COSC252: Programming Languages Lecture: Bottom Up Parsing

Jeremy Bolton, PhD

Asst Professor

For the excellent slides: A very special thanks to the faculty at Stanford, and the authors of THE Dragon Book, Aho et al.

GEORGETOWN UNIVERSITY

Outline

I. Bottom Up Parsing

Different Types of Parsing

- **Top-Down Parsing**
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- **Bottom-Up Parsing**
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

What is Bottom-Up Parsing?

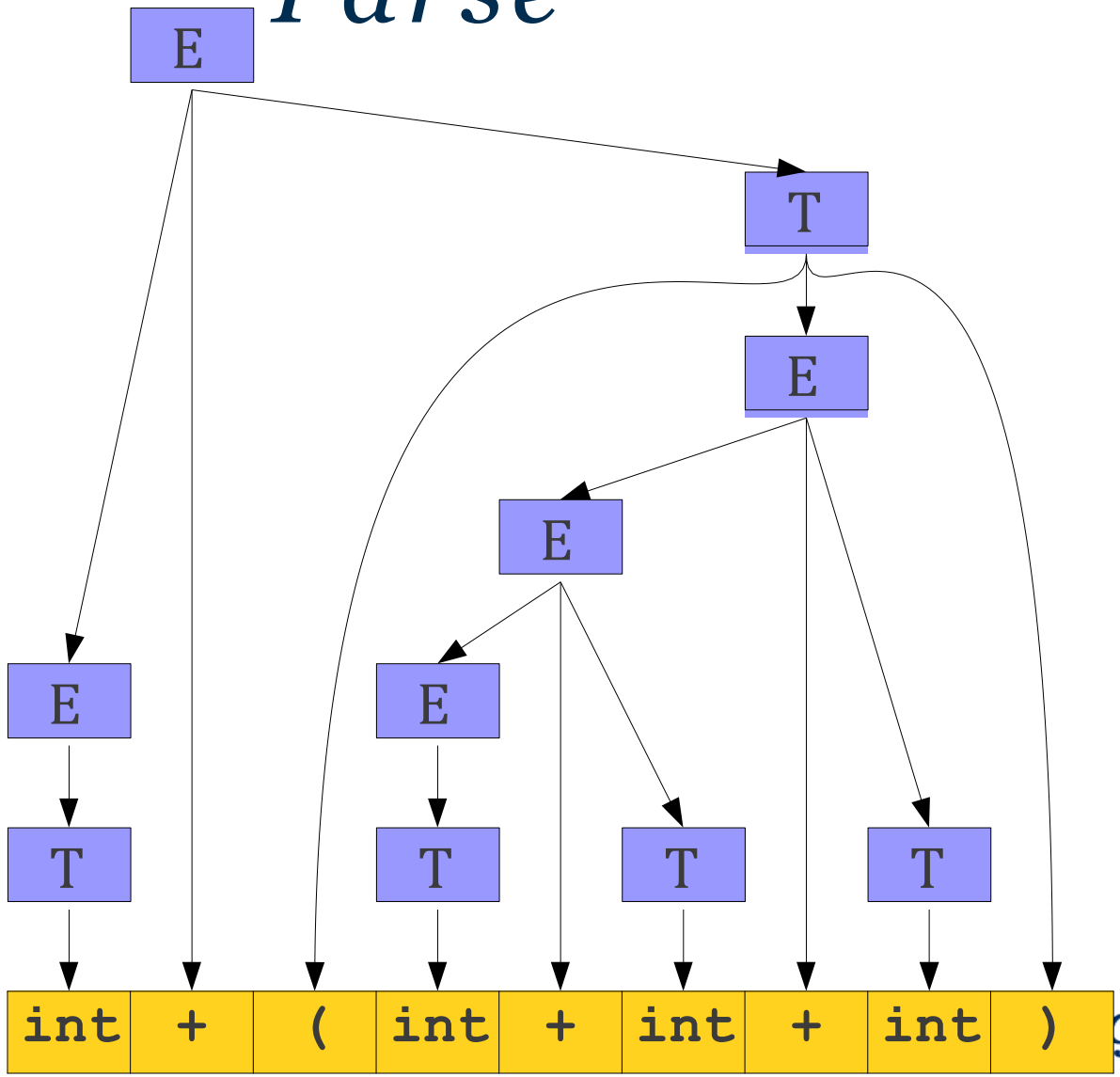
- Idea: Apply productions **in reverse** to convert the user's program to the start symbol.
- As with top-down, could be done with a DFS or BFS, though this is rarely done in practice.
- We'll be exploring 4-ish **directional**, **predictive**
- bottom-up parsing techniques:

Directional: Scan the input from left-to-right.

Predictive: Guess which production should be inverted.

One View of a Bottom-Up Parse

E → **T**
E → **E** + **T**
T → `int`
T → `(` **E** `)`



A Second View of a Bottom-Up Parse

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

$$
\begin{aligned}
& \texttt{int} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & T + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (T + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + T + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int}) \\
\Rightarrow\ & E + (E + T) \\
\Rightarrow\ & E + (E) \\
\Rightarrow\ & E + T \\
\Rightarrow\ & E
\end{aligned}
$$

A Second View of a Bottom-Up Parse

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

GEORGETOWN UNIVERSITY

A left-to-right, bottom-up parse is a rightmost derivation traced in reverse.

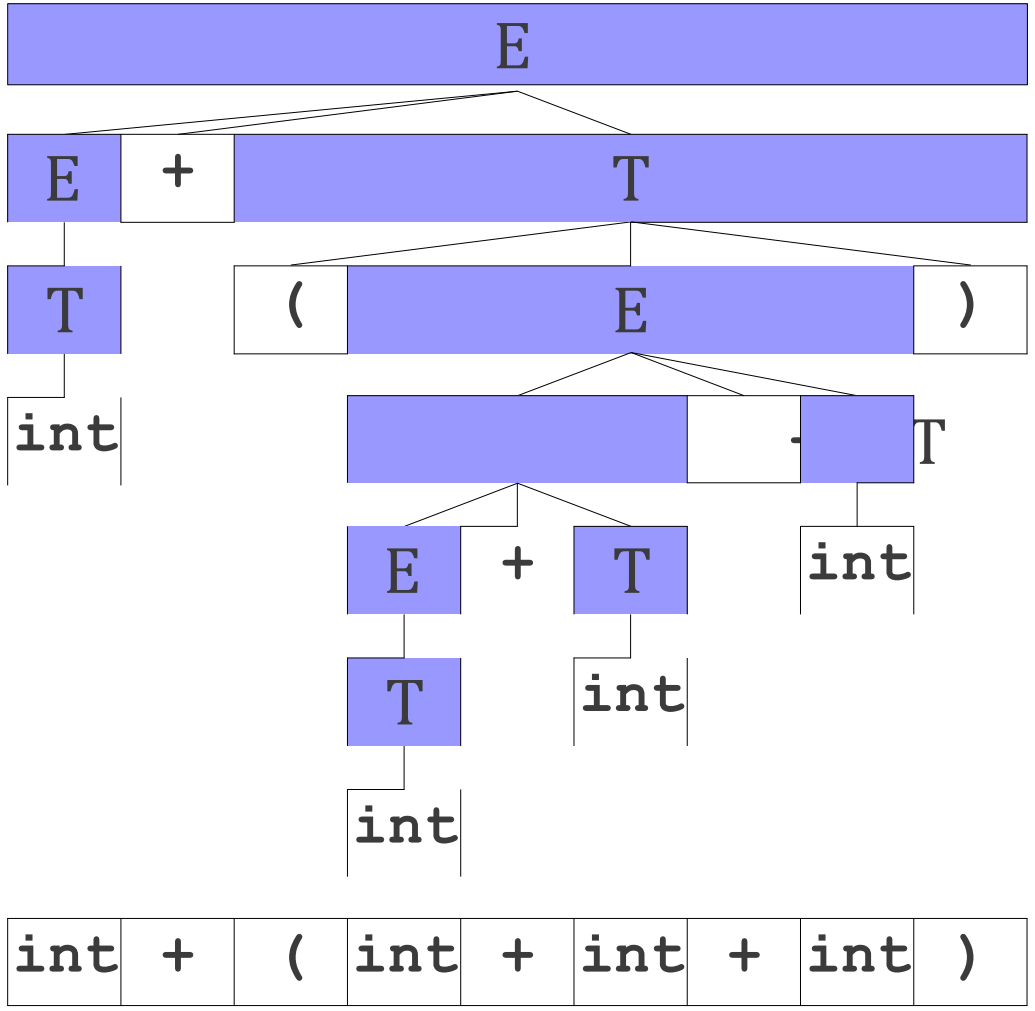
A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E  + (E  + int)
⇒ E  + (E  + T)
⇒ E  + (E)
⇒ E  + T
⇒ E
```

Each step in this bottom-up parse is called a **reduction**. We **reduce** a substring of the sentential form back to a nonterminal.

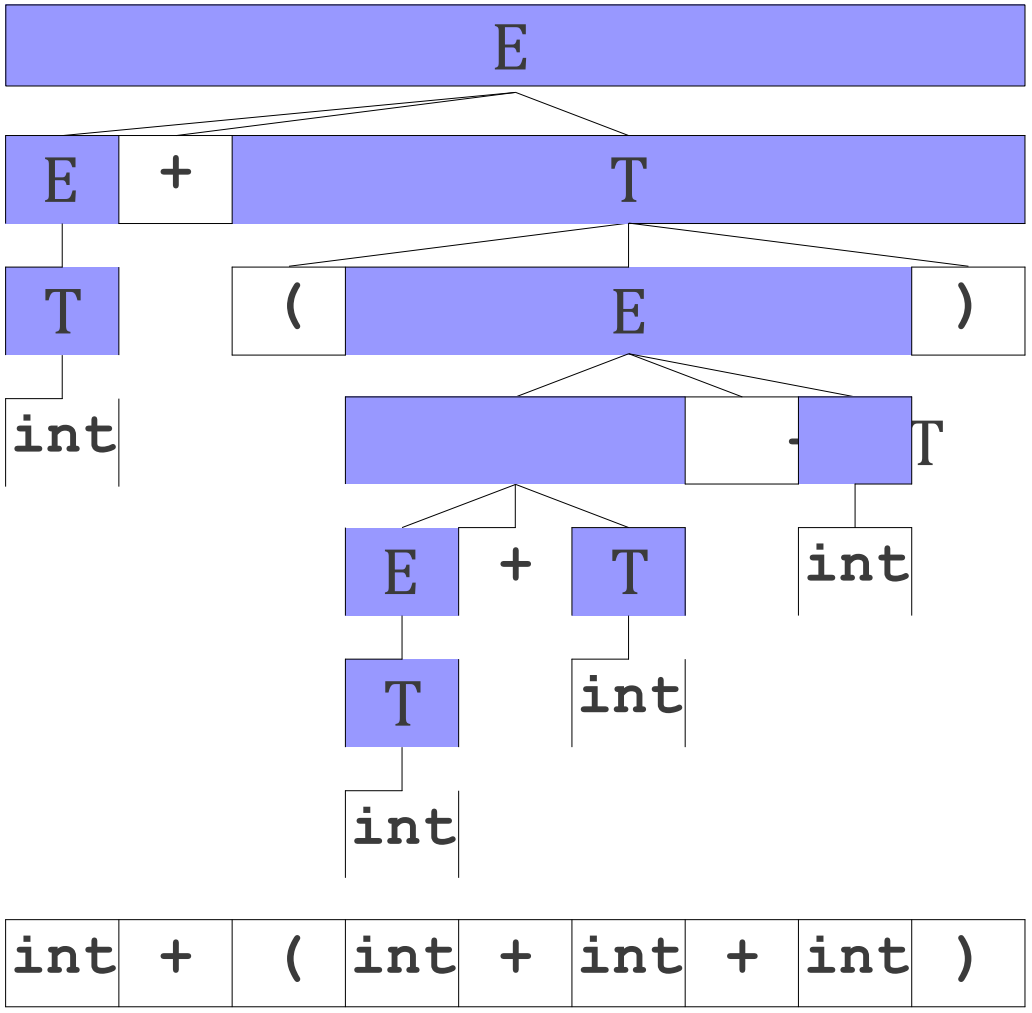
A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



A Third View of a Bottom-Up Parse

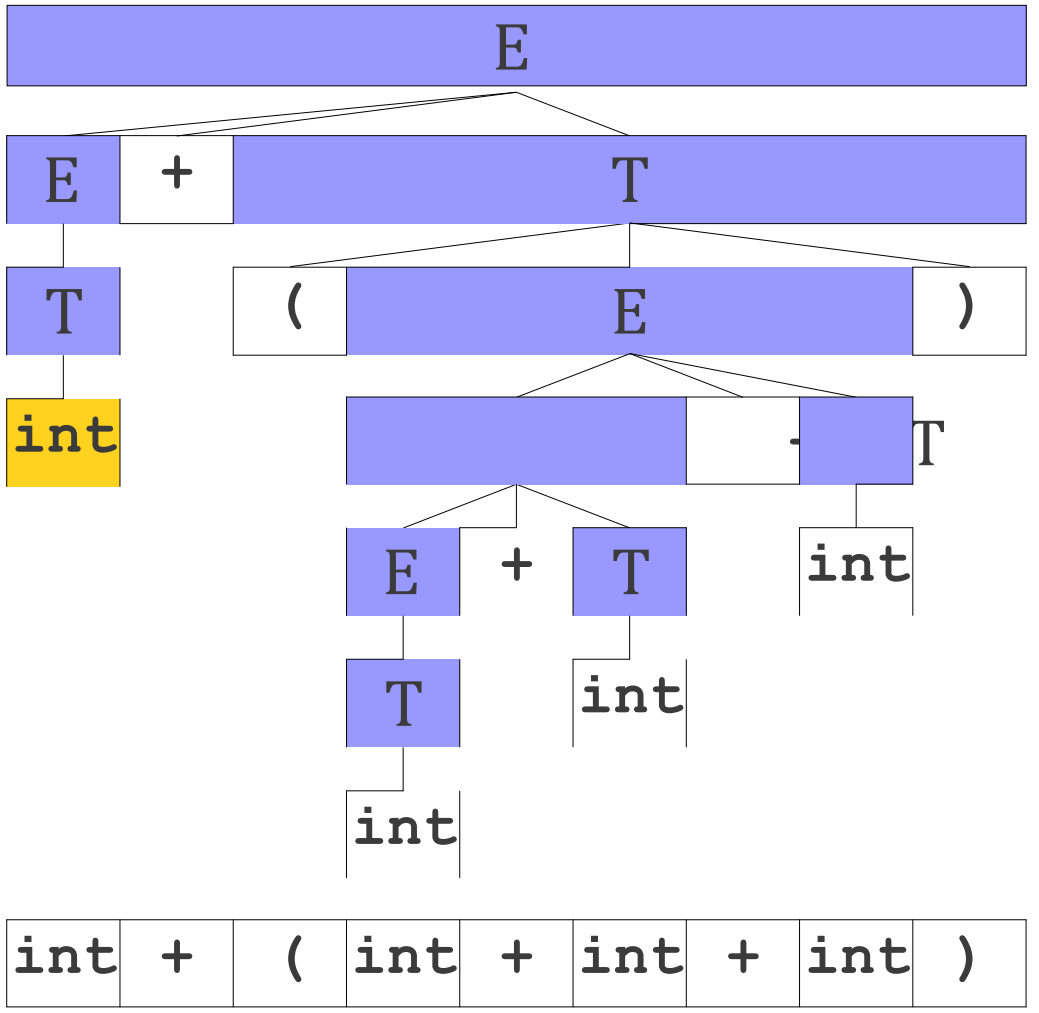
```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

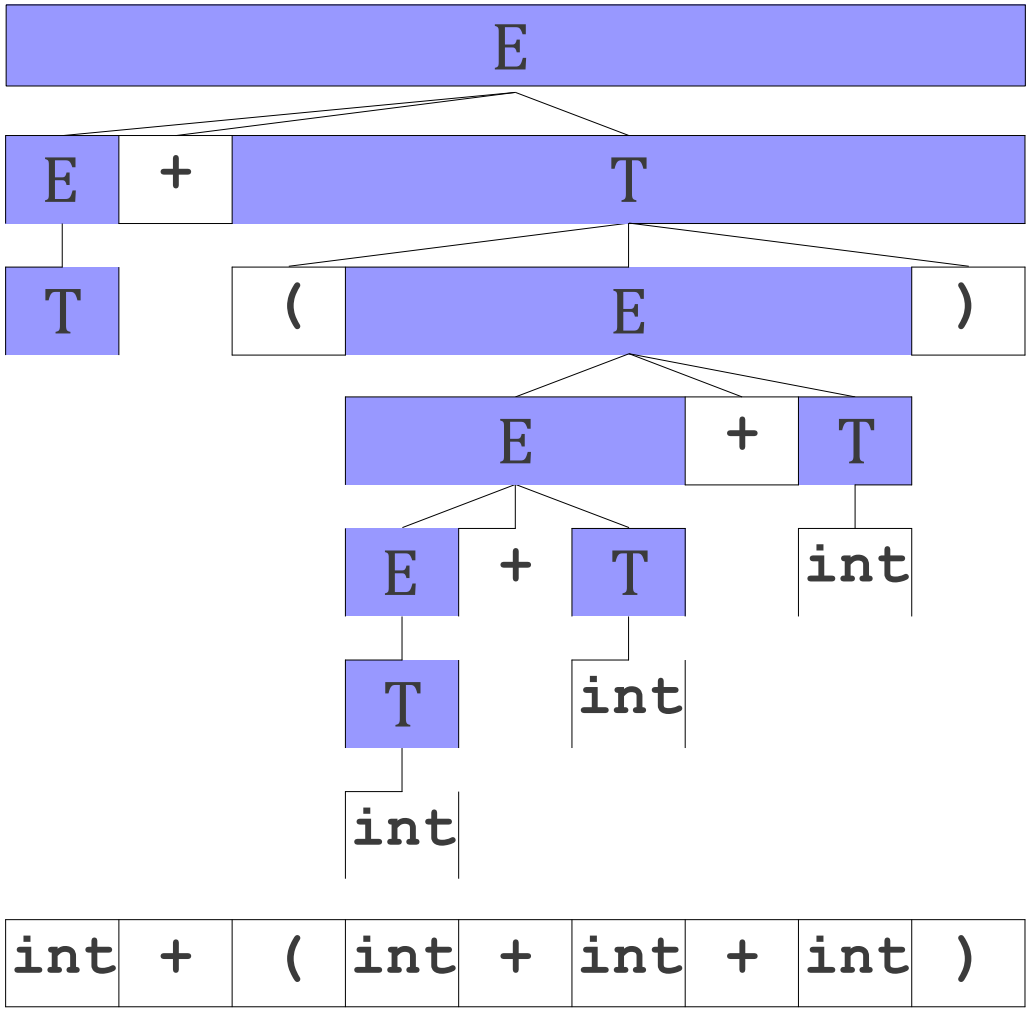
```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



GEORGETOWN UNIVERSITY

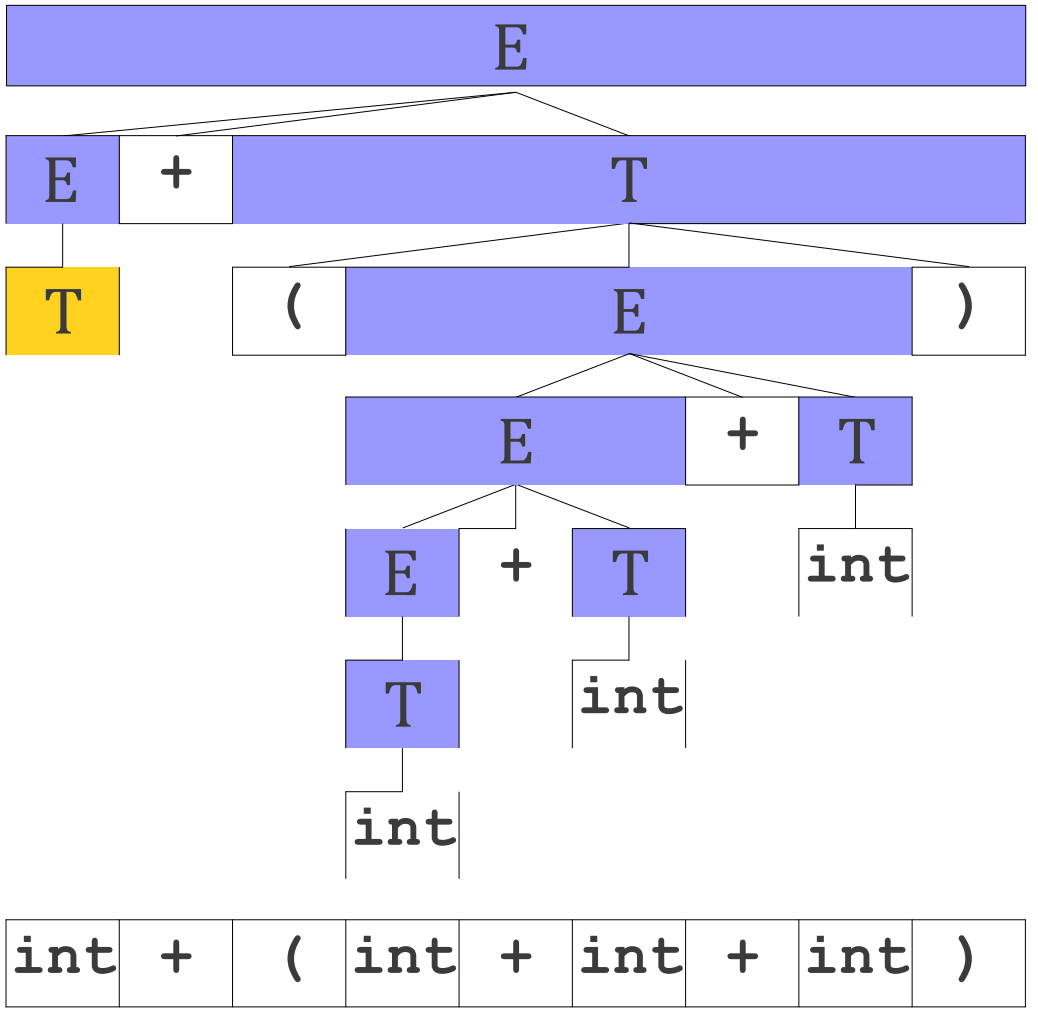
A Third View of a Bottom-Up Parse

```
⇒ T   +  (int    + int + int)
⇒ E   +  (int    + int + int)
⇒ E   +  (T +    int + int)
⇒ E   +  (E +    int + int)
⇒ E   +  (E +    T + int)
⇒ E   +  (E +    int)
⇒ E   +  (E +    T)
⇒ E   +  (E)
⇒ E   +  T
⇒ E
```



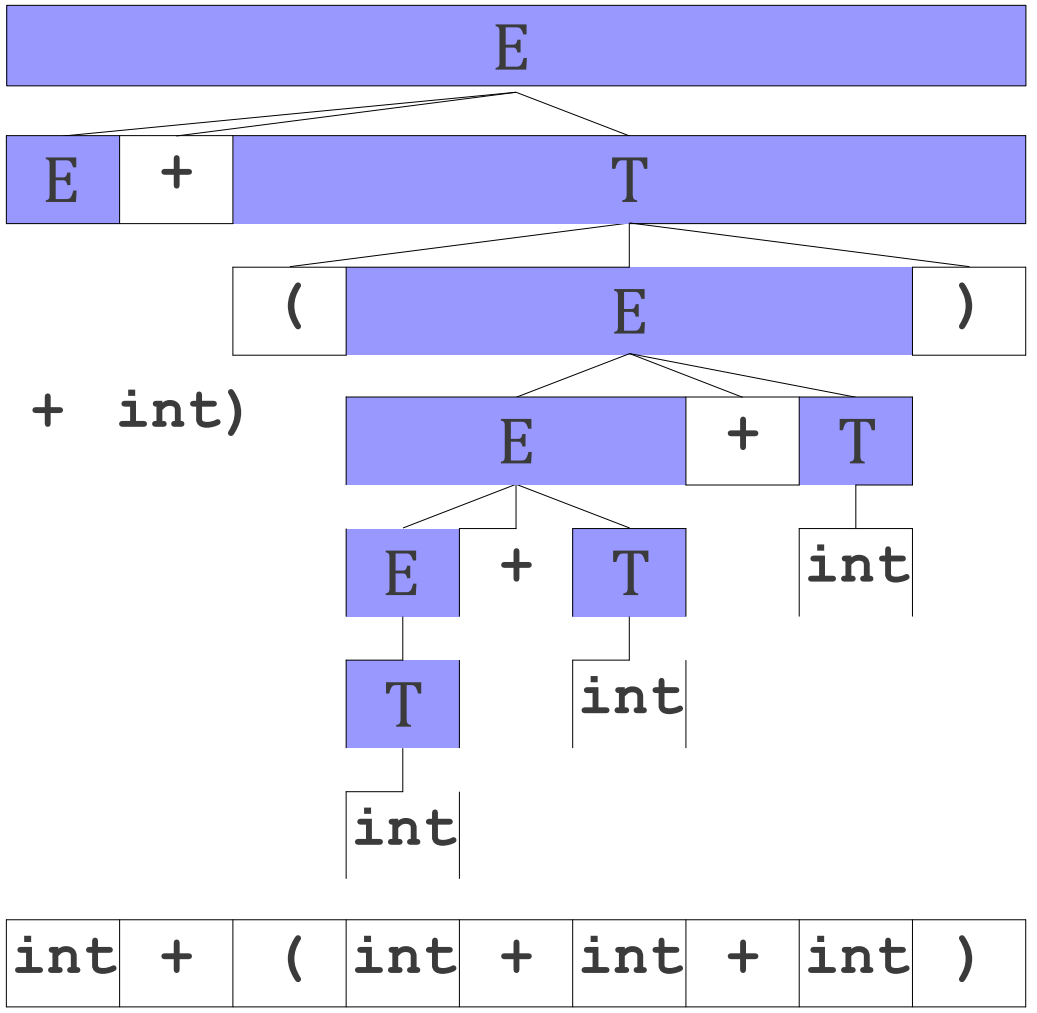
A Third View of a Bottom-Up Parse

```
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



A Third View of a Bottom-Up Parse

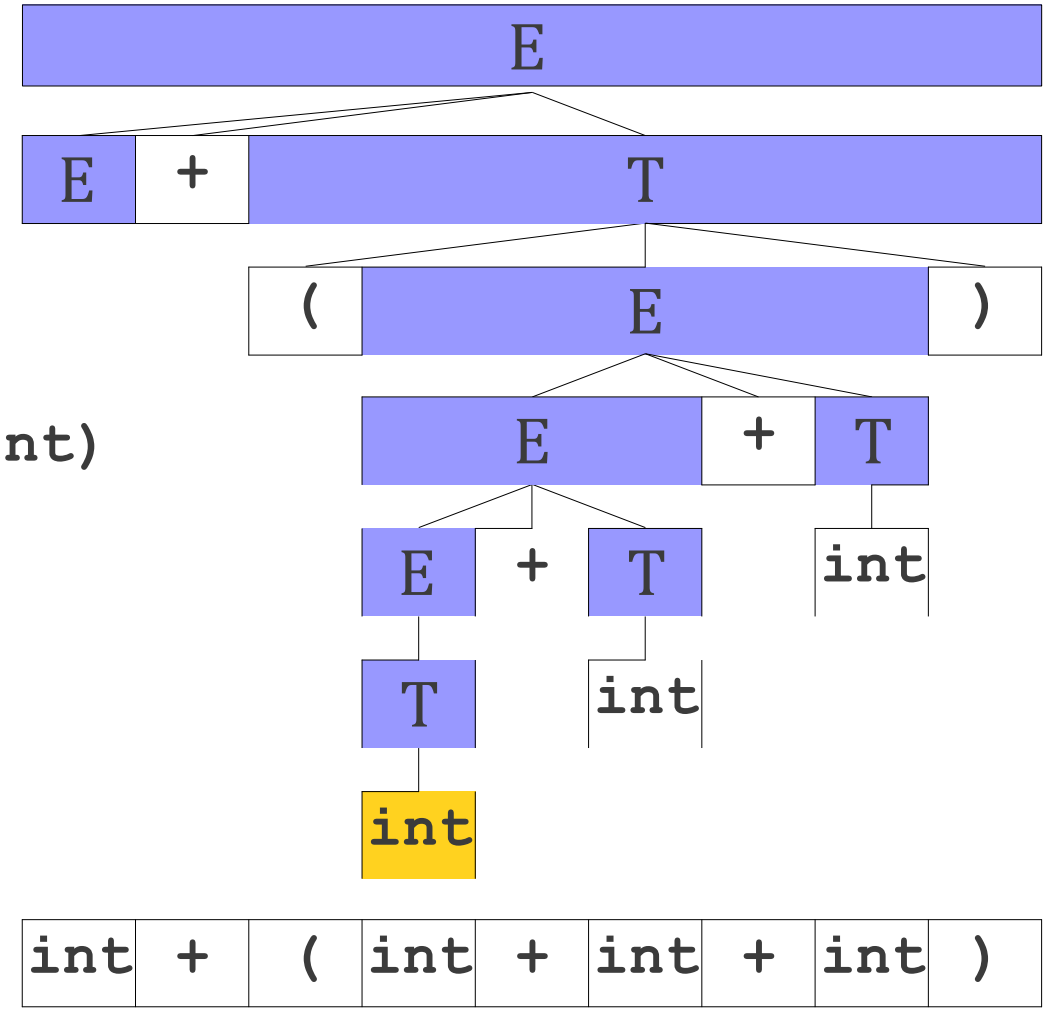
```
⇒ E      +    (int        + int + int)
⇒ E      +    (T  +       int  +  int)    int  +  int)
⇒ E      +    (E  +       T + int)
⇒ E      +    (E  +
⇒ E      +    (E  +       int)
⇒ E      +    (E  +       T)
⇒ E      +    (E)
⇒ E      +    T
⇒ E
```



GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

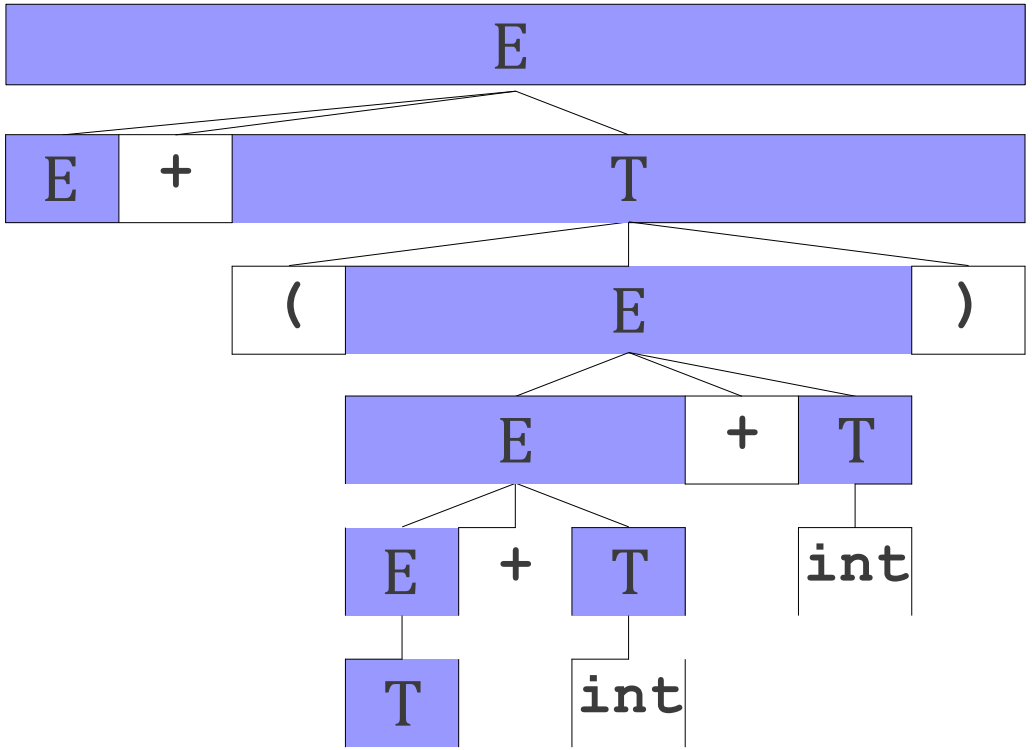
```
⇒ E      +    (int        + int + int)
⇒ E      +    (T  +       int + int)   int + int)
⇒ E      +    (E  +       T + int)
⇒ E      +    (E  +
⇒ E      +    (E  +       int)
⇒ E      +    (E  +       T)
⇒ E      +    (E)
⇒ E      +    T
⇒ E
```



GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

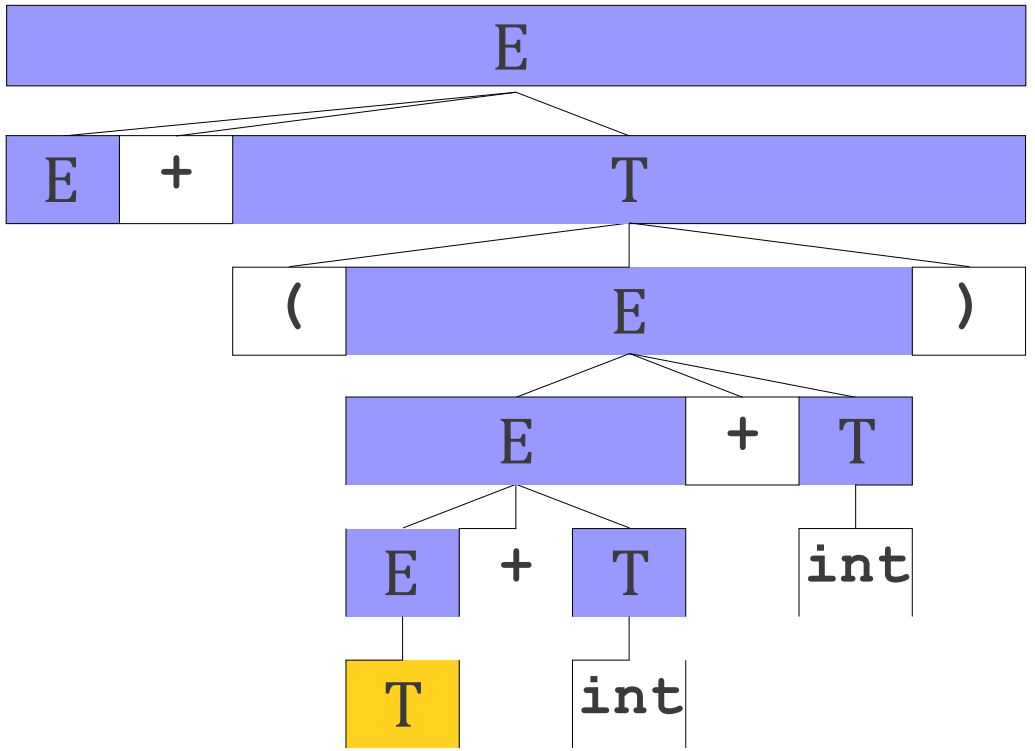
⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



int	+	(	int	+	int	+	int	)

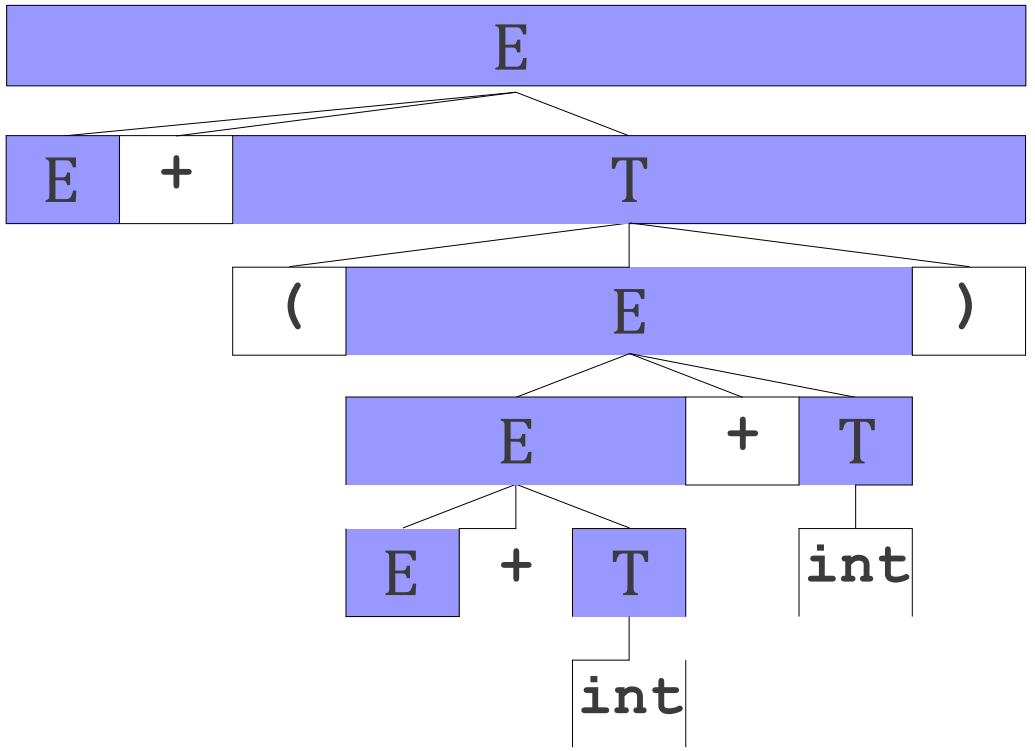
A Third View of a Bottom-Up Parse

⇒ **E** + (**T** + int + int)
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse



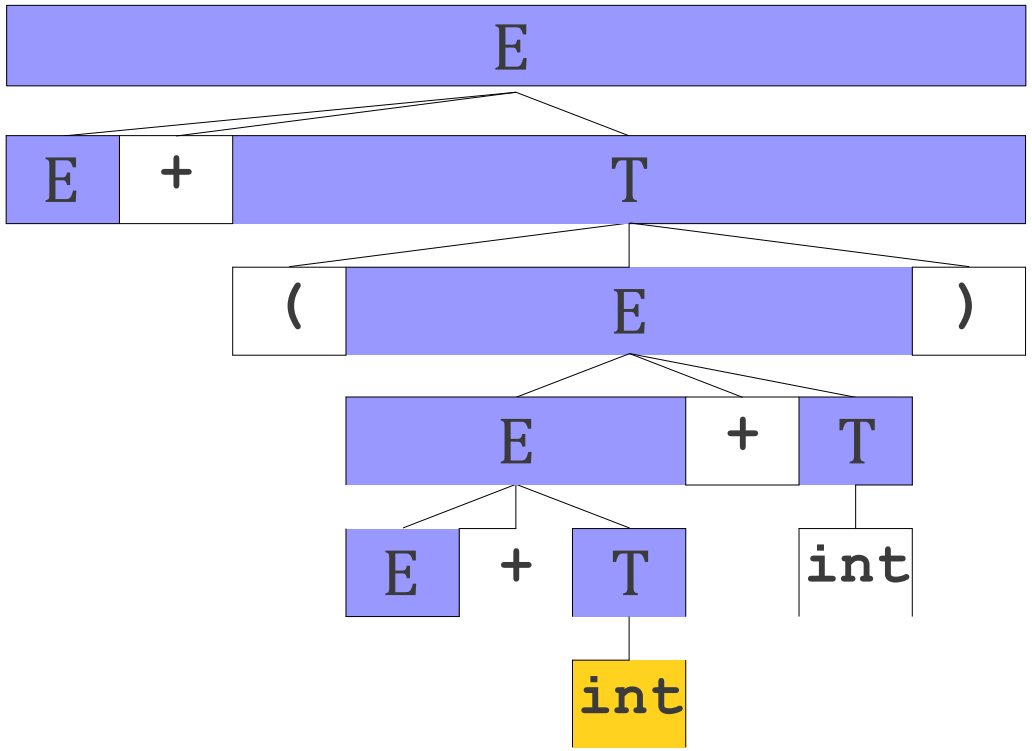
⇒ **E** + (**E** + int + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

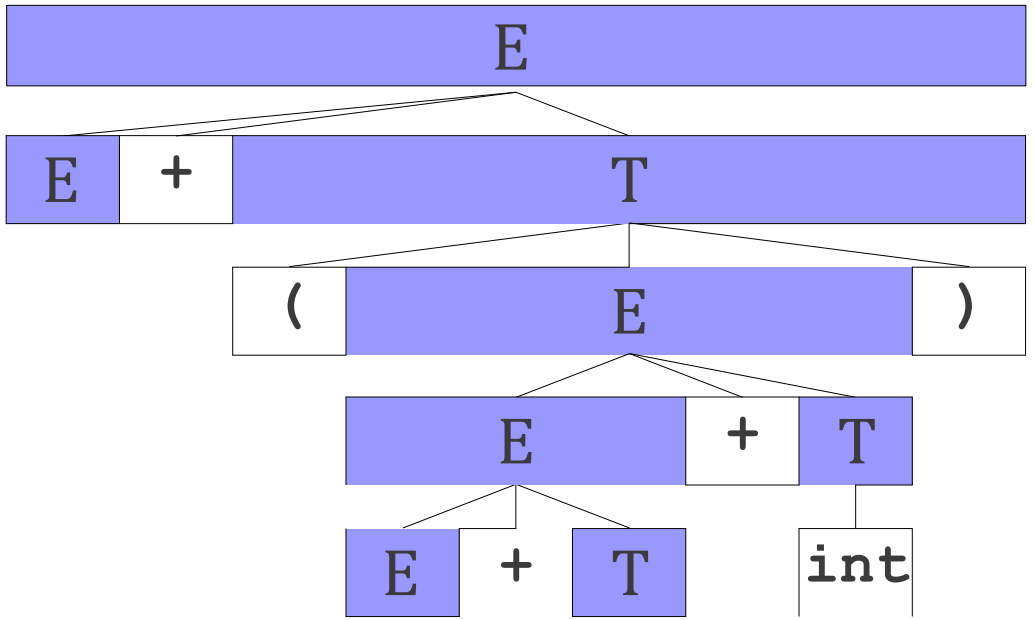
A Third View of a Bottom-Up Parse

⇒ **E** + (**E** + **int** + int)
⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**



int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

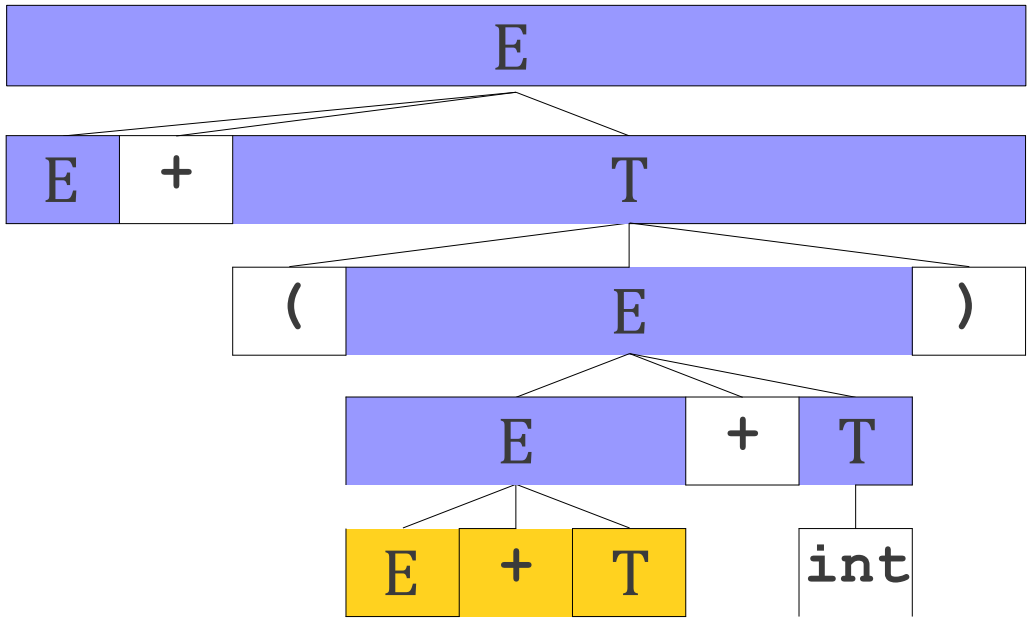


⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

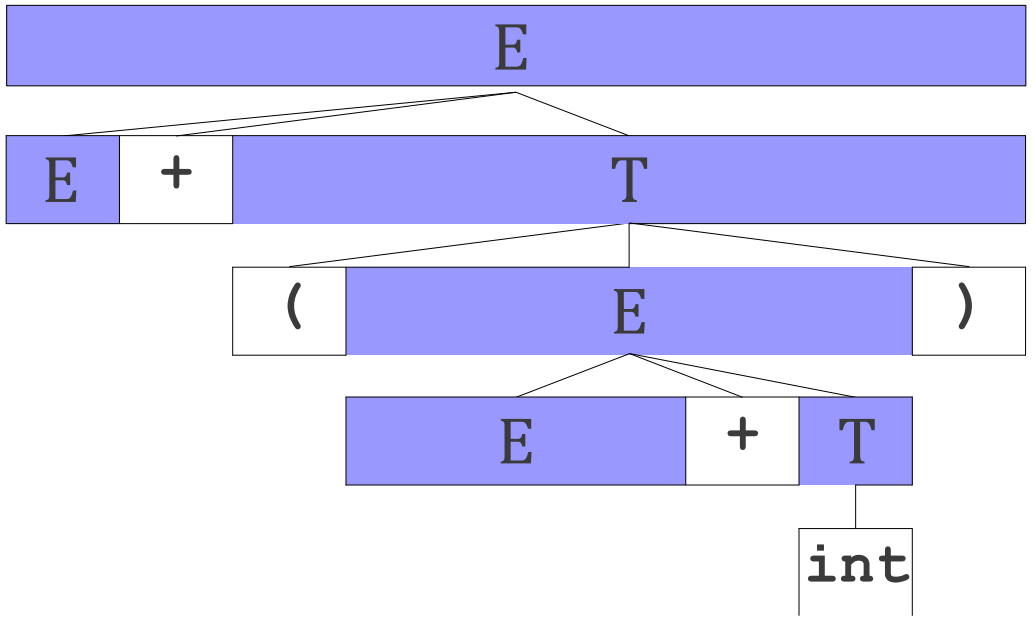


⇒ **E** + (**E** + **T** + int)
⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

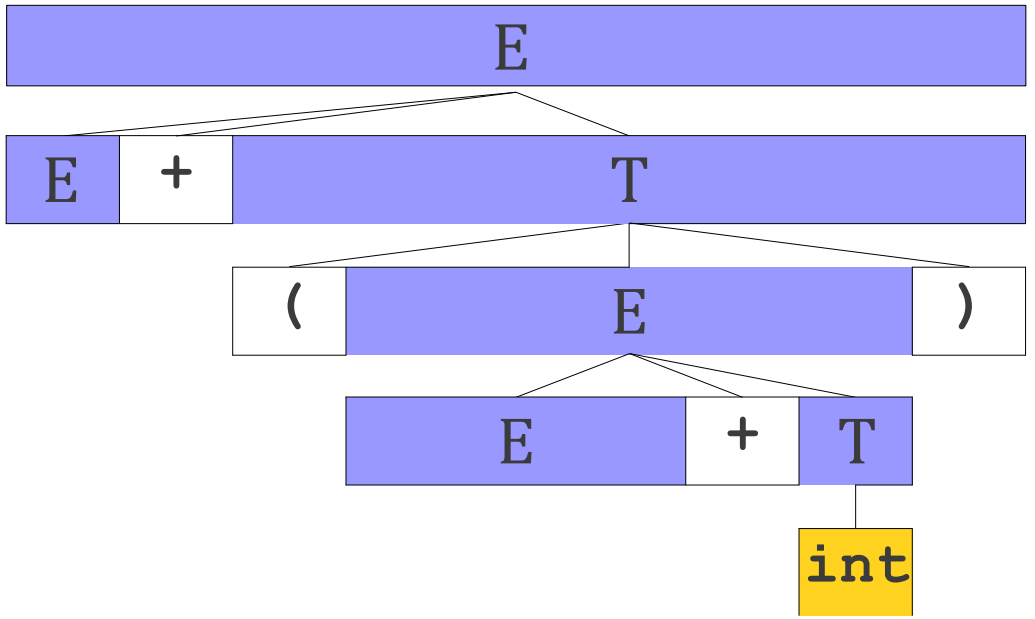
A Third View of a Bottom-Up Parse



⇒ **E** + (**E** + int)
⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

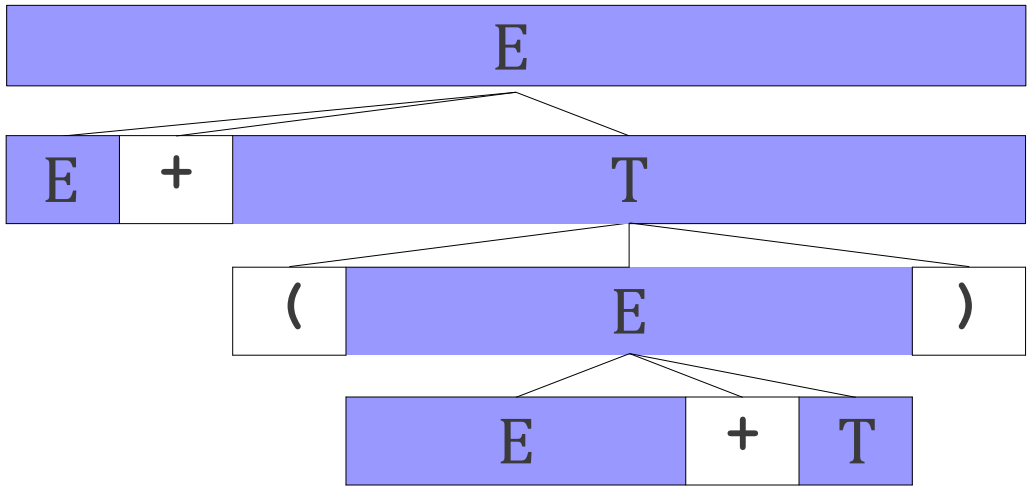


⇒ **E + (E + int)**
⇒ **E + (E + T)**
⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

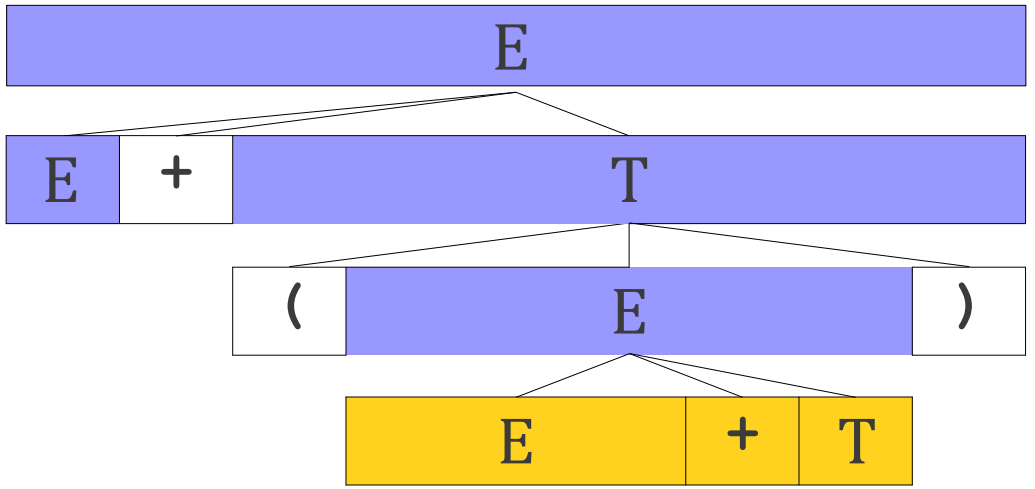


⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

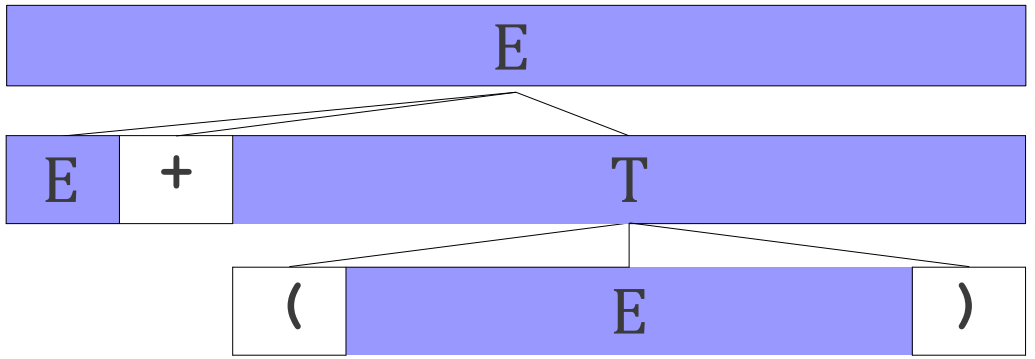


⇒ **E** + (**E** + **T**)
⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

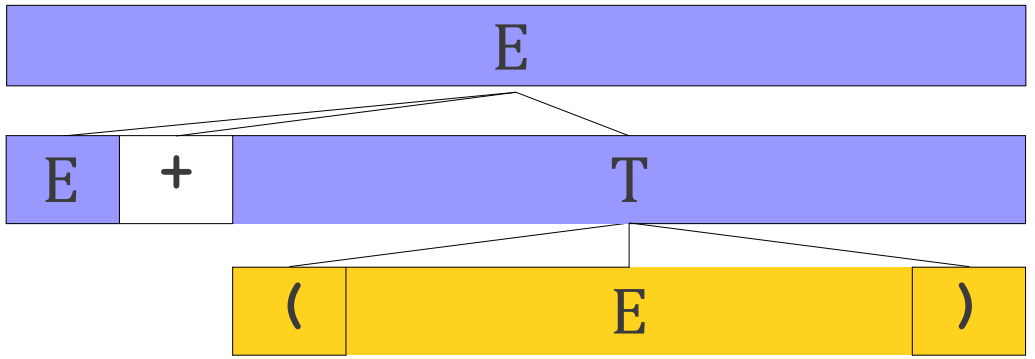
A Third View of a Bottom-Up Parse



⇒ **E** + (**E**)
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

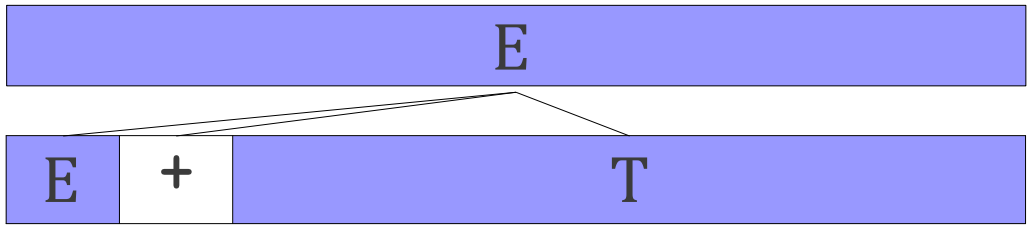


⇒ **E** + **(E)**
⇒ **E** + **T**
⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse



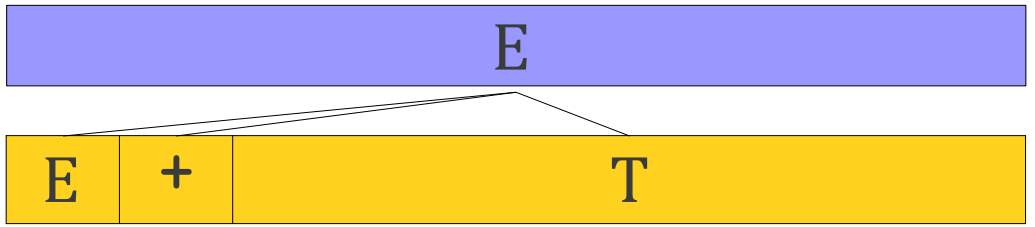
⇒ **E + T**

⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse



⇒ **E + T**

⇒ **E**

int	+	(	int	+	int	+	int	)

GEORGETOWN UNIVERSITY

A Third View of a Bottom-Up Parse

⇒ **E**

int	+	(	int	+	int	+	int	)

Handles

- The **handle** of a parse tree *T* is the leftmost complete cluster of leaf nodes.
  - Refers to LHS non-terminal chosen during resolve step
- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.

Summarizing Our Intuition

- Our first intuition (reconstructing the parse tree bottom-up) motivates how the parsing should work.
- Our second intuition (rightmost derivation in reverse) describes the order in which we should build the parse tree.
- Our third intuition (handle pruning) is the basis for the bottom-up parsing algorithms we will explore.

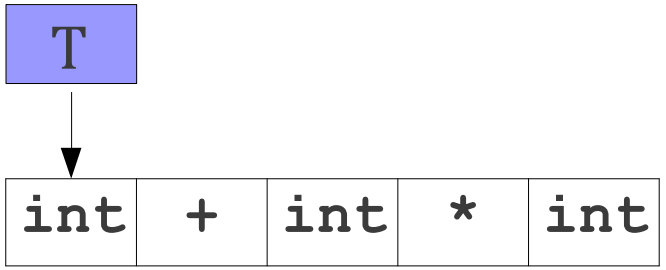
A Detail about Handles

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

int	+	int	*	int

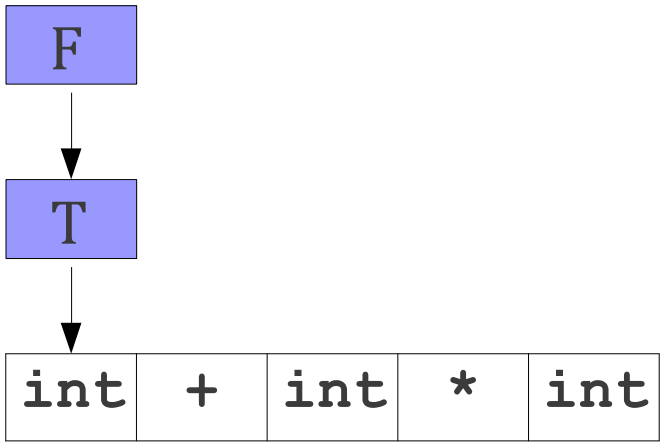
A Detail about Handles

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



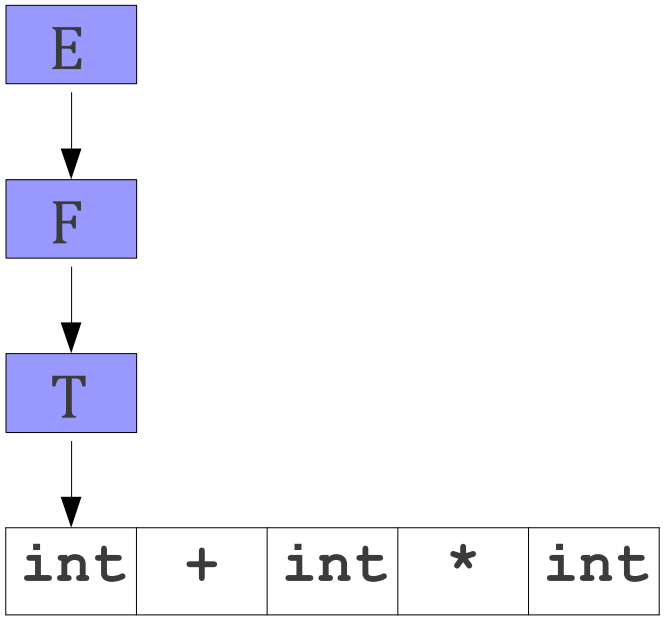
A Detail about Handles

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



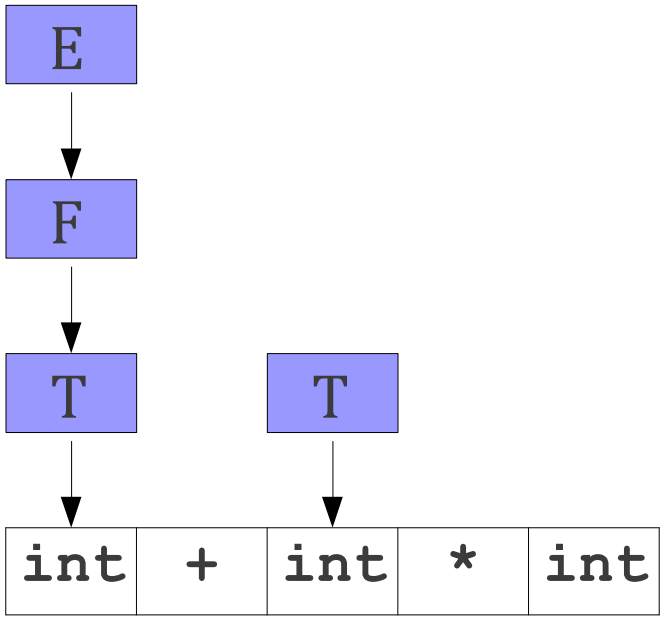
A Detail about Handles

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



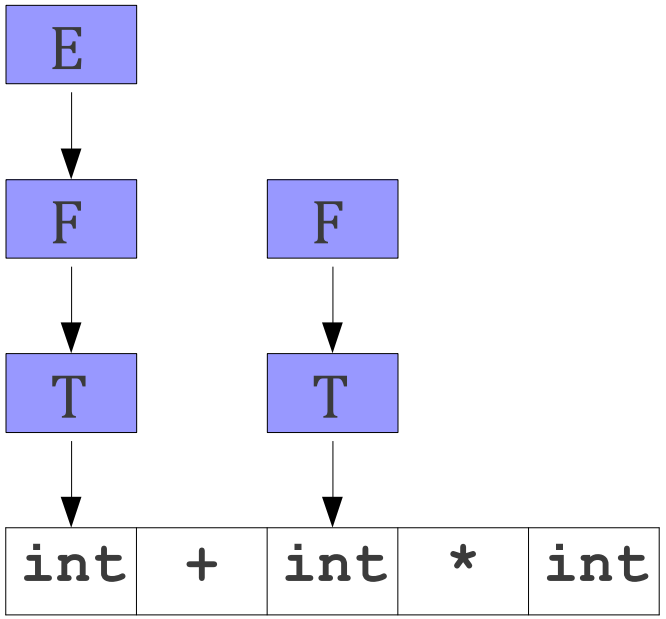
A Detail about Handles

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



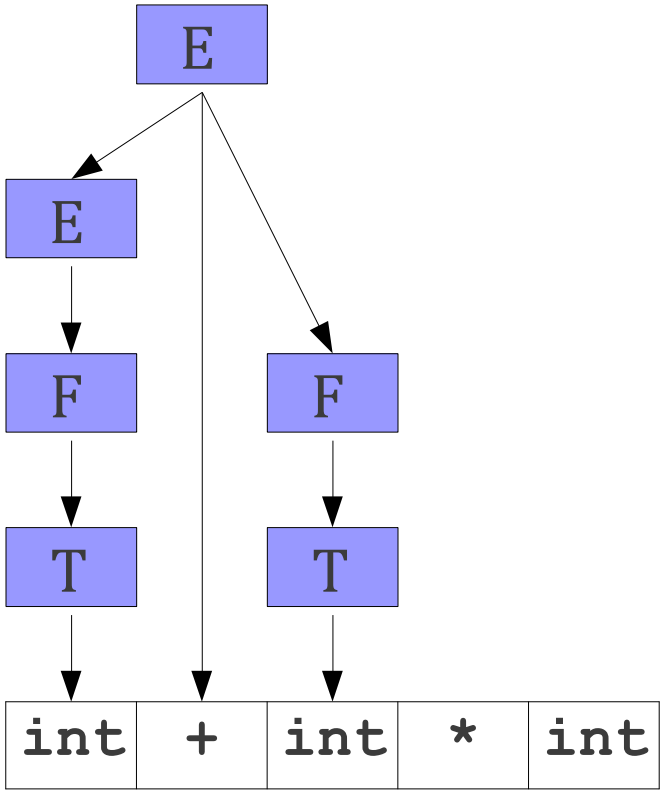
A Detail about Handles

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



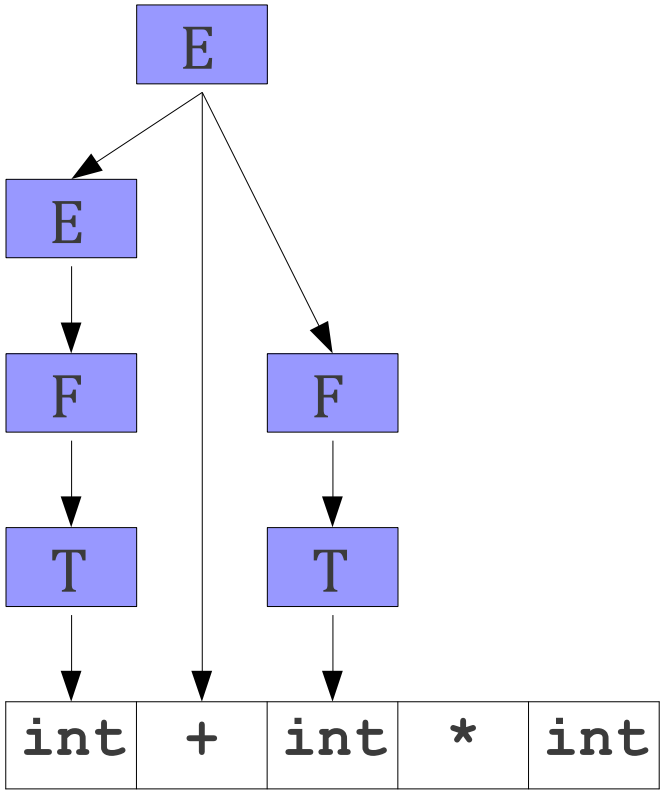
A Detail about Handles

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



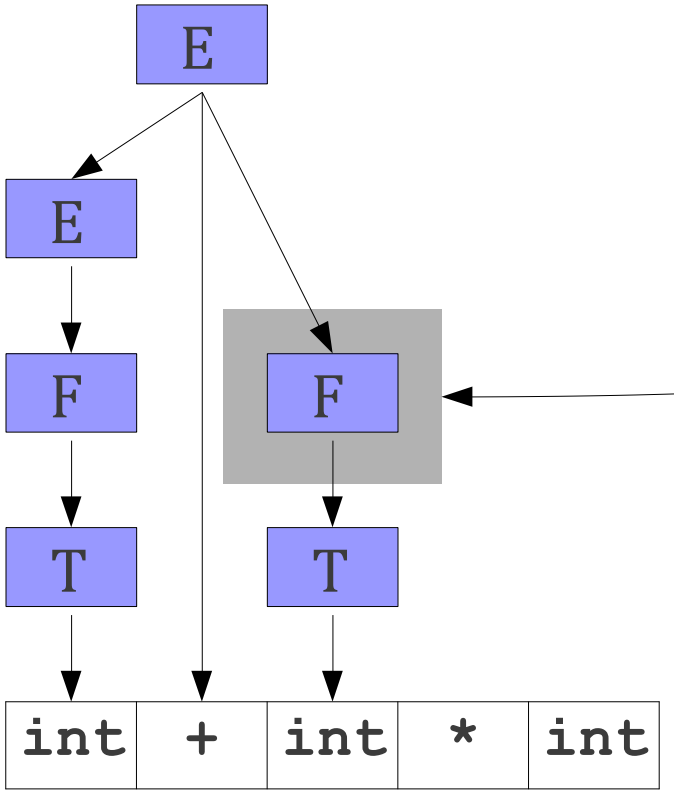
A Detail about Handles

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



A Detail about Handles

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



This reduction wasn't a handle!

The leftmost reduction isn't always the handle.

Finding Handles

- Where do we look for handles?
  - Where in the string might the handle be?
- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  - Once we've found a candidate handle, how do we check that it really is the handle?

Question One:

Where are handles?

GEORGETOWN UNIVERSITY

Where are Handles?

- Recall: A left-to-right, bottom-up parse traces a rightmost derivation in reverse.
- Each time we do a reduction, we are reversing a production applied to the *rightmost* nonterminal symbol.
- Suppose that our current sentential form is $\alpha\gamma\omega$, where γ is the handle and $\mathbf{A} \rightarrow \gamma$ is a production rule.
- After reducing γ back to **A**, we have the string $\alpha\mathbf{A}\omega$.
- Thus ω must consist purely of terminals, since otherwise the reduction we just did was not for the rightmost terminal.

Why This Matters

- Suppose we want to parse the string γ.
- We will break γ into two parts, α and ω, where
  - α consists of both terminals and nonterminals, and
  - ω consists purely of terminals.
- Our search for handles will concentrate purely in α.
- As necessary, we will start moving terminals from ω over into α.

