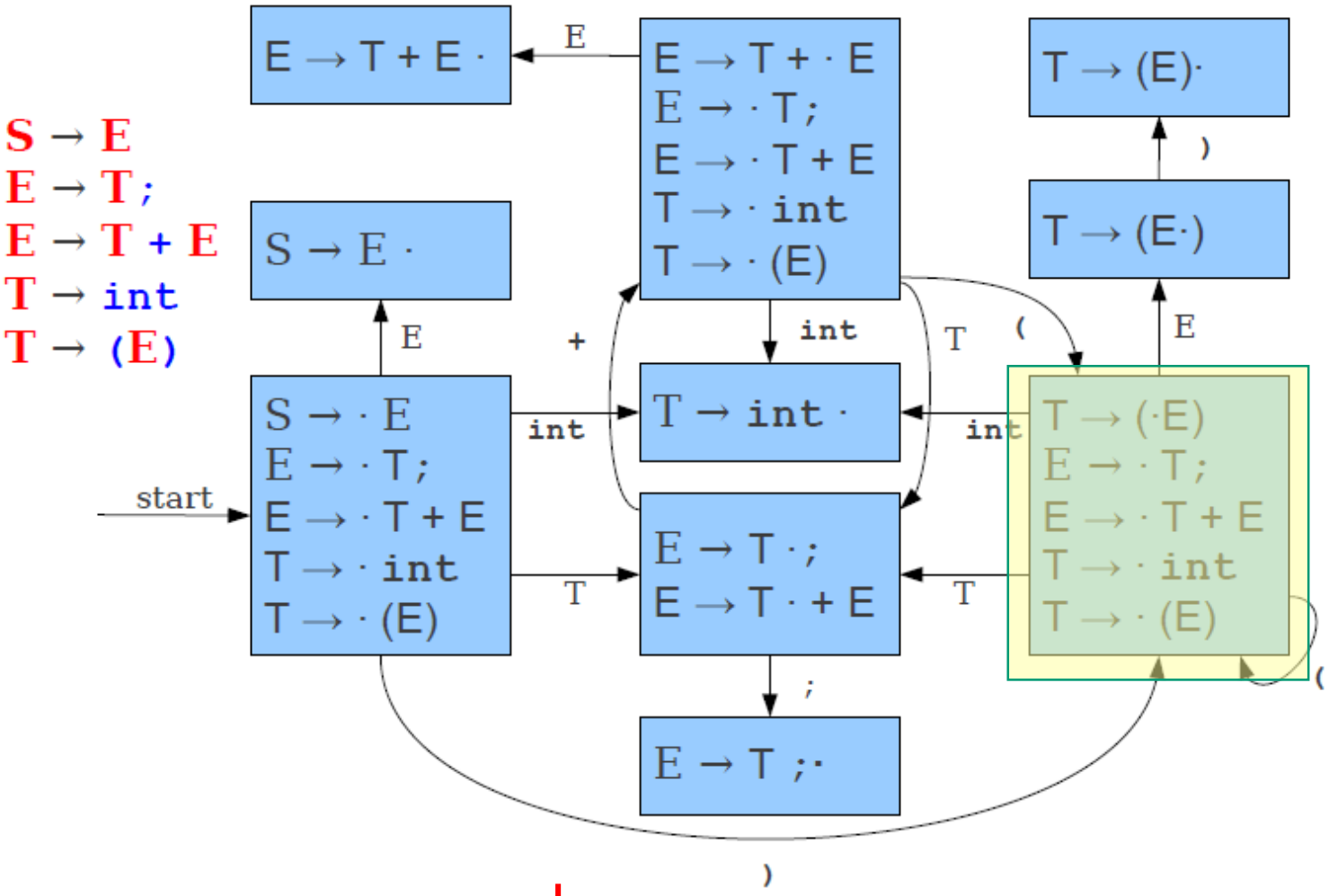
T	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(

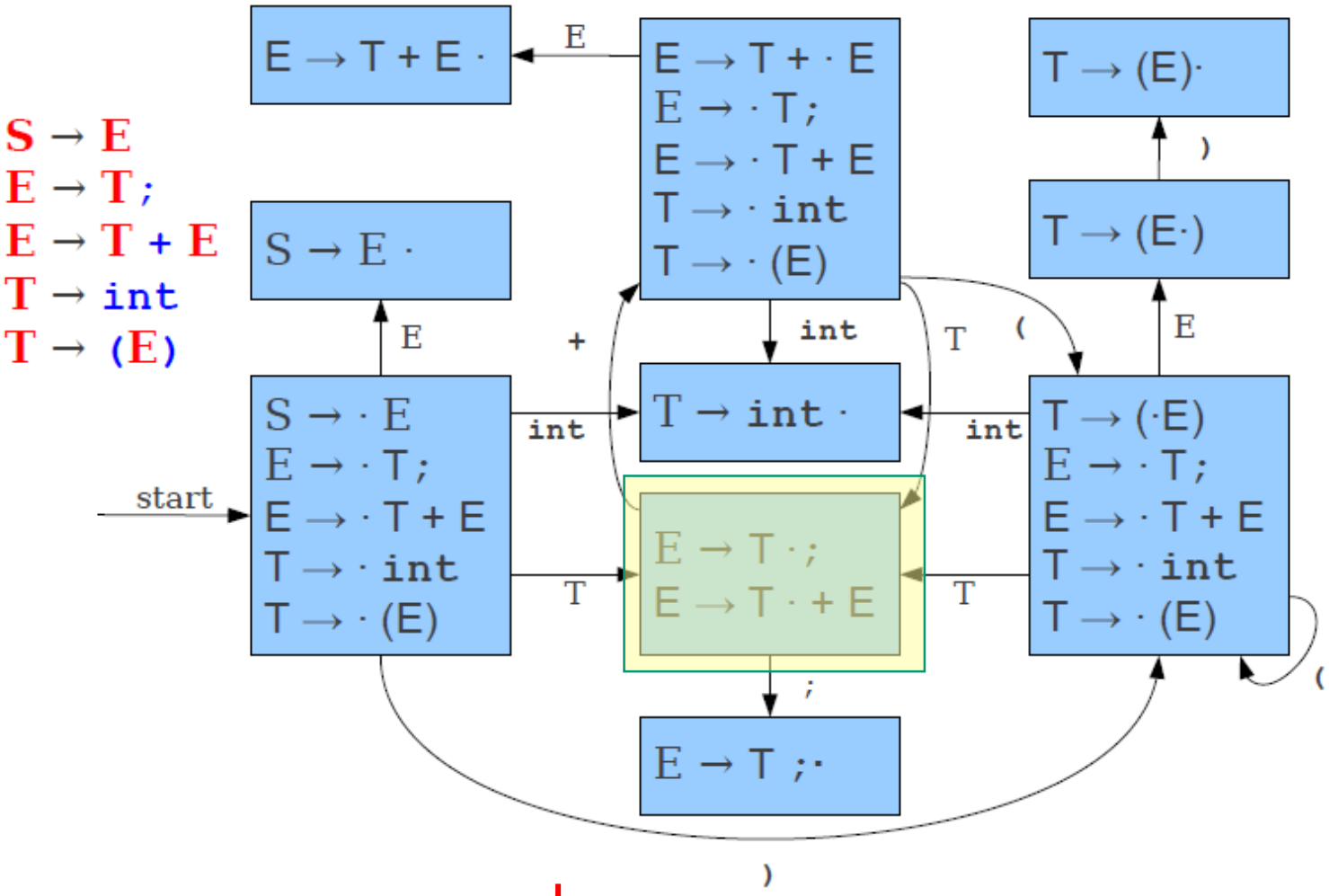
T	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(

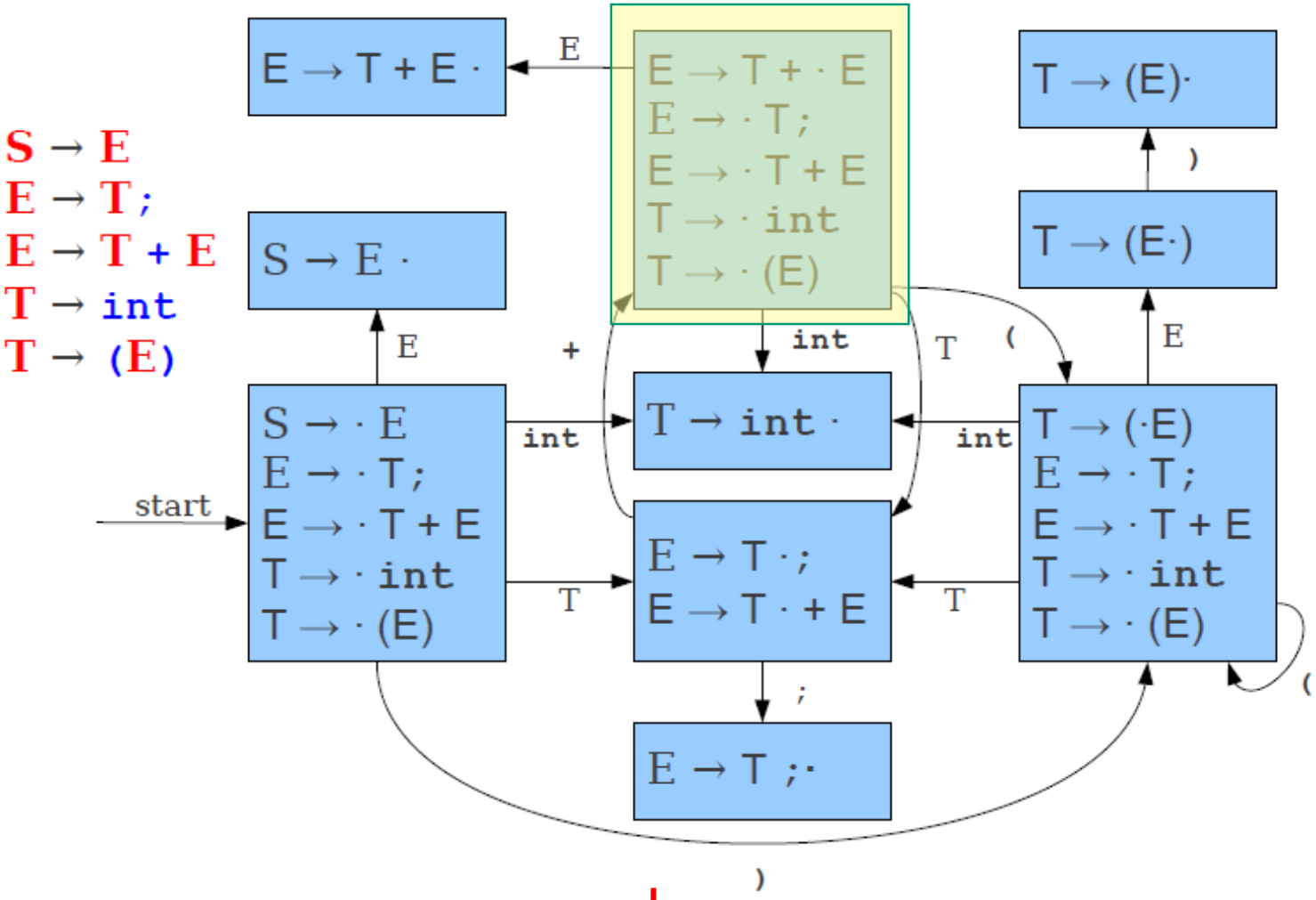
T	+	(	int	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
E
+
int
T
(
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(

