

Algoritmi di Parsing: Bottom-up Parsing

Slides based on material
by Ras Bodik available at
<http://inst.eecs.berkeley.edu/~cs164/fa04>

Welcome to the running example

- we'll build a parser for this grammar:

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

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

(non ambiguous grammar with no useless variables)

- see, the grammar is

- left-recursive

- not left-factored

- ... and our parser won't mind!

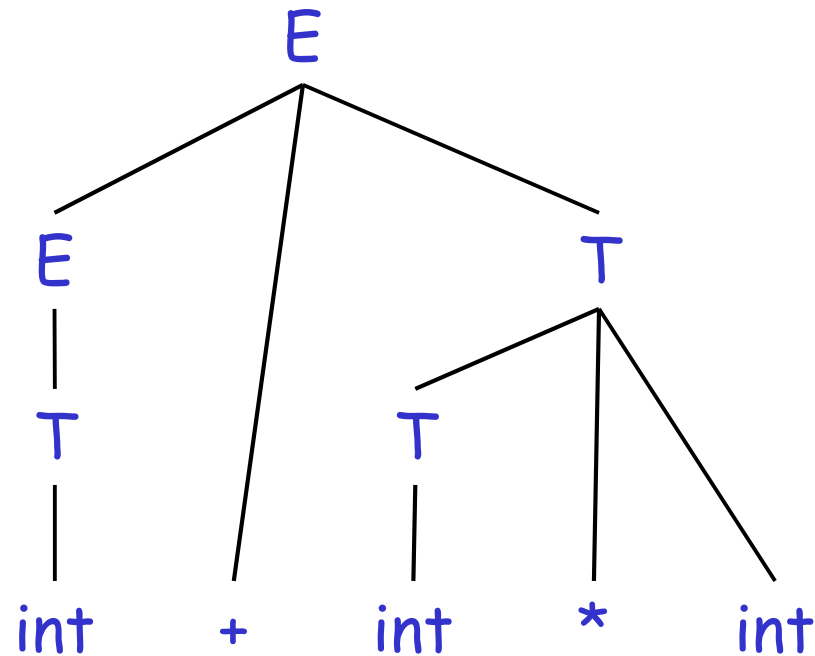
Gli algoritmi di parsing Bottom-Up (in particolare quelli LR) sono molto più potenti dei parser Top-Down (LL). I parser LR possono gestire grammatiche ricorsive a sinistra e molte grammatiche che non sono fattorizzate a sinistra senza necessità di riscriverle, rendendole ideali per la sintassi dei linguaggi di programmazione reali.

Example input, parse tree

- input:

int + int * int

- its parse tree:



1

Chaotic bottom-up parsing

Derivazione al Contrario

Key idea: build the derivation in reverse

(Riduzione da Corpo della
Produzione a Testa della Produzione)

E

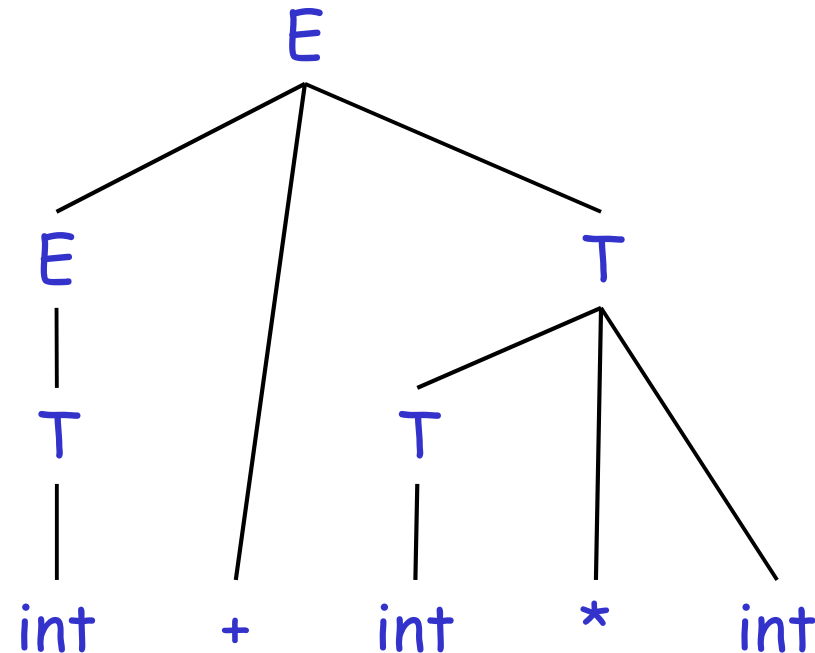
E + T

T + T

T + T * int

int + T * int

int + int * int



Guardo al punto in cui
sono arrivato ed
applico una produzione
al contrario (scelta a
caso).

Algo:
Guarda dove vuoi nella stringa (int + int * int). Trova
il corpo di una produzione presente nella stringa.
Riduci la stringa trovata alla testa. Continua fino a
che ho il Parse Tree (cioè arrivo alla radice)

Chaotic bottom-up parsing

- The algorithm:

1. stare at the input string s
 - feel free to look anywhere in the string
2. find in s a right-hand side r of a production $N \rightarrow r$ (trova una produzione a cui ridurre alla testa)
 - ex.: found int for a production $T \rightarrow \text{int}$
3. reduce the found string r into its non-terminal N (Riduzione)
 - ex.: replace int with T
4. if all string reduced to start non-terminal
 - we're done, string is parsed, we got a parse tree
5. otherwise continue in step 1

Don't celebrate yet!

NON SEMPRE FUNZIONA L'ALGORITMO (funziona solo se si ha fortuna di scegliere la giusta produzione da ridurre)

- not guaranteed to parse a correct string (Posso arrivare in un punto bloccante)
 - is this surprising?

- example:

and we are stuck

$E + E * E$

$E + E * T$

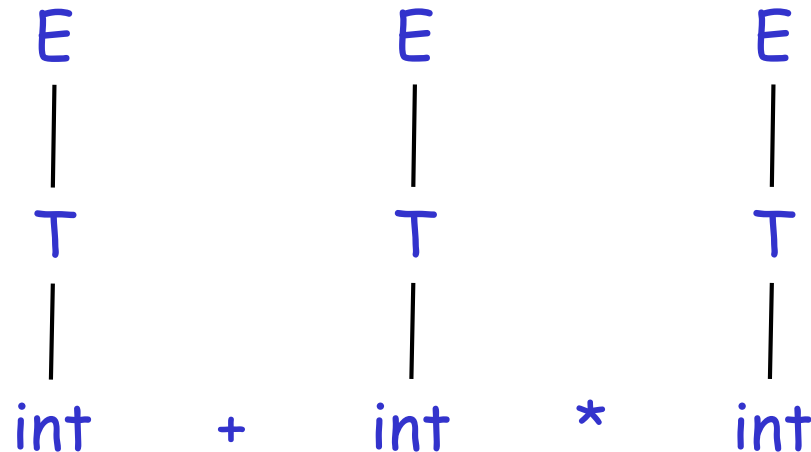
$E + E * \text{int}$

$T + E * \text{int}$

$\text{int} + E * \text{int}$

$\text{int} + T * \text{int}$

$\text{int} + \text{int} * \text{int}$



Lesson from chaotic parser

- Lesson:

- if you're lucky in selecting the string to reduce next, then you will successfully parse the string

→ Come posso superare il problema?

How to “beat the odds”?

- that is, how to find a lucky sequence of reductions that gives us a derivation of the input string?
- use non-determinism! (Prova tutte le possibili combinazioni di Riduzioni)

2

Se ci sono k modi per ridurre la stringa attuale, crea k copie identiche del parser, dove applicare una delle k riduzioni in ciascun copia del parser

Non-deterministic chaotic parser Inefficiente

The algorithm:

1. find in input all strings that can be reduced
 - assume there are k of them (cioé su quella stringa attuale, posso effettuare k riduzioni diverse)
2. create k copies of the (partially reduced) input
 - it's like spawning k identical instances of the parser
3. in each instance, perform one of k reductions
 - and then go to step 1, advancing and further spawning all parser instances
4. stop when at least one parser instance reduced the string to start non-terminal (Termina quando una copia trova il Parse Tree)

Properties of the n.d. chaotic parser

Claim:

- the input will be parsed by (at least) one parser instance

But:

- ¹ exponential blowup: $k * k * k * \dots * k$ parser copies

Also:

- ² Multiple (usually many) instances of the parser produce the correct parse tree (due to multiple corresponding derivations). This is wasteful.

Overview

- ① • Chaotic bottom-up parser
 - it will give us the parse tree, but only if it's lucky
- ② • Non-deterministic chaotic parser
 - creates many parser instances to make sure at least one builds the parse tree for the string
 - an instance either builds the parse tree or gets stuck
- ③ • Non-deterministic LR parser (next)
 - restrict where a reduction can be made
 - as a result, fewer instances necessary

Non-deterministic LR parser

→ Vogliamo avere un unico successo e la possibilità di clonare i parser (mantenere il non-determinismo)

- What we want:

- create multiple parser instances

- to find the lucky sequence of reductions

but the parse tree is found by at most ^{al massimo} one instance

- zero if the input has syntax error

②
1:2
→ Deve risolvere i due problemi
dell'algoritmo precedente
(Esplosione delle copie del
parser e Multiple soluzioni)

Riduzione (Reduce): rimpiazzo la cima dello stack (scelgo la produzione)
Shift: metto nuova stringa in cima allo stack per avere più riduzioni possibili

Two simple rules to restrict # of instances

Shift (left to right) and Reduce (right to left)

1. split the input in two parts:

- **right:** unexamined by parser
- **left:** in the parser (we'll do the reductions here)

Divido l'input in due parti: il parser esamina solo la parte a sinistra della freccia e la riduzione ammessa è quella vicina al punto di divisione delle due parti

$\text{int} \triangleright + \text{int} * \text{int}$ after reduction: $T \triangleright + \text{int} * \text{int}$

2. reductions allowed only on part adjacent to split

Derivazione canonica a destra (quindi l'algoritmo trova un'unica derivazione, se ha successo)

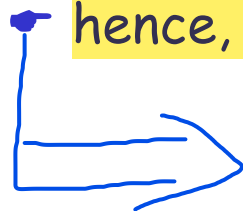
allowed: $T + \underline{\text{int}} \triangleright * \text{int}$

after reduction: $T + \underline{T} \triangleright * \text{int}$

not allowed: $\underline{\text{int}} + \text{int} \triangleright * \text{int}$

after reduction: $\underline{T} + \text{int} \triangleright * \text{int}$

hence, left part of string can be kept on the stack



Si può vedere come uno stack con cima a dx per capire quali riduzioni sono ammesse

Wait a minute!

Queste restrizioni non sono fatalmente severe?