Shift/Reduce Parsing

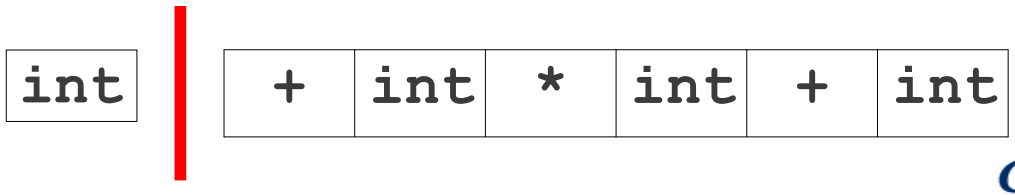
- The bottom-up parsers we will consider are called **shift/reduce** parsers.
  - Contrast with the LL(1) **predict/match** parser.
- Idea: Split the input into two parts:
  - Left substring is our work area; all handles must be here.
  - Right substring is input we have not yet processed; consists purely of terminals.
- At each point, decide whether to:
  - Move a terminal across the split (**shift**)
  - Reduce a handle (**reduce**)

A Sample Shift/Reduce Parse

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

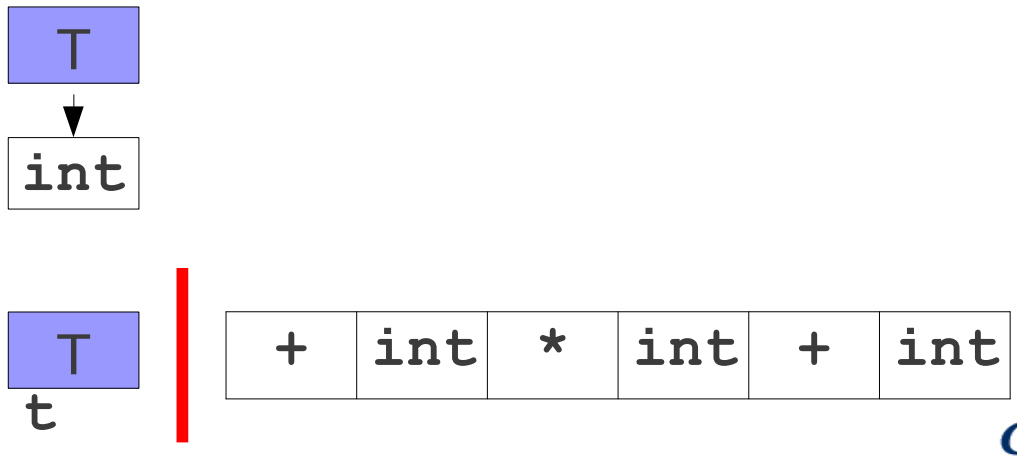
A Sample Shift/Reduce Parse

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



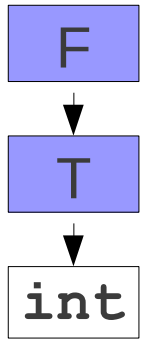
A Sample Shift/Reduce Parse

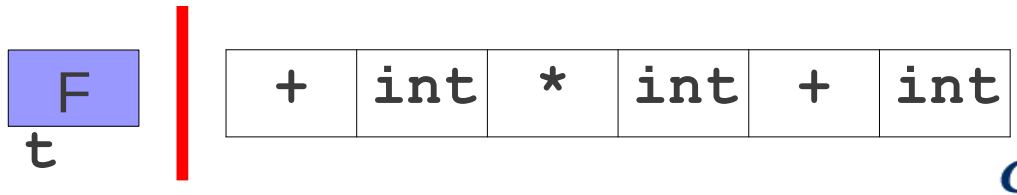
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

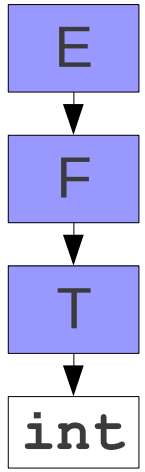
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

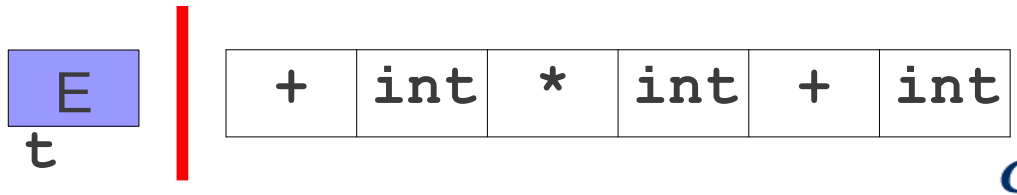




A Sample Shift/Reduce Parse

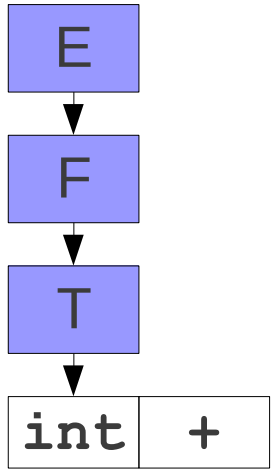
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

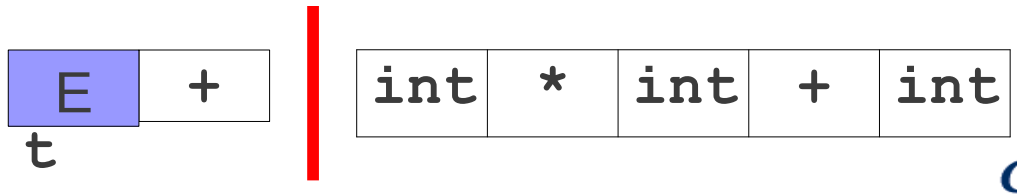




A Sample Shift/Reduce Parse

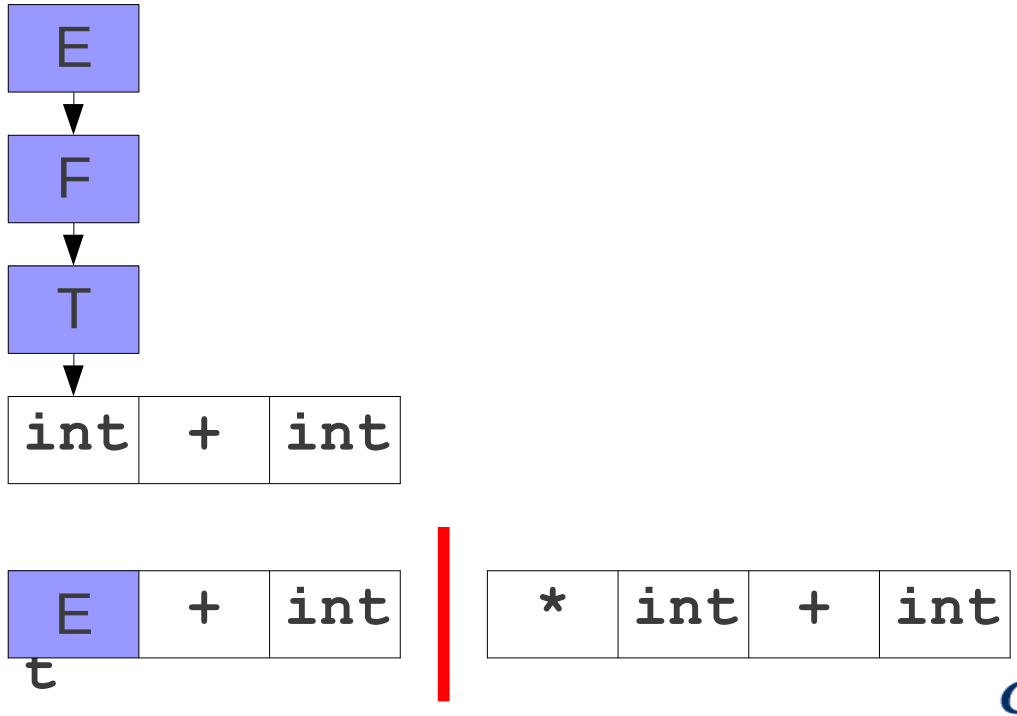
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





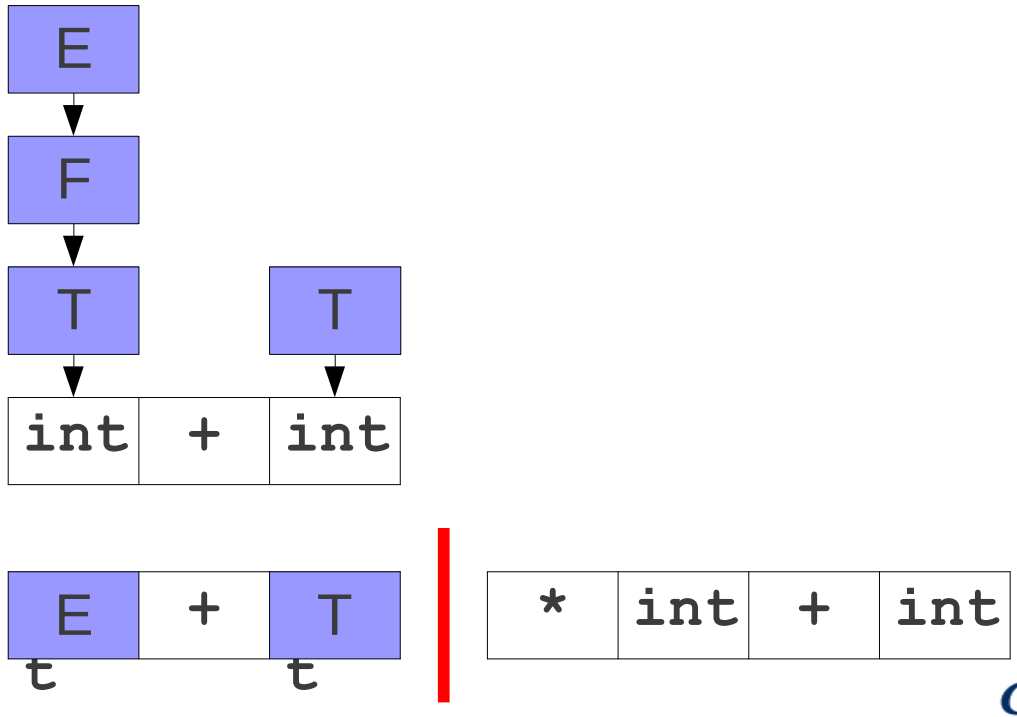
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

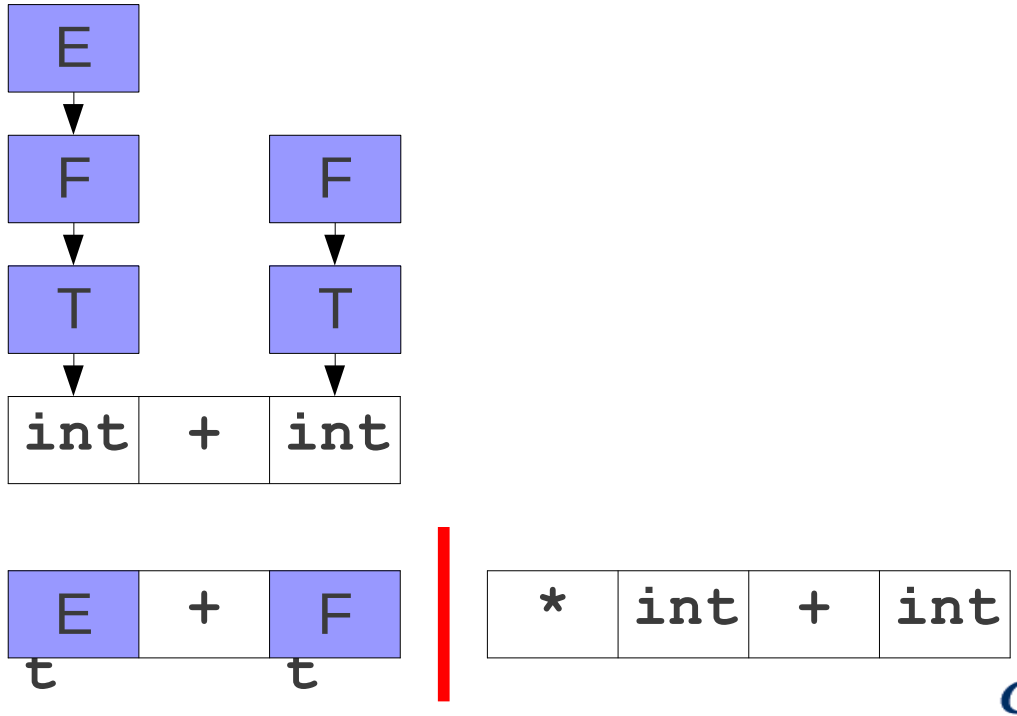
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

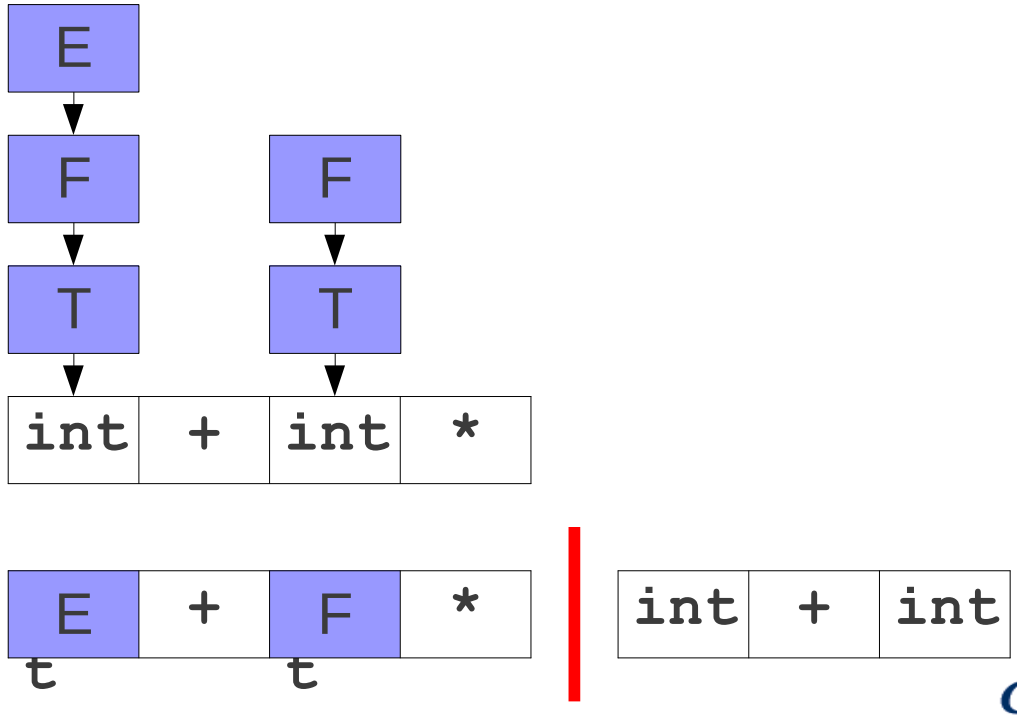
A Sample Shift/Reduce Parse

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



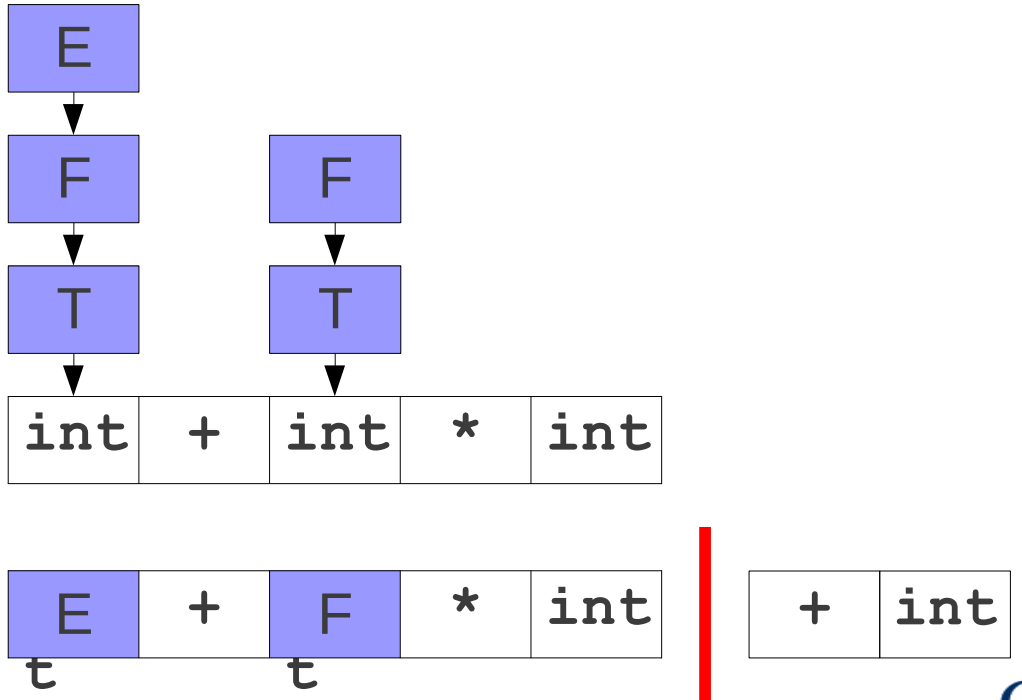
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

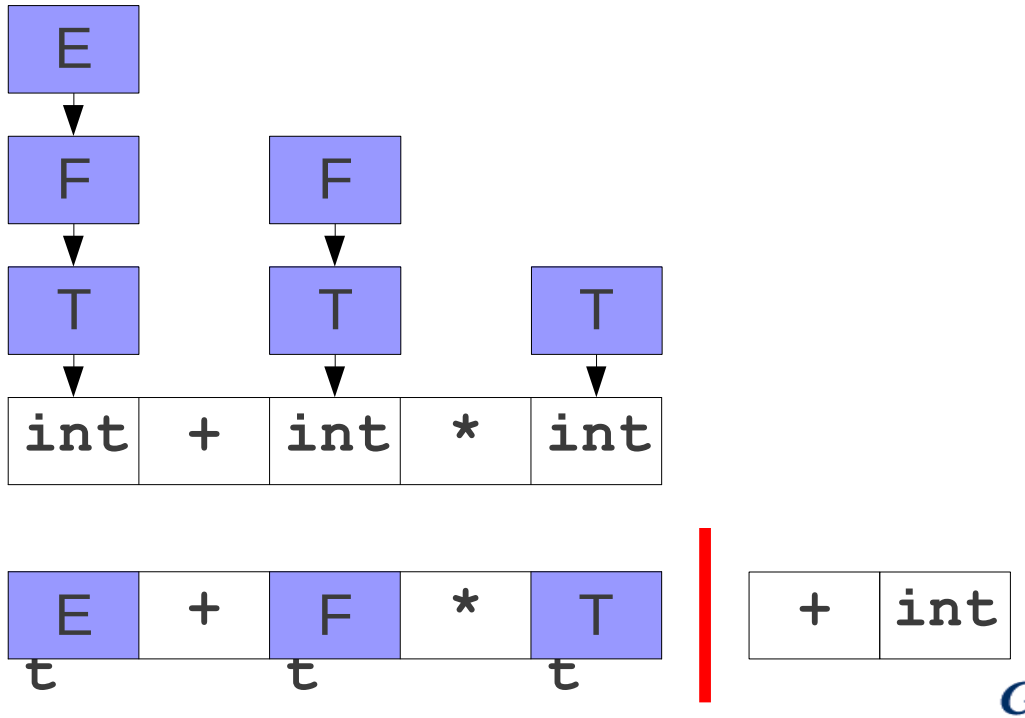
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

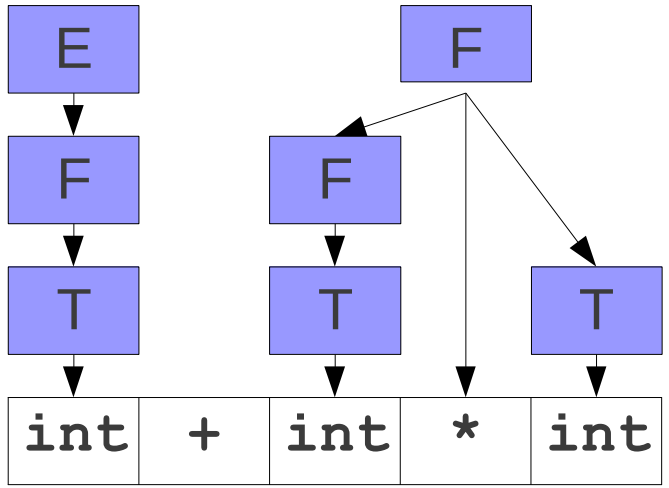
A Sample Shift/Reduce Parse

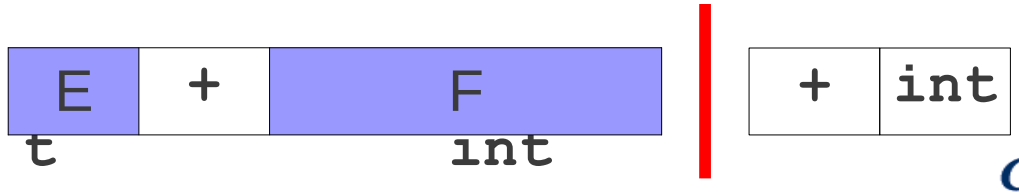
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



A Sample Shift/Reduce Parse

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

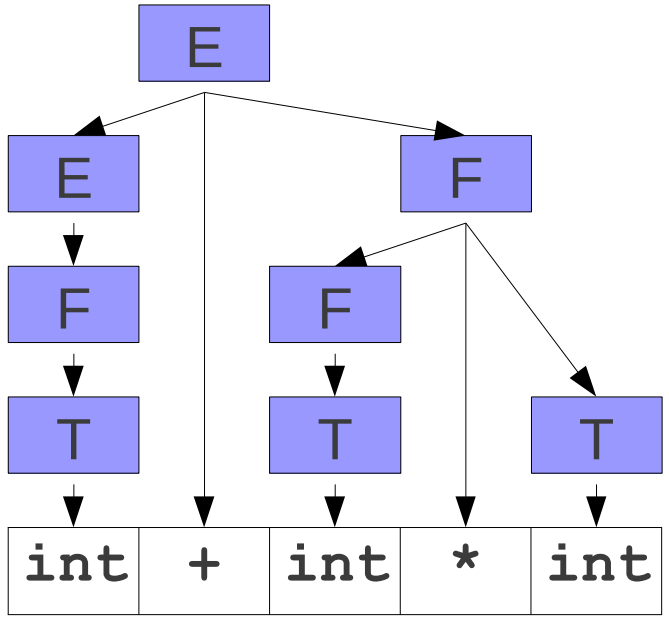


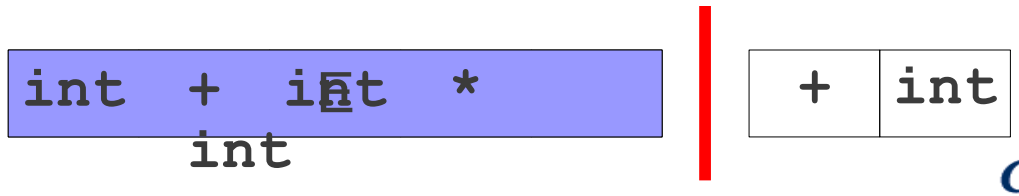


GEORGETOWN UNIVERSITY

A Sample Shift/Reduce Parse

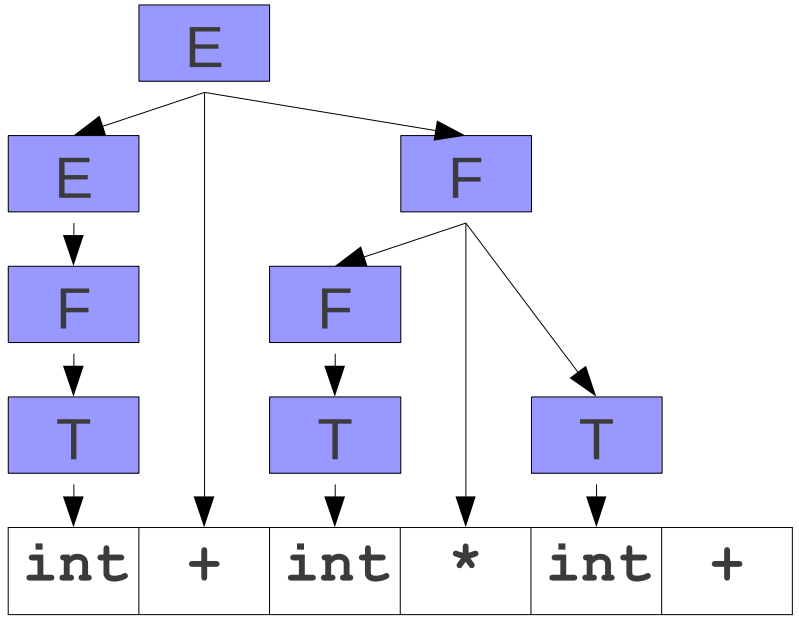
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

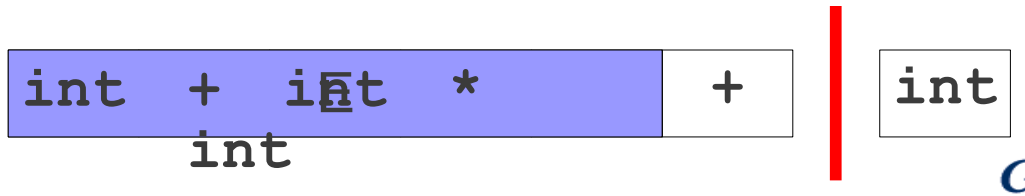




A Sample Shift/Reduce Parse

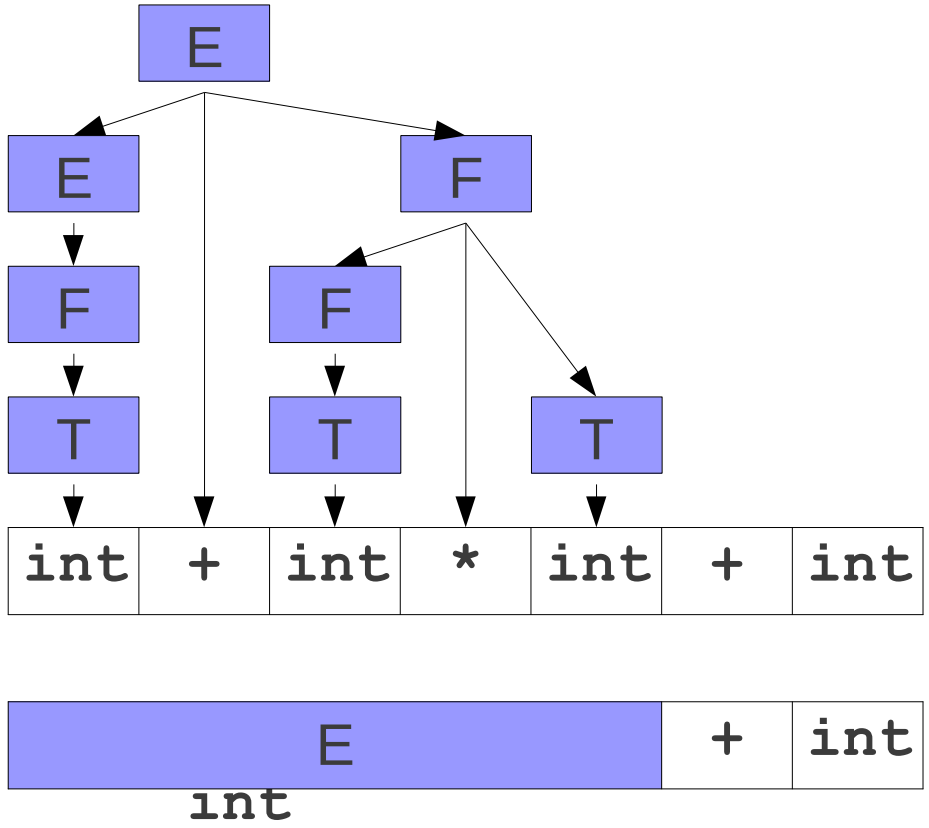
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





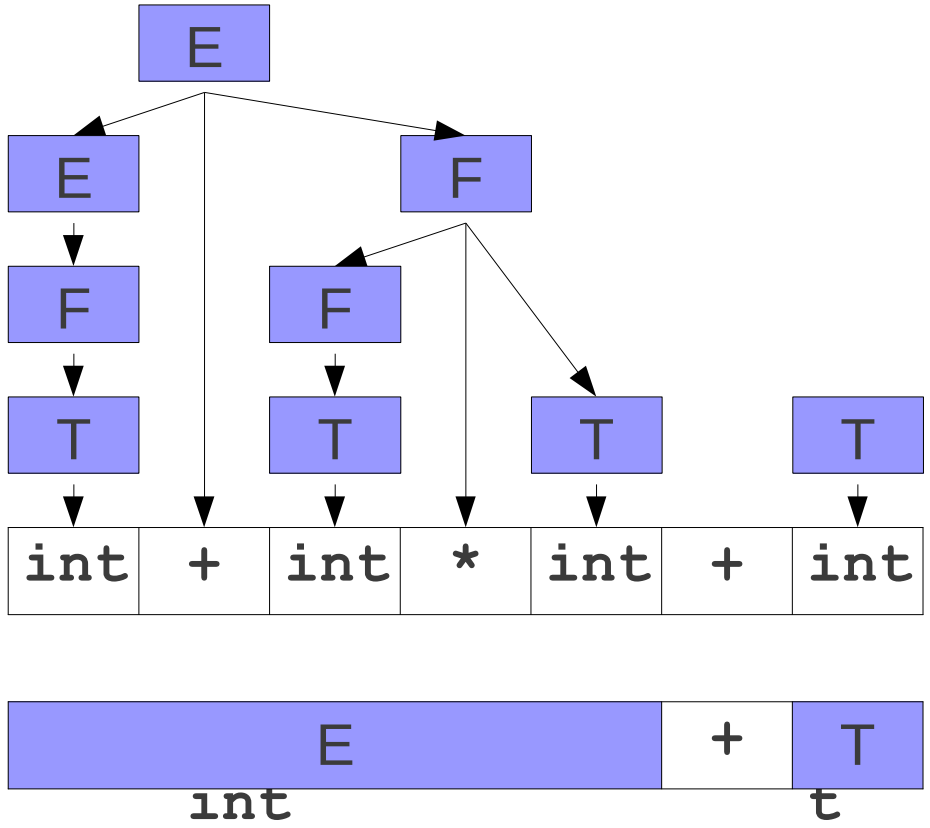
A Sample Shift/Reduce Parse

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



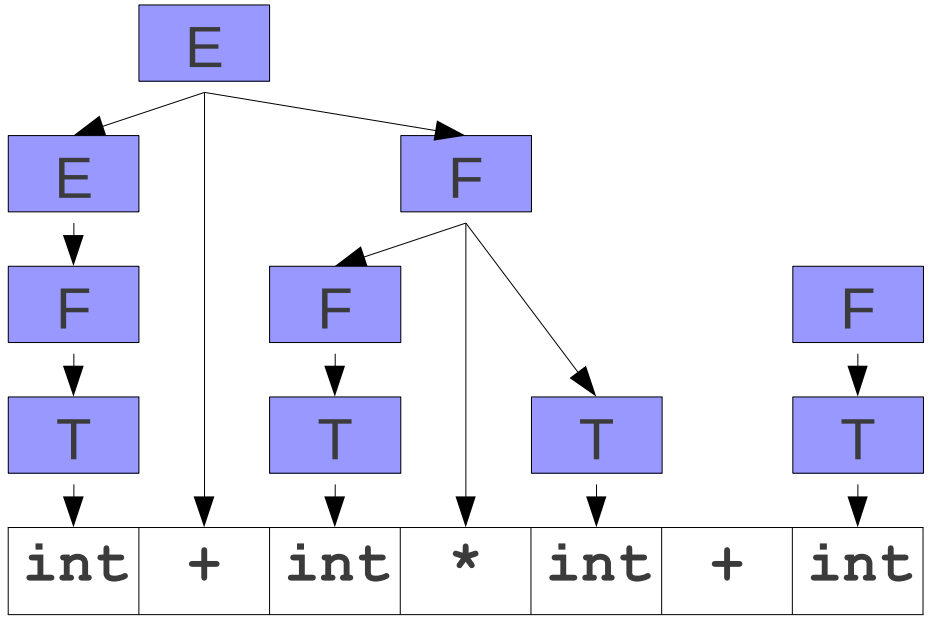
A Sample Shift/Reduce Parse

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



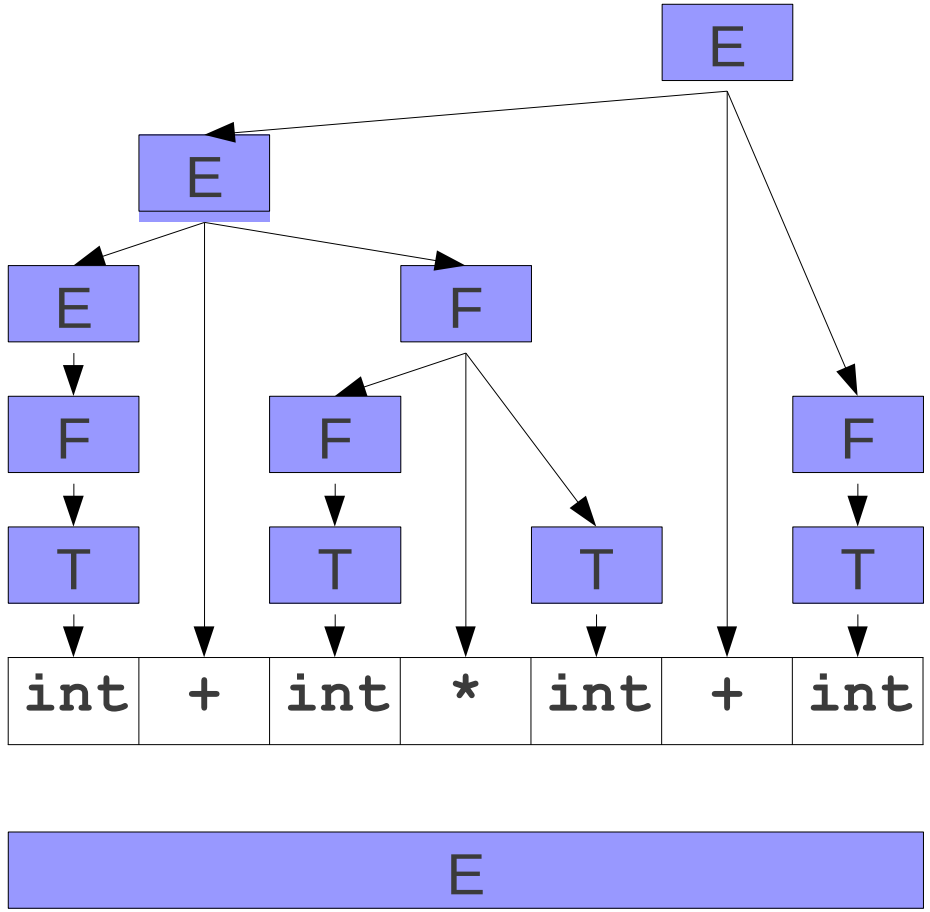
A Sample Shift/Reduce Parse

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



A Sample Shift/Reduce Parse

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



An Important Observation

- All of the reductions we applied were to the far right end of the left area.
- This is not a coincidence; all reductions are always applied all the way to the end of the left area.
- Inductive proof sketch:
  - After no reduces, the first reduction can be done at the right end of the left area.
  - After at least one reduce, the very right of the left area is a nonterminal. This nonterminal must be part of the next reduction, since we're tracing a rightmost derivation backwards.

An Important Corollary

- Since reductions are always at the right side of the left area, we never need to shift from the left to the right.
- No need to "uncover" something to do a reduction.
- Consequently, shift/reduce parsing means

Shift: Move a terminal from the right to the left area.

Reduce: Replace some number of symbols at the right side of the left area.

Simplifying our Terminology

- All activity in a shift/reduce parser is at the far right end of the left area.
- Idea: Represent the left area as a stack.
- Shift: Push the next terminal onto the stack.
- Reduce: Pop some number of symbols from the stack, then push the appropriate nonterminal.

Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**
- How do we search for handles?
  - What algorithm do we use to try to discover a handle?
- How do we recognize handles?
  - Once we've found a possible handle, how do we confirm that it's correct?

Question Two:

How do we search for handles?

GEORGETOWN UNIVERSITY

Searching for Handles

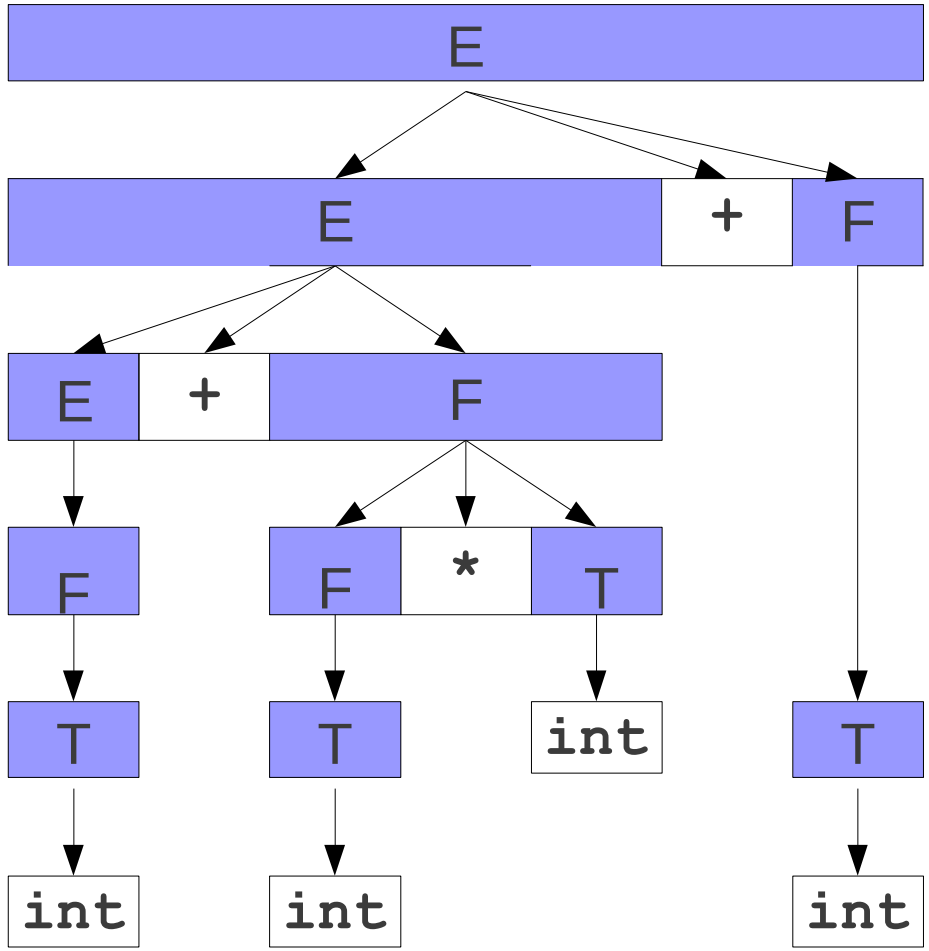
- When using a shift/reduce parser, we must decide whether to shift or reduce at each point.
- We only want to reduce when we know we have a handle.
- **Question:** How can we tell that we might be looking at a handle?

Exploring the Left Side

- The handle will always appear at the end of string in the left side of the parser.
- Can *any* string appear on the left side of the parser, or are there restrictions on what sorts of strings can appear there?
- If we can find a pattern to the strings that can appear on the left side, we might be able to exploit it to detect handles.

Another Look at Handles

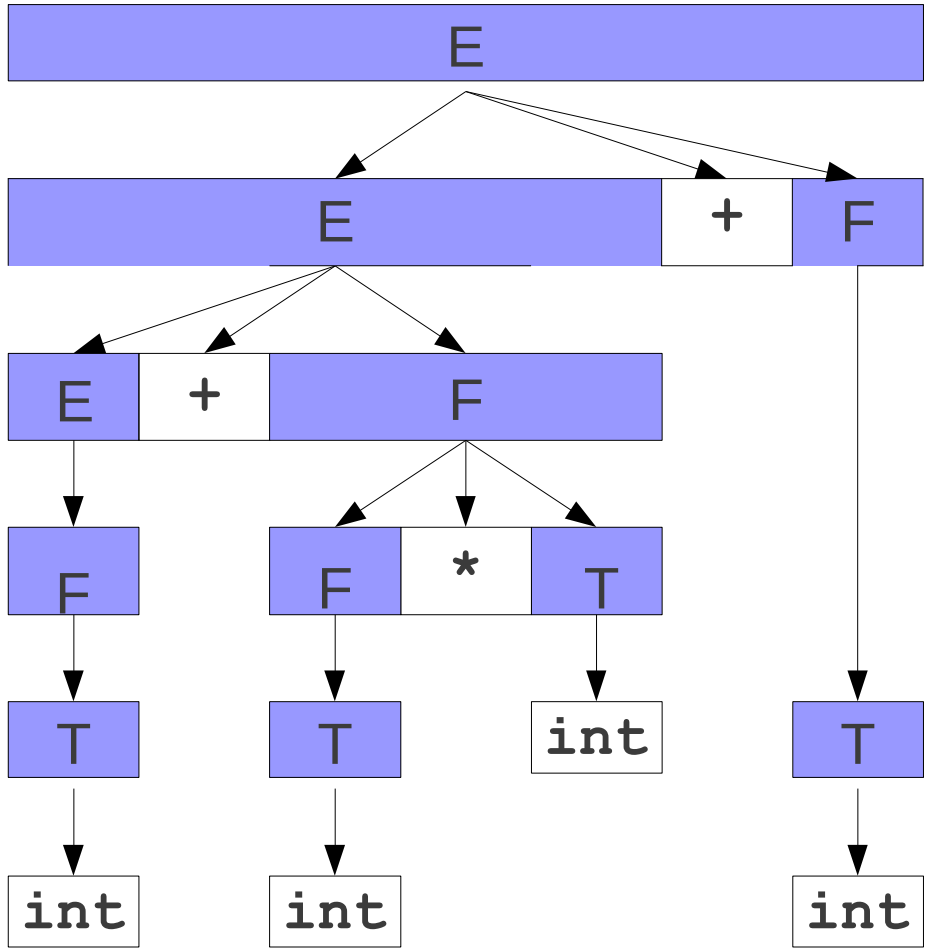
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

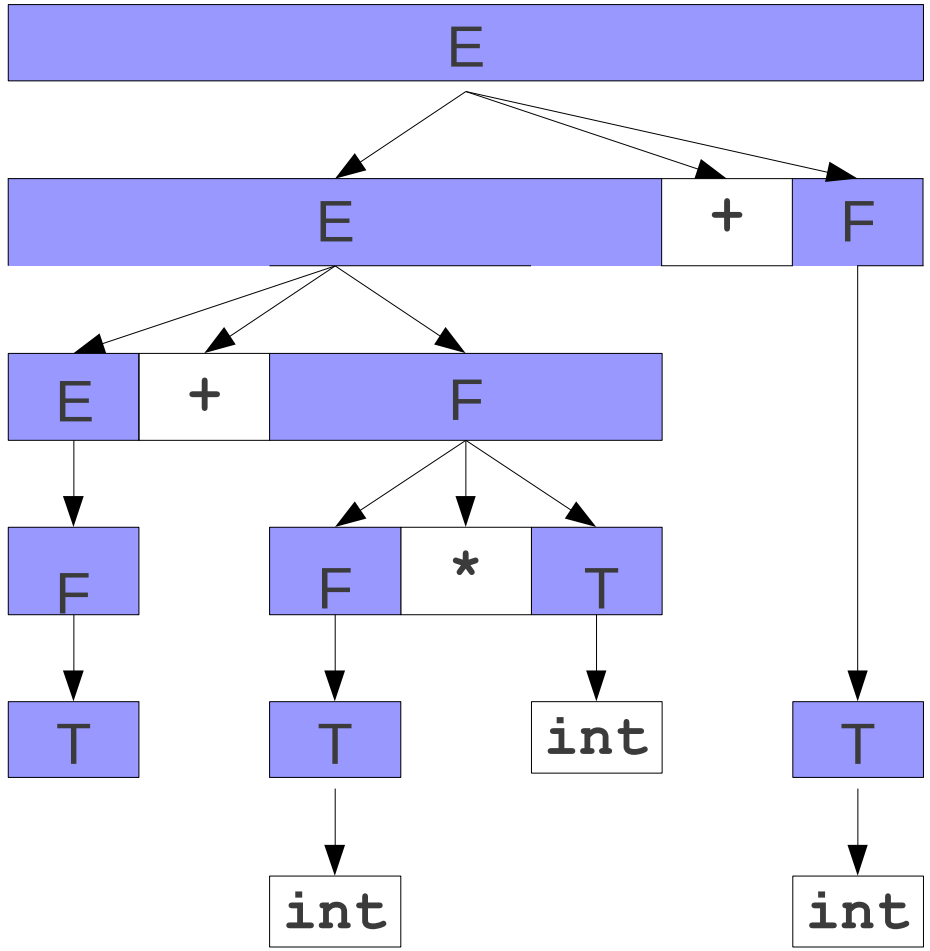
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

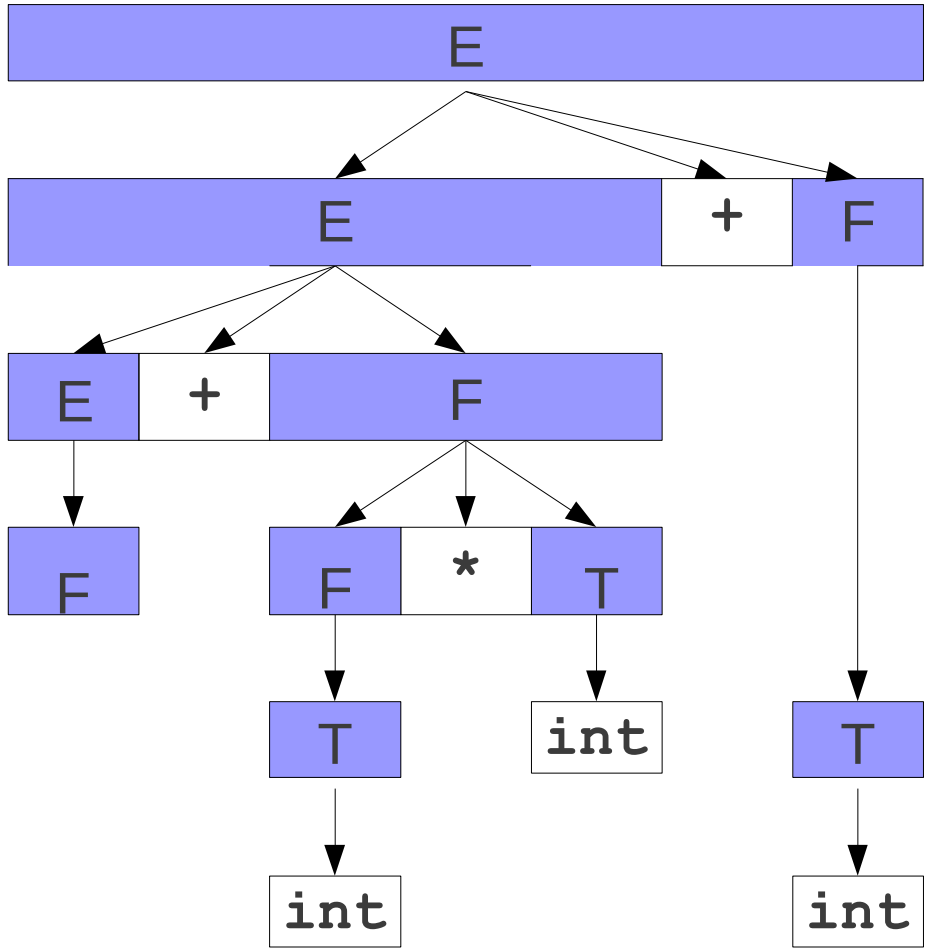
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

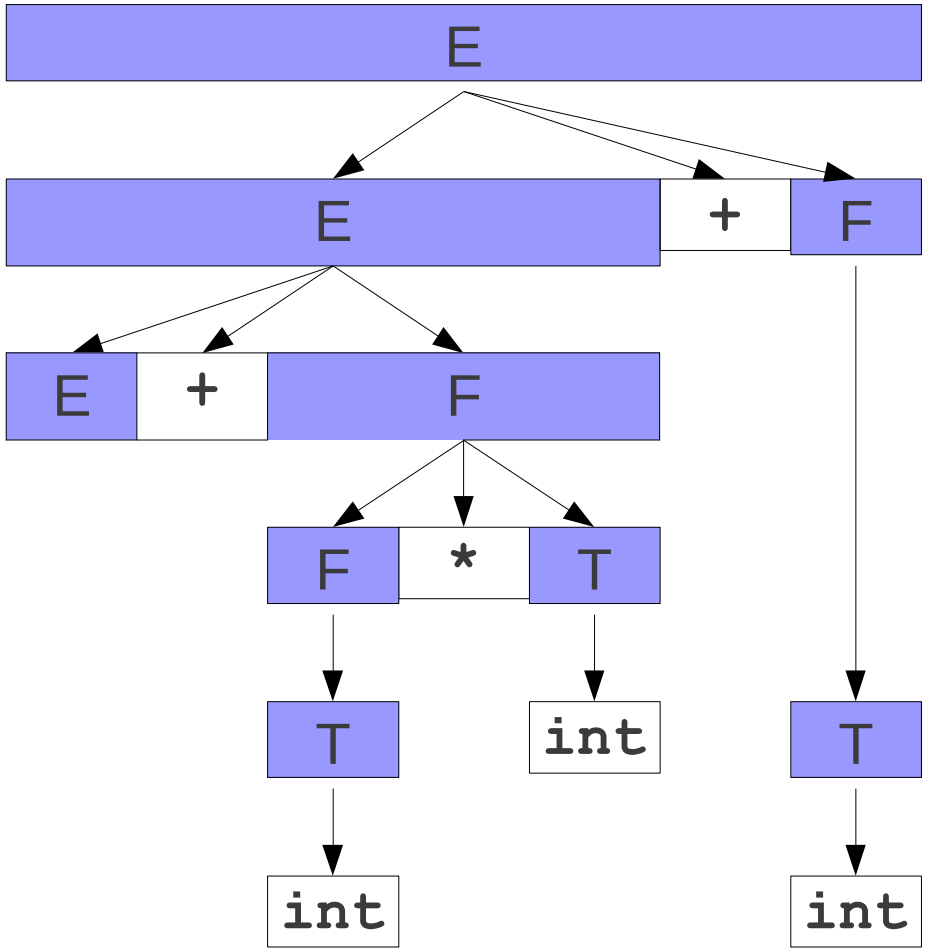
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

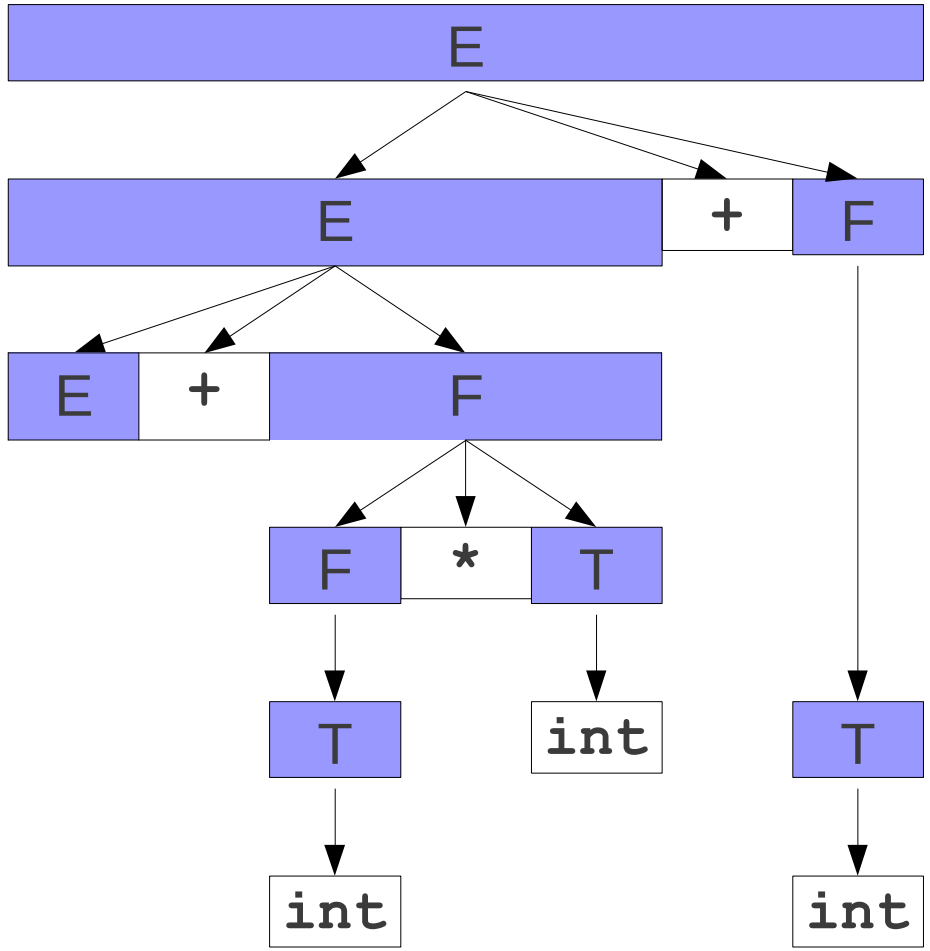
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

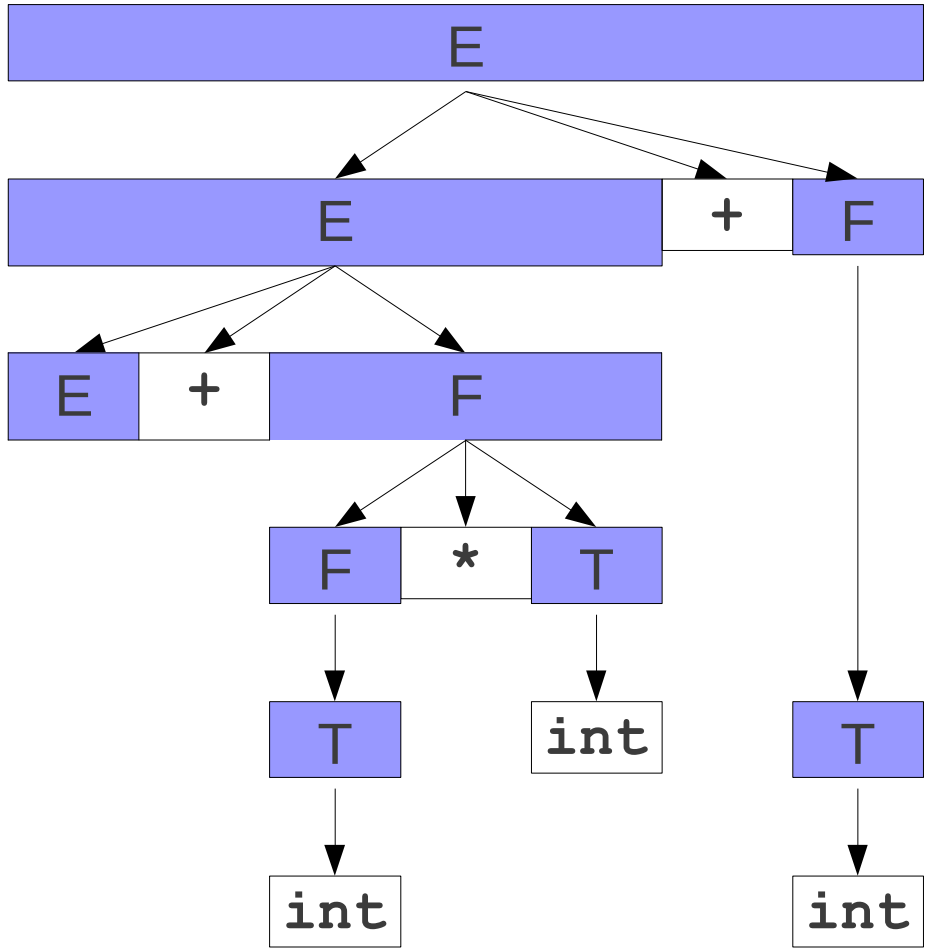
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

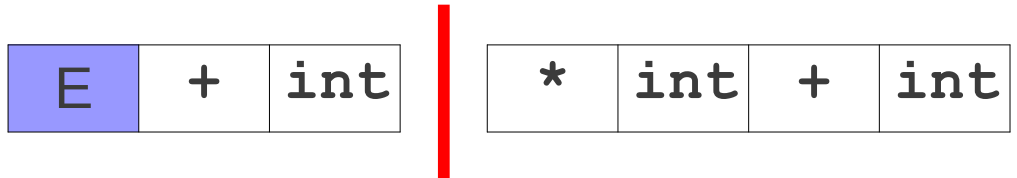


GEORGETOWN UNIVERSITY

Another Look at Handles

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

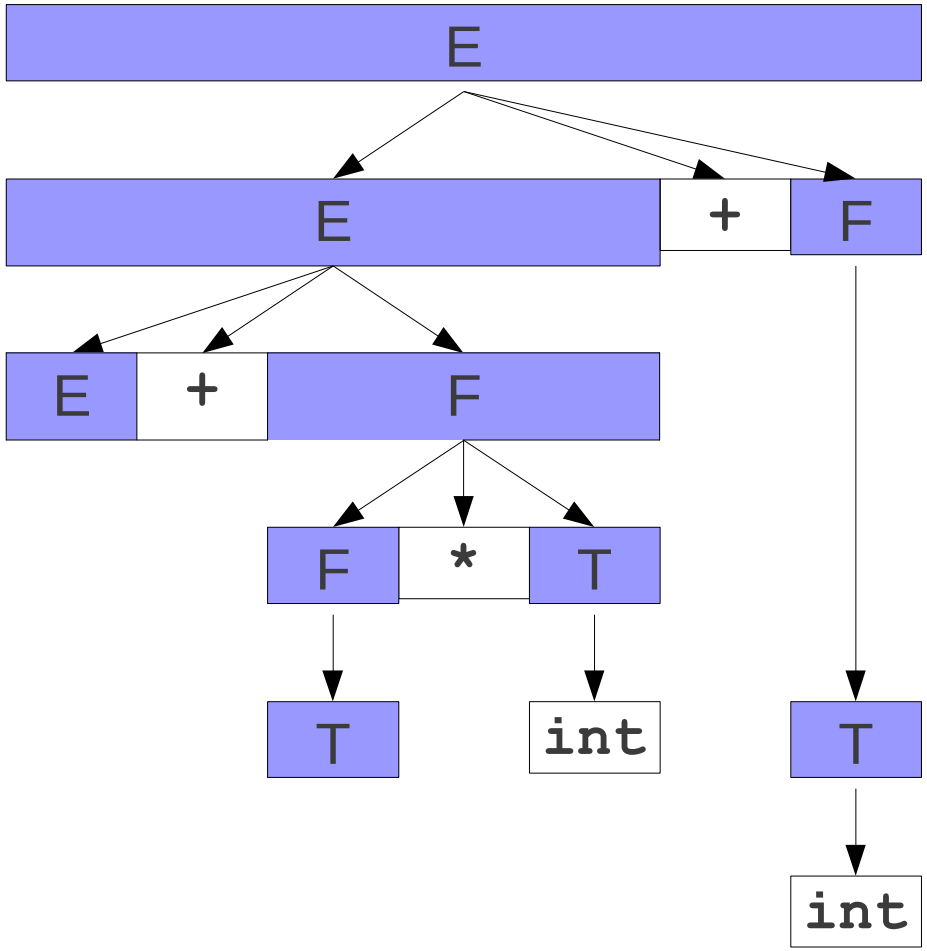




GEORGETOWN UNIVERSITY

Another Look at Handles

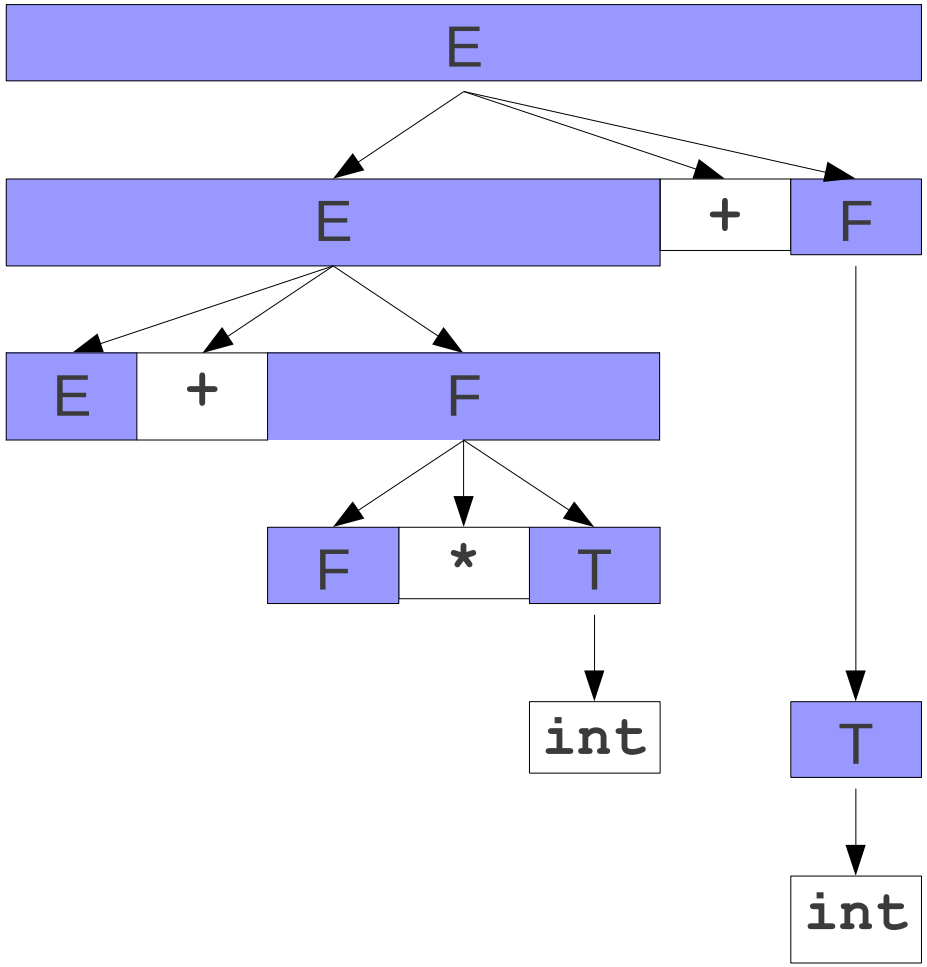
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



GEORGETOWN UNIVERSITY

Another Look at Handles

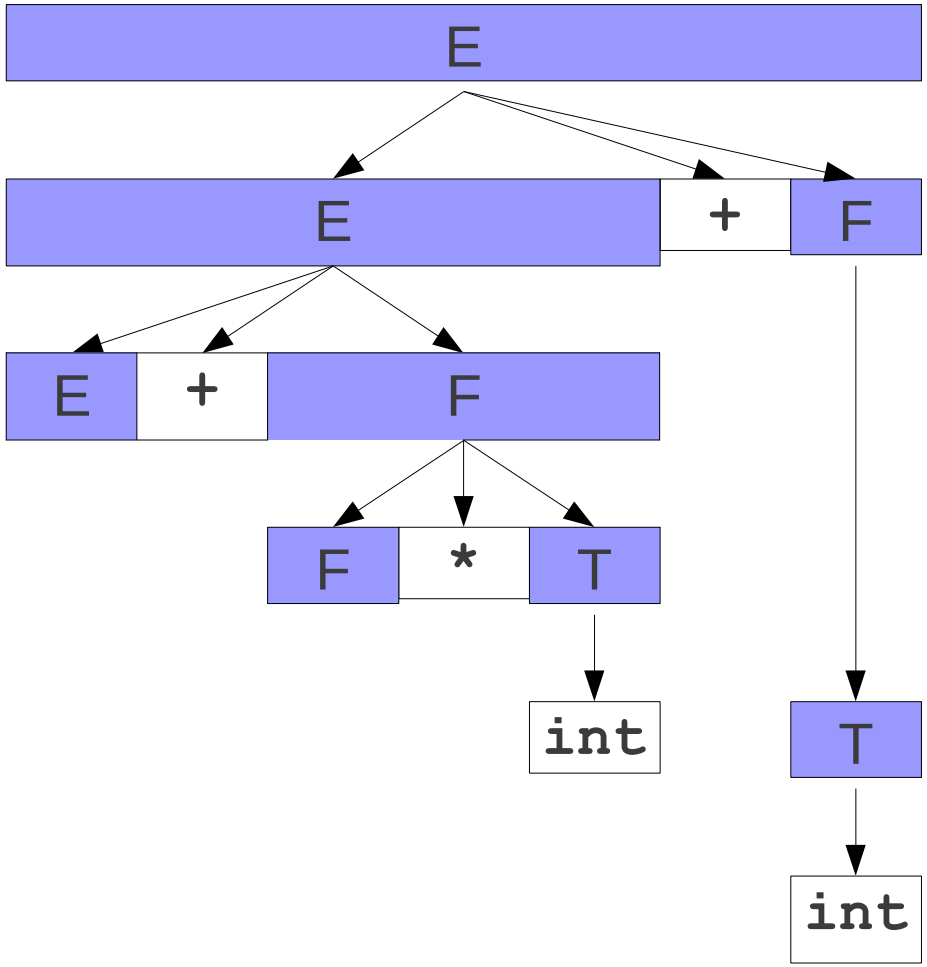
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

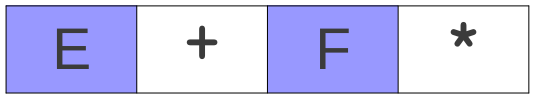


GEORGETOWN UNIVERSITY

Another Look at Handles

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

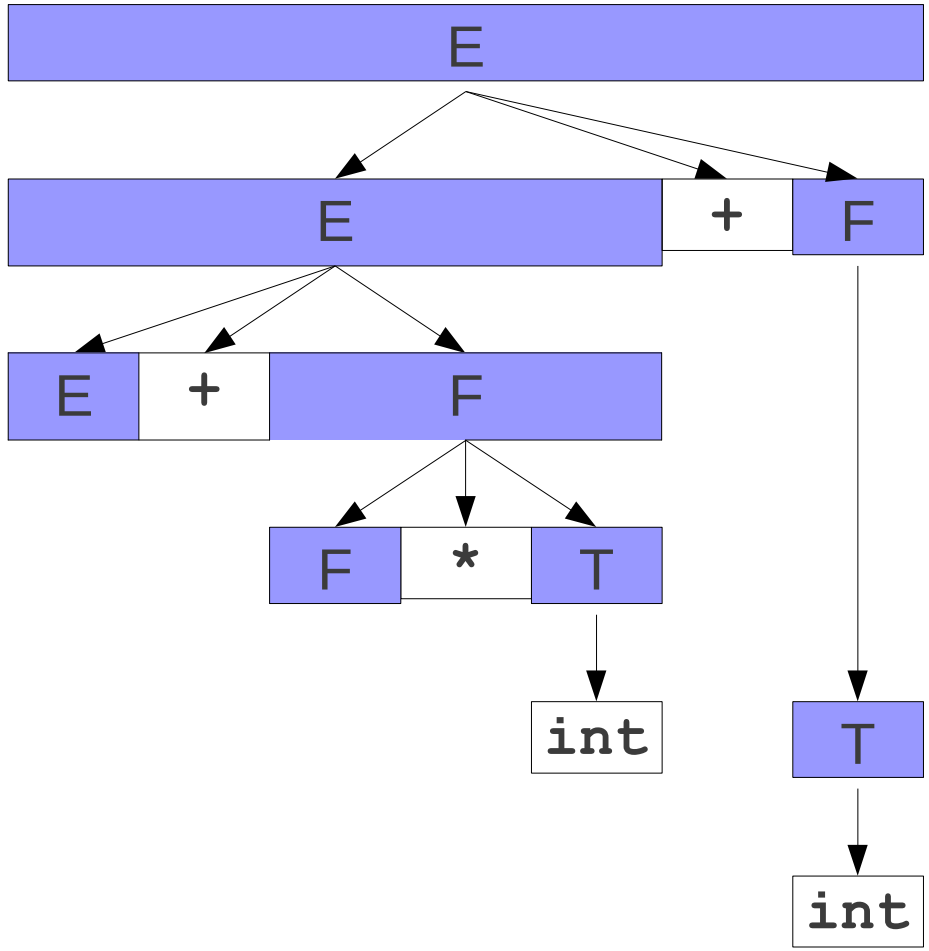


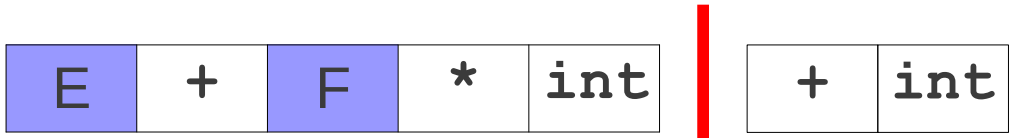


GEORGETOWN UNIVERSITY

Another Look at Handles

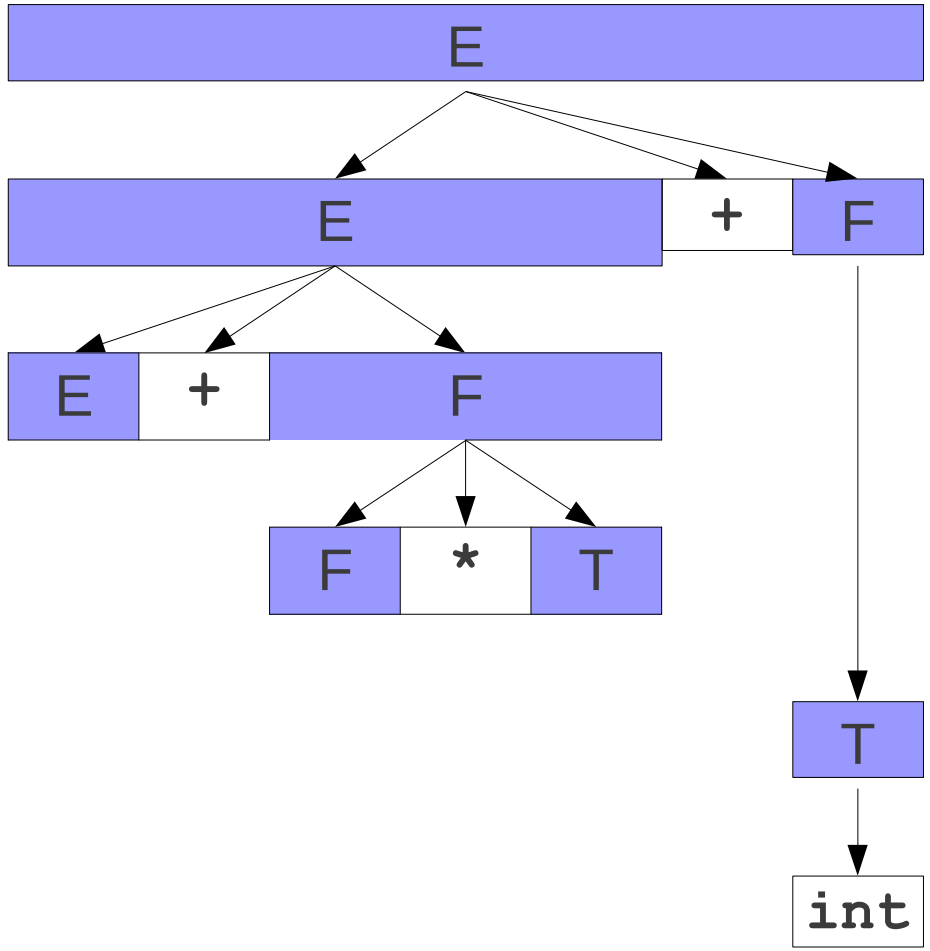
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

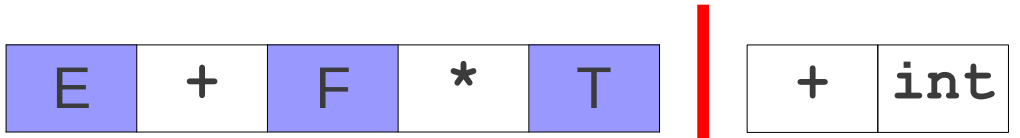




Another Look at Handles

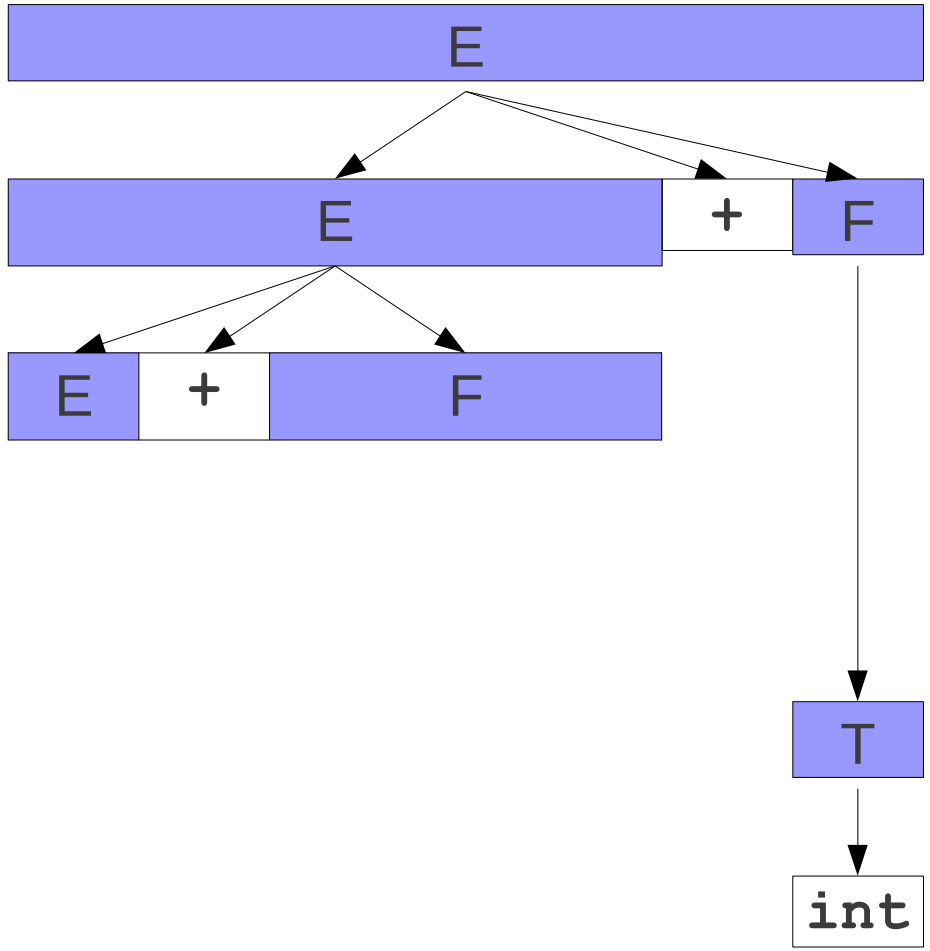
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)





