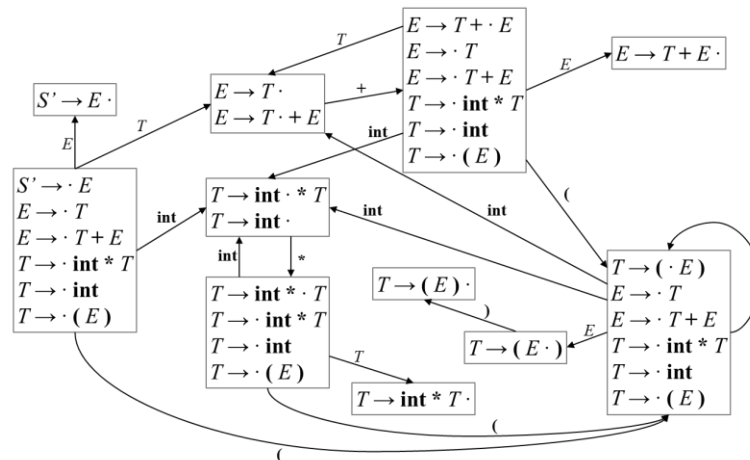


Top-Down and Bottom-Up Parsing



Recap: EBNF

- A formal and concise way to specify context-free grammars

Notation	Usage	Example
=	definition	letter = "A".."Z".
.	termination	letter = "A".."Z".
	alternation	letter = "A".."Z" "a".."z".
[...]	option	number = ["-"] digit.
{ ... }	repetition (≥ 0)	number = ["-"] digit {digit}.
(...)	grouping	factor = [unaryOp] (ident number).
"...", '...'	terminal symbol	"module", '''

- Example

```
letter  =  "A".."Z" | "a".."z".
digit   =  "0".."9".
ident   =  letter { letter | digit }.
number  =  digit {digit} [ [ "." digit {digit} ] [ "E" ["+"|"-"]
                        digit {digit} ] ].
```

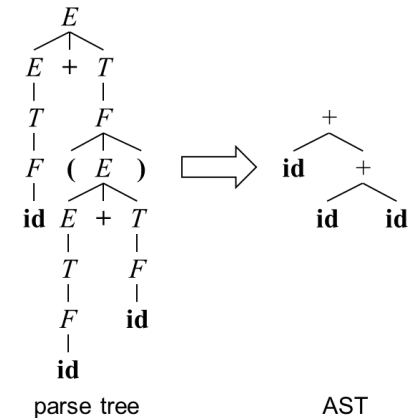
Recap: Error Handling

- Error Types
 - lexical, syntactical, semantical
- Error Handling:
 - panic mode
 - ▶ skip to synchronizing tokens; continue parsing
 - error productions
 - ▶ include frequent syntax errors in grammar
 - automatic error correction
 - ▶ try to fix the error automatically

Recap: Abstract Syntax Trees

- ASTs (Abstract Syntax Trees) are a common form of an intermediate representation in a parser

- represent the same information as the parse tree
- do not preserve the derivations



- Construction during parsing

- "syntax-directed translation"
 - ▶ add actions to each production rule

E	\rightarrow	$E + T$	<code>{ return new BinaryOp('+', E(), T()); }</code>	
		$ $	T	<code>{ return T(); }</code>
T	\rightarrow	$T * F$	<code>{ return new BinaryOp('*', T(), F()); }</code>	
		$ $	F	<code>{ return F(); }</code>
F	\rightarrow	(E)	<code>{ return E(); }</code>	
		$ $	id	<code>{ return new Ident(GetSymbol(id)); }</code>

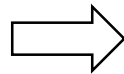
Top-Down Parsing

Top-Down Parsing

- Also called recursive descent parsing
- Recursive descent parsers use a set of recursive procedures to process the input left-to-right, top-to-bottom.
 - usually, each nonterminal is implemented by a separate procedure

$A \rightarrow aB \mid b$

$B \rightarrow c \mid dB$

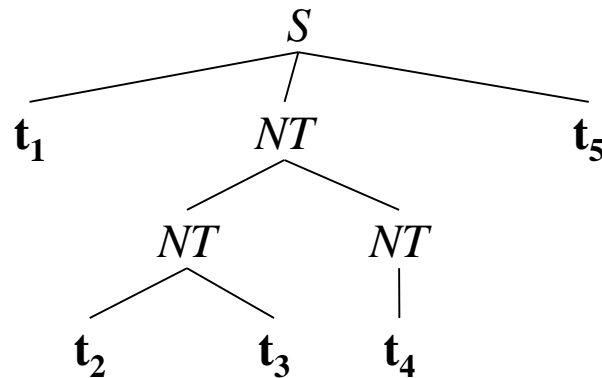


```
procedure A()  
{  
    Token t = GetNextToken();  
  
    if      (t == "a") B;  
    else if (t == "b") ;  
    else syntax_error(t);  
}  
  
procedure B()  
{  
    Token t = GetNextToken();  
  
    if      (t == "c") ;  
    else if (t == "d") B;  
    else syntax_error(t);  
}
```

Top-Down Parsing

- Implicitly constructs the parse tree
 - non-terminals: inner nodes
 - terminals: leaf nodes
- Terminals are seen in order of appearance in the token stream

$t_1 t_2 t_3 t_4 t_5$



Top-Down Parsing

- Typical procedure for a nonterminal in a recursive-descent parser
 - mimics the chosen production body by executing it

```
procedure A()  
{  
  choose an A-production,  $A \rightarrow X_1X_2...X_k$   
  
  for i = 1 to k {  
    if ( $X_i$  is a nonterminal) {  
      call  $X_i()$   
    }  
    else if ( $X_i$  matches next token in input stream) {  
      advance input to next token  
    }  
    else {  
      syntax error  
    }  
  }  
}
```


Top-Down Parsing

■ Example:

E	\rightarrow	TE'
E'	\rightarrow	$+ TE' \mid \varepsilon$
T	\rightarrow	FT'
T'	\rightarrow	$* FT' \mid \varepsilon$
F	\rightarrow	$(E) \mid \mathbf{id}$

id + id * id

Top-Down Parsing

- Key problem:

which production to choose?

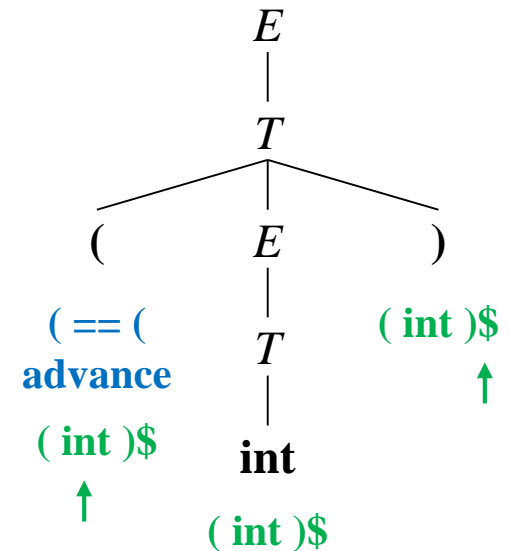
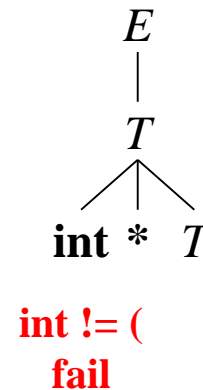
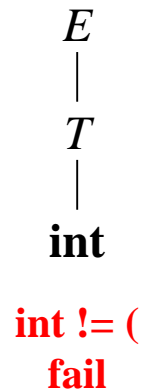
- Try production rules in order

- backtracking may be required

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

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

(int)\$
↑



Top-Down Parser Construction

■ Given

- token type `TOKEN`
- global pointer `next` pointing to the next input token
- helper function `term`

```
bool term(TOKEN t) {  
    return *next++ == t;  
}
```

■ Mechanically define procedures for nonterminals as follows

- i-th production of `S`:

```
bool Si() { ... }
```
- all production of `S`:

```
bool S() {  
    TOKEN *save = next;  
    return (next = save, S1())  
        || (next = save, S2())  
        ...  
        || (next = save, Sk()) ;  
}
```

Top-Down Parser Construction

- Mechanically define procedures for nonterminals as follows (cont'd)
 - i-th production of S:

$$S_i \rightarrow X_1 X_2 \dots X_k$$

if grammar symbol X_j is a

- ▶ nonterminal
invoke the respective procedure: $X_j()$
- ▶ terminal
match the input with the expected terminal: $\text{term}(X_j)$

then return the result of ANDing all grammar symbols in order

```
bool Si() {  
    return X1() && term(X2) && ... && Xk();  
}
```

Top-Down Parser Construction

- Parsing the input then consists of setting the next pointer to the first token and calling the start production:

```
Token *next;
```

```
void main()
{
    next = pointer to first token;
    if (S() && (*next == END_OF_INPUT)) {
        printf("parsed successfully.\n");
    } else {
        printf("parse error.\n");
    }
}
```

Top-Down Parser Construction

■ Example

Grammar

$$E \rightarrow T \mid T + E$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

```
TOKEN *next;
bool term(TOKEN t) { return *next++ == t; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {
    TOKEN *save = next;
    return (next = save, E1())
        || (next = save, E2());
}

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(LBRAK) && E() && term(RBRAK); }
bool T() {
    TOKEN *save = next;
    return (next = save, T1())
        || (next = save, T2())
        || (next = save, T3());
}
```

Left Recursion and Top-Down Parsers

- Left-recursive grammars will lead to endless recursion for certain input strings in recursive-descent parsers

- consider the grammar

$E \rightarrow E + \text{int} / \text{int}$

its recursive-descent parser

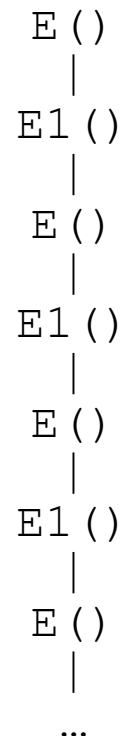
```
TOKEN *next;
bool term(TOKEN t) { return *next++ == t; }

bool E1() { return E() && term(PLUS) && term(INT); }
bool E2() { return term(INT); }
bool E() {
    TOKEN *save = next;
    return (next = save, E1())
        || (next = save, E2());
}
```

and the input string

1 + 2
↑

Callgraph:



Predictive Parsing

Predictive Parsing

- Predictive parsing is a special case of recursive-descent parsing that requires no backtracking
 - choose (correct) production simply by looking at the k next input symbols
 - typically, we choose $k = 1$, i.e., a one-token lookahead.
 - big advantage: no backtracking
 - accepts grammars in the class $LL(k)$
 - ▶ L : left-to-right
 - ▶ L : leftmost derivation
 - ▶ k : amount of lookahead symbols
 - parse tables for predictive parsers can be generated automatically, using two functions, FIRST and FOLLOW

Predictive Parsing

- For a grammar G in LL(1) at each step and for any string in $L(G)$ there is only one choice of production

$\omega A \beta$ with A being the first nonterminal

with input t , there can be *at most* one production starting with symbol t .

$A \rightarrow \alpha$

choose production $A \rightarrow \alpha$

$\omega \alpha \beta$

Predictive Parsing

■ Example:

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

$$T \rightarrow \mathbf{int} \mid \mathbf{int} * T \mid (E)$$

- cannot use a predictive parser for this grammar
 - ▶ on nonterminal T with input **int**: two matching rules
 - ▶ for E : how do we predict which rule to use?
- left-factor grammar G first (see lecture Syntax Analysis – Context-Free Grammars)

$$E \rightarrow TE'$$

$$E' \rightarrow + E \mid \varepsilon$$

$$T \rightarrow \mathbf{int} T' \mid (E)$$

$$T' \rightarrow * T \mid \varepsilon$$

Parsing Table for Predictive Parsing

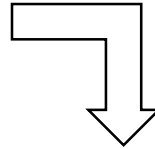
- similar to DFAs for regular expressions, we can construct a parsing table for LL(1) grammars that guide the parser
 - show derivation for each non-terminal and input symbol

$$E \rightarrow TE'$$

$$E' \rightarrow +E \mid \varepsilon$$

$$T \rightarrow \text{int } T' \mid (E)$$

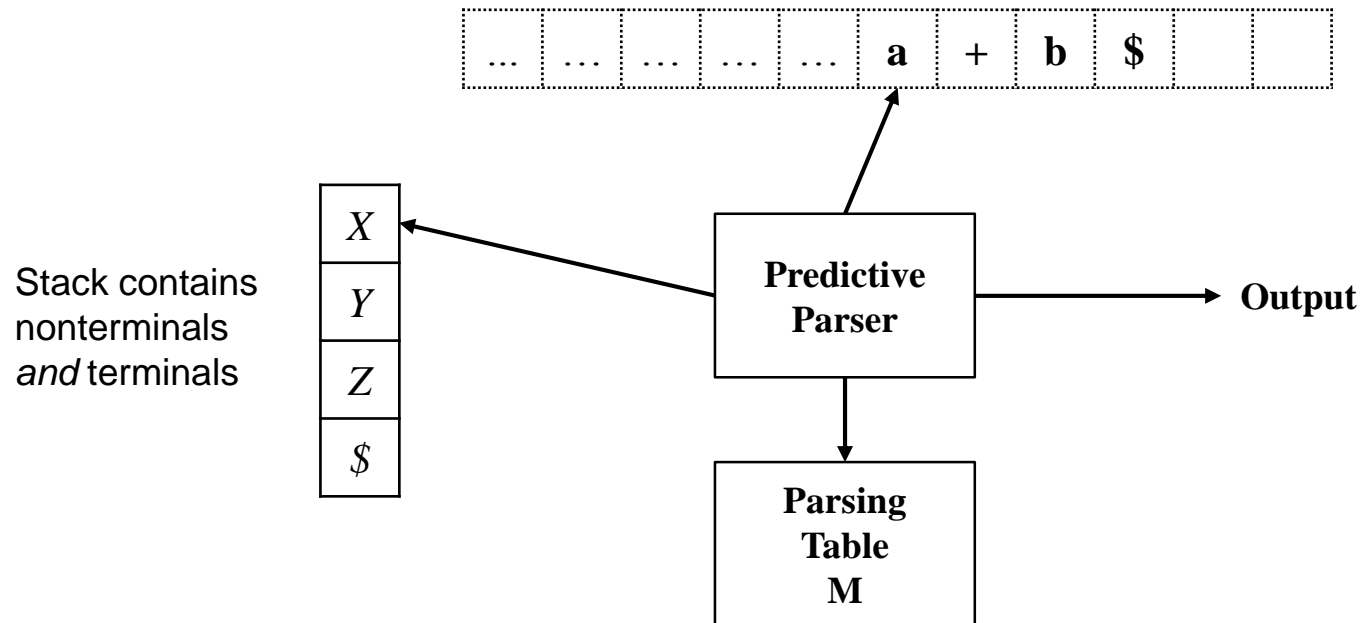
$$T' \rightarrow *T \mid \varepsilon$$



	int	+	*	()	\$
<i>E</i>	<i>TE'</i>			<i>TE'</i>		
<i>E'</i>		<i>+ E</i>			ε	ε
<i>T</i>	int <i>T'</i>			<i>(E)</i>		
<i>T'</i>		ε	<i>* T</i>		ε	ε

Predictive Parsing using a Parse Table

- Non-recursive implementation
 - instead, maintain a stack explicitly



- operation: for the leftmost non-terminal X and the next input symbol a choose production $M[X, a]$
- accept on end of input and empty stack

Predictive Parsing using a Parse Table

■ Predictive Parsing Algorithm

```
set next to first token in input stream
initialize stack = <S $>

while (stack != < >) do begin
    X = top of stack

    if (X is a terminal) then begin

        if (X == *next++) pop stack;
        else error();

    end else begin // X is a non-terminal

        if M[X,*next] = Y1...Yk then begin
            pop stack;
            push Yk...Y1 onto the stack (Y1 on top);
        end else error();

    end
end
```

Predictive Parsing using a Parse Table

■ Example: $\text{int} * \text{int}$

	int	+	*	()	\$
E	TE'			TE'		
E'		$+E$			ε	ε
T	$\text{int } T'$			(E)		
T'		ε	$*T$		ε	ε