Aren't these restrictions fatally severe?

nessuna istanza riesce
ad analizzare l'input

← the doubt: no instance succeeds to parse the input

No. Recall:

⇒ one parse tree corresponds to multiple derivations

- in n.d. chaotic parser, the instances that build the same parse tree each follow a different derivation

Le caratteristiche del parsing LR non impediscono di trovare il Parse Tree quando l'input è valido, perché:

- Se la stringa è valida, esiste un Parse Tree unico.
- Un Parse Tree unico implica una sola derivazione canonica a destra.
- L'algoritmo LR è progettato per trovare l'inverso di questa derivazione canonica a destra unica attraverso le sue operazioni di Shift e Reduce.

Wait a minute! (cont)

recall: two interesting derivations

- left-most derivation, right-most derivation

LR parser builds right-most derivation

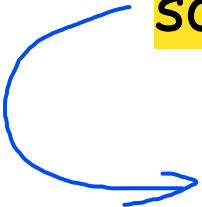
- but does so in reverse: first step of derivation is the last reduction (the reduction to start nonterminal)
- example coming in two slides

hence the name:

- L: scan input left to right
- R: right-most derivation

so, if there is a parse tree, LR parser will build it!

- this is the key theorem



Se la stringa di input appartiene al linguaggio definito dalla grammatica (cioè, se esiste un albero di parsing per quella stringa). Allora, l'algoritmo LR è garantito per trovare la corretta sequenza di Shift/Reduce che corrisponde all'inverso della derivazione più a destra, e quindi accettare l'input.

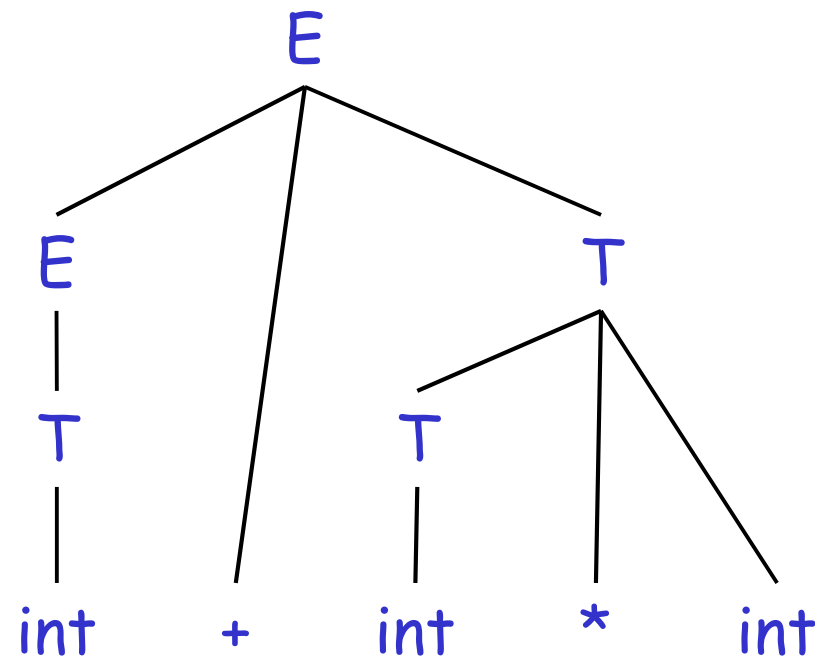
LR parser actions

- The left part of the string will be on the stack
 - the ► symbol is the top of stack
- Two simple actions
 - reduce:
 - like in chaotic parser,
 - but must replace a string on top of stack
 - shift:
 - shifts ► to the right,
 - which moves a new token from input onto stack, potentially enabling different reductions
- These actions will be chosen non-deterministically

Example of a correct LR parser sequence

A “lucky” sequence of shift/reduce actions (string parsed!):

E ▶
E + T ▶
E + T * int ▶
E + T * ▶ int
E + T ▶ * int
E + int ▶ * int
E + ▶ int * int
E ▶ + int * int
T ▶ + int * int
int ▶ + int * int
▶ int + int * int



Example of an incorrect LR parser sequence

will be stuck after reduction to $T + E$!

why can't we reduce to $E + T$, instead?

$T + T \triangleright$

$T + T * \text{int} \triangleright$

$T + T * \triangleright \text{int}$

$T + T \triangleright * \text{int}$

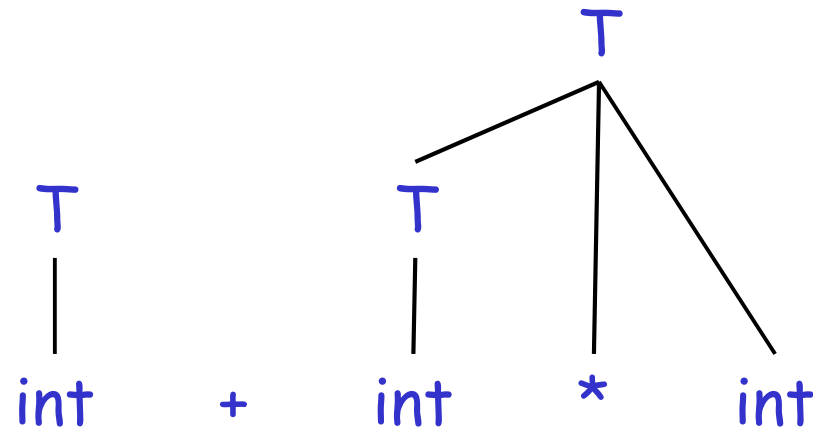
$T + \text{int} \triangleright * \text{int}$

$T + \triangleright \text{int} * \text{int}$

$T \triangleright + \text{int} * \text{int}$

$\text{int} \triangleright + \text{int} * \text{int}$

$\triangleright \text{int} + \text{int} * \text{int}$



Where did the parser instance make the mistake?

C'è sempre una derivazione corretta (a meno che ci siano degli errori sintattici) ed è una derivazione canonica a destra (il non determinismo permette di avere al massimo un successo in una istanza del parser e le altre istanze diventano bloccanti/fallite)

③

Non-deterministic LR parser

The algorithm: (compare with chaotic n.d. parser)

1. find all reductions allowed on top of stack
 - assume there are k of them
2. create k new identical instances of the parser
3. in each instance, perform one of the k reductions; in original instance, do no reduction, shift instead
 - and go to step 1
4. stop when a parser instance reduced the string to start non-terminal

Il parser esamina la cima del suo stack. Se i simboli in cima allo stack corrispondono al corpo di k diverse regole di produzione della grammatica, quelle k riduzioni sono possibili in quel momento.

Per gestire le scelte multiple, il parser si clona. Vengono create k nuove copie del parser, ognuna con lo stesso contenuto dello stack e lo stesso puntatore all'input.

(In ogni istanza, esegue una delle k riduzioni; nell'istanza originale, non esegue riduzioni, esegue invece uno shift)

- Istanze 1 a k (Reduce): Ogni nuova istanza esegue una delle k possibili riduzioni.
- Istanza Originale (Shift): L'istanza da cui sono partite le clonazioni non esegue riduzioni. Tenta invece l'operazione di Shift (sposta il prossimo simbolo di input sullo stack). Questo passaggio copre tutti i possibili conflitti che possono sorgere nel parsing Bottom-Up:
 - > Conflitto Shift/Reduce: Un simbolo permette sia una riduzione che uno shift. L'algoritmo crea due percorsi per esplorarle entrambe.
 - > Conflitto Reduce/Reduce: Più riduzioni sono possibili contemporaneamente. L'algoritmo crea un percorso per ognuna.
- Tutte le istanze (le k nuove ridotte e l'originale shiftata) ripetono l'intero processo tornando al Passo 1.

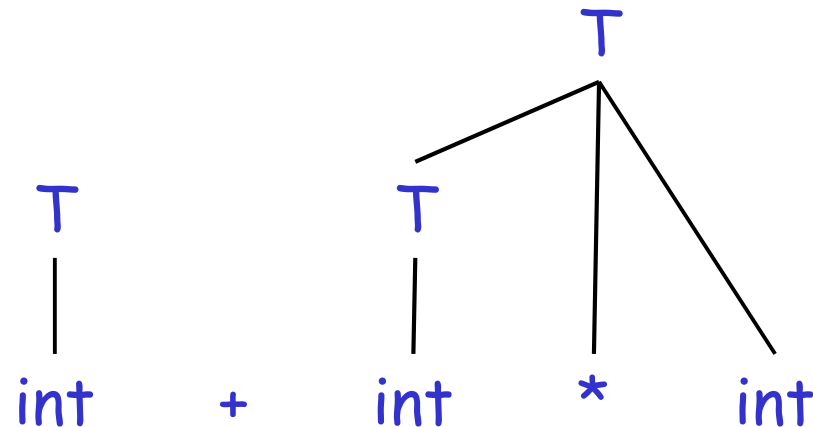
- Il parsing ha successo quando una singola istanza è riuscita a eseguire tutte le riduzioni necessarie, fino a che l'intero input è rappresentato dal solo simbolo iniziale sullo stack e l'input è esaurito.

Overview

- ① • Chaotic bottom-up parser
 - tries one derivation (in reverse)
 - ② • Non-deterministic chaotic parser
 - tries all ways to build the parse tree
 - ③ • Non-deterministic LR parser
 - restricts where a reduction can be made
 - as a result,
 - only one instance succeeds (on an unambiguous grammar)
 - all others get stuck
 - ④ • Generalized LR parser (next)
 - idea: kill off instances that are going to get stuck ASAP
- ↪ il prima possibile

Revisit the incorrect LR parser sequence

T + T ▶
T + T * int ▶
T + T * ▶ int
T + T ▶ * int
T + int ▶ * int
T + ▶ int * int
T ▶ + int * int
int ▶ + int * int
▶ int + int * int



Key question:

What was the earliest stack configuration where we could tell this instance was doomed to get stuck?

Doomed stack configurations

The parser made a mistake to shift to

$T + \triangleright \text{int} * \text{int}$

rather than reducing to

$E \triangleright + \text{int} * \text{int}$

The first configuration is doomed

- because the T will never appear on top of stack so that it can be reduced to E
- hence this instance of the parser can be killed (it will never produce a parse tree)

How to find doomed parser instances?