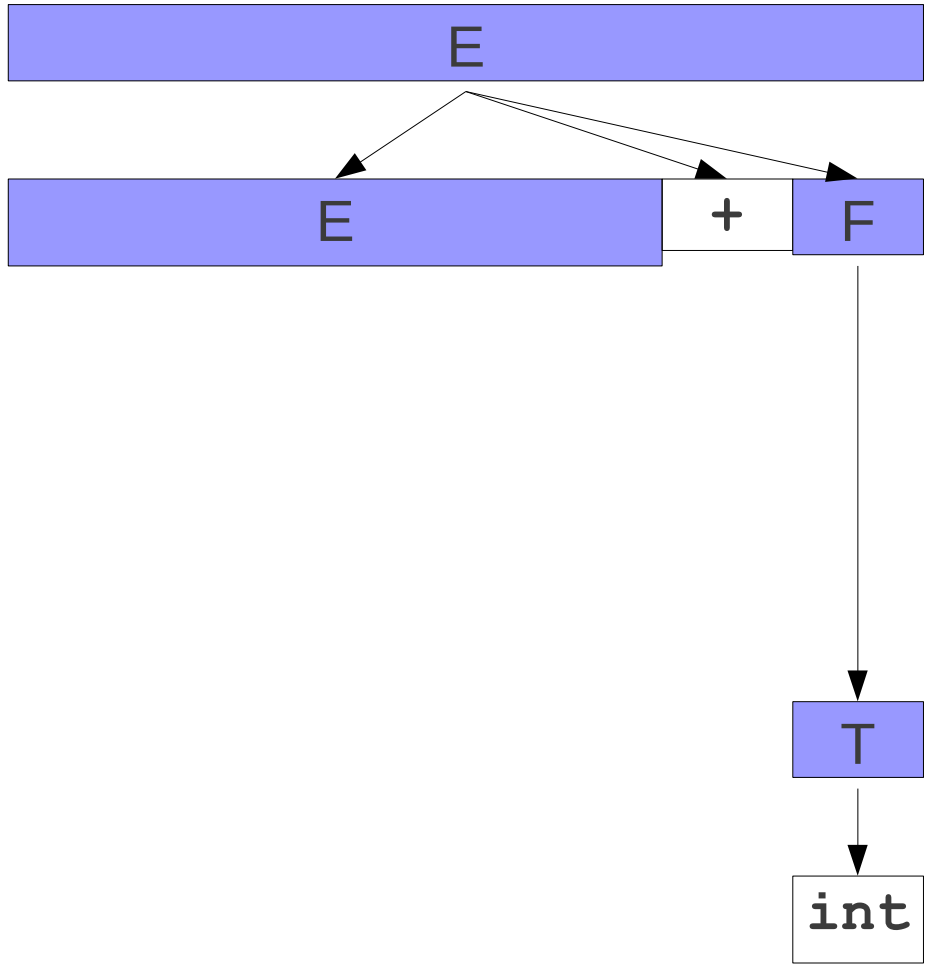
Another Look at Handles

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



Another Look at Handles

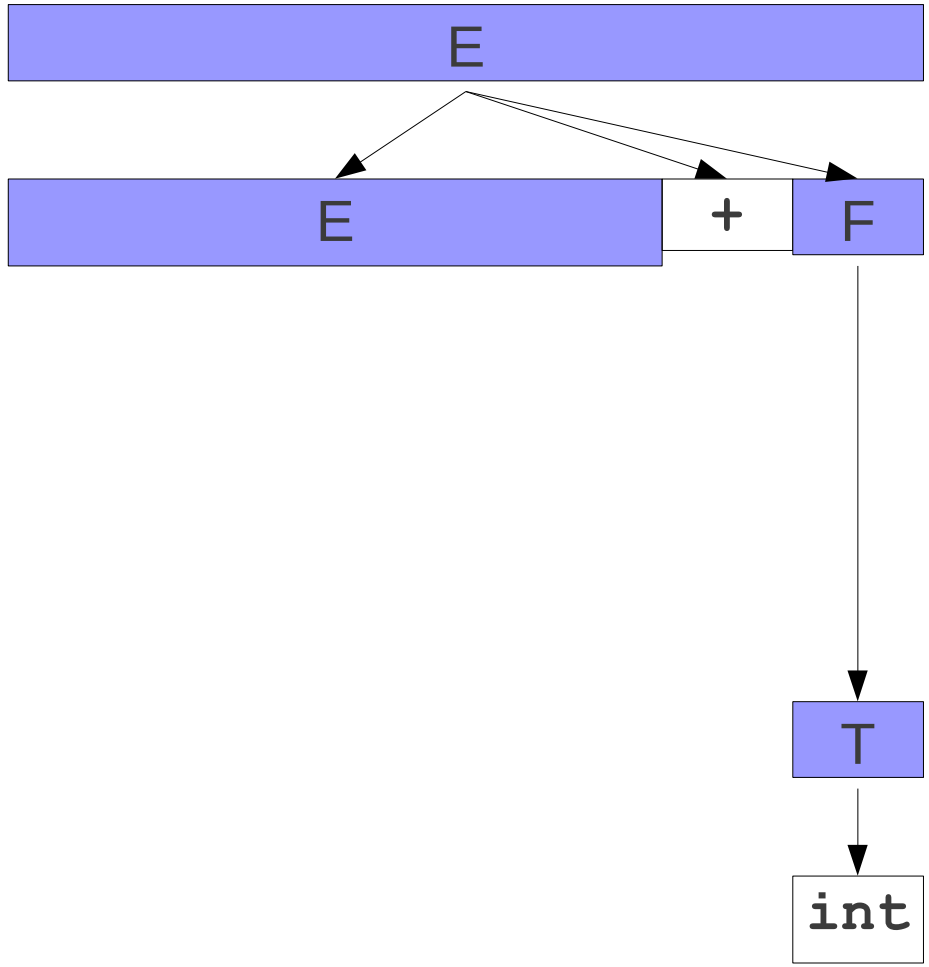
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

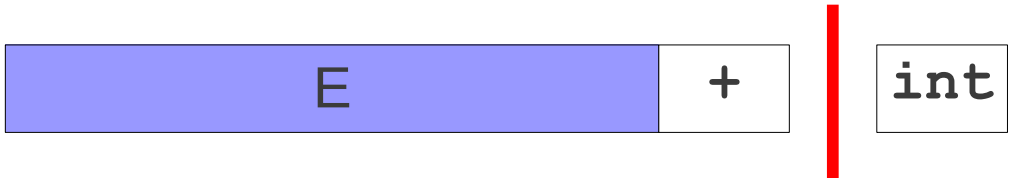


E | + int

Another Look at Handles

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

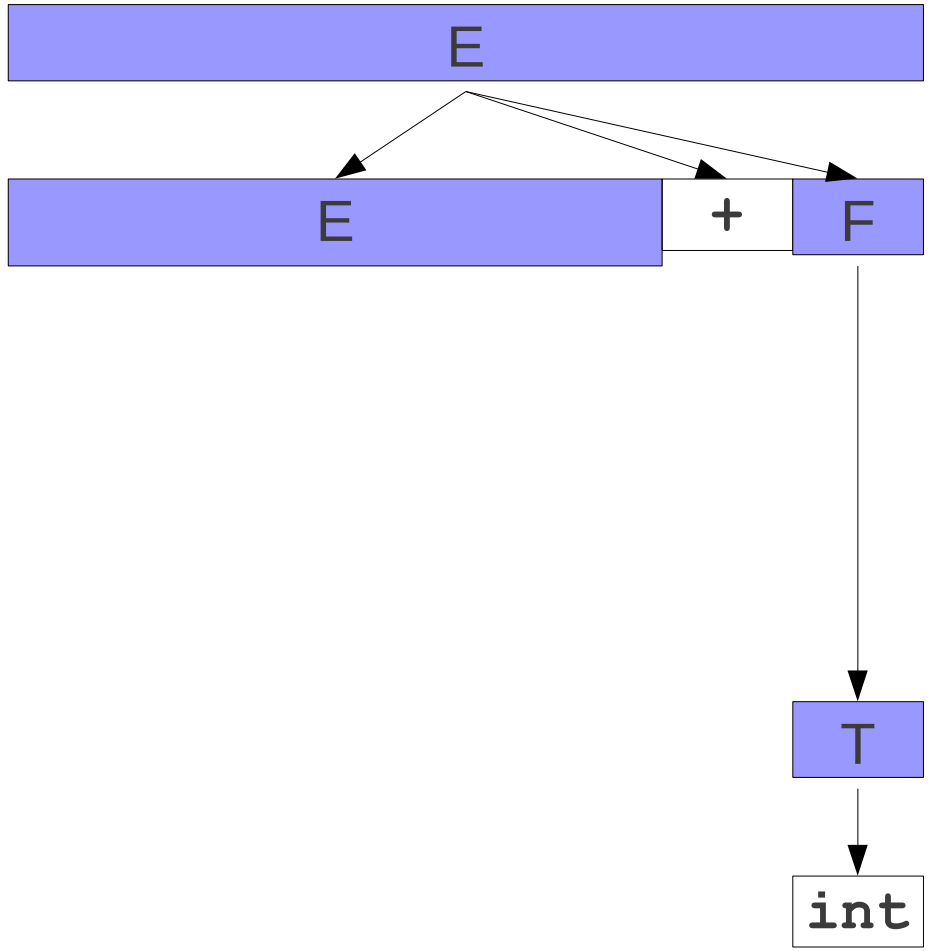




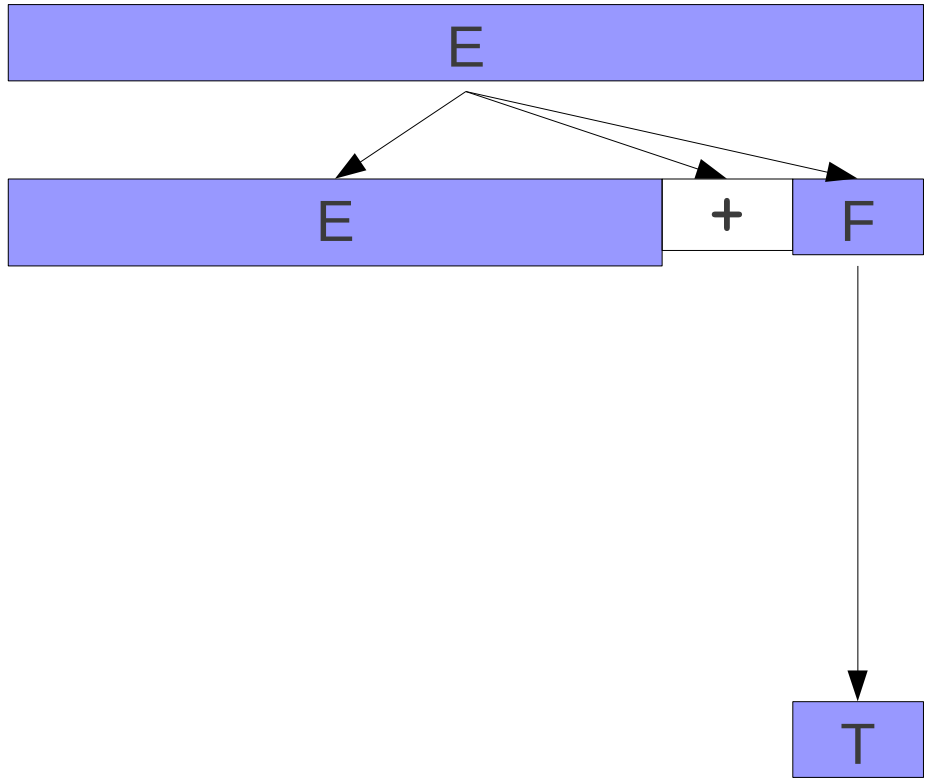
GEORGETOWN UNIVERSITY

Another Look at Handles

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

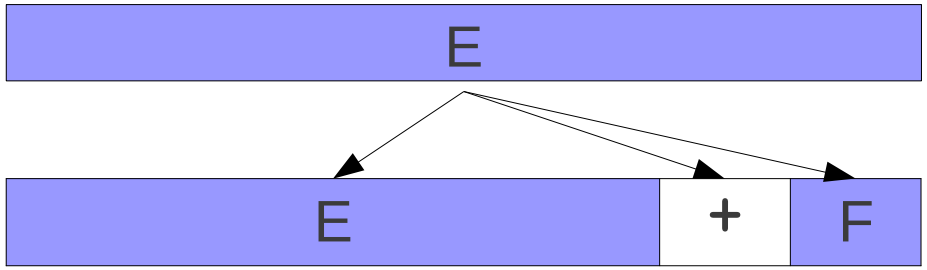


Another Look at Handles



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

Another Look at Handles



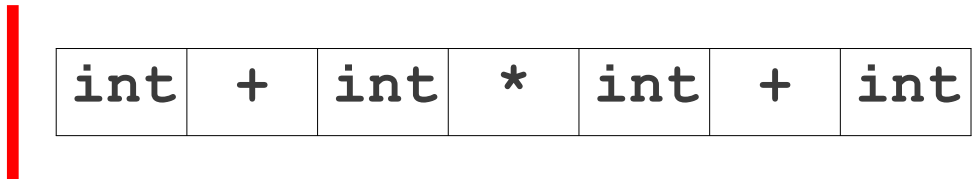
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

Another Look at Handles


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

Tracking Our Position

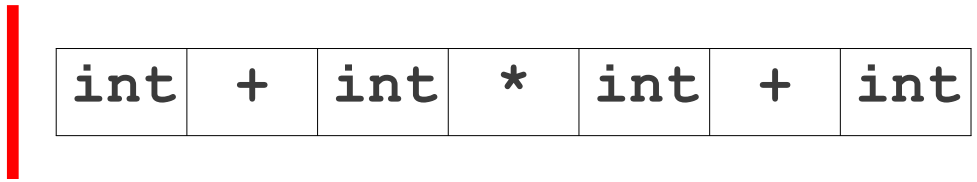
```
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)
```



GEORGETOWN UNIVERSITY

Tracking Our Position

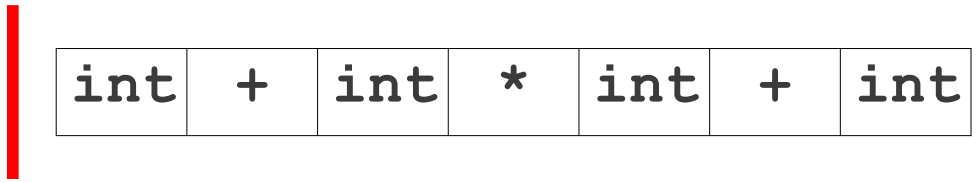
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



Tracking Our Position

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

S → · E



Tracking Our Position

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

S → · E
E → · E + F

int	+	int	*	int	+	int

Tracking Our Position

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

S →· E
E →·E + F
E →·E + F

int	+	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → · F

int	+	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → · F
F → · T

int	+	int	*	int	+	int

Tracking Our Position

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

S → ·E
E → ·E + F
E → ·E + F
E → ·F
F → ·T
T → ·int

int	+	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → · E + F
E → · F
F → · T
T → int ·

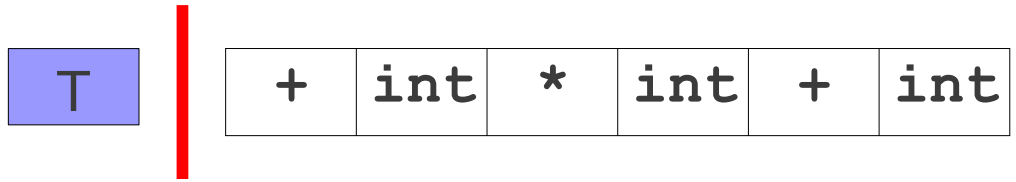
int	+	int	*	int	+	int

GEORGETOWN UNIVERSITY

Tracking Our Position

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

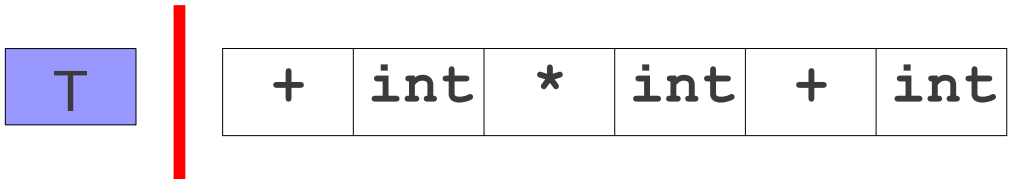
S → · E
E → · E + F
E → · E + F
E → · F
F → · T



Tracking Our Position

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

S → ·E
E → ·E + F
E → ·E + F
E → ·F
F → T·

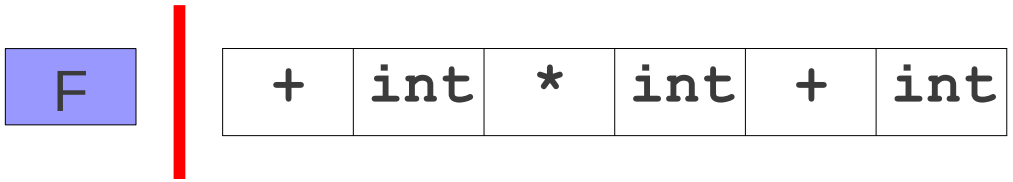


GEORGETOWN UNIVERSITY

Tracking Our Position

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

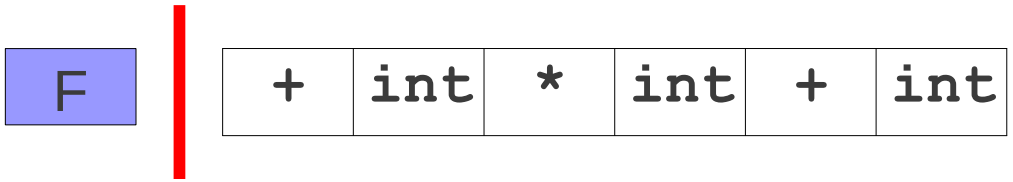
S → · E
E → · E + F
E → · E + F
E → · F



Tracking Our Position

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

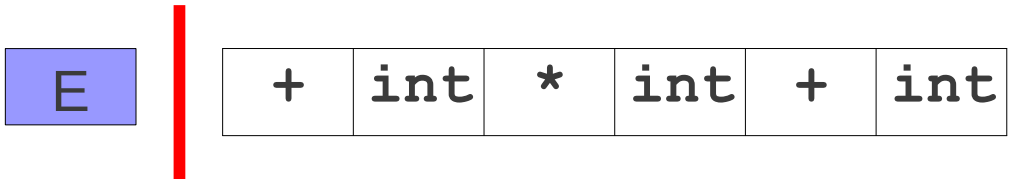
S → ·E
E → ·E + F
E → ·E + F
E → F ·



Tracking Our Position

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

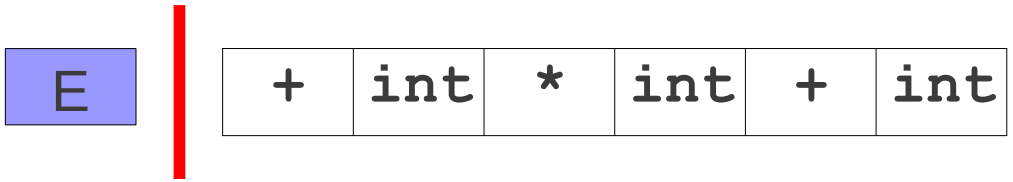
S → · E
E → · E + F
E → · E + F



Tracking Our Position

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

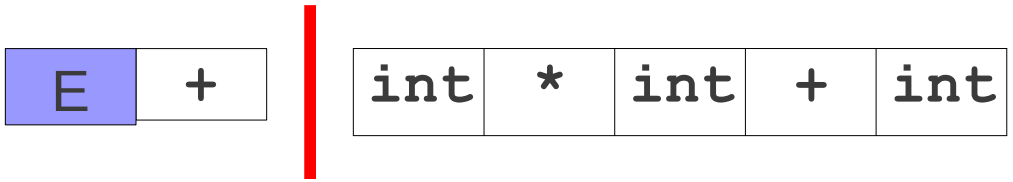
S → · E
E → · E + F
E → E · + F



Tracking Our Position

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

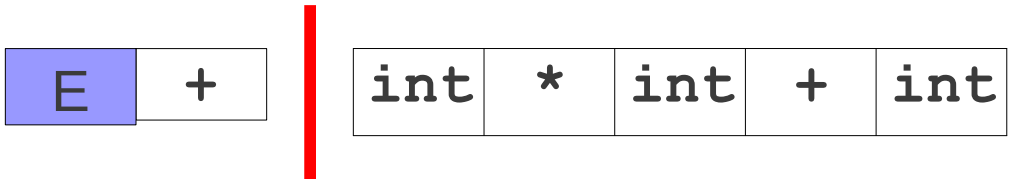
S → · E
E → · E + F
E → E + · F



Tracking Our Position

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

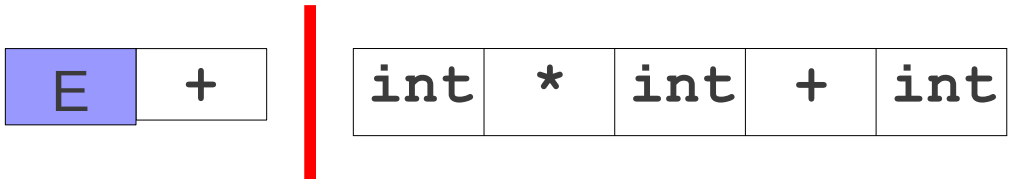
S → · E
E → · E + F
E → E + · F
F → · F * T



Tracking Our Position

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

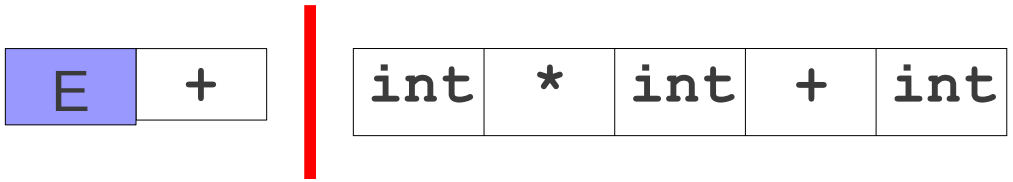
S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T



Tracking Our Position

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

S → ·E
E → ·E + F
E → E + ·F
F → ·F * T
F → ·T
T → ·int



Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T
T → int ·

E	+	int	*	int	+	int

Tracking Our Position

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

S → · E
E → · E + F
E → E + · F
F → · F * T
F → · T

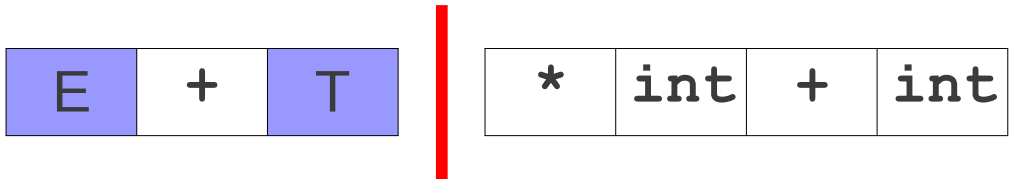
E	+	T

*	int	+	int

Tracking Our Position

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

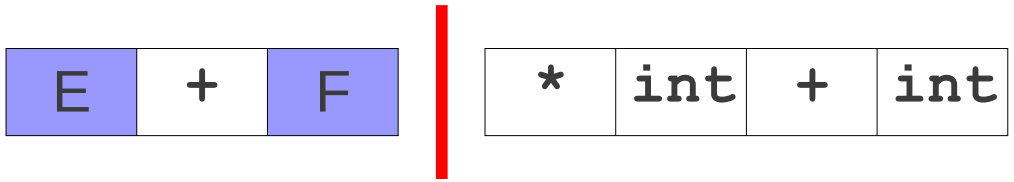
S → ·E
E → ·E + F
E → E + ·F
F → ·F * T
F → T·



Tracking Our Position

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

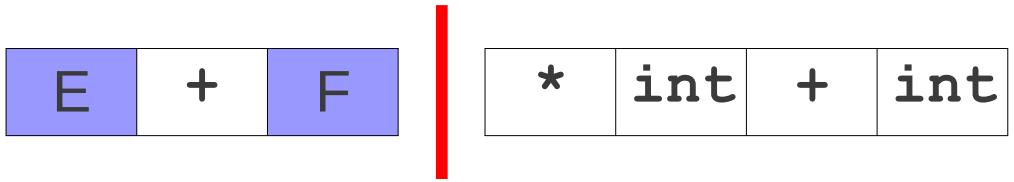
S → ·E
E → ·E + F
E → E + ·F
F → ·F *T



Tracking Our Position

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

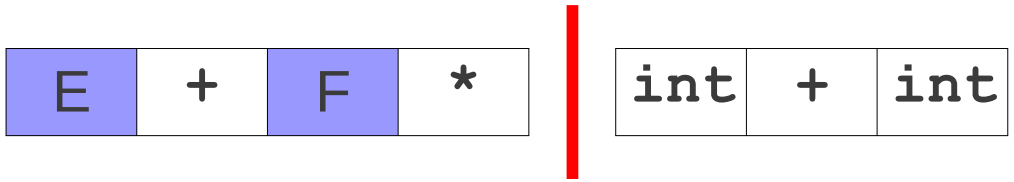
S → · E
E → · E + F
E → E + · F
F → F · * T



Tracking Our Position

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

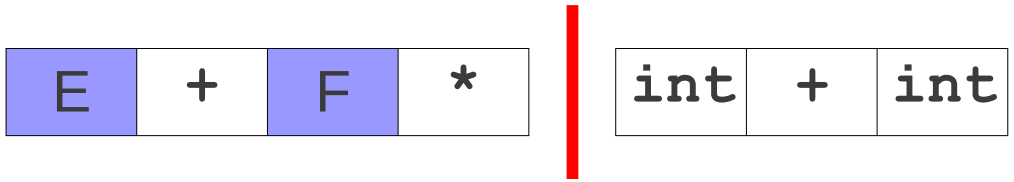
S → · E
E → · E + F
E → E + · F
F → F * · T



Tracking Our Position

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

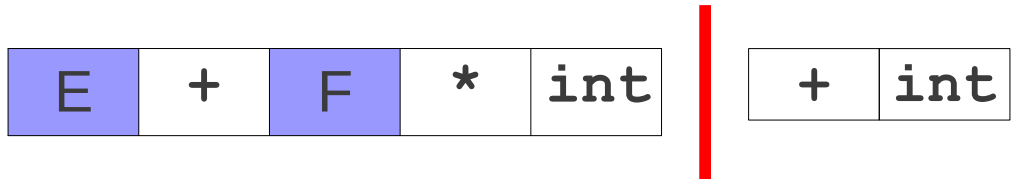
S → ·E
E → ·E + F
E → E + ·F
F → F * ·T
T → ·int



Tracking Our Position

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

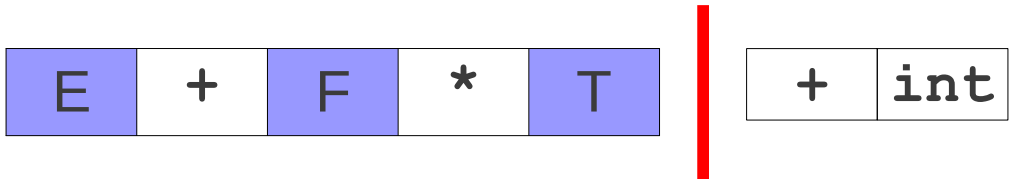
S → · E
E → · E + F
E → E + · F
F → F * · T
T → int ·



Tracking Our Position

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

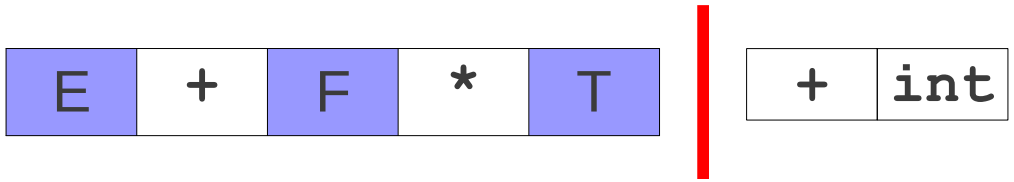
S → · E
E → · E + F
E → E + · F
F → F * · T



Tracking Our Position

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

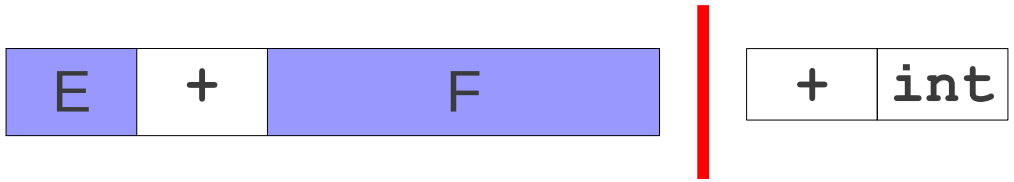
S → · E
E → · E + F
E → E + · F
F → F * T ·



Tracking Our Position

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

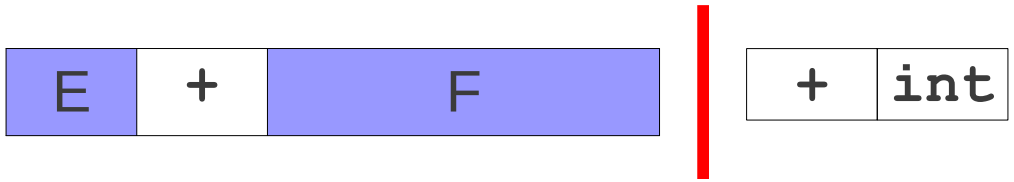
S → · E
E → · E + F
E → E + · F



Tracking Our Position

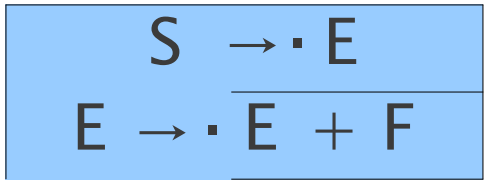
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

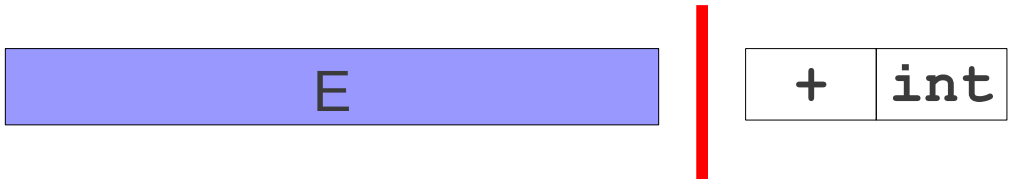
S → · E
E → · E + F
E → E + F ·



Tracking Our Position

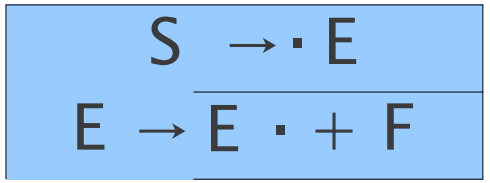
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

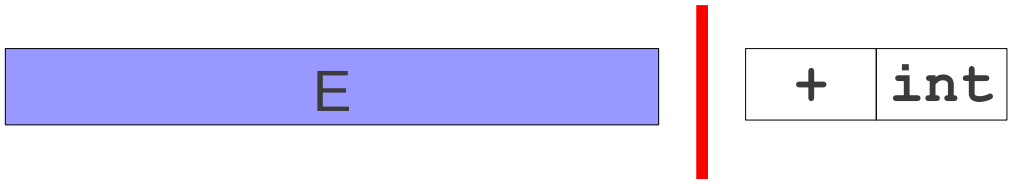




Tracking Our Position

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





Tracking Our Position

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

S → · E
E → E + · F

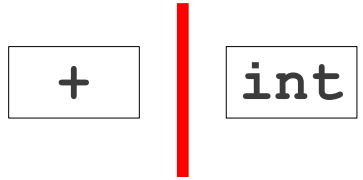
E | + | int

Tracking Our Position

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

S → ·E
E → E + ·F
F → ·T

E



Tracking Our Position

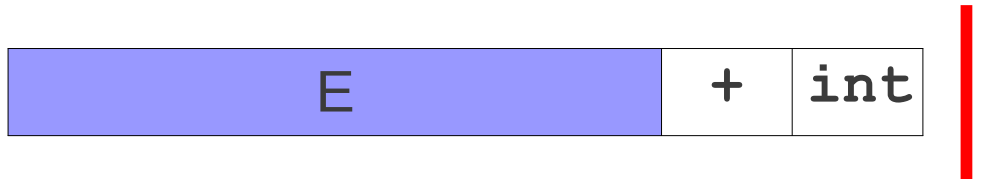
```
S →E
E →F
E →E + F
F →F * T
F →T
T →int
T →(E)
```

S →·E
E →E +·F
F →·T
T →·int

Tracking Our Position

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

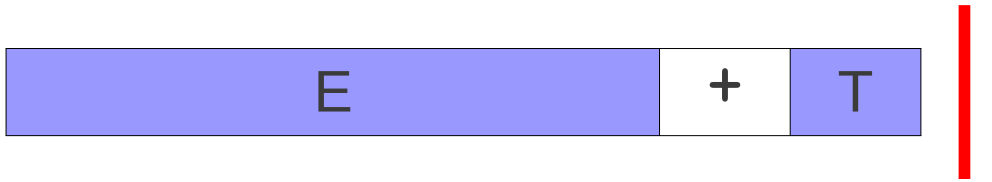
S → · E
E → E + · F
F → · T
T → int ·



Tracking Our Position

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

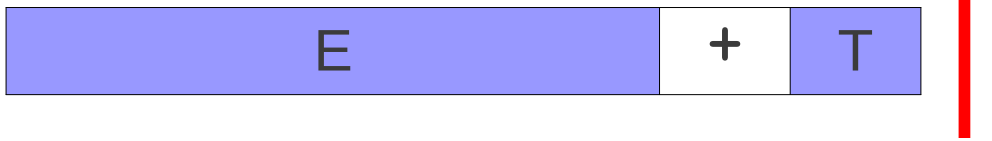
S → · E
E → E + · F
F → · T



Tracking Our Position

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

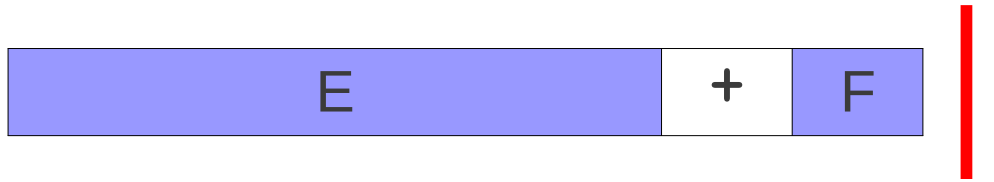
S → ·E
E → E + ·F
F → T·



Tracking Our Position

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

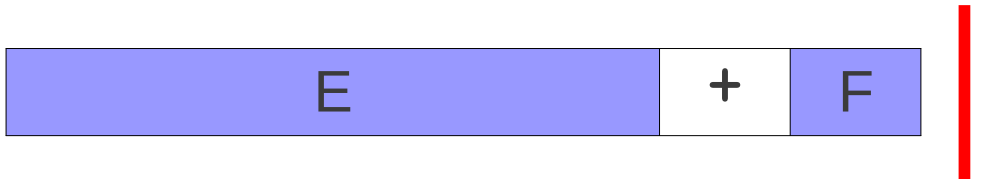
S →· E
E → E + · F



Tracking Our Position

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

S → · E
E → E + F ·



GEORGETOWN UNIVERSITY

Tracking Our Position

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

S → · E

E

Tracking Our Position

S → E ·

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

Generating Left-Hand Sides

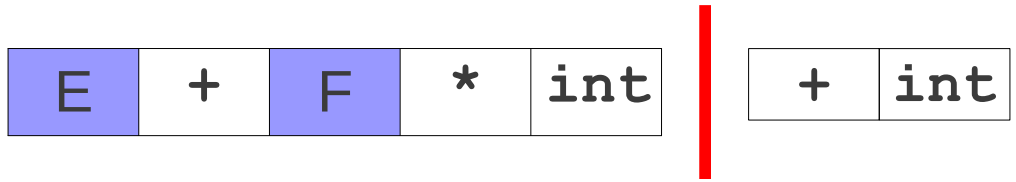
- At any instant in time, the contents of the left side of the parser can be described using the following process:
  - Trace out, from the start symbol, the series of productions that have not yet been completed and where we are in each production.
  - For each production, in order, output all of the symbols up to the point where we change from one production to the next.

Recognizing Left-Hand Sides

- Given that we have a procedure for *generating* left-hand sides, can we build a procedure for *recognizing* those left-hand sides?
- Idea: At each point, track
  - Which production we are in, and
  - Where we are in that production.
- At each point, we can do one of two things:
  - Match the next symbol of the candidate left-hand side with the next symbol in the current production, or
  - If the next symbol of the candidate left-hand side is a nonterminal, nondeterministically guess which production to try next.

Recognizing Left-Hand Sides

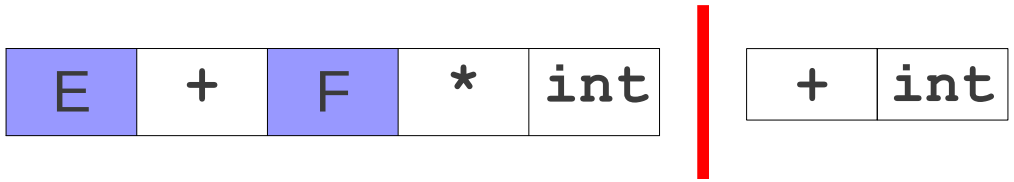
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



Recognizing Left-Hand Sides

S →·E

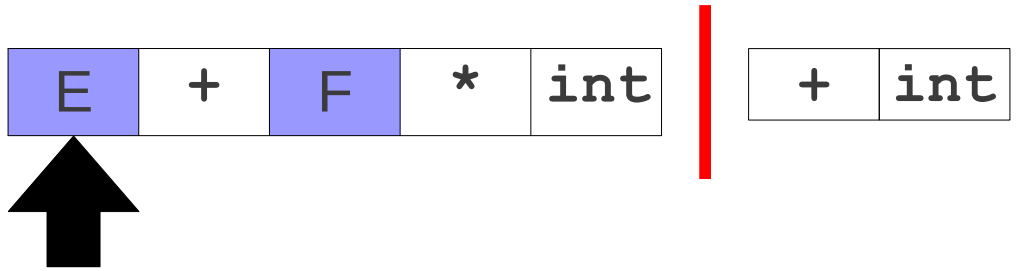
S →E
E →F
E →E + F
F →F * T
F →T
T →int
T →(E)



Recognizing Left-Hand Sides

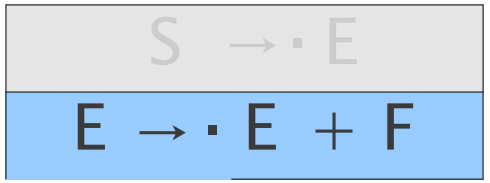
S →·E

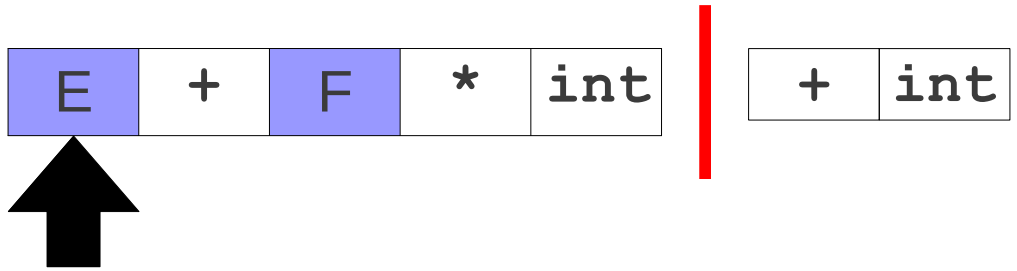
S →E
E →F
E →E + F
F →F * T
F →T
T →int
T →(E)



Recognizing Left-Hand Sides

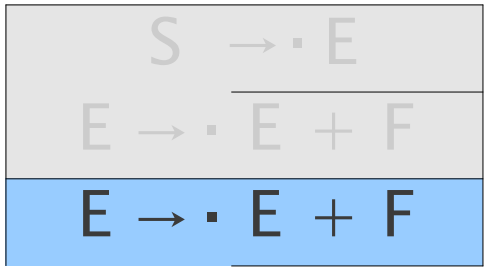
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

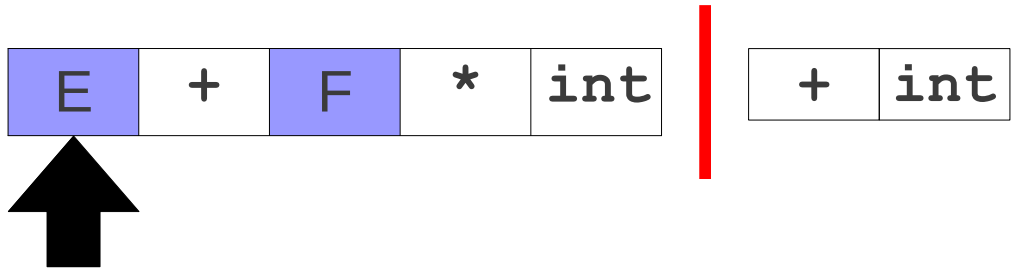




Recognizing Left-Hand Sides

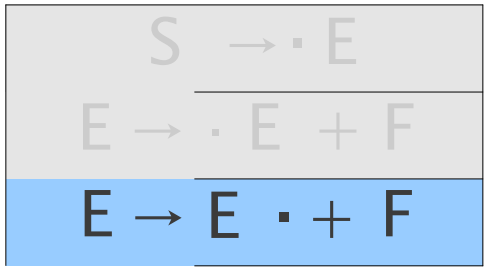
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

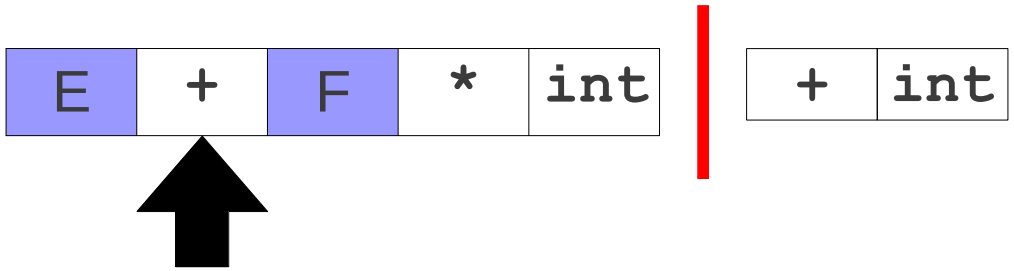




Recognizing Left-Hand Sides

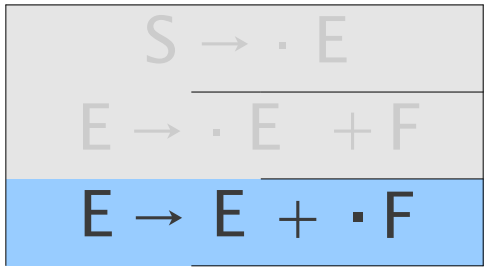
S →E
E →F
E →E + F
F →F * T
F →T
T →int
T →(E)

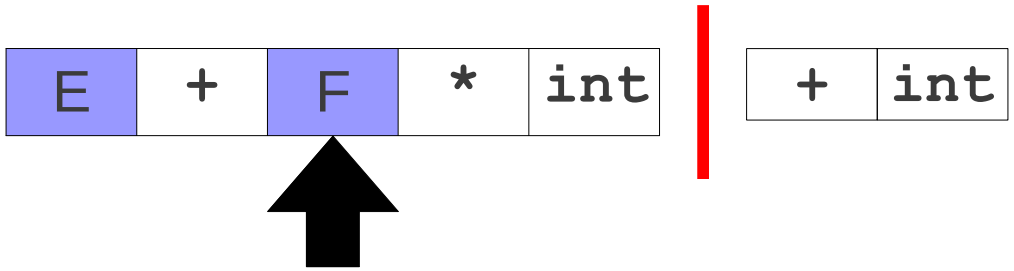




Recognizing Left-Hand Sides

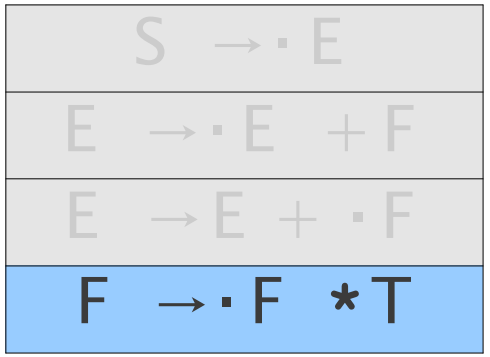
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

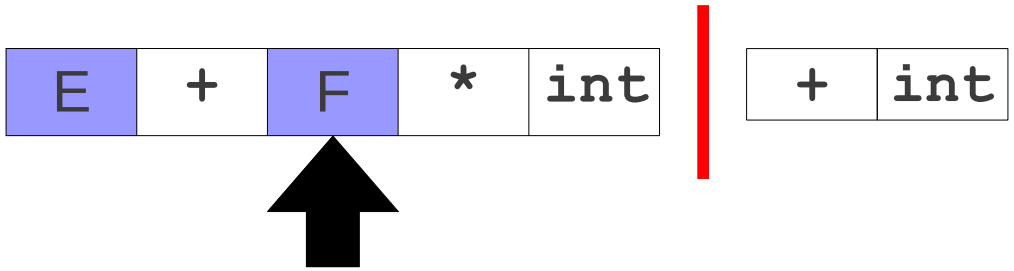




Recognizing Left-Hand Sides

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

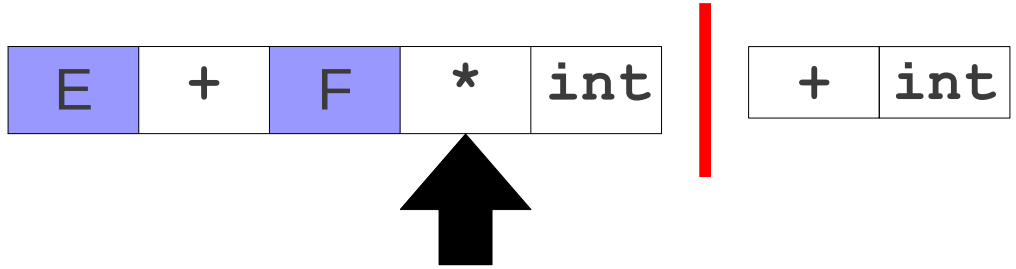




Recognizing Left-Hand Sides

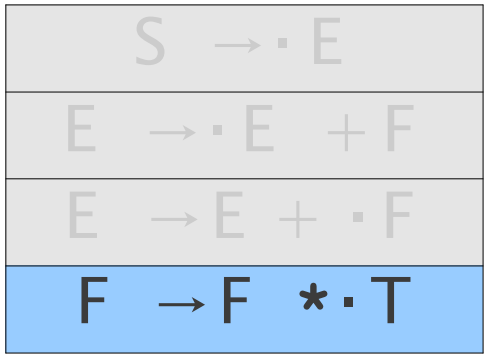
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

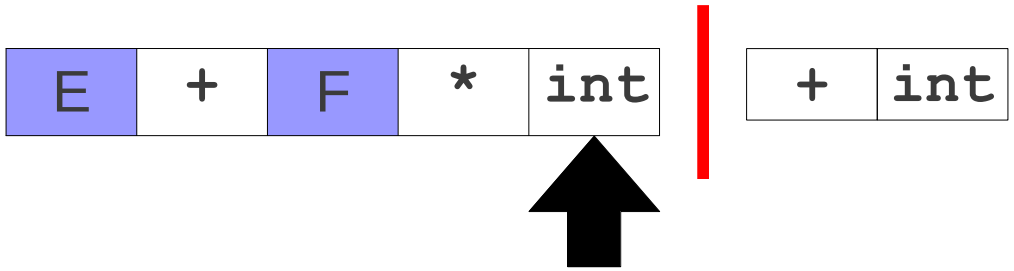
S → · E
E → · E + F
E → E + · F
F → F · * T



Recognizing Left-Hand Sides

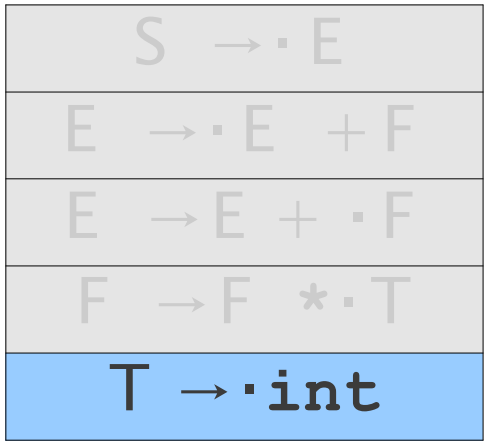
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

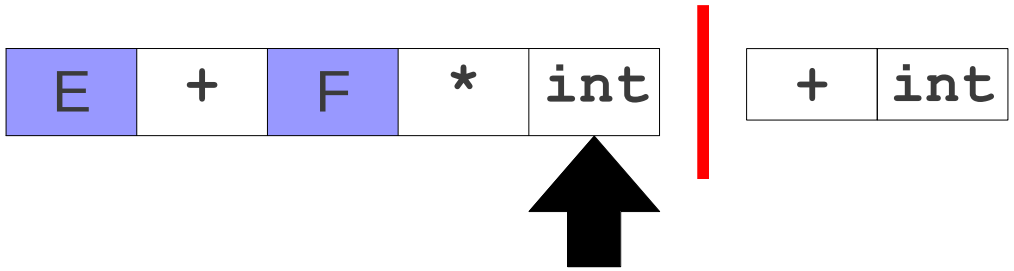




Recognizing Left-Hand Sides

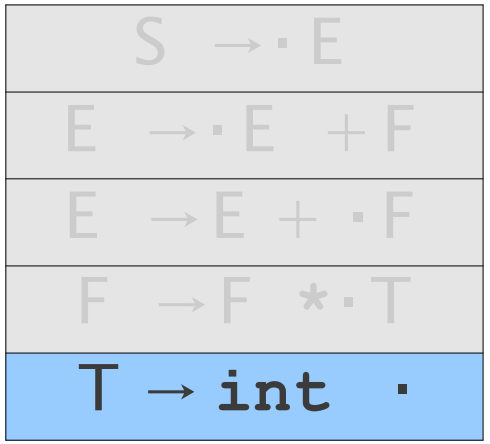
S → E
E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

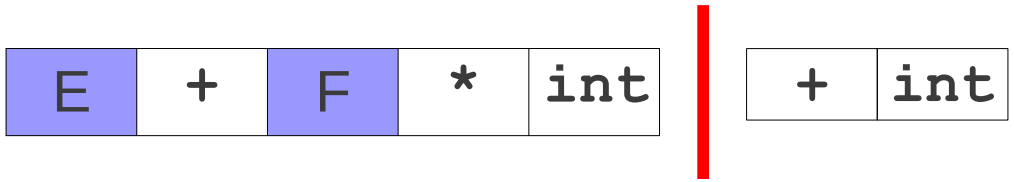




Recognizing Left-Hand Sides

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

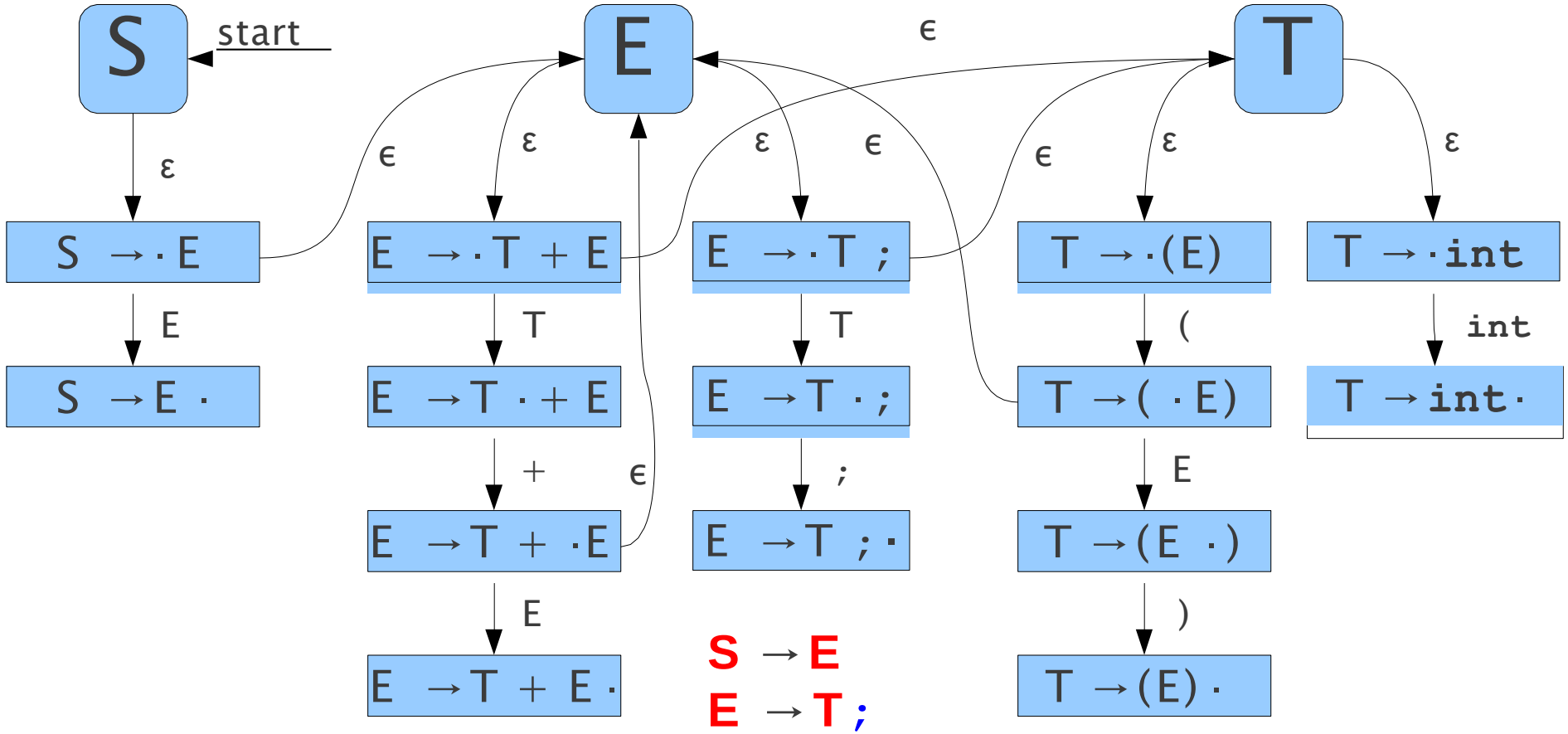




An Important Result

- There are only finitely many productions, and within those productions only finitely many positions.
- At any point in time, we only need to track where we are in one production.
- There are only finitely many options we can take at any one point.
- **We can use a finite automaton as our recognizer.**

An Automaton for Left Areas



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

GEORGETOWN UNIVERSITY

Constructing the Automaton

- Create a state for each nonterminal.
- For each production $\mathbf{A} \rightarrow \gamma$:
  - Construct states $\mathbf{A} \rightarrow \alpha \cdot \omega$ for each possible way of splitting γ into two substrings α and ω.
  - Add transitions on x between $\mathbf{A} \rightarrow \alpha \cdot x\omega$ and $\mathbf{A} \rightarrow \alpha x \cdot \omega$.
- For each state $\mathbf{A} \rightarrow \alpha \cdot \mathbf{B}\omega$ for nonterminal $\mathbf{B}$, add an ϵ-transition from $\mathbf{A} \rightarrow \alpha \cdot \mathbf{B}\omega$ to $\mathbf{B}$.

Why This Matters

- Our initial goal was to find handles.
- When running this automaton, if we ever end up in a state with a rule of the form

$$A \rightarrow \omega \cdot$$

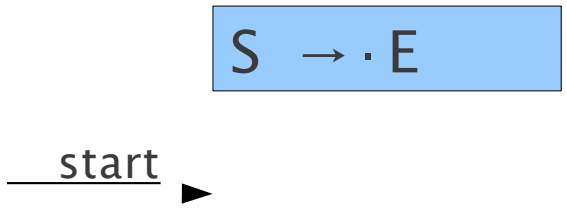
- Then we might be looking at a handle.
- This automaton can be used to discover possible handle locations!

Adding Determinism

- Typically, this handle-finding automaton is implemented deterministically.
- We could construct a deterministic parsing automaton by constructing the nondeterministic automaton and applying the subset construction, but there is a more direct approach.