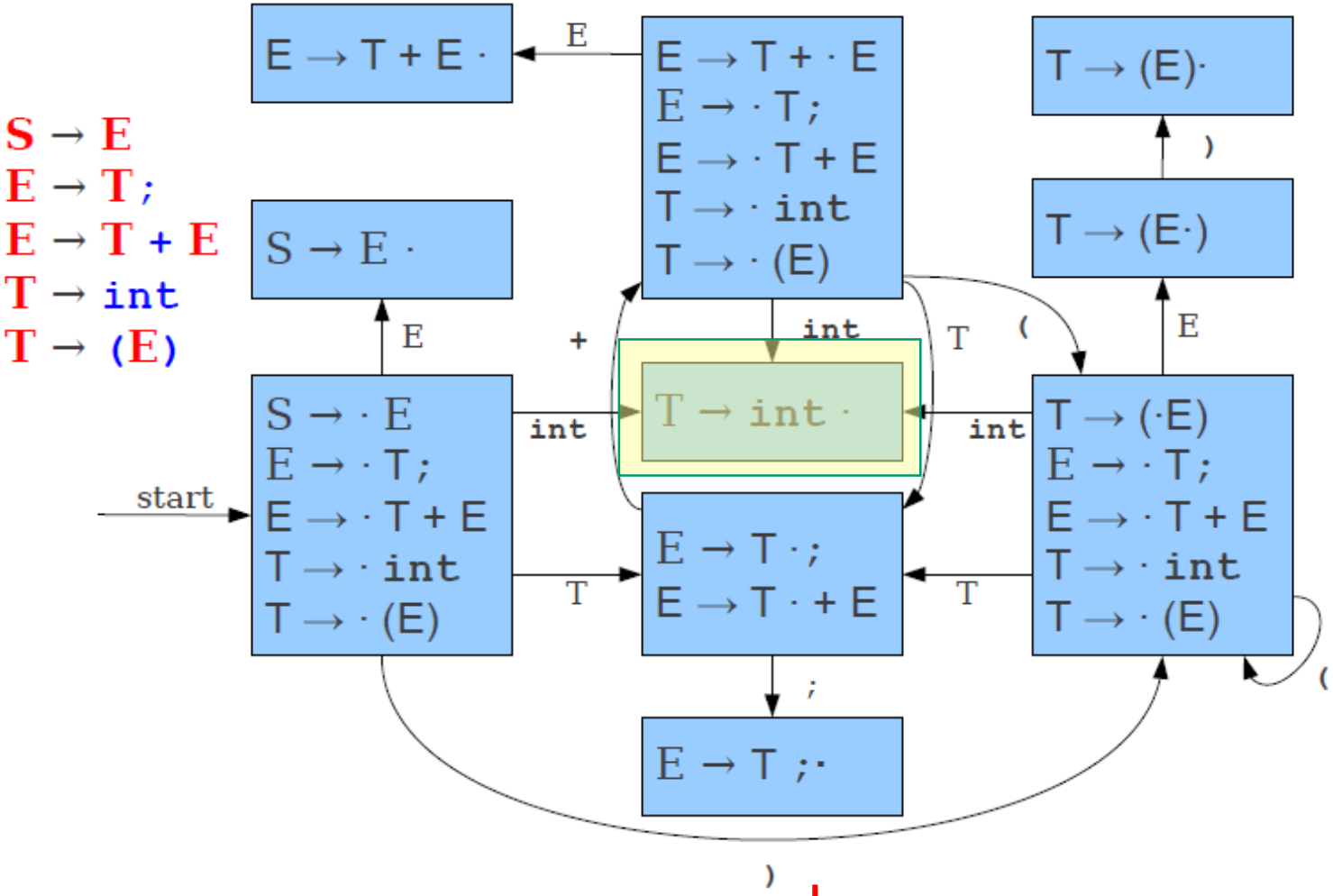
T	+	(	T	+	int	;	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E → T ·;
E → T · + E
;
E → T ;·
)

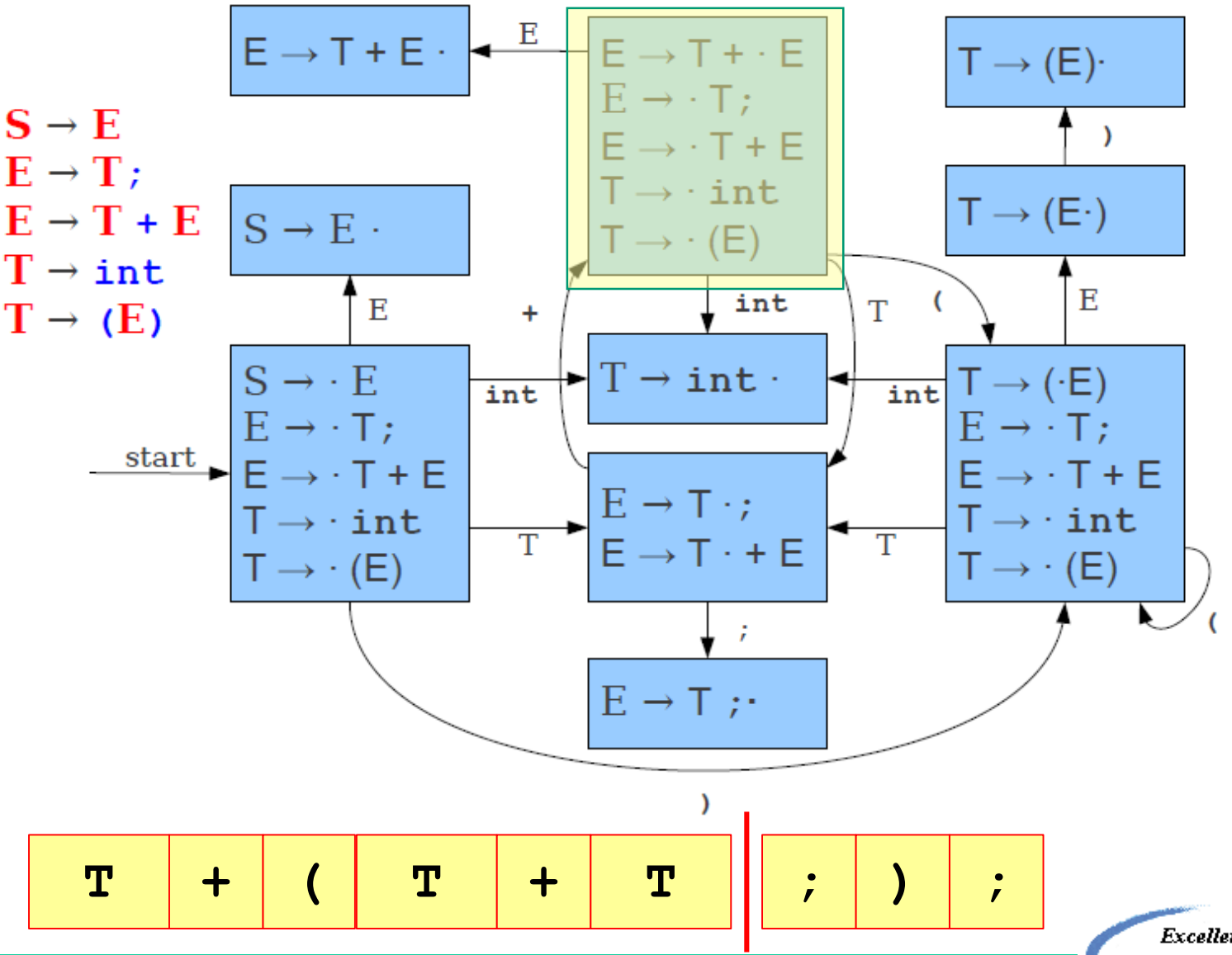
T	+	(	T	+	int	;	)	;