- Look at their stack!
- How to tell if a stack is doomed:
 - list all legal (non yet doomed) stack configurations
 - if a stack is not legal, kill the instance
- Listing legal stack configurations
 - list **prefixes (stack content) of all right-most derivations**
 - describe them as a **DFA**
 - if the stack configuration is not from the DFA, it's doomed

Our example grammar

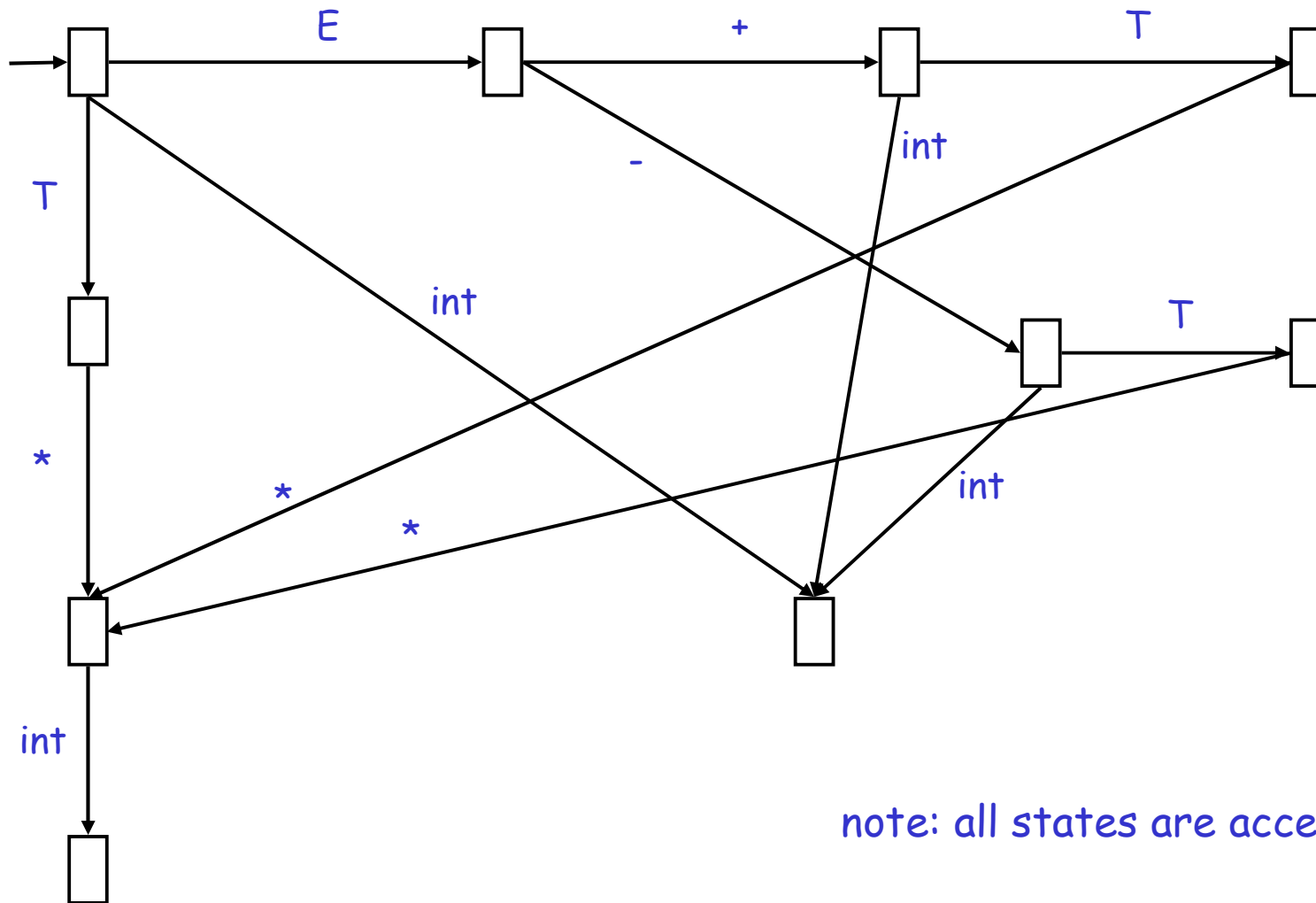
- we'll build a parser for this grammar:

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

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

- which are **prefixes (stack content)** of strings reached during **right-most derivations**?
 - E.g. those of length one.
 - They are "E", "T" and "int"
 - Then think about those of length two, etc...

The stack-checking DFA



note: all states are accepting states

Simple LR parsing

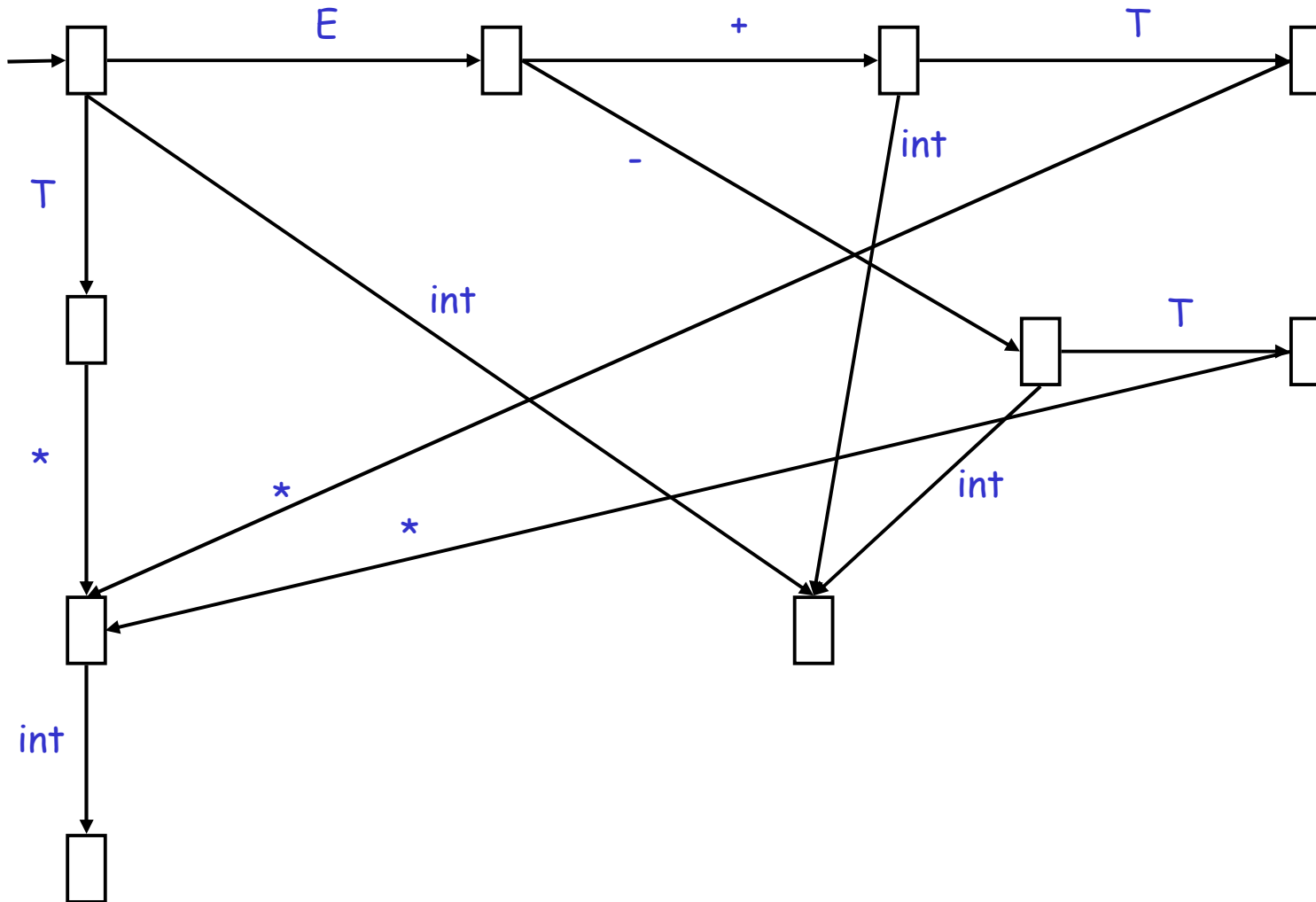
Introducing a new grammar initial variable

- In order to compute the stack checking DFA we preliminarily add to the grammar
 - a new initial variable E' and
 - a new production $E' \rightarrow E$
(with E being the old initial variable)
- Our example grammar thus becomes:

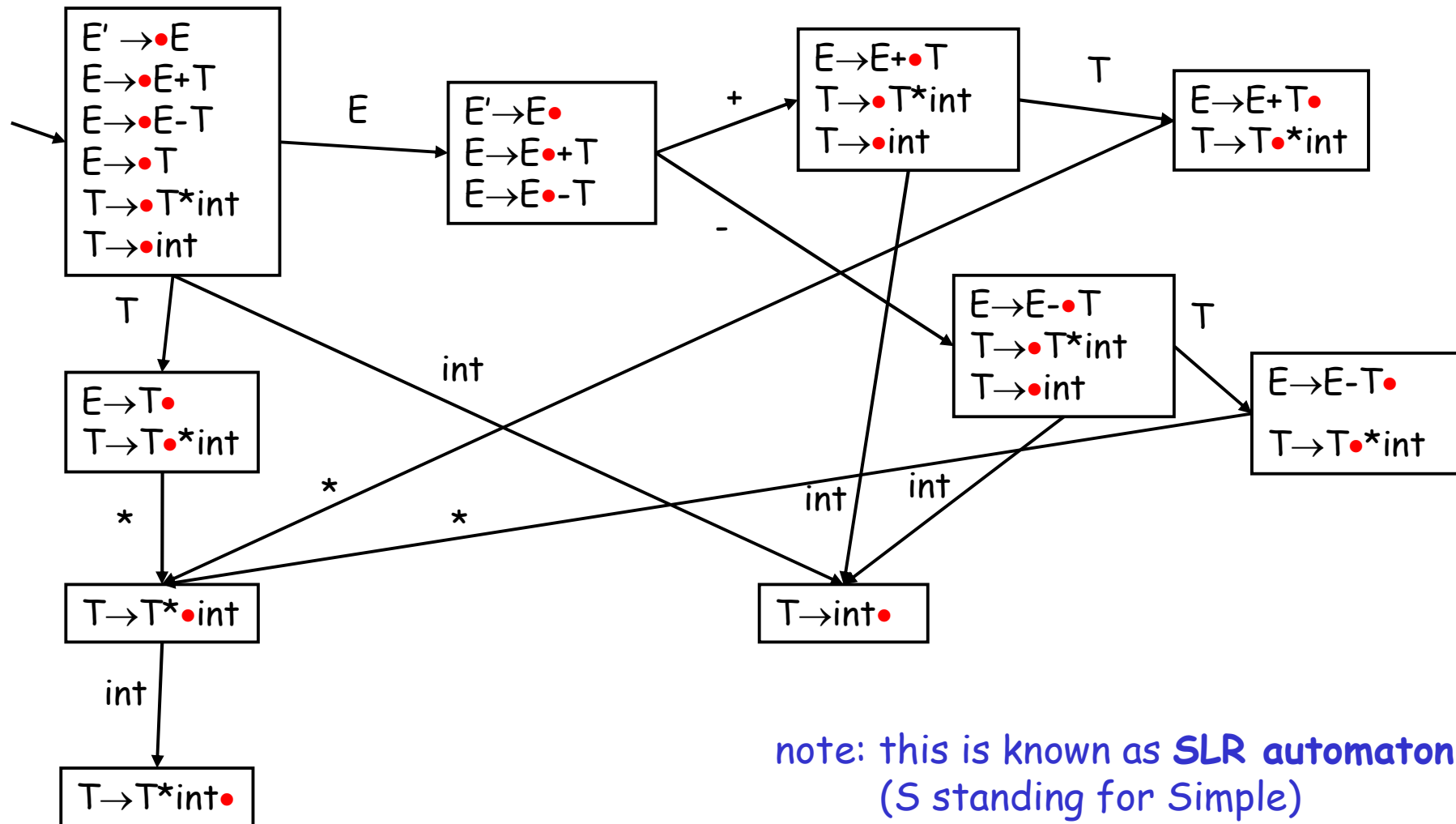
$$E' \rightarrow E$$

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

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



Constructing the stack-checking DFA



SLR Items

- A *SLR item* has the form:

$$X \rightarrow \alpha \bullet \beta$$

with $X \rightarrow \alpha\beta$ being a production

- $X \rightarrow \alpha \bullet \beta$ describes **context** information
 - We are trying to find an X , and
 - We have α already on top of the stack
 - Thus we need to see next a string derived from β