Stack	Input	Action
$E\$$	$\text{int} * \text{int} \$$	TE'
$TE' \$$	$\text{int} * \text{int} \$$	$\text{int } T'$
$\text{int } T'E' \$$	$\text{int} * \text{int} \$$	terminal
$T'E' \$$	$* \text{int} \$$	$*T'$
$*TE' \$$	$* \text{int} \$$	terminal
$TE' \$$	$\text{int} \$$	$\text{int } T'$
$\text{int } T'E' \$$	$\text{int} \$$	terminal
$T'E' \$$	$\$$	ε
$E' \$$	$\$$	ε
$\$$	$\$$	ACCEPT

Generating Parse Tables for Predictive Parsers

Parse Table Generation

- The parse table for a non-terminal A , a production $A \rightarrow \alpha$, and a token t should contain the entry

$$M[A, t] = \alpha$$

in two cases:

1. α can derive t in the first position
 - ▶ $\alpha \Rightarrow^* t\beta$
 - ▶ $t \in \text{FIRST}(\alpha)$
2. α can be derived to ε , and t is a possible follower of α
 - ▶ $A \rightarrow \alpha$, $\alpha \Rightarrow^* \varepsilon$, and $S \Rightarrow^* \beta A t \delta$
 - ▶ “getting rid of A ”
 - top of stack is A , input is t , but A cannot derive t
 - ▶ $t \in \text{FOLLOW}(A)$

FIRST and FOLLOW

- FIRST and FOLLOW help in choosing the correct production based on the next input symbols
 - FOLLOW can also be used as synchronizing tokens during panic-mode recovery
 - FIRST: terminals produced by A
 - FOLLOW: terminals that may follow A

FIRST

■ FIRST(X)

FIRST(X), for any string of grammar symbols X , is the set of terminals that begin strings derived from X . If $X \Rightarrow^* \varepsilon$, then ε is also in FIRST(X).

$$\text{FIRST}(X) = \{ \mathbf{t} \mid X \Rightarrow^* \mathbf{t}\alpha \} \cup \{ \varepsilon \mid X \Rightarrow^* \varepsilon \}$$

■ Computing FIRST(X) by applying the following rules

- X is a terminal $\rightarrow \text{FIRST}(X) = \{ X \}$.
- $\varepsilon \in \text{FIRST}(X)$
 - ▶ $X \rightarrow \varepsilon$ is a production
 - ▶ $X \rightarrow Y_1 Y_2 \dots Y_k$ and $\varepsilon \in \text{FIRST}(Y_i)$ for $\forall i : 1 \leq i \leq k$
- $\text{FIRST}(\alpha) \subseteq \text{FIRST}(X)$ if $X \rightarrow Y_1 Y_2 \dots Y_k \alpha$ and $\varepsilon \in \text{FIRST}(Y_i)$ for $\forall i : 1 \leq i \leq k$

FIRST

■ Example:

● Grammar

$$E \rightarrow TE'$$

$$E' \rightarrow +E \mid \varepsilon$$

$$T \rightarrow \text{int } T' \mid (E)$$

$$T' \rightarrow *T \mid \varepsilon$$

● terminals

$$\text{FIRST}(+) = \{ + \}$$

$$\text{FIRST}(*) = \{ * \}$$

$$\text{FIRST}(()) = \{ (\}$$

$$\text{FIRST}()) = \{) \}$$

$$\text{FIRST}(\text{int}) = \{ \text{int} \}$$

● nonterminals

$$\text{FIRST}(E) \supseteq \text{FIRST}(T)$$

$$\text{FIRST}(T) = \{ (, \text{int} \}$$

$$\text{no } \varepsilon \rightarrow \text{FIRST}(E) = \text{FIRST}(T)$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

FOLLOW

■ FOLLOW(X)

FOLLOW(X), for nonterminal X , to be the set of terminals a that can appear immediately to the right of X in some sentential form; i.e., the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α and β .

$$\text{FOLLOW}(X) = \{ t \mid S \Rightarrow^* \beta X t \delta \}$$

■ Computing FOLLOW(X) (intuition)

- if $X \rightarrow AB$ then
 - ▶ $\text{FIRST}(B) \subseteq \text{FOLLOW}(A)$ and
 - ▶ $\text{FOLLOW}(X) \subseteq \text{FOLLOW}(B)$
 - if $B \Rightarrow^* \varepsilon$ then $\text{FOLLOW}(X) \subseteq \text{FOLLOW}(A)$
- if S is the start symbol then $\$ \in \text{FOLLOW}(S)$

FOLLOW

- Algorithm to compute FOLLOW()
 - $\$ \in \text{FOLLOW}(S)$
 - $\text{FIRST}(\beta) - \{ \varepsilon \} \subseteq \text{FOLLOW}(X)$
for each $A \rightarrow \alpha X \beta$
 - $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$
for each $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{FIRST}(\beta)$

FOLLOW

■ Example:

● Grammar

$$E \rightarrow TE'$$
$$E' \rightarrow +E \mid \varepsilon$$
$$T \rightarrow \text{int } T' \mid (E)$$
$$T' \rightarrow *T \mid \varepsilon$$

● relations

$$\begin{aligned} \text{FOLLOW}(E') &\subseteq \text{FOLLOW}(E) \\ \text{FOLLOW}(E) &\subseteq \text{FOLLOW}(E') \\ \rightarrow \text{FOLLOW}(E) &= \text{FOLLOW}(E') \end{aligned}$$
$$\text{FOLLOW}(E) \subseteq \text{FOLLOW}(T)$$
$$\begin{aligned} \text{FOLLOW}(T') &\subseteq \text{FOLLOW}(T) \\ \text{FOLLOW}(T) &\subseteq \text{FOLLOW}(T') \\ \rightarrow \text{FOLLOW}(T) &= \text{FOLLOW}(T') \end{aligned}$$

● FOLLOW sets

$$\text{FOLLOW}(E) = \{ \$,) \}$$
$$\text{FOLLOW}(E') = \{ \$,) \}$$
$$\text{FOLLOW}(T) = \{ +, \$,) \}$$
$$\text{FOLLOW}(T') = \{ +, \$,) \}$$
$$\text{FOLLOW}(() = \{ \text{int}, (\}$$
$$\text{FOLLOW}()) = \{ +, \$,) \}$$
$$\text{FOLLOW}(+) = \{ \text{int}, (\}$$
$$\text{FOLLOW}(*) = \{ \text{int}, (\}$$
$$\text{FOLLOW}(\text{int}) = \{ *, +, \$,) \}$$

Construct LL(1) Parsing Tables

- For each production $A \rightarrow \alpha$ of the grammar

- for each terminal \mathbf{a} in $\text{FIRST}(A)$, set

$$M[A, \mathbf{a}] = \alpha$$

- if ε is in $\text{FIRST}(\alpha)$, then for each terminal \mathbf{b} in $\text{FOLLOW}(A)$, set

$$M[A, \mathbf{b}] = \alpha$$

- if ε is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add

$$M[A, \$] = \alpha$$

as well.

Construct LL(1) Parsing Tables

■ Example

$$E \rightarrow TE'$$

$$E' \rightarrow +E \mid \varepsilon$$

$$T \rightarrow \text{int } T' \mid (E)$$

$$T' \rightarrow *T \mid \varepsilon$$

$$\text{FIRST}(E) = \{ (, \text{int} \}$$

$$\text{FIRST}(T) = \{ (, \text{int} \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FIRST}(+) = \{ + \}$$

$$\text{FIRST}(*) = \{ * \}$$

$$\text{FIRST}(() = \{ (\}$$

$$\text{FIRST}()) = \{) \}$$

$$\text{FIRST}(\text{int}) = \{ \text{int} \}$$

$$\text{FOLLOW}(E) = \{ \$,) \}$$

$$\text{FOLLOW}(E') = \{ \$,) \}$$

$$\text{FOLLOW}(T) = \{ +, \$,) \}$$

$$\text{FOLLOW}(T') = \{ +, \$,) \}$$

$$\text{FOLLOW}(() = \{ \text{int}, (\}$$

$$\text{FOLLOW}()) = \{ +, \$,) \}$$

$$\text{FOLLOW}(+) = \{ \text{int}, (\}$$

$$\text{FOLLOW}(*) = \{ \text{int}, (\}$$

$$\text{FOLLOW}(\text{int}) = \{ *, +, \$,) \}$$

	int	+	*	()	\$
<i>E</i>						
<i>E'</i>						
<i>T</i>						
<i>T'</i>						

Construct LL(1) Parsing Tables

■ Example

$$E \rightarrow TE'$$

$$E' \rightarrow +E \mid \varepsilon$$

$$T \rightarrow \text{int } T' \mid (E)$$

$$T' \rightarrow *T \mid \varepsilon$$

$$\text{FIRST}(E) = \{ (, \text{int} \}$$

$$\text{FIRST}(T) = \{ (, \text{int} \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FIRST}(+) = \{ + \}$$

$$\text{FIRST}(*) = \{ * \}$$

$$\text{FIRST}(() = \{ (\}$$

$$\text{FIRST}() = \{) \}$$

$$\text{FIRST}(\text{int}) = \{ \text{int} \}$$

$$\text{FOLLOW}(E) = \{ \$,) \}$$

$$\text{FOLLOW}(E') = \{ \$,) \}$$

$$\text{FOLLOW}(T) = \{ +, \$,) \}$$

$$\text{FOLLOW}(T') = \{ +, \$,) \}$$

$$\text{FOLLOW}(() = \{ \text{int}, (\}$$

$$\text{FOLLOW}() = \{ +, \$,) \}$$

$$\text{FOLLOW}(+) = \{ \text{int}, (\}$$

$$\text{FOLLOW}(*) = \{ \text{int}, (\}$$

$$\text{FOLLOW}(\text{int}) = \{ *, +, \$,) \}$$

	int	+	*	()	\$
<i>E</i>	<i>TE'</i>			<i>TE'</i>		
<i>E'</i>		<i>+ E</i>			ε	ε
<i>T</i>	int <i>T'</i>			(<i>E</i>)		
<i>T'</i>		ε	<i>* T</i>		ε	ε

Bottom-Up Parsing

Introduction

■ Bottom-Up Parsing

- construct the parse tree beginning at the leaves, working upwards towards the root
- more general than top-down parsing
- just as efficient

Introduction

■ Grammars used throughout this chapter

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

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \mathbf{int}$$

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

$$T \rightarrow \mathbf{int} * T \mid \mathbf{int} / (E)$$

- observation: the left grammar is left-recursive
 - ▶ left-recursion poses no problem for bottom-up parsers
 - ▶ allows for a more natural representation compared to the left-factored version:

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \varepsilon$$

$$F \rightarrow (E) \mid \mathbf{int}$$

Introduction

■ Example:

int * int + int

***F* * int + int**

***T* * int + int**

***T* * *F* + int**

***T* + int**

T* + *F

T* + *T

T* + *E

E

***F* → int**

T* → *F

***F* → int**

T* → *T* * *F

***F* → int**

T* → *F

E* → *T

E* → *T* + *E

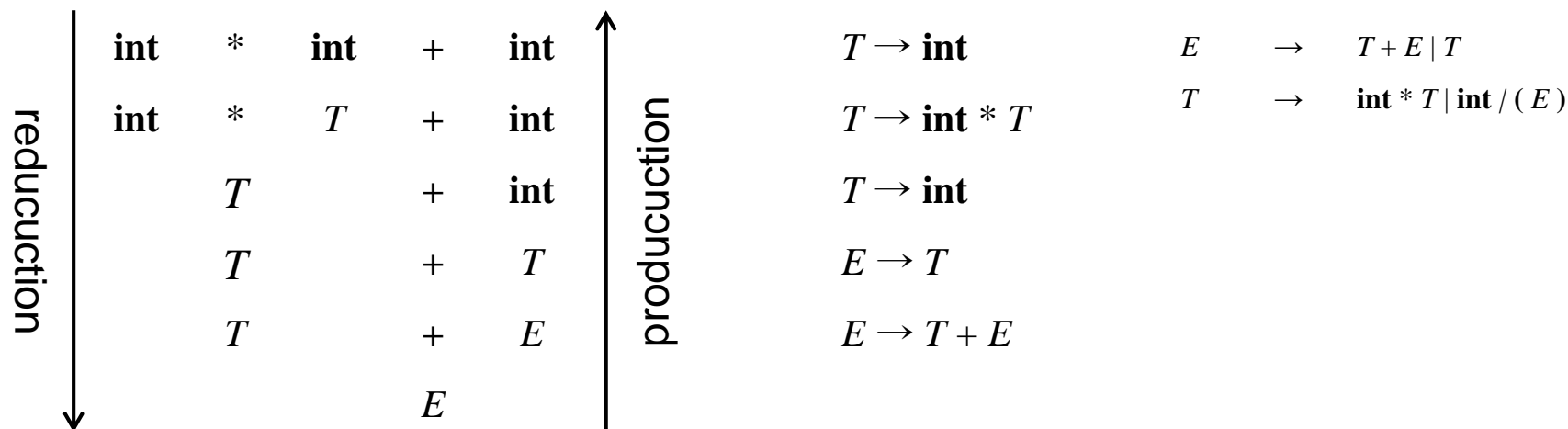
E → *T* + *E* | *T*

T → *T* * *F* | *F*

F → (*E*) | **int**

Reductions

- A reduction step is the action of replacing a specific substring matching the body of a production by the head of that production



- Bottom-up parsing reduces a string to the start symbol by inverting productions

Reductions

- Observation: the productions, read from bottom to top, trace a rightmost derivation of the parse tree

int	*	int	+	int
int	*	<i>T</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	

int	*	int	+	int
<i>F</i>	*	int	+	int
<i>T</i>	*	int	+	int
<i>T</i>	*	<i>F</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>F</i>
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	

Bottom-Up Parsing

- Insight: a bottom-up parser traces a rightmost derivation in reverse

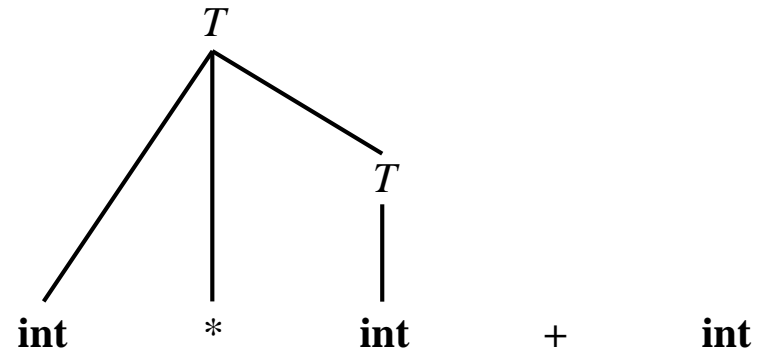
int	*	int	+	int
int	*	<i>T</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	

		<i>T</i>		
int	*	int	+	int

Bottom-Up Parsing

- Insight: a bottom-up parser traces a rightmost derivation in reverse

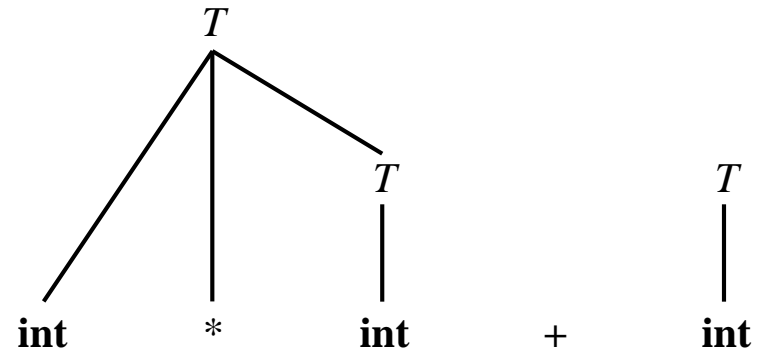
int	*	int	+	int
int	*	<i>T</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	