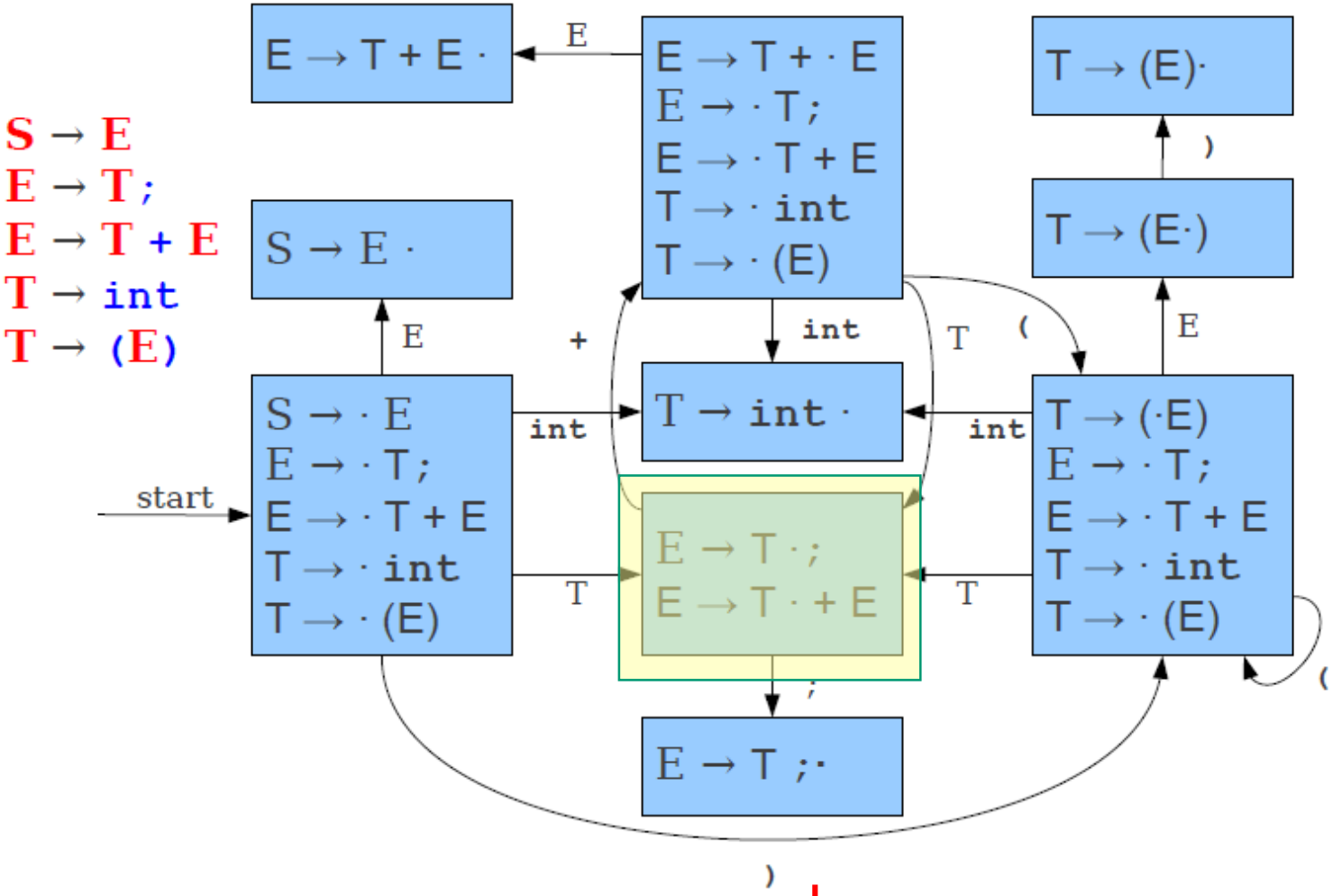

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(

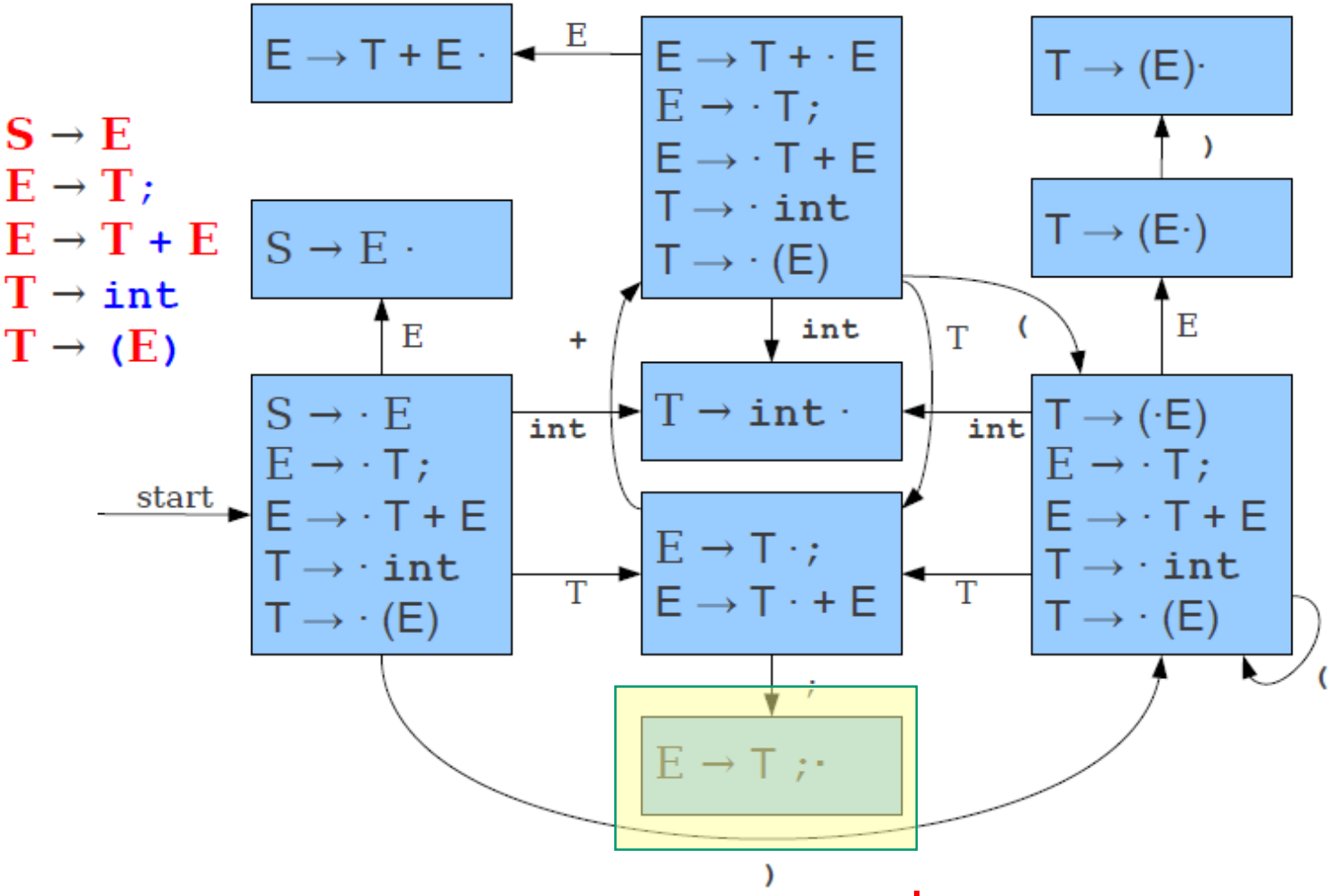
T	+	(	T	+	int	;	)	;

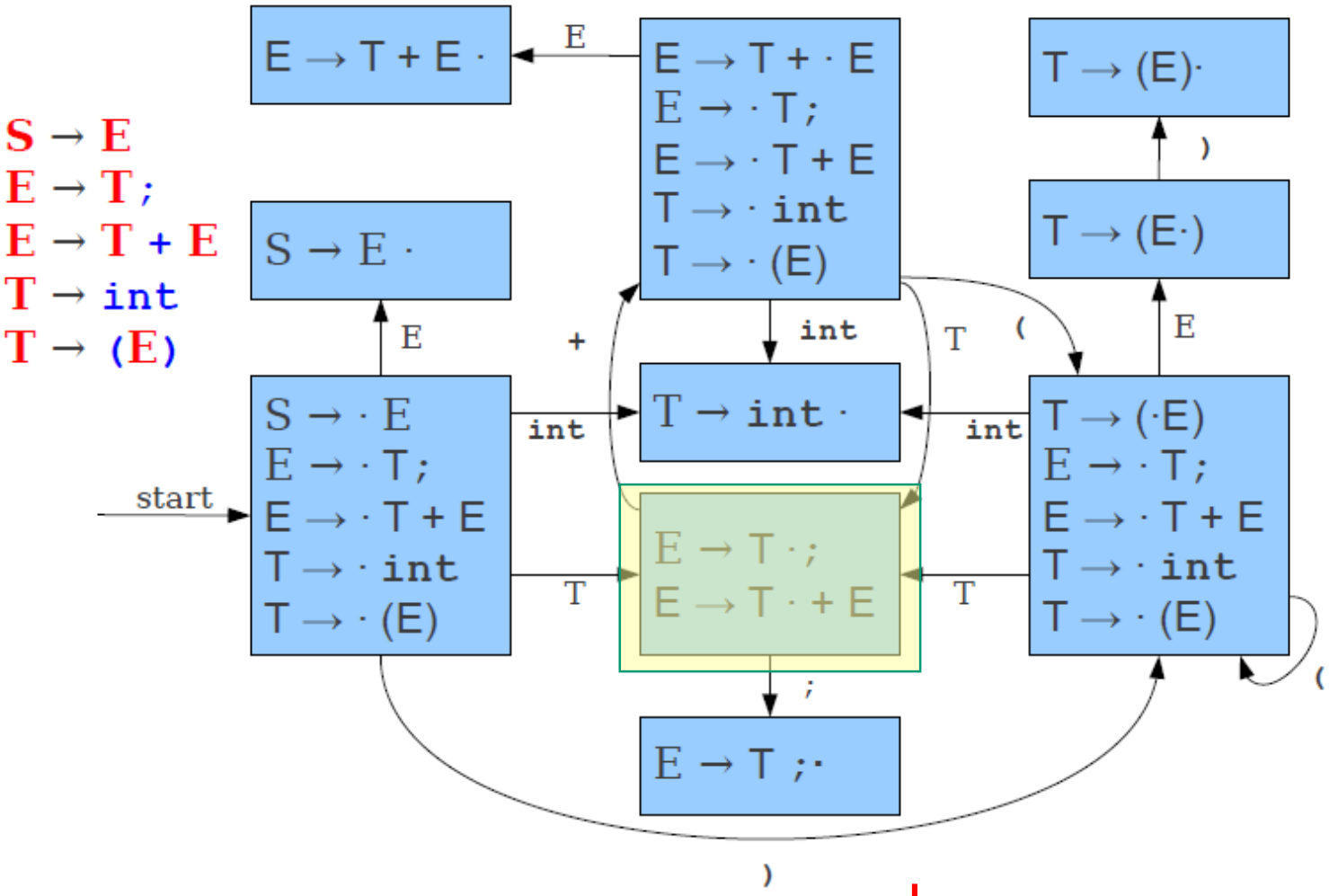

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
E
+
int
T
(
int
T → int ·
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(

T	+	(	T	+	int	;	)	;

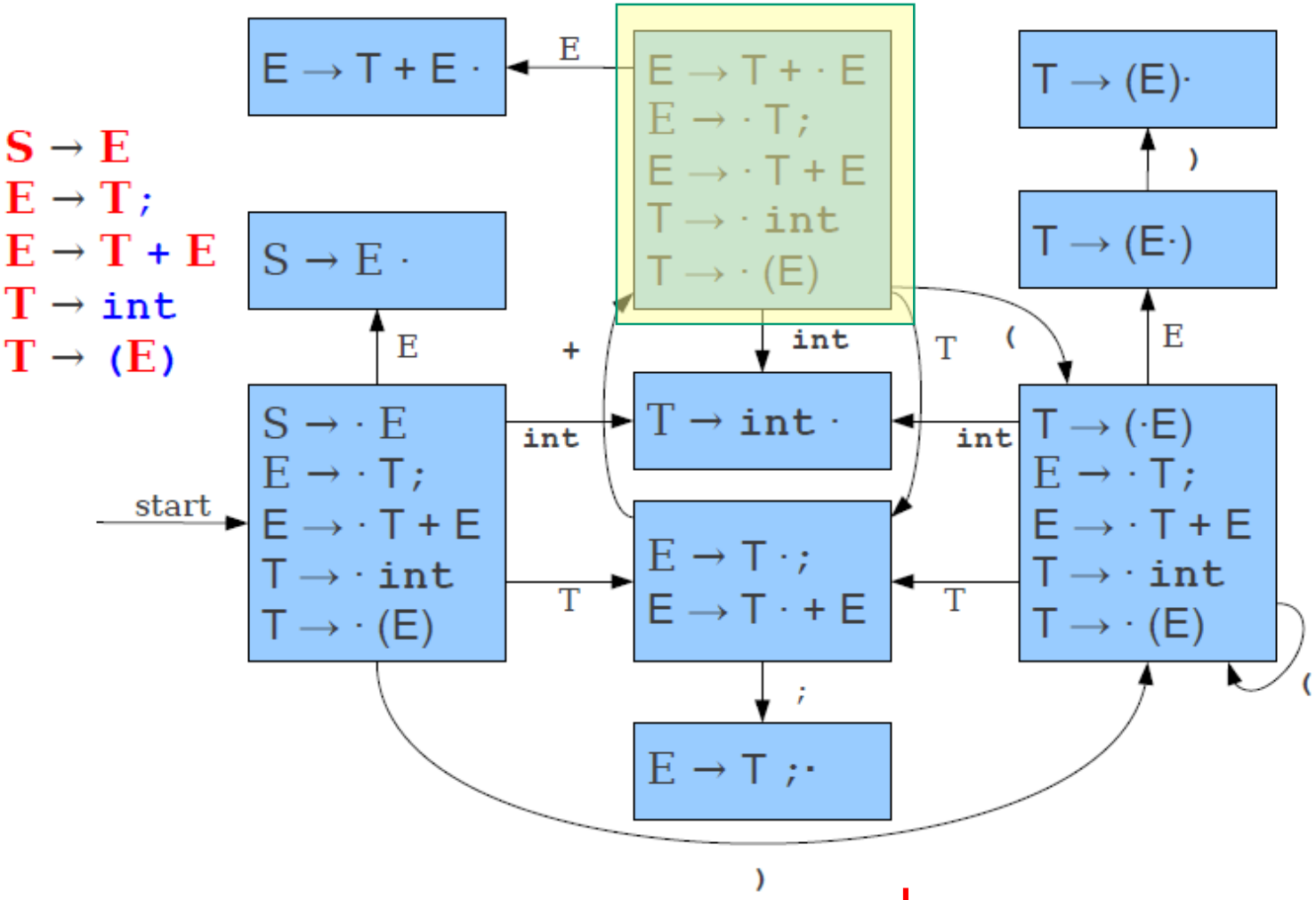

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(
T + (T + T ;) ;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E → T ·;
E → T · + E
E → T ;·
;
)
T + (T + T ;) ;