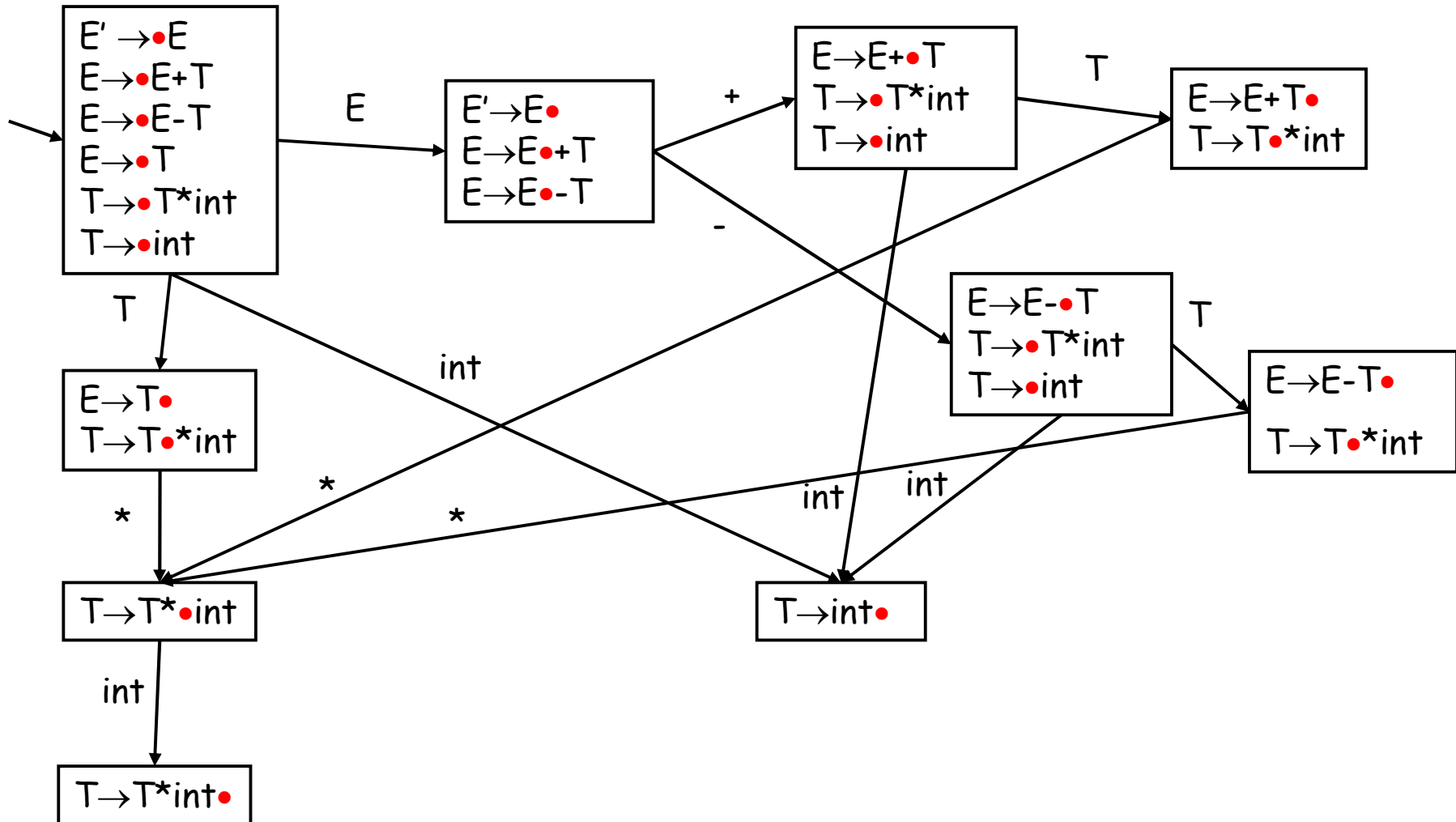
DFA state: a set of SLR items

- A DFA state describes a **parsing context**: a **set of SLR items** representing possible productions to apply
 - We are trying to find $E \rightarrow E + \bullet T$
 - on the top of the stack we have $E +$ already
 - Thus we are also trying to find $T \rightarrow \bullet T * \text{int}$ or $T \rightarrow \bullet \text{int}$
 - on the top of the stack we do not have anything yet

Note

- The symbol \blacktriangleright was used before to separate the stack from the rest of input
 - $\alpha \blacktriangleright \gamma$, where α is the stack and γ is the remaining string of terminals
- In SLR items \bullet is used to mark a prefix of a production rhs:
$$X \rightarrow \alpha \bullet \beta$$
 - Here β might contain non-terminals as well
- In both cases the stack is on the left

Constructing the stack-checking DFA



Constructing the stack-checking DFA

- A DFA state is a *closed* set of SLR(1) items
 - This means that we performed Closure
- The start state is $\text{Closure}(\{E' \rightarrow \bullet E\})$

The Closure Operation

- The operation of extending the context with items is called the *closure operation*

Closure(*Items*) =

repeat

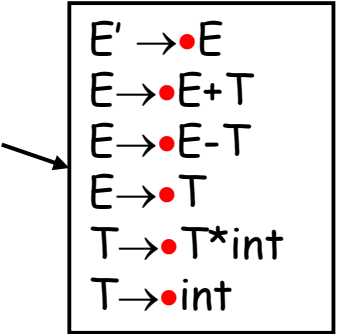
for each $X \rightarrow \alpha \bullet Y \beta$ in *Items*

for each production $Y \rightarrow \gamma$

add $Y \rightarrow \bullet \gamma$ to *Items*

until *Items* is unchanged

Constructing the stack-checking DFA



$E' \rightarrow \bullet E$
 $E \rightarrow \bullet E + T$
 $E \rightarrow \bullet E - T$
 $E \rightarrow \bullet T$
 $T \rightarrow \bullet T * \text{int}$
 $T \rightarrow \bullet \text{int}$

The DFA Transitions

- A state $State$ that contains at least an item $X \rightarrow \alpha \bullet y \beta$ has a transition labeled y to a state that contains the items: $Transition(State, y)$
 - y can be a terminal or a non-terminal