Bottom-Up Parsing

- Insight: a bottom-up parser traces a rightmost derivation in reverse

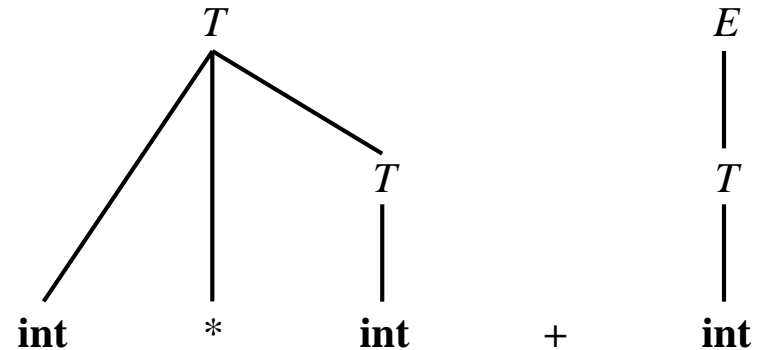
int	*	int	+	int
int	*	<i>T</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	



Bottom-Up Parsing

- Insight: a bottom-up parser traces a rightmost derivation in reverse

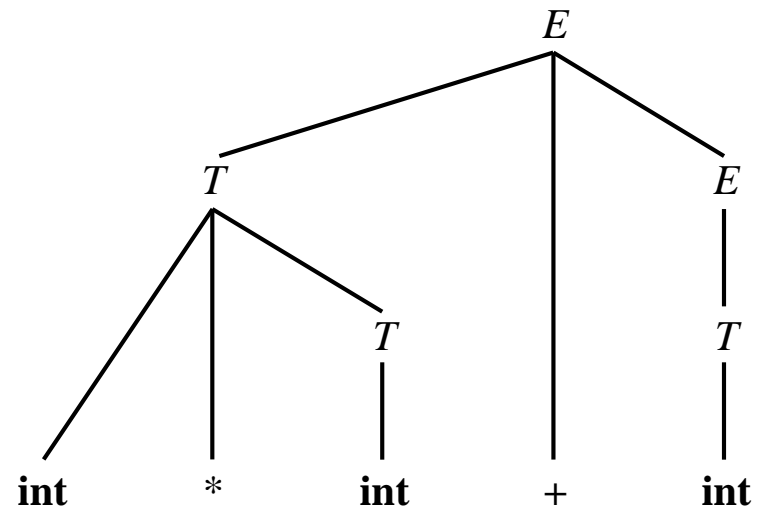
int	*	int	+	int
int	*	<i>T</i>	+	int
	<i>T</i>		+	int
	<i>T</i>		+	<i>T</i>
	<i>T</i>		+	<i>E</i>
			<i>E</i>	



Bottom-Up Parsing

- Insight: a bottom-up parser traces a rightmost derivation in reverse

int	*	int	+	int
int	*	<i>T</i>	+	int
		<i>T</i>	+	int
		<i>T</i>	+	<i>T</i>
		<i>T</i>	+	<i>E</i>
				<i>E</i>



Bottom-Up Parsing

- The “only” problem we have to solve:
 - when and what reductions to apply?

Handles

- A handle is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a right most derivation

Right Sentential Form			Handle	Reduction
int_1	$*$	int_2	int_1	$F \rightarrow \text{int}$
F	$*$	int_2	F	$T \rightarrow F$
T	$*$	int_2	int_2	$F \rightarrow \text{int}$
T	$*$	F	$T * F$	$T \rightarrow T * F$
	T		T	$E \rightarrow T$
	E			

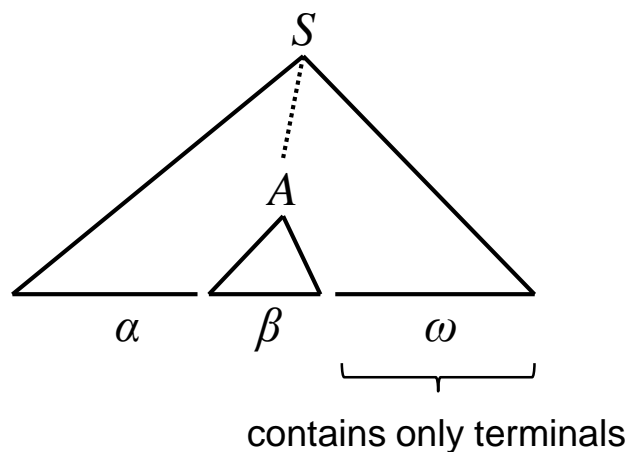
E	\rightarrow	$T + E \mid T$
T	\rightarrow	$T * F \mid F$
F	\rightarrow	$(E) \mid \text{int}$

- the leftmost substring that matches the body of some production is not necessarily a handle!

Handles

- Formal definition: a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ .

$$S \xRightarrow{rm}^* \alpha A \omega \xRightarrow{rm} \alpha \beta \omega$$



Handle Pruning

- Repeated handle pruning yields a rightmost derivation in reverse order:

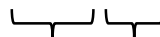
$$S = \gamma_0 \xRightarrow{rm} \gamma_1 \xRightarrow{rm} \gamma_2 \xRightarrow{rm} \dots \xRightarrow{rm} \gamma_{n-1} \xRightarrow{rm} \gamma_n = \omega$$

- obtain γ_{n-1} by locating β_n in γ_n , and replacing β_n by A_n in the production $A_n \rightarrow \beta_n$
- obtain γ_{n-2} by locating β_{n-1} in γ_{n-1} , and replacing β_{n-1} by A_{n-1} in the production $A_{n-1} \rightarrow \beta_{n-1}$
- ...
- obtain γ_0 by locating β_1 in γ_1 , and replacing β_1 by A_1 in the production $A_1 \rightarrow \beta_1$

Shift-Reduce Parsing

- Remember from handle pruning that given the next reduction $A \rightarrow \beta$ for a configuration $\alpha\beta\omega$ encountered during our bottom-up parse to $\alpha A\omega$, the string ω only contains terminals.
- Idea: split input string into two substrings
 - right substring: contains only terminals, not yet examined by parser
 - left substring: contains terminals and non-terminals
 - indicate the dividing point by a |

$\alpha\beta|\omega$



seen and possibly reduced,
contains terminals and
non-terminals

not yet seen, contains
only terminals

Shift-Reduce Parsing

■ The only actions in a shift-reduce parser are...

- *Shift* and
- *Reduce*

Shift-Reduce Parsing

- Shift: move the separator one position to the right
 - here: one position = one terminal

$$ABvC|xyz \xRightarrow[\text{shift}]{} ABvCx|yz$$

- Reduce: apply an inverse production at the right end of the left substring
 - given production $T \rightarrow Cx$, then

$$ABvCx|yz \xRightarrow[\text{reduce}]{} ABvT|yz$$

Shift-Reduce Parsing

- Back to our example:

int	*	int	+	int	reduce $T \rightarrow \mathbf{int}$
int	*	T	+	int	reduce $T \rightarrow \mathbf{int} * T$
T	+	int			reduce $T \rightarrow \mathbf{int}$
T	+	T			reduce $E \rightarrow T$
T	+	E			reduce $E \rightarrow T + E$
		E			

Shift-Reduce Parsing

- Back to our example:

int	*	int	+	int	shift
int	*	int	+	int	shift
int	*	int	+	int	shift
int	*	int	+	int	reduce $T \rightarrow \text{int}$
int	*	T	+	int	reduce $T \rightarrow \text{int} * T$
	T		+	int	shift
	T		+	int	shift
	T		+	int	reduce $T \rightarrow \text{int}$
	T		+	T	reduce $E \rightarrow T$
	T		+	E	reduce $E \rightarrow T + E$
			E		

Shift-Reduce Parsing

- Note: the reduce operation is only allowed to reduce a handle at the *right end* of the left substring.
- Can there be any handles appearing not at the very right end of the left substring?
 - No. The handle will always eventually appear on the right end.
 - Consider all possible forms of two successive steps in any rightmost derivation:

$$(1) \quad S \xRightarrow{rm}^* \alpha Az \xRightarrow{rm} \alpha \beta Byz \xRightarrow{rm} \alpha \beta \gamma yz$$

$$(2) \quad S \xRightarrow{rm}^* \alpha BxAz \xRightarrow{rm} \alpha Bxyz \xRightarrow{rm} \alpha \gamma xyz$$

Shift-Reduce Parsing

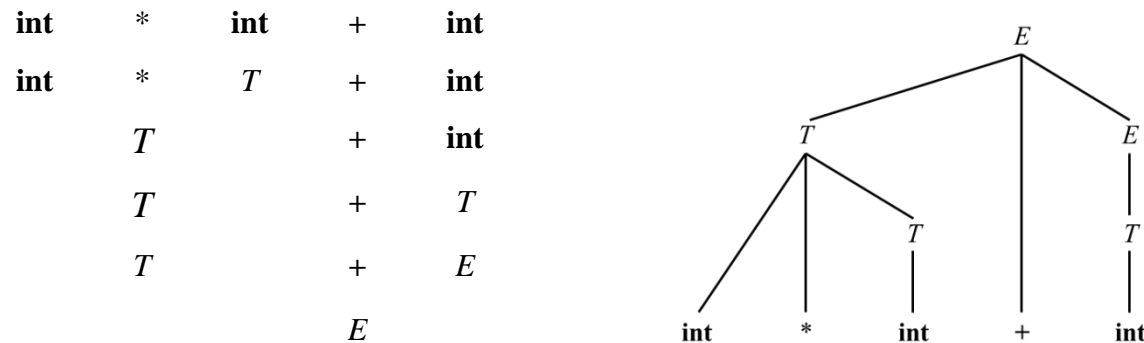
- The left-string can be implemented by a stack
 - top of stack is the |
 - shift = push terminal onto stack
 - reduce = pop some symbols off the stack (rhs), push a nonterminal (lhs)
- Conflicts during shift-reduce parsing
one than more action may lead to a *valid* parse
 - shift-reduce: both shift and reduce are legal
 - ▶ can usually be dealt with precedence rules
 - reduce-reduce conflict: legal to reduce by two different productions
 - ▶ usually a problem with the grammar

Shift-Reduce Parsing

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left on the rightmost non-terminal
 - shift-reduce moves are sufficient, the | never has to move to the left
- Bottom-up parsing algorithms = “handle recognizers”
 - how do we know whether to shift or to reduce, or, in other words, how to recognize handles?

Recap: Bottom-Up Parsing

- construct the parse tree beginning at the leaves, working upwards towards the root



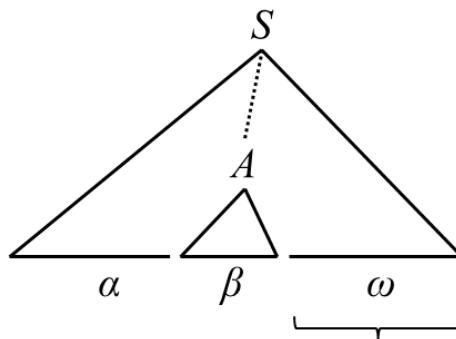
- performs one reduction at a time: replace a productions RHS with its LHS (a non-terminal)
 - a reduction is the inverse operation to a deduction in top-down parsing
- in bottom-up parsing, we are trying to find an order of reductions so that each reduction produces the previous right-sentential form in a rightmost derivation of the input string.

Recap: Bottom-Up Parsing

■ Handle pruning

- a handle is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a right most derivation
- note that the leftmost substring that matches the body of some production is not necessarily a handle!
- repeated handle pruning will eventually reduce to S , and in the process generate the order of reductions.

$$S \xRightarrow[*]{rm} \alpha A \omega \xRightarrow{rm} \alpha \beta \omega$$



contains only terminals

Recap: Bottom-Up Parsing

■ Shift-Reduce Parsing

- split the input into two halves $\alpha\beta|\omega$

$\underbrace{\alpha\beta}_{\text{seen and possibly reduced, contains terminals and non-terminals}} \underbrace{|\omega}_{\text{not yet seen, contains only terminals}}$

- handles always appear immediately to the left to the bar
- grammar symbols to the left of the bar are organized on a stack
- stack operations
 - ▶ shift: read next input symbol and push it onto the stack

$$ABvC|xyz \xRightarrow{\text{shift}} ABvCx|yz$$

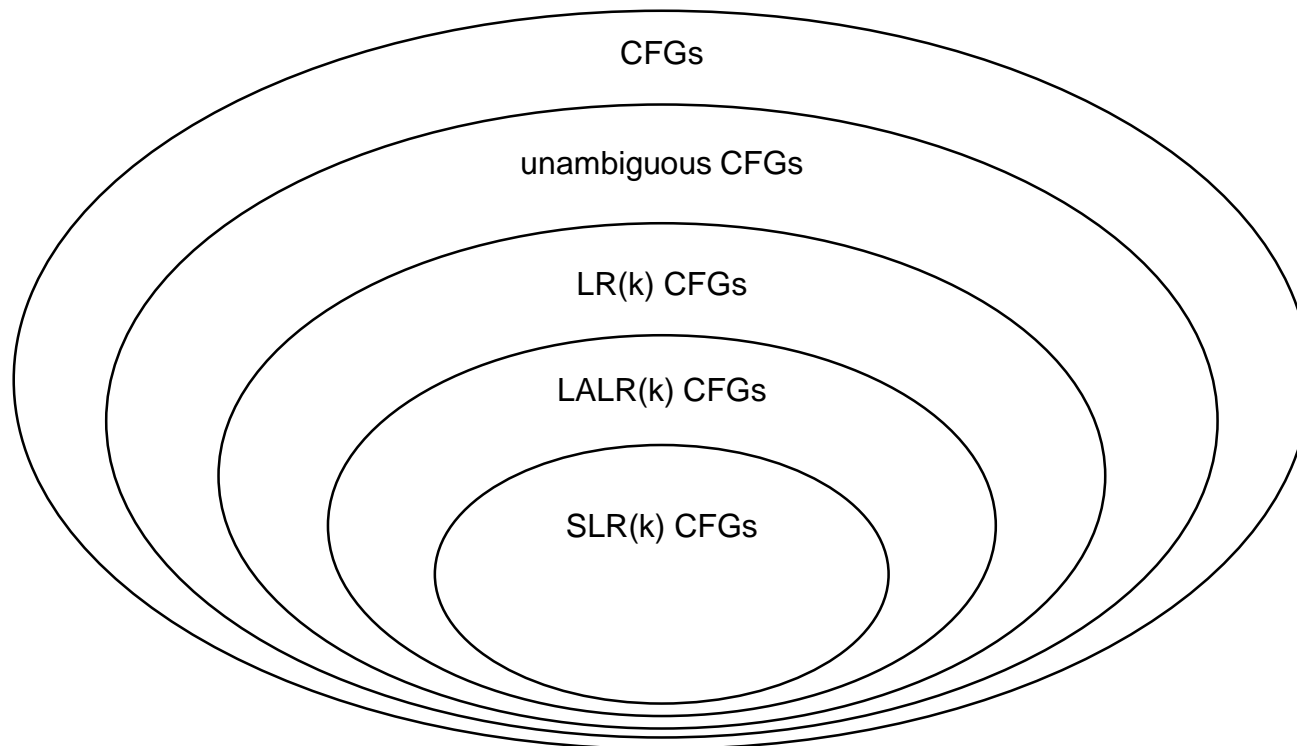
- ▶ reduce: replace the RHS of a production on top of the stack by its LHS

$$ABvCx|yz \xRightarrow{\text{reduce}} ABvT|yz$$

for a production $T \rightarrow Cx$


Recognizing Handles

- For general context free grammars, no known efficient algorithm exists to recognize handles
- For certain context free grammars, heuristics exist that always identify the handle correctly



Viable Prefixes

- In a shift-reduce parser, only the contents of the stack are known
 - rest of input not yet seen

$$\alpha | \omega$$


seen and possibly reduced,
contains terminals and
non-terminals

not yet seen, contains
only terminals

- α is a viable prefix if there is an ω such that $\alpha | \omega$ occurs as a valid configuration during a parse using a shift-reduce parser
 - viable prefixes are prefixes of handles
 - ▶ i.e., they do not extend past the right end of the handle
 - only viable prefixes on the stack \leftrightarrow no parsing error has been detected

Viable Prefixes and Regular Languages

- For any grammar, the set of viable prefixes of is a regular language
 - this is the key of bottom-up parsing tools: viable prefixes can be detected using a DFA

LR(0) Items

- An item is a production with one “.” somewhere in the RHS of a production
 - example:

$$T \rightarrow T + E / (E)$$

$$T \rightarrow \cdot T + E$$

$$T \rightarrow T \cdot + E$$

$$T \rightarrow T + \cdot E$$

$$T \rightarrow T + E \cdot$$

$$T \rightarrow \cdot (E)$$

$$T \rightarrow (\cdot E)$$

$$T \rightarrow (E \cdot)$$

$$T \rightarrow (E) \cdot$$

- special case: $X \rightarrow \varepsilon$ produces the item $X \rightarrow \cdot$
- the LR(0) items comprise the set of all items of a grammar

Viable Prefixes and Items

- The parser can apply a reduce action only if it has a complete RHS of a production on top of the stack
 - most of the time, only a part of the complete RHS is on the stack
 - these parts of the RHS are always prefixes of the RHS of a production, and thus must be one of the LR(0) items
 - example

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} / (E)$

with input **(int)**

- ▶ $(E \mid)$ will be a state encountered during the shift-reduce parse
- ▶ $(E$ is the prefix of the RHS of $T \rightarrow (E)$
- ▶ this prefix corresponds to the item $T \rightarrow (E \cdot)$
- items describe all possible configurations of productions, in particular, capture what we have seen so far (to the left of the \cdot) and what we expect to see (to the right of the \cdot)

Viable Prefixes and the Stack

- We know that the top of the stack contains some prefix of the production $X \rightarrow \alpha$ to be reduced next

$$\dots \text{prefix} \mid \omega \quad \Longrightarrow \quad \dots X \mid \omega$$

- Observations

- prefix is one of the items of $X \rightarrow \alpha$
- X must be part of some prefix of the production to be reduced after X
 - ▶ this means that the stack can only contain prefixes:

$$\text{prefix}_1 \text{prefix}_2 \dots \text{prefix}_{n-1} \text{prefix}_n$$

- ▶ let prefix_i be a prefix of the production $X_i \rightarrow \alpha_i$
 - prefix_i will eventually get reduced to X_i
 - there is an $X_{i-1} \rightarrow \text{prefix}_{i-1} X_i \beta$ for some β , i.e., the missing part of α_{i-1} starts with X_i

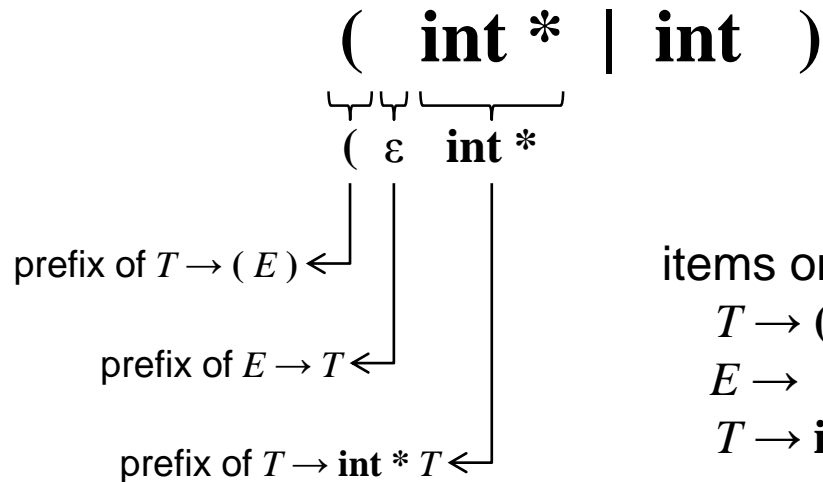
Viable Prefixes and the Stack

■ Example:

- input string (int * int)
- shift-reduce parser state

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

$$T \rightarrow \text{int} * T \mid \text{int} / (E)$$



items on stack

$$T \rightarrow (\cdot E)$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \text{int} * \cdot T$$

meaning

seen “(“ of $T \rightarrow (E)$

seen nothing of $E \rightarrow T$

seen int * of $T \rightarrow \text{int} * T$

Recognizing Viable Prefixes

■ Recognizing viable prefixes with an automaton

1. Augmented grammar of G :

add a new production $S' \rightarrow S$ to G

2. LR(0) automaton

create an NFA that recognizes the viable prefixes

- ▶ input of the NFA: stack, bottom to top
- ▶ output of the NFA: yes, can unwind stack into viable prefixes / no
- ▶ states of the NFA: items of G , start state is $S' \rightarrow \cdot S$
 - every state is accepting
- ▶ transition table
 - for every item $Y \rightarrow \alpha \cdot X \beta$ add a transition $Y \rightarrow \alpha \cdot X \beta \xrightarrow{X} Y \rightarrow \alpha X \cdot \beta$
 - for every item $Y \rightarrow \alpha \cdot X \beta$, X a non-terminal, and a production $X \rightarrow \gamma$, add a transition $Y \rightarrow \alpha \cdot X \beta \xrightarrow{\epsilon} X \rightarrow \cdot \gamma$
- ▶ starting state: $S' \rightarrow \cdot S$, all state are accepting states

Recognizing Viable Prefixes

■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$

$$S' \rightarrow \cdot E$$

$$S' \rightarrow E \cdot$$

$$E \rightarrow \cdot T + E$$

$$E \rightarrow T \cdot + E$$

$$E \rightarrow T + \cdot E$$

$$E \rightarrow T + E \cdot$$

$$E \rightarrow \cdot T$$

$$E \rightarrow T \cdot$$

$$T \rightarrow \cdot \mathbf{int} * T$$

$$T \rightarrow \mathbf{int} \cdot * T$$

$$T \rightarrow \mathbf{int} * \cdot T$$

$$T \rightarrow \mathbf{int} * T \cdot$$

$$T \rightarrow \cdot \mathbf{int}$$

$$T \rightarrow \mathbf{int} \cdot$$

$$T \rightarrow \cdot (E)$$

$$T \rightarrow (\cdot E)$$

$$T \rightarrow (E \cdot)$$

$$T \rightarrow (E) \cdot$$

Recognizing Viable Prefixes

■ Example:

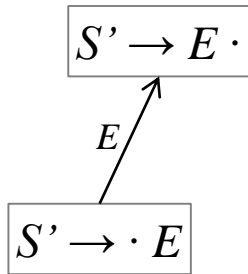
$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$

$$S' \rightarrow \cdot E$$

Recognizing Viable Prefixes

■ Example:

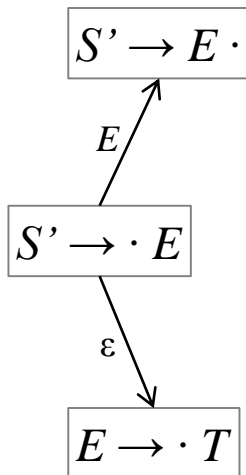
$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$



Recognizing Viable Prefixes

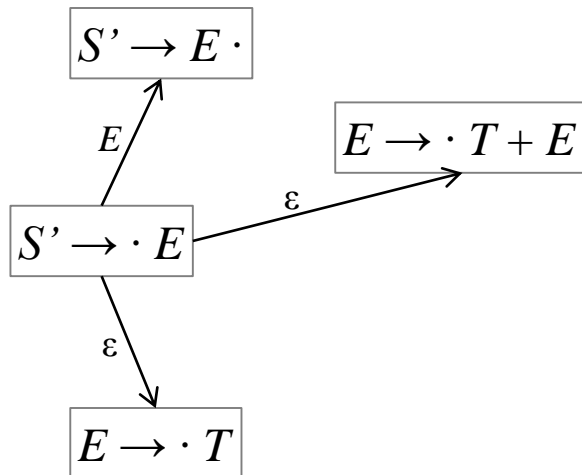
■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$



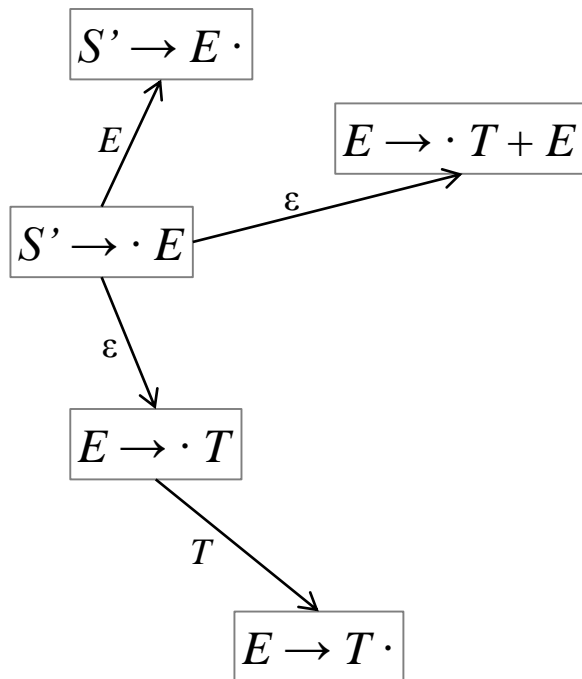
Recognizing Viable Prefixes

■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$


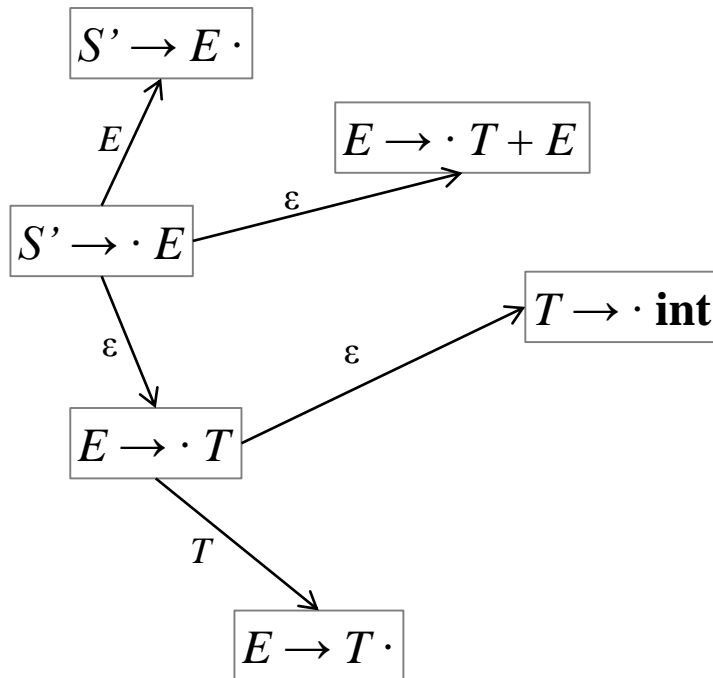
Recognizing Viable Prefixes

■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$


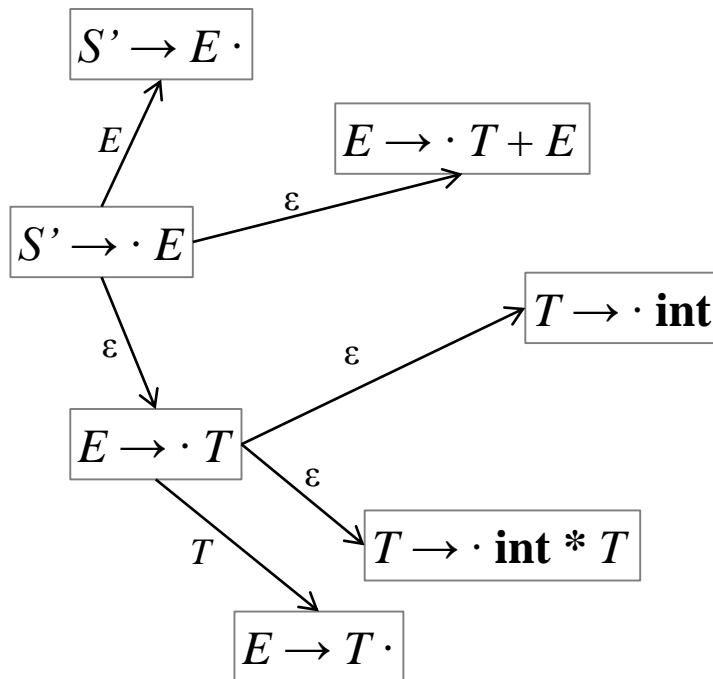
Recognizing Viable Prefixes

■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$


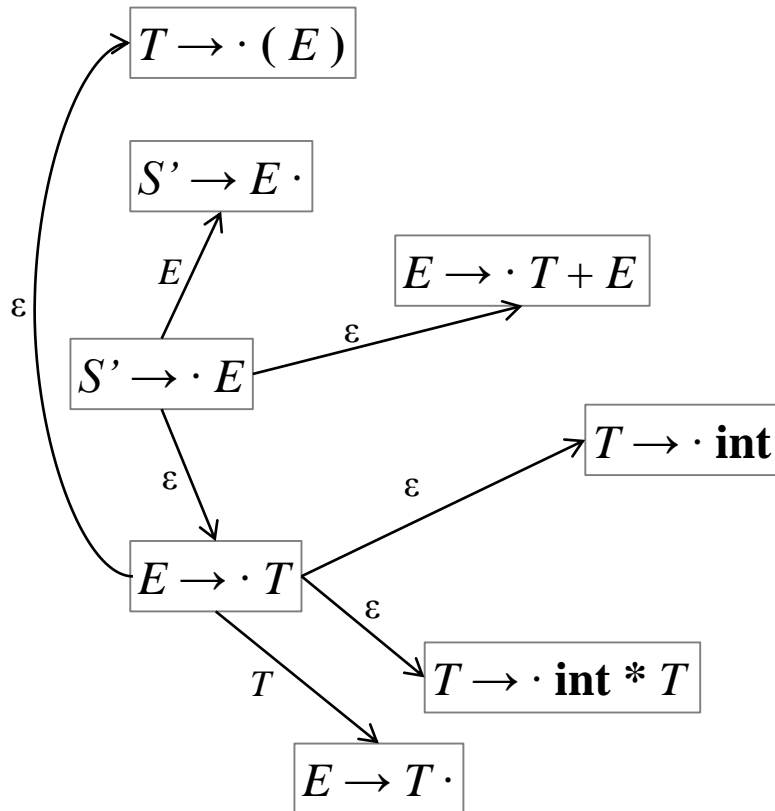
Recognizing Viable Prefixes

■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \mathbf{int} * T \mid \mathbf{int} / (E) \end{aligned}$$


Recognizing Viable Prefixes

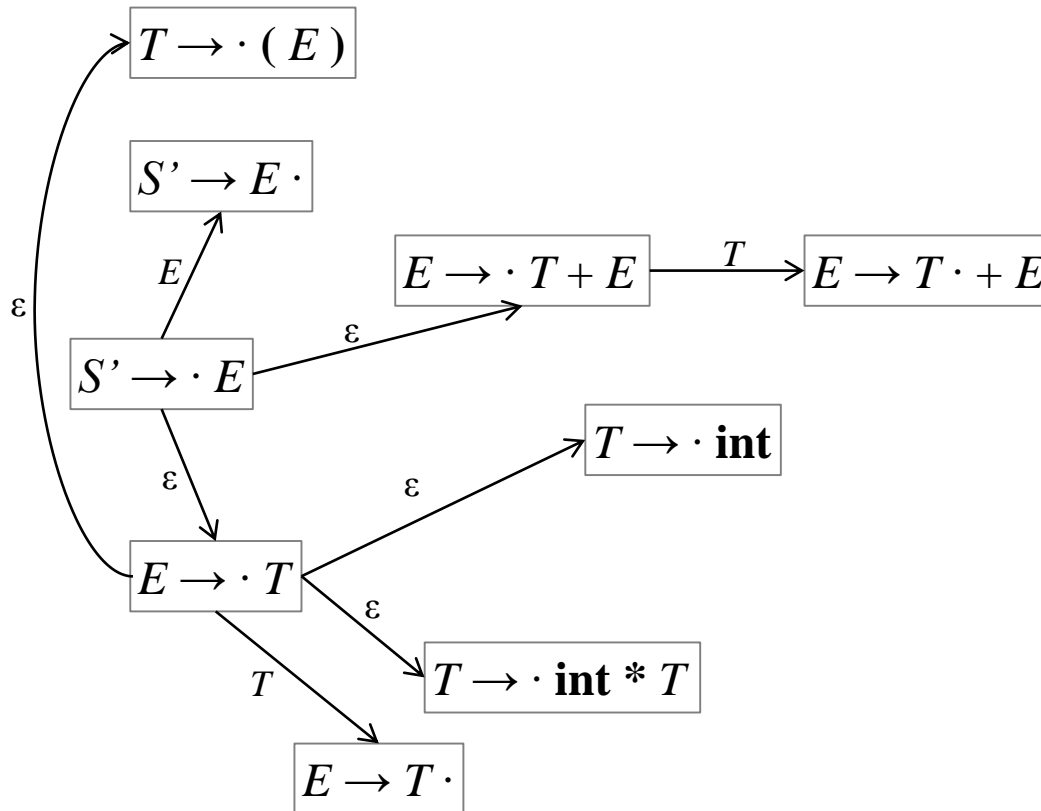
■ Example:



$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

Recognizing Viable Prefixes

■ Example:



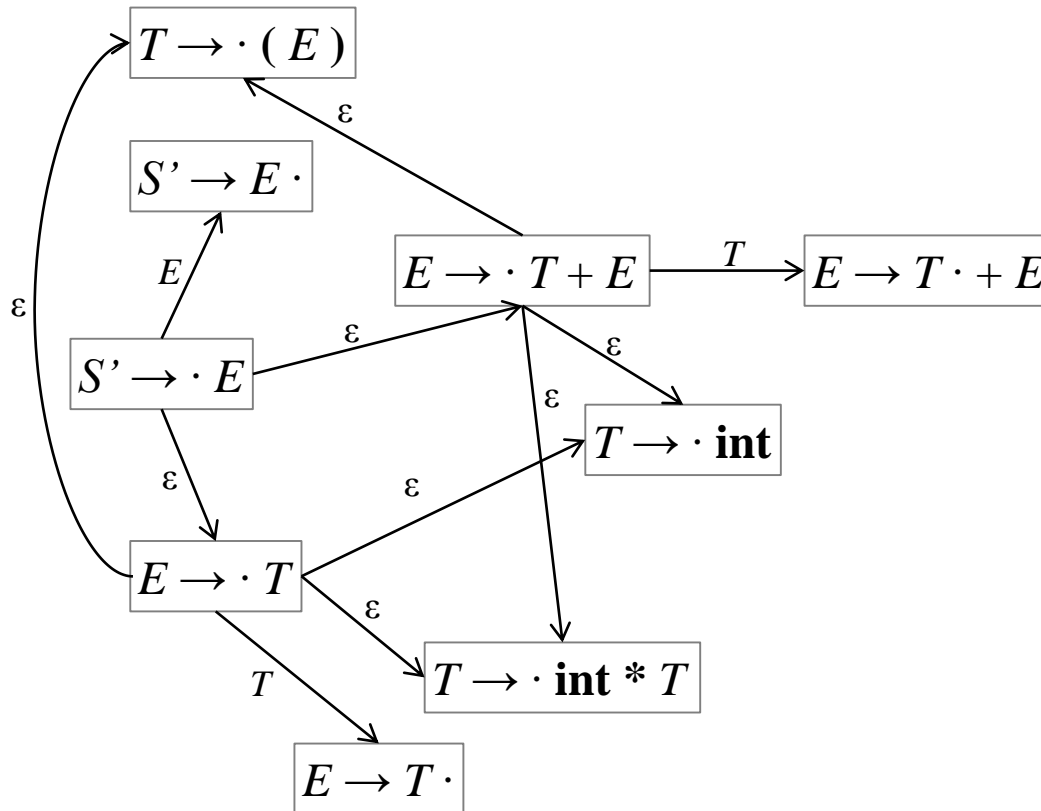
$S' \rightarrow E$

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} / (E)$

Recognizing Viable Prefixes

■ Example:



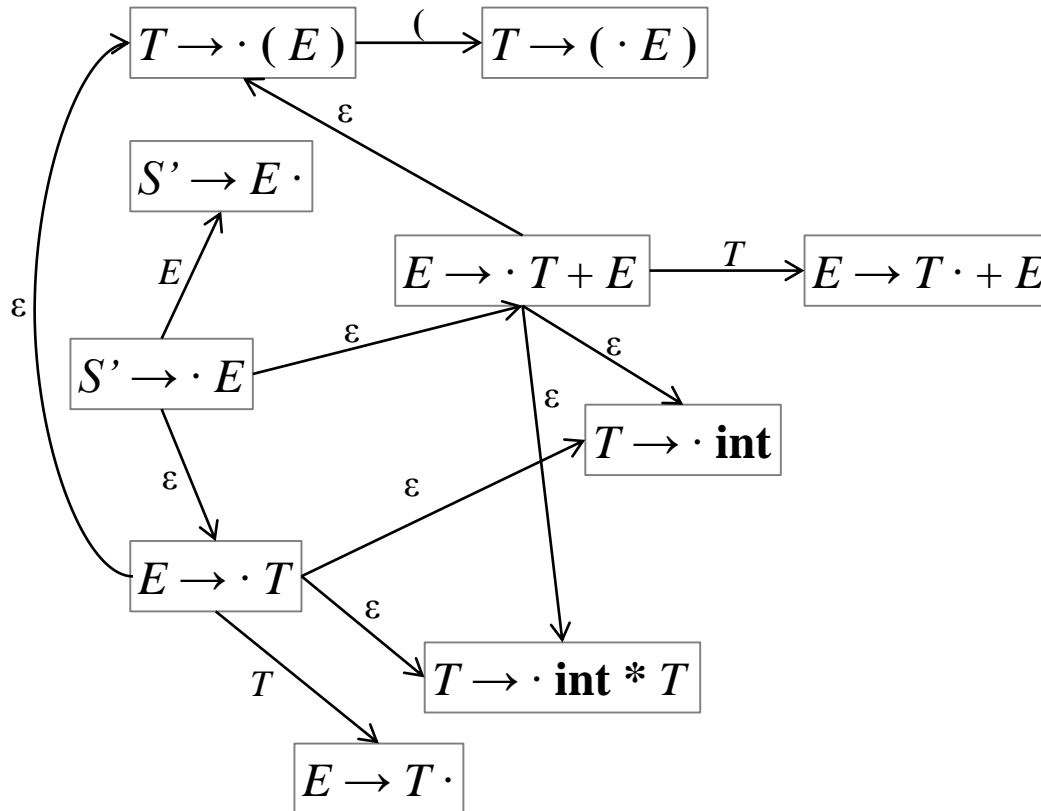
$S' \rightarrow E$

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} / (E)$

Recognizing Viable Prefixes

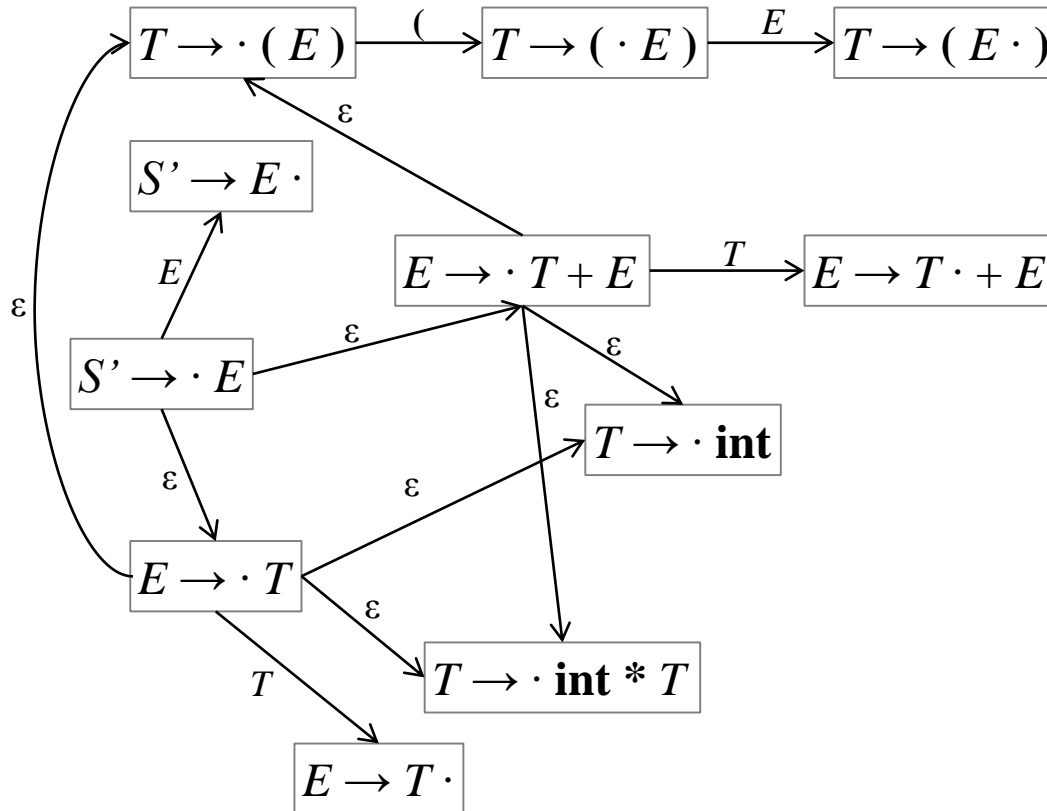
■ Example:



$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

Recognizing Viable Prefixes

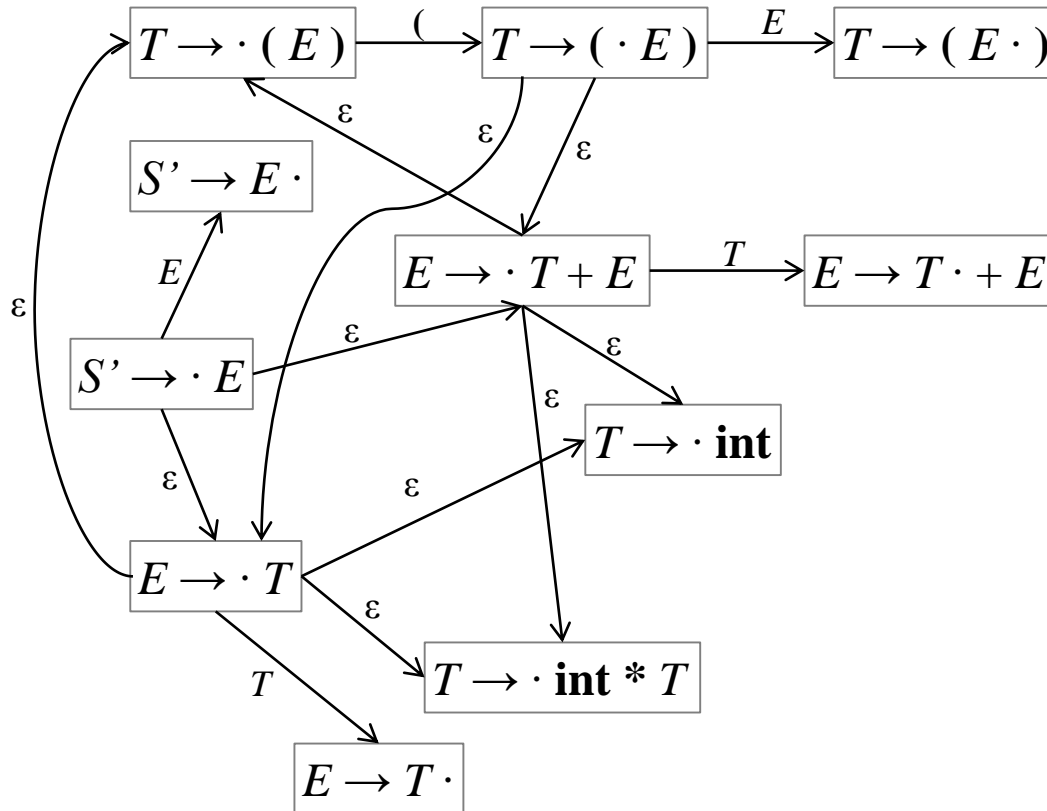
■ Example:



$$\begin{aligned}
 S' &\rightarrow E \\
 E &\rightarrow T + E \mid T \\
 T &\rightarrow \text{int} * T \mid \text{int} \mid (E)
 \end{aligned}$$

Recognizing Viable Prefixes

■ Example:



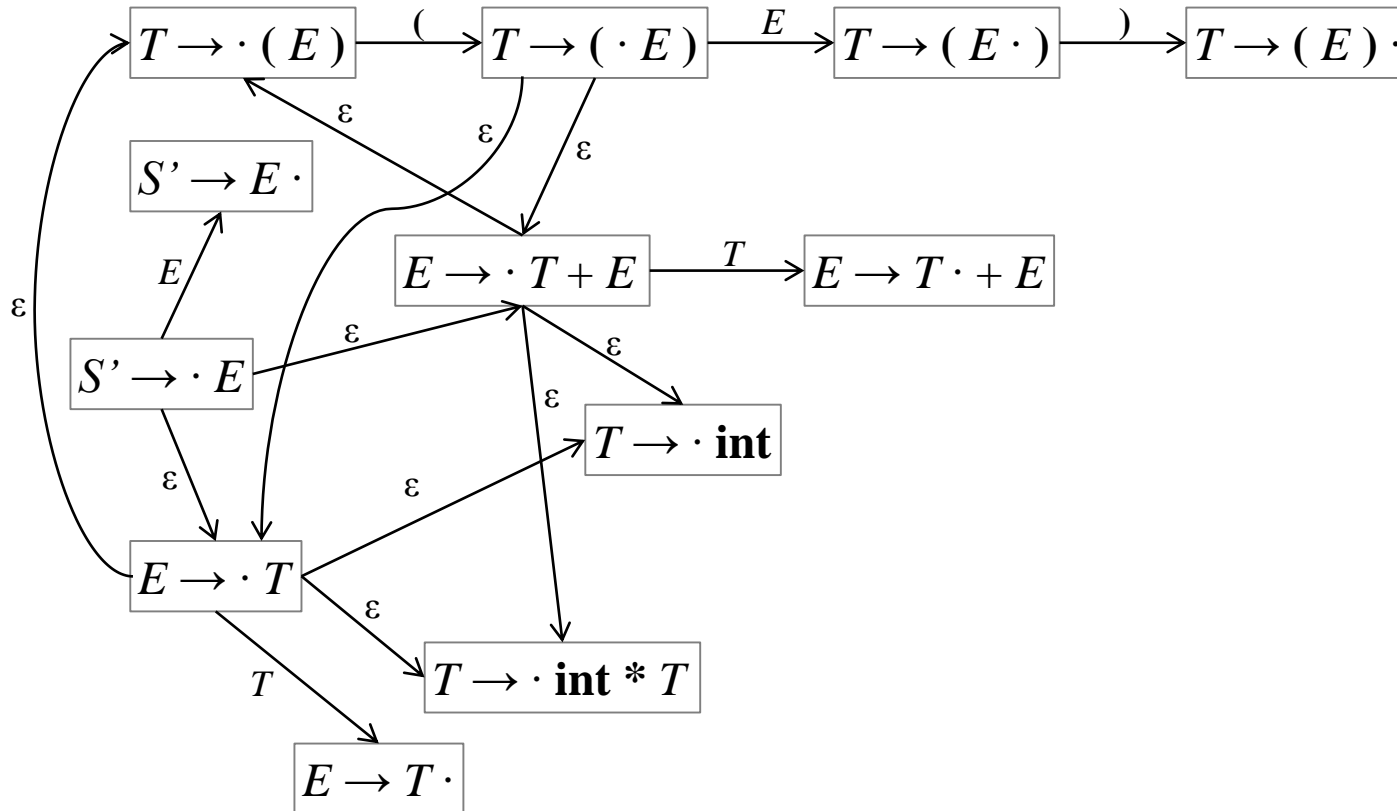
$S' \rightarrow E$

$E \rightarrow T + E \mid T$

$T \rightarrow \mathbf{int} * T \mid \mathbf{int} / (E)$

Recognizing Viable Prefixes

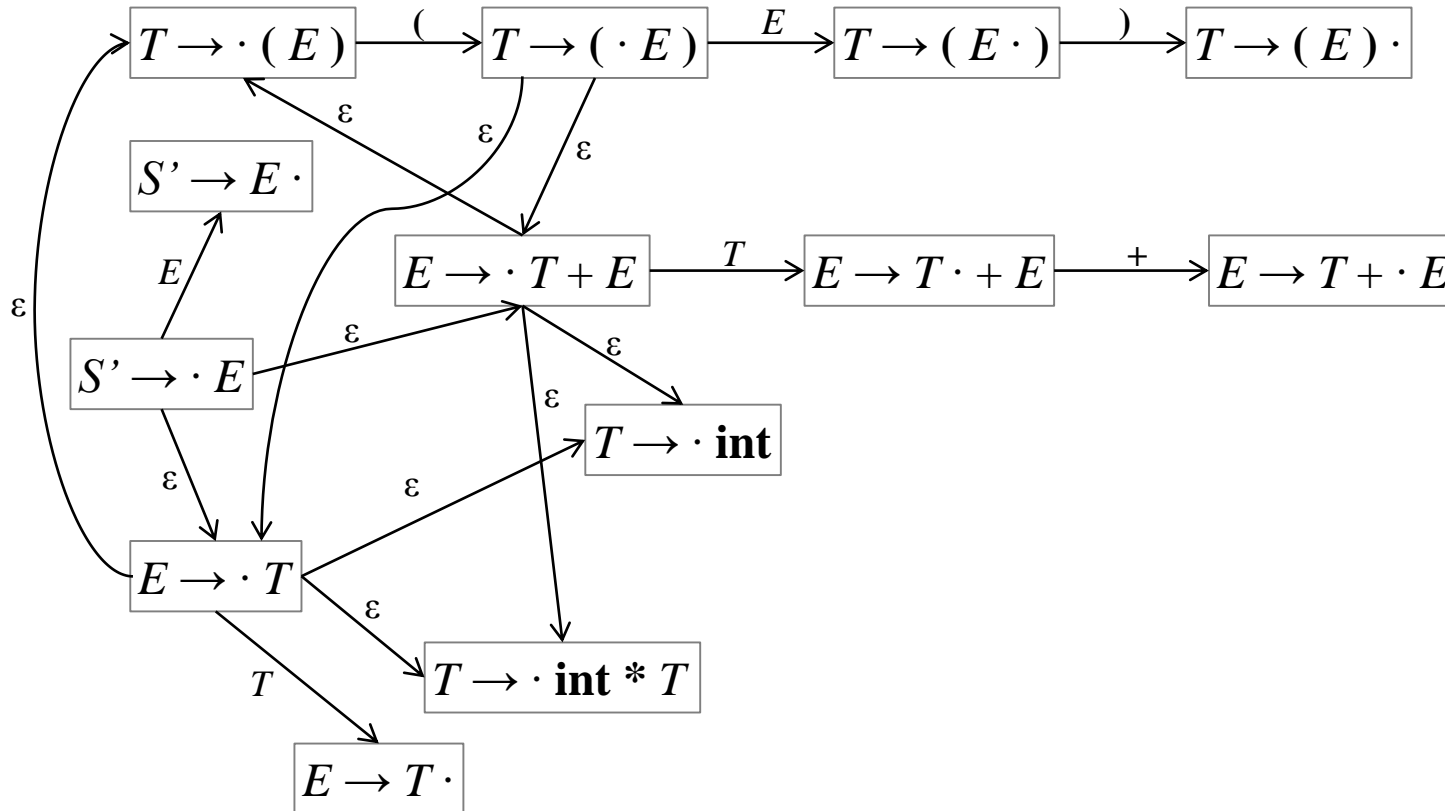
■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$


Recognizing Viable Prefixes

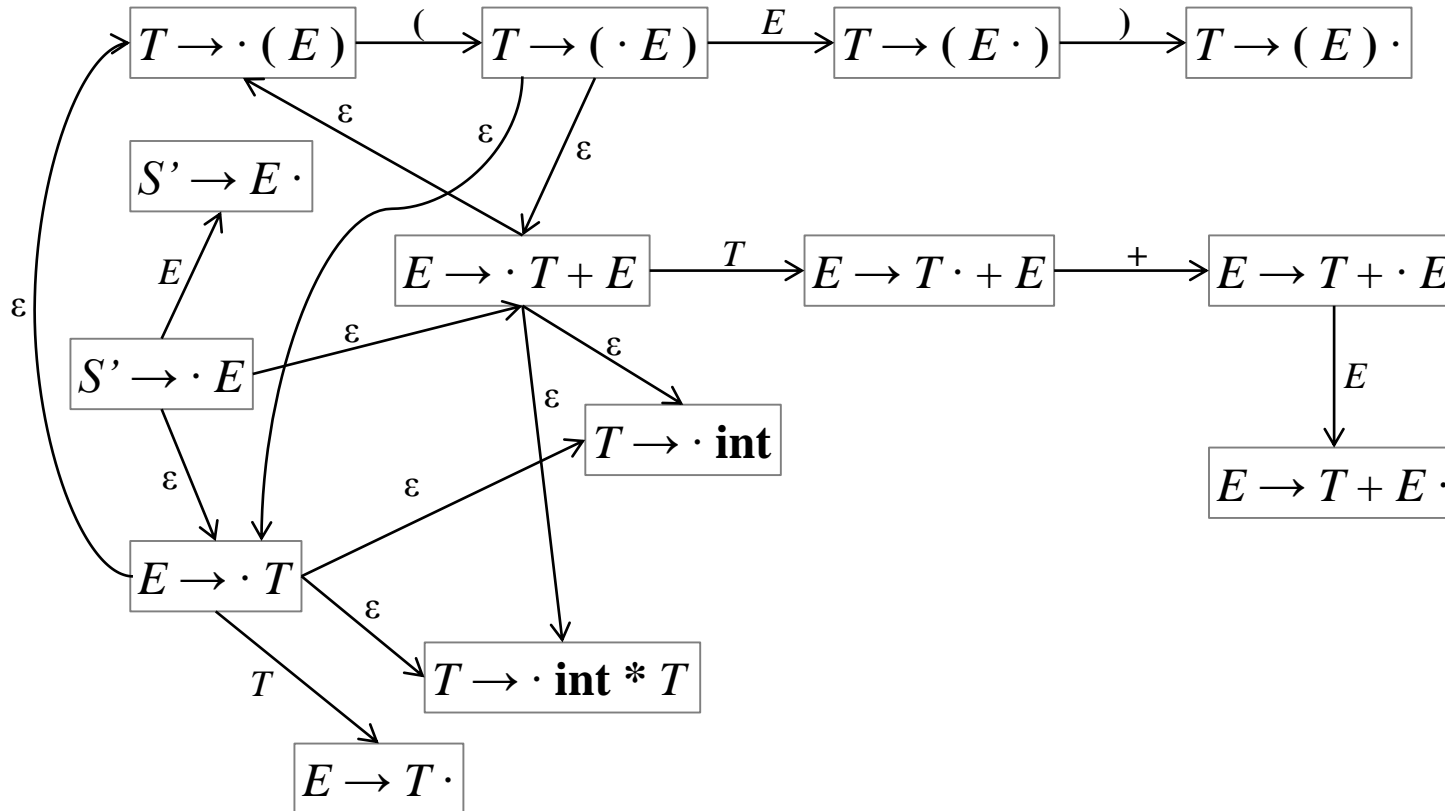
■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$



Recognizing Viable Prefixes

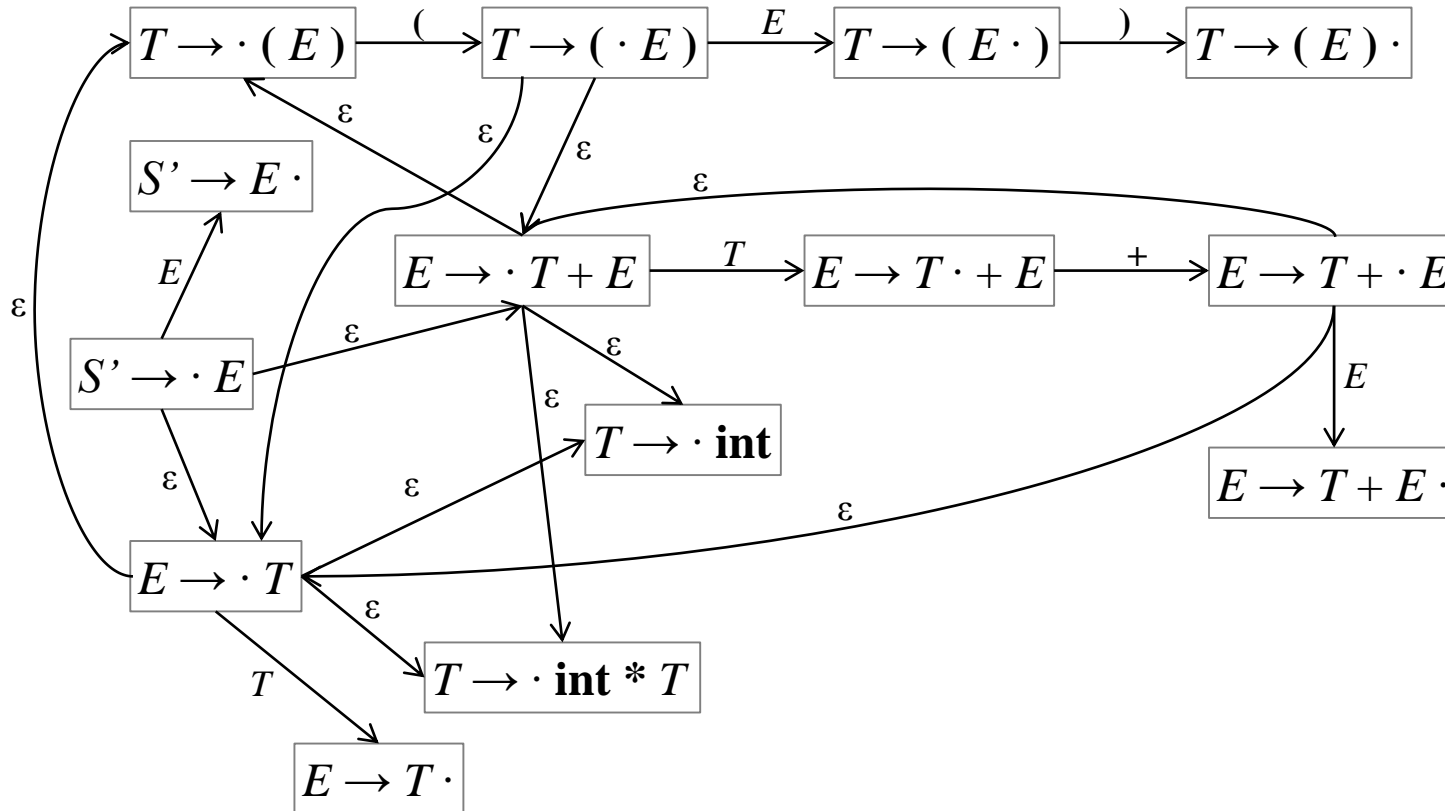
■ Example:

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$


Recognizing Viable Prefixes

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

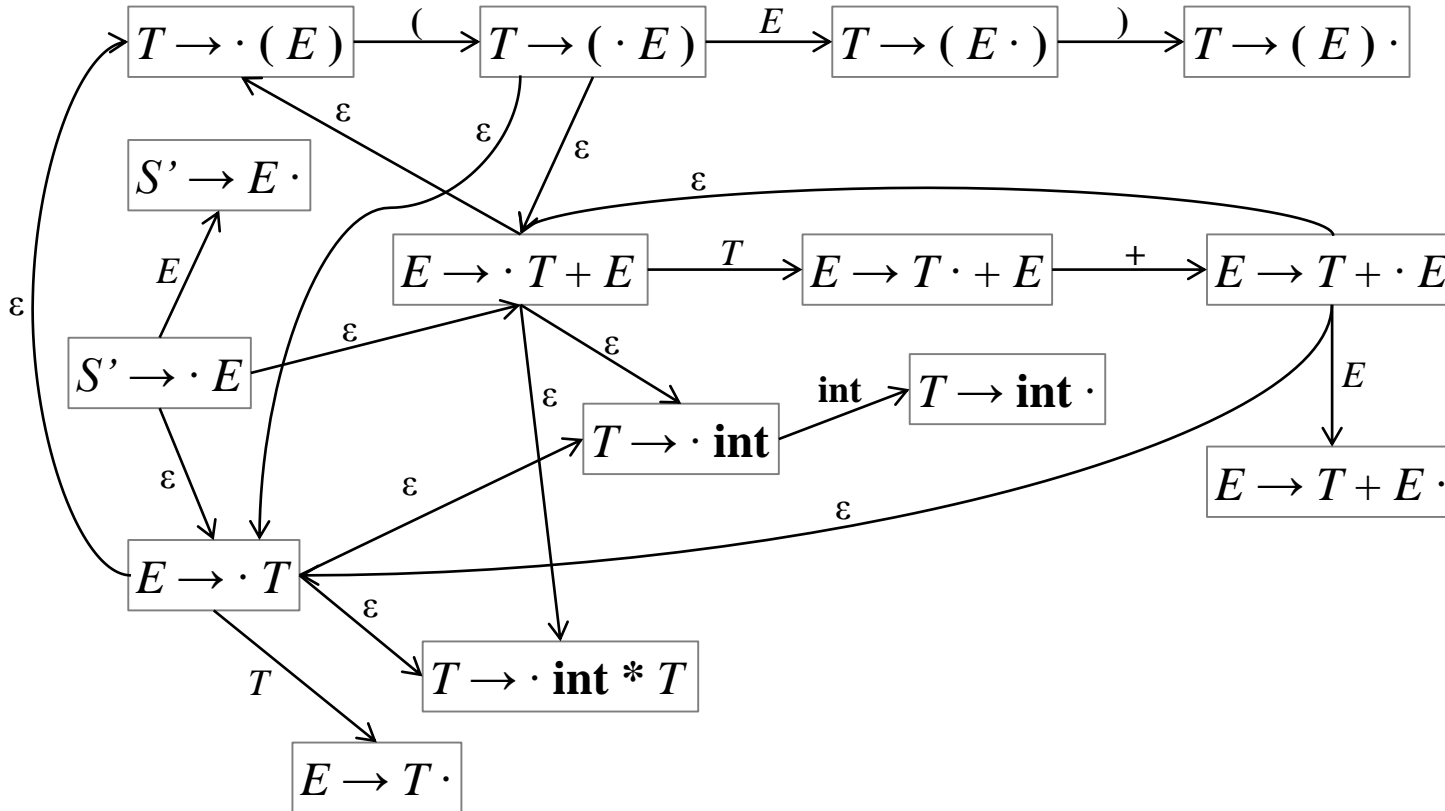
■ Example:



Recognizing Viable Prefixes

$$S' \rightarrow E$$
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \mathbf{int} * T \mid \mathbf{int} / (E)$$

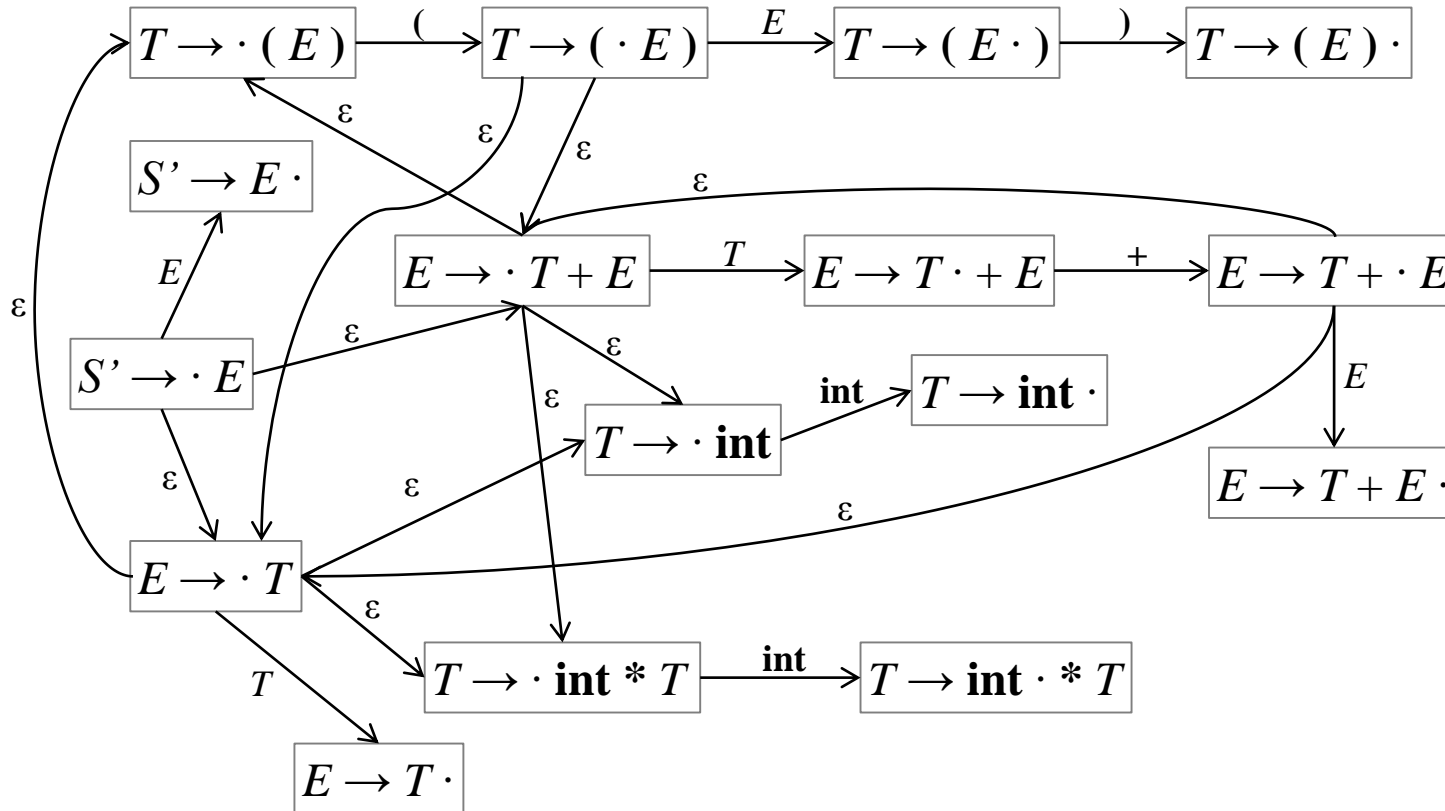
■ Example:



Recognizing Viable Prefixes

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

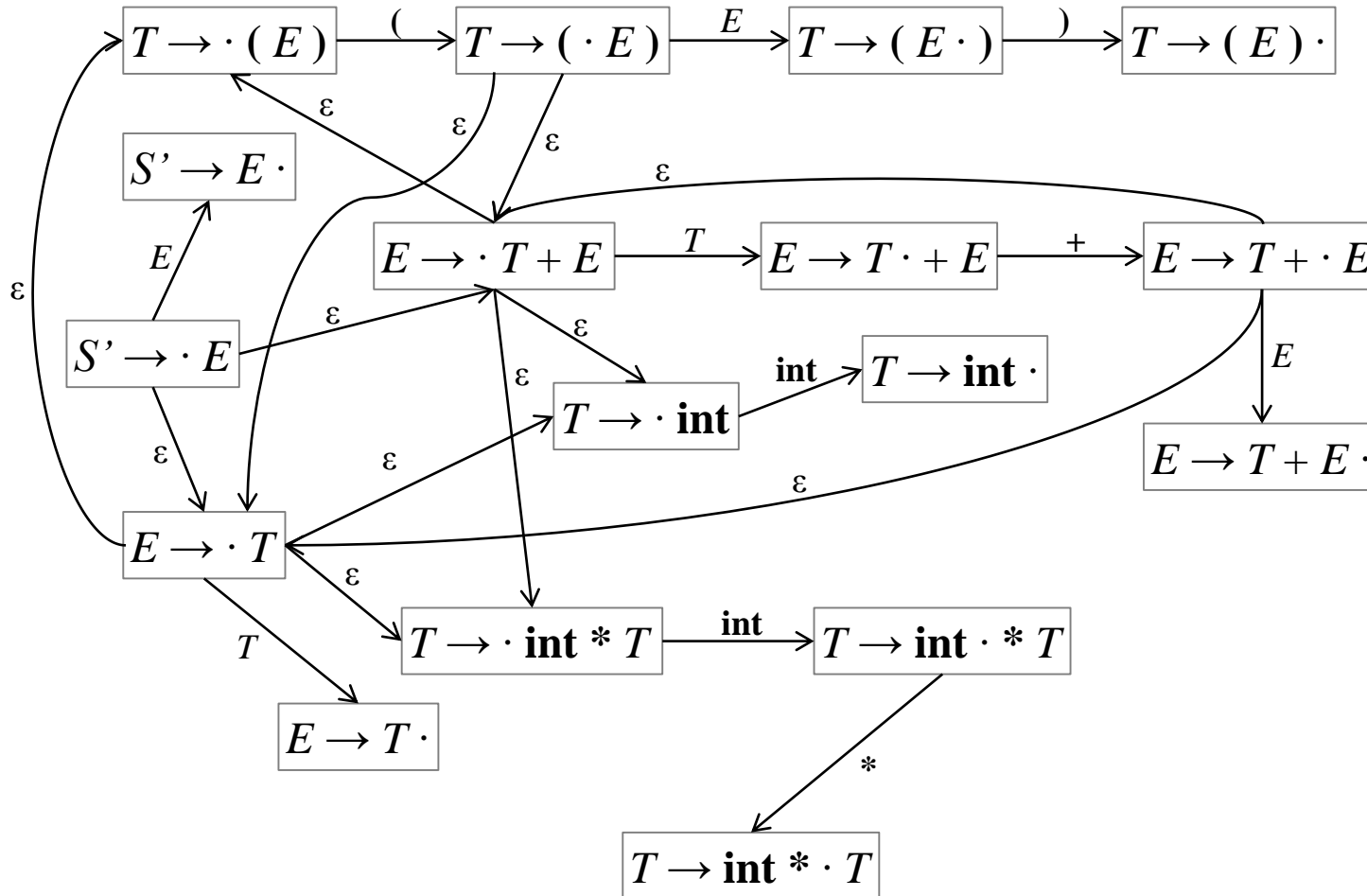
■ Example:



Recognizing Viable Prefixes

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

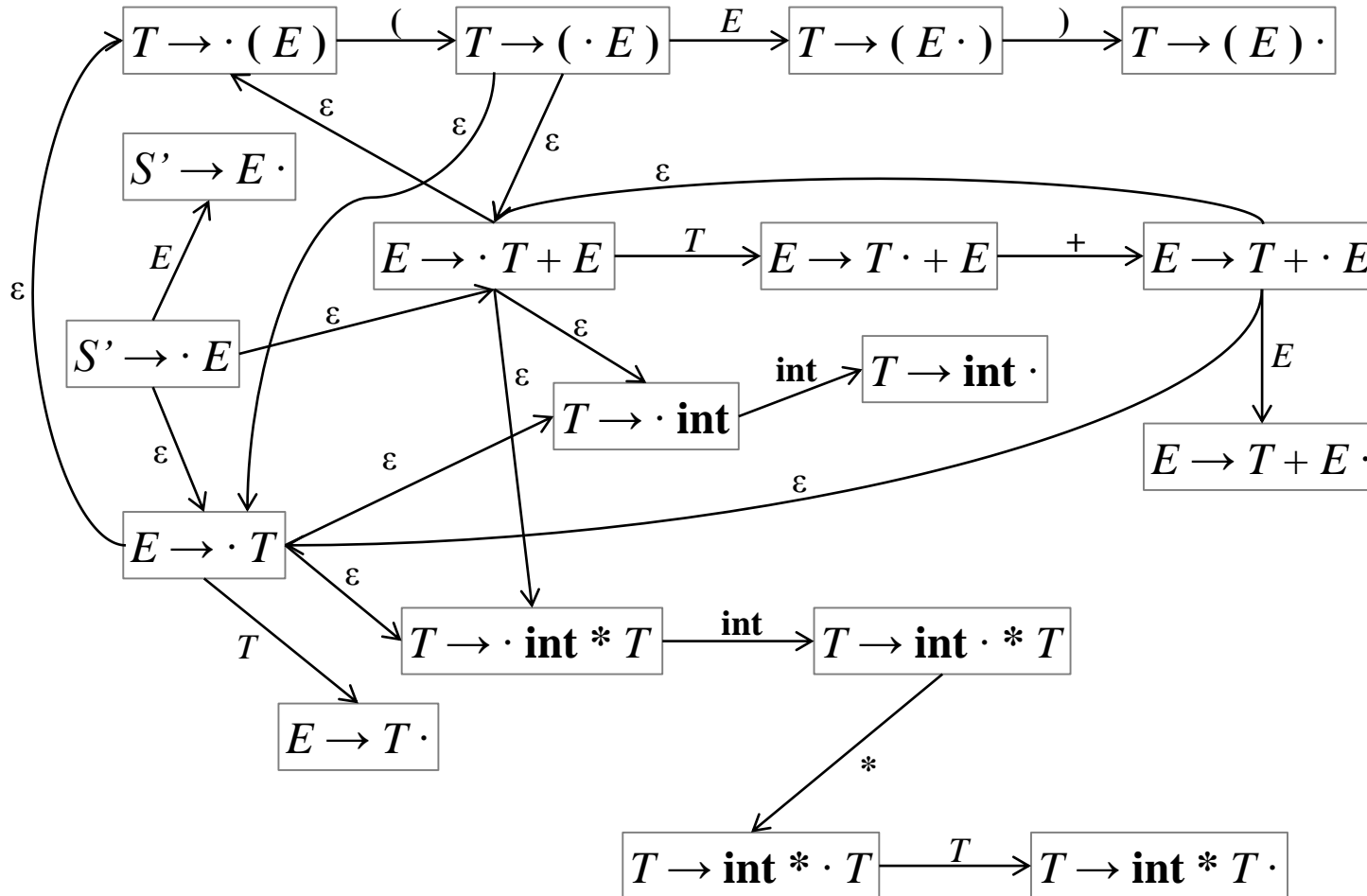
■ Example:



Recognizing Viable Prefixes

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

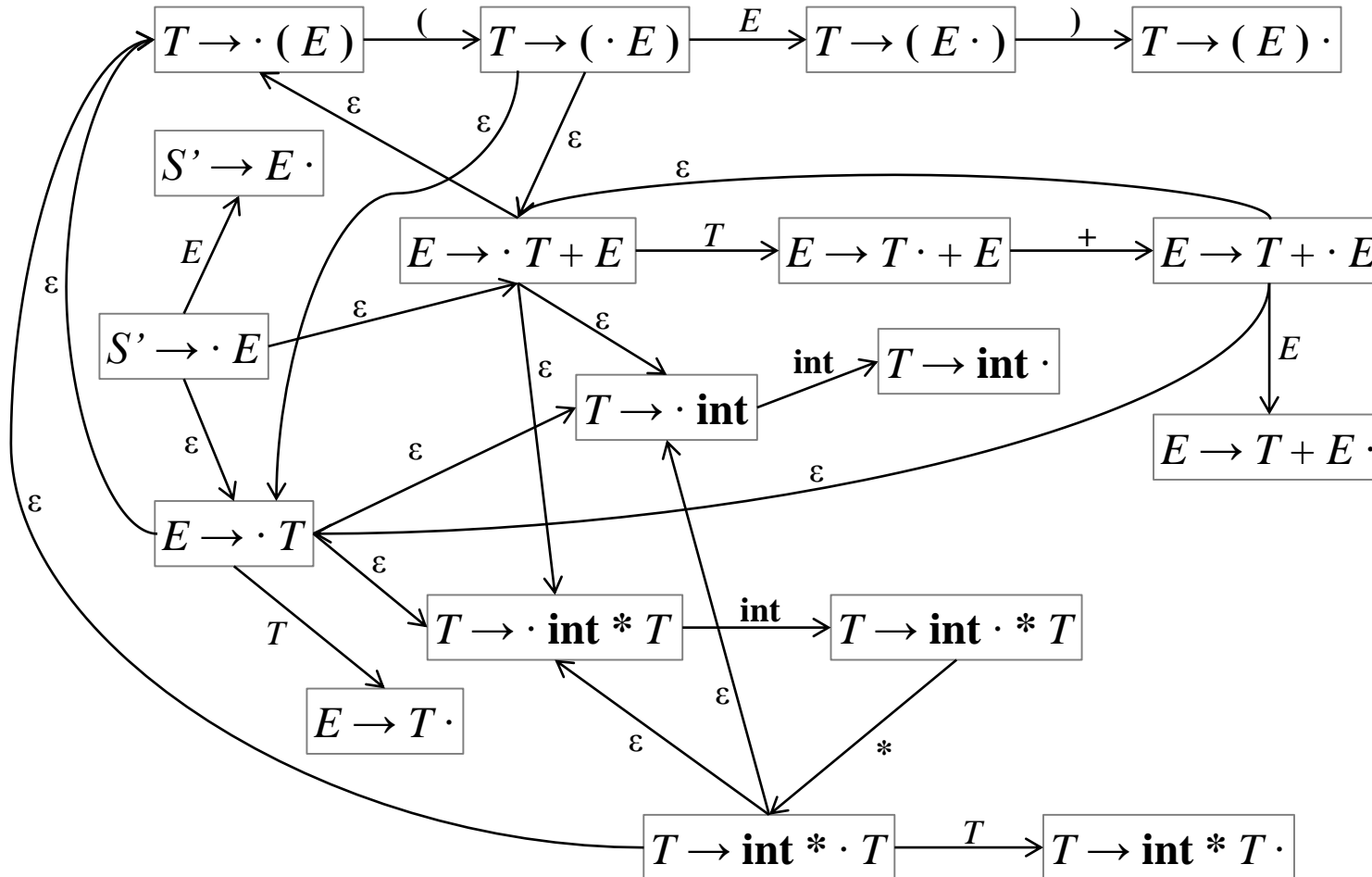
■ Example:



Recognizing Viable Prefixes

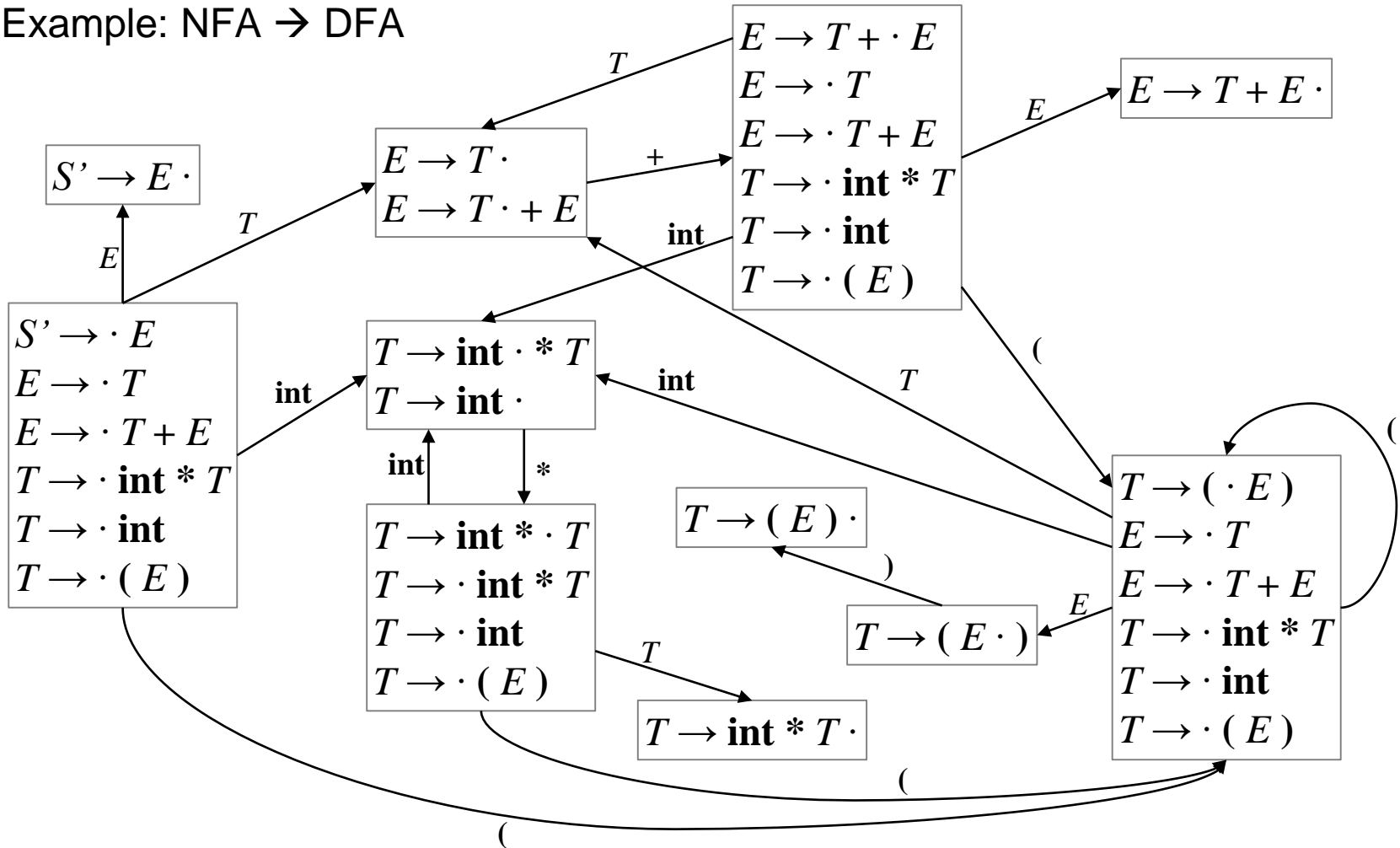
$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} / (E) \end{aligned}$$

■ Example:



Recognizing Viable Prefixes - DFA

■ Example: NFA → DFA



Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

- Input: augmented grammar of $G = G + \{S' \rightarrow S\}$
- CLOSURE of item sets

for a set I of items for a grammar G , $\text{CLOSURE}(I)$ is the set of items constructed by

- add every item in I to $\text{CLOSURE}(I)$
- if $A \rightarrow \alpha \cdot X \beta$ is in $\text{CLOSURE}(I)$, and $X \rightarrow \gamma$ is a production, then add the items $X \rightarrow \cdot \gamma$ unless already part of I
- repeat until no changes occur
- example: $I = \{ [S' \rightarrow \cdot E] \}$
 $\text{CLOSURE}(I) = \{ [S' \rightarrow \cdot E], \dots$

$$\begin{array}{lcl} S' & \rightarrow & E \\ E & \rightarrow & T + E \mid T \\ T & \rightarrow & \text{int} * T \mid \text{int} / (E) \end{array}$$

Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

- Observation: if a \cdot appears to the left of a non-terminal X , all productions of non-terminal X are added with the \cdot in the leftmost position
 - kernel items: initial item $S' \rightarrow S$, and all items whose \cdot are not in the leftmost position
 - non-kernel items: all items with the \cdot in the leftmost position (except $S' \rightarrow S$)

■ Algorithm CLOSURE

```
set<Item> CLOSURE (set<Item> I)
{
    J = I;
    repeat
        for each item  $A \rightarrow \alpha \cdot X \beta$  in J do
            for each production  $X \rightarrow \gamma$  of G do
                if  $(X \rightarrow \cdot \gamma \notin J)$  then
                    add  $X \rightarrow \cdot \gamma$  to J
    until no changes to J occur
    return J
}
```

Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

- The $\text{GOTO}(I, X)$ function

$$\text{GOTO}(I, X) = \text{CLOSURE}(\{ [A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X \beta] \in I \})$$

$\text{GOTO}(I, X)$ defines the transfer function for the LR(0) automaton

Example:

$$I = \{ [E' \rightarrow \cdot E], [E \rightarrow E \cdot + T] \}$$

$$\begin{aligned} \text{GOTO}(I, +) = \{ & [E \rightarrow E + \cdot T], \\ & [T \rightarrow \cdot T * F], \\ & [T \rightarrow \cdot F], \\ & [F \rightarrow \cdot (E)], \\ & [F \rightarrow \cdot \mathbf{id}] \} \end{aligned}$$

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

■ Algorithm to compute the LR(0) items

```
set<set<Item> > items(augmented grammar G')
{
    C = CLOSURE([S' → · S]);
    repeat
        for each set of items I in C do
            for each grammar symbol X in G' do
                if (GOTO(I, X) is not empty and not in C) then
                    add GOTO(I, X) to C
    until no changes to C occur
    return C
}
```


Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

$E' \rightarrow E$

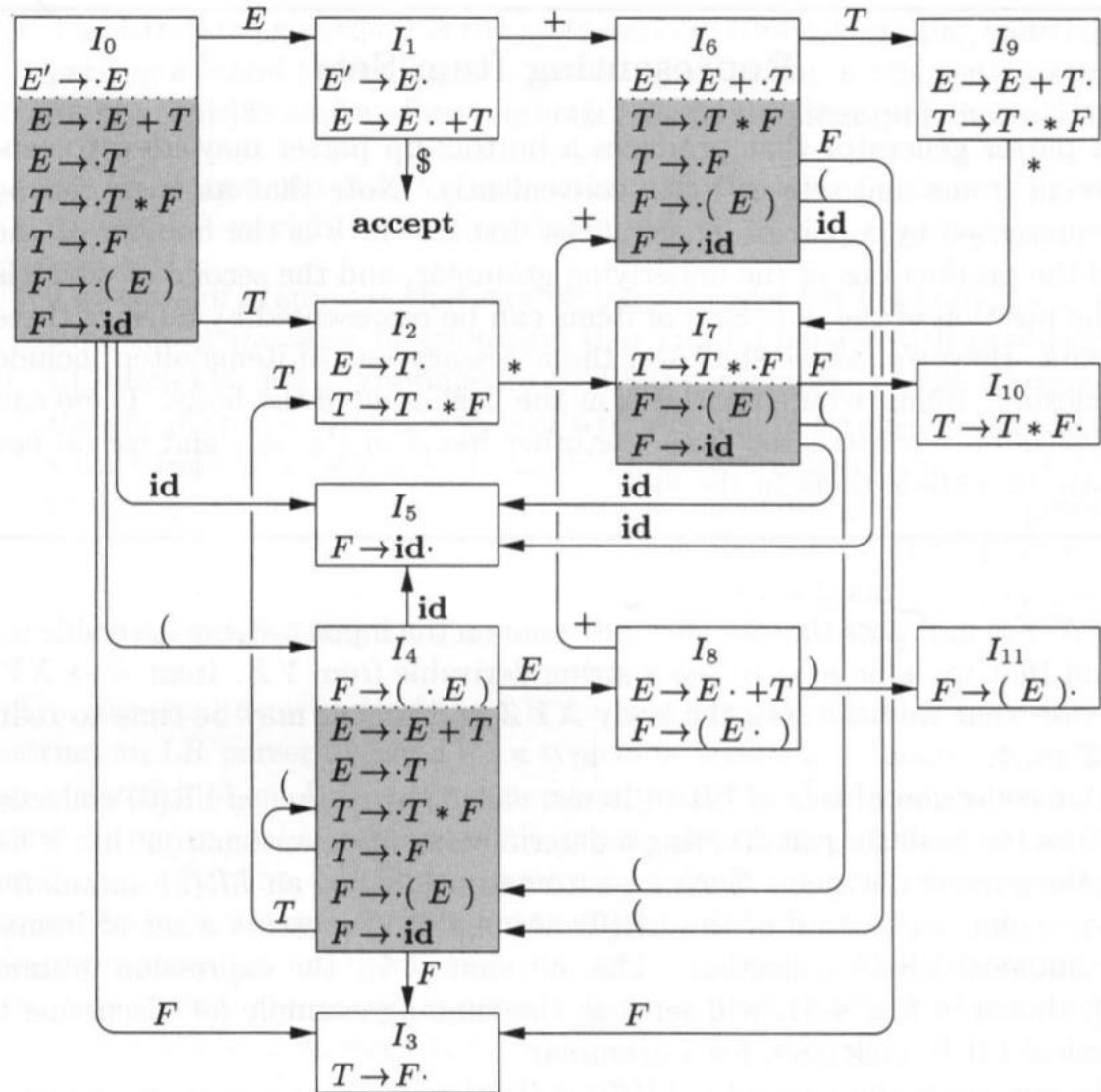
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

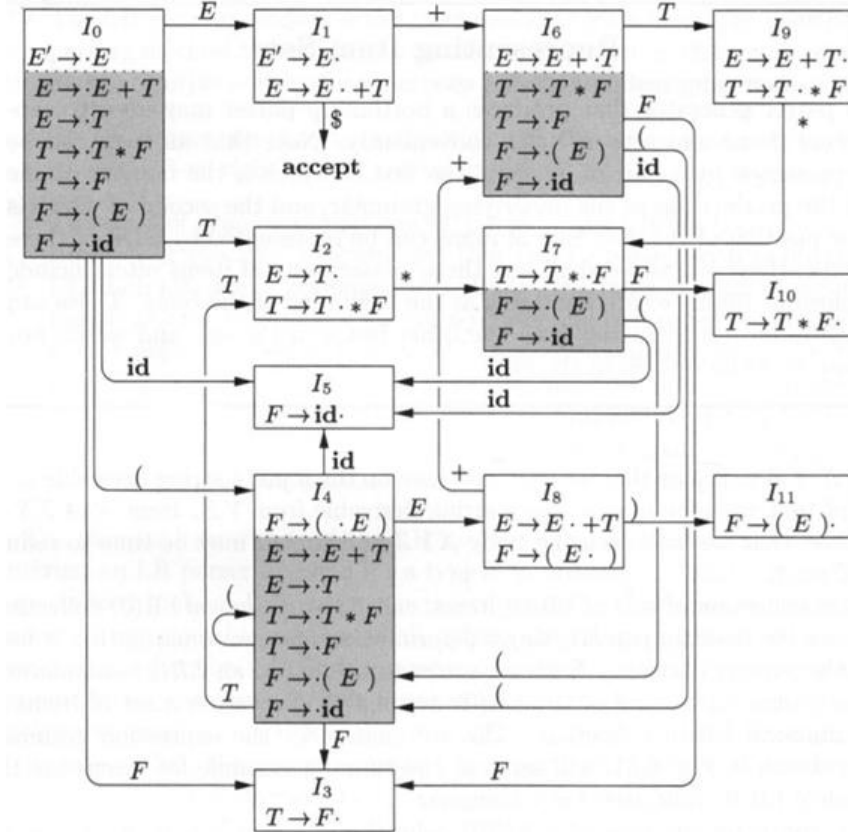
$E' \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid \text{id}$



Constructing the Canonical Collection of LR(0) items using CLOSURE and GOTO

■ Observations

- each set of items is the closure of its kernel items:
 - ▶ kernel items of $I_0 = \{ [E' \rightarrow \cdot E] \}$
 - ▶ closure of $\{ [E' \rightarrow \cdot E] \}$ adds all non-kernel items and gives I_0
- i.e., the canonical collection of LR(0) items can be represented by the kernel items only and thus with very little storage (at the expense of computing the closure of the kernel items whenever needed)



Valid Items

- Item $X \rightarrow \beta \cdot \gamma$ is valid for a viable prefix $\alpha\beta$ if there is a derivation

$$S' \xRightarrow{*}_{rm} \alpha X \omega \xRightarrow{rm} \alpha \beta \gamma \omega$$

i.e., the valid items are the possible items on top of the stack after seeing $\alpha\beta$.

In terms of the DFA, the valid items for a viable prefix α comprise the set of items described by the state the DFA is in after seeing input α .

- One item may be (and often is) valid for many prefixes
 - example: item $T \rightarrow (\cdot E)$ is valid for the prefixes $(, ((, (((, ((((, \dots$

LR(0) Parsing

■ LR(0) Parser

- next input t
- stack contains α
- LR(0) DFA recognizing viable prefixes
 - ▶ terminates in state s on input α
- actions:
 - ▶ if s contains the item $X \rightarrow \beta \cdot$ then **reduce** $X \rightarrow \beta$
 - ▶ if s contains the item $X \rightarrow \beta \cdot t$ then **shift**

LR(0) Parsing

■ Reduce-reduce conflict

- occur in any state s that contains two items $X_1 \rightarrow \beta_1 \cdot, X_2 \rightarrow \beta_2 \cdot$
 - ▶ reduce by $X_1 \rightarrow \beta_1$ or $X_2 \rightarrow \beta_2$?
 - ▶ lack of information, non-deterministic parse

■ Shift-reduce conflict

- occur if the final state of the DFA contains two items $X_1 \rightarrow \beta_1 \cdot, X_2 \rightarrow \beta_2 \cdot \mathbf{\$}$
 - ▶ reduce by $X_1 \rightarrow \beta_1$ or shift?
 - ▶ can be solved by introducing precedence rules

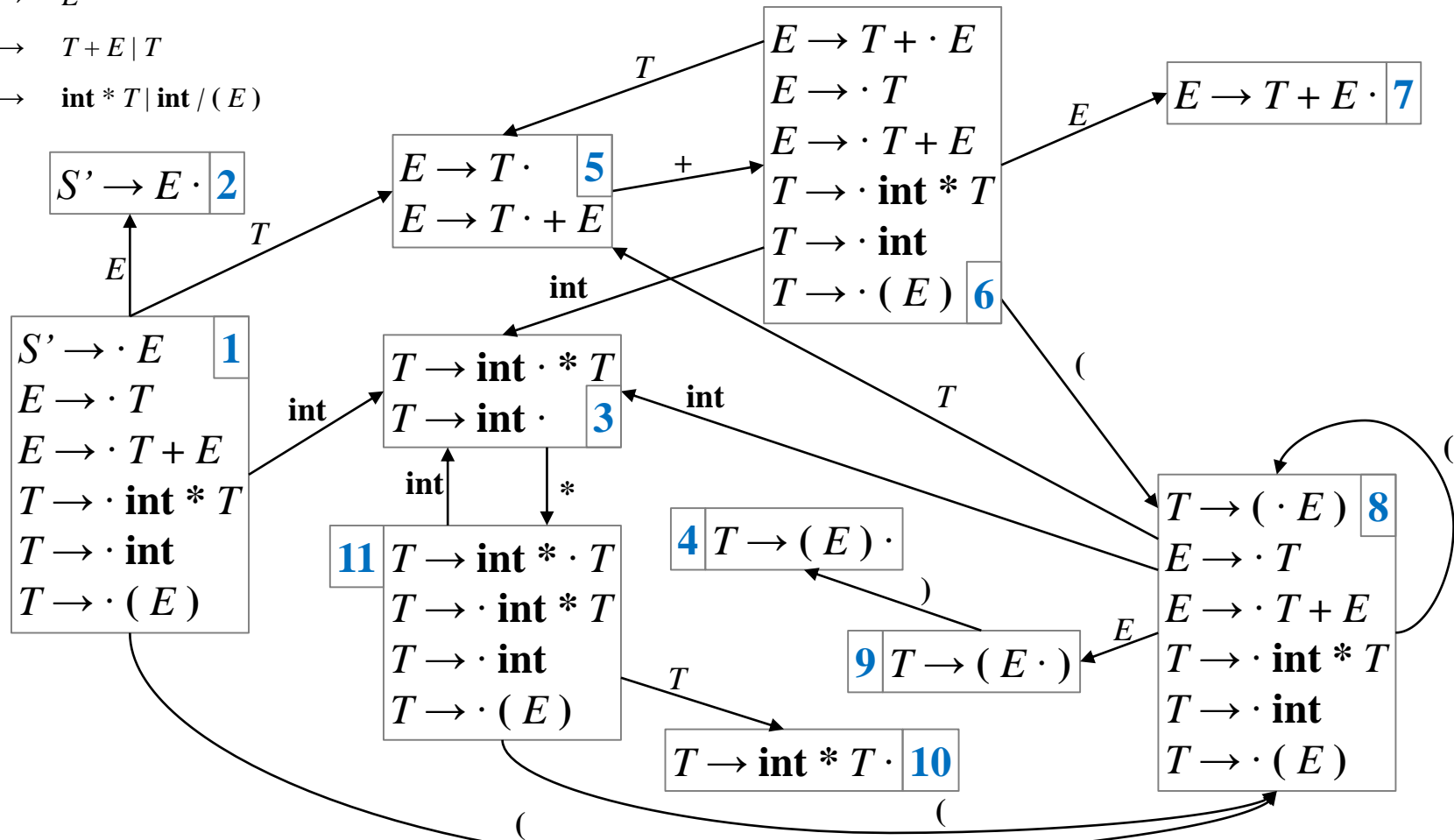
LR(0) Parsing

■ Example: LR(0) DFA for

$S' \rightarrow E$

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} / (E)$



LR(0) Parsing

$$\begin{aligned} S' &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

- Example: LR(0) DFA
 - shift-reduce conflicts