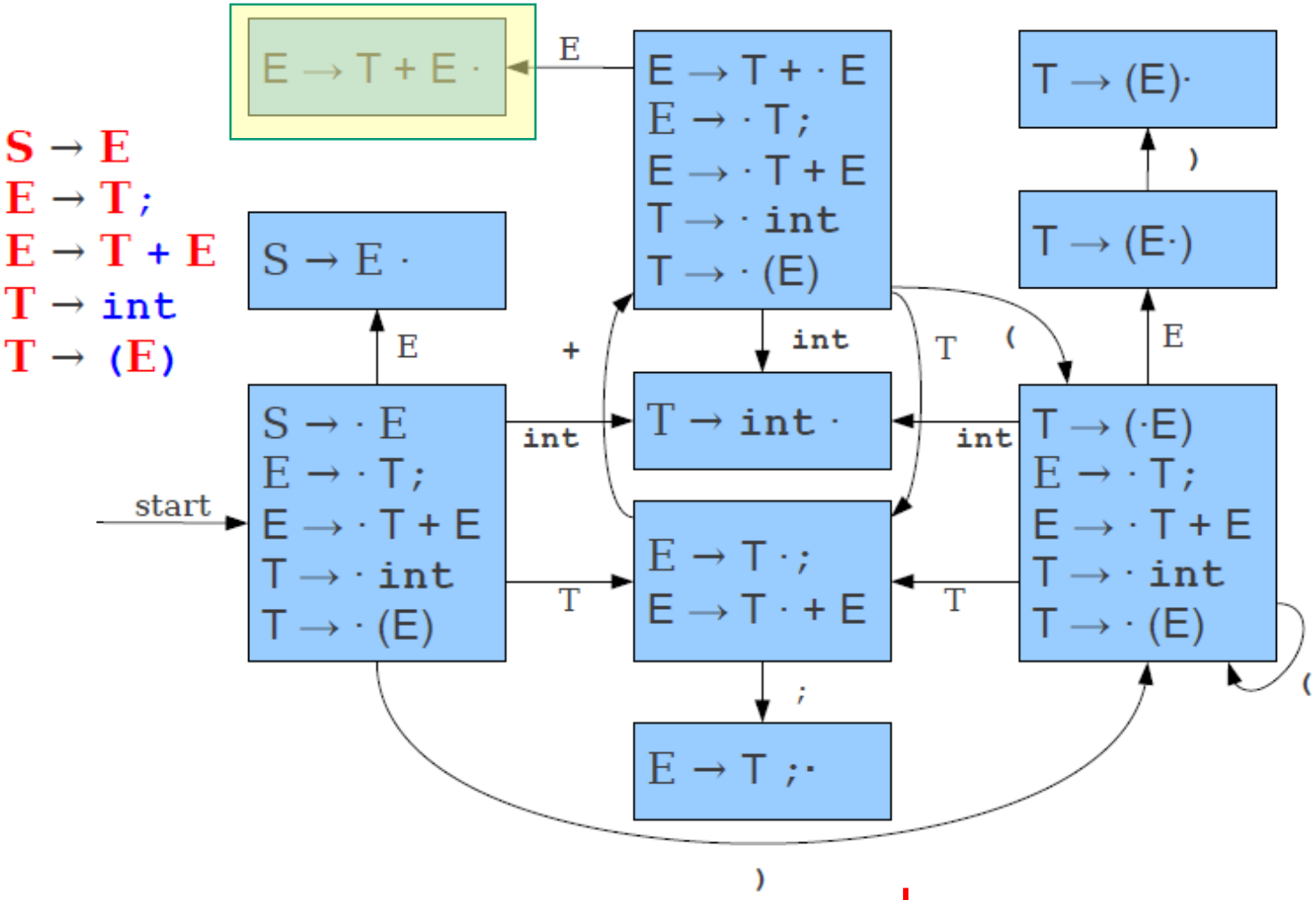

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(
T + (T + T ;) ;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → int ·
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
E → T ·;
E → T · + E
;
E → T ;·
)
(

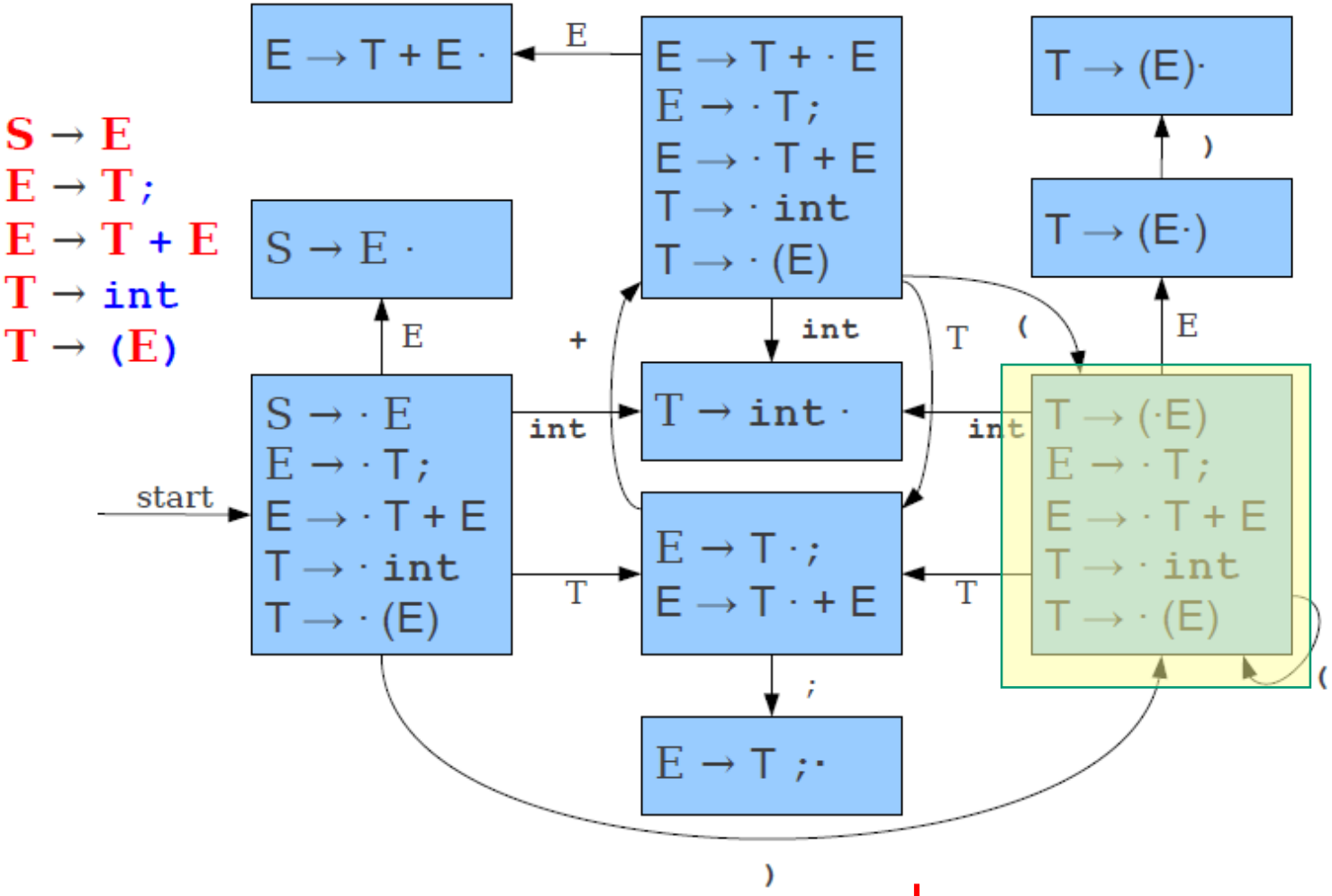
T	+	(	T	+	E	)	;