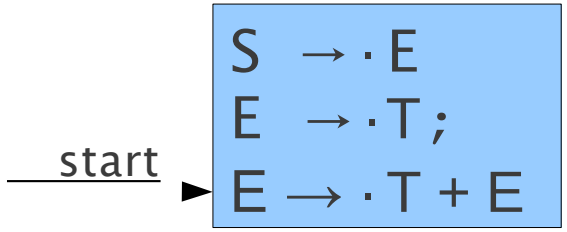
A Deterministic Automaton

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



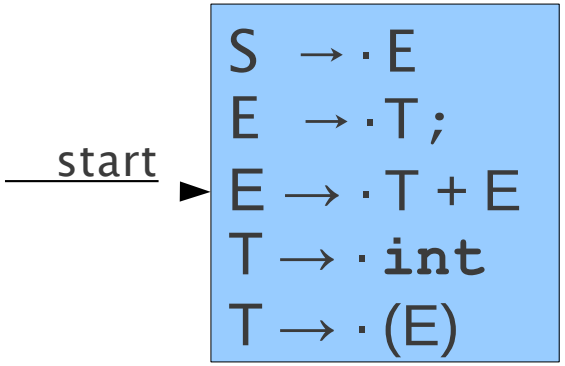
A Deterministic Automaton

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



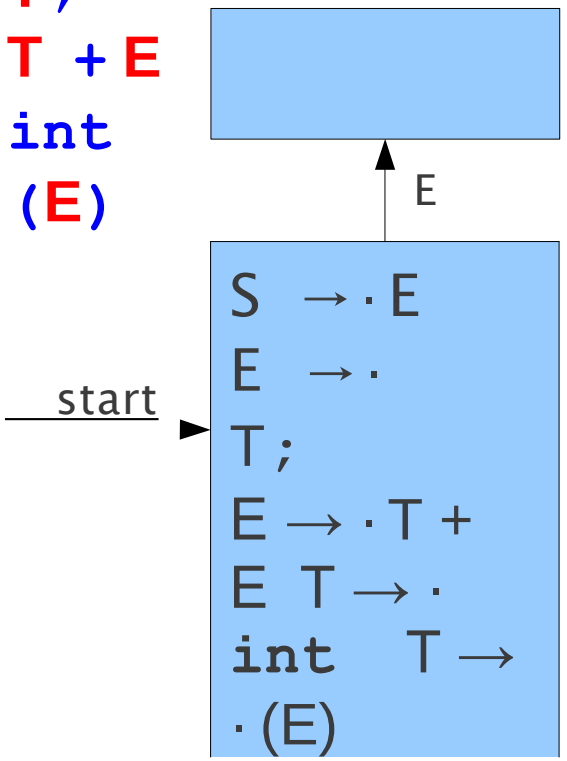
A Deterministic Automaton

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



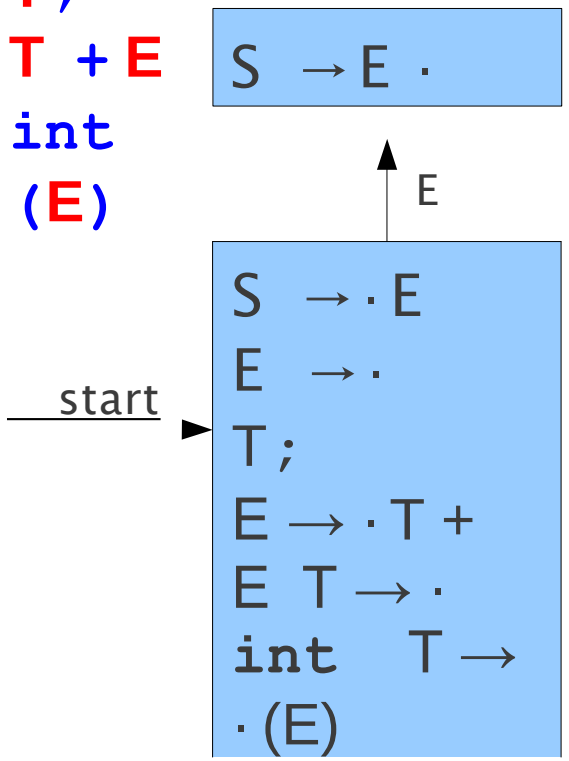
A Deterministic Automaton

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



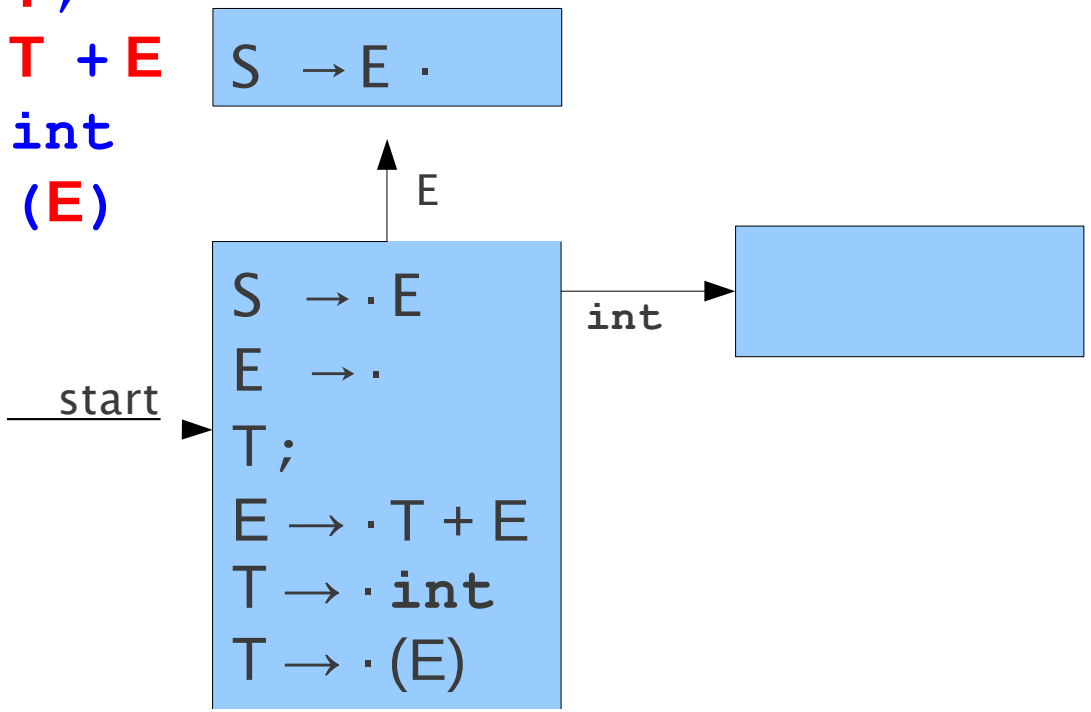
A Deterministic Automaton

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

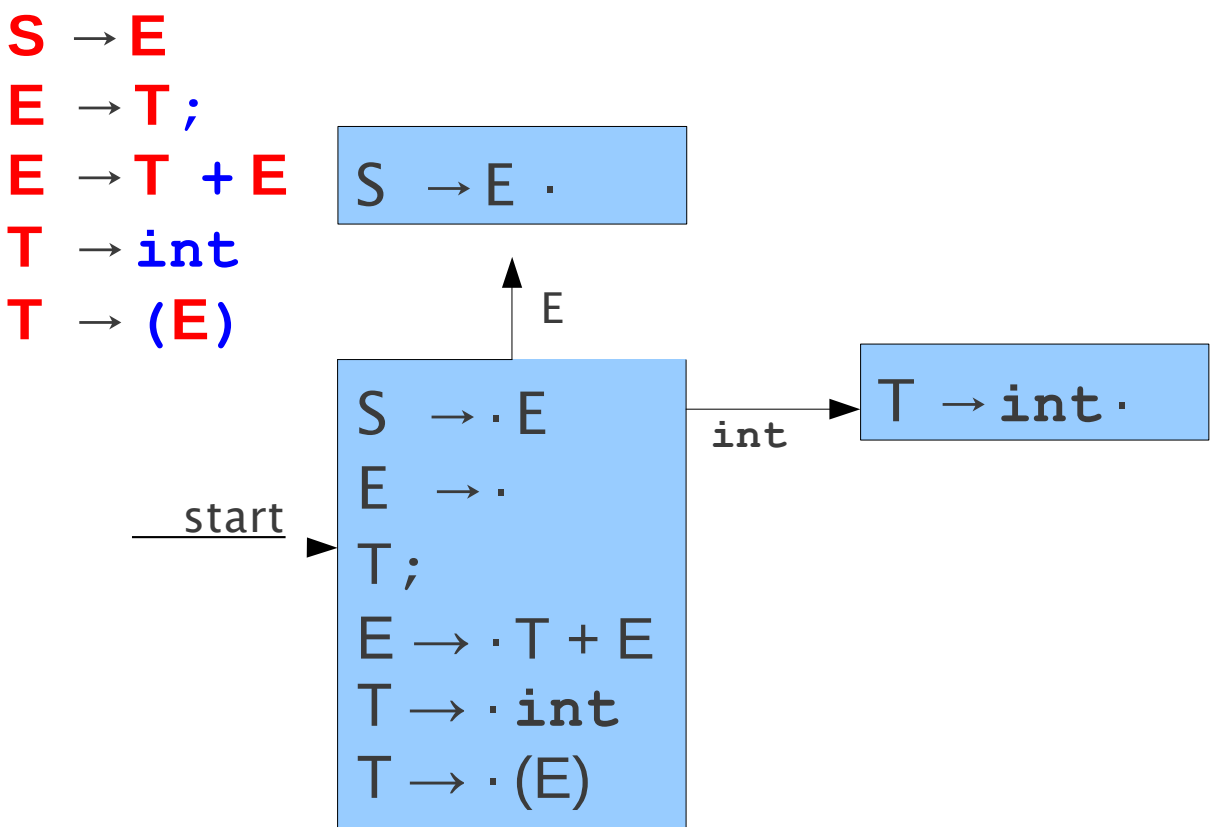


A Deterministic Automaton

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

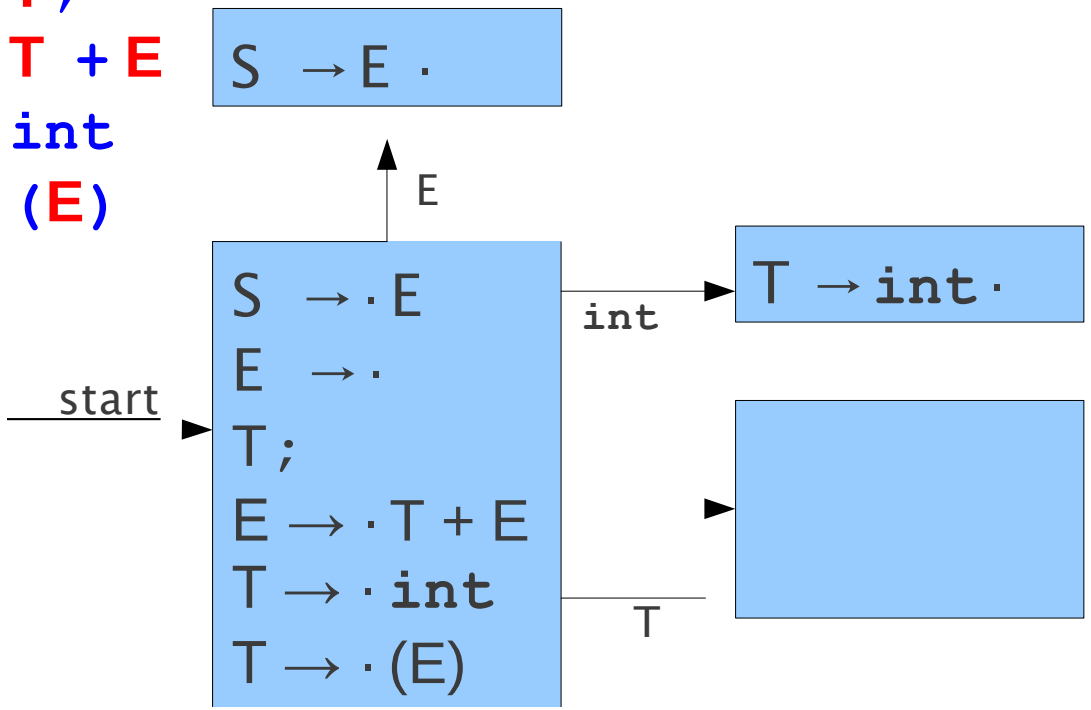


A Deterministic Automaton



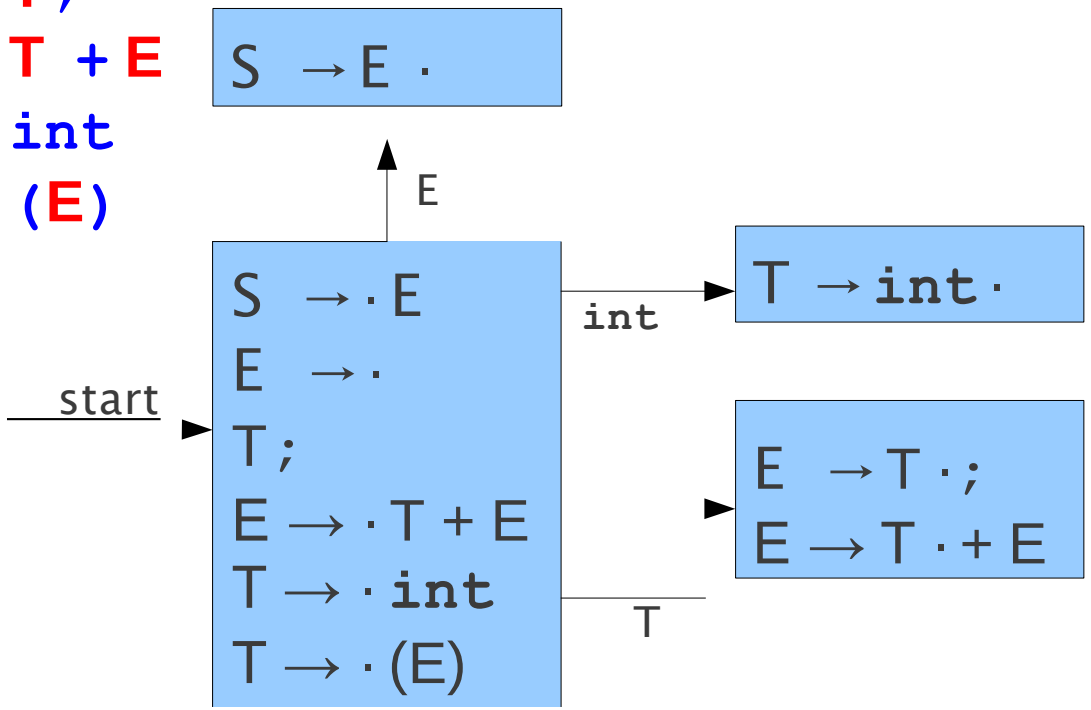
A Deterministic Automaton

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



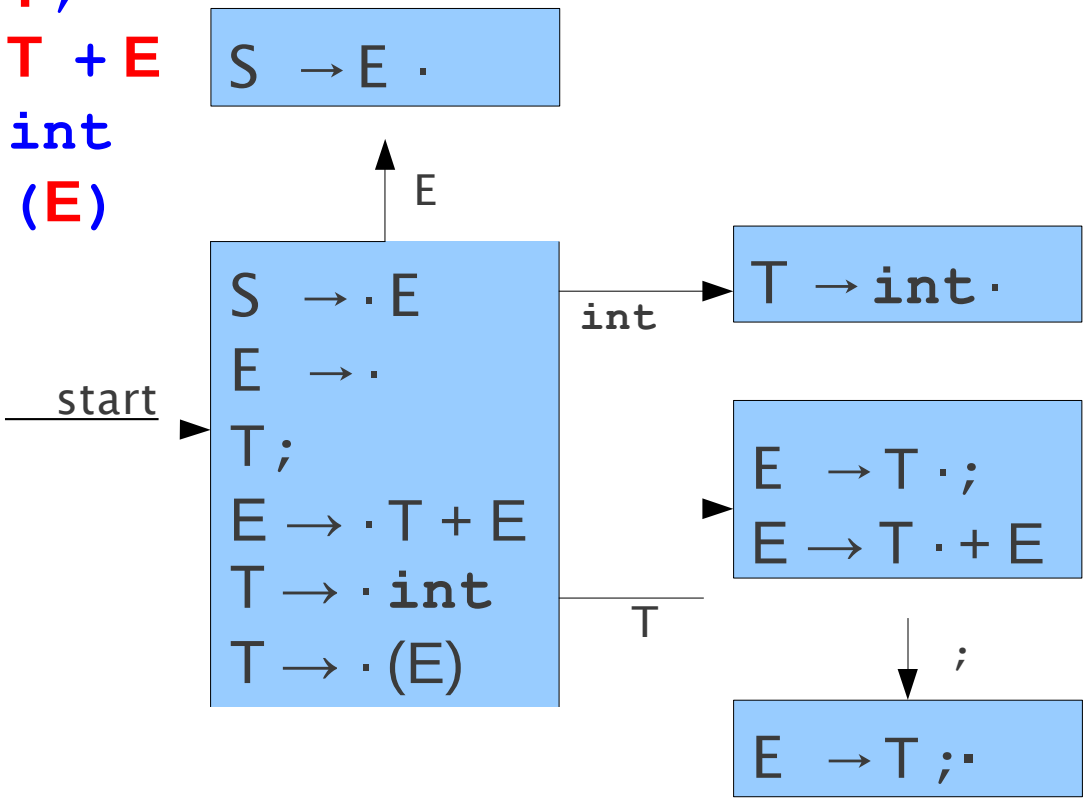
A Deterministic Automaton

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



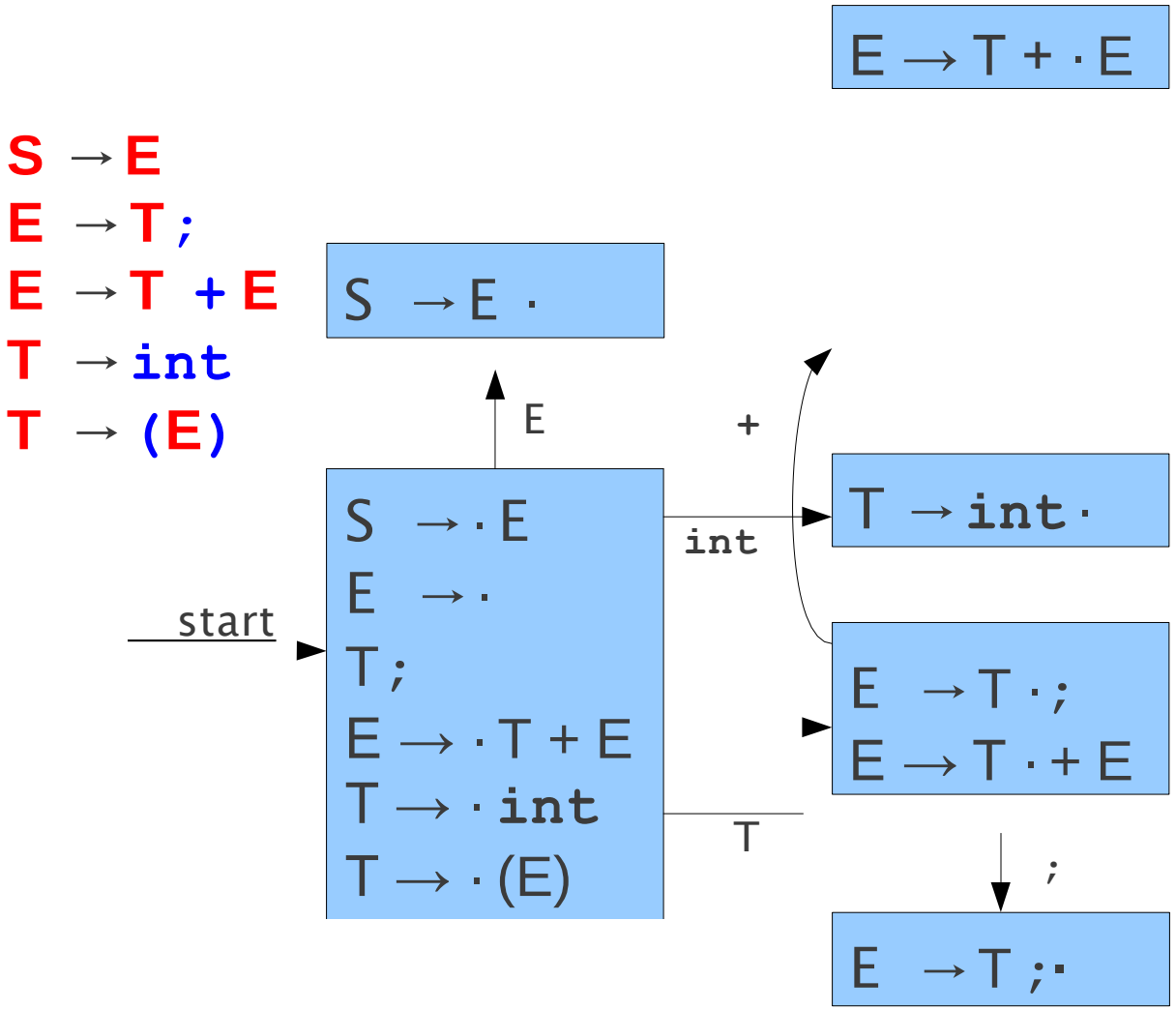
A Deterministic Automaton

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

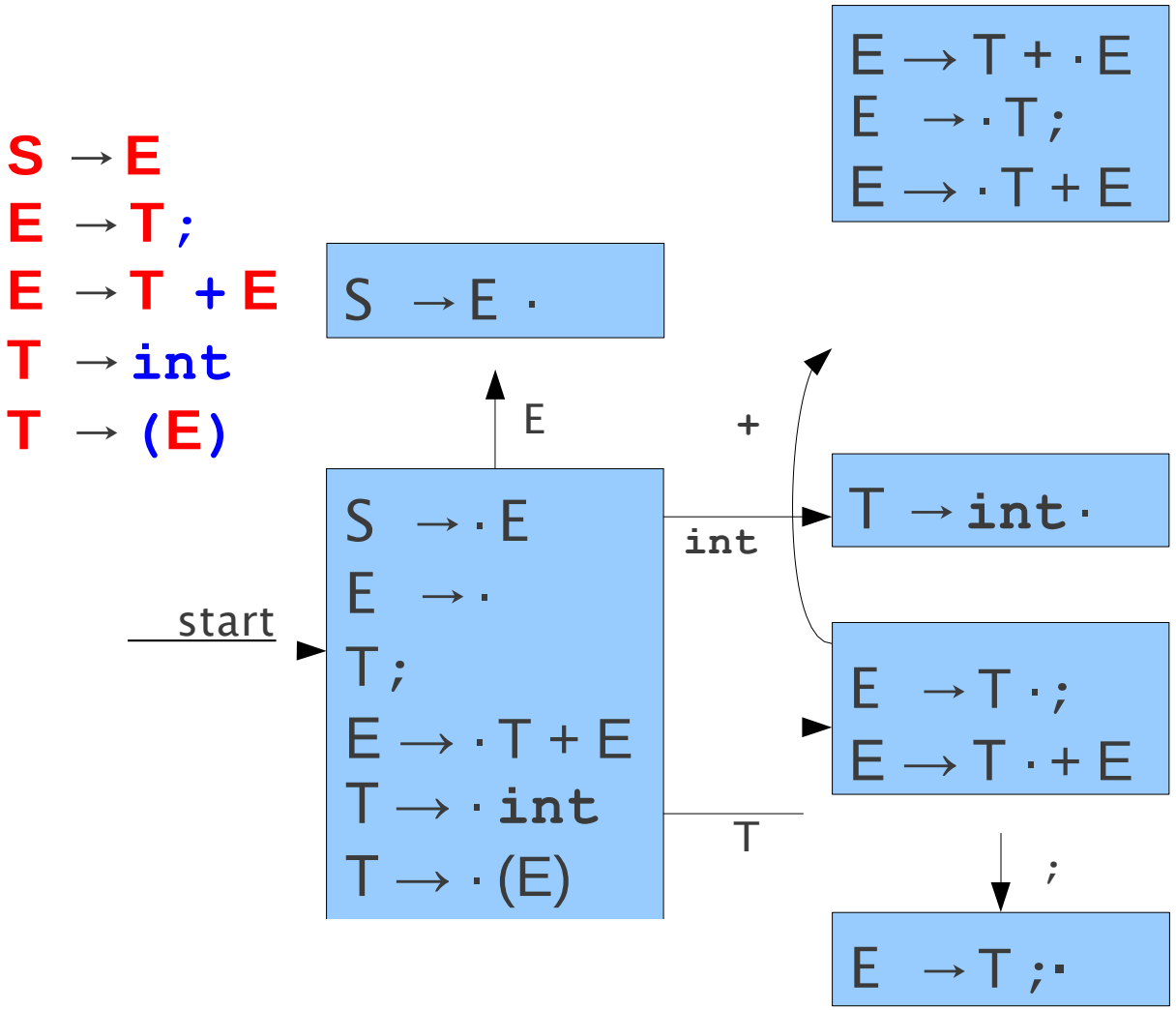


GEORGETOWN UNIVERSITY

A Deterministic Automaton

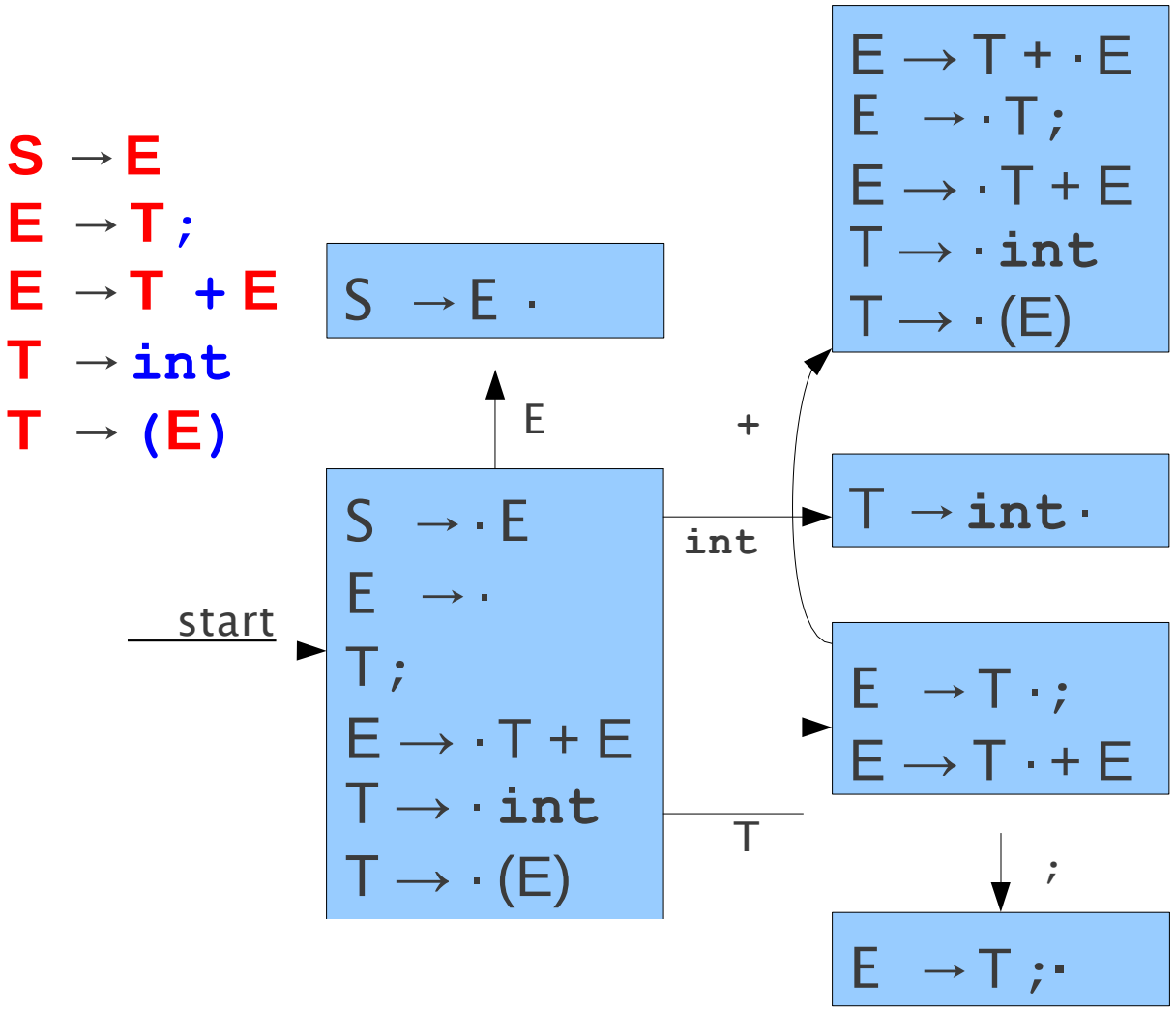


A Deterministic Automaton



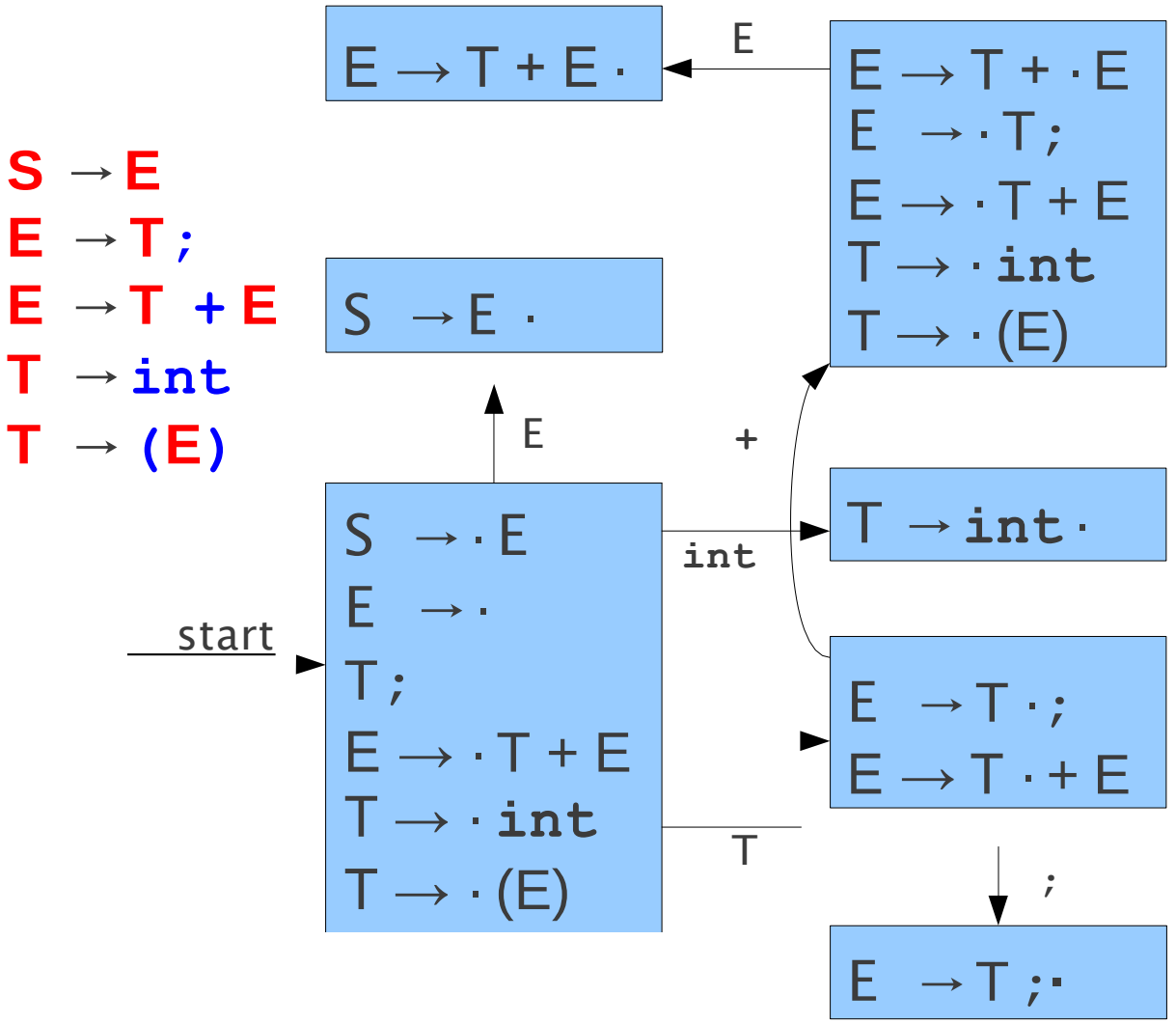
GEORGETOWN UNIVERSITY

A Deterministic Automaton

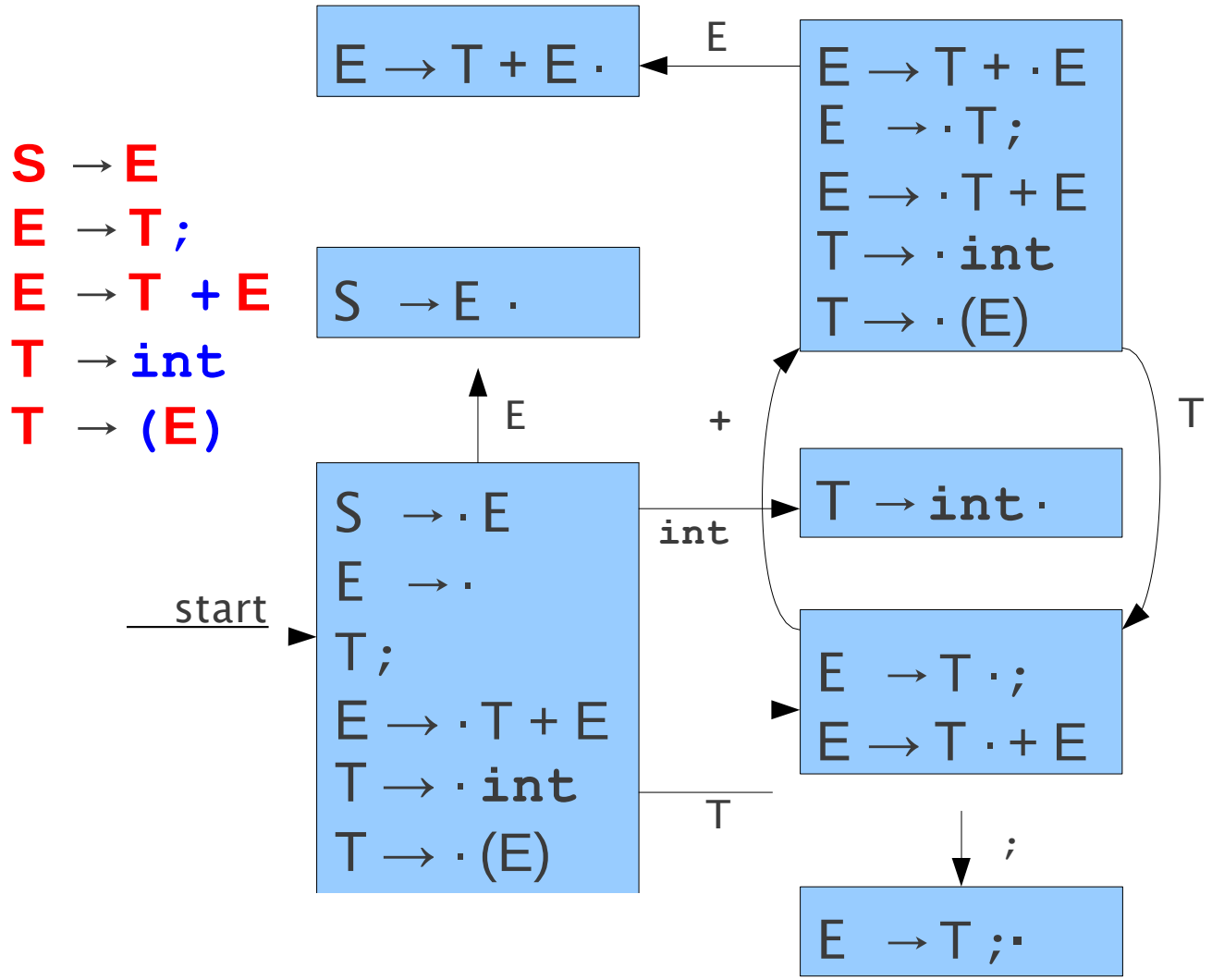


GEORGETOWN UNIVERSITY

A Deterministic Automaton

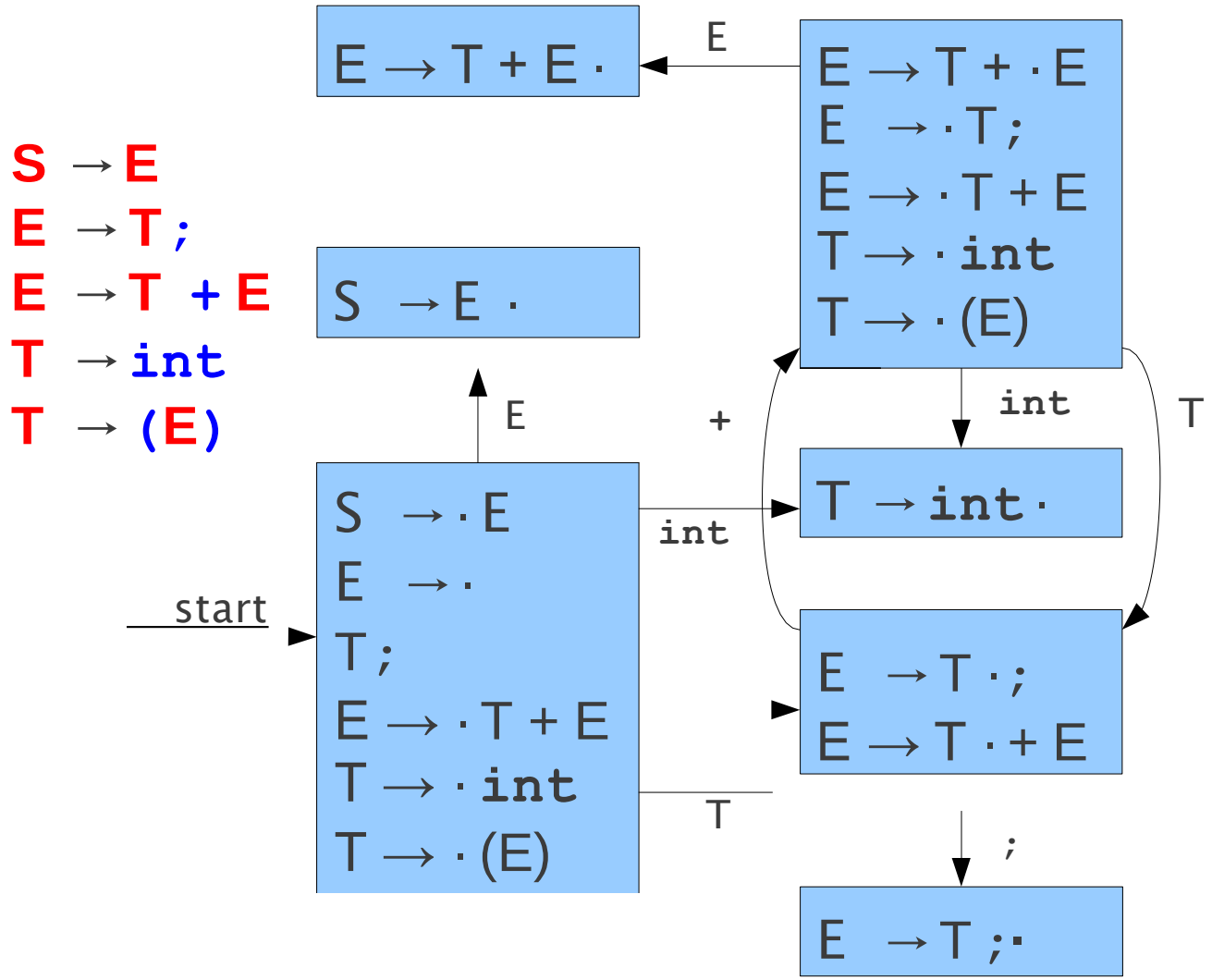


A Deterministic Automaton



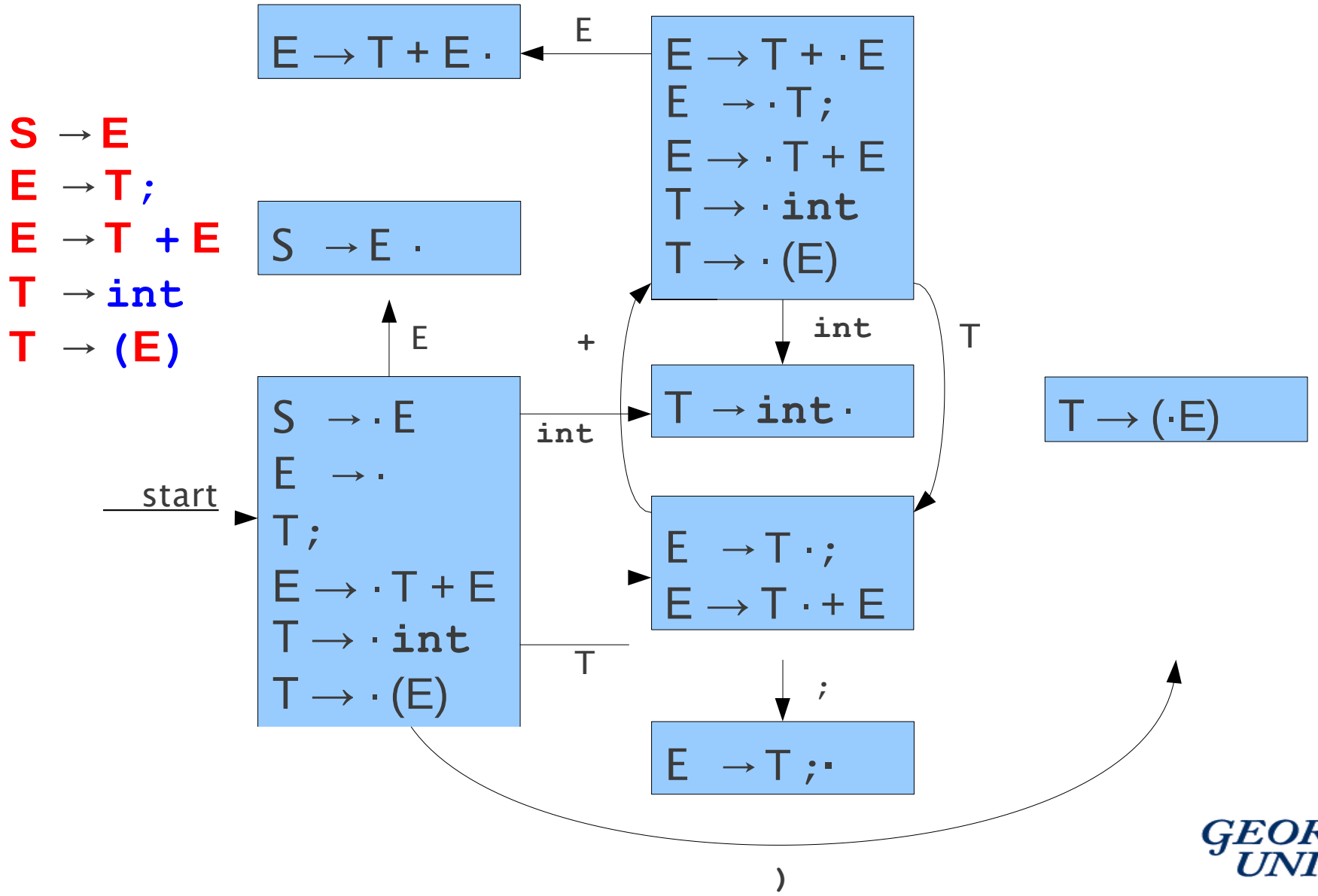
GEORGETOWN UNIVERSITY

A Deterministic Automaton



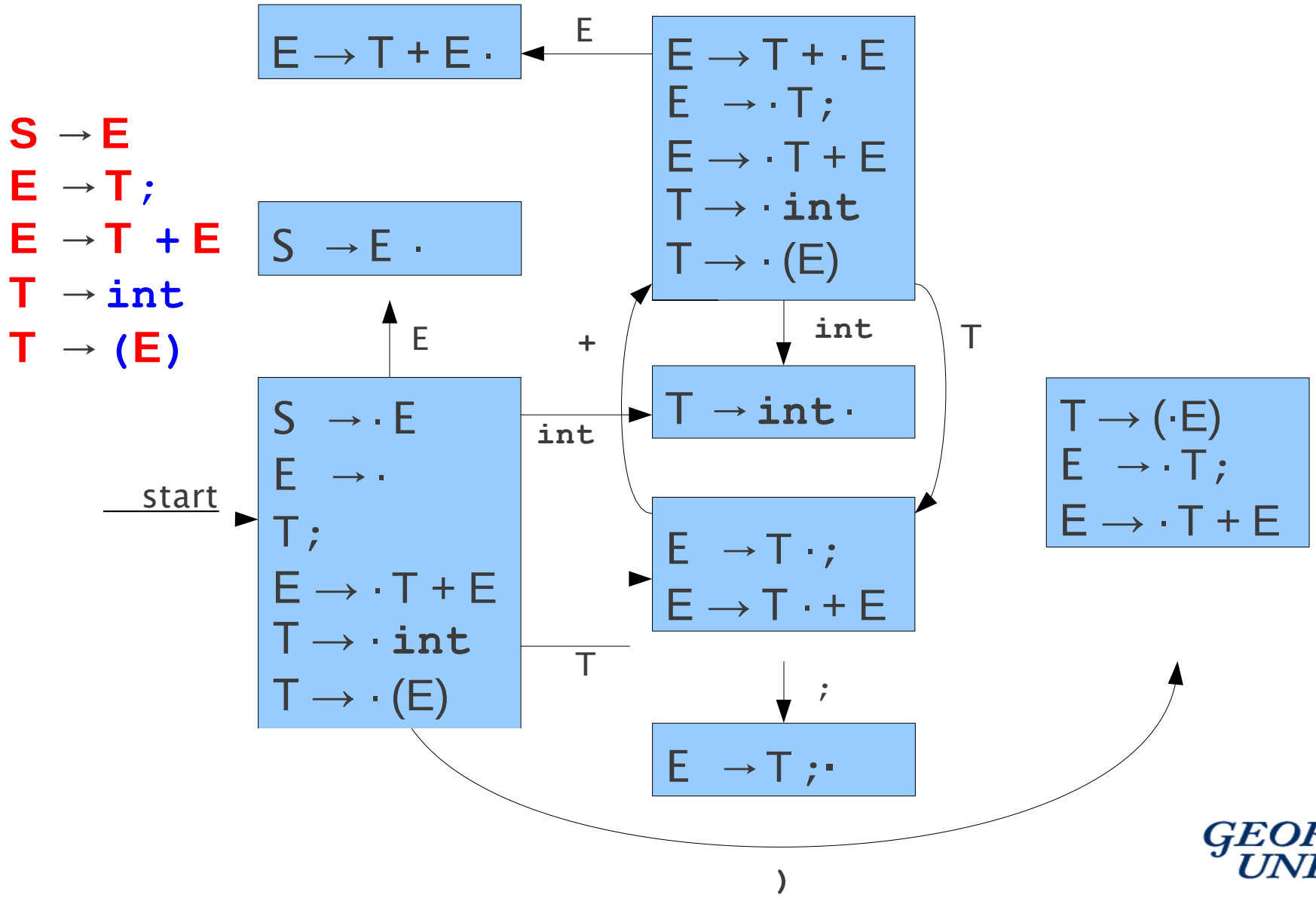
GEORGETOWN UNIVERSITY

A Deterministic Automaton



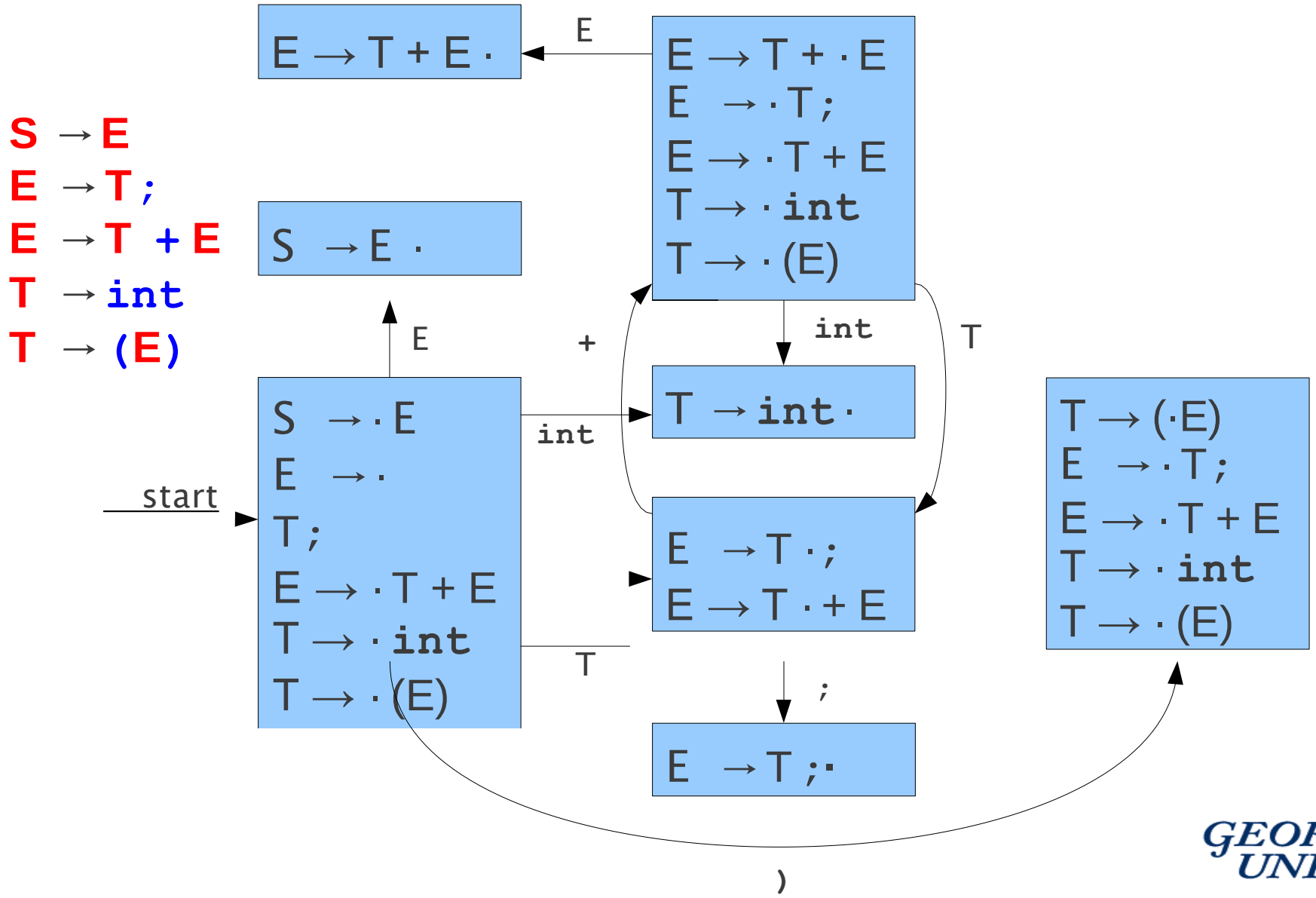
GEORGETOWN UNIVERSITY

A Deterministic Automaton



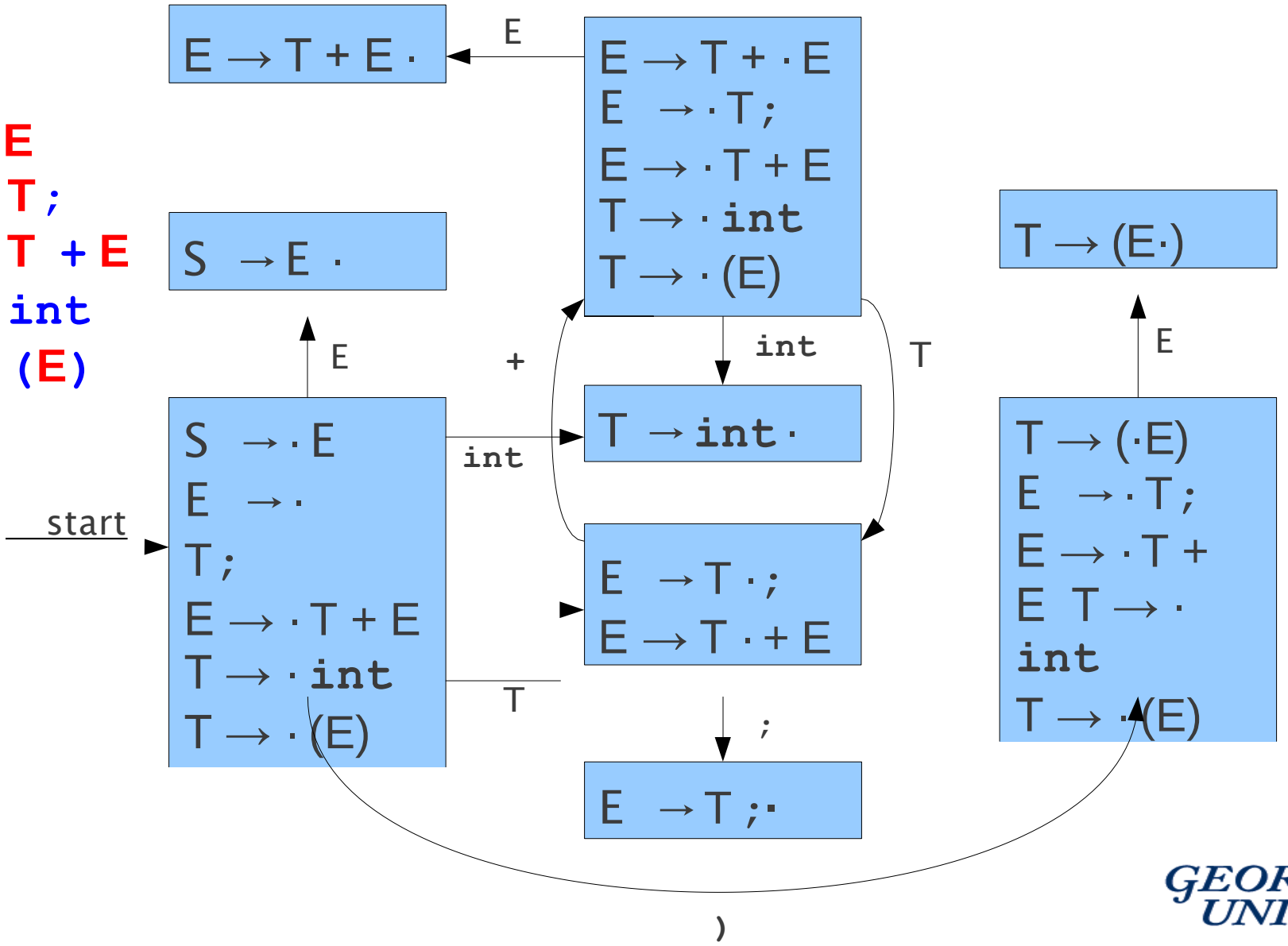
GEORGETOWN UNIVERSITY

A Deterministic Automaton



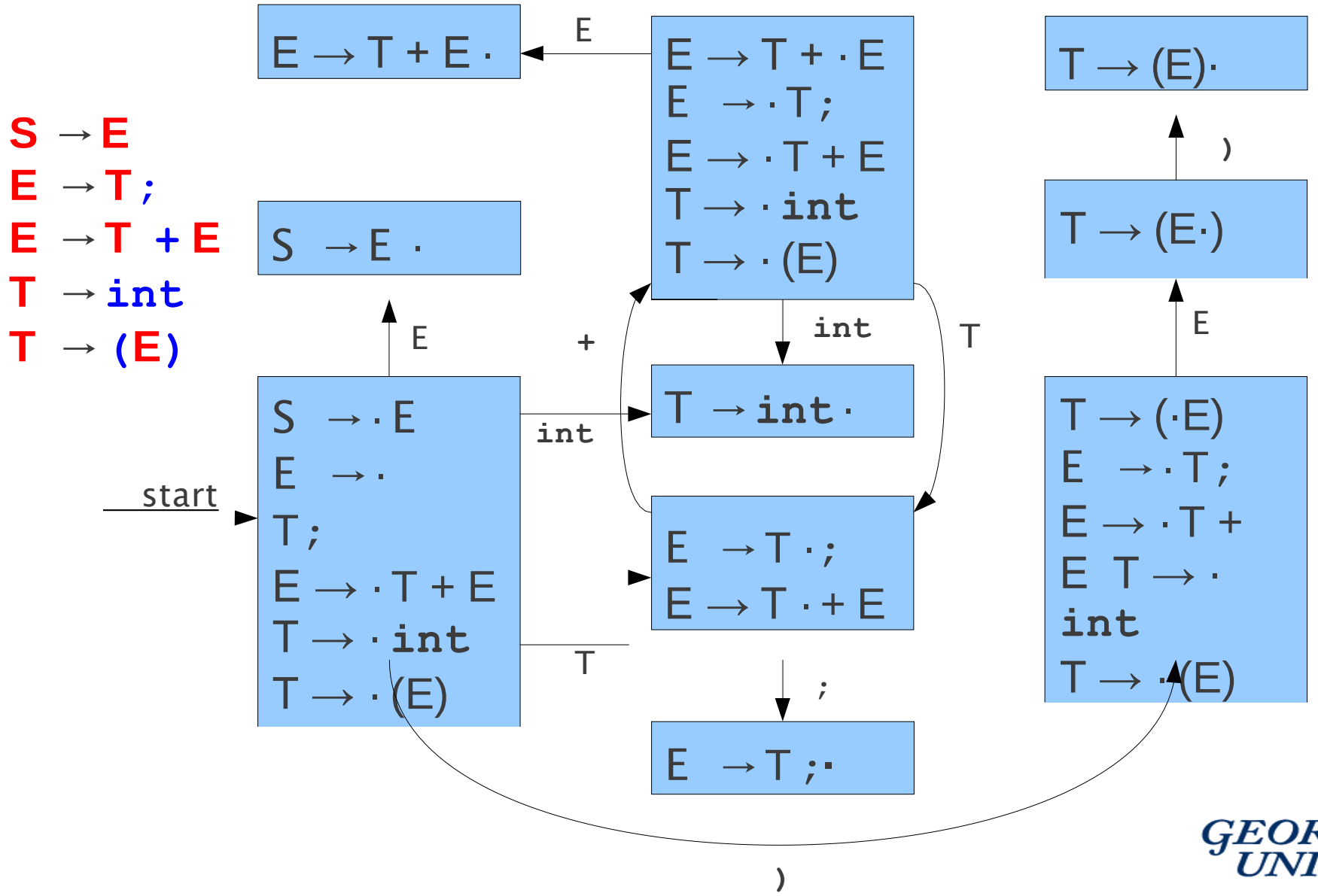
A Deterministic Automaton

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



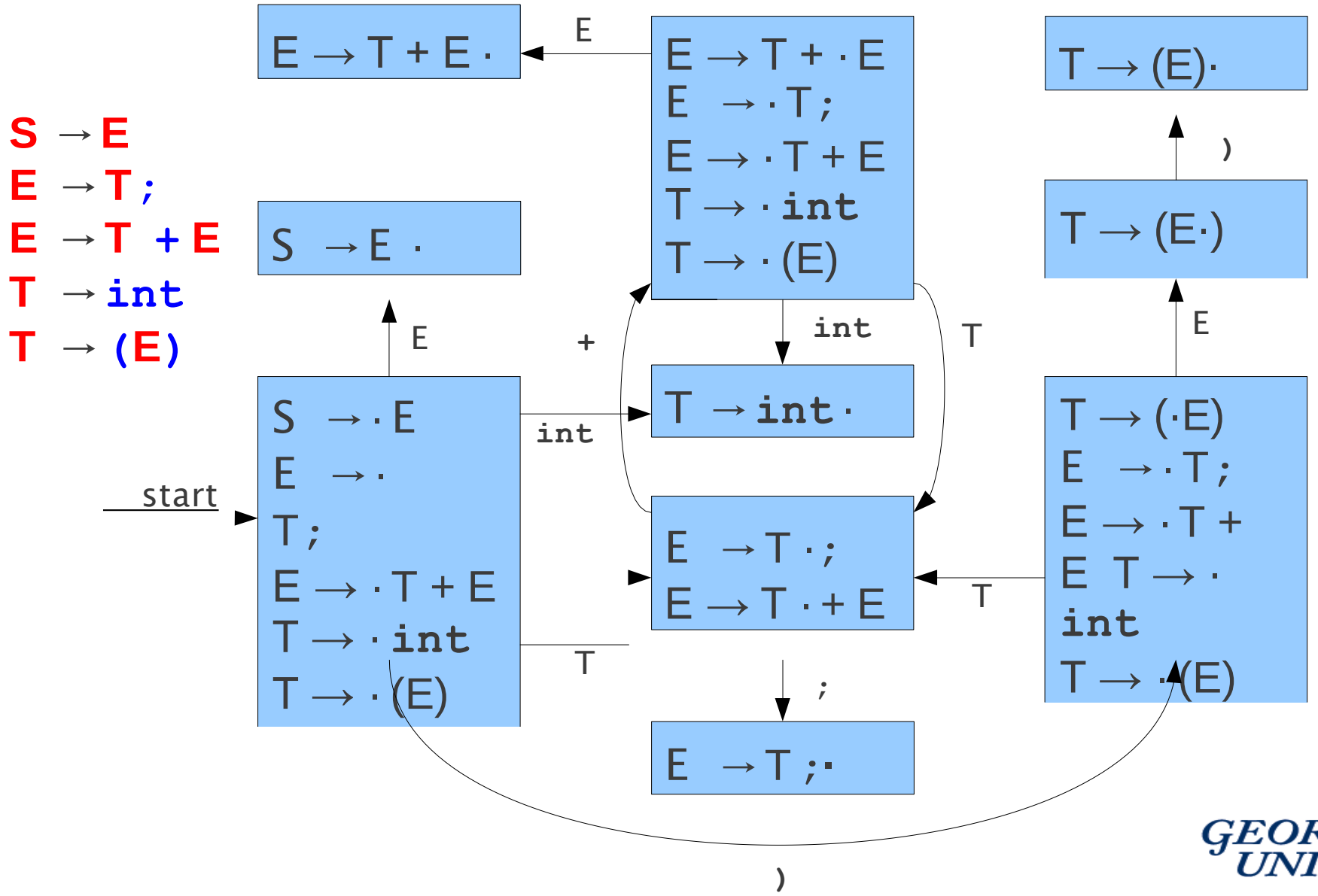
GEORGETOWN UNIVERSITY

A Deterministic Automaton



GEORGETOWN UNIVERSITY

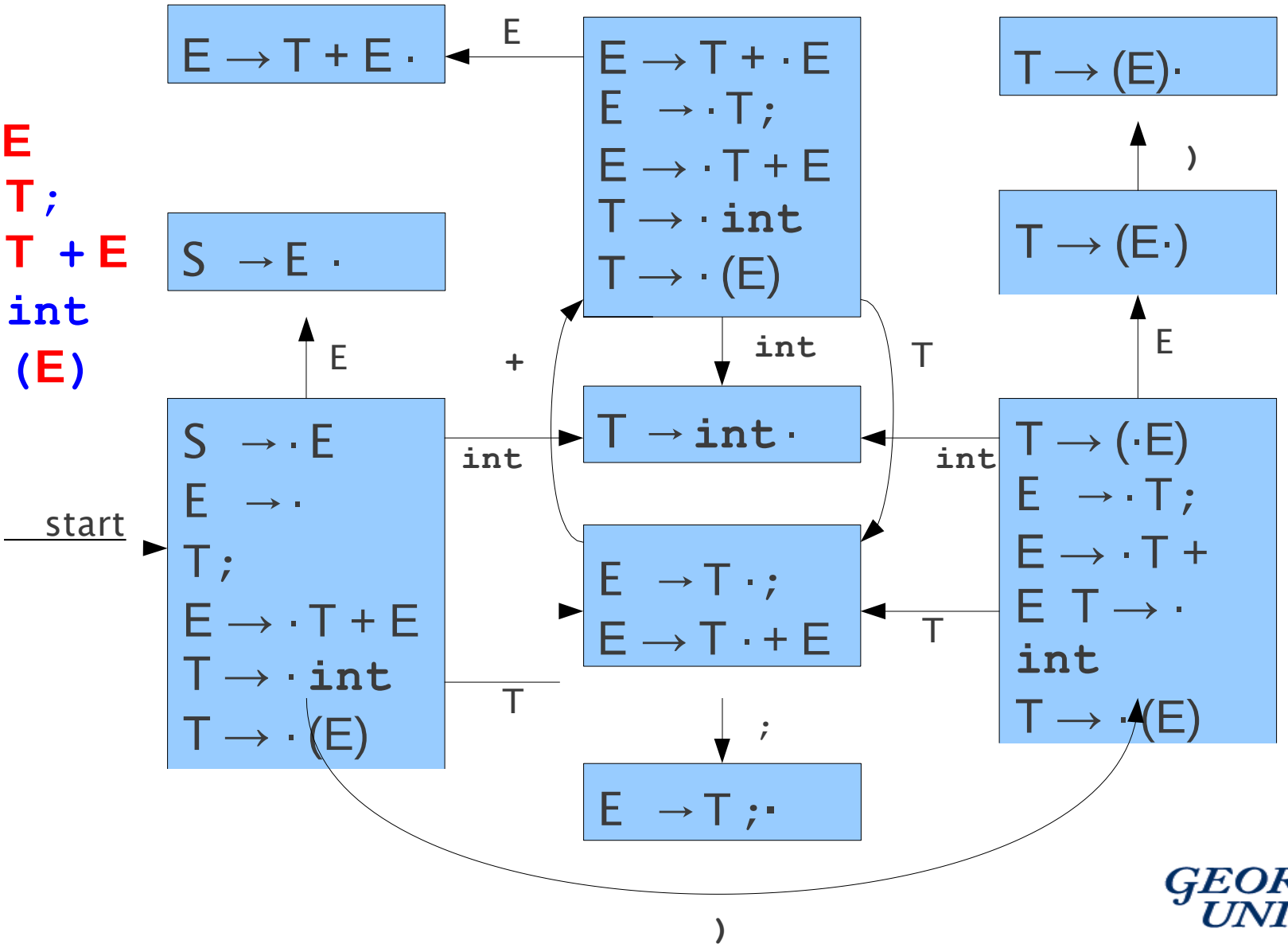
A Deterministic Automaton



GEORGETOWN UNIVERSITY

A Deterministic Automaton

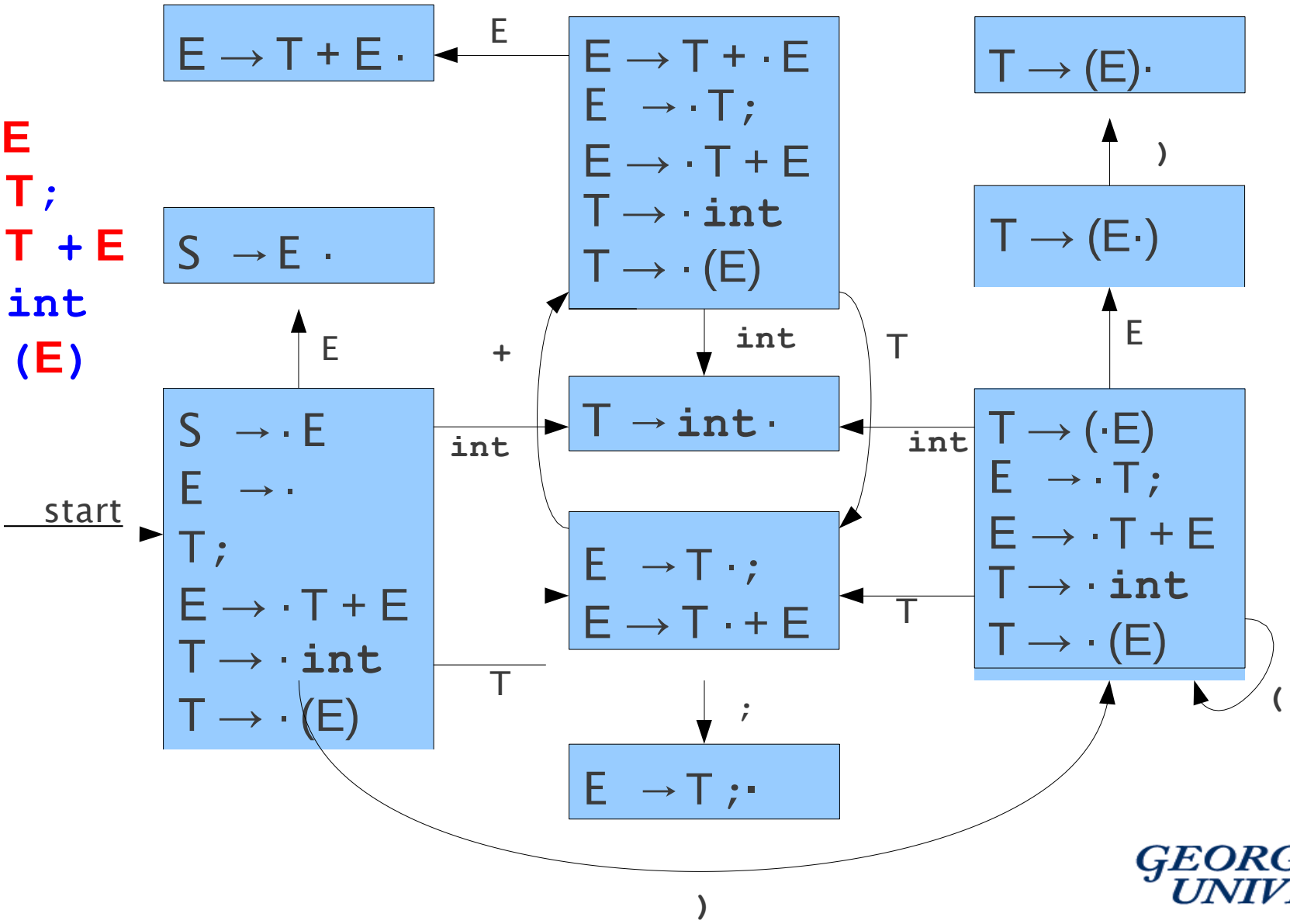
S → E
E → T ;
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

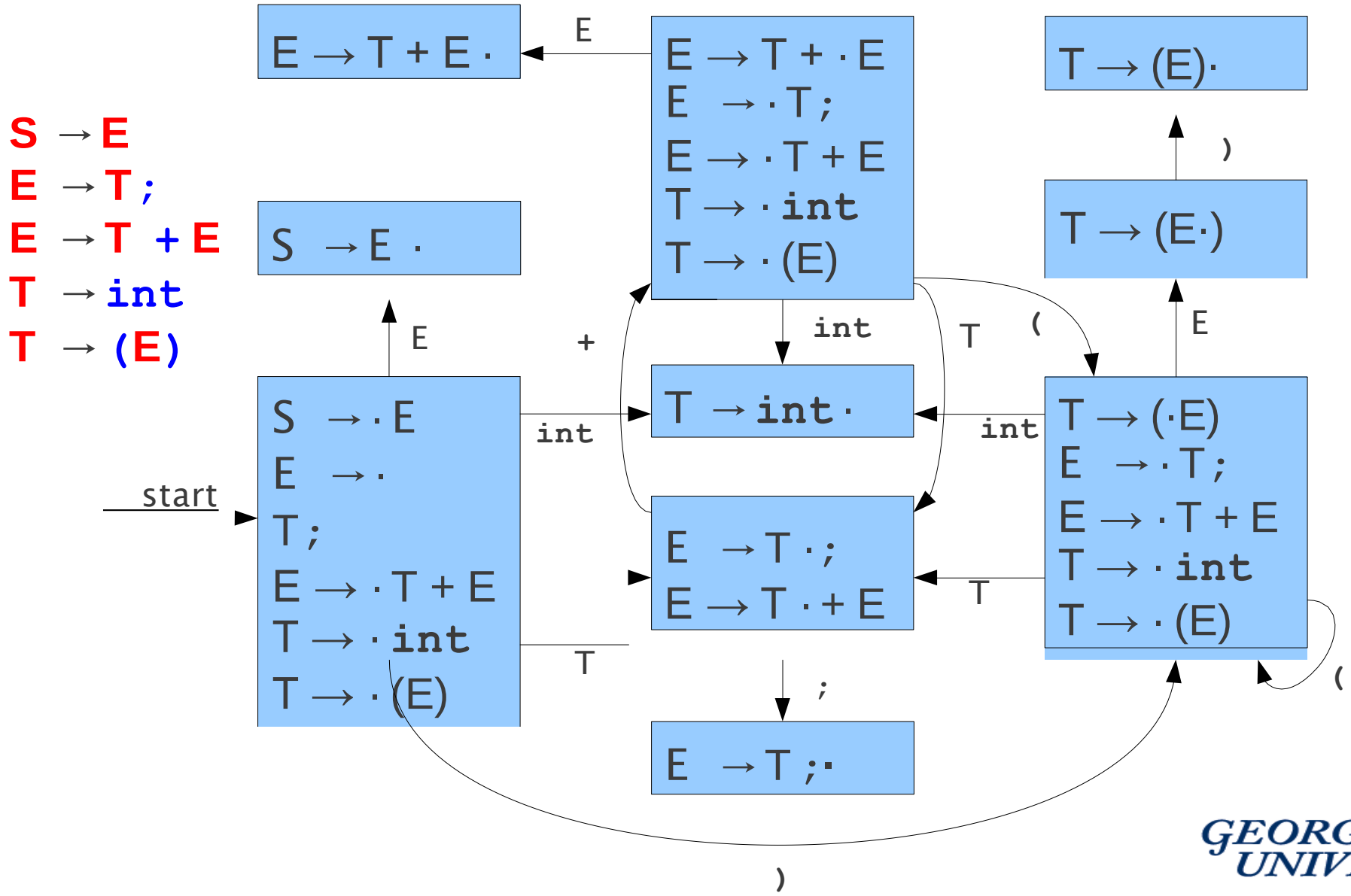
A Deterministic Automaton

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



GEORGETOWN UNIVERSITY

A Deterministic Automaton



GEORGETOWN UNIVERSITY

Constructing the Automaton

- Begin in a state containing **S** → · **A**, where **S** is the augmented start symbol.
- Compute the **closure** of the state:
  - If **A** → *a* · **B** *ω* is in the state, add **B** → · *γ* to the state for each production **B** → *γ*.
  - Yet another fixed-point iteration!
- Repeat until no new states are added:
  - If a state contains a production **A** → *a* · ***x*** *ω* for symbol ***x***, add a transition on ***x*** from that state to the state containing the closure of **A** → *a* ***x*** · *ω*
- This is equivalent to a subset construction on the NFA.

GEORGETOWN UNIVERSITY

Handle-Finding Automata

- Handling-finding automata can be very large.
- NFA has states proportional to the size of the grammar, so DFA can have size exponential in the size of the grammar.
  - There are grammars that can exhibit this worst-case.
- Automata are almost always generated by tools like **`bison`**.

Finding Handles

- Where do we look for handles?
  - **At the top of the stack.**
- How do we search for handles?
  - **Build a handle-finding automaton.**
- How do we recognize handles?
  - Once we've found a possible handle, how do we confirm that it's correct?

Question Three:

How do we recognize handles?

GEORGETOWN UNIVERSITY

Handle Recognition

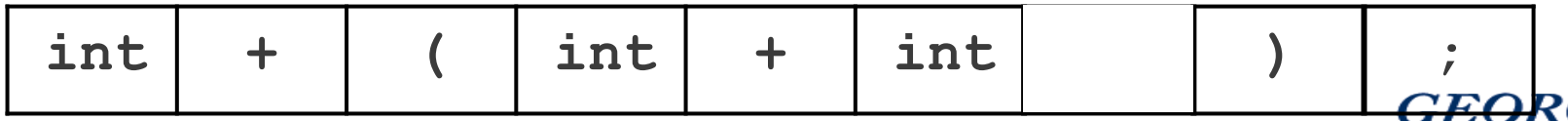
- Our automaton will tell us all places where a handle might be.
- However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.
- We'll thus use **predictive bottom-up parsing**:
  - Have a deterministic procedure for guessing where handles are.
- There are many predictive algorithms, each of which recognize different grammars.

Our First Algorithm: **LR(0)**

- Bottom-up predictive parsing with:
  - **L**: **L**eft-to-right scan of the input.
  - **R**: **R**ightmost derivation.
  - (**0**): Zero tokens of lookahead.
- Use the handle-finding automaton, without any lookahead, to predict where handles are.

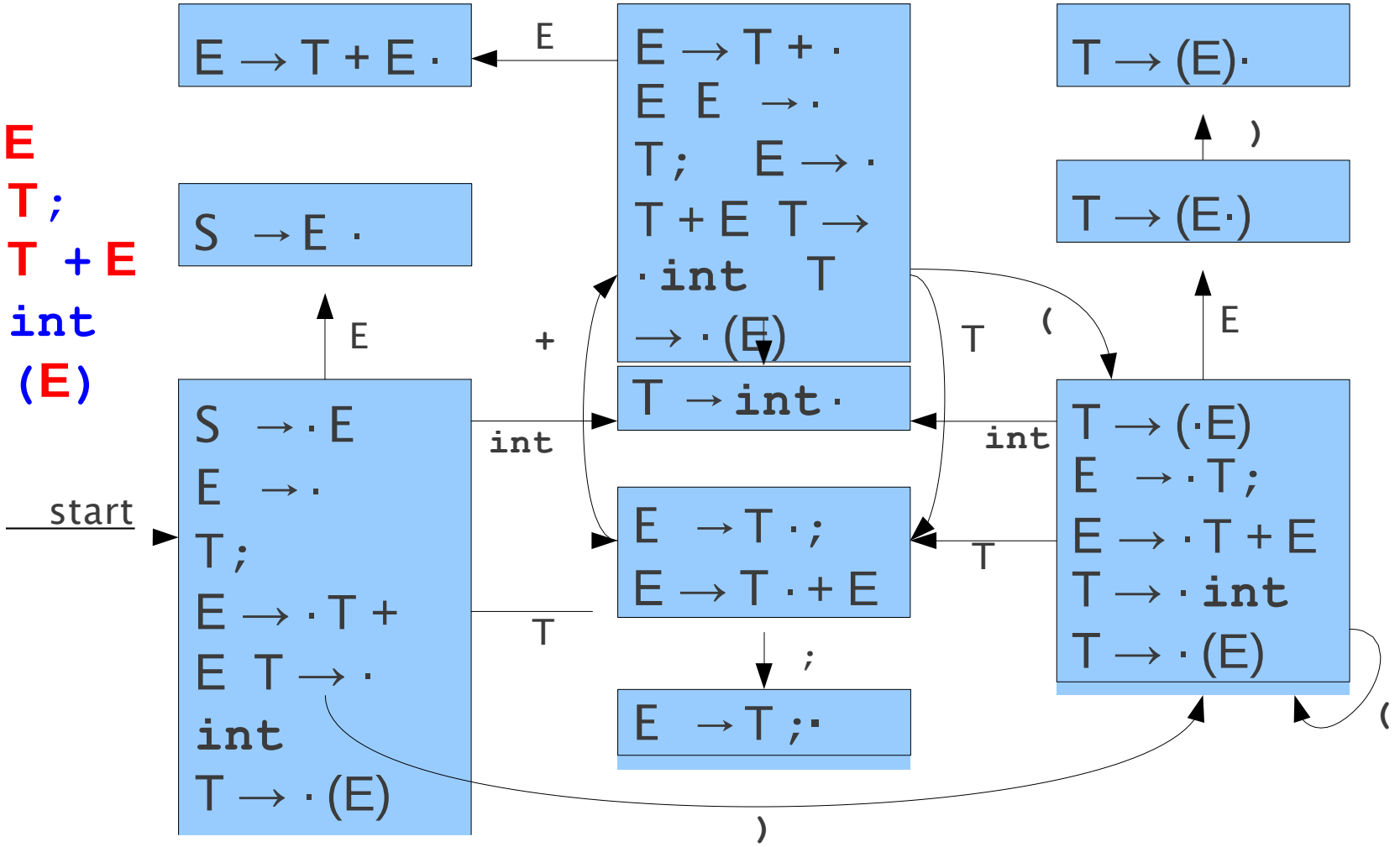
LR(0) Parsing

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



LR(0) Parsing

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

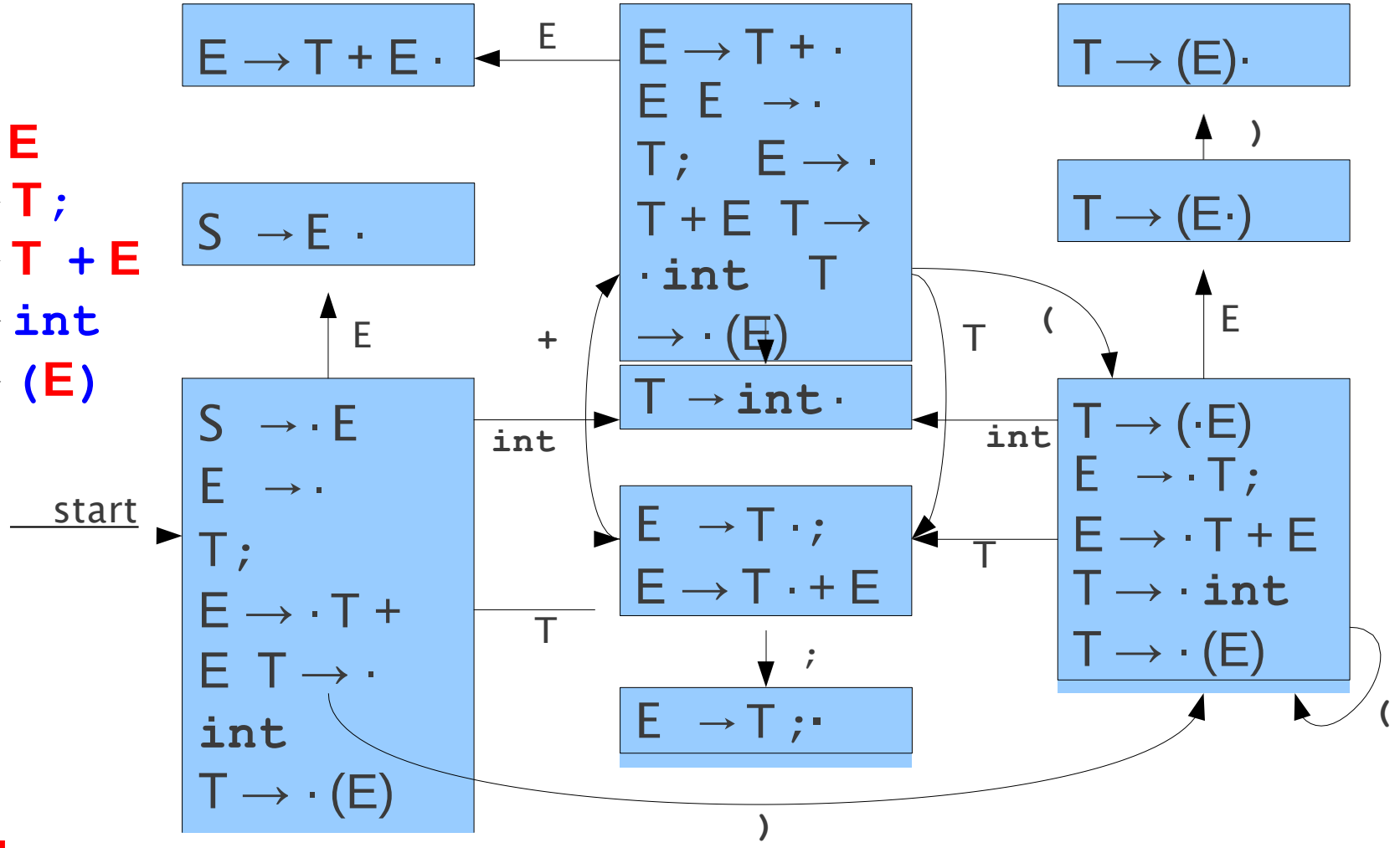


int	+	(	int	+	int		)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

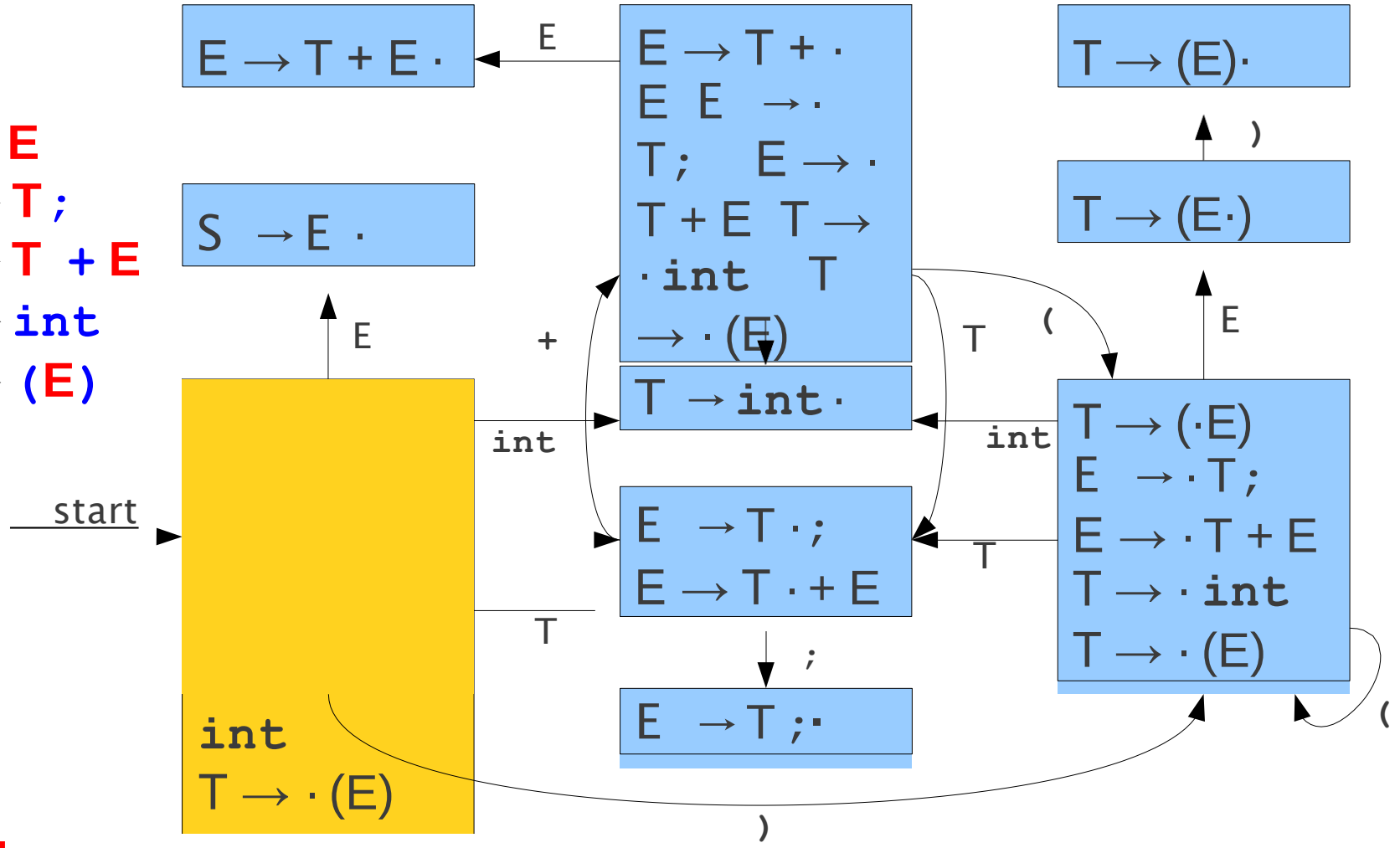
S → E
E → T;
E → T + E
T → int
T → (E)



int	+	(	int	+	int		)	;