$Transition(State, y) =$

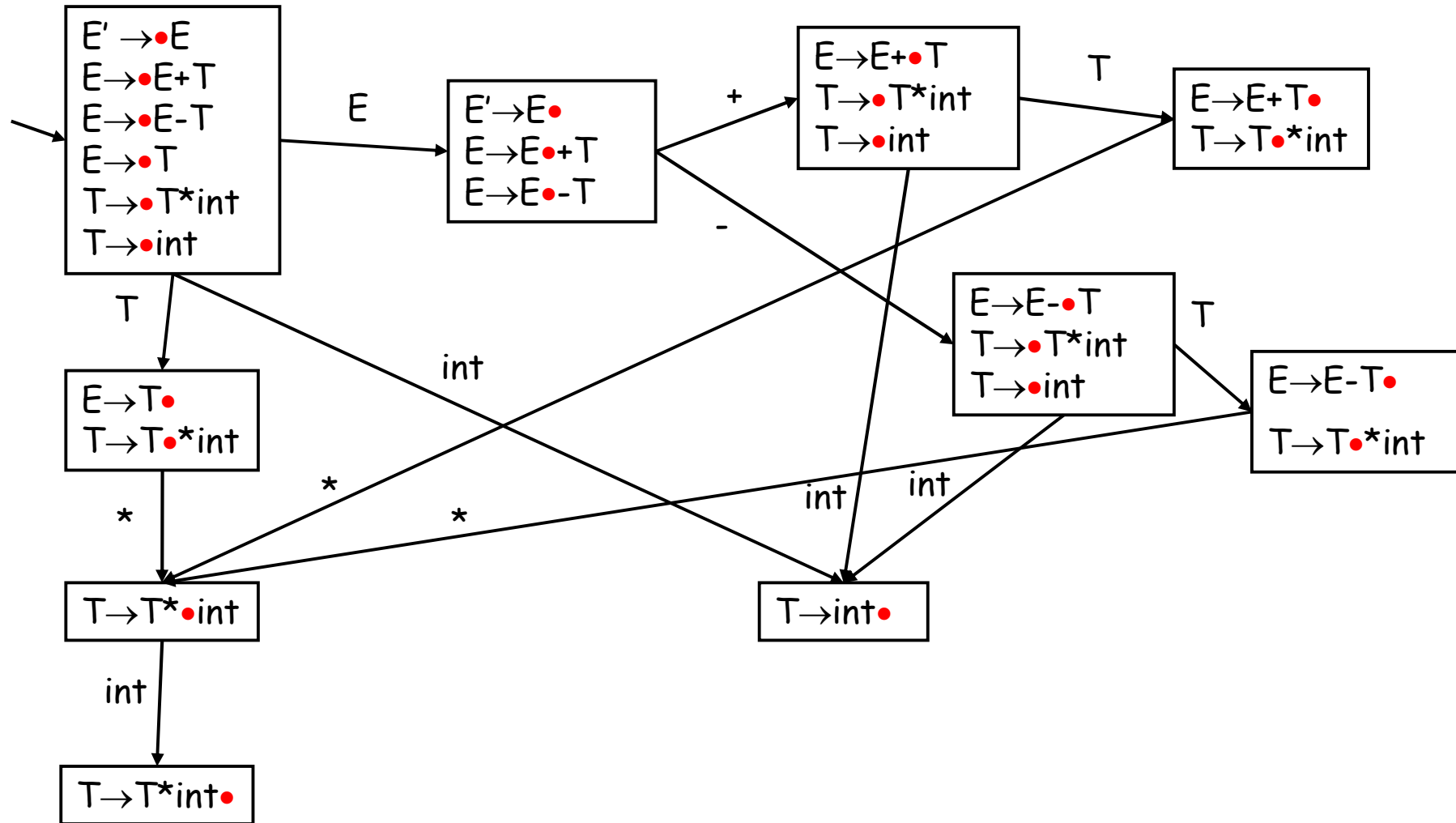
$Items \leftarrow \emptyset$

for each $X \rightarrow \alpha \bullet y \beta \in State$

add $X \rightarrow \alpha y \bullet \beta$ to $Items$

return $Closure(Items)$

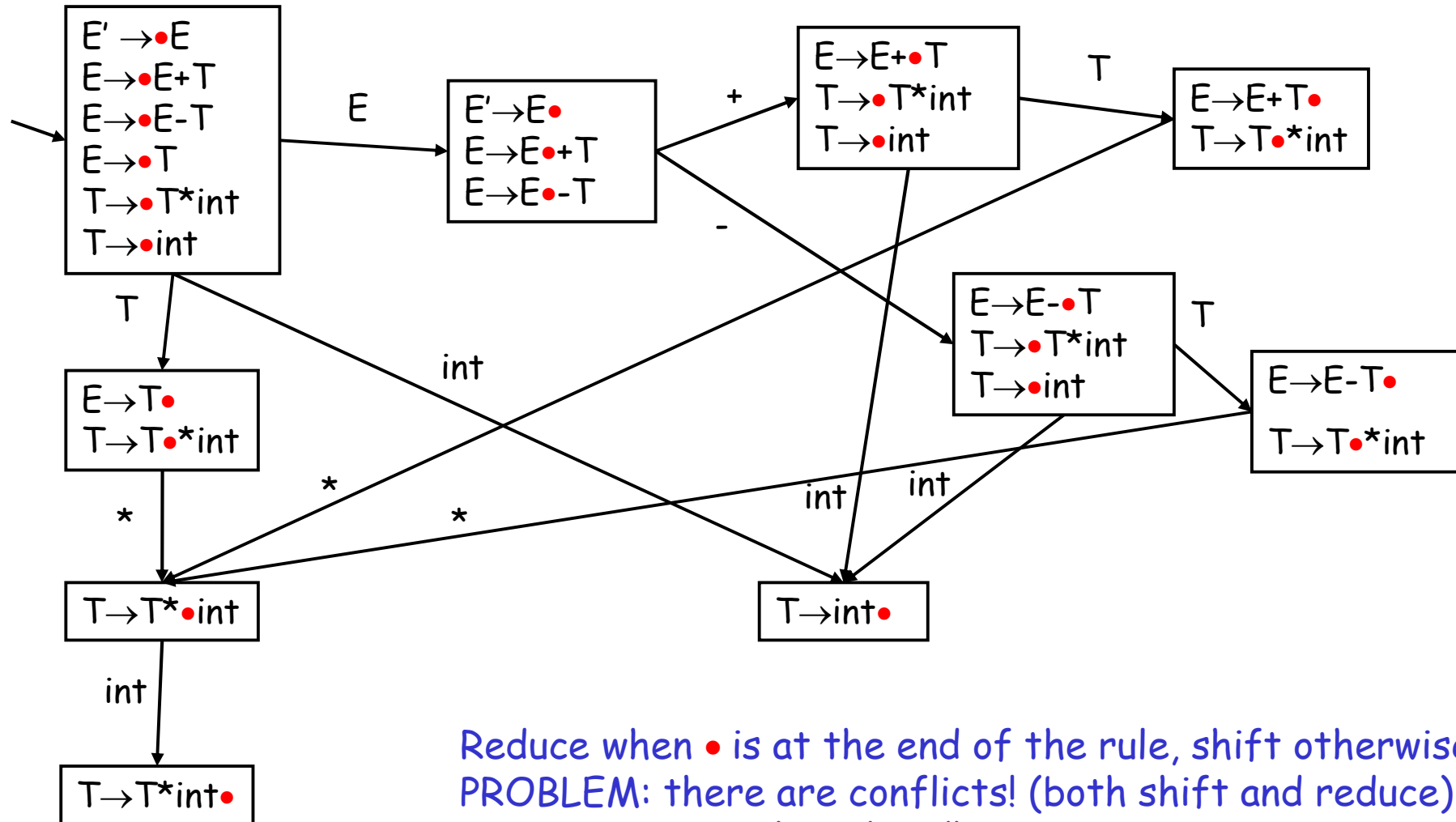
Constructing the stack-checking DFA



How to use the SLR automaton

- The automaton indicates the correct stack configurations...
- ...but it also dictates when to do shift/reduce actions...

Shift/reduce based on stack-checking DFA



Reduce when \bullet is at the end of the rule, shift otherwise
PROBLEM: there are conflicts! (both shift and reduce)
SOLUTION: use lookahead!

Example of a correct LR parser sequence

If we do not use lookahead ("*") we still have non-determinism!

The lookahead, instead, tells that here we cannot reduce $E+T$ to E

- because it would cause E to be followed by "*" and we know $* \notin \text{Follow}(E)$

$E \triangleright$

$E + T \triangleright$

$E + T * \text{int} \triangleright$

$E + T * \triangleright \text{int}$

$E + T \triangleright * \text{int}$

$E + \text{int} \triangleright * \text{int}$

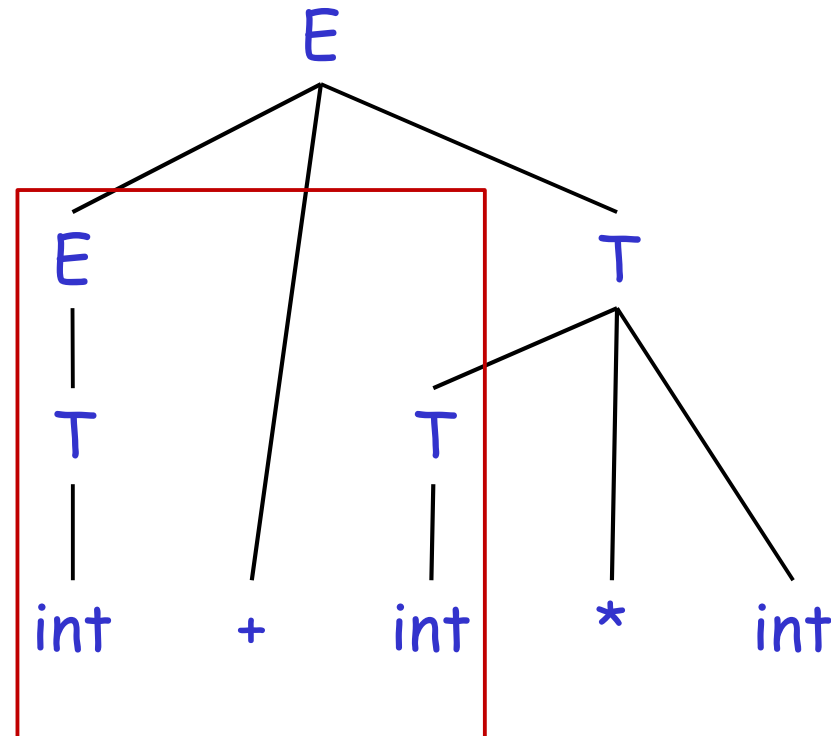
$E + \triangleright \text{int} * \text{int}$

$E \triangleright + \text{int} * \text{int}$

$T \triangleright + \text{int} * \text{int}$

$\text{int} \triangleright + \text{int} * \text{int}$

$\triangleright \text{int} + \text{int} * \text{int}$



How to use the SLR automaton

- Lookahead for "shift":
 - "Shift" only on the expected terminals
- Lookahead for "reduce":
 - "Reduce" only on terminals in the **Follow set** of the variable on the l.h.s. of the rule used to reduce

Computation of first and follow sets

- Consider our example grammar

$$E' \rightarrow E$$

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

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

$\text{First}(E') = \{\text{int}\}$ (no need to calculate it)

$\text{First}(E) = \{\text{int}\}$

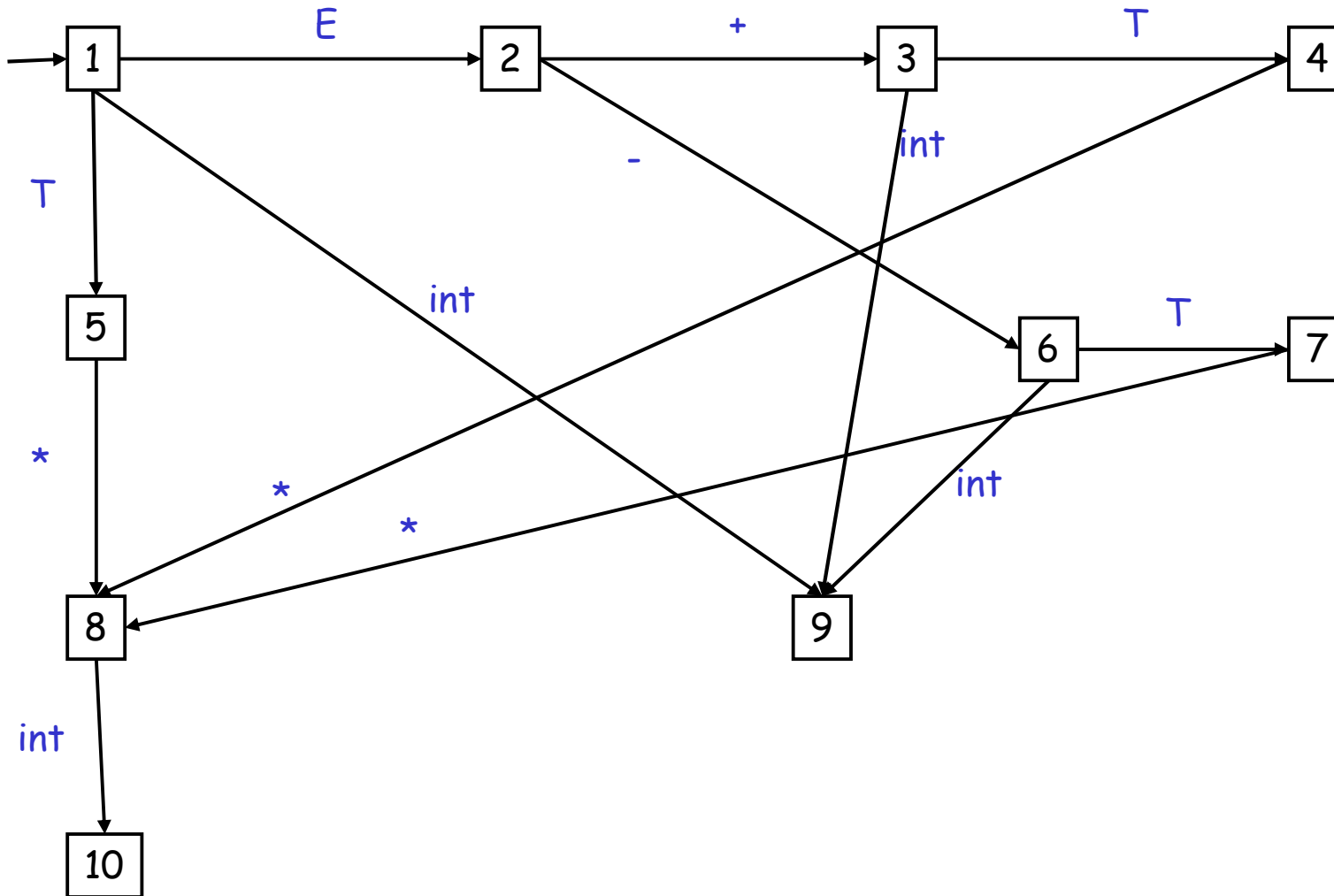
$\text{First}(T) = \{\text{int}\}$

$\text{Follow}(E') = \{\$\}$ (no need to calculate it, always $\{\$\}$)

$\text{Follow}(E) = \{\$, +, -\}$

$\text{Follow}(T) = \{\$, +, -, *\}$

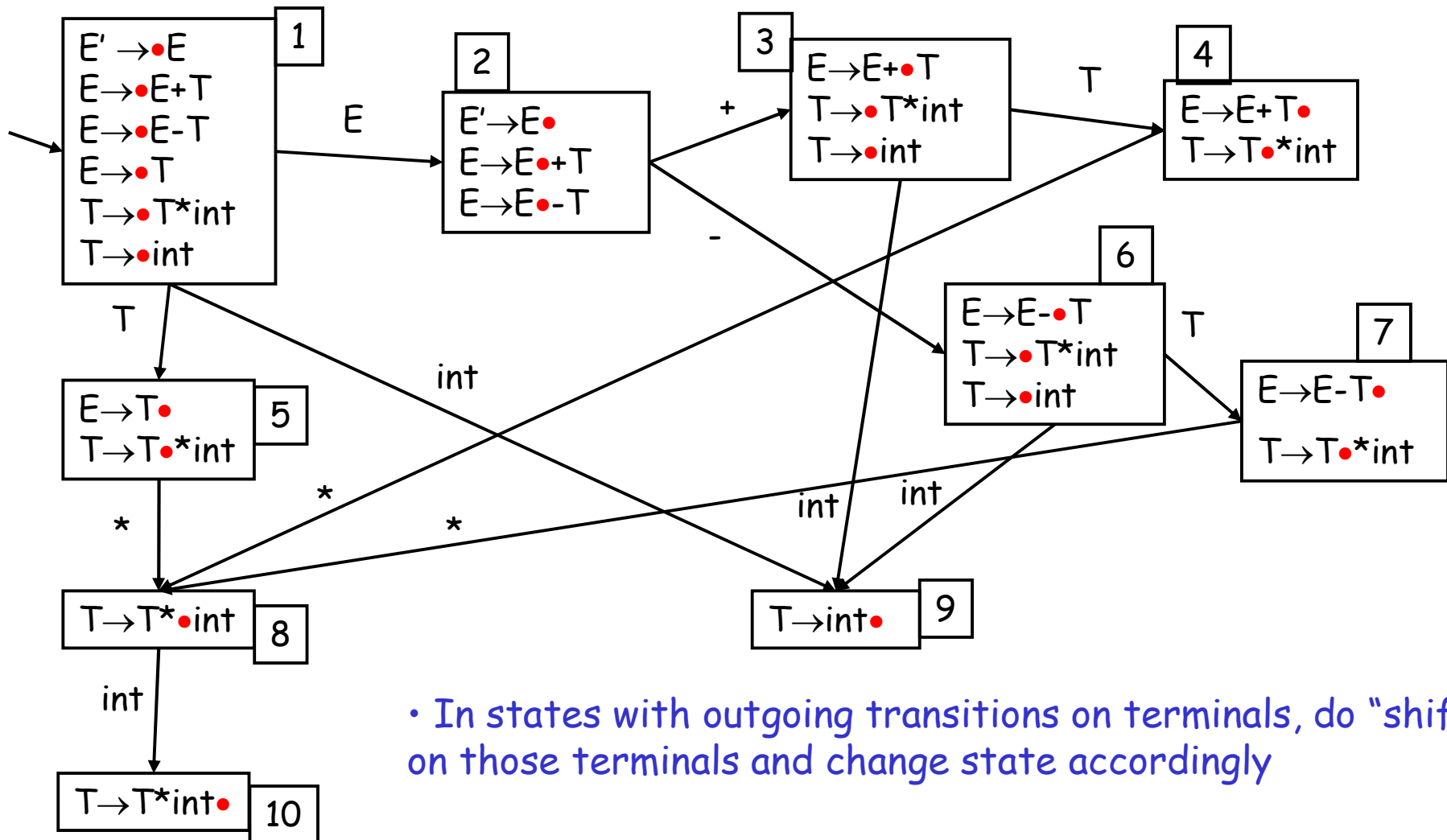
How to use the SLR automaton



How to use the SLR automaton (tabular representation)

	int	+	-	*	E	T
1	9				2	5
2		3	6			
3	9					4
4				8		
5				8		
6	9					7
7				8		
8	10					
9						
10						

How to use the SLR automaton

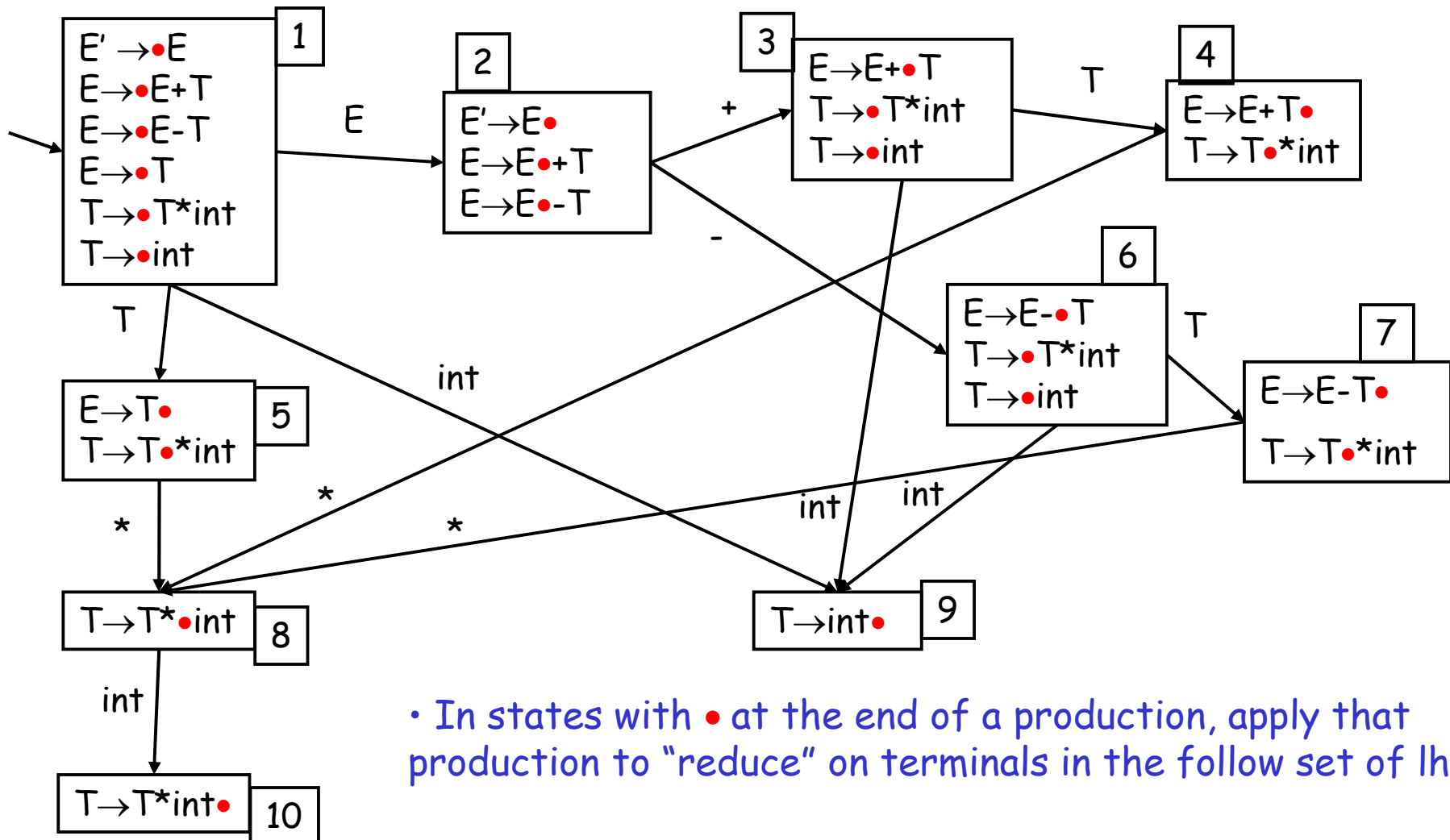


- In states with outgoing transitions on terminals, do "shift" on those terminals and change state accordingly

How to use the SLR automaton (tabular representation)

	int	+	-	*	E	T
1	shift,9				2	5
2		shift,3	shift,6			
3	shift,9					4
4				shift,8		
5				shift,8		
6	shift,9					7
7				shift,8		
8	shift,10					
9						
10						

How to use the SLR automaton

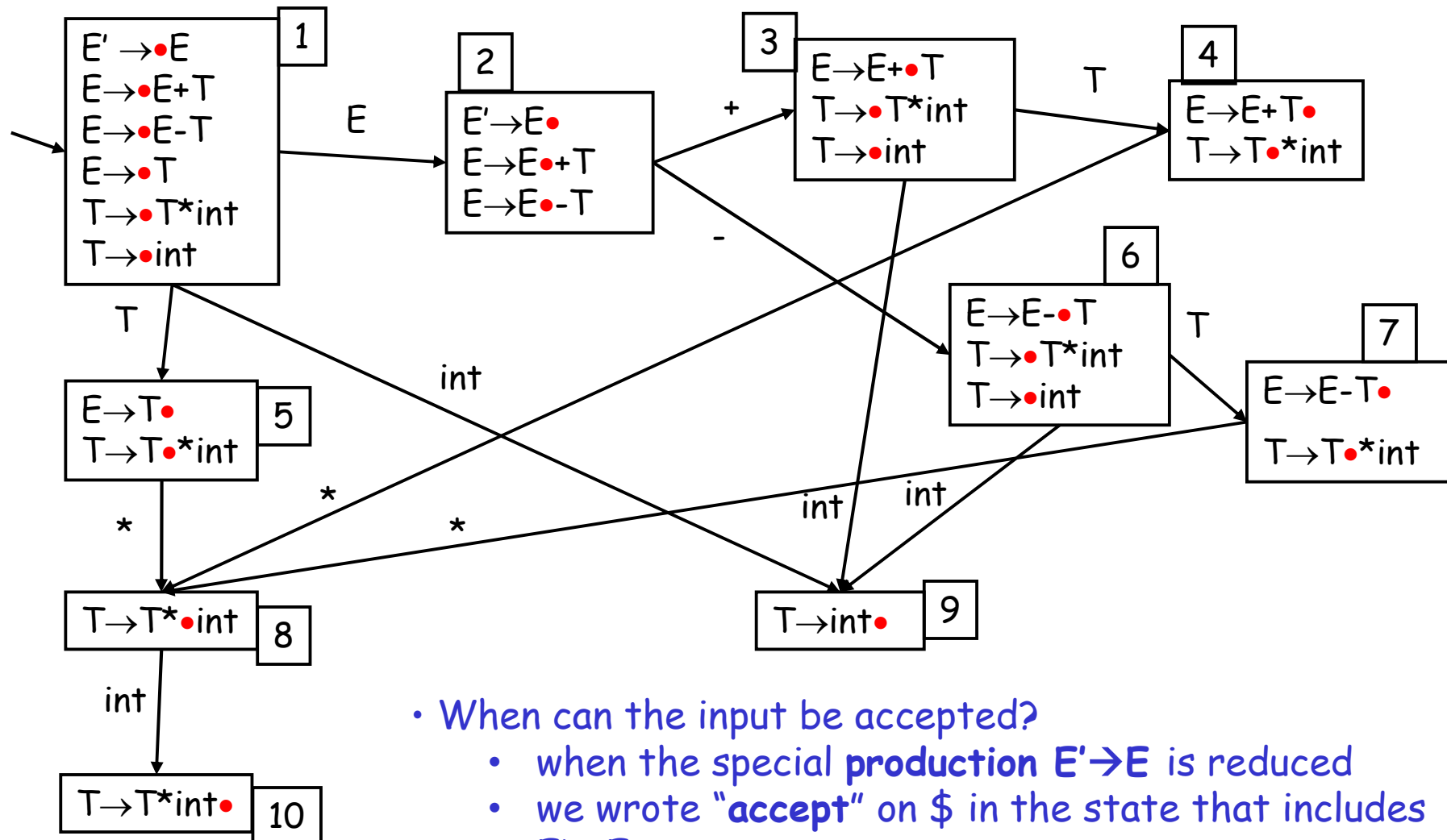


- In states with \bullet at the end of a production, apply that production to "reduce" on terminals in the follow set of lhs

How to use the SLR automaton (tabular representation, \$ column added)

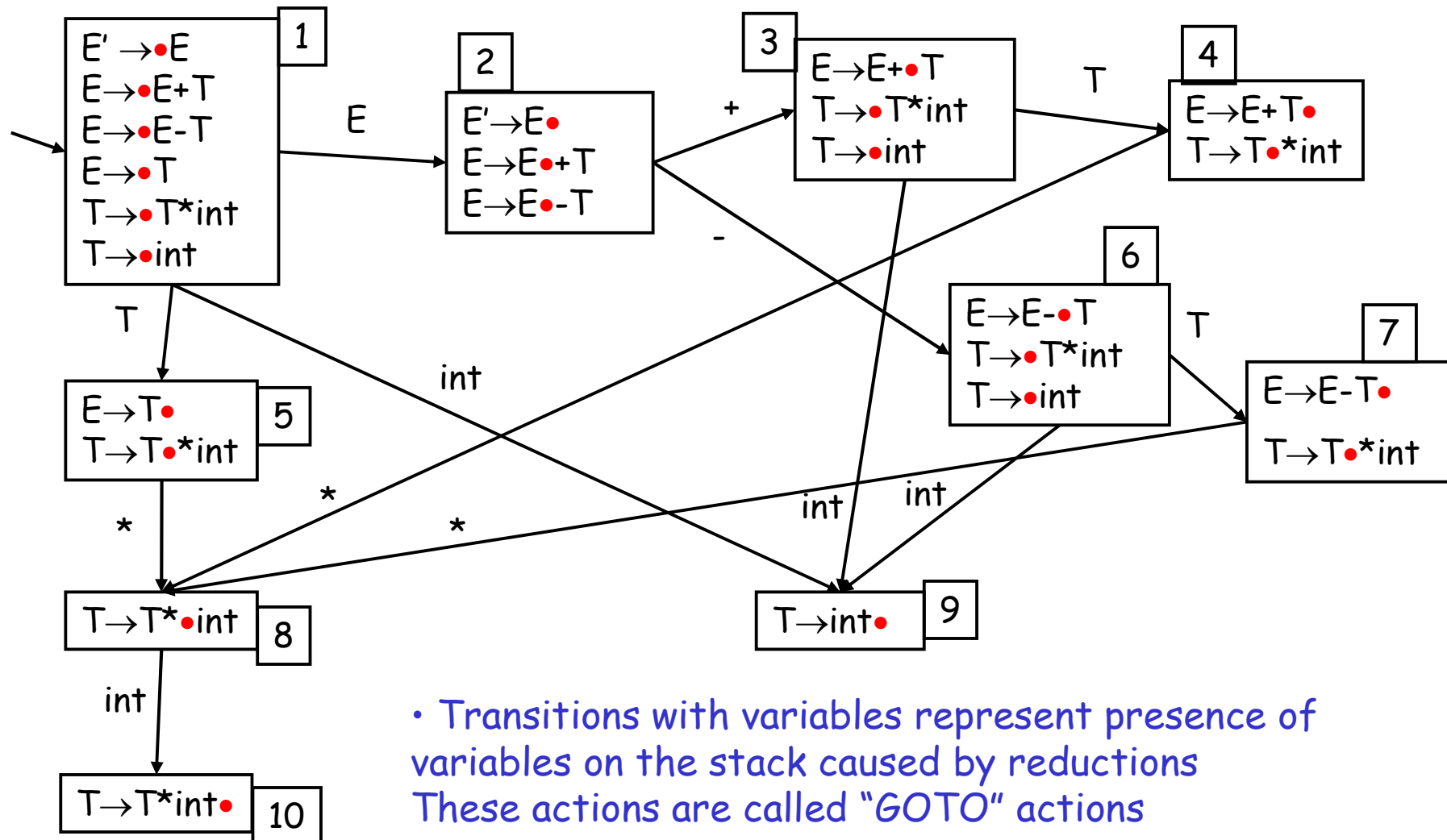
	int	+	-	*	\$	E	T
1	shift,9					2	5
2		shift,3	shift,6		accept		
3	shift,9						4
4		red, $E \rightarrow E+T$	red, $E \rightarrow E+T$	shift,8	red, $E \rightarrow E+T$		
5		red, $E \rightarrow T$	red, $E \rightarrow T$	shift,8	red, $E \rightarrow T$		
6	shift,9						7
7		red, $E \rightarrow E-T$	red, $E \rightarrow E-T$	shift,8	red, $E \rightarrow E-T$		
8	shift,10						
9		red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$		
10		red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$		

How to use the SLR automaton



- When can the input be accepted?
 - when the special production $E' \rightarrow E$ is reduced
 - we wrote "accept" on \$ in the state that includes $E' \rightarrow E \bullet$

How to use the SLR automaton



- Transitions with variables represent presence of variables on the stack caused by reductions
These actions are called "GOTO" actions

How to use the SLR automaton (complete tabular representation)

	int	+	-	*	\$	E	T
1	shift,9					goto,2	goto,5
2		shift,3	shift,6		accept		
3	shift,9						goto,4
4		red, $E \rightarrow E+T$	red, $E \rightarrow E+T$	shift,8	red, $E \rightarrow E+T$		
5		red, $E \rightarrow T$	red, $E \rightarrow T$	shift,8	red, $E \rightarrow T$		
6	shift,9						goto,7
7		red, $E \rightarrow E-T$	red, $E \rightarrow E-T$	shift,8	red, $E \rightarrow E-T$		
8	shift,10						
9		red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$	red, $T \rightarrow \text{int}$		
10		red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$	red, $T \rightarrow T*\text{int}$		

How to use the SLR automaton

- The LR predictive algorithm governed by this table is called SLR(1) parsing algorithm
 - Note that it works only if each cell of the table contains at most one action! (i.e. no conflict)
 - In this case, the grammar is a SLR(1) grammar

Bottom-up parsing algorithm

Remember the idea of LR parsing

- LR parsing *reduces* a string to the start symbol by inverting productions:

str such that $\text{str} \rightarrow^* \text{input string of terminals}$
while $\text{str} \neq S$:

- Identify β in *str* such that $A \rightarrow \beta$ is a production and $S \rightarrow^* \alpha A \gamma \rightarrow \alpha \beta \gamma = \text{str}$
- Replace β by A in *str* (so $\alpha A \gamma$ becomes new *str*)
- Stronger than top-down parsing!
 - make decisions *after* seeing all symbols β rather than before (LL(1) make decisions knowing at most one of those symbols, the lookahead)

Bottom-up parsing algorithm

- Add \$ at the end of the input
- Execute the bottom-up parsing algorithm using an automaton (e.g. the SLR automaton) to decide whether to shift or reduce
 - Every time a decision should be taken, use the stack as input for the automaton
 - The automaton will communicate whether to shift or reduce (based on the lookahead)

A Running Example

- We will study the bottom-up parsing algorithm considering

- The grammar:

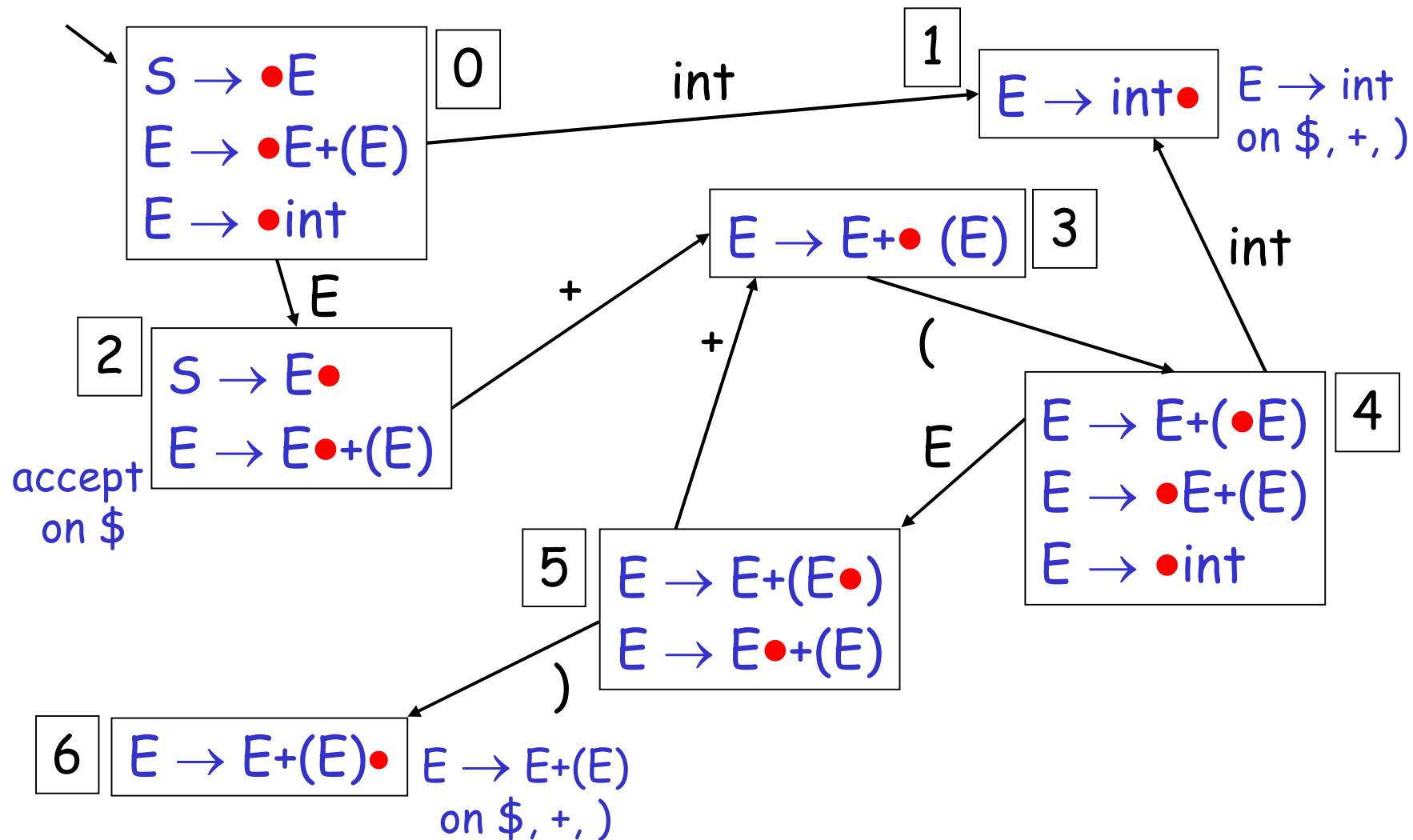
$$E \rightarrow E + (E) \mid \text{int}$$