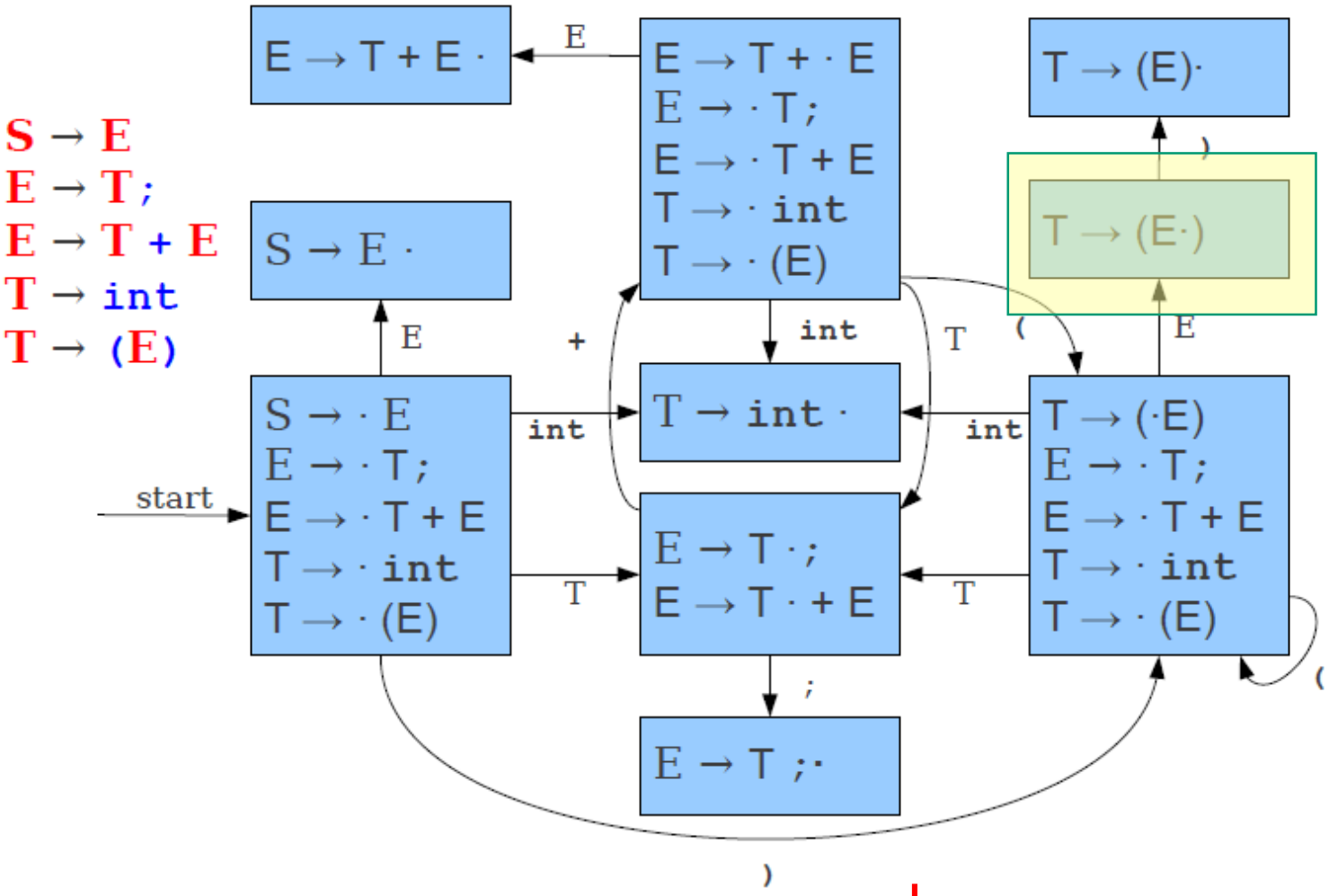

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(

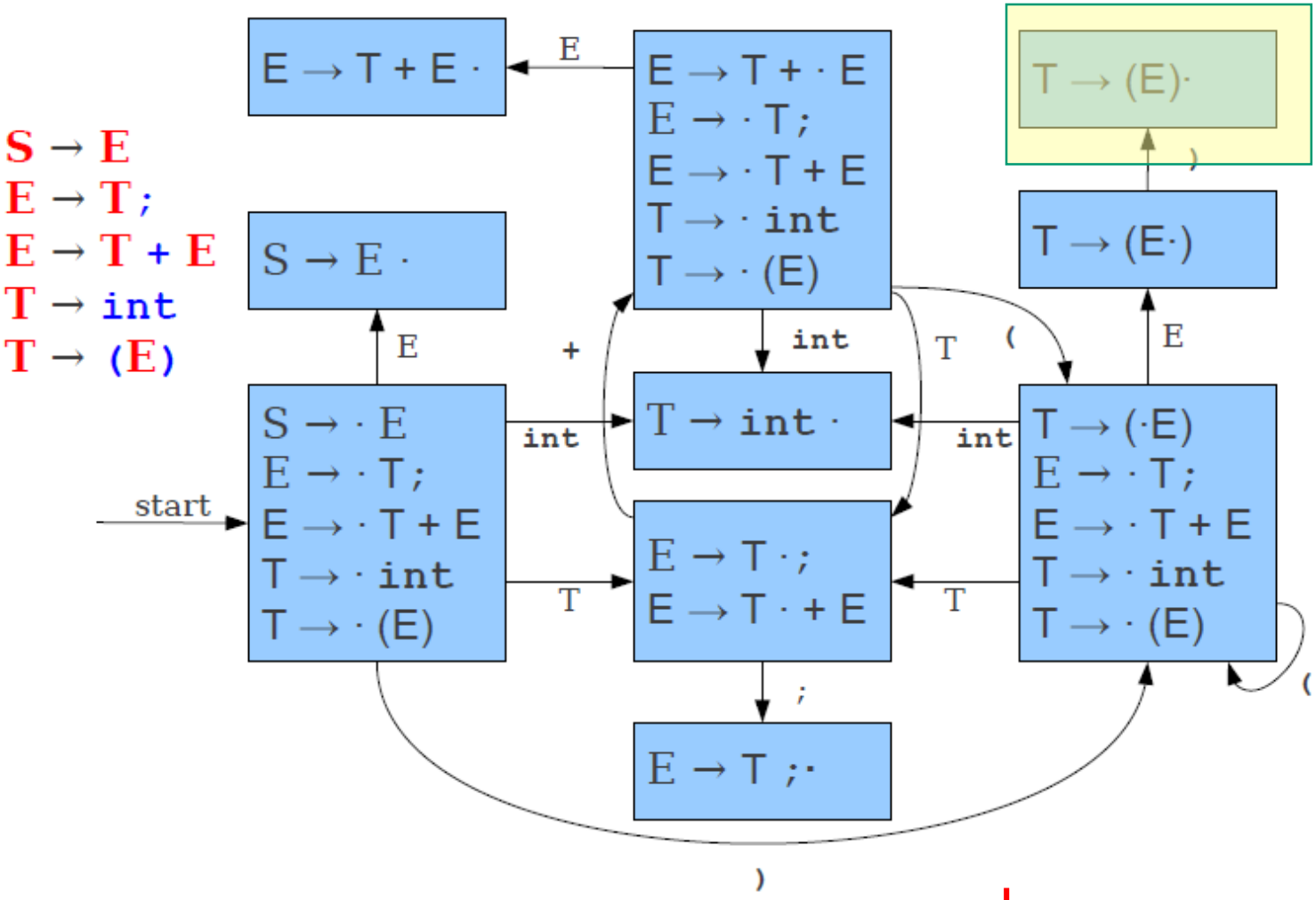
T	+	(	T	+	E	)	;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
S → E ·
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
+
int
T → int ·
int
T
E → T · ;
E → T · + E
T
;
E → T ; ·
)
T → (E)·
)
T → (E·)
E
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
(
int
T
(
E
(

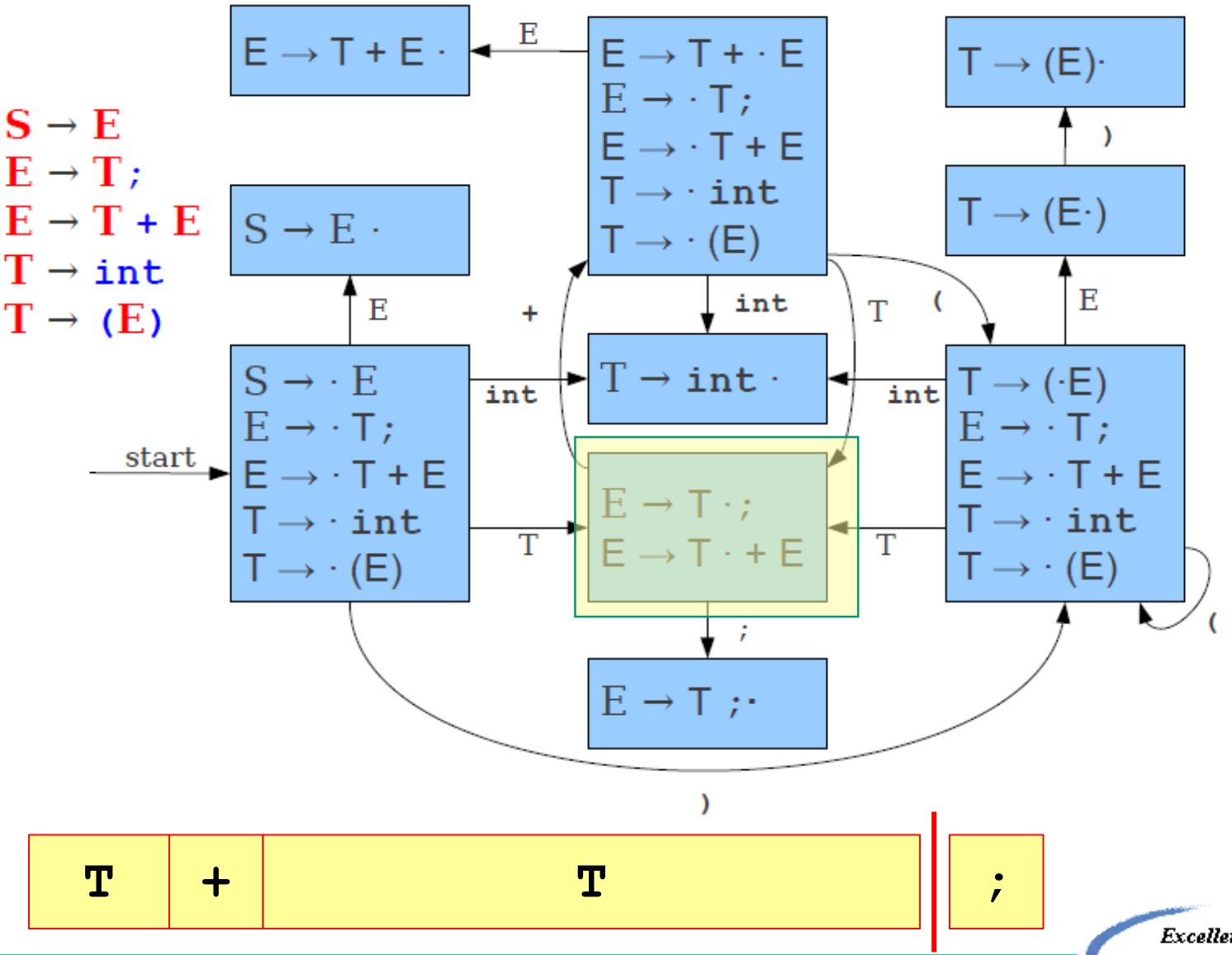
T	+	(	T	+	E	)	;

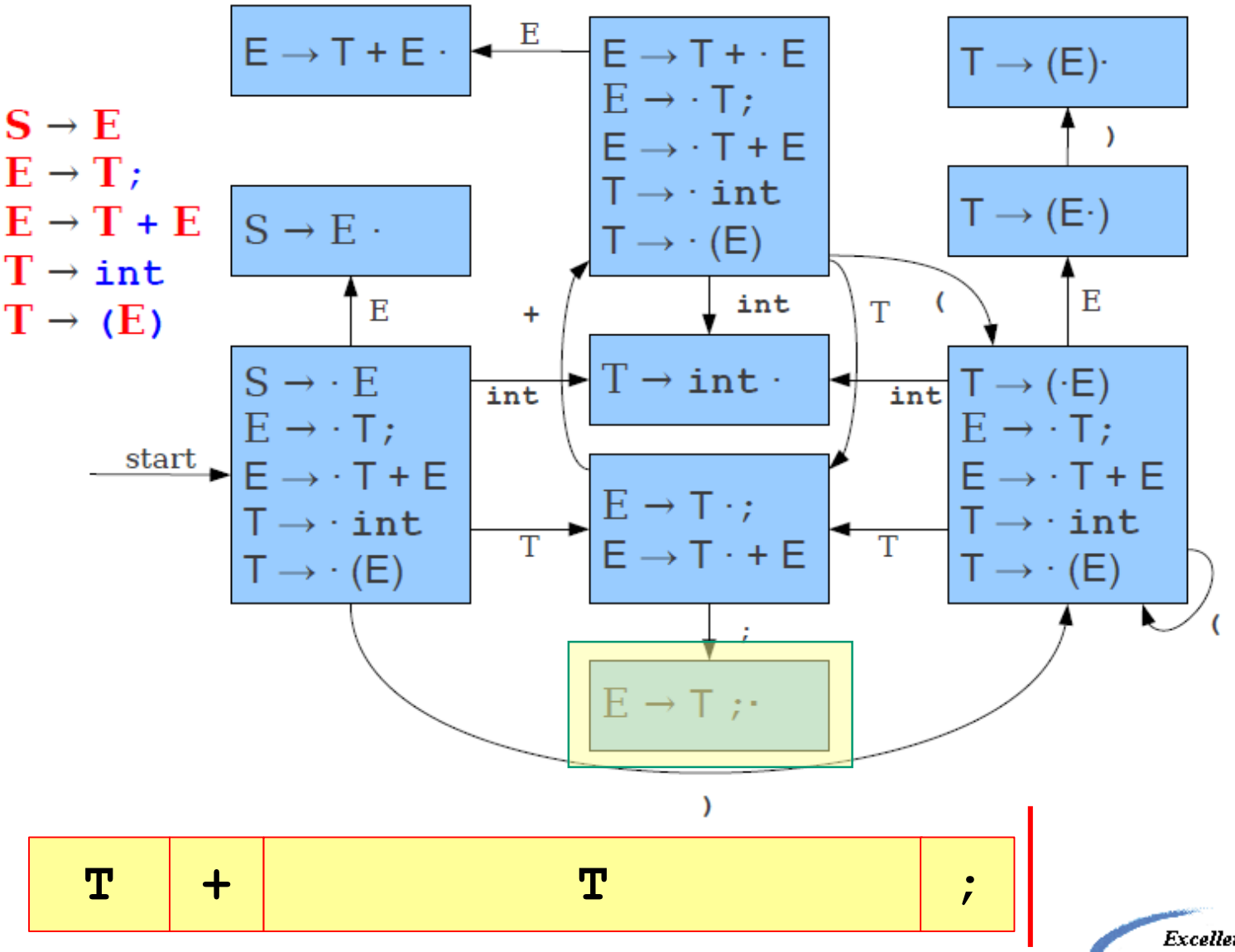

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
T
+
(
E
)
;