LR(0) Parsing

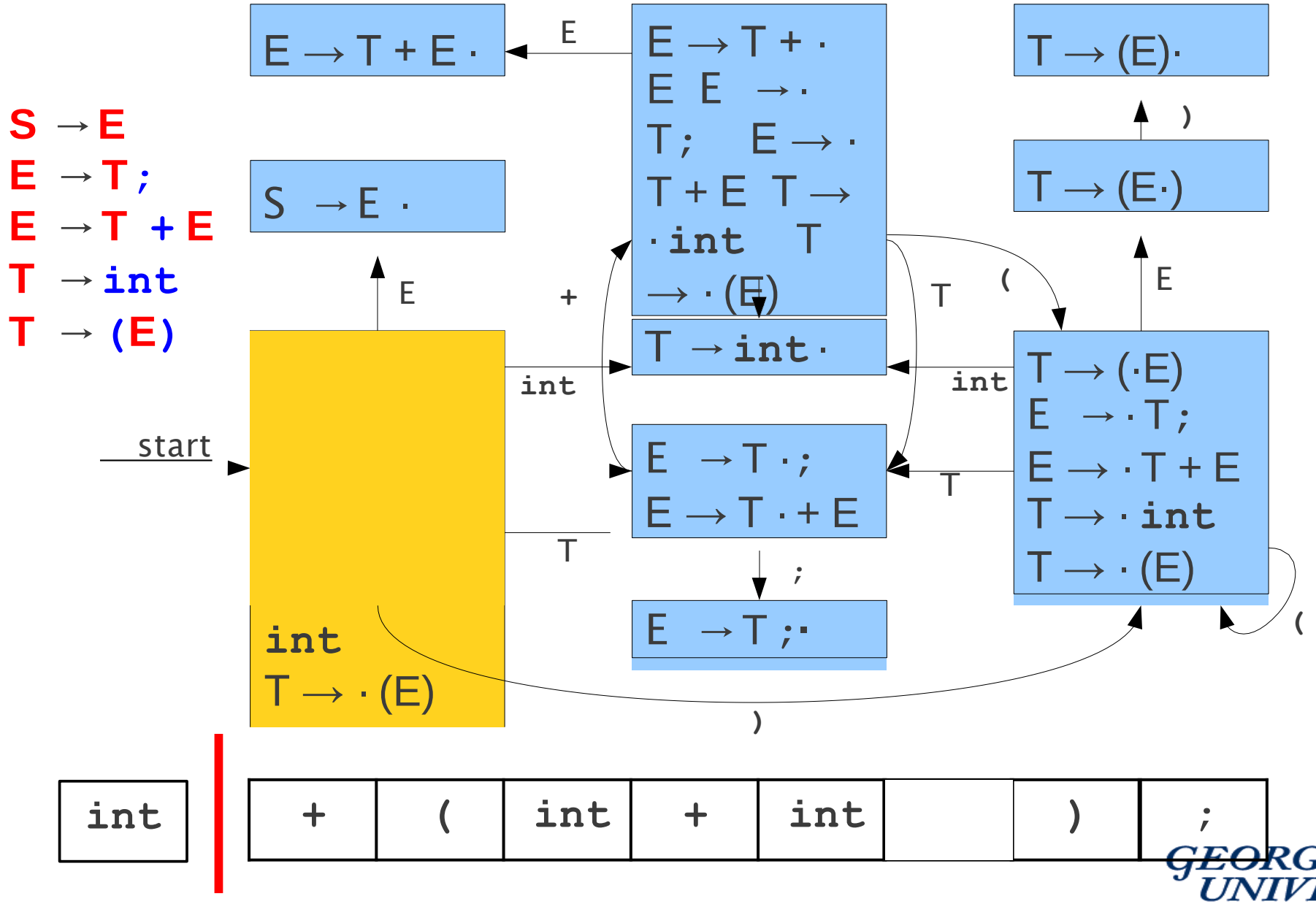
S → E
E → T;
E → T + E
T → int
T → (E)



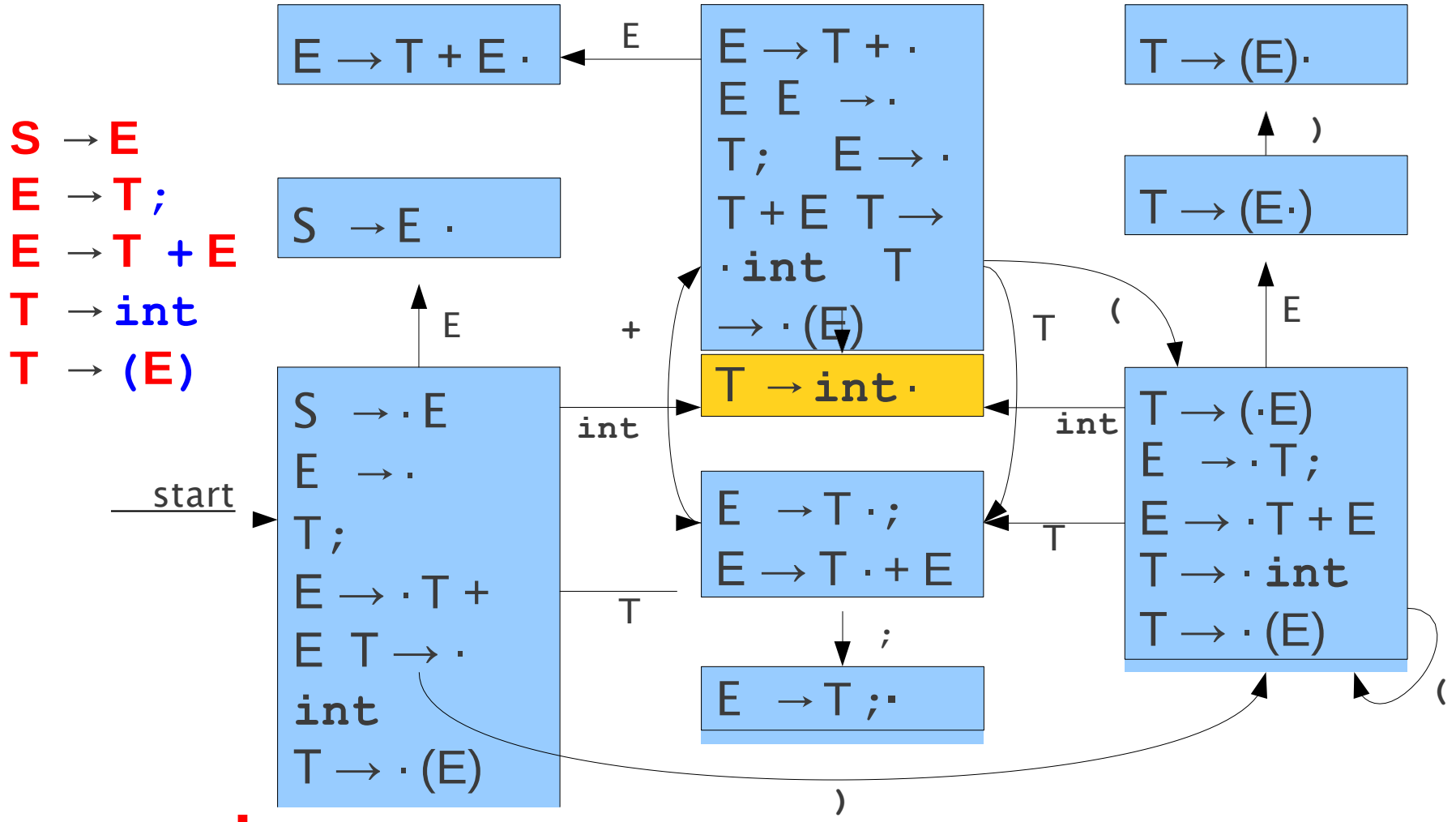
int	+	(	int	+	int		)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

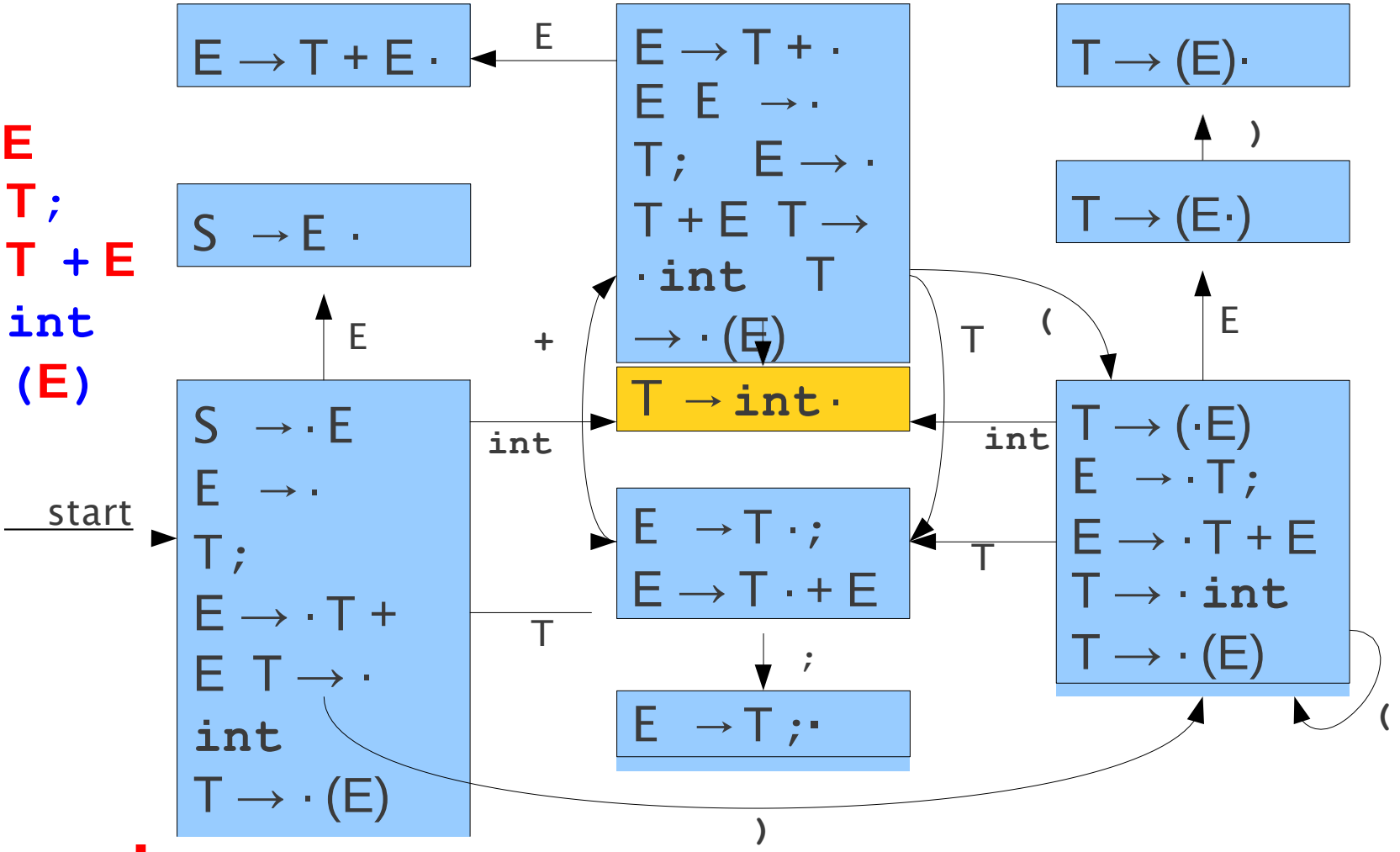


LR(0) Parsing



LR(0) Parsing

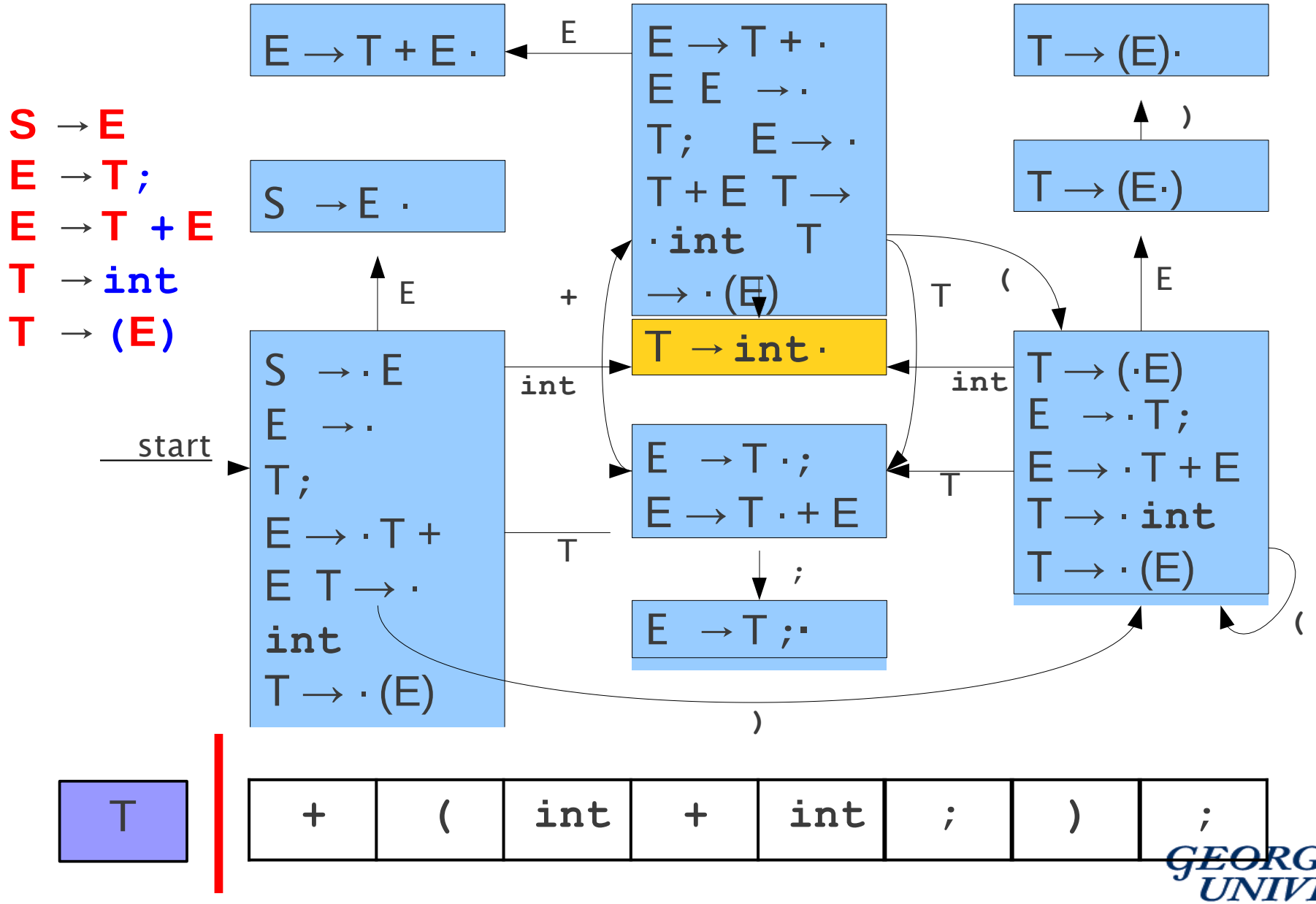
S → E
E → T;
E → T + E
T → int
T → (E)



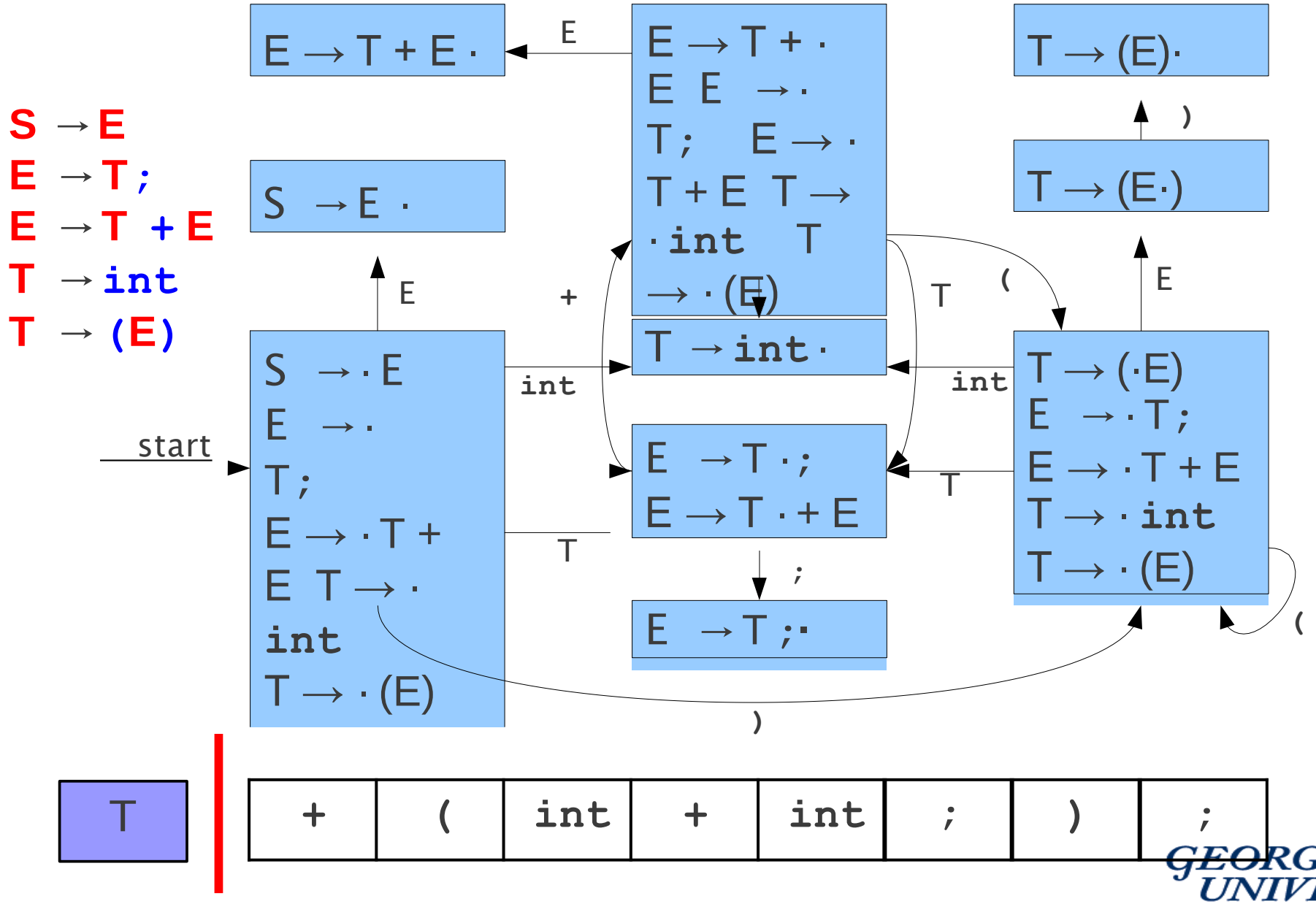
+	(	int	+	int	;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

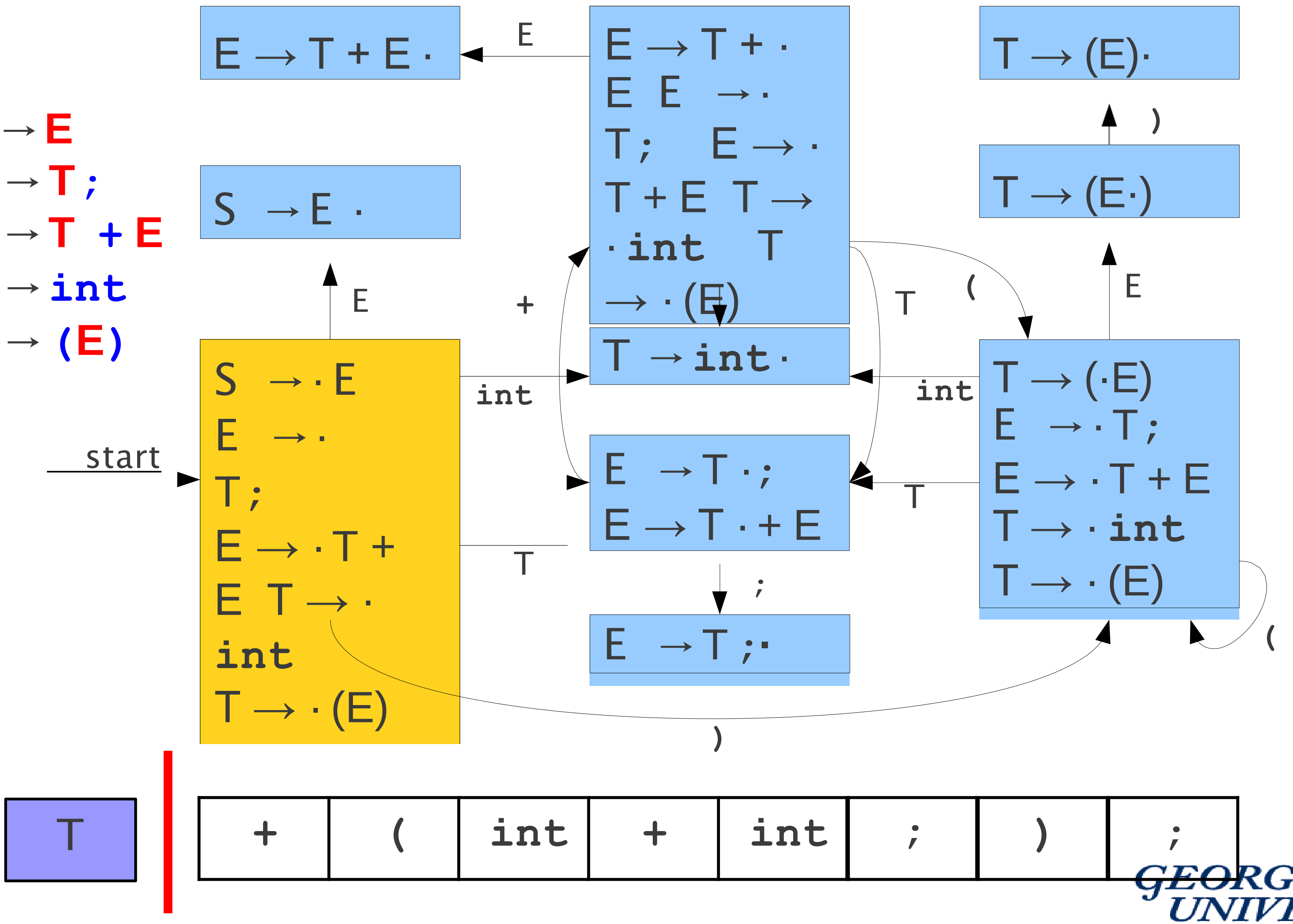


LR(0) Parsing

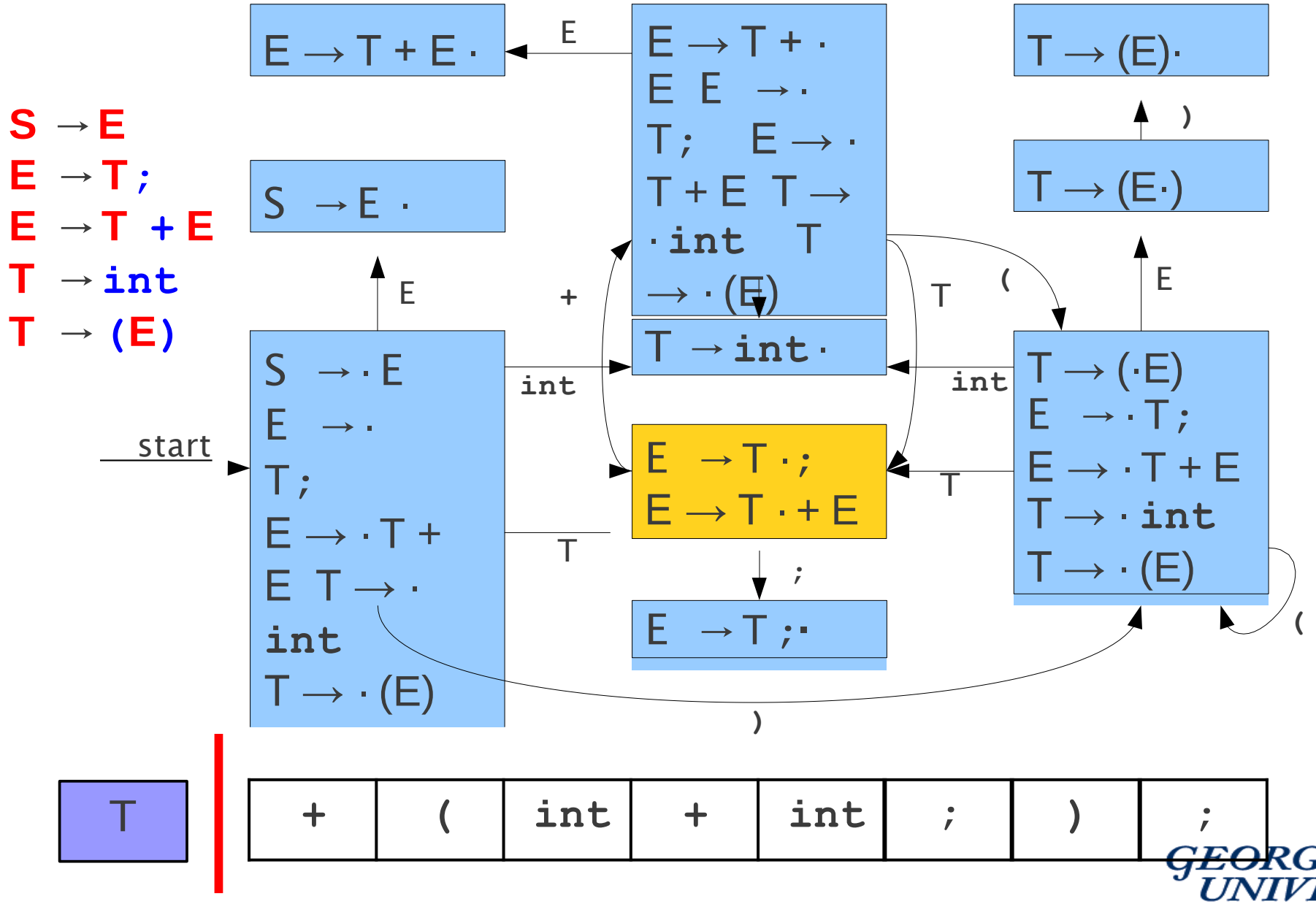


LR(0) Parsing

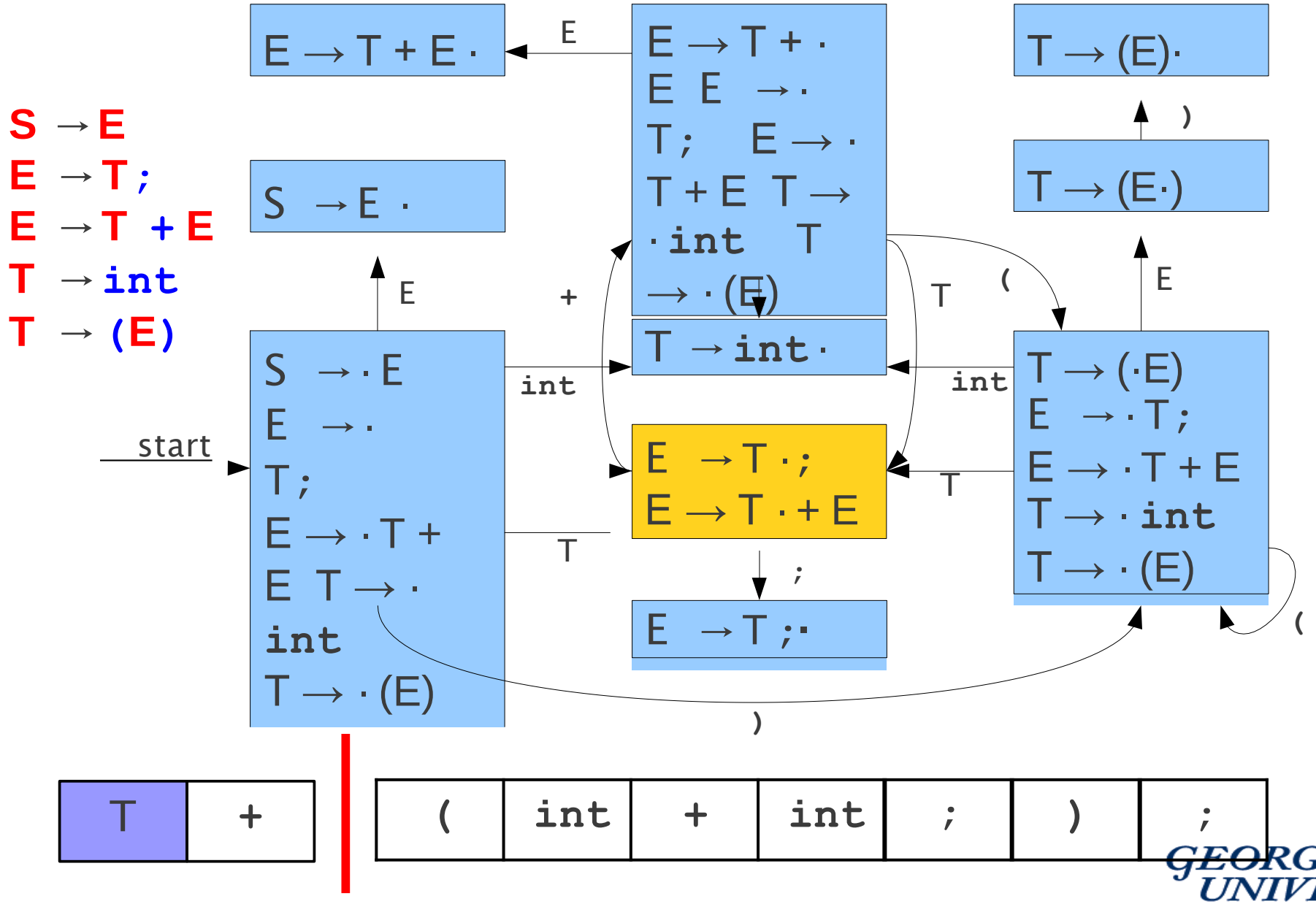
S → E
E → T;
E → T + E
T → int
T → (E)



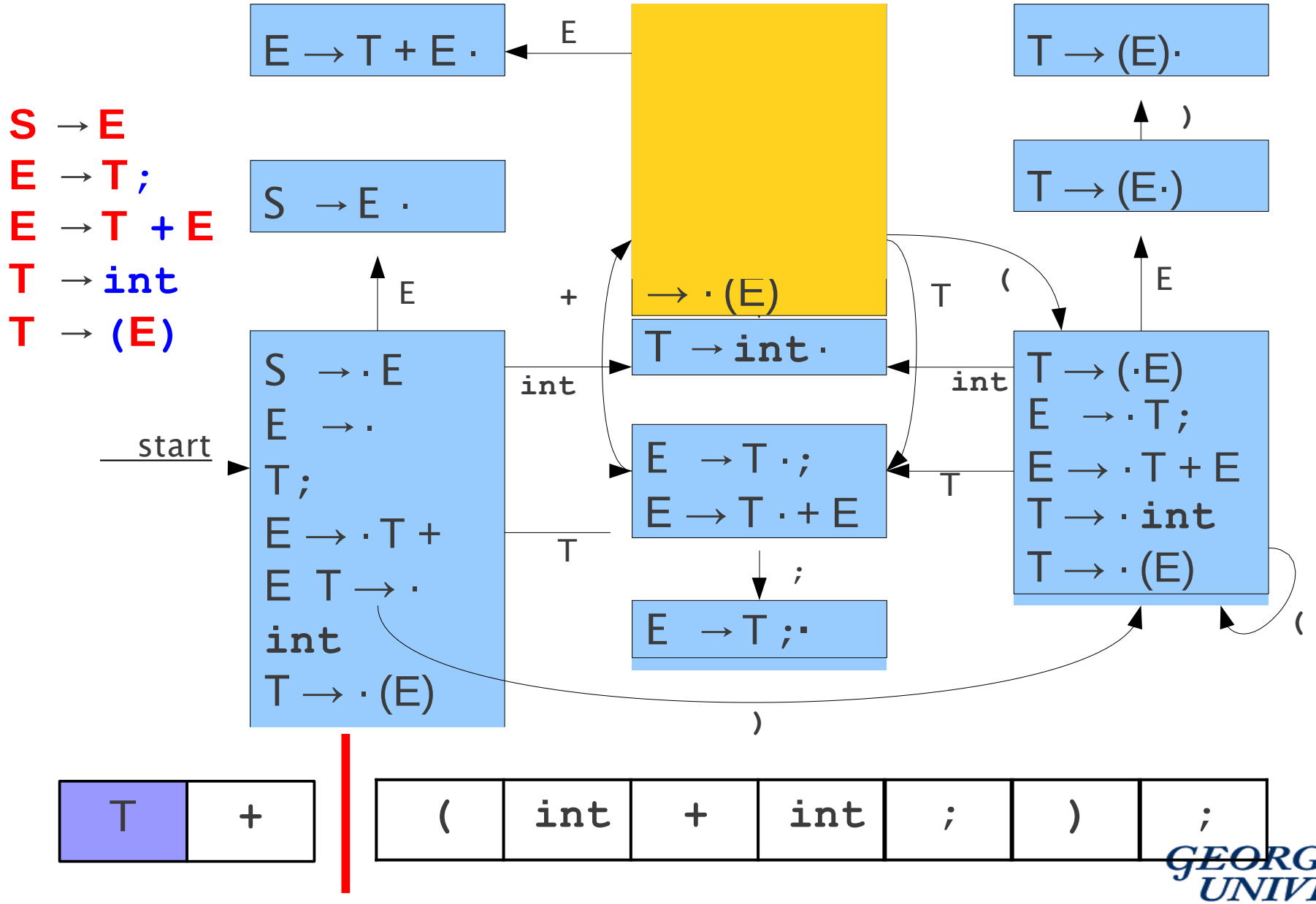
LR(0) Parsing



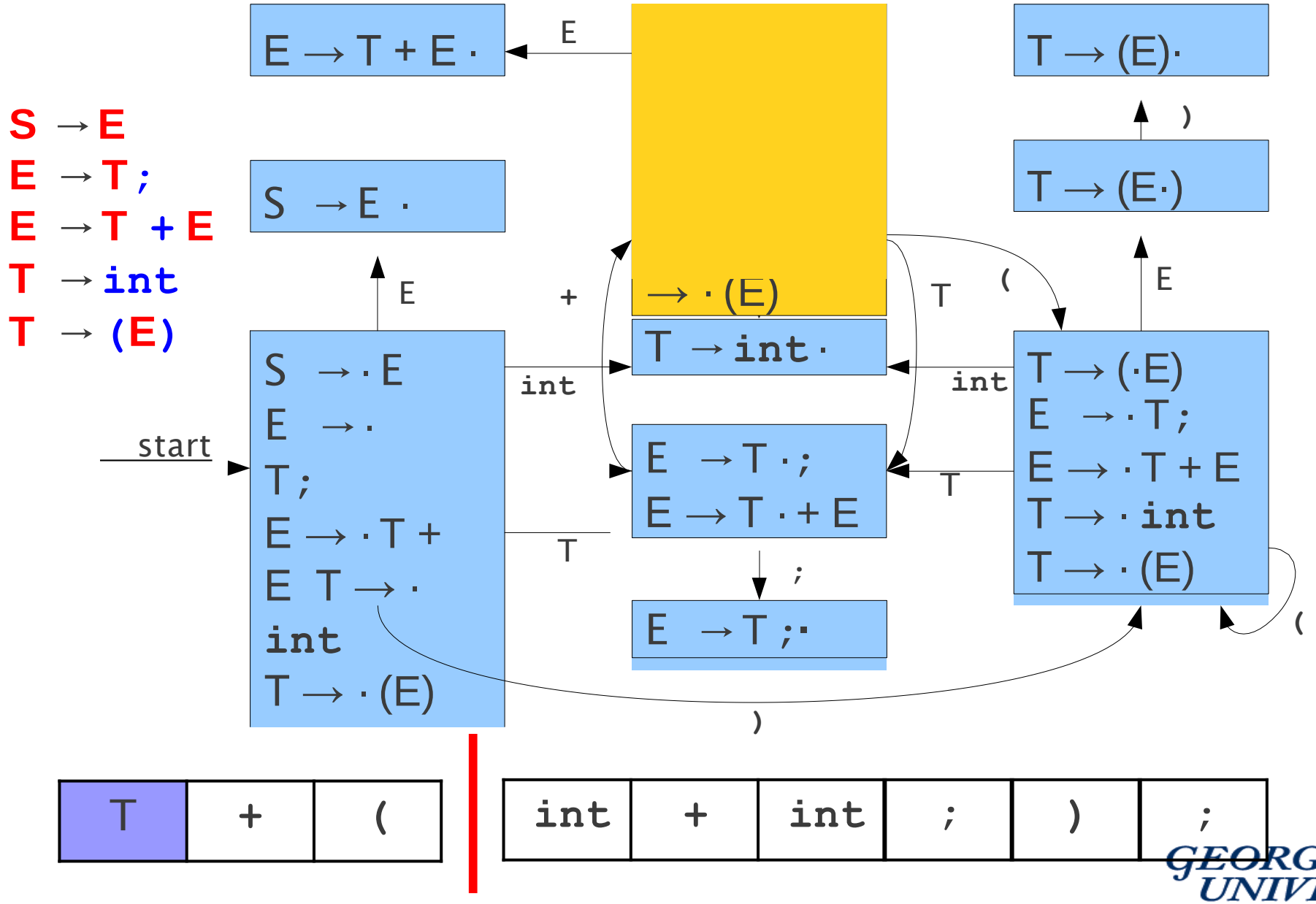
LR(0) Parsing



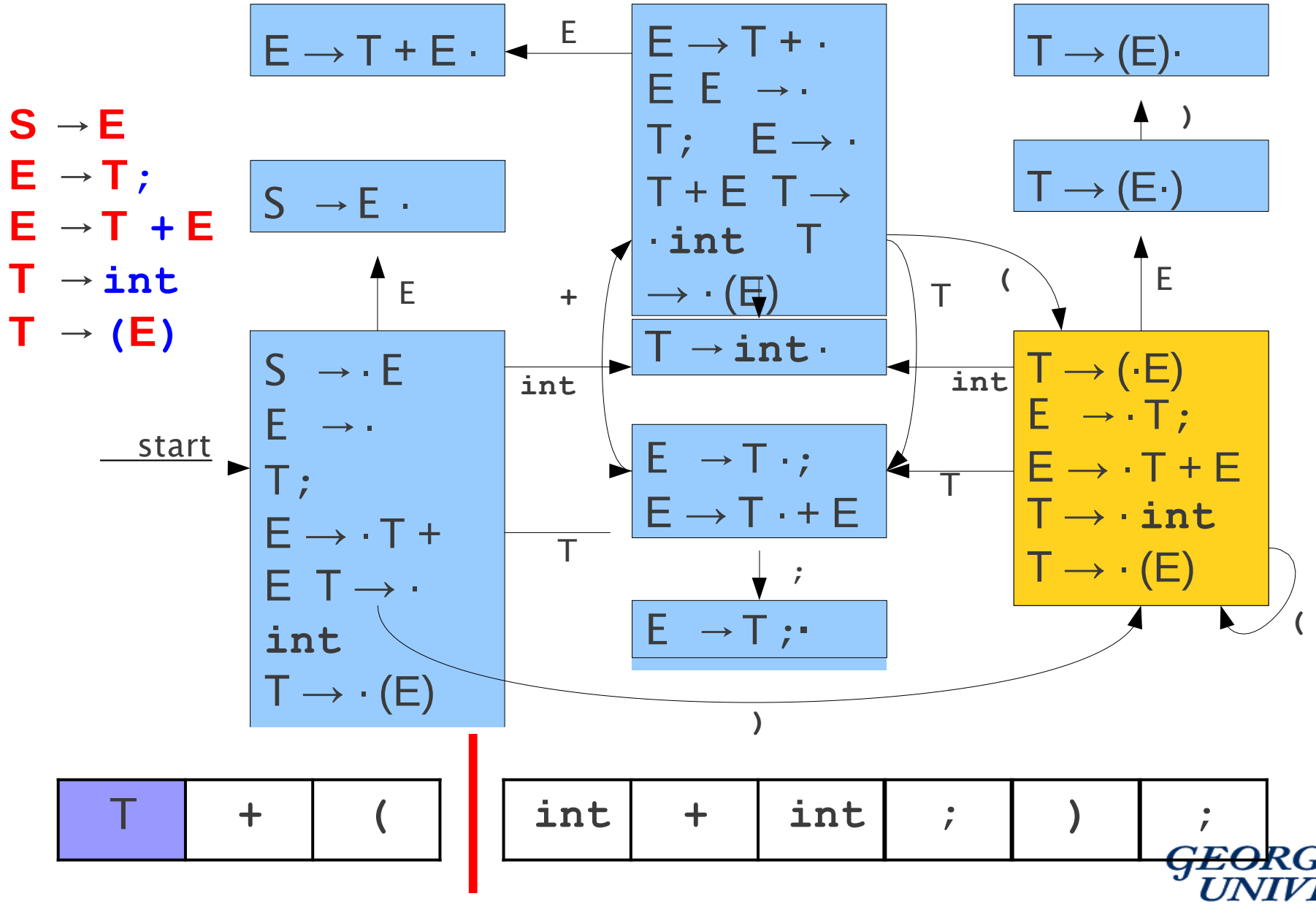
LR(0) Parsing



LR(0) Parsing

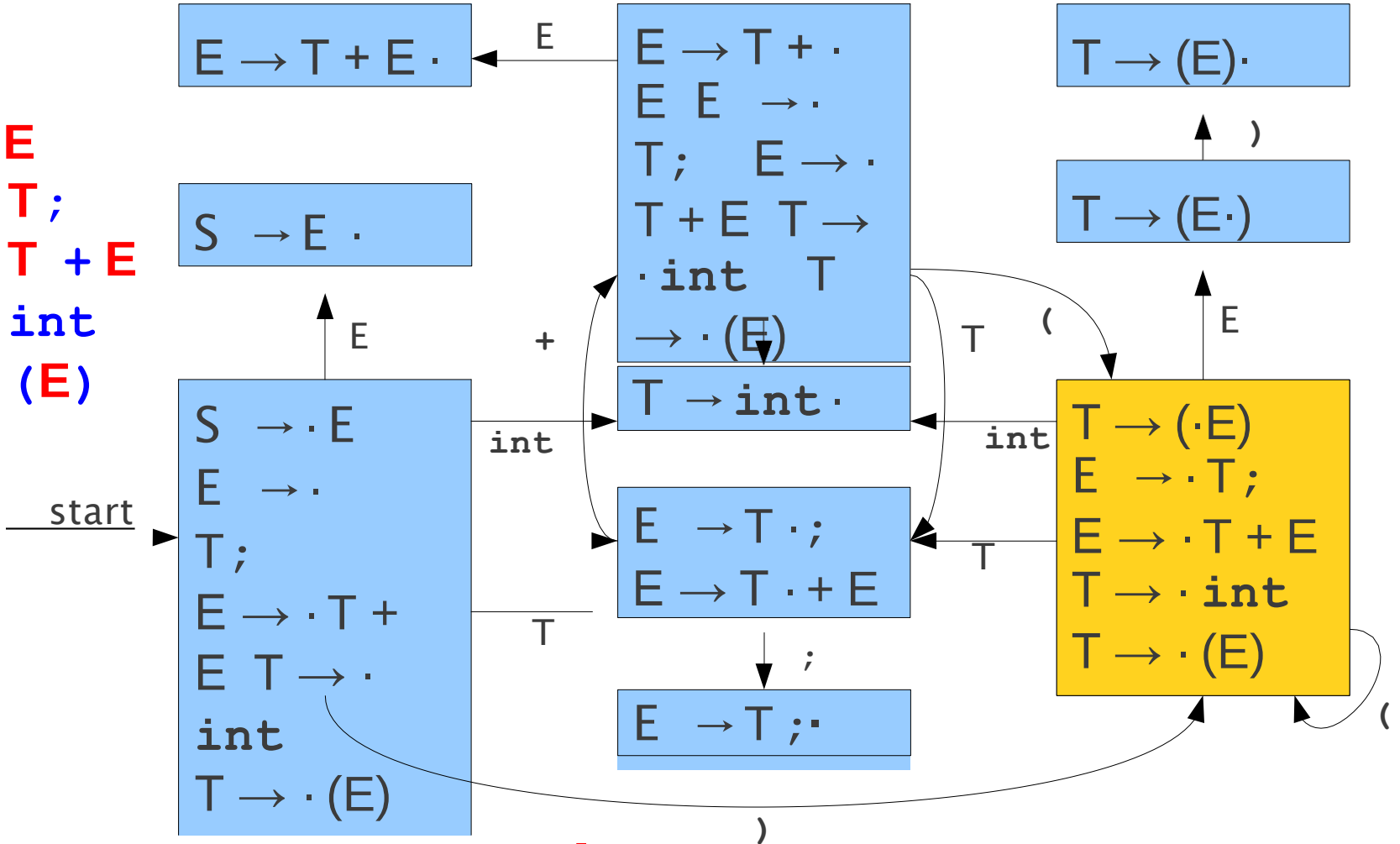


LR(0) Parsing



LR(0) Parsing

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

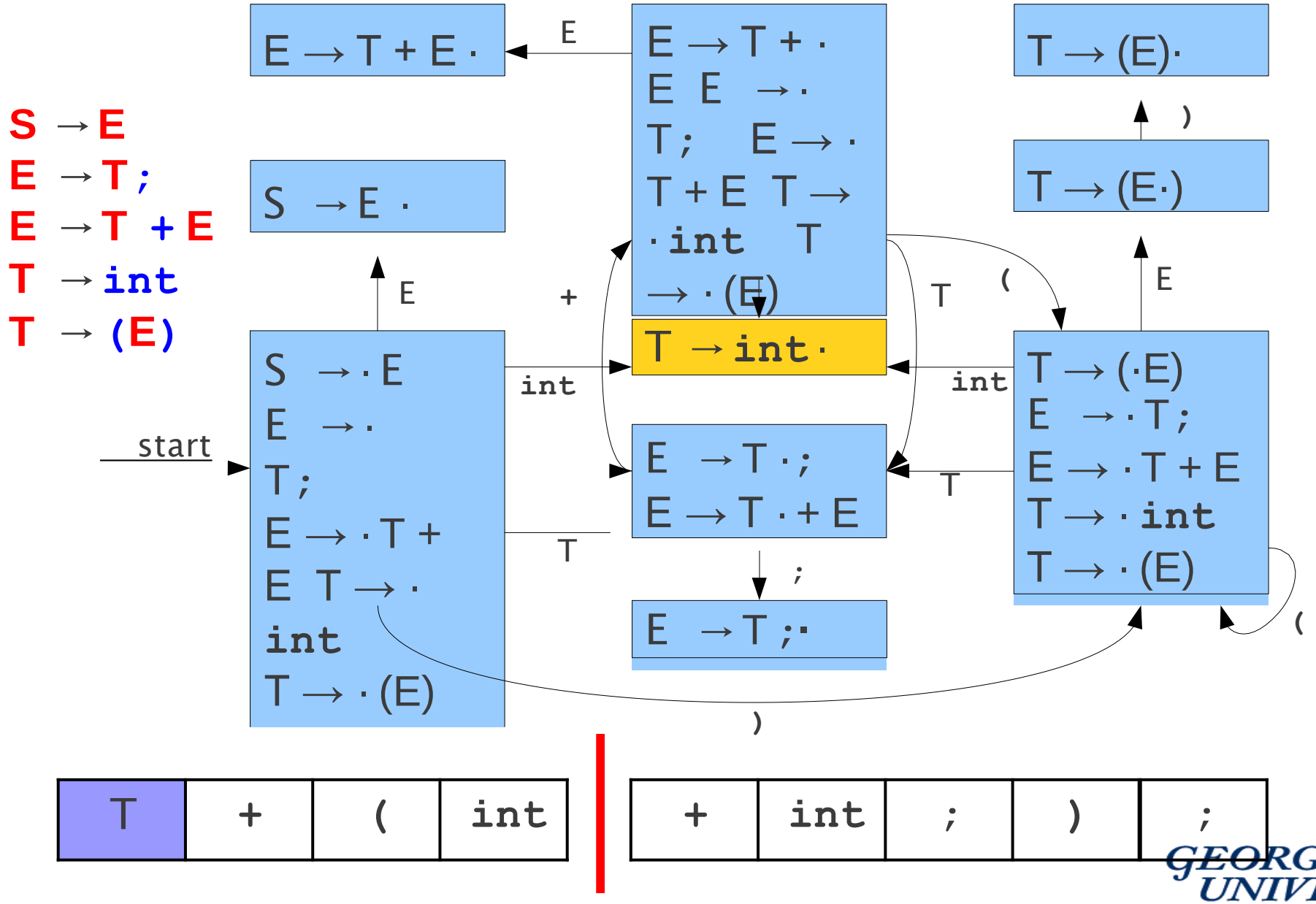


T	+	(	int

+	int	;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

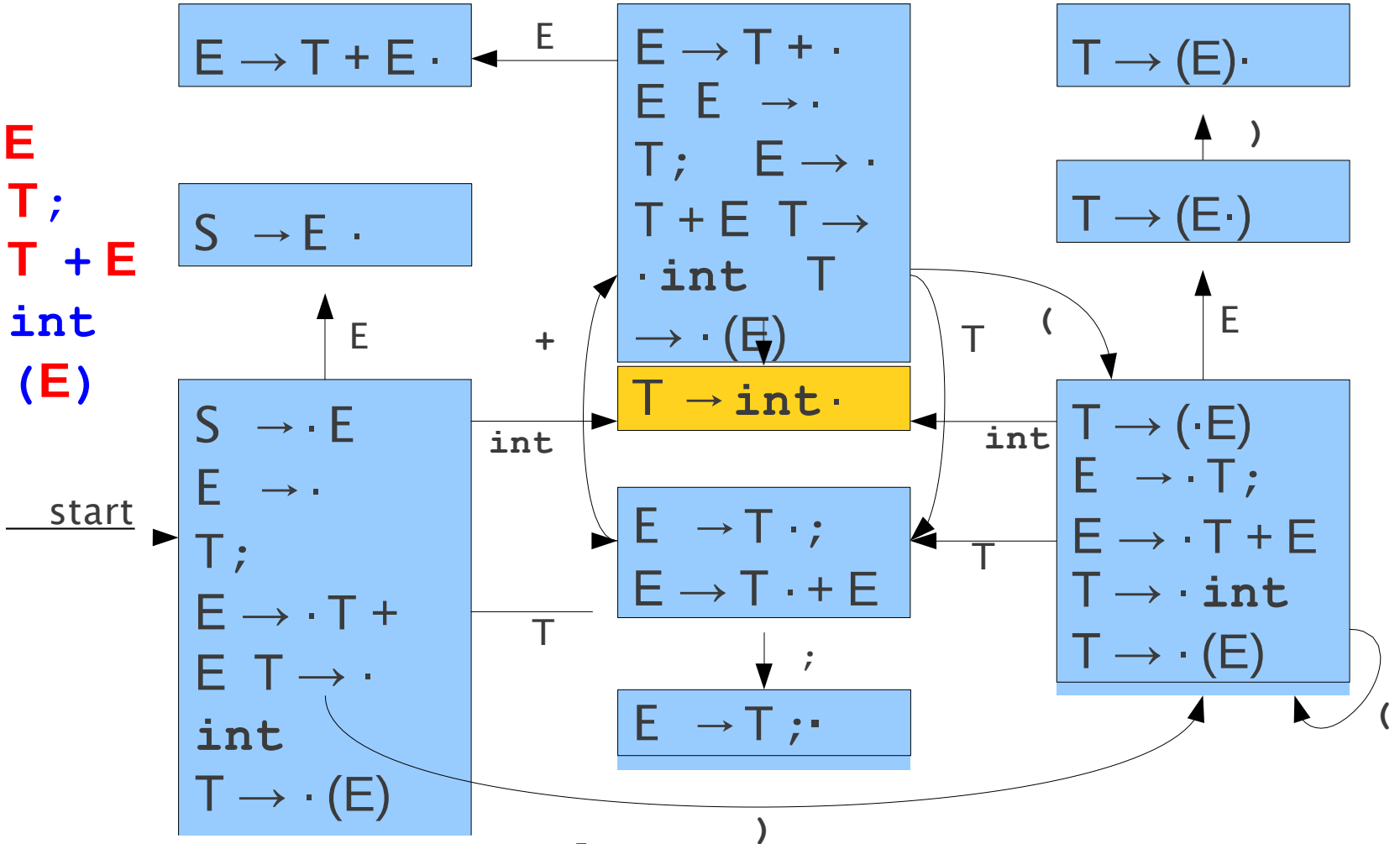


T	+	(	int	+	int	;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

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



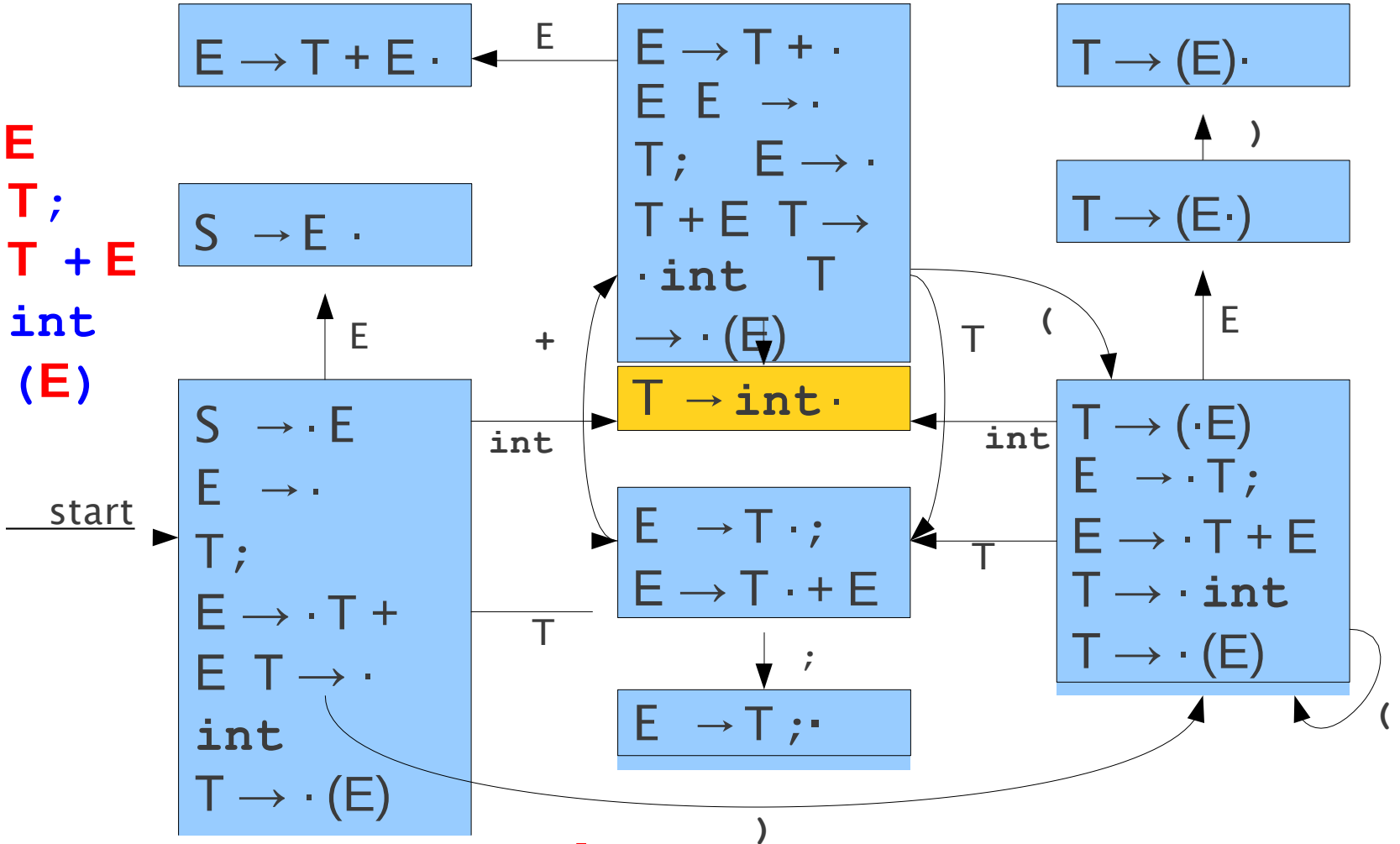
T	+	(

+	int	;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

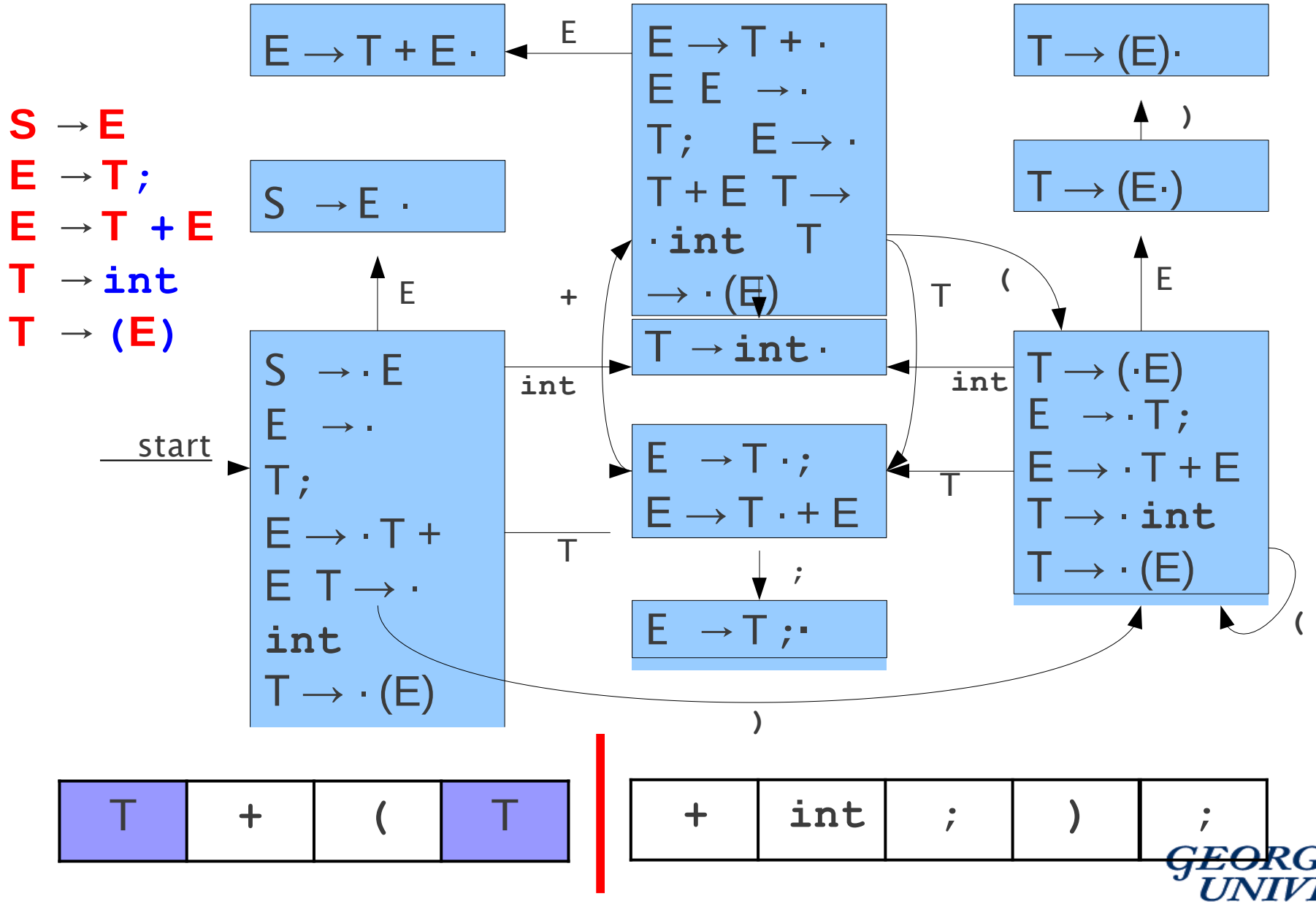
S → E
E → T ;
E → T + E
T → int
T → (E)



T	+	(	T	+	int	;	)	;

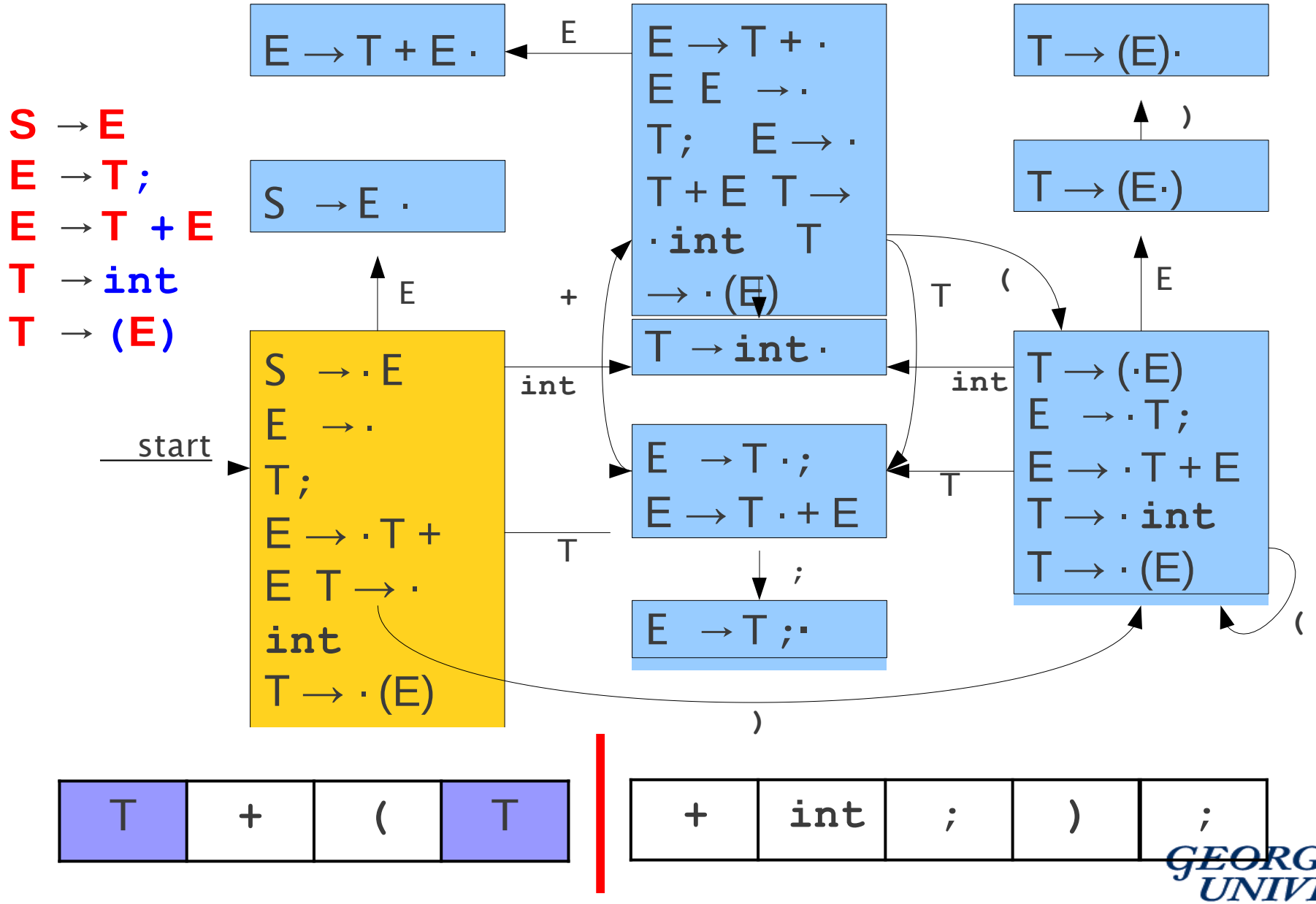
GEORGETOWN UNIVERSITY

LR(0) Parsing



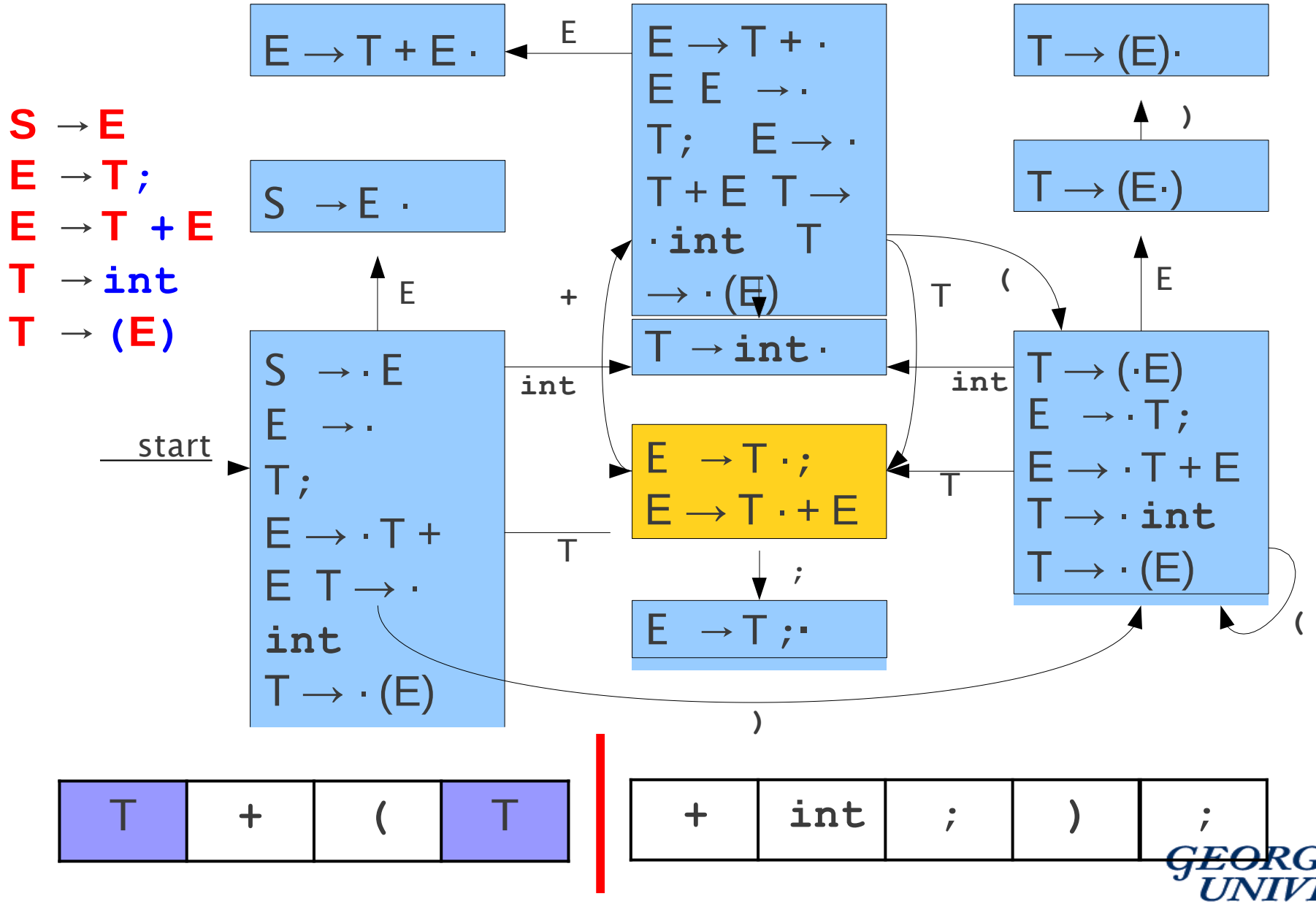
GEORGETOWN UNIVERSITY

LR(0) Parsing



GEORGETOWN UNIVERSITY

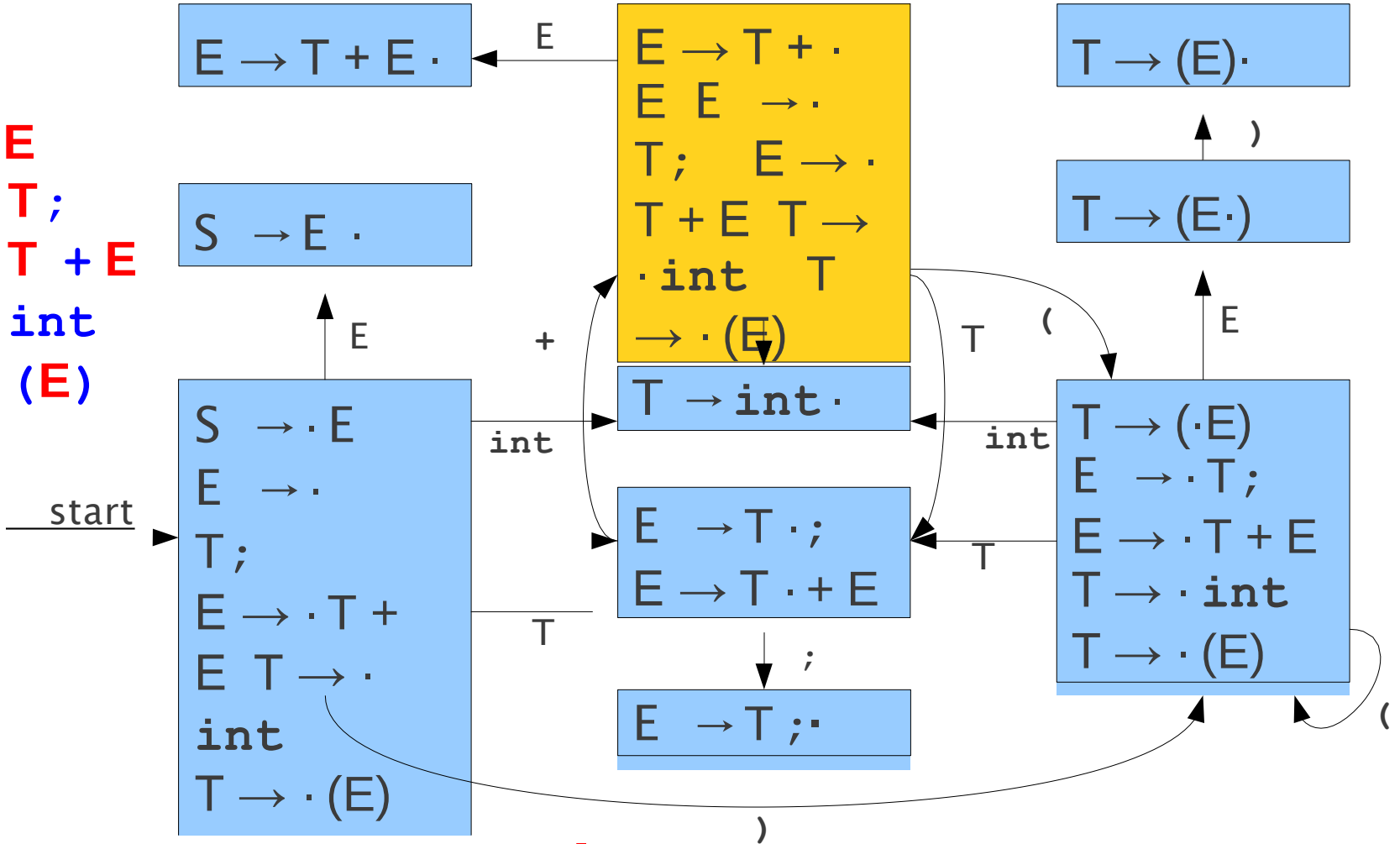
LR(0) Parsing

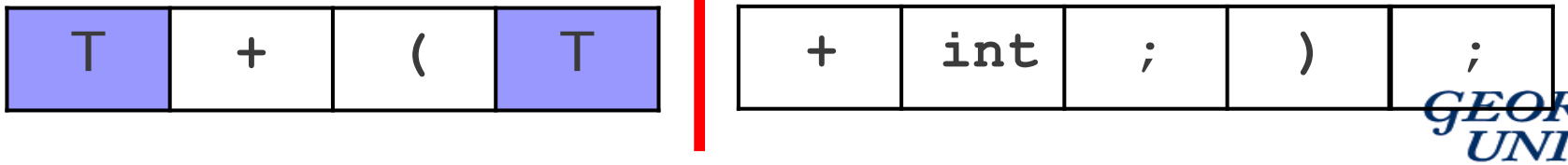


GEORGETOWN UNIVERSITY

LR(0) Parsing

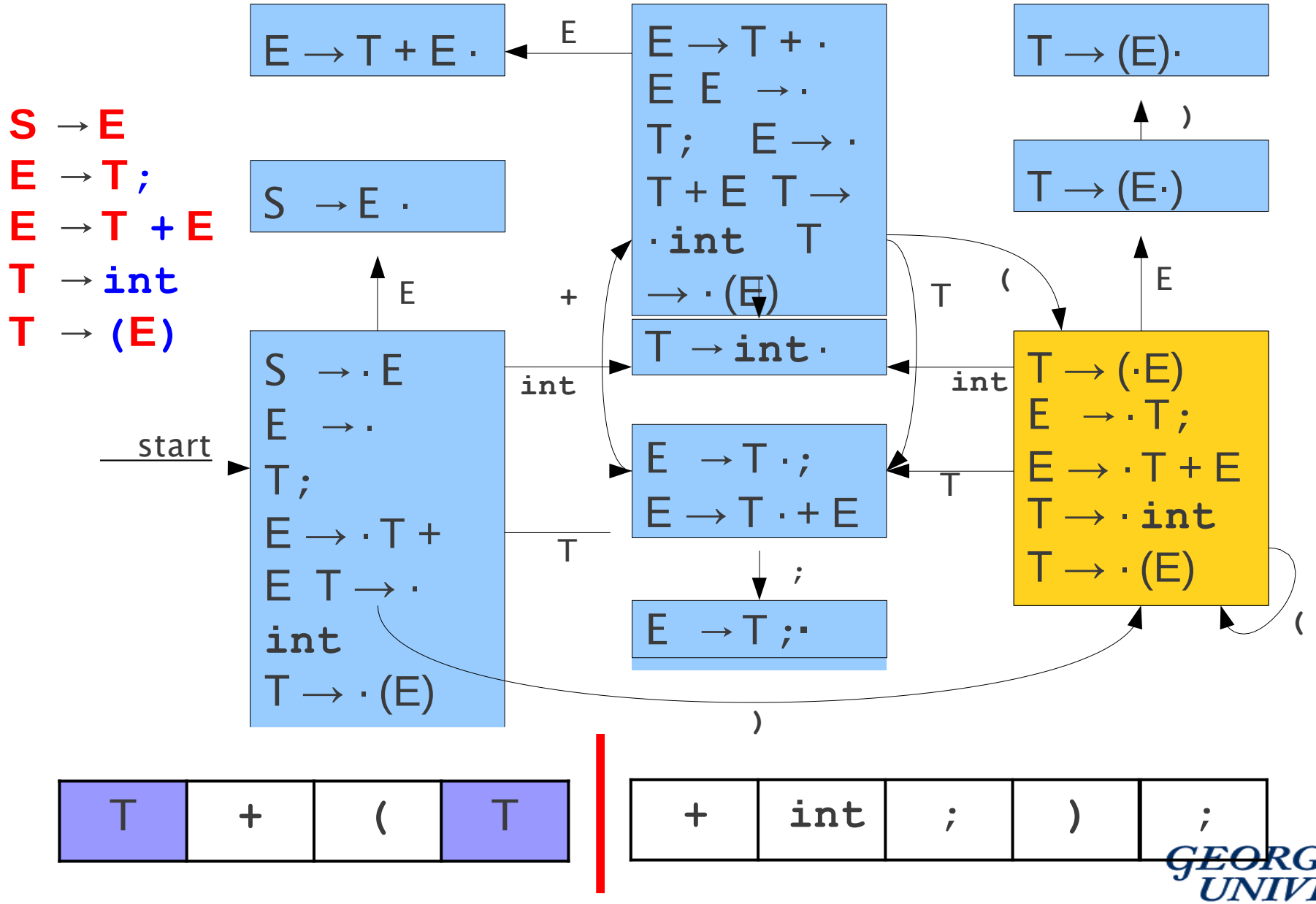
S → E
E → T ;
E → T + E
T → int
T → (E)