- An automaton/parsing table for such a grammar, i.e. its **SLR(1)** automaton

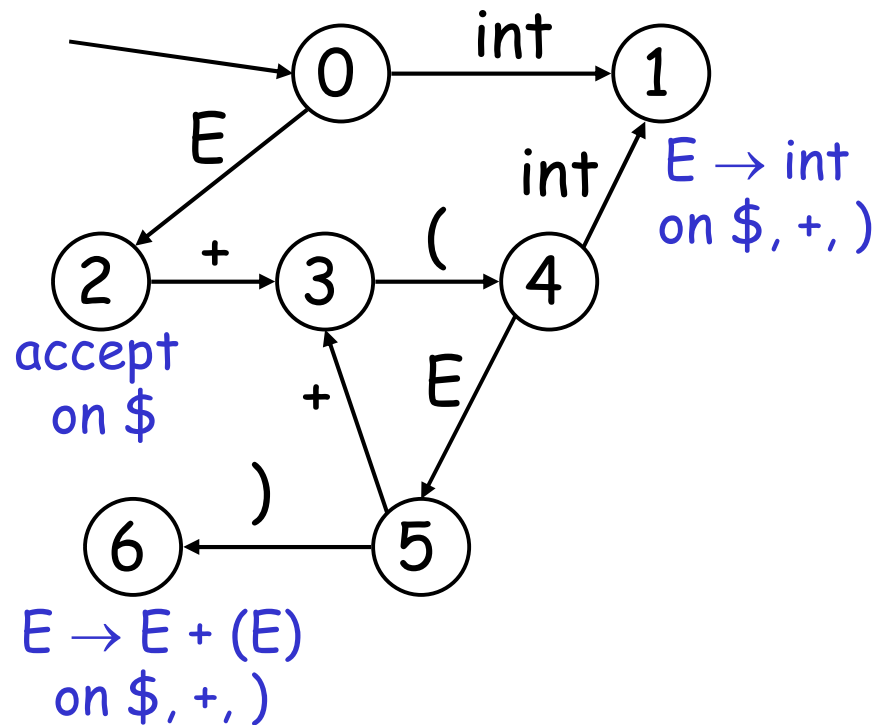
- $\text{First}(E) = \{\text{int}\}$

- $\text{Follow}(E) = \{\$, +,)\}$

Constructing the Parsing DFA. Example.



Bottom-up parsing algorithm example



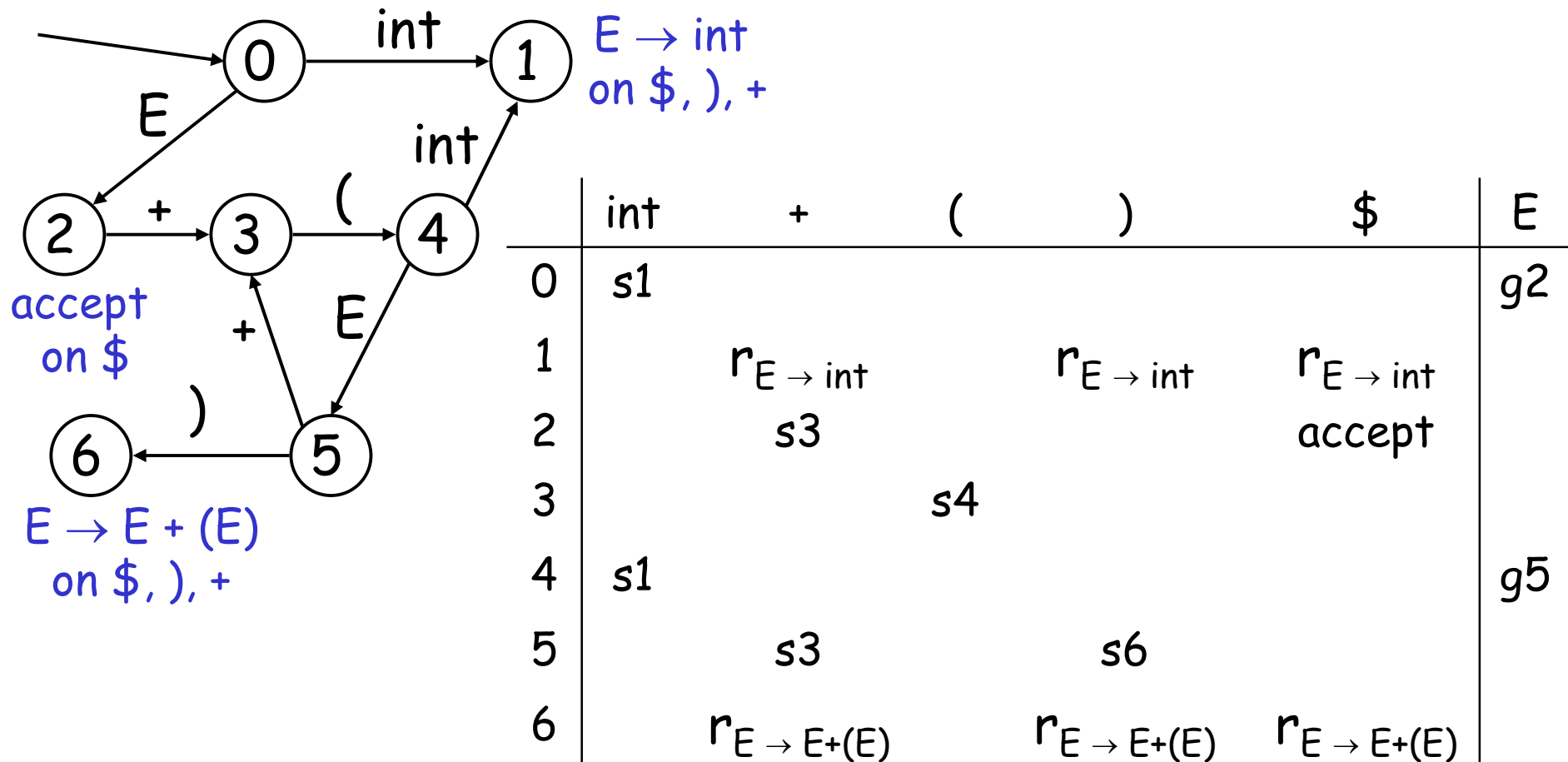
▶ $\text{int} + (\text{int}) + (\text{int})\$$ shift
 $\text{int} \triangleright + (\text{int}) + (\text{int})\$$ $E \rightarrow \text{int}$
 $E \triangleright + (\text{int}) + (\text{int})\$$ shift(x3)
 $E + (\text{int} \triangleright) + (\text{int})\$$ $E \rightarrow \text{int}$
 $E + (E \triangleright) + (\text{int})\$$ shift
 $E + (E) \triangleright + (\text{int})\$$ $E \rightarrow E+(E)$
 $E \triangleright + (\text{int})\$$ shift (x3)
 $E + (\text{int} \triangleright)\$$ $E \rightarrow \text{int}$
 $E + (E \triangleright)\$$ shift
 $E + (E) \triangleright \$$ $E \rightarrow E+(E)$
 $E \triangleright \$$ accept

DFA table representation is actually used

- Parsers represent the DFA as a 2D table
 - As for table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- In classical treatments, columns are split into:
 - Those for terminals: action table
 - Those for non-terminals: goto table

Representing the DFA. Example

- The table for our DFA:



The Bottom-up Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
 - This is wasteful, since most of the work is repeated
- So record, for each stack element, state of the DFA after that element
- Parser maintains a stack
$$\langle \text{sym}_1, \text{state}_1 \rangle \dots \langle \text{sym}_n, \text{state}_n \rangle$$
$$\text{state}_k \text{ is the final state of the DFA on } \text{sym}_1 \dots \text{sym}_k$$

The Bottom-up Parsing Algorithm

Let $I = w_1w_2\dots w_n\$$ be initial input

Let $j = 1$

Let DFA state 0 be the start state

Let $\text{stack} = \langle \text{dummy}, 0 \rangle$

repeat

case $\text{action}[\text{top_state}(\text{stack}), I[j]]$ of

shift k : push $\langle I[j], k \rangle$; $j += 1$

reduce $X \rightarrow \alpha$:

pop $|\alpha|$ pairs,

push $\langle X, \text{goto}[\text{top_state}(\text{stack}), X] \rangle$

accept: halt normally

error: halt and report error

bison: a parser generator

- **bison** generates parsers building the LALR(1) automaton/table
 - slightly more complex variant of SLR(1) such that a conflict-free table is generated for more grammars
- bison input:
 - file with token description, priority and associativity rules, grammar rules, and some auxiliary **C** instructions
- Bison generates two files in output:
 - a **C** program containing the code for the parser
 - the LALR(1) table

bison: structure of bison input

%{

C declarations

%}

Bison declarations

%%

Grammar rules

%%

Programs

bison: example of input

```
%{  
#include <stdio.h>  
int yylex() ;  
void yyerror(char *s){ printf("parser error") ; }  
%}  
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN  
%start prog  
%%  
prog      : stmlist;  
stmlist   : stm  
           | stmlist SEMI stm;  
stm       : ID ASSIGN ID  
           | WHILE ID DO stm  
           | BEGIN stmlist END  
           | IF ID THEN stm  
           | IF ID THEN stm ELSE stm;
```

bison: example of output

- In the generated table we read:

state 18 contains 1 shift/reduce conflict.

...

state 18

stm -> IF ID THEN stm . (rule 7)

stm -> IF ID THEN stm . ELSE stm (rule 8)

ELSE shift, and go to state 19

ELSE [reduce using rule 7 (stm)]

- If there are no explicit precedence declarations, BISON follows the following:
 - In shift/reduce conflicts the shift is executed
 - In reduce/reduce conflicts the first rule is applied