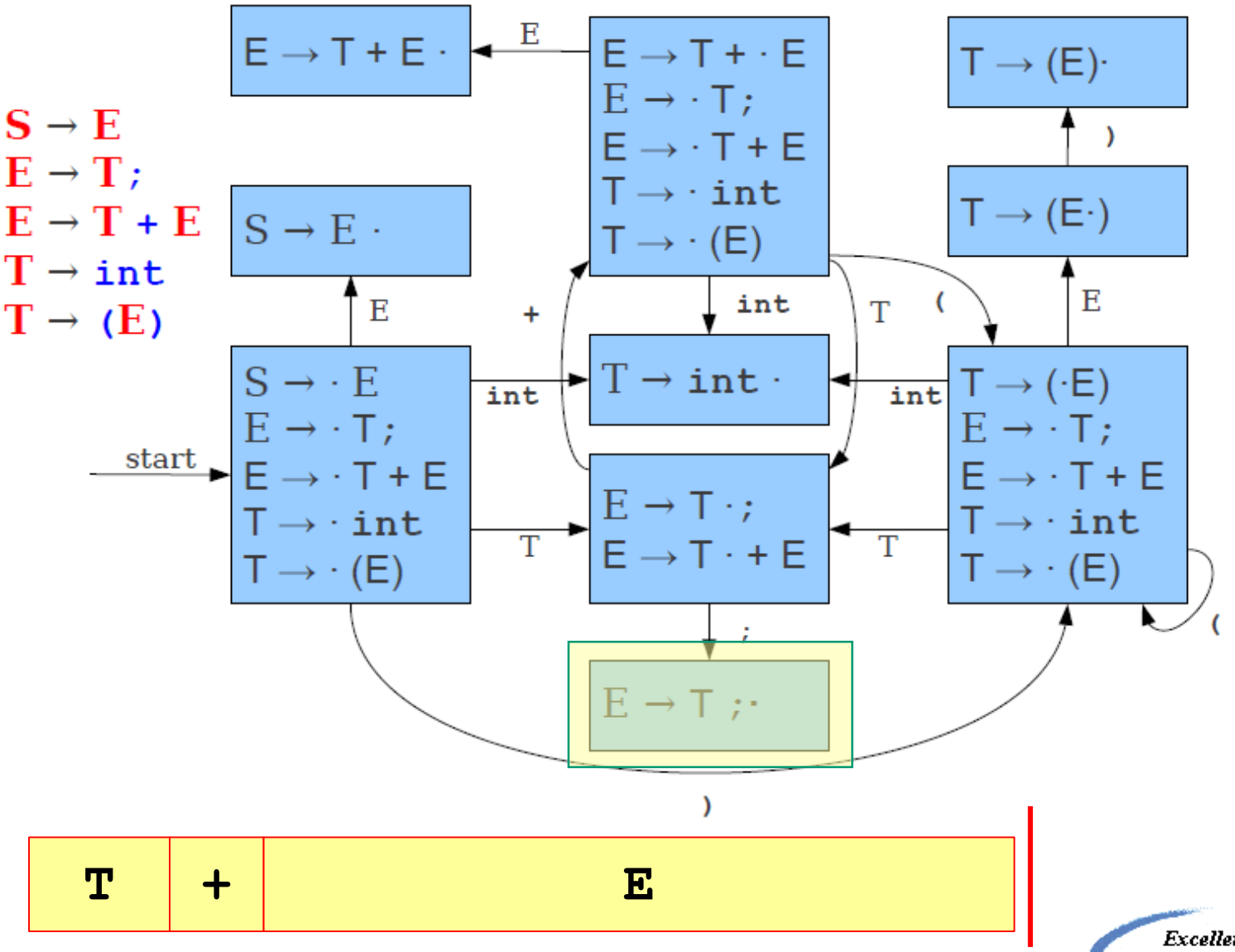

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
(
;
E → T ;·
)
T
+
(
E
)
;

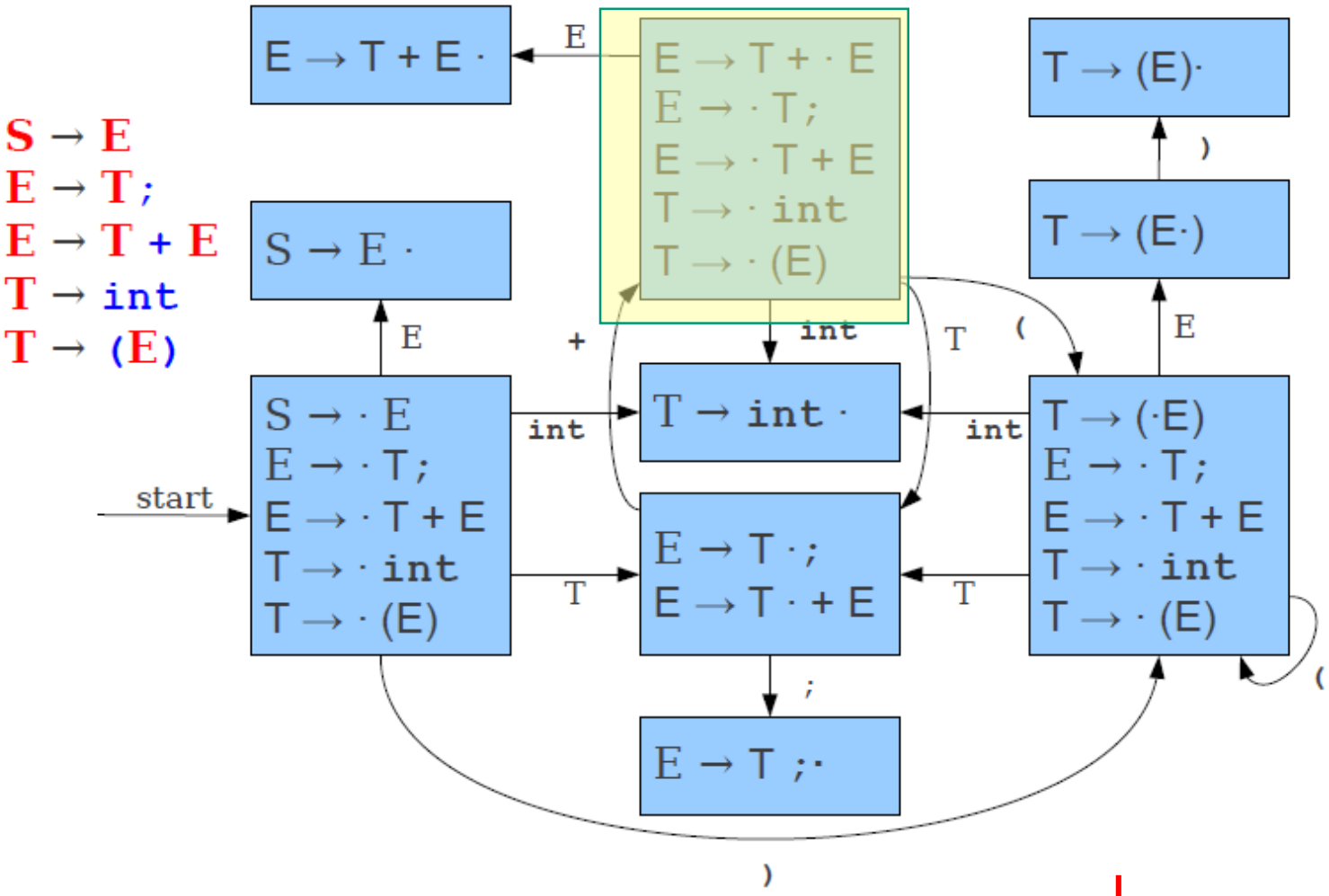

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
S → E ·
T → (E·)
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
(
;
E → T ;·
)