GEORGETOWN UNIVERSITY

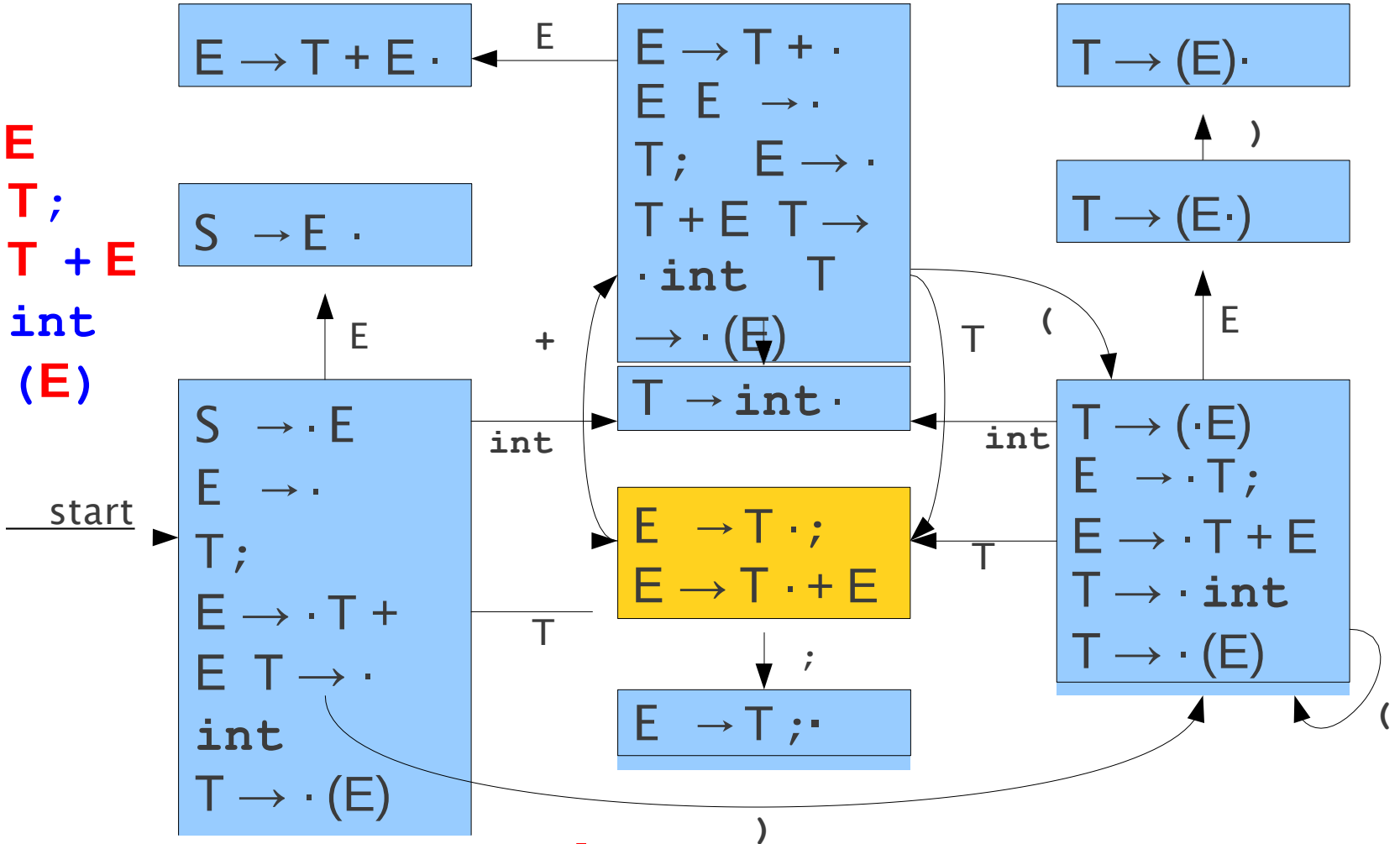
LR(0) Parsing



GEORGETOWN UNIVERSITY

LR(0) Parsing

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

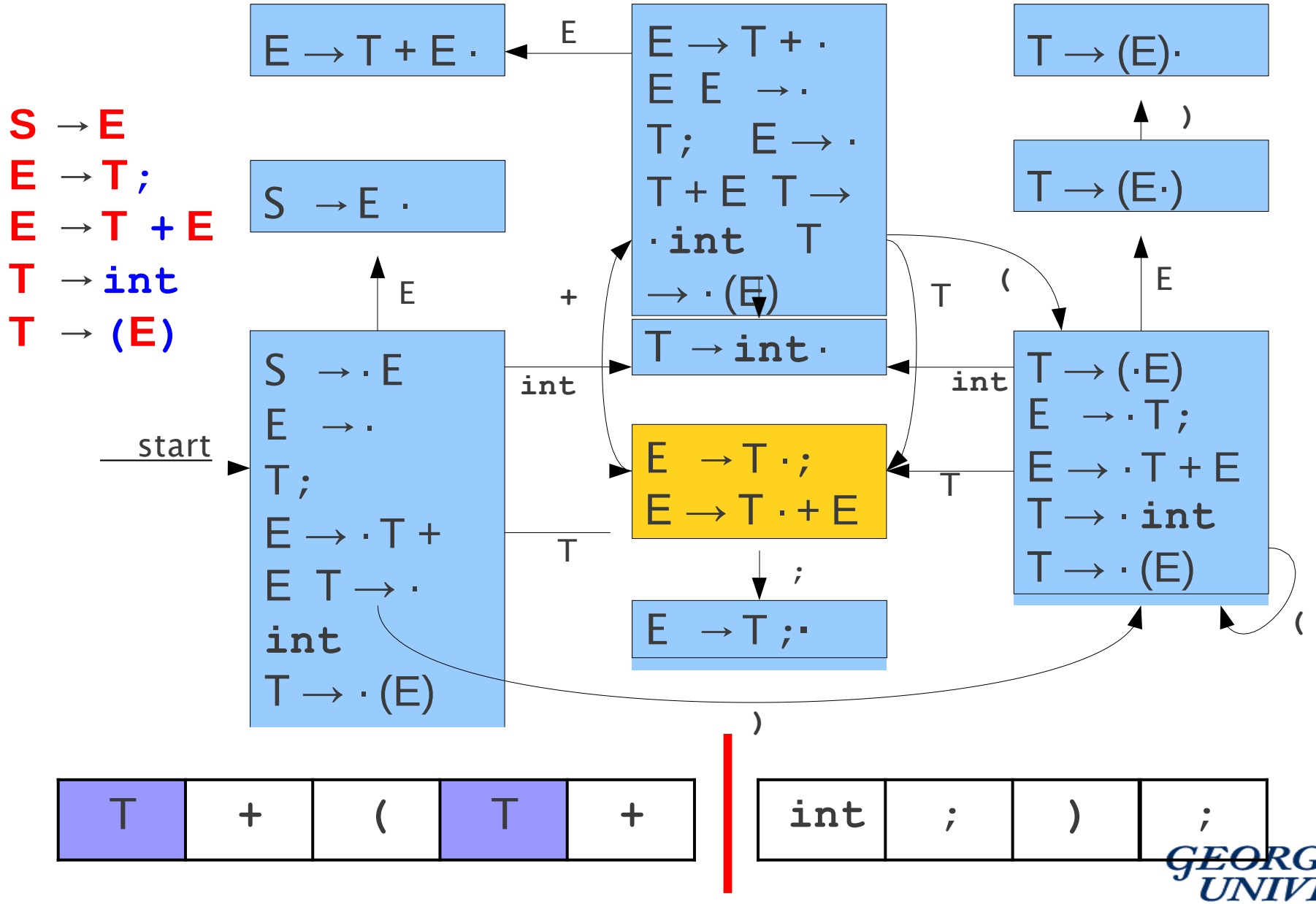


T	+	(	T

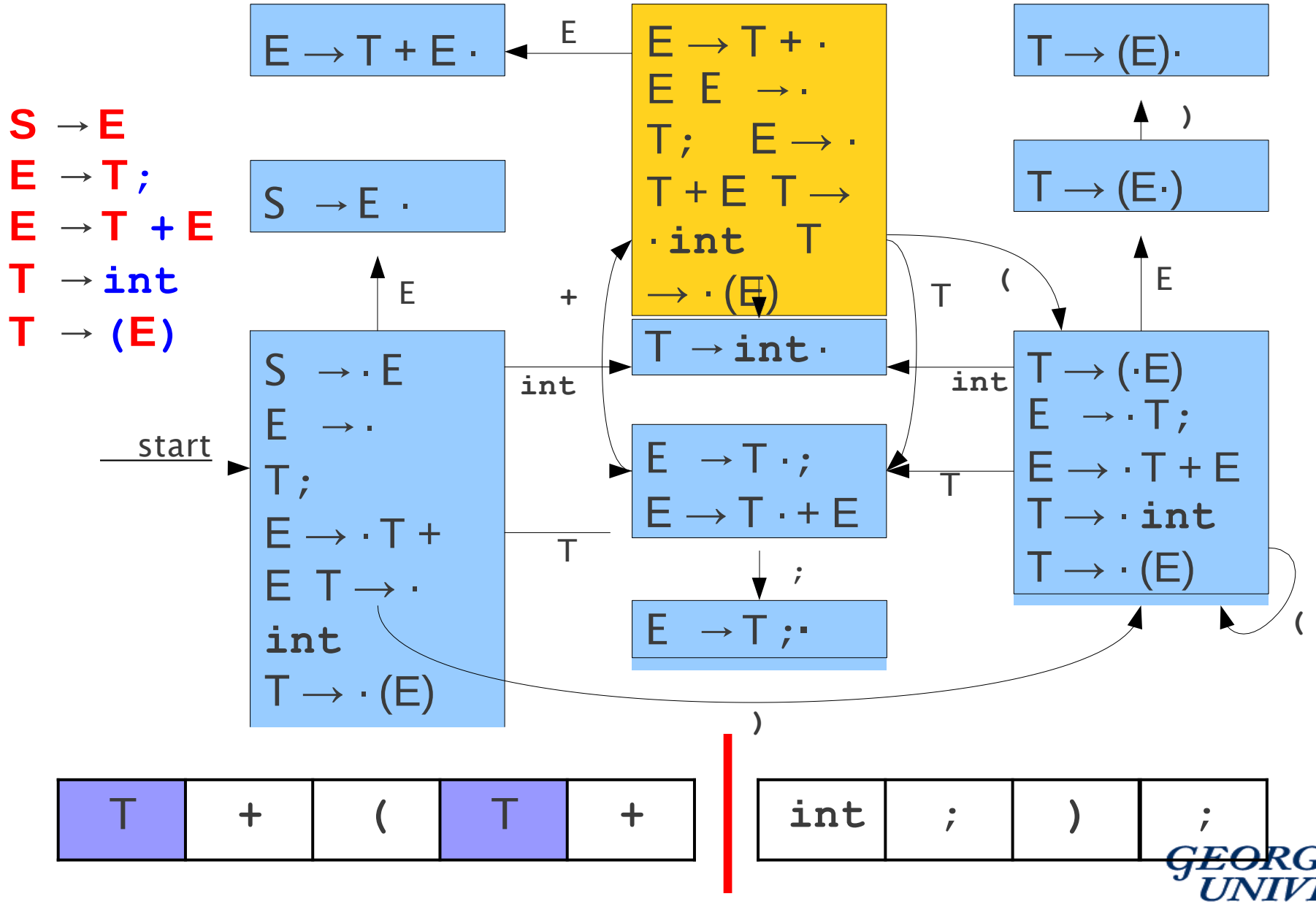
+	int	;	)	;

GEORGETOWN UNIVERSITY

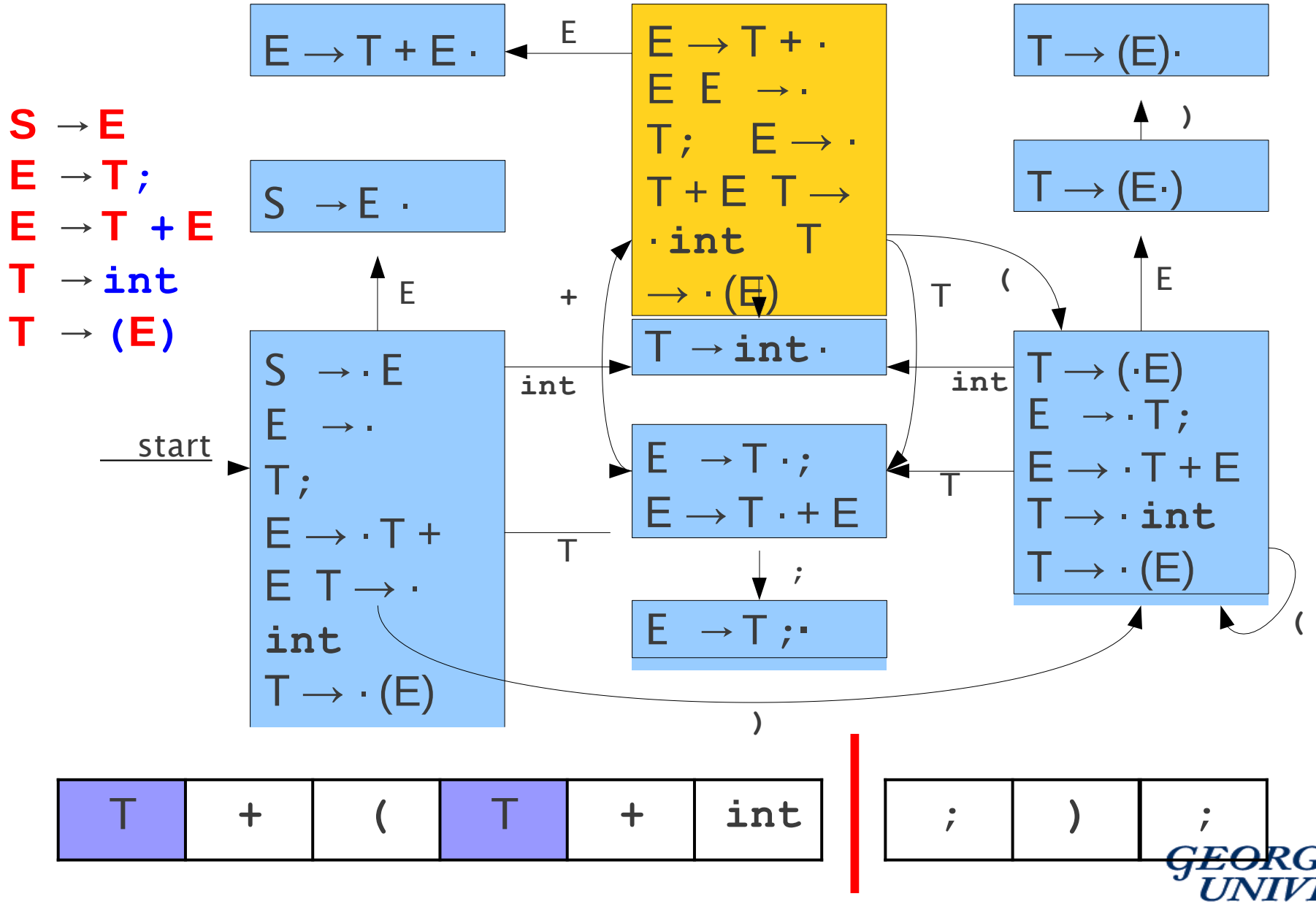
LR(0) Parsing


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

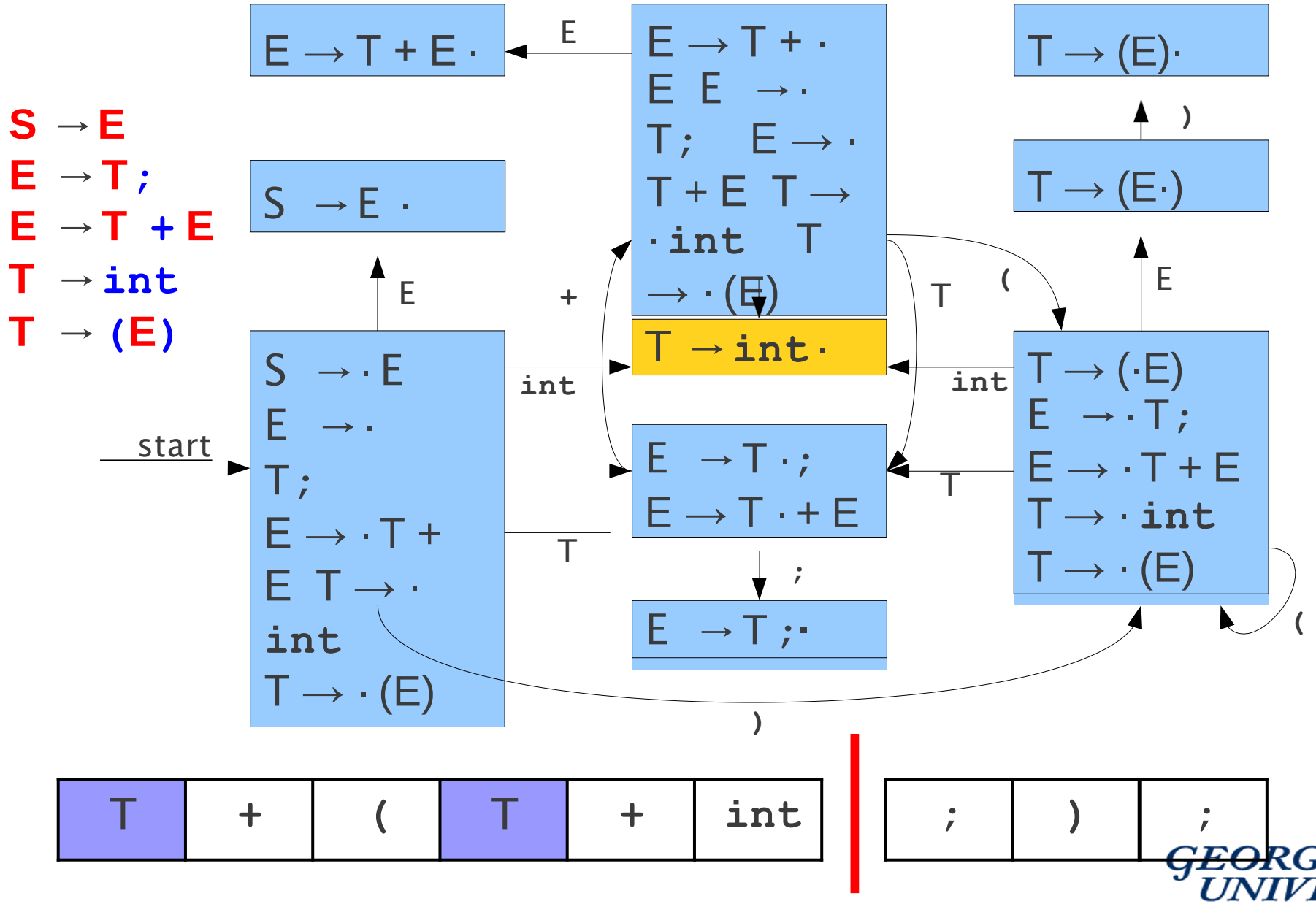
LR(0) Parsing



LR(0) Parsing

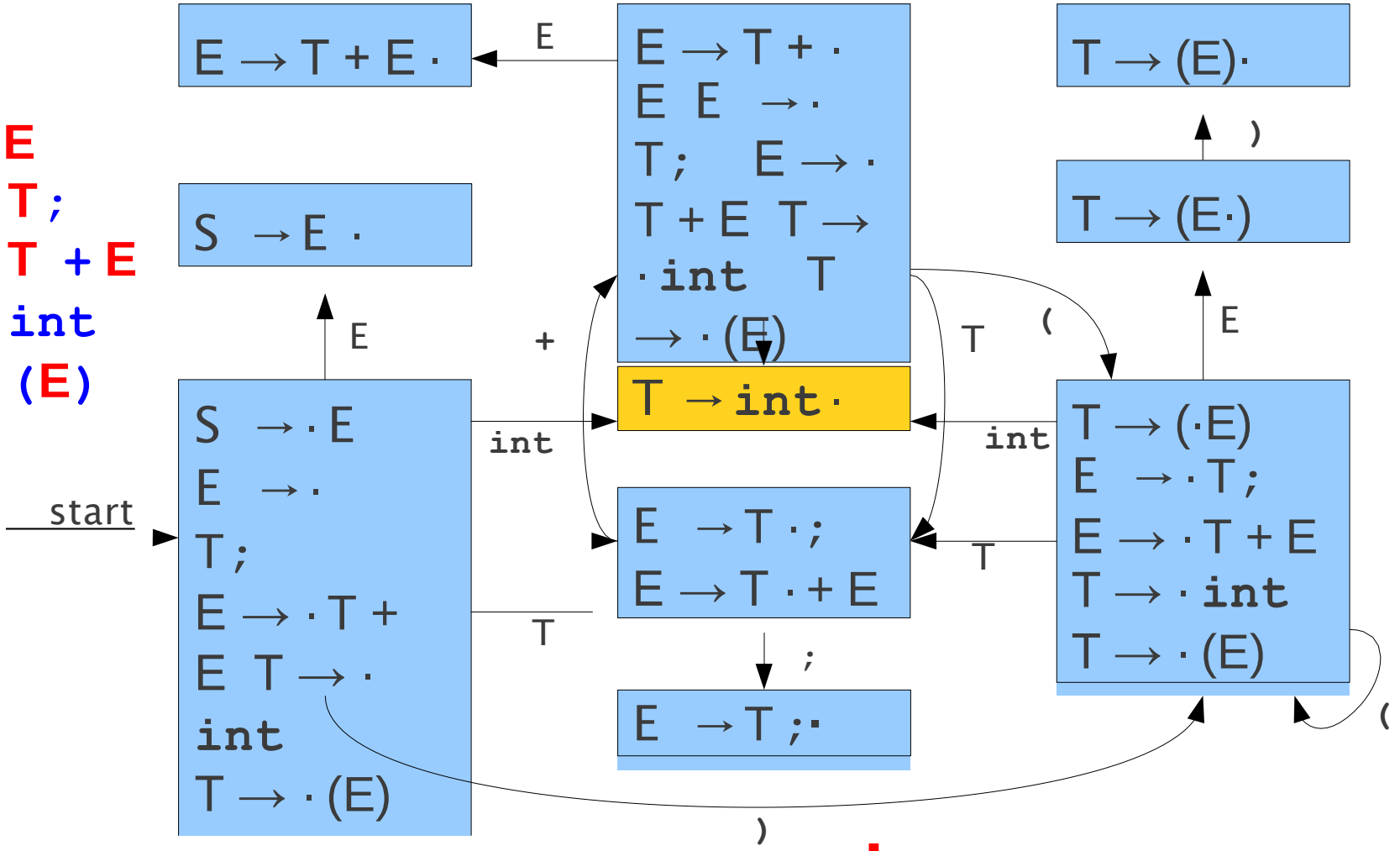


LR(0) Parsing



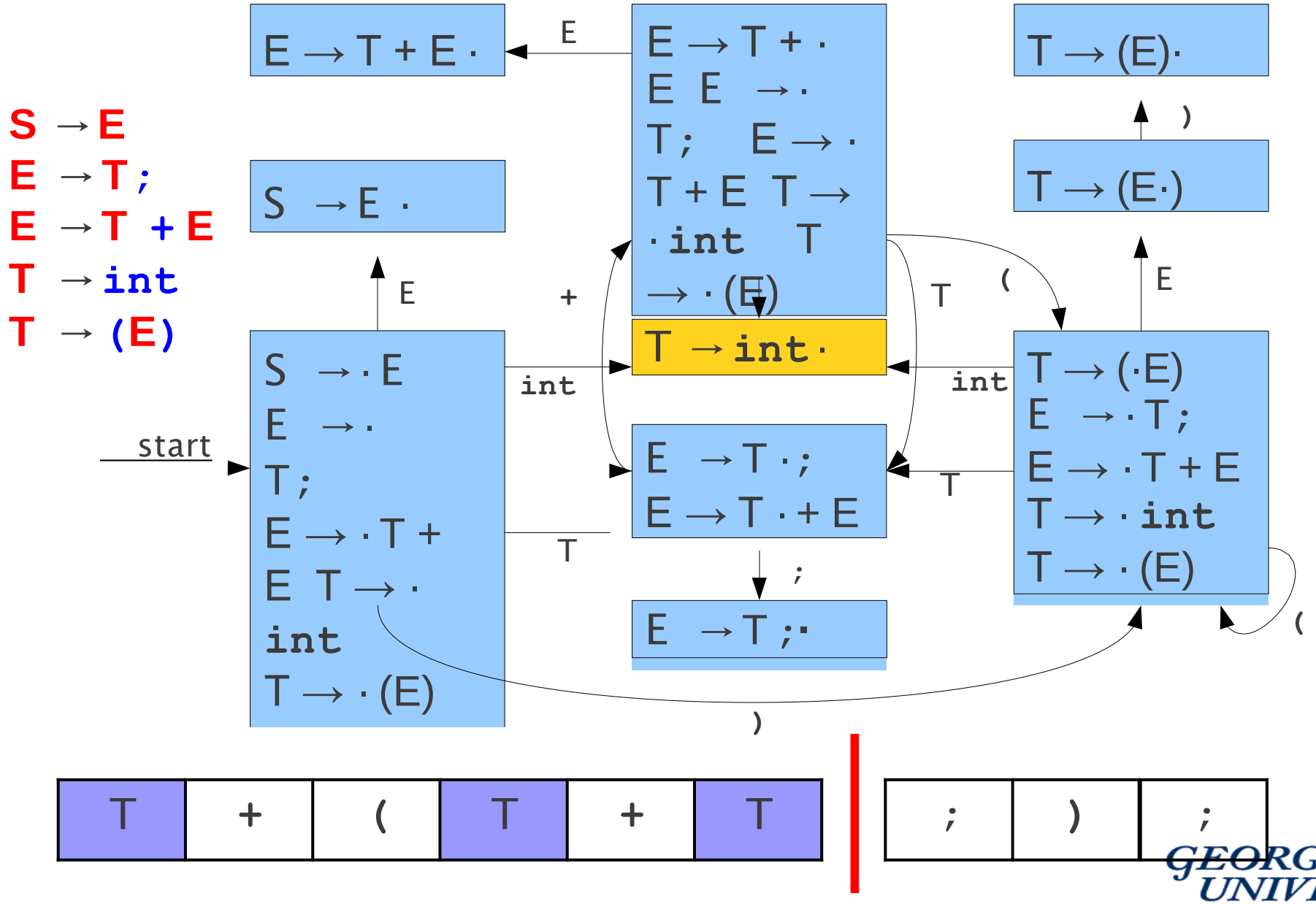
LR(0) Parsing

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

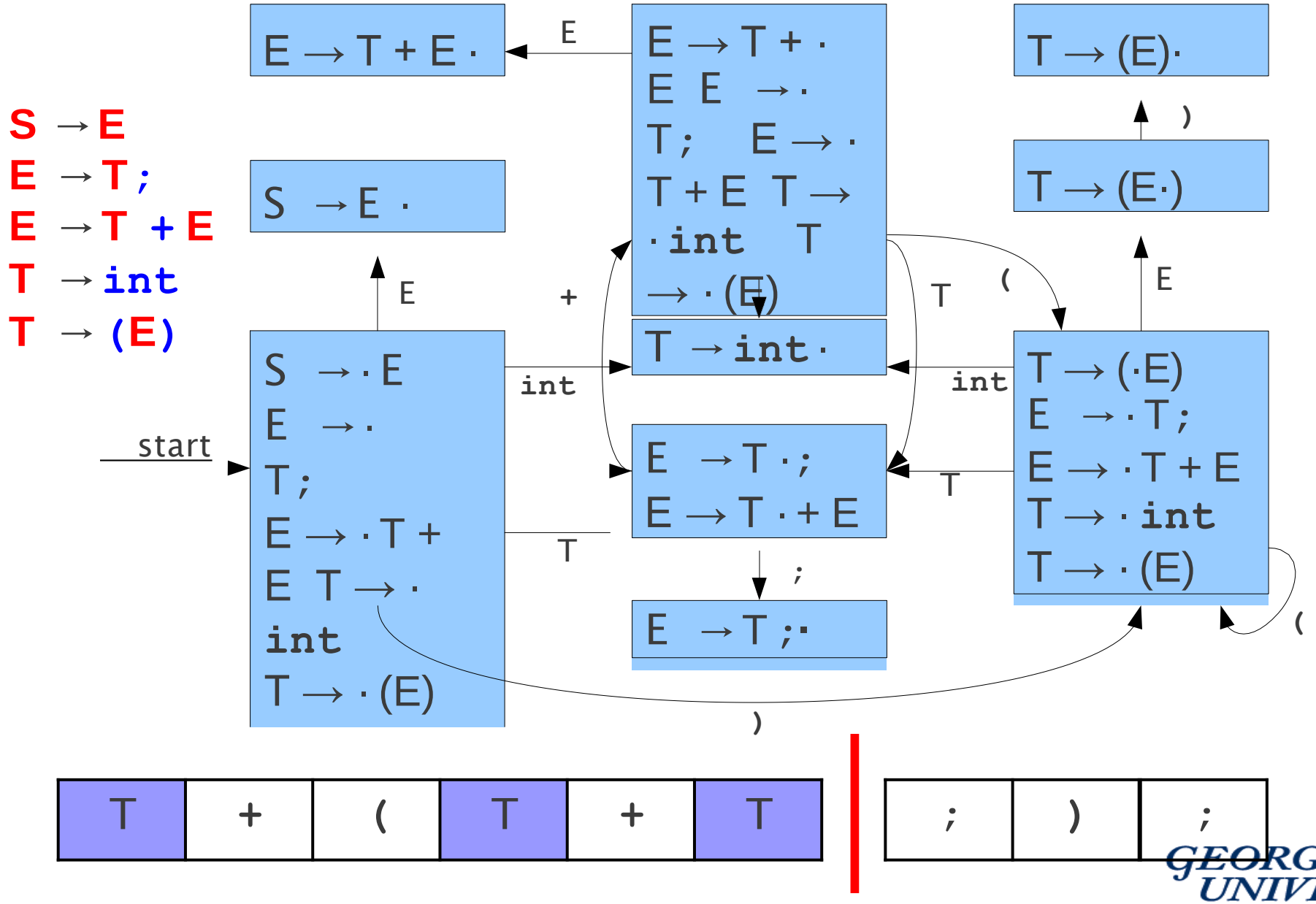


GEORGETOWN UNIVERSITY

LR(0) Parsing



LR(0) Parsing



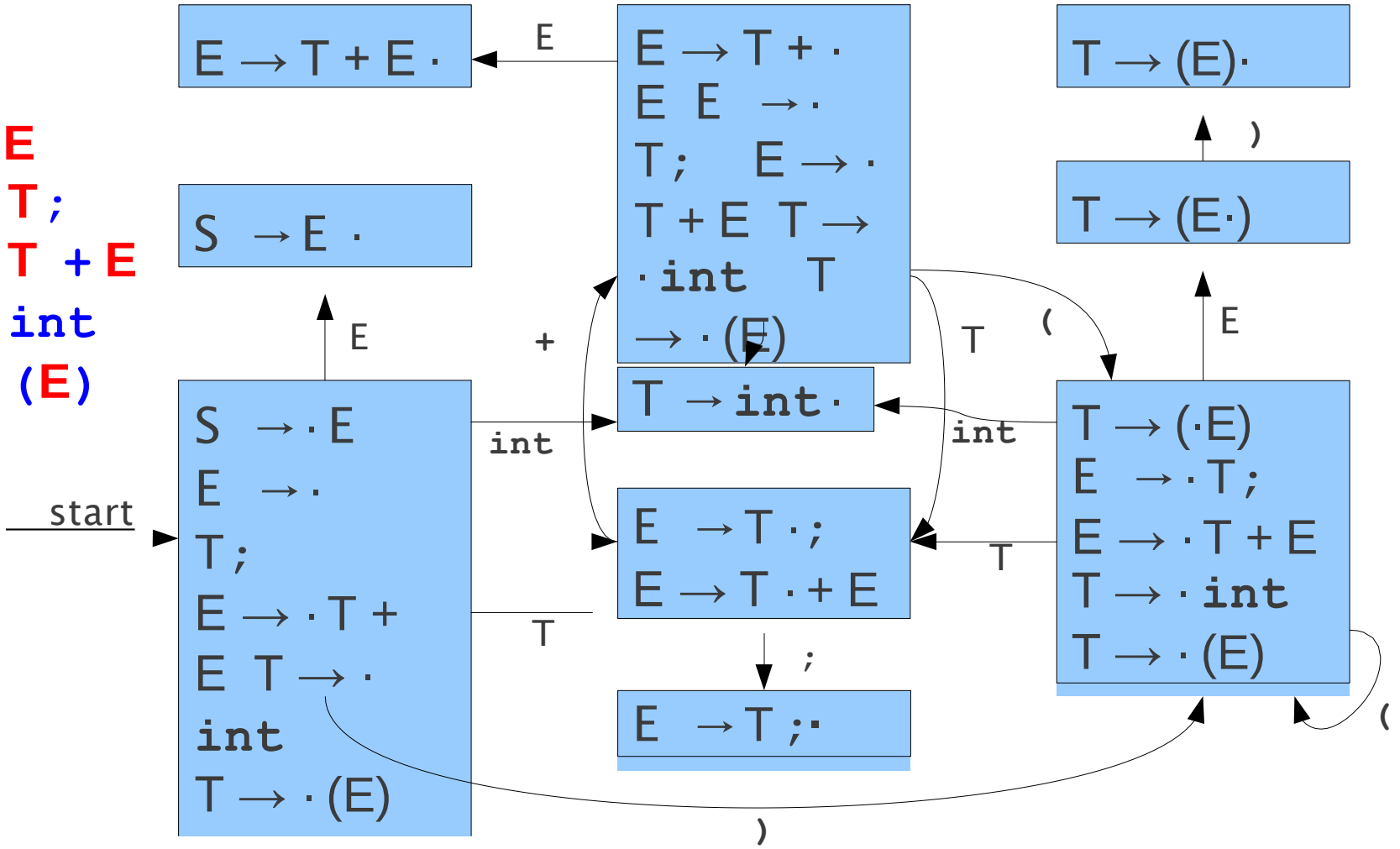
GEORGETOWN UNIVERSITY

An Optimization

- Rather than restart the automaton on each reduction, remember what state we were in for each symbol.
- When applying a reduction, restart the automaton from the last known good state.

LR(0) Parsing

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

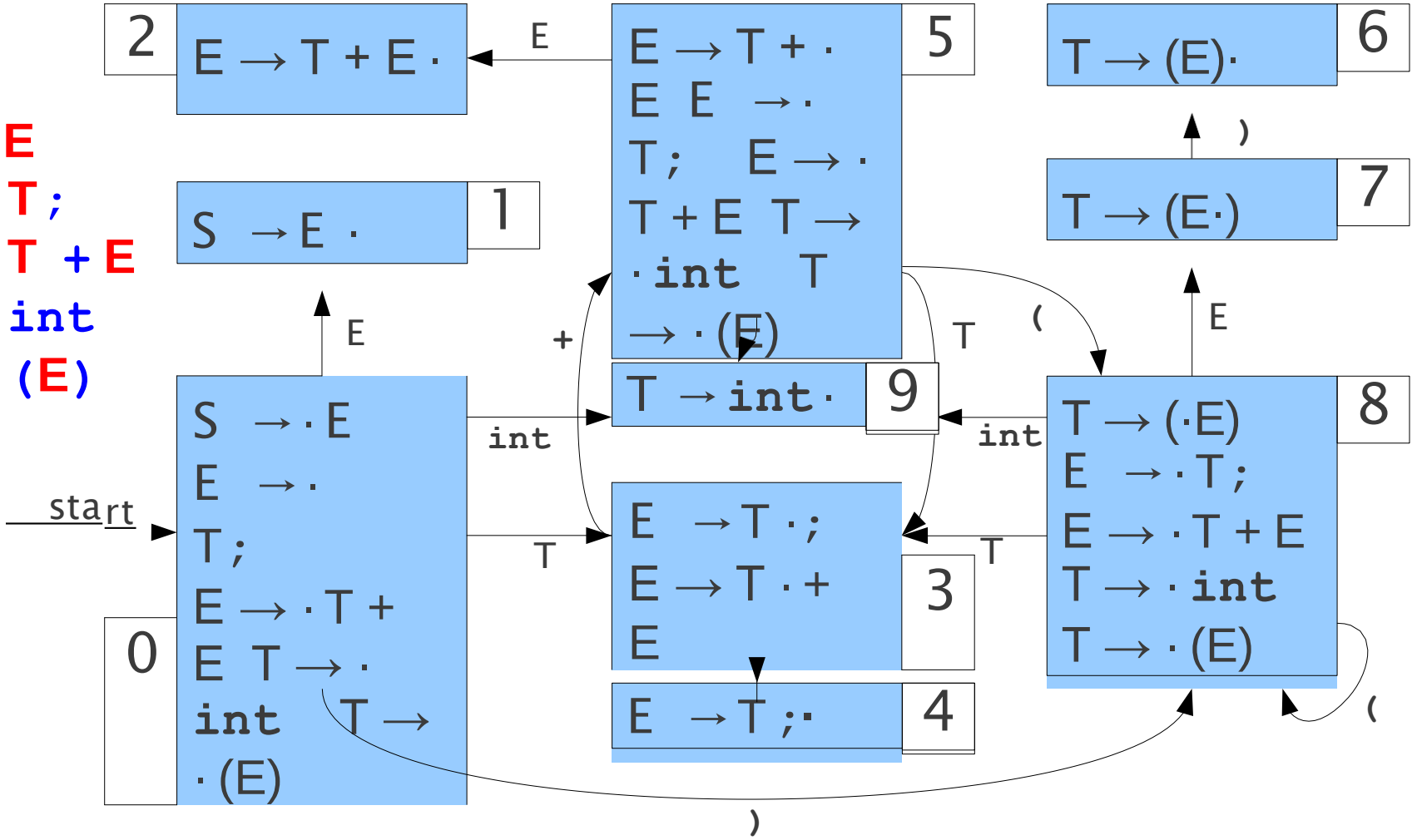


int	+	(	int	+	int	;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

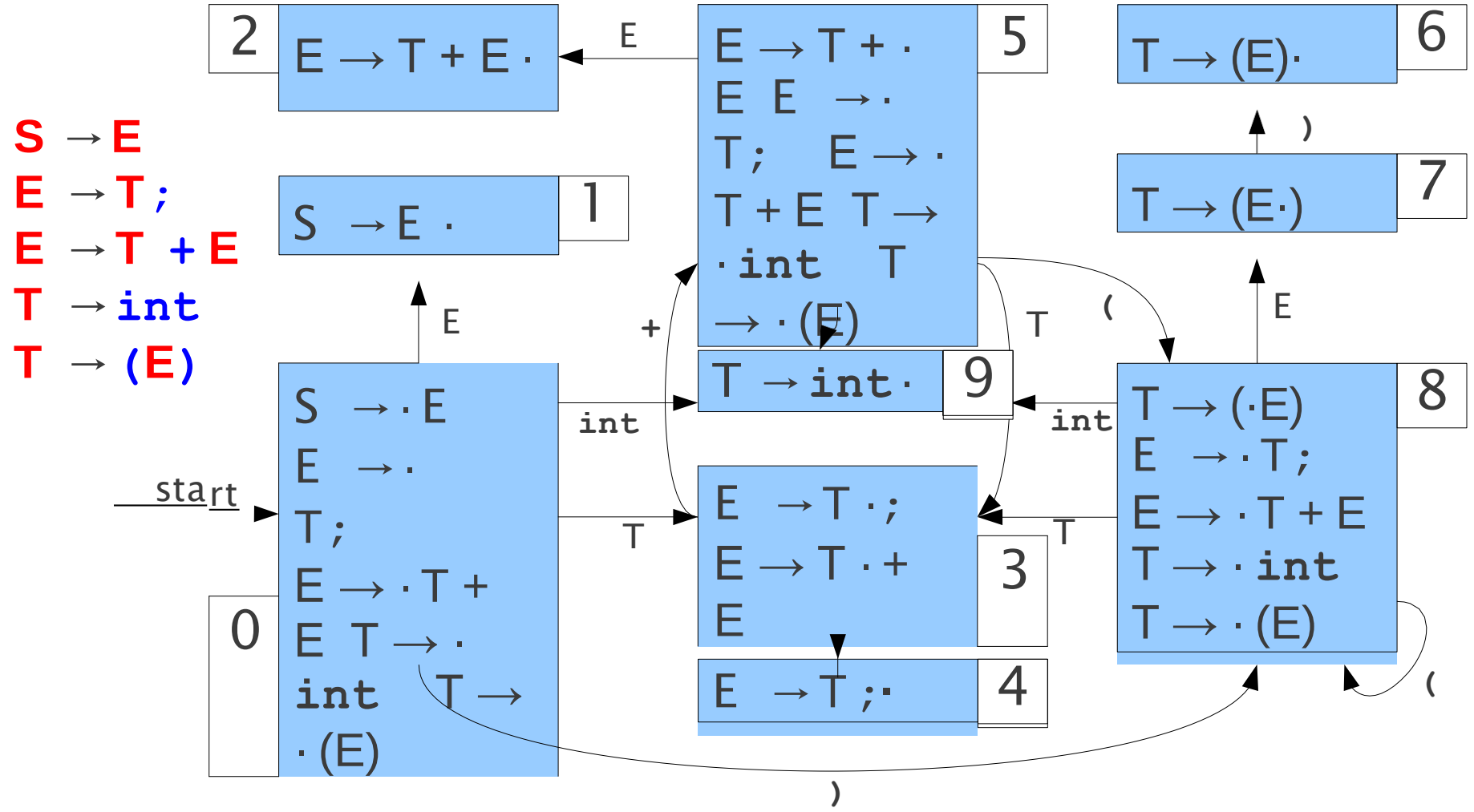
S → E
E → T;
E → T + E
T → int
T → (E)

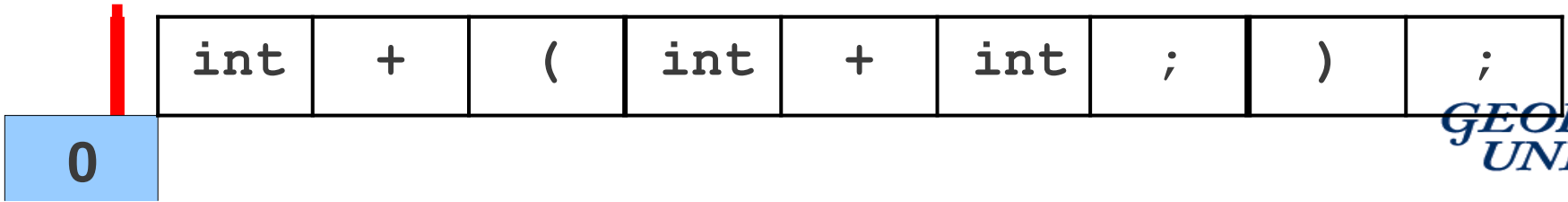


int	+	(	int	+	int	;	)	;

GEORGETOWN UNIVERSITY

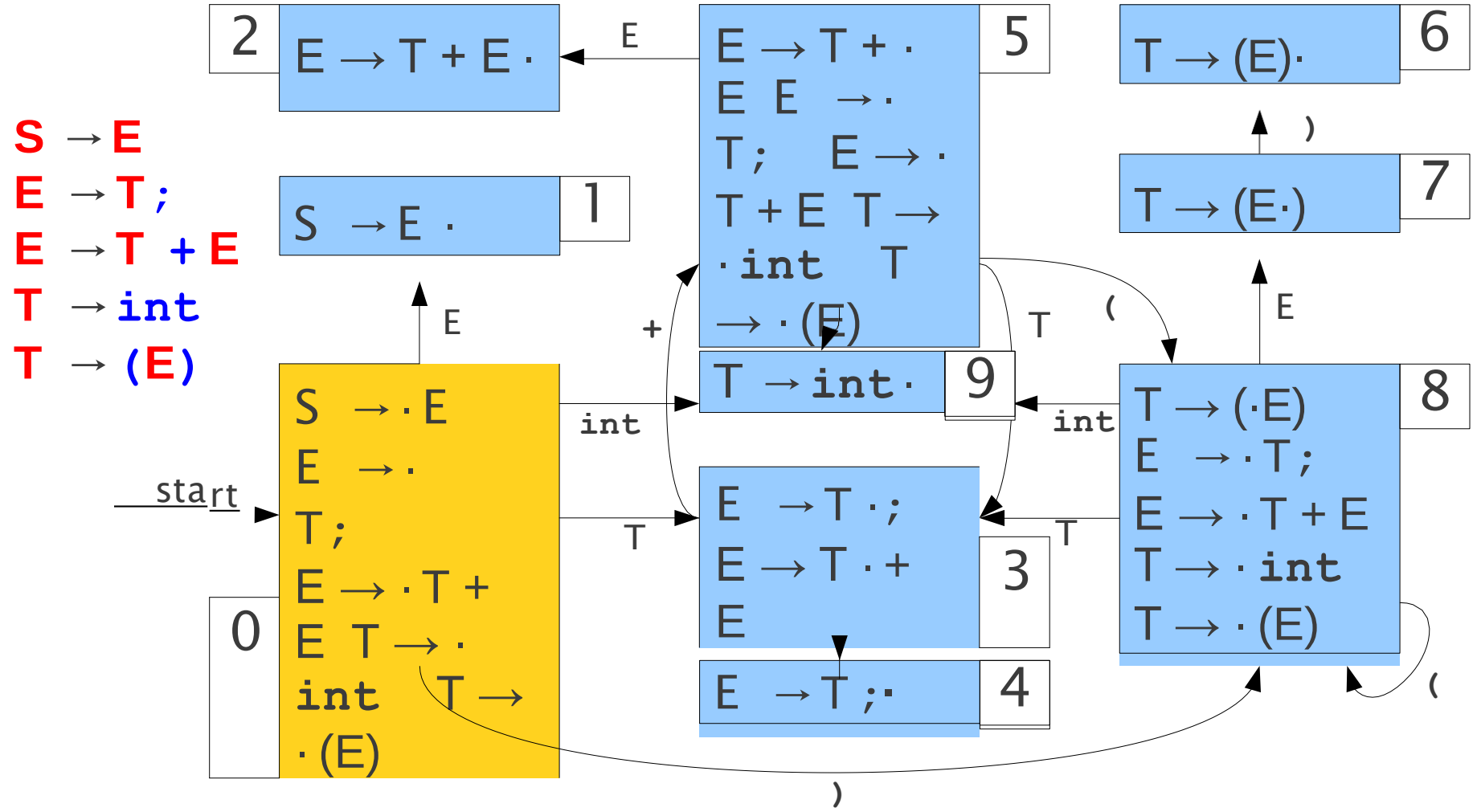
LR(0) Parsing

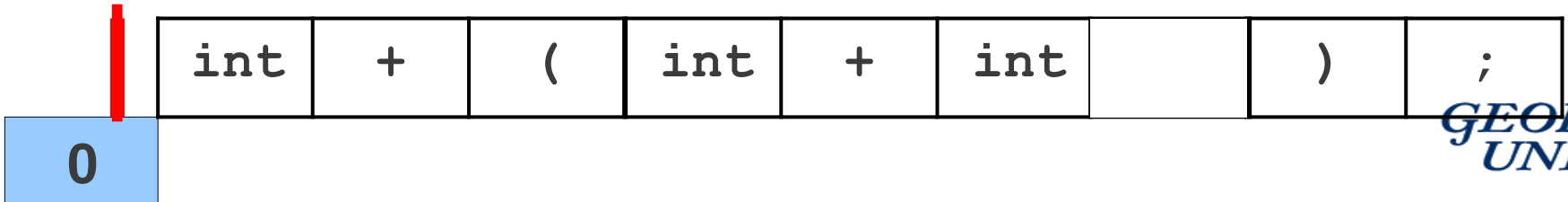




GEORGETOWN UNIVERSITY

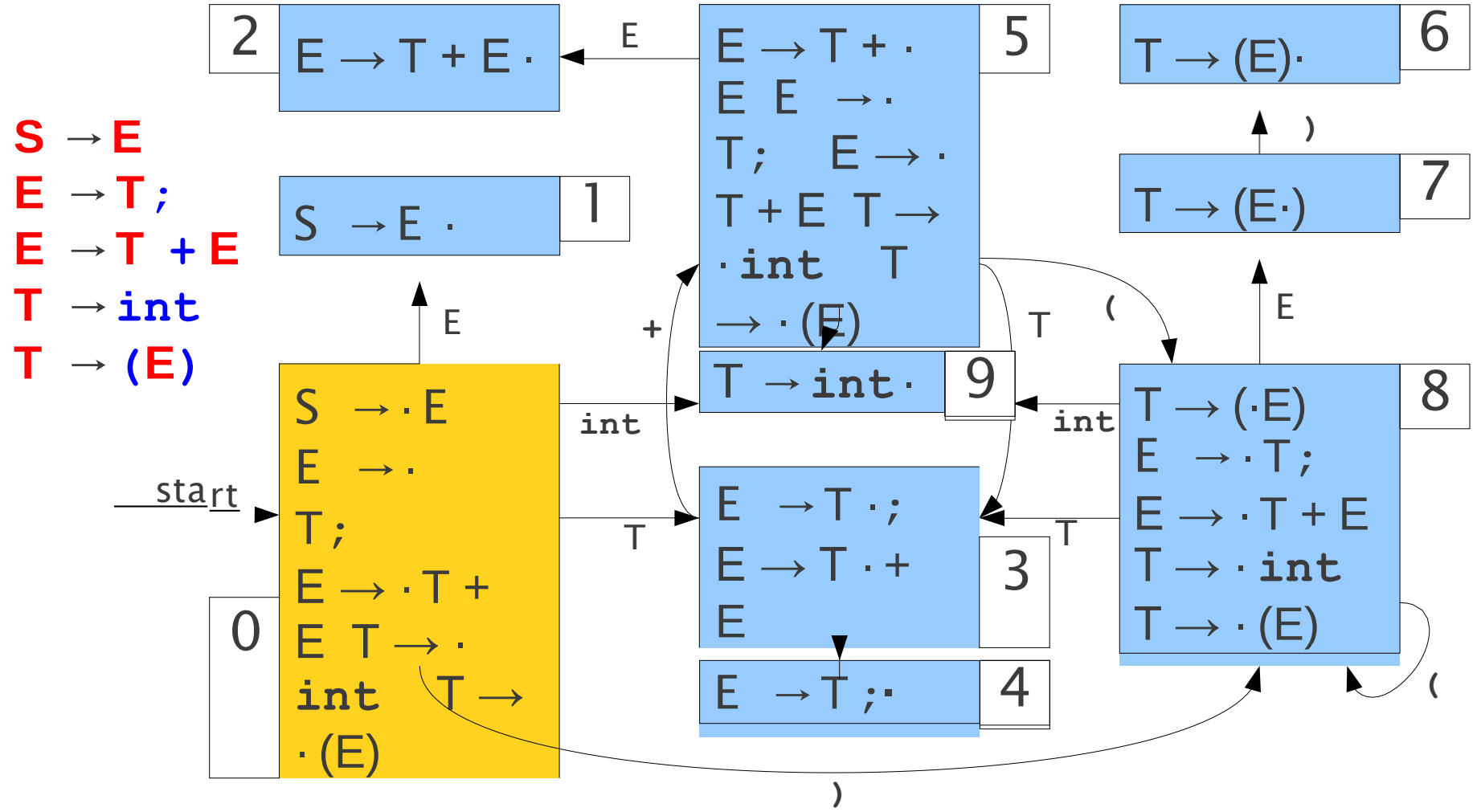
LR(0) Parsing





GEORGETOWN UNIVERSITY

LR(0) Parsing

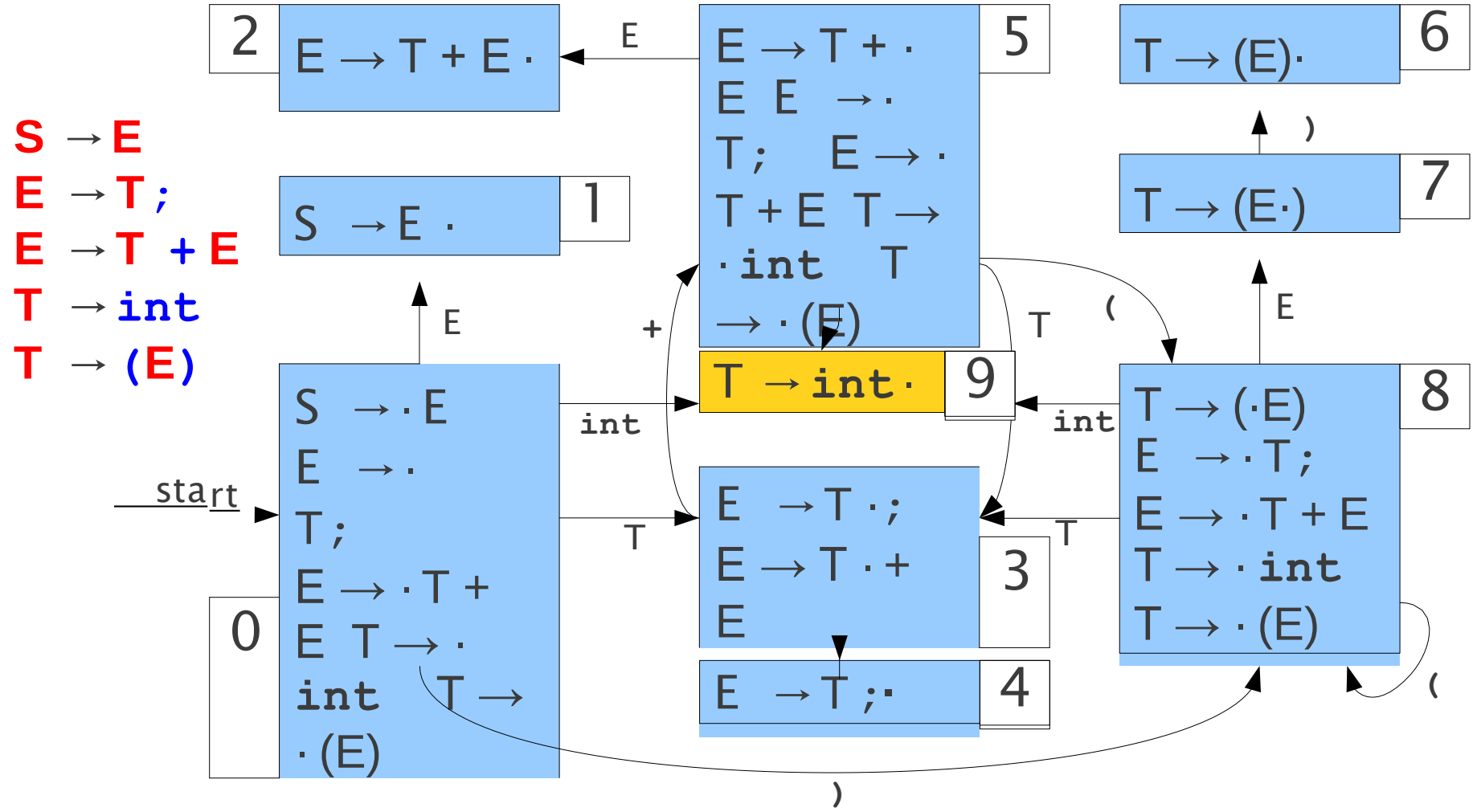


int	+	(	int	+	int	;	)	;

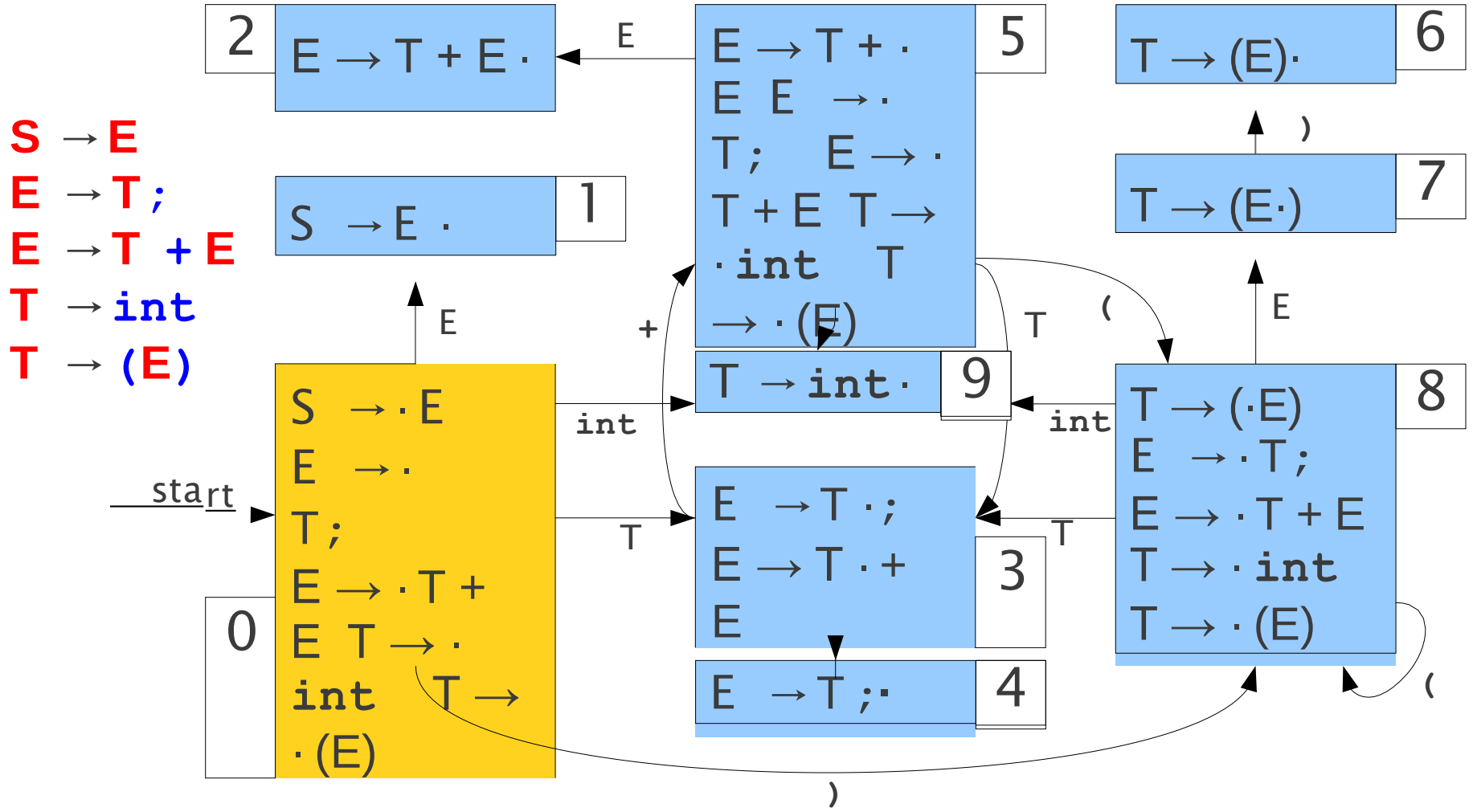
0

GEORGETOWN UNIVERSITY

LR(0) Parsing



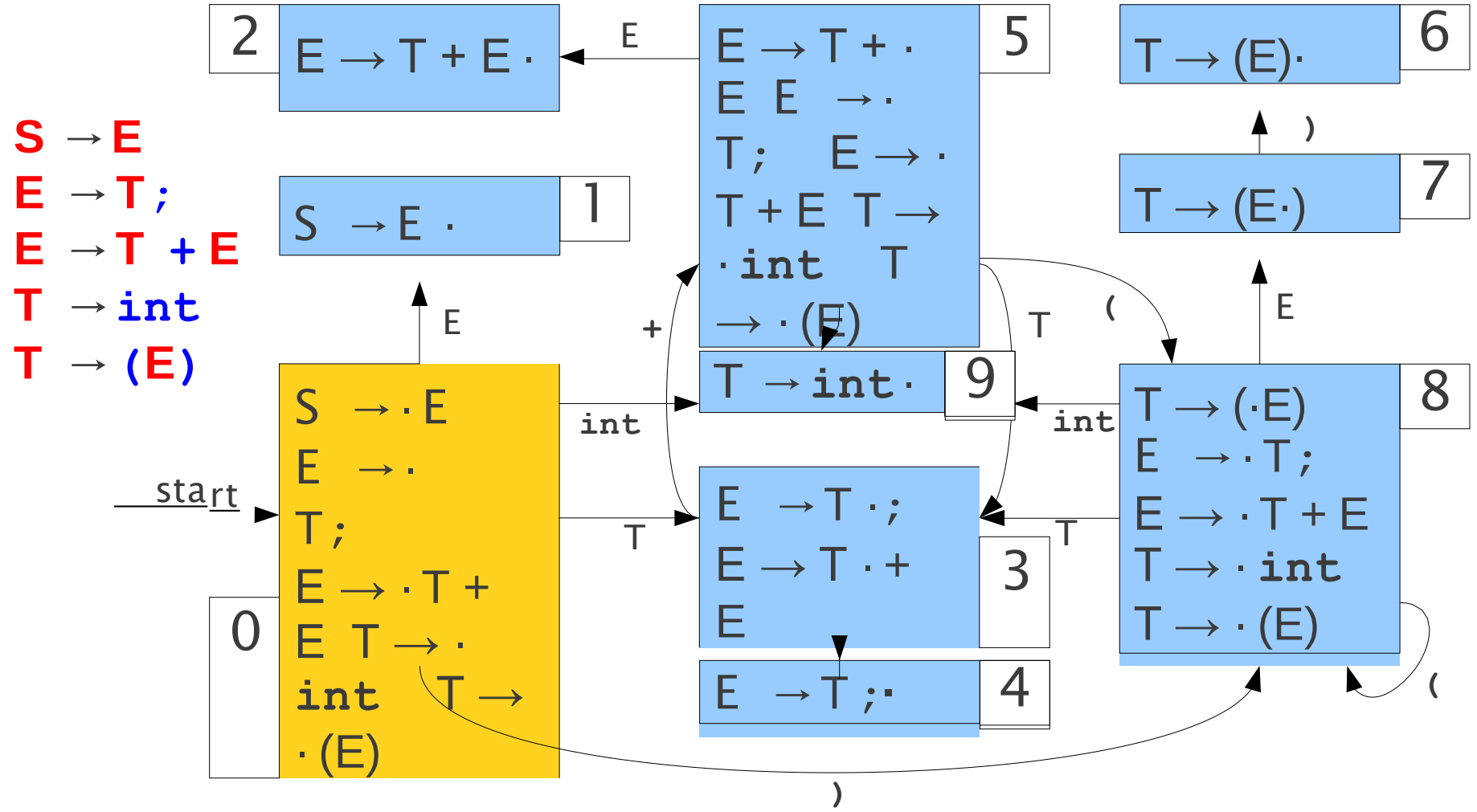
LR(0) Parsing

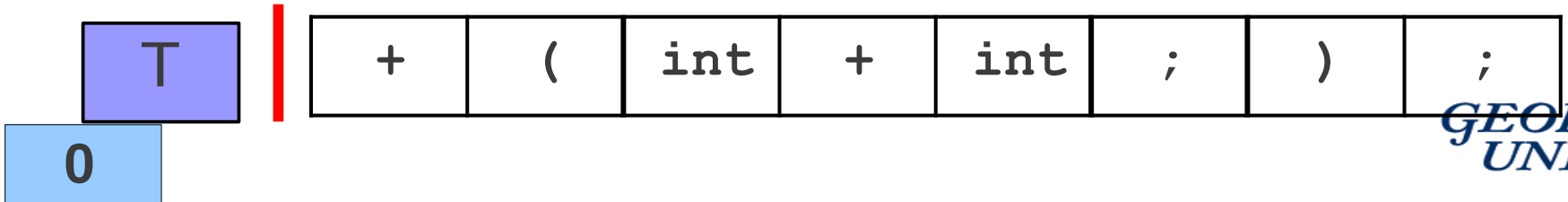


+	(	int	+	int	;	)	;

0

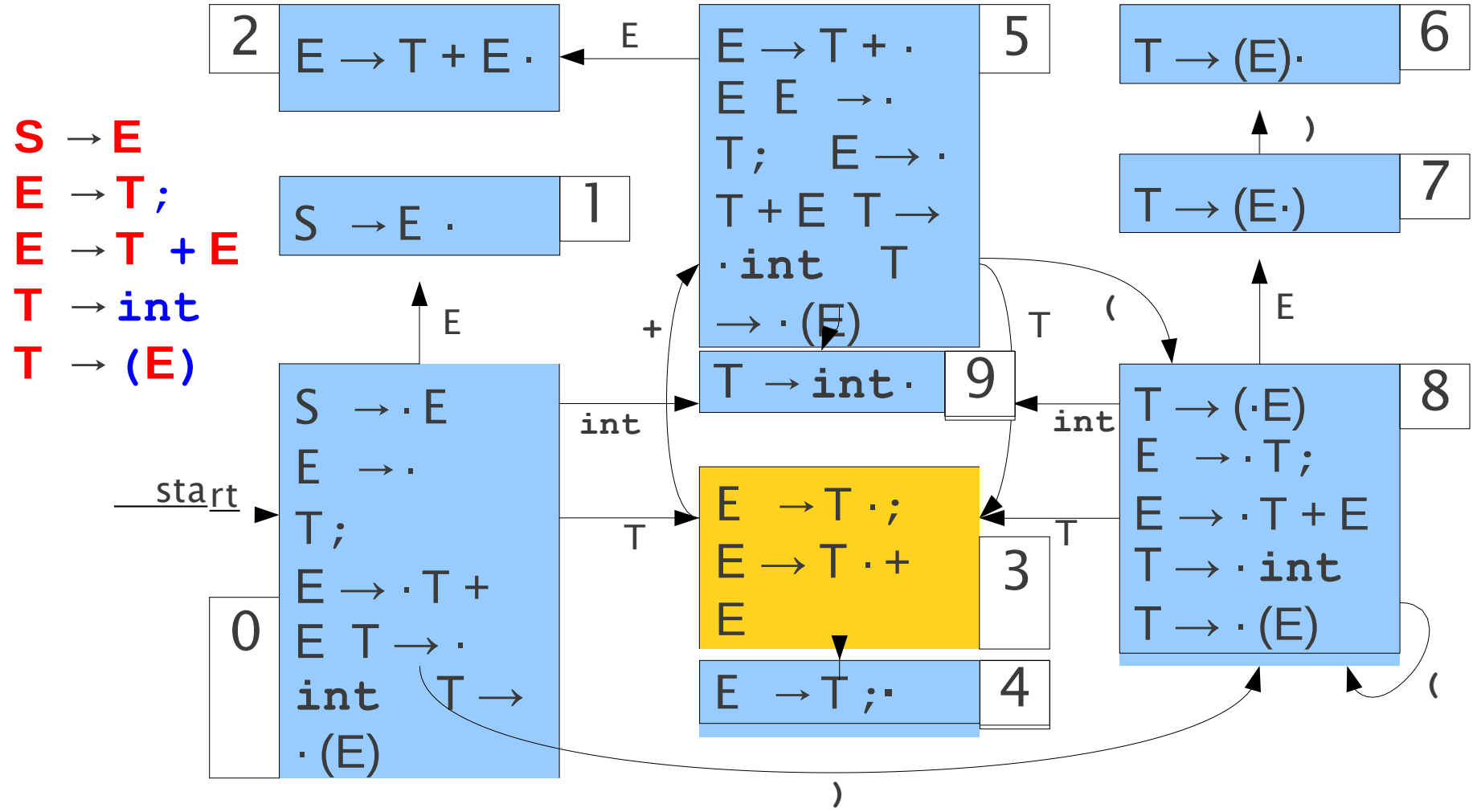
LR(0) Parsing

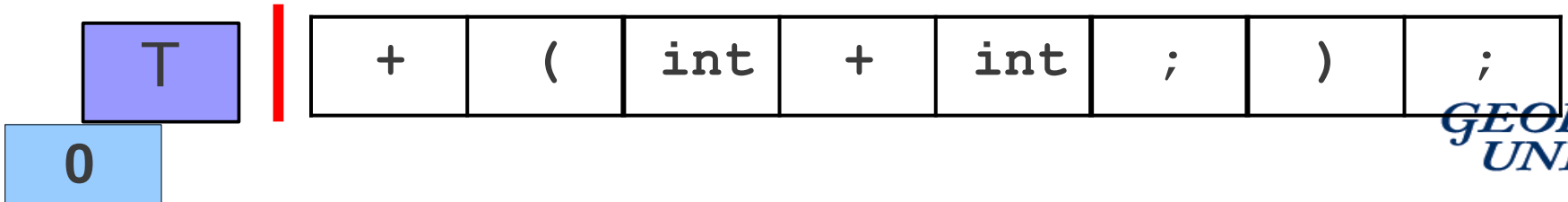




GEORGETOWN UNIVERSITY

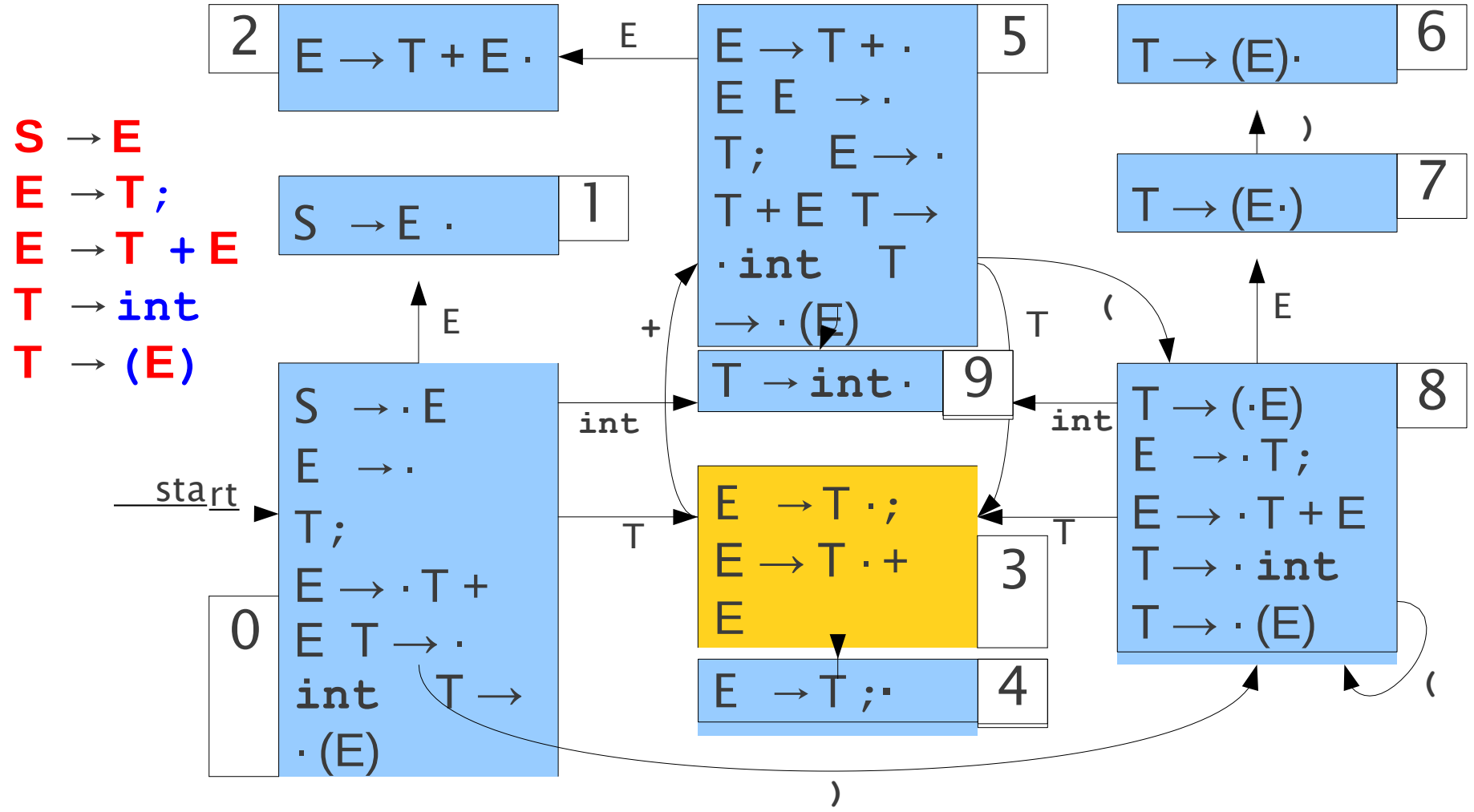
LR(0) Parsing

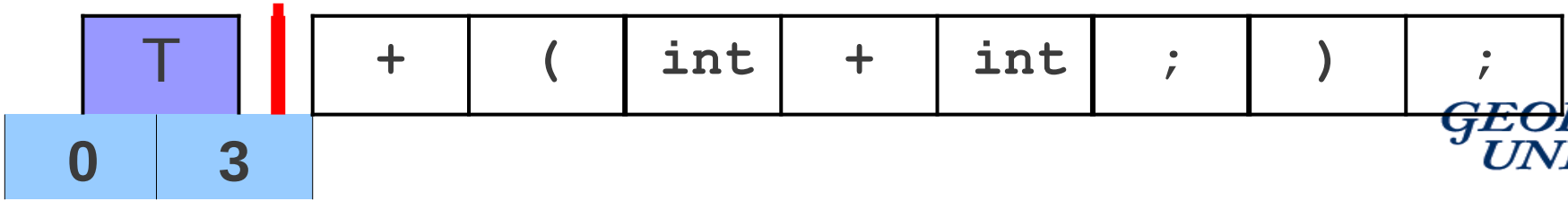




GEORGETOWN UNIVERSITY

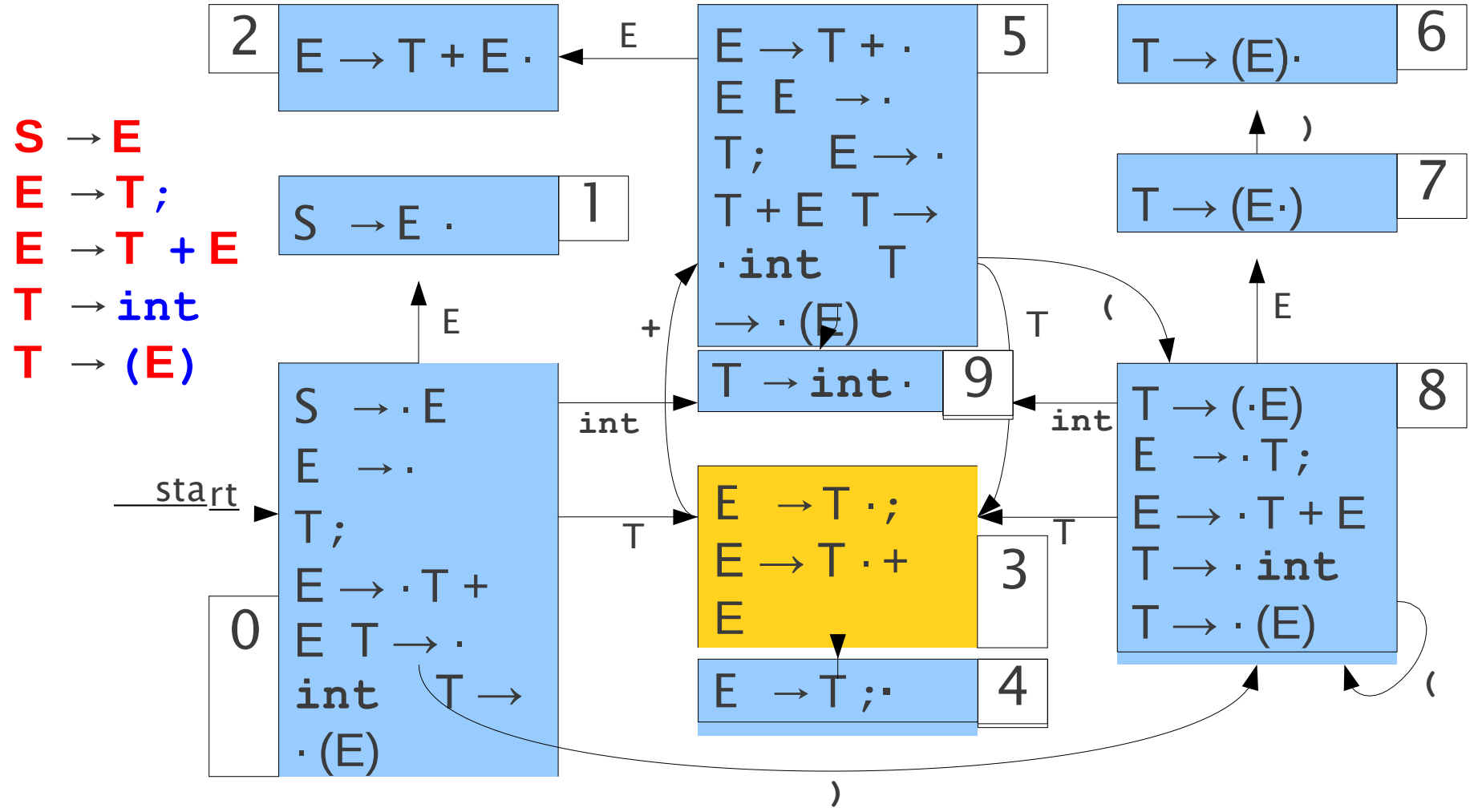
LR(0) Parsing

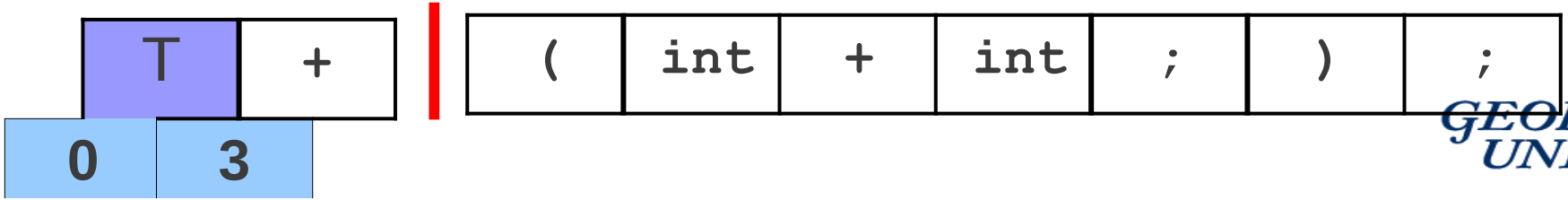




GEORGETOWN UNIVERSITY

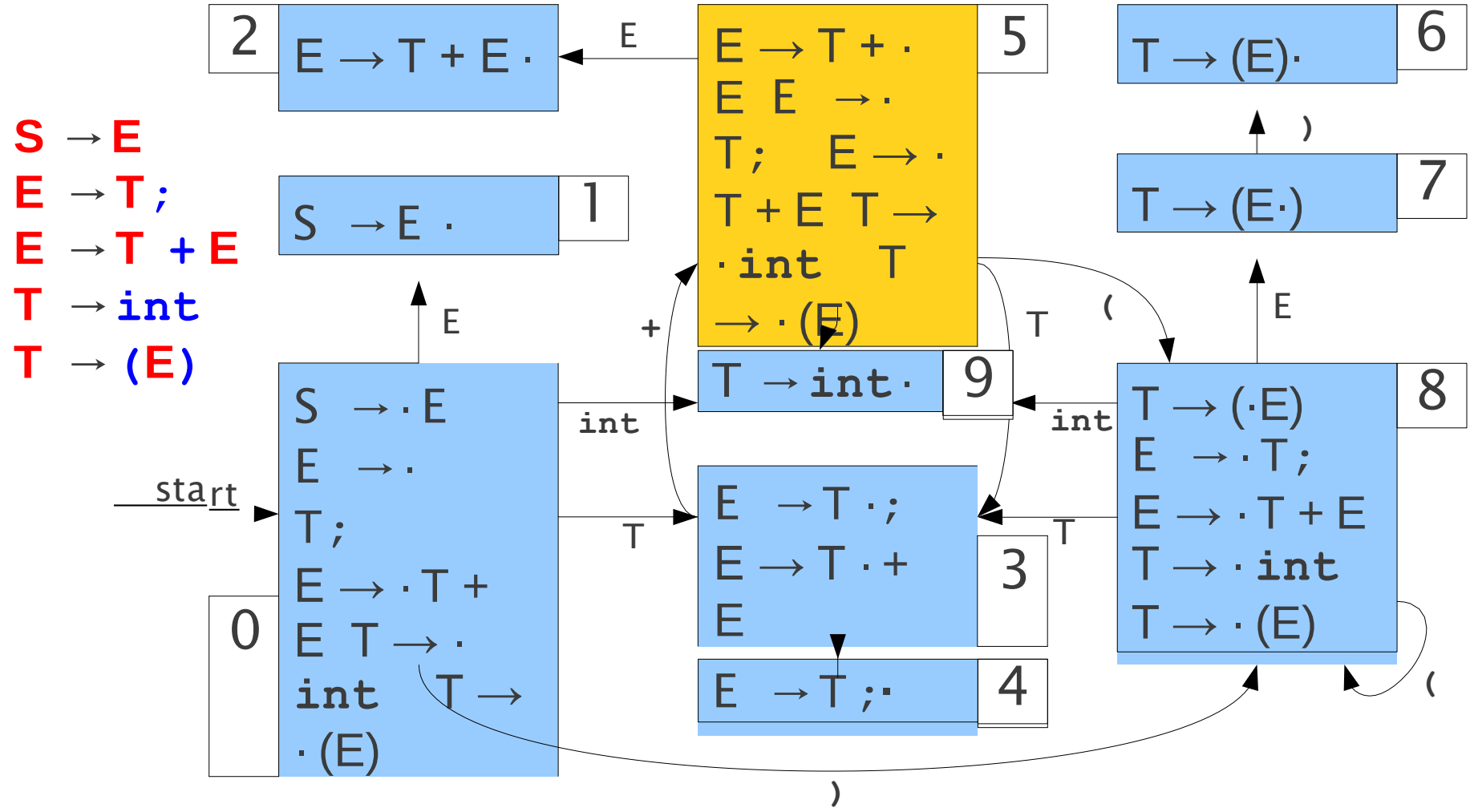
LR(0) Parsing

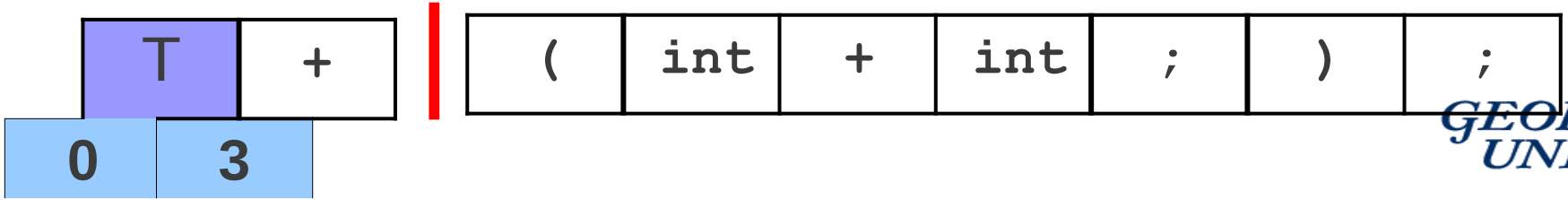




GEORGETOWN UNIVERSITY

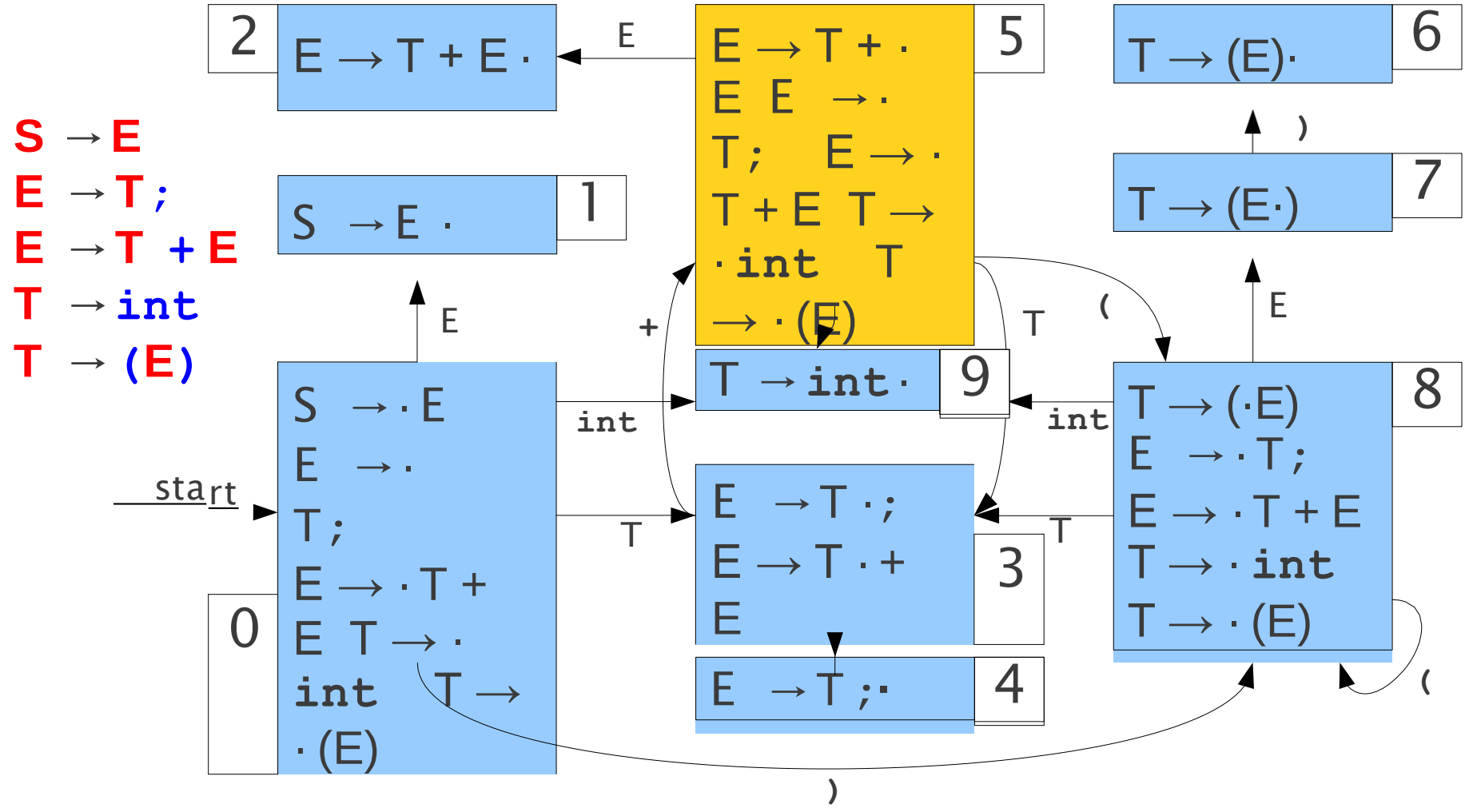
LR(0) Parsing

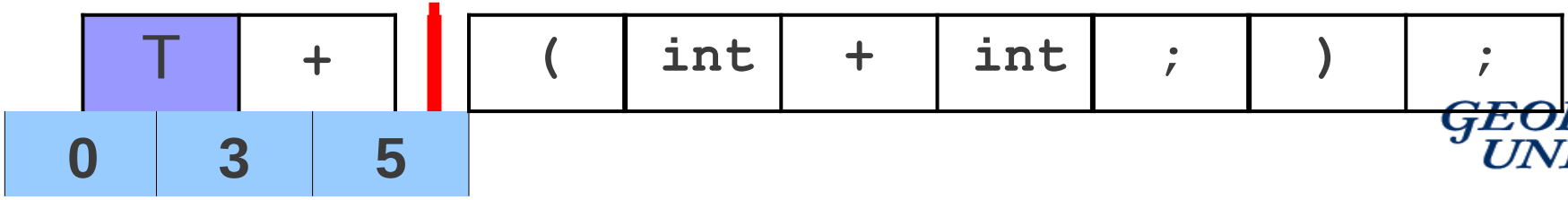




GEORGETOWN UNIVERSITY

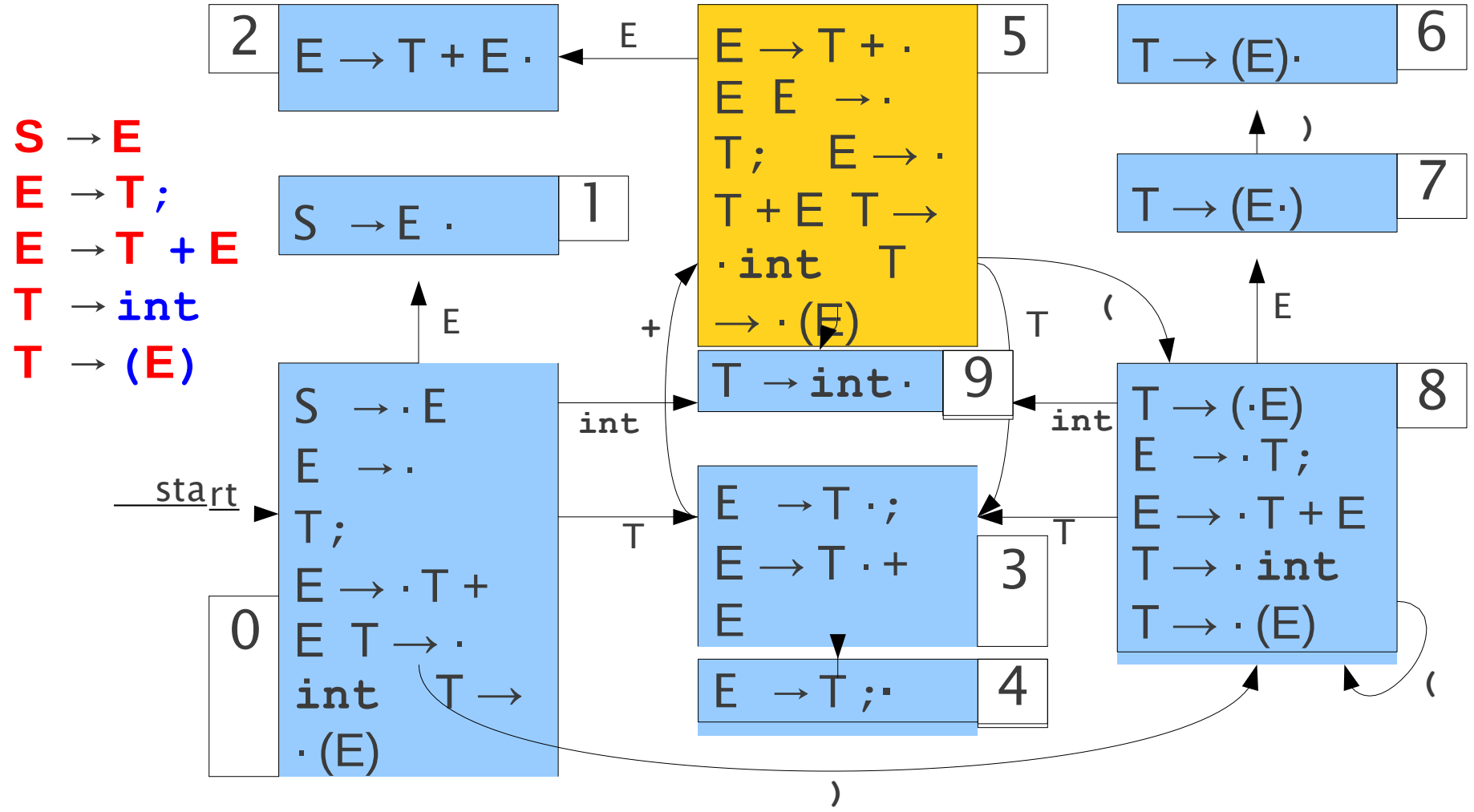
LR(0) Parsing

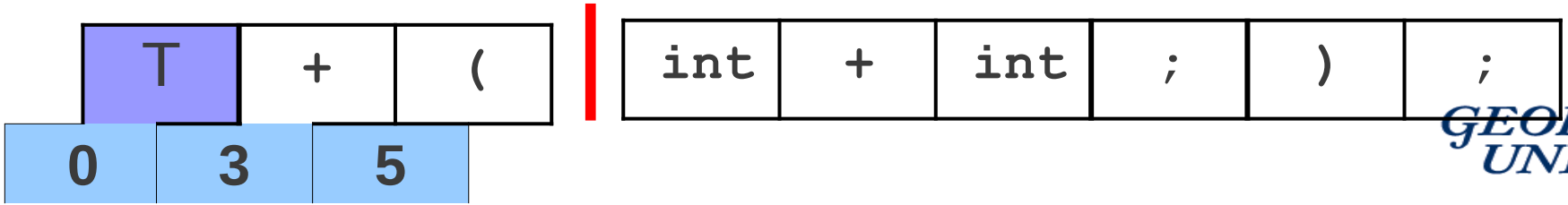




GEORGETOWN UNIVERSITY

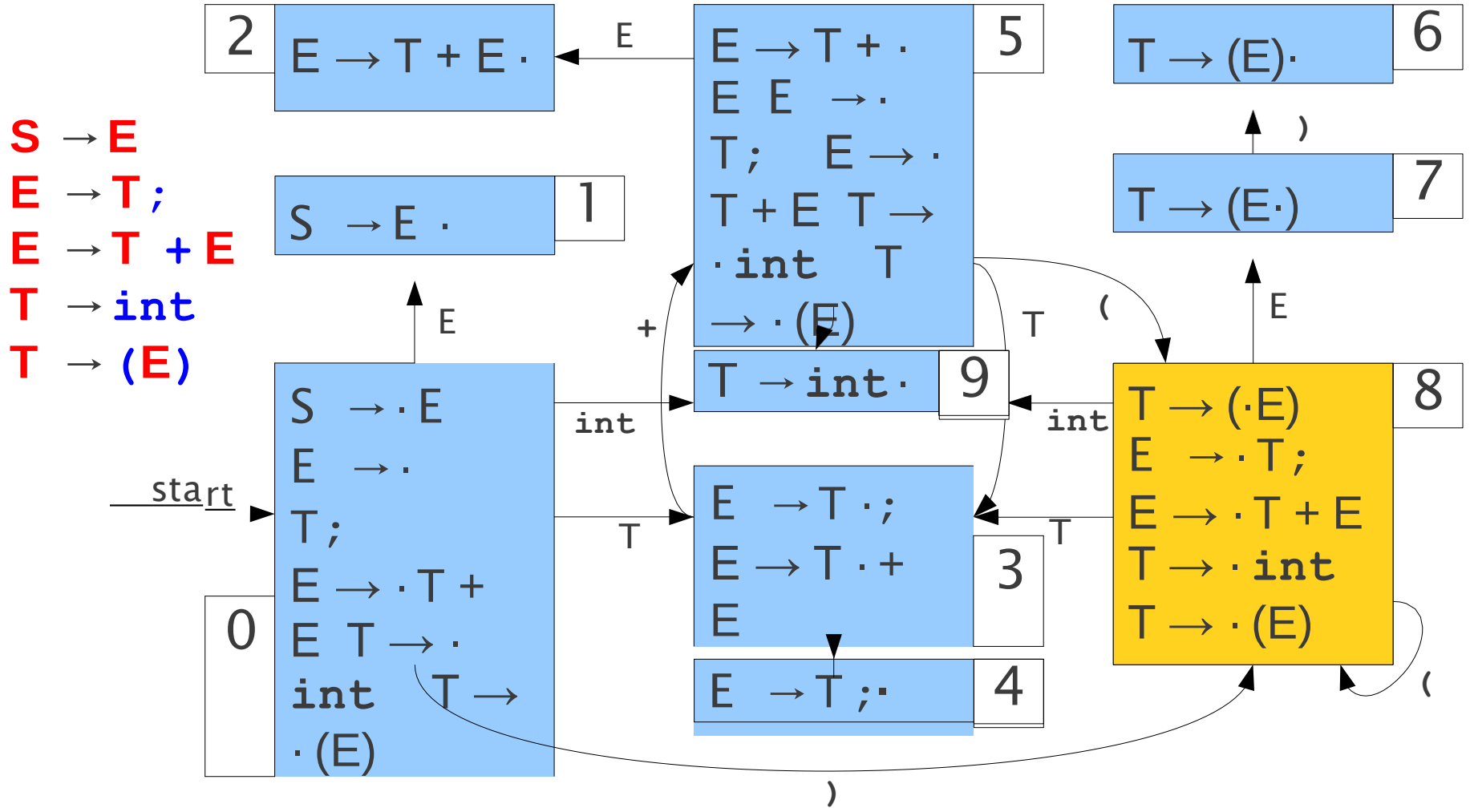
LR(0) Parsing

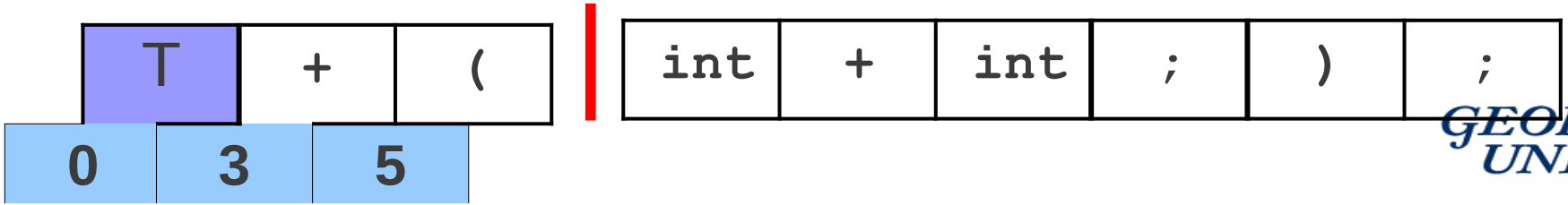




GEORGETOWN UNIVERSITY

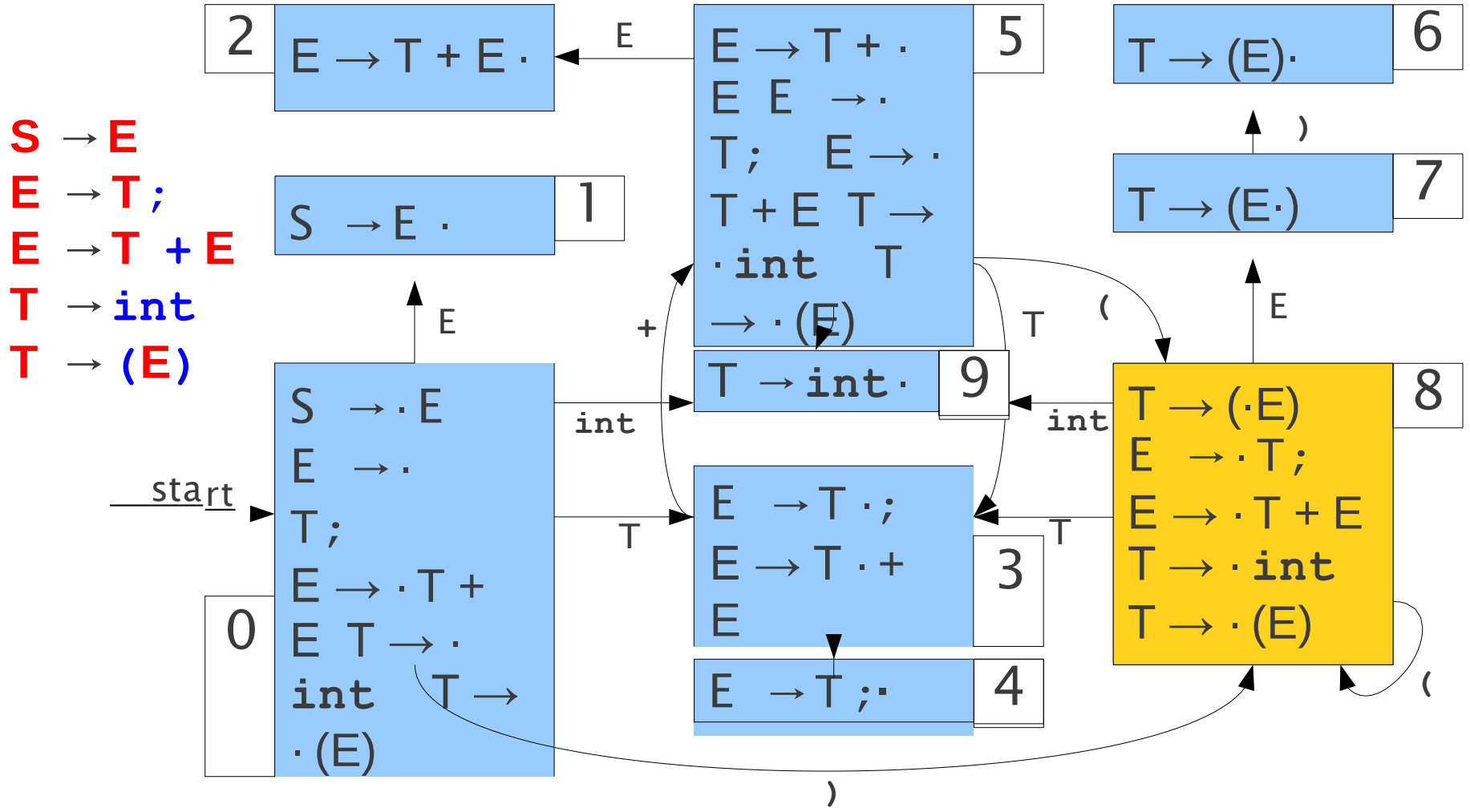
LR(0) Parsing

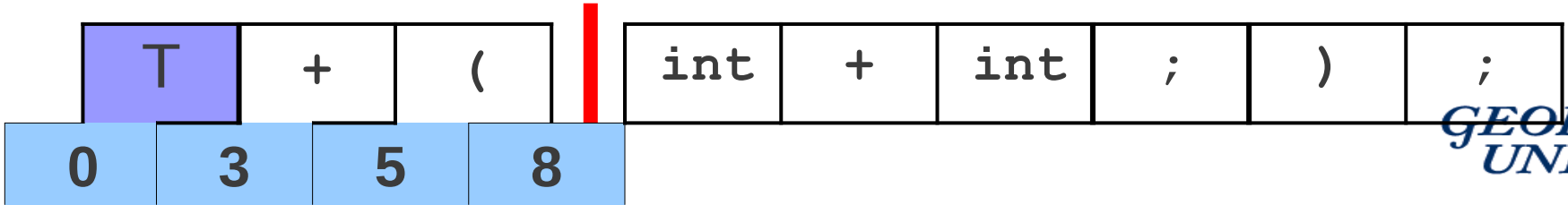




GEORGETOWN UNIVERSITY

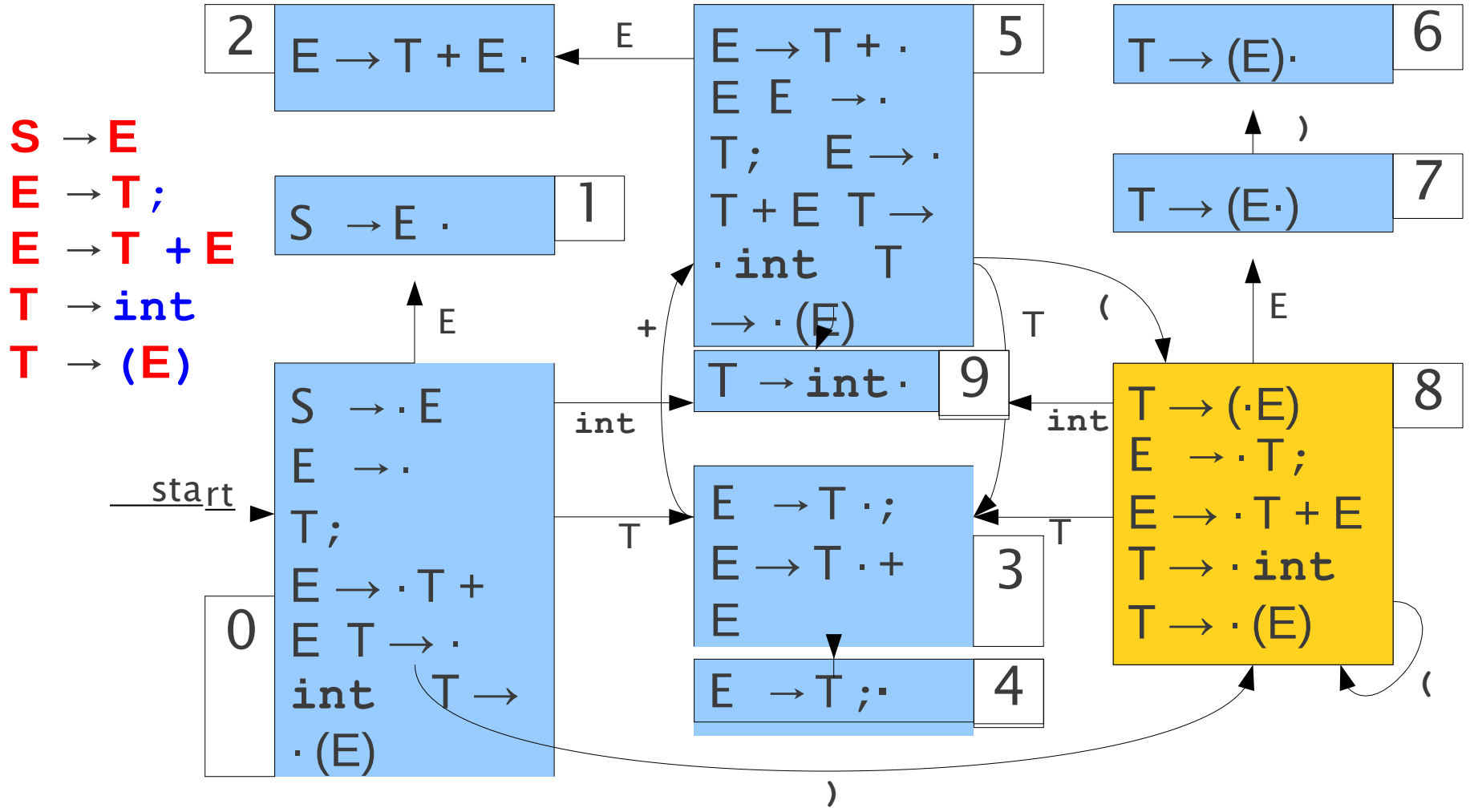
LR(0) Parsing

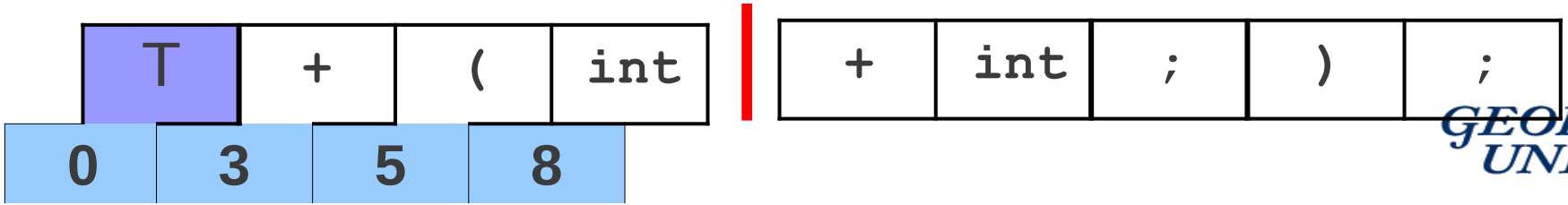




GEORGETOWN UNIVERSITY

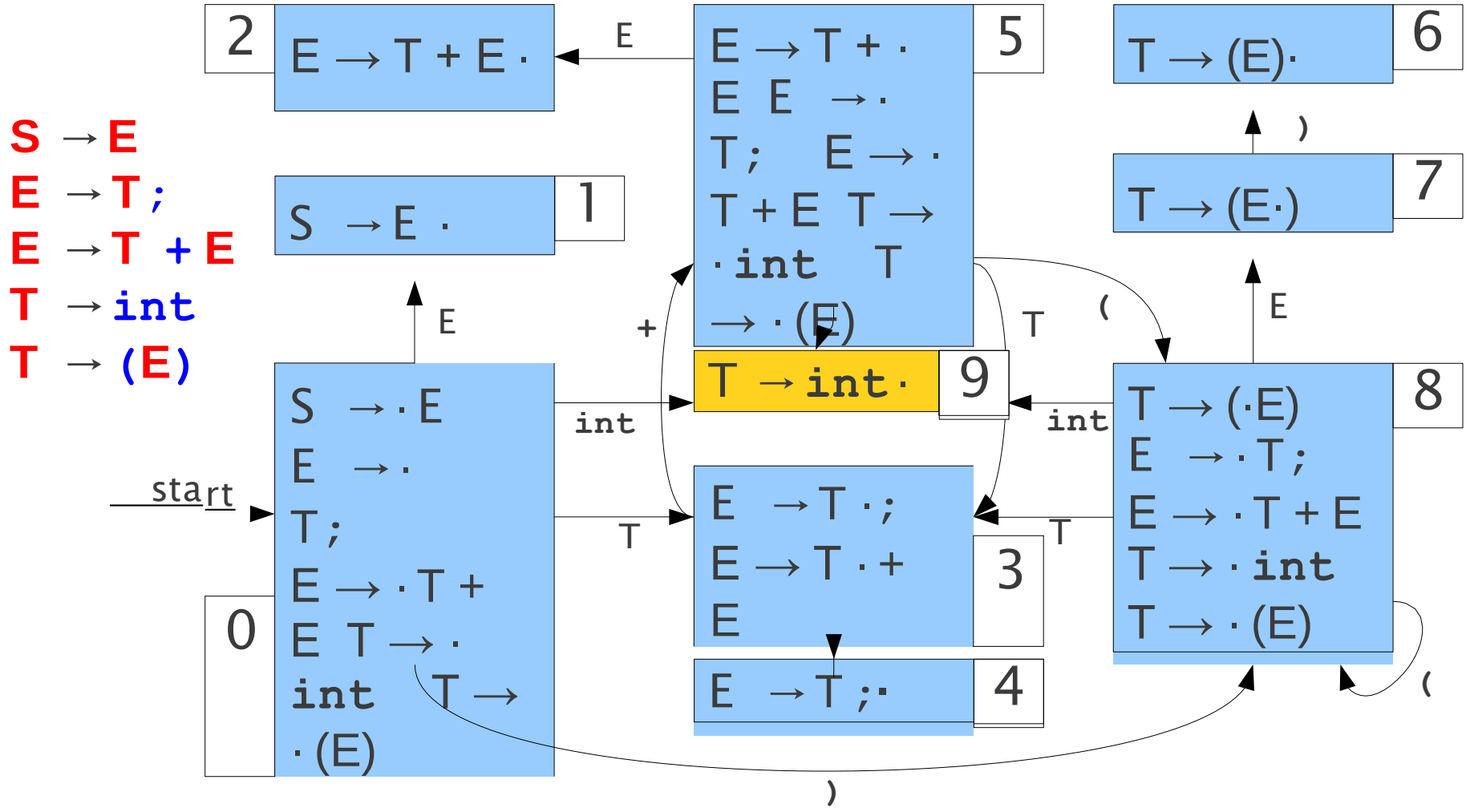
LR(0) Parsing

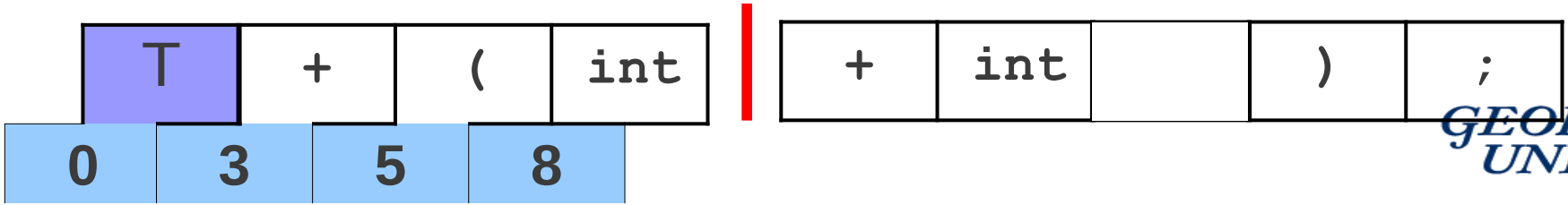




GEORGETOWN UNIVERSITY

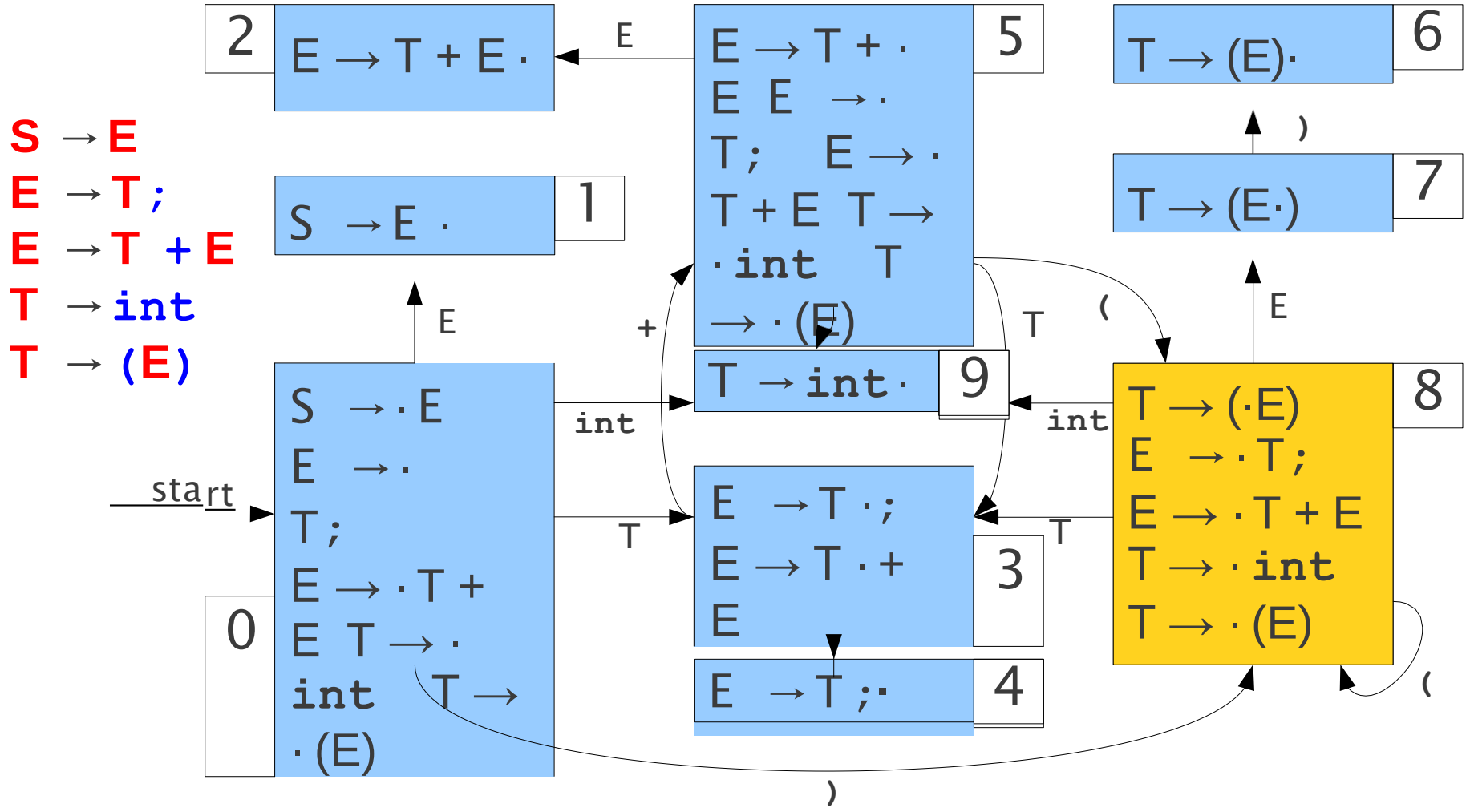
LR(0) Parsing

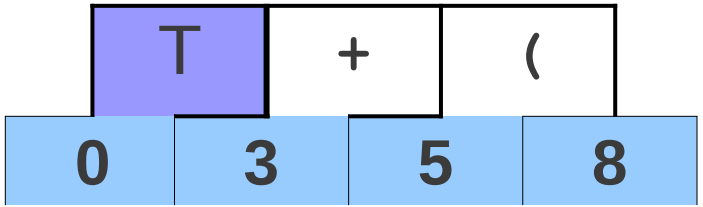


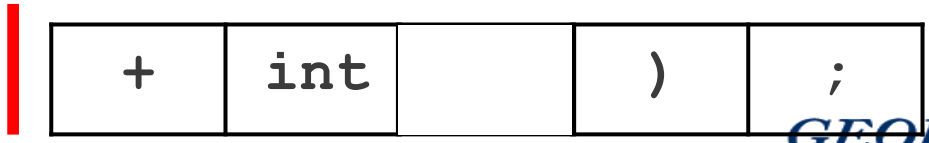


GEORGETOWN UNIVERSITY

LR(0) Parsing

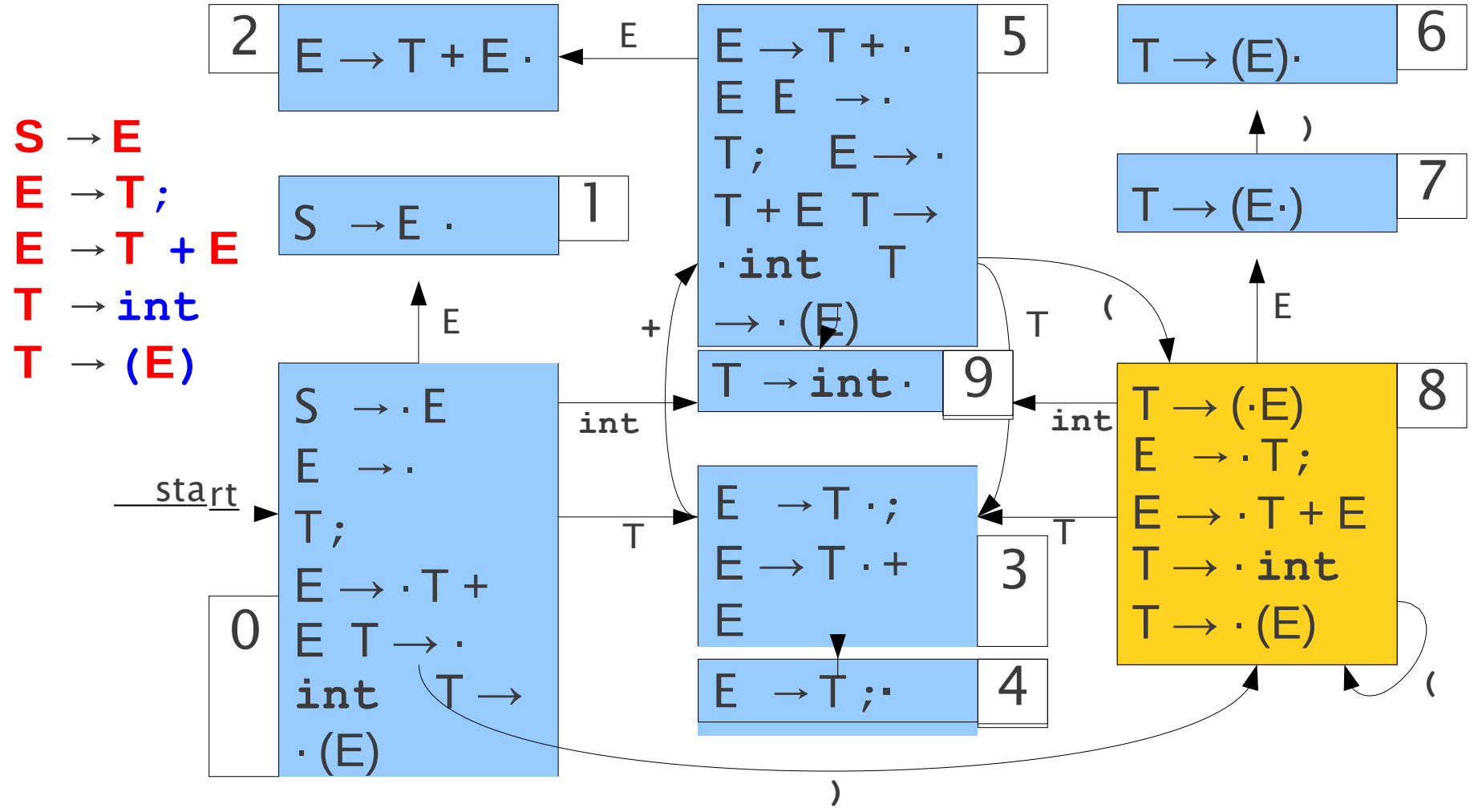


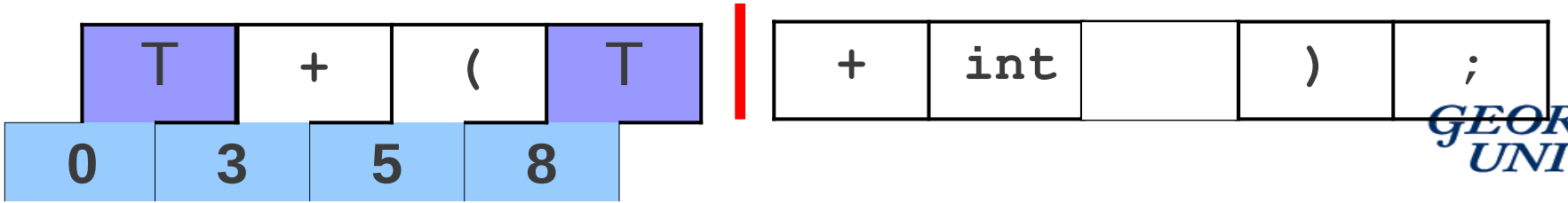




GEORGETOWN UNIVERSITY

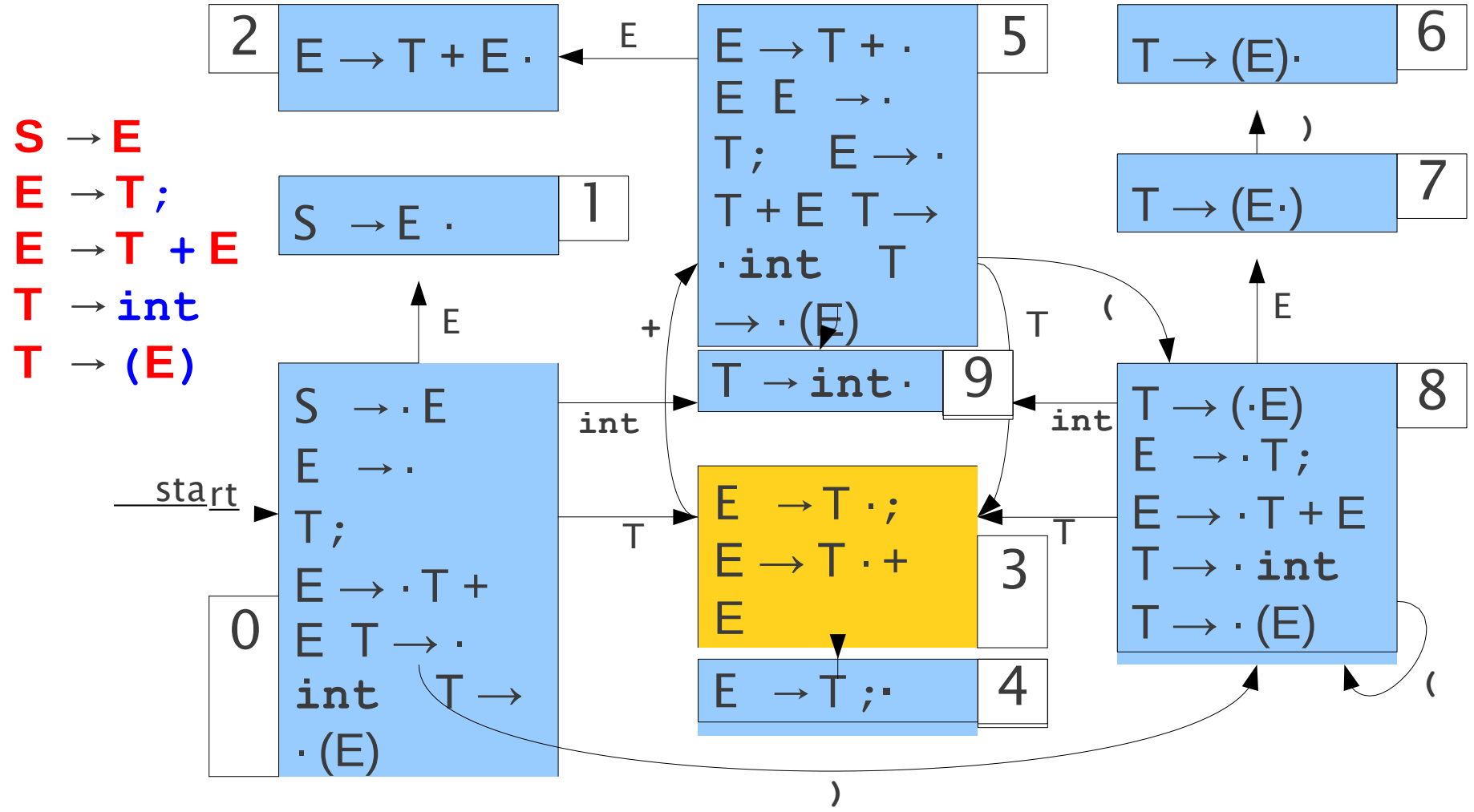
LR(0) Parsing

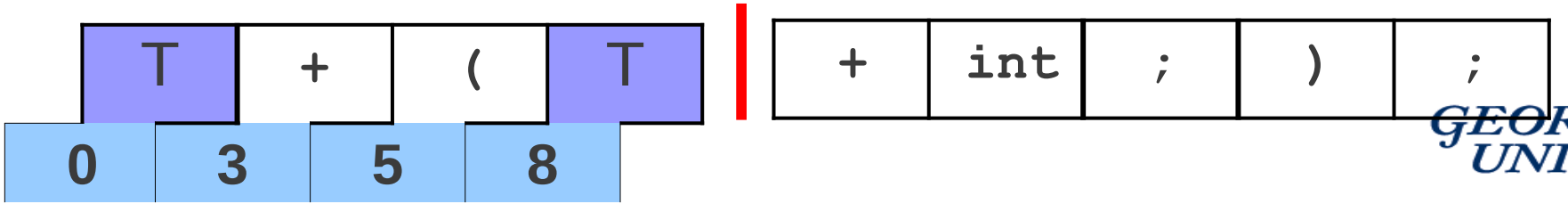




GEORGETOWN UNIVERSITY

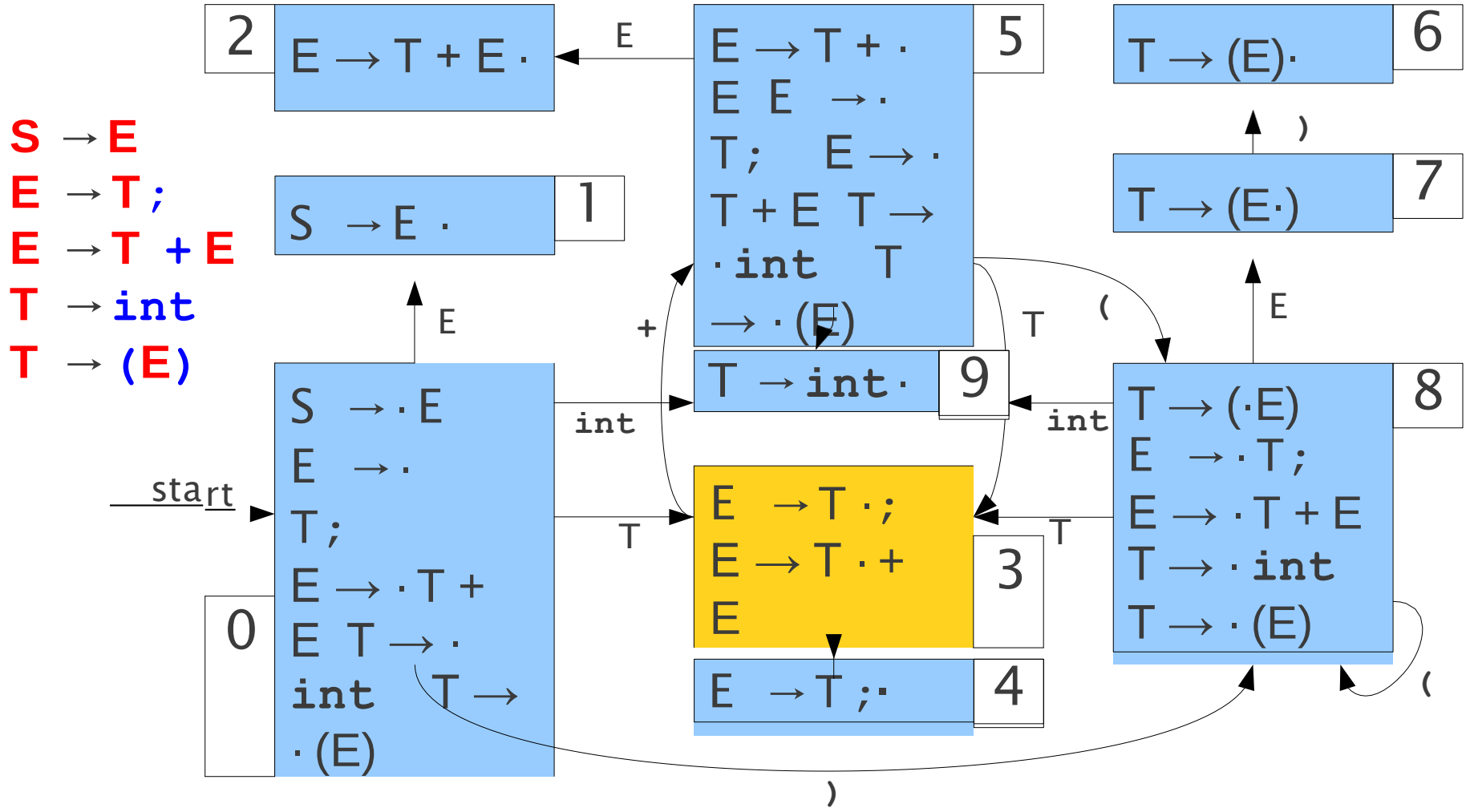
LR(0) Parsing

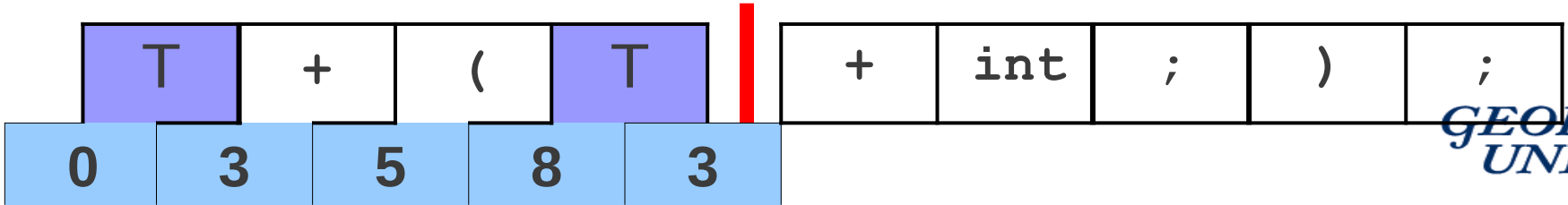




GEORGETOWN UNIVERSITY

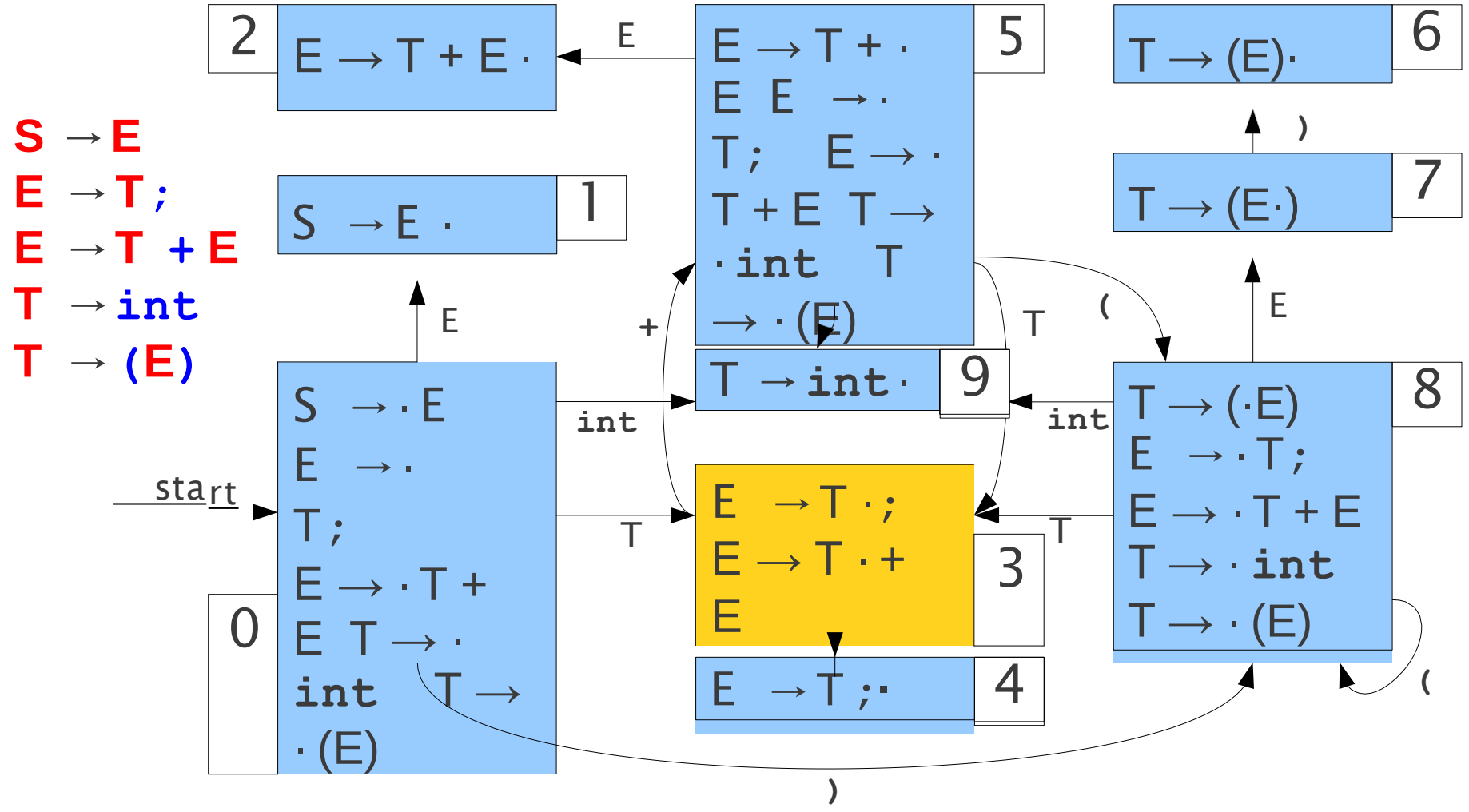
LR(0) Parsing

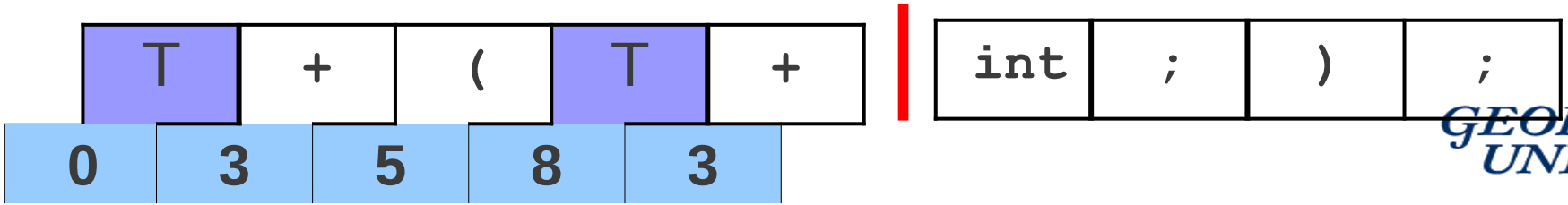




GEORGETOWN UNIVERSITY

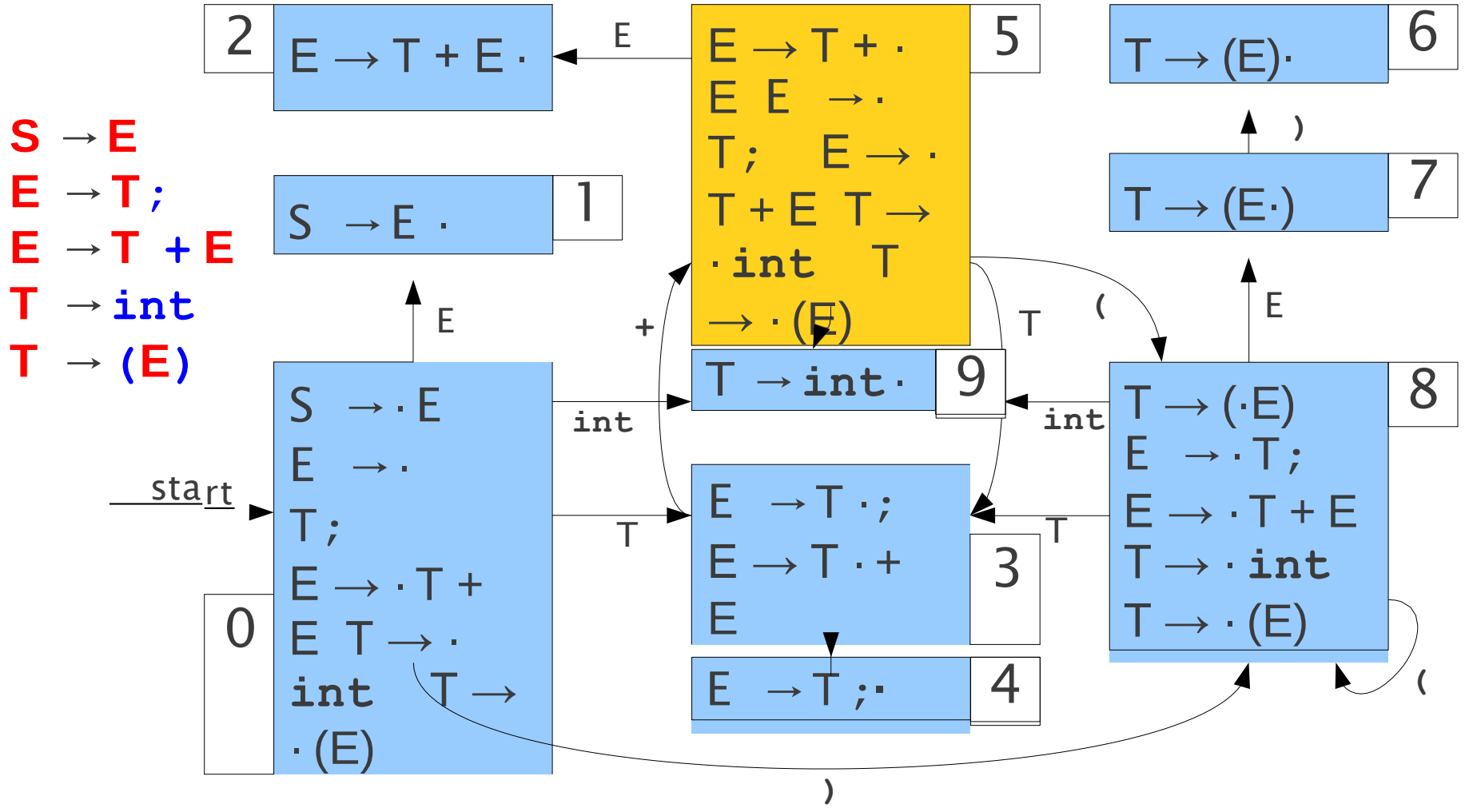
LR(0) Parsing

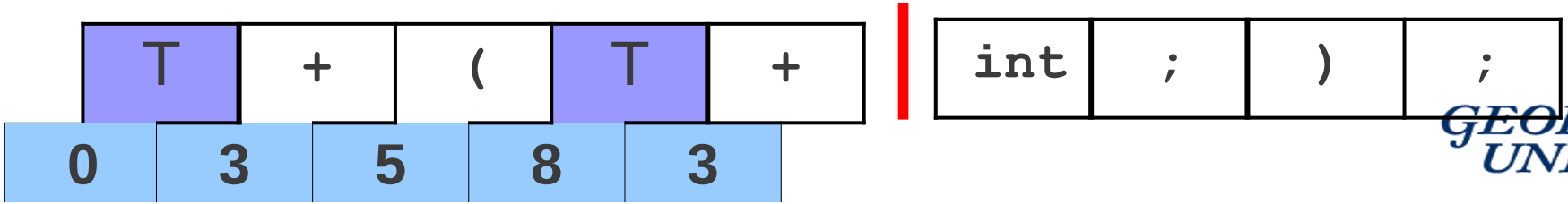




GEORGETOWN UNIVERSITY

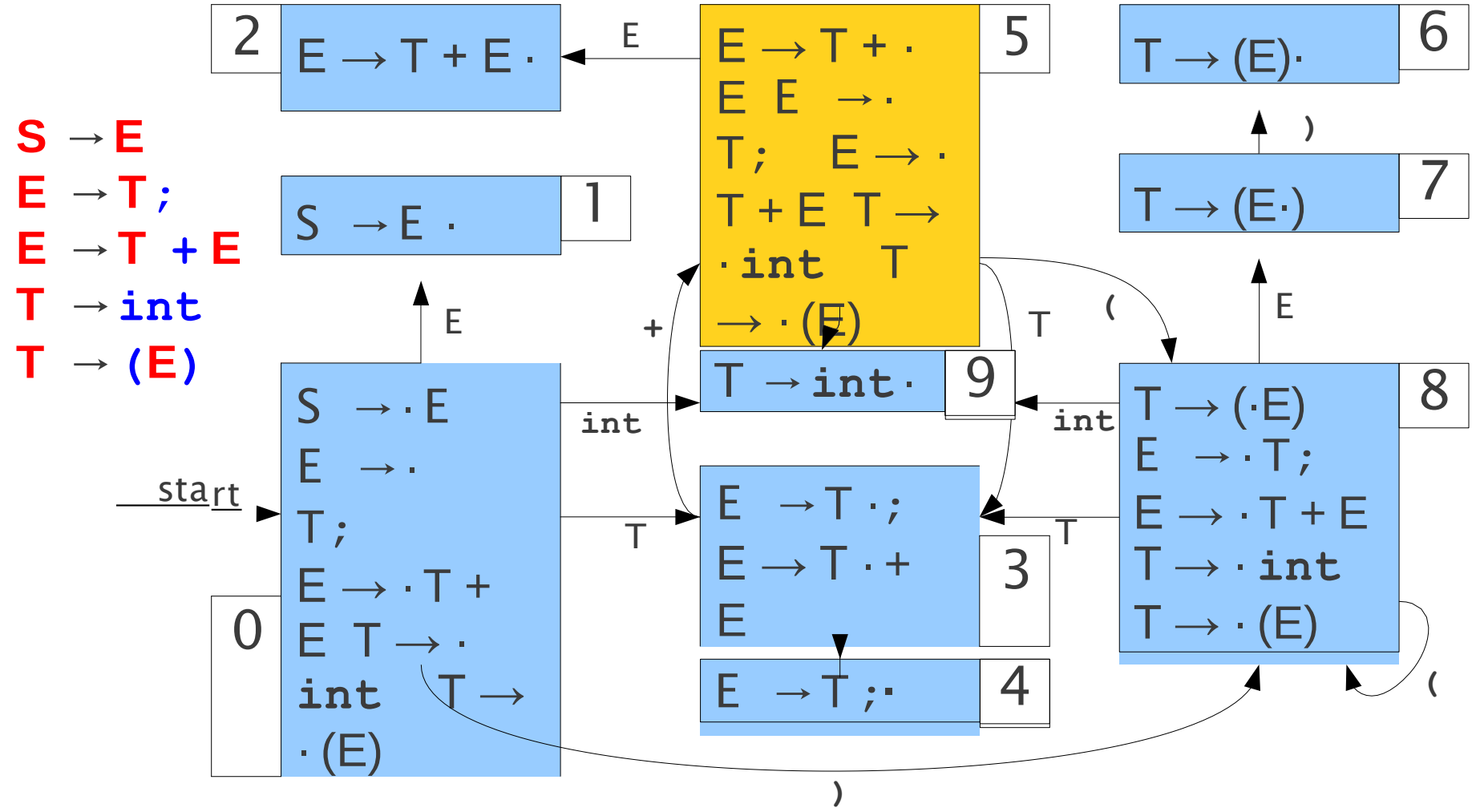
LR(0) Parsing

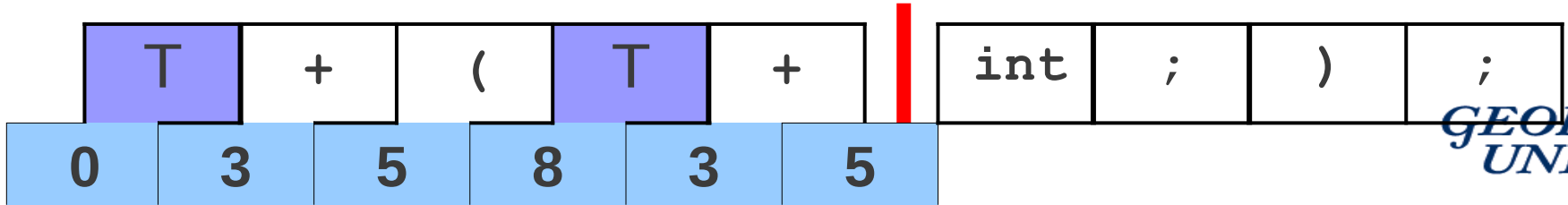




GEORGETOWN UNIVERSITY

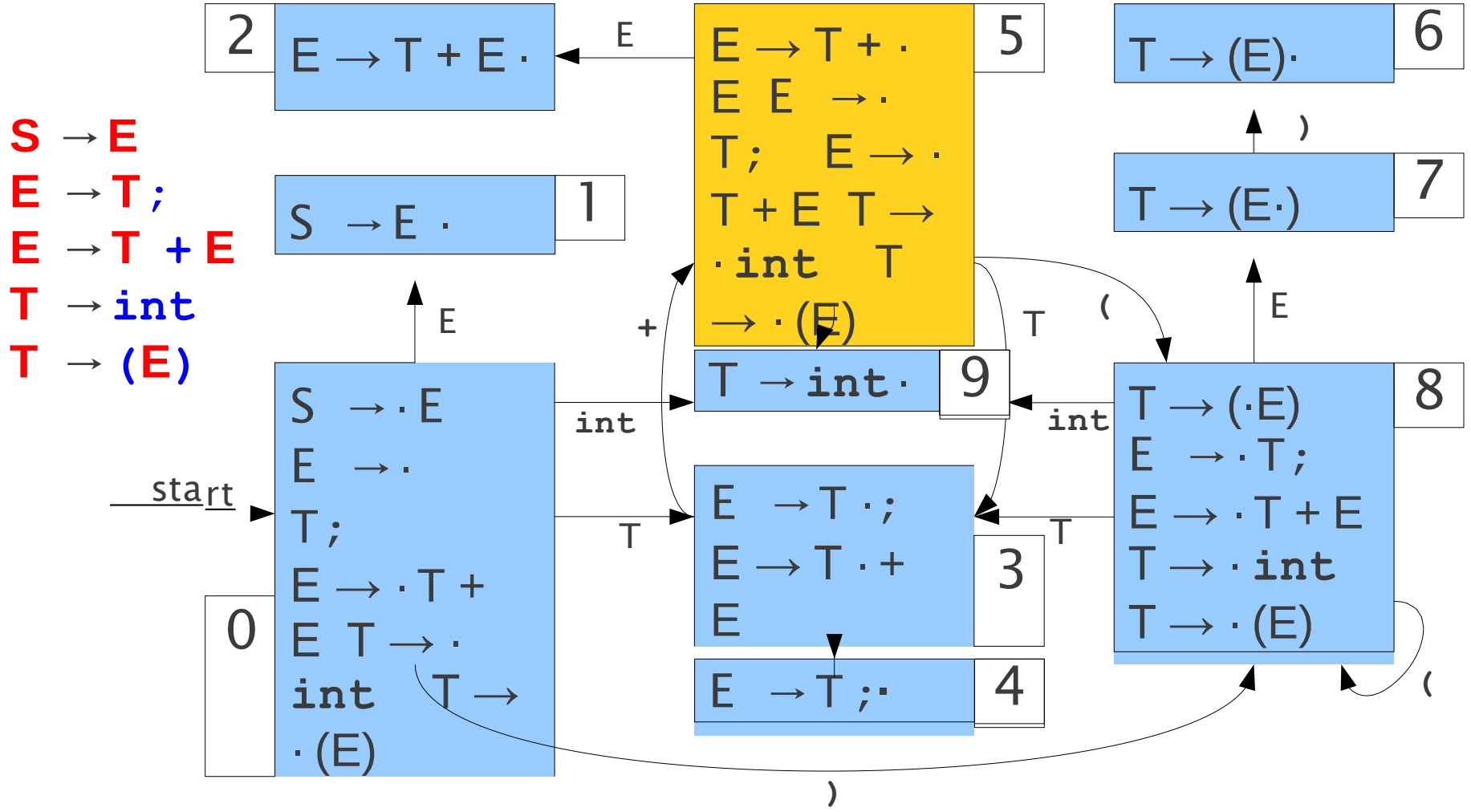
LR(0) Parsing





GEORGETOWN UNIVERSITY

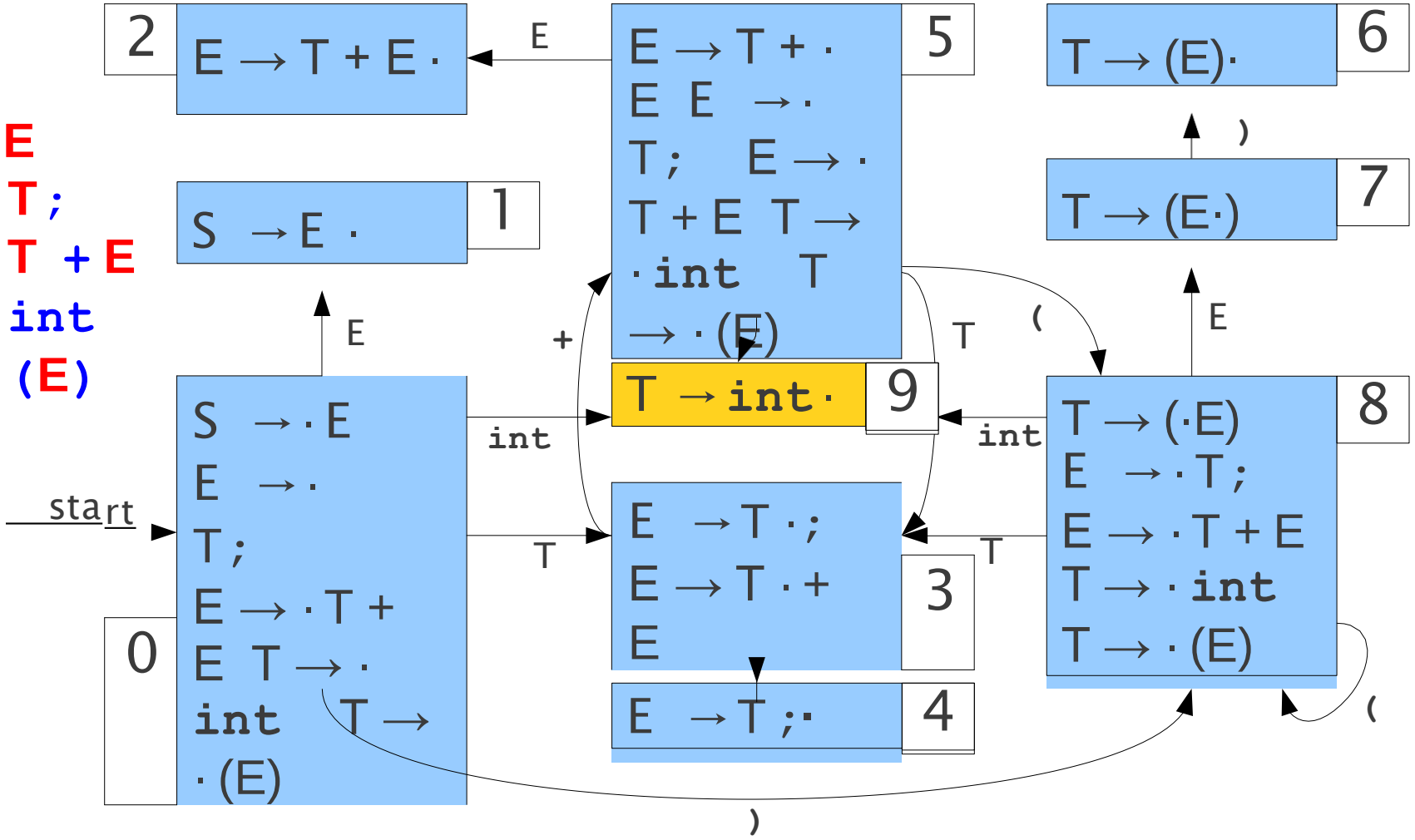
LR(0) Parsing



GEORGETOWN UNIVERSITY

LR(0) Parsing

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

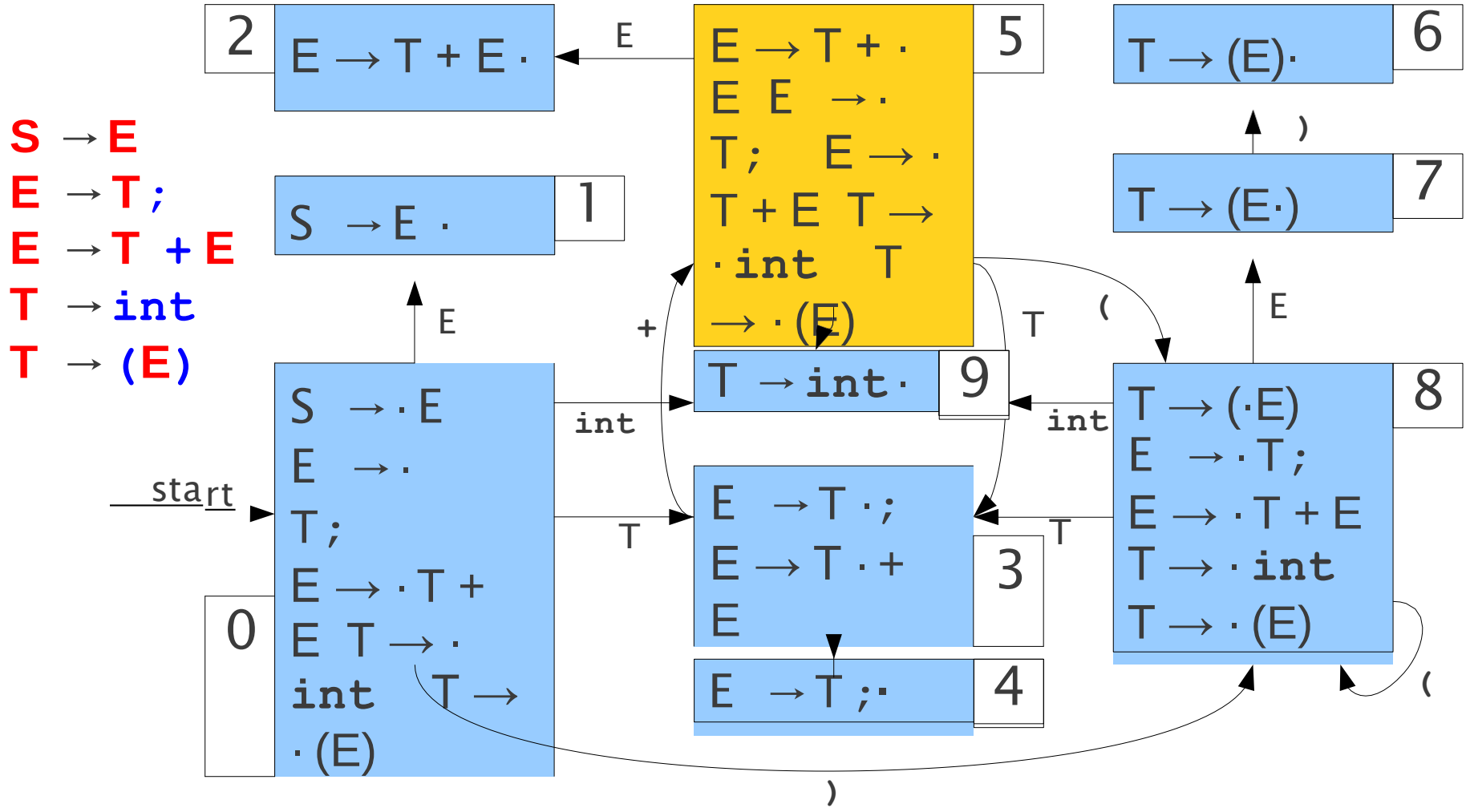


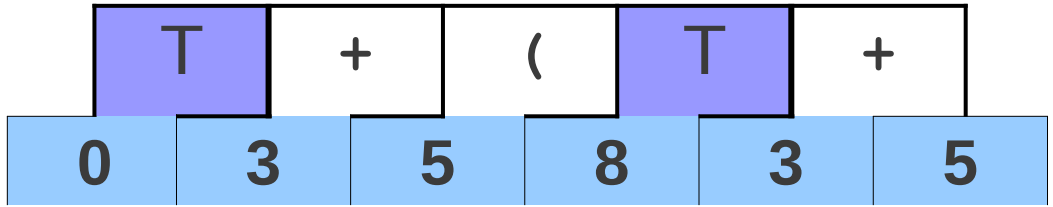
T	+	(	T	+	int
0	3	5	8	3	5

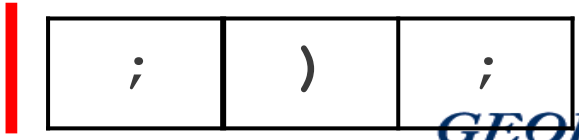
;	)	;

GEORGETOWN UNIVERSITY

LR(0) Parsing

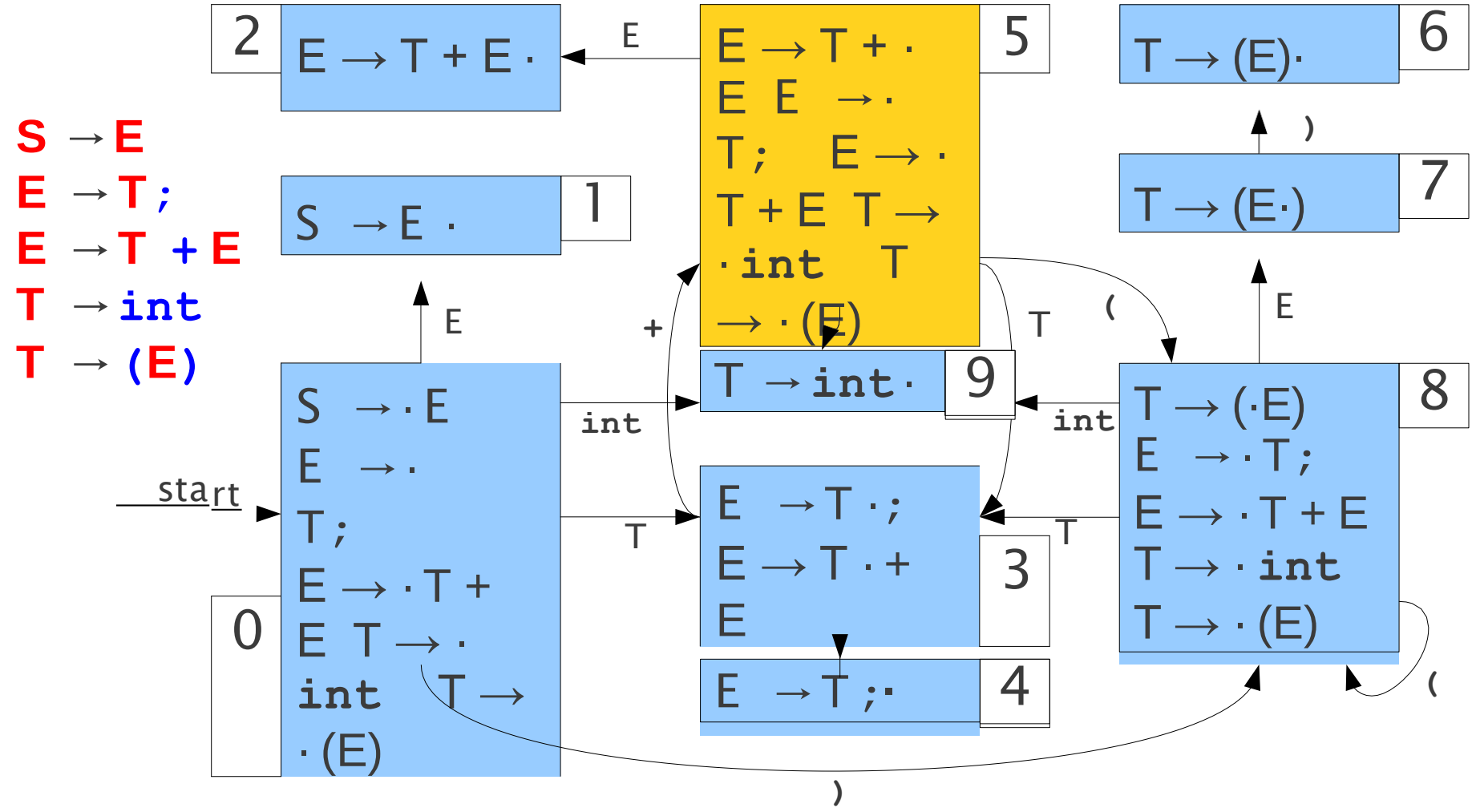


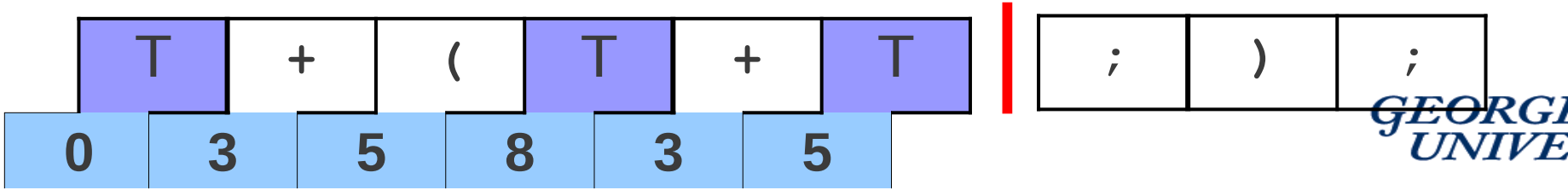




GEORGETOWN UNIVERSITY

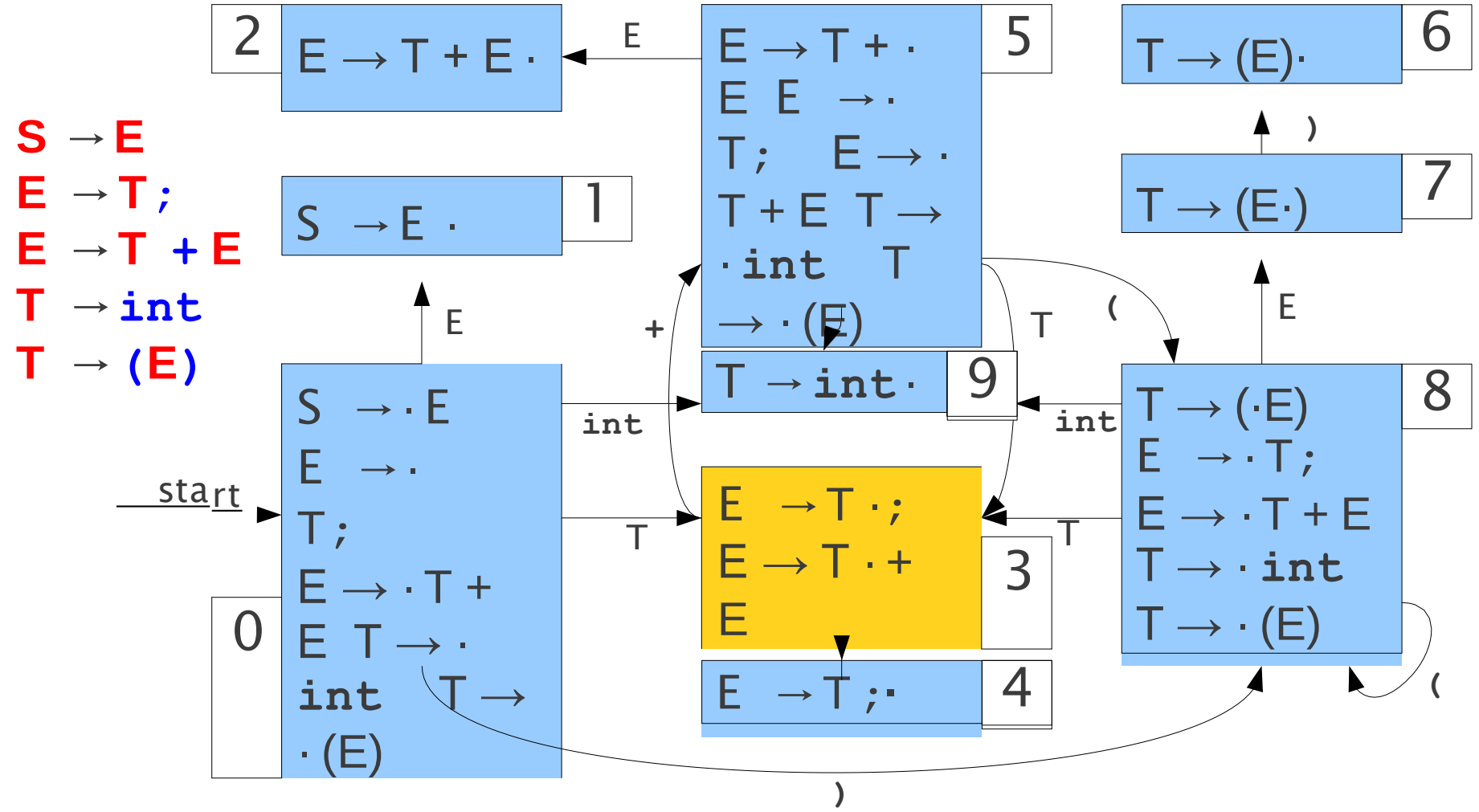
LR(0) Parsing

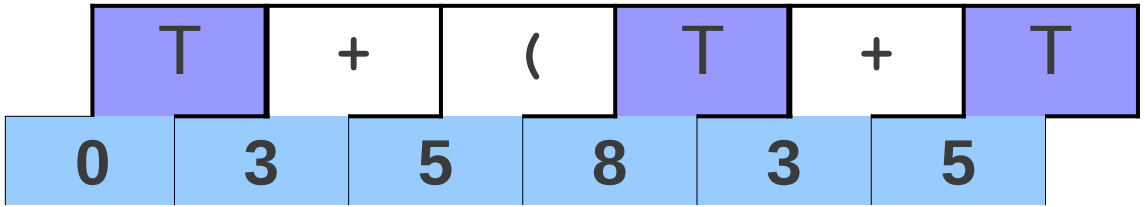




GEORGETOWN UNIVERSITY

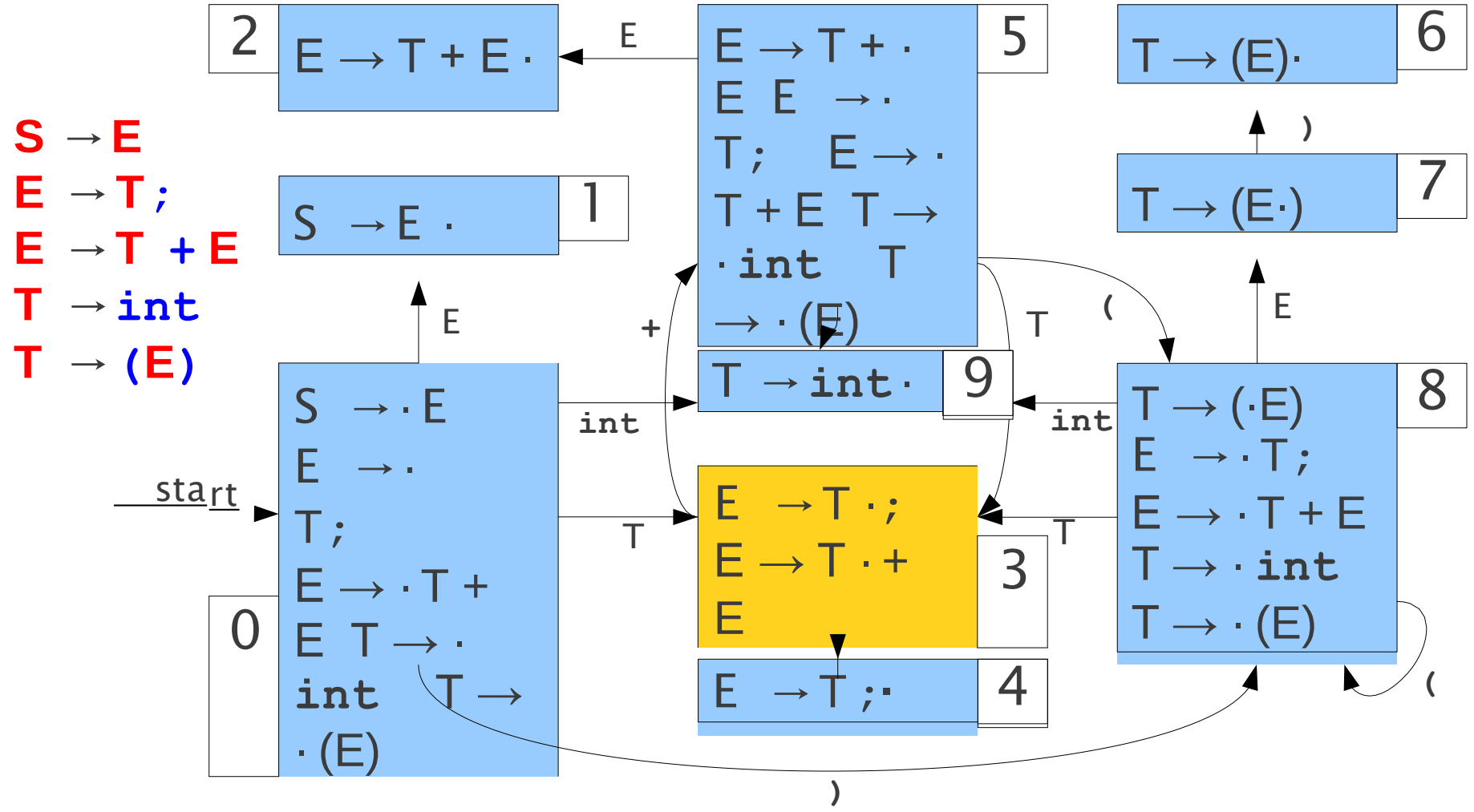
LR(0) Parsing

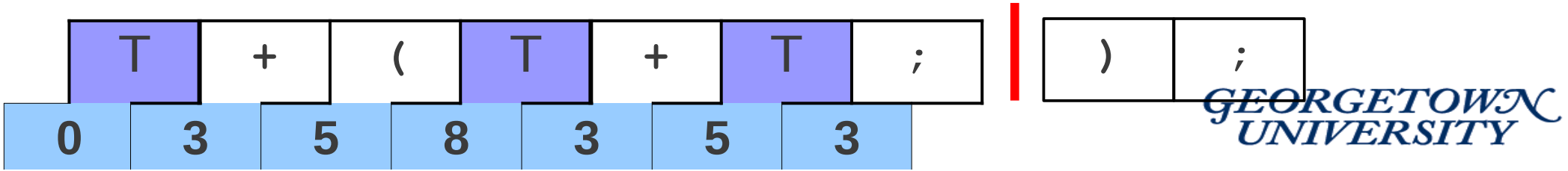




GEORGETOWN UNIVERSITY

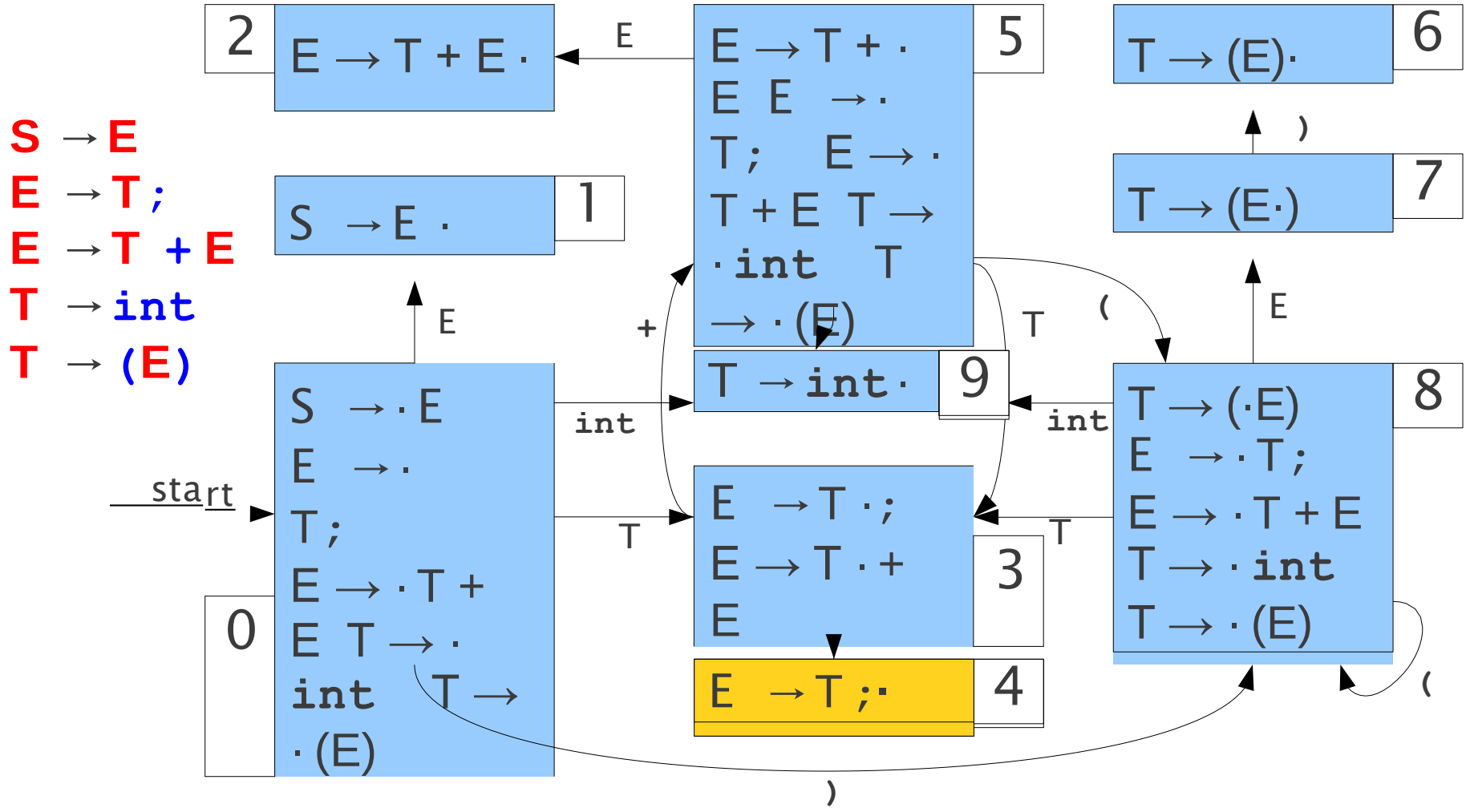
LR(0) Parsing

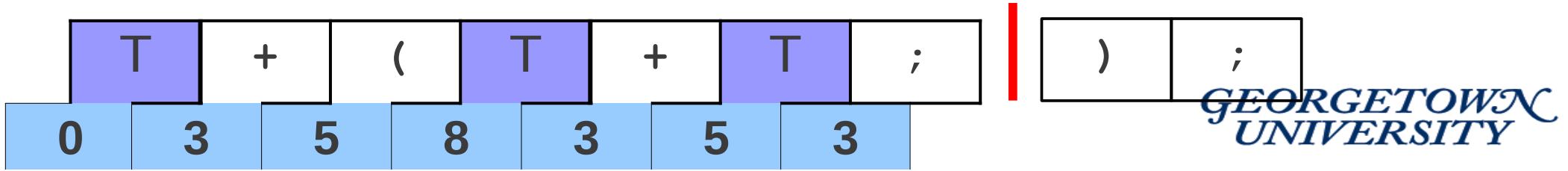




GEORGETOWN UNIVERSITY

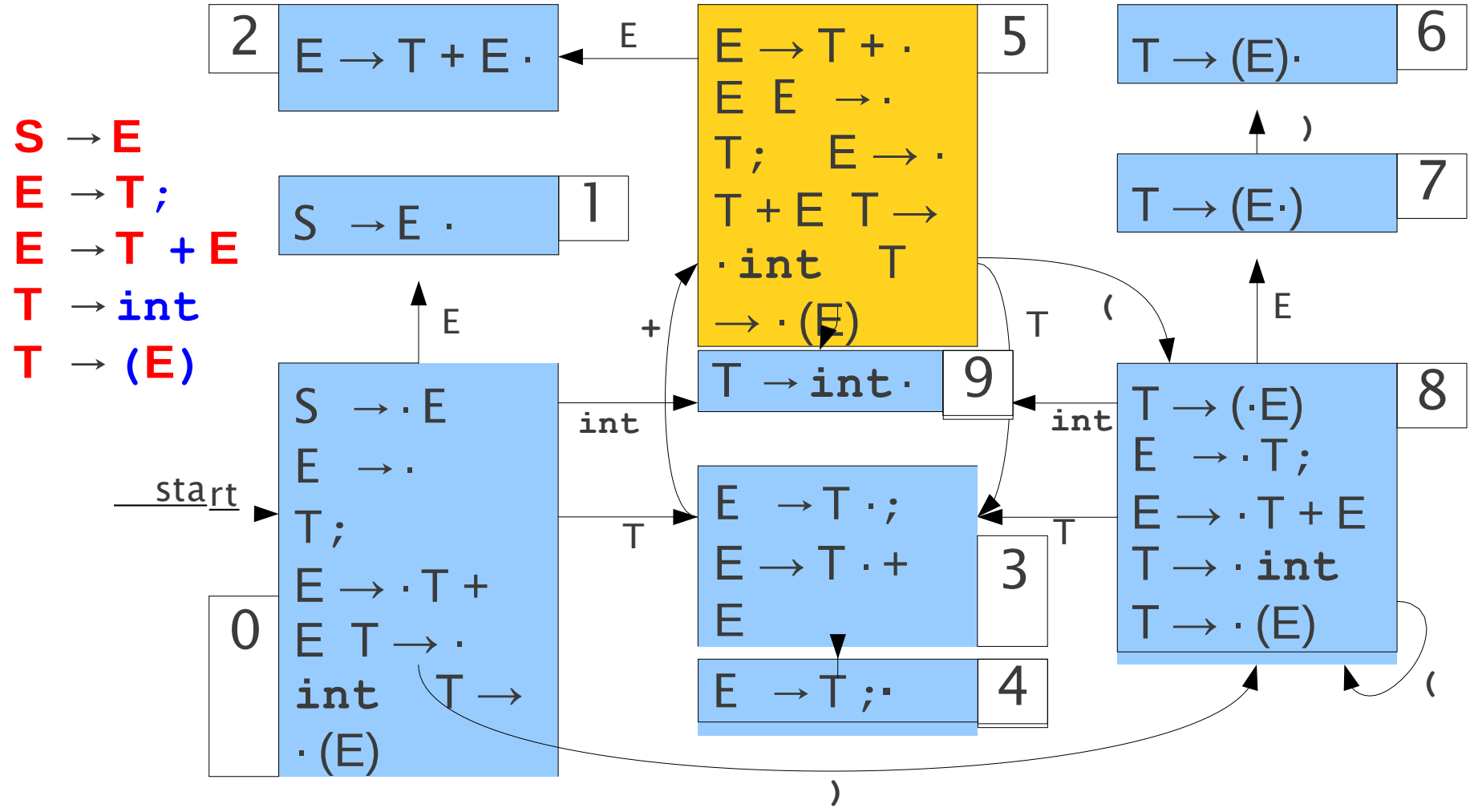
LR(0) Parsing



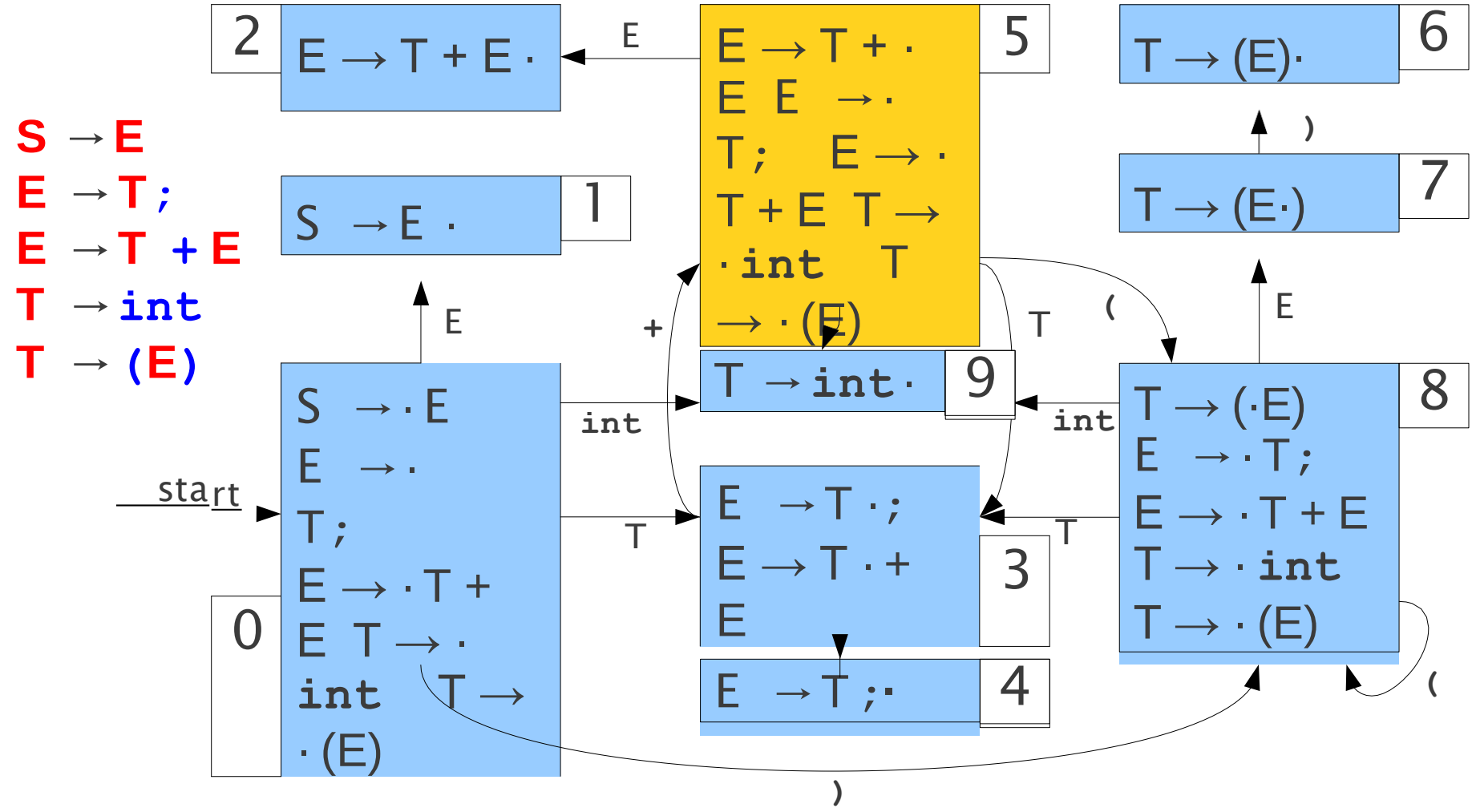


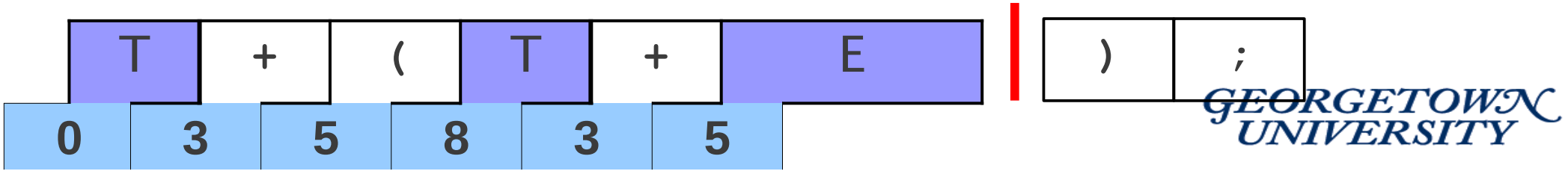
GEORGETOWN UNIVERSITY

LR(0) Parsing



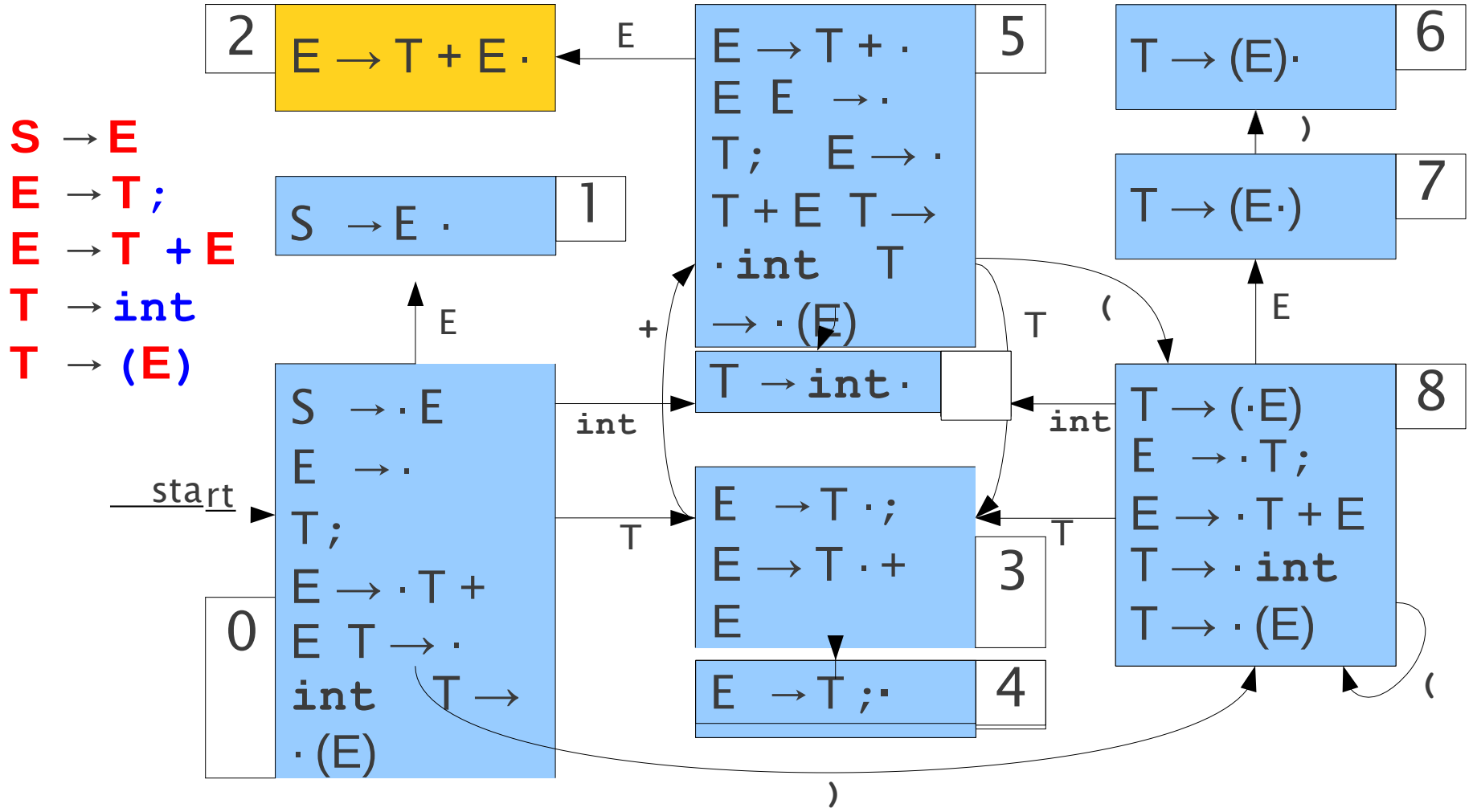
LR(0) Parsing

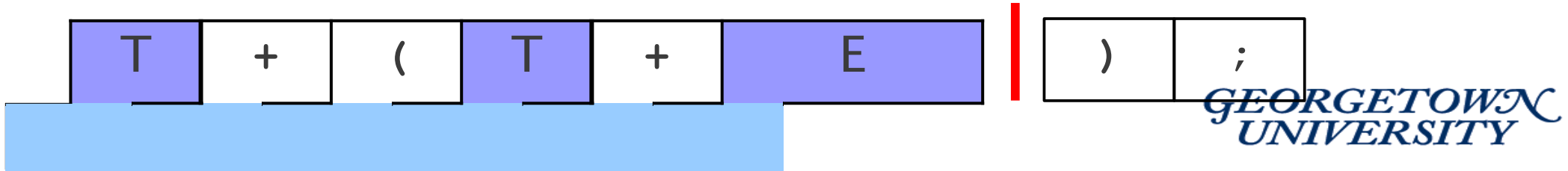




GEORGETOWN UNIVERSITY

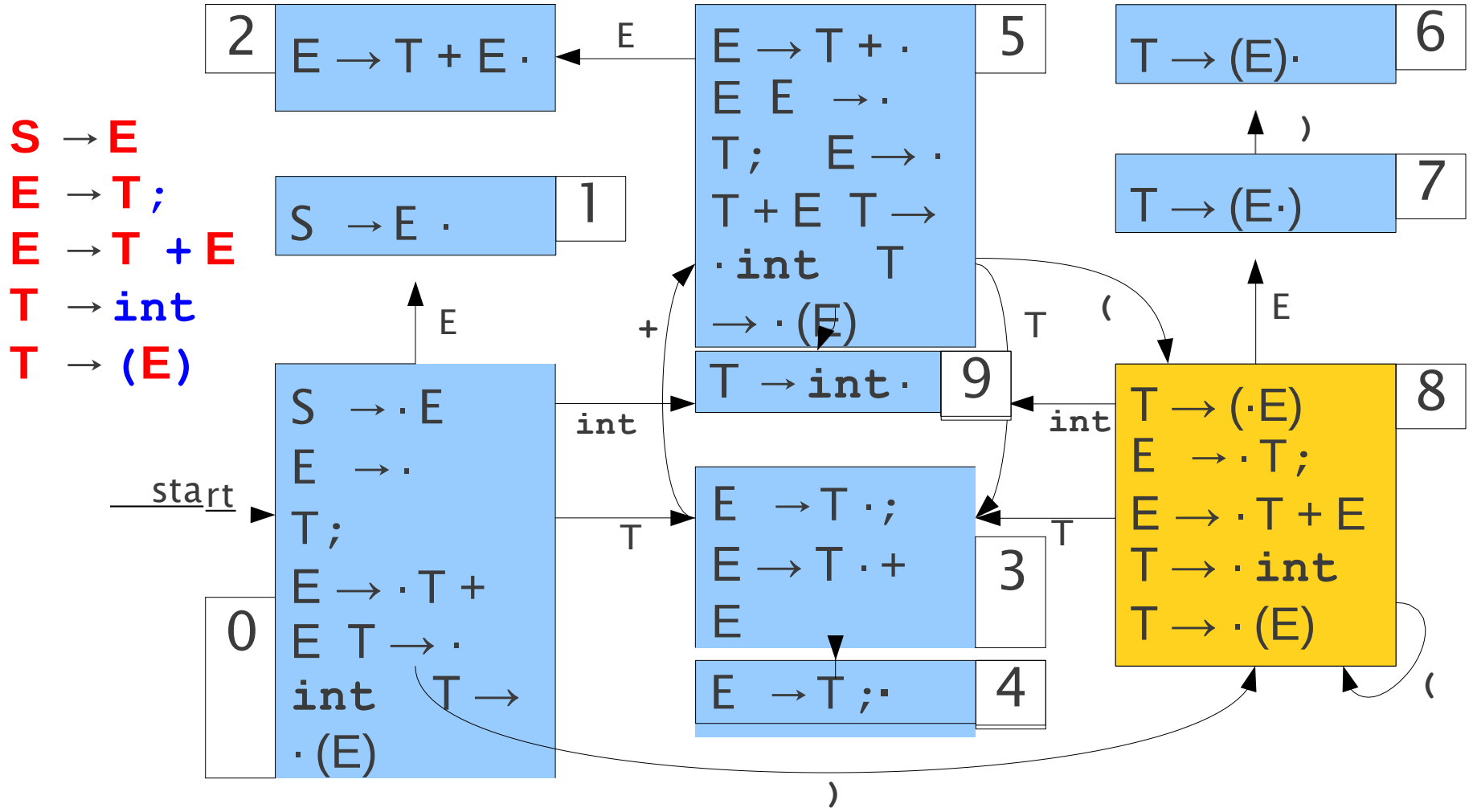
LR(0) Parsing





GEORGETOWN UNIVERSITY

LR(0) Parsing

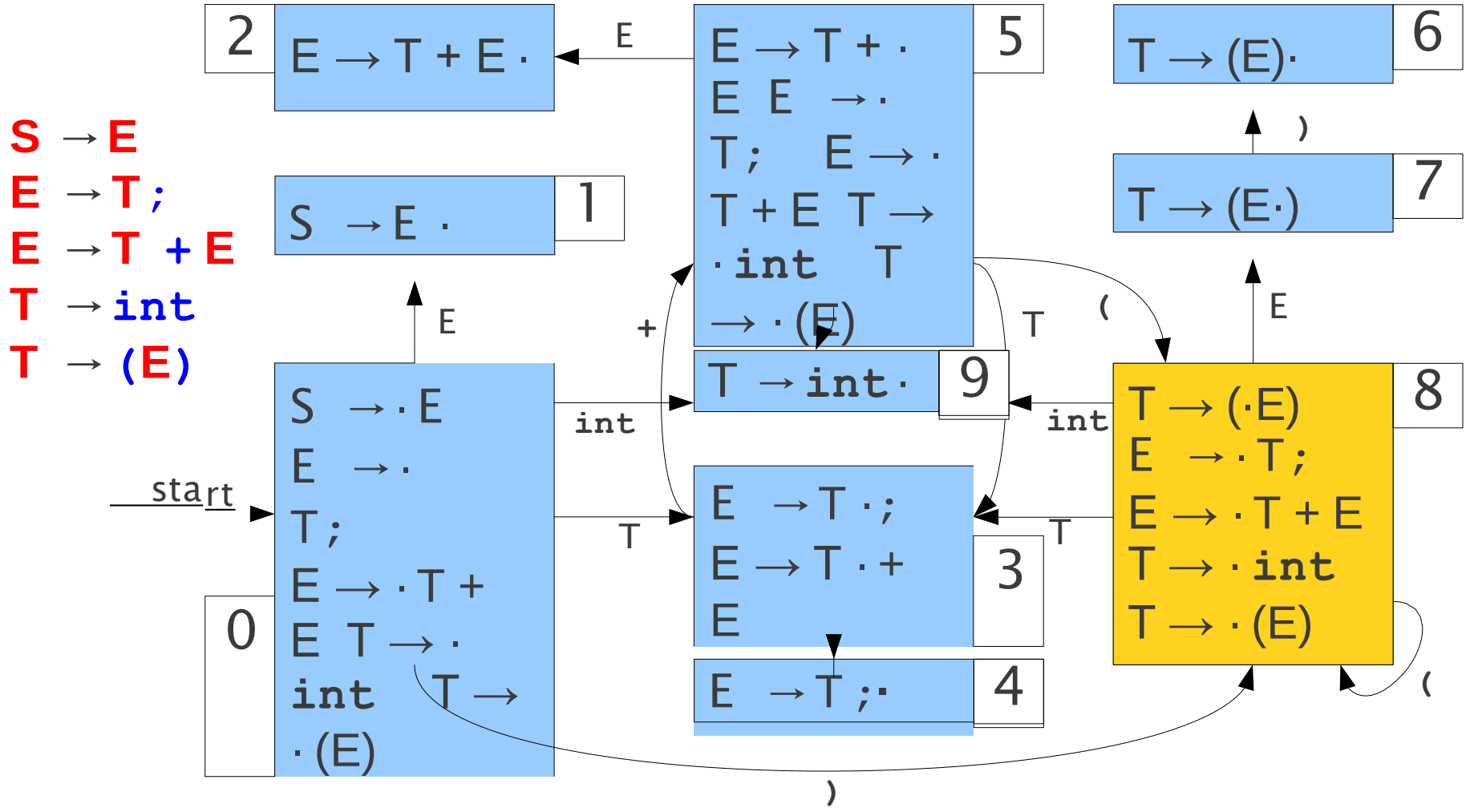


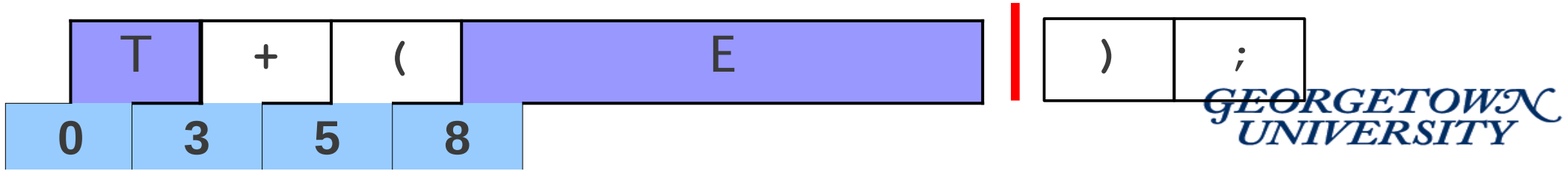
T	+	(
0	3	5	8

)	;

GEORGETOWN UNIVERSITY

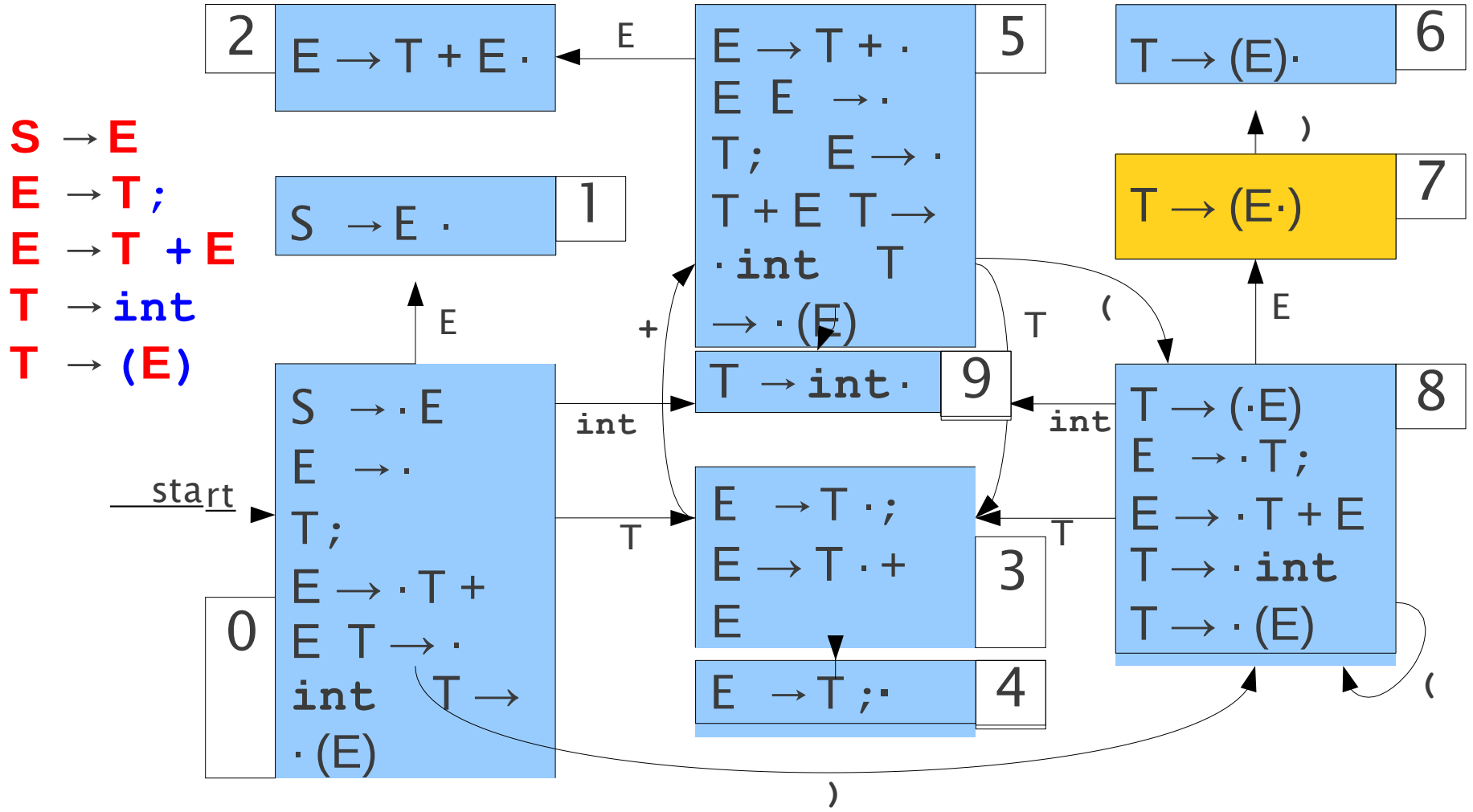
LR(0) Parsing

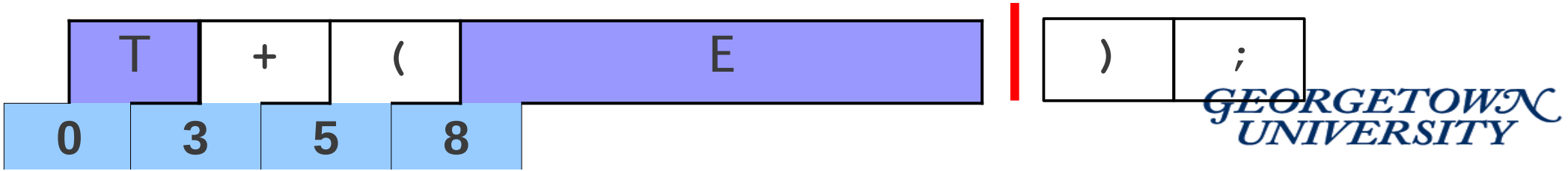




GEORGETOWN UNIVERSITY

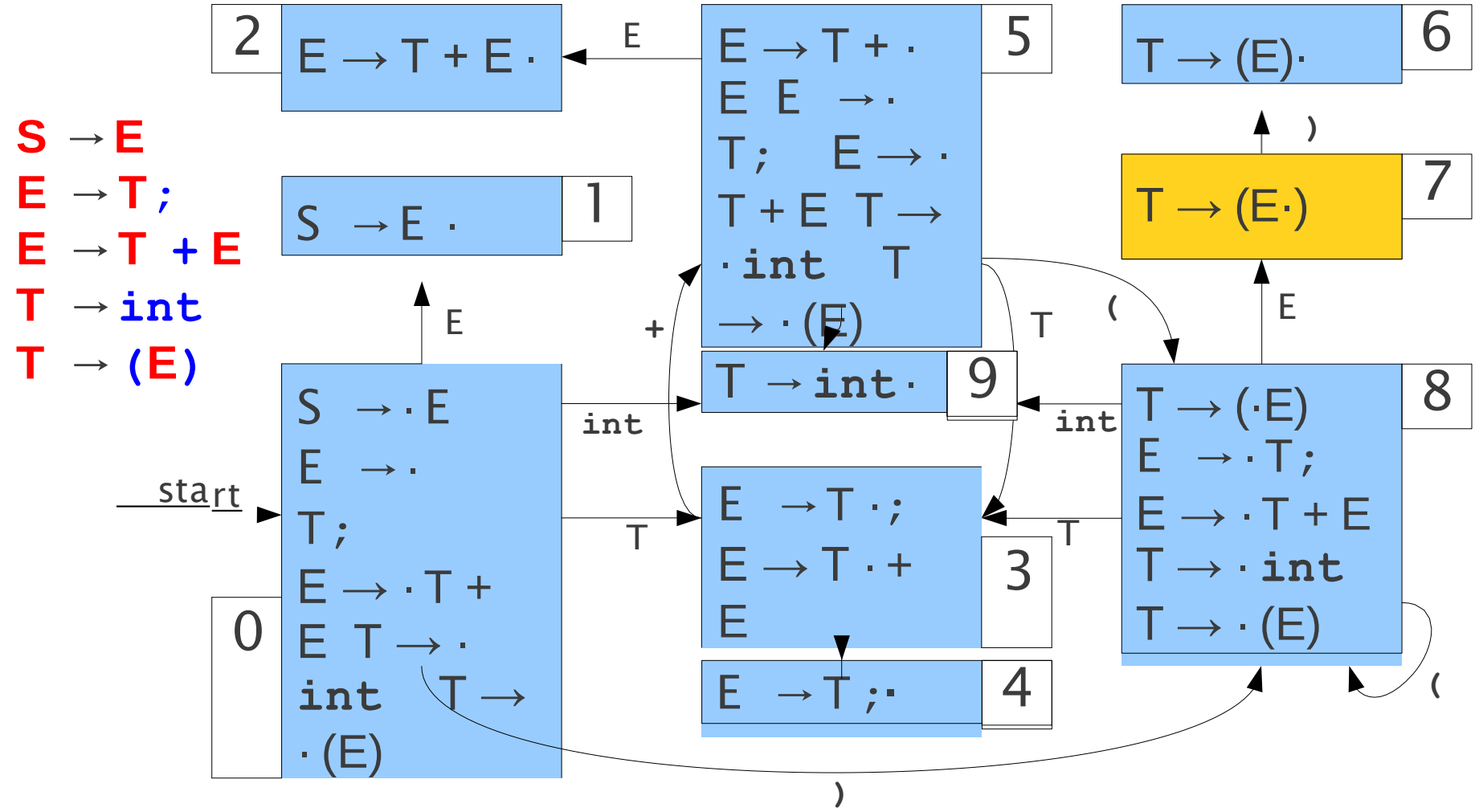
LR(0) Parsing

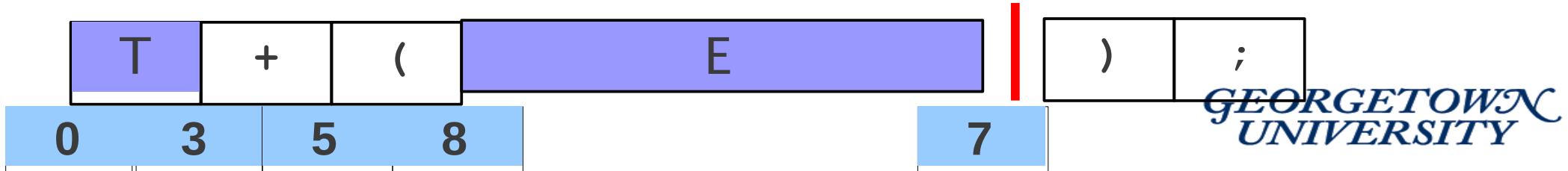




GEORGETOWN UNIVERSITY

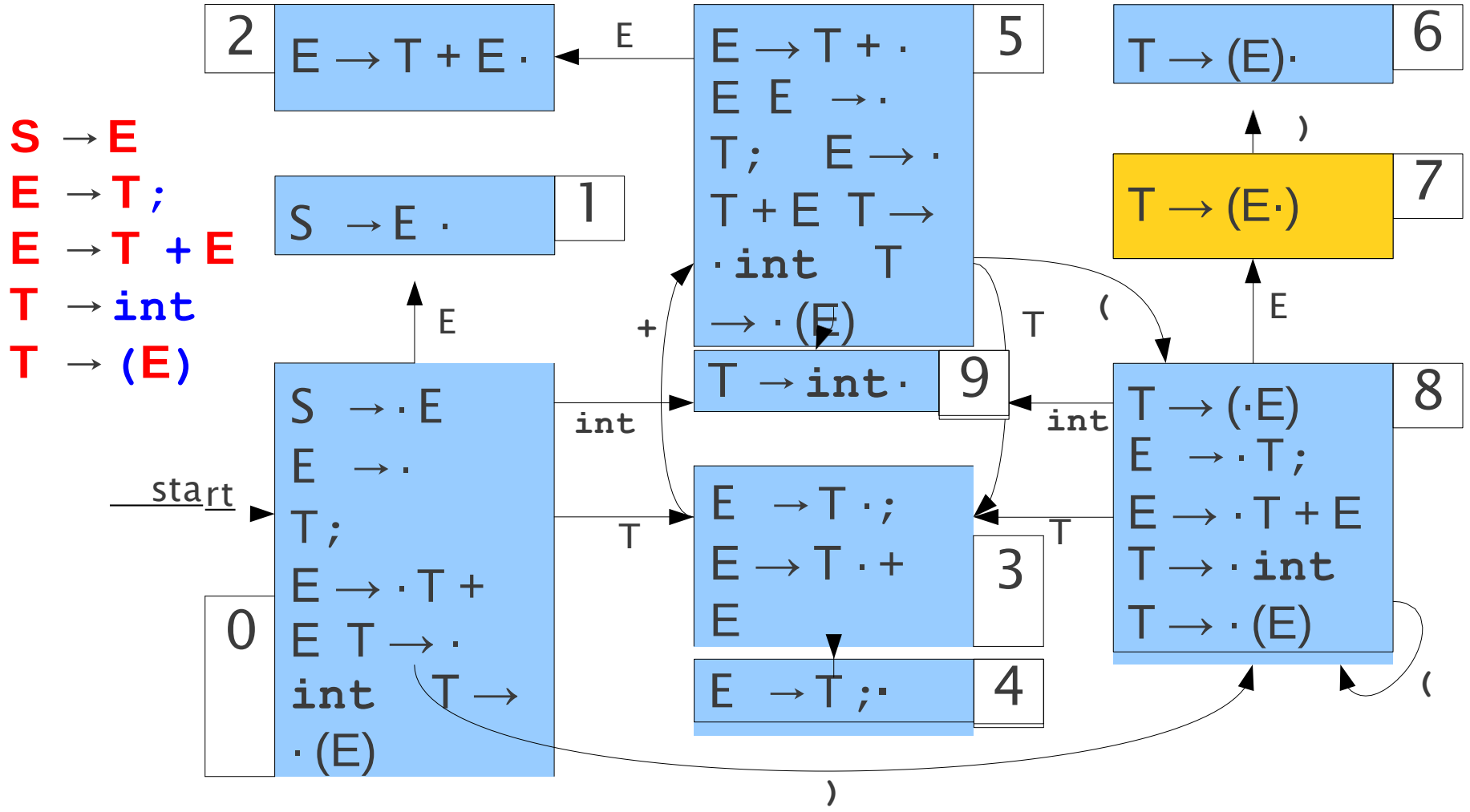
LR(0) Parsing

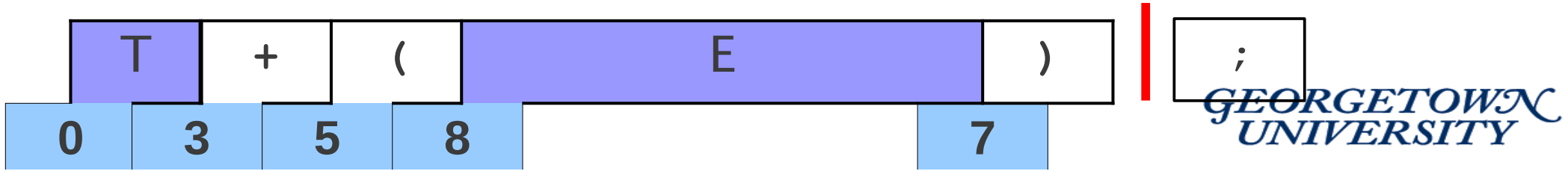




GEORGETOWN UNIVERSITY