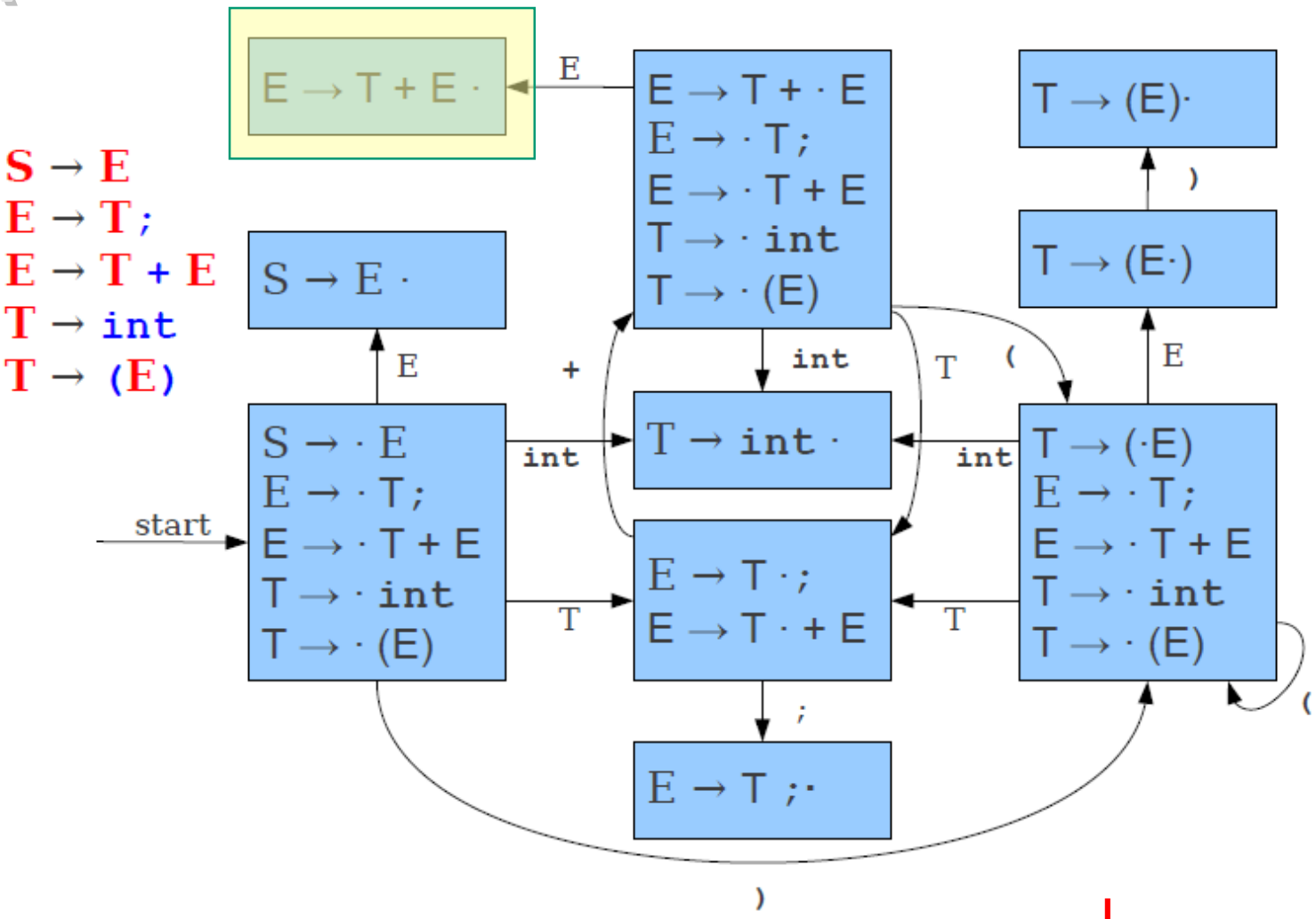
T	+	(	E	)	;

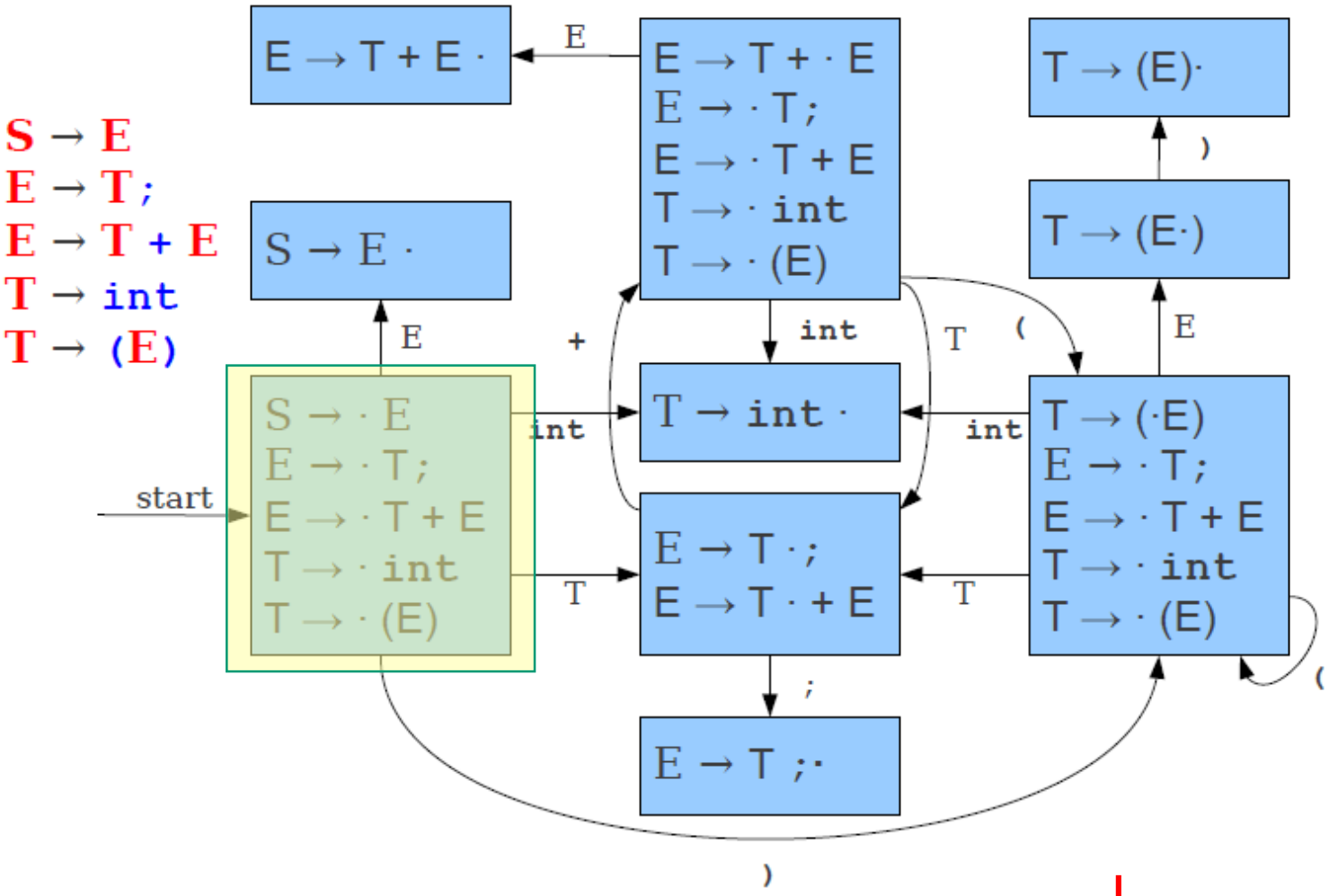

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(
T
+
T
;

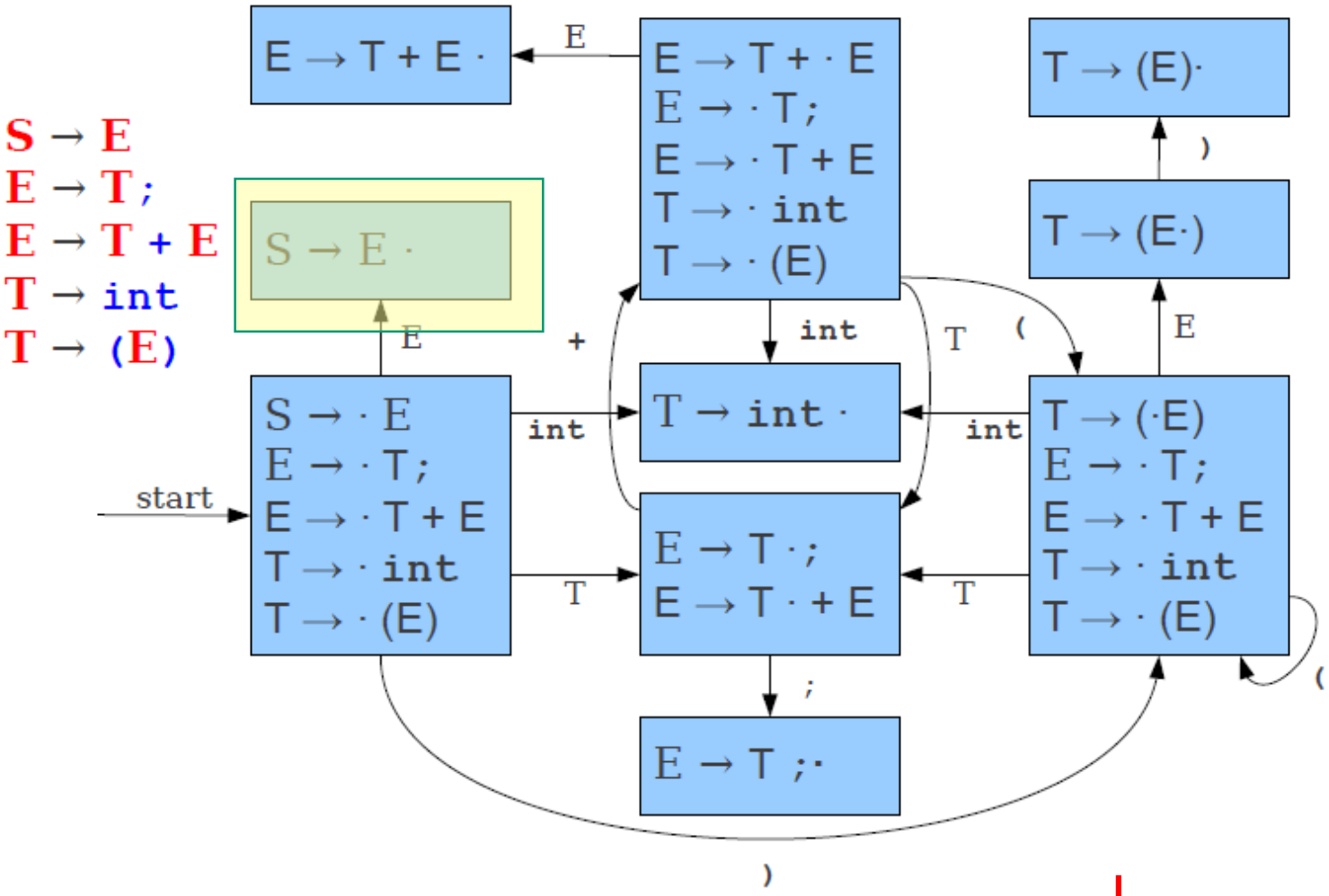

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(
T
+
T
;


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
int
T
(
E
+
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
start
T
E → T ·;
E → T · + E
T
;
E → T ;·
)
(
T
+
E


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(
T
+
E

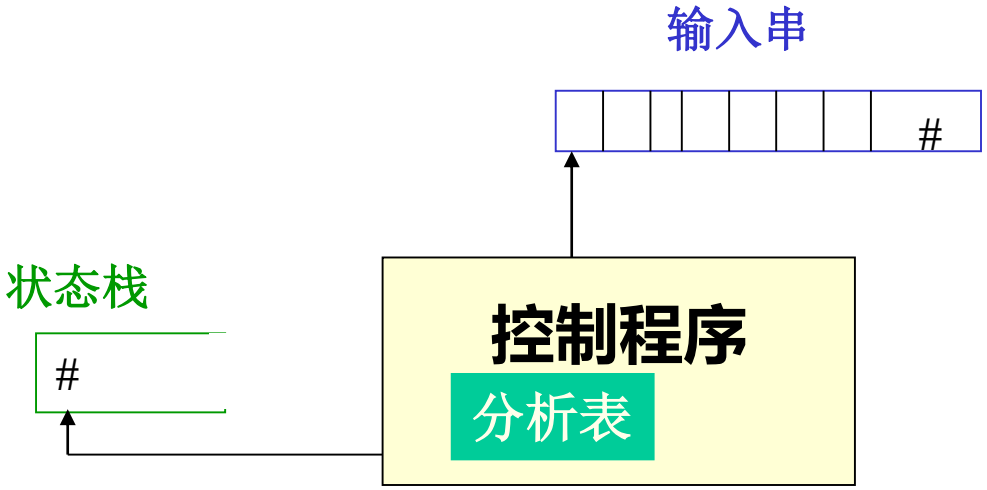

S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
)
(
E


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
S → E ·
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
T → (E·)
T → int ·
T → (·E)
E → T ·;
E → T · + E
E → T ;·
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
E
+
int
T
(
)
;
E
T
int
int
int
T
T
(
)
E


S → E
E → T;
E → T + E
T → int
T → (E)
E → T + E ·
E
E → T + · E
E → · T;
E → · T + E
T → · int
T → · (E)
T → (E)·
)
T → (E·)
S → E ·
E
+
int
T
(
E
S → · E
E → · T;
E → · T + E
T → · int
T → · (E)
start
int
T → int ·
int
T → (·E)
E → · T;
E → · T + E
T → · int
T → · (E)
T
T
E → T ·;
E → T · + E
;
E → T ;·
(
)
S

(2) LR分析器有三部分：状态栈　分析表　控制程序

状态栈：放置分析器状态和文法符号。

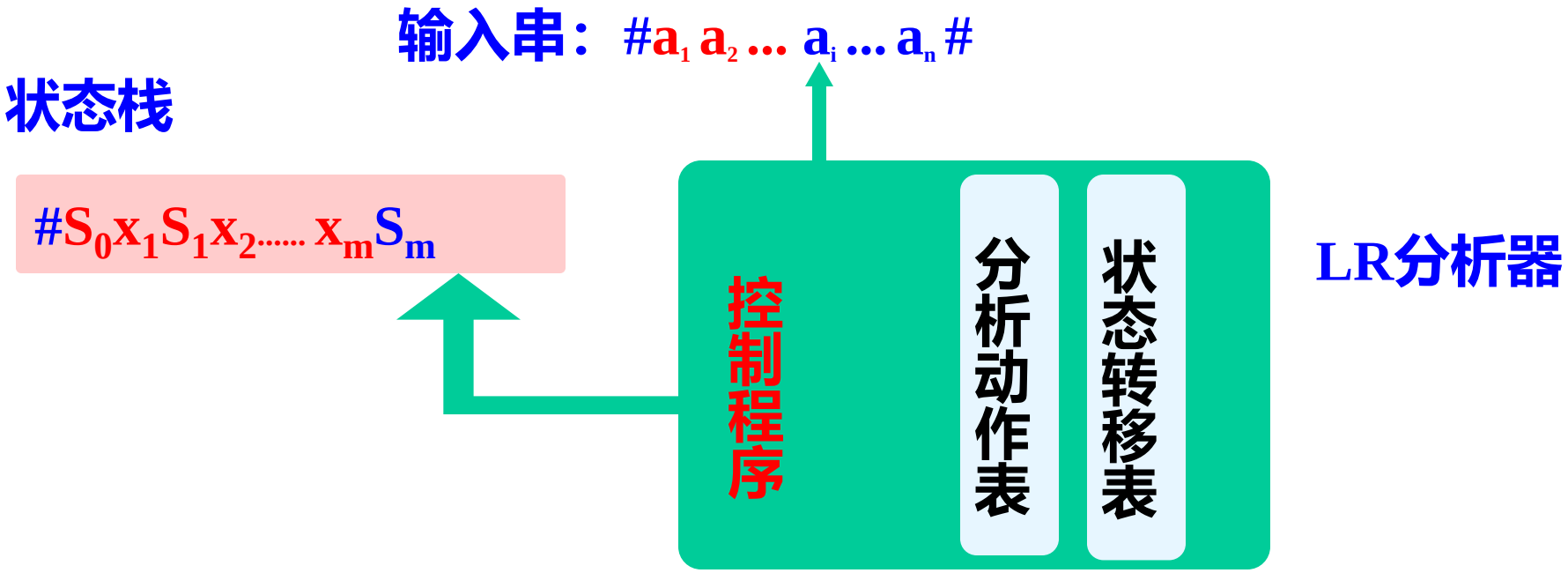


分析表：由两个矩阵组成，其功能是指示分析器的动作，是移进还是归约，根据不同的文法类要采用不同的构造方法。

控制程序：执行分析表所规定的动作，对栈进行操作。

LR分析的基本原理：带动作的有穷状态自动机

（1）逻辑结构



确定的有穷自动机（DFA）

(Deterministic Finite Automata)

一个确定的有穷自动机（DFA）M是一个五元式:

$$M=(S,\ \Sigma,\ \delta,\ s_0,\ Z)$$

其中:

1. S —有穷状态集
2. Σ —输入字母表
3. δ —映射函数(也称状态转换函数)

   $S\times\Sigma\rightarrow S$

   $\delta(s,a)=s'$, $s, s' \in S$, $a\in\Sigma$
4. s_0 —初始状态　　$s_0 \in S$
5. Z—终止状态　　$Z\subseteq S$

$\#S_0x_1S_1x_2\ldots\ldots x_mS_m$ → $S_0S_1\ldots\ldots S_m$ / $\# x_1x_2\ldots\ldots x_m$

状态栈: $S_0,S_1,\ldots,S_m$ 状态

S_0---初始状态　　S_m---栈顶状态

栈顶状态概括了从分析开始到该状态的全部**分析历史**和**展望信息**

符号串: $X_1X_2\ldots\ldots X_m$

为从开始状态(S_0)到当前状态(S_m)所识别的**规范句型的活前缀**。

规范句型: 通过**规范归约（Right-most）**得到的句型。

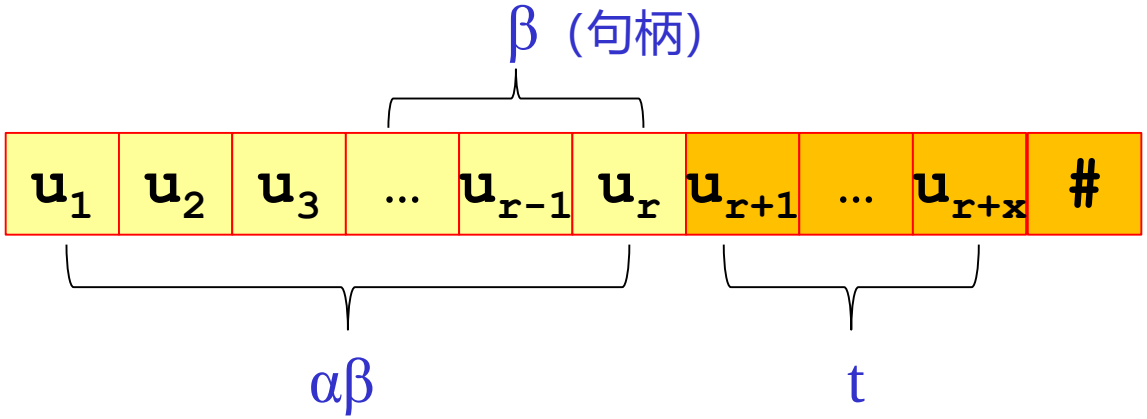
规范句型前缀: 将输入串的剩余部分与其连接起来就构成了规范句型。

如： $x_1x_2\ldots x_m a_i \ldots a_n$为规范句型（xi已处理，ai未处理）

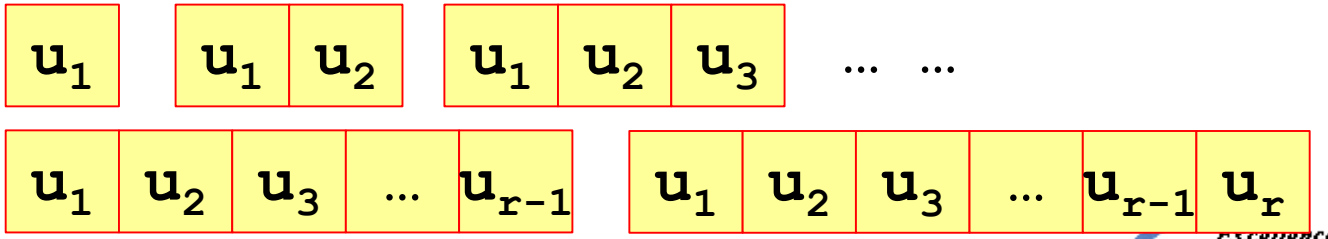
活前缀: 若分析过程能够保证栈中符号串均是规范句型的前缀，则表示输入串已分析过的部分没有语法错误，所以称为规范句型的活前缀。

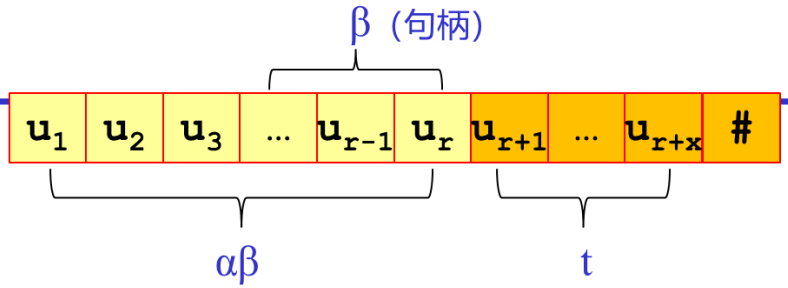
规范句型的活前缀:

对于句型αβt， β表示句柄, 如果αβ= $u_1u_2\ldots u_r$
那么符号串$u_1u_2\ldots u_i$ $(1\le i\le r)$ 即是句型αβt的**活前缀.**



句型αβt的**活前缀:**





规范句型的活前缀:

对于句型αβt， β表示句柄, 如果αβ= $u_1u_2\ldots u_r$
那么符号串$u_1u_2\ldots u_i$ $(1\le i\le r)$ 即是句型αβt的**活前缀.**

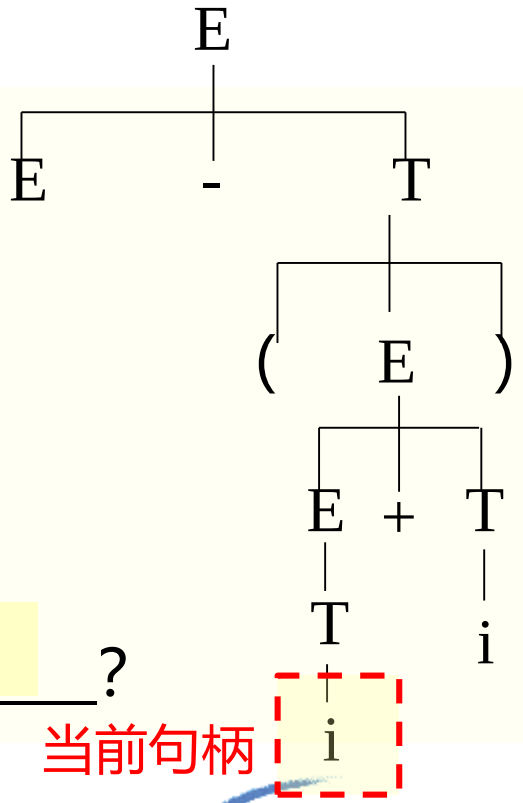
例: 有文法G[E]： E→T|E+T|E-T
T→i|(E)

拓广文法G′[S]： S→E#
E→T|E+T|E-T
T→i|(E)

当前句柄

句型 E-(i+i) #

当前句型的活前缀是 E, E-, E-(, E-(i ?



谢谢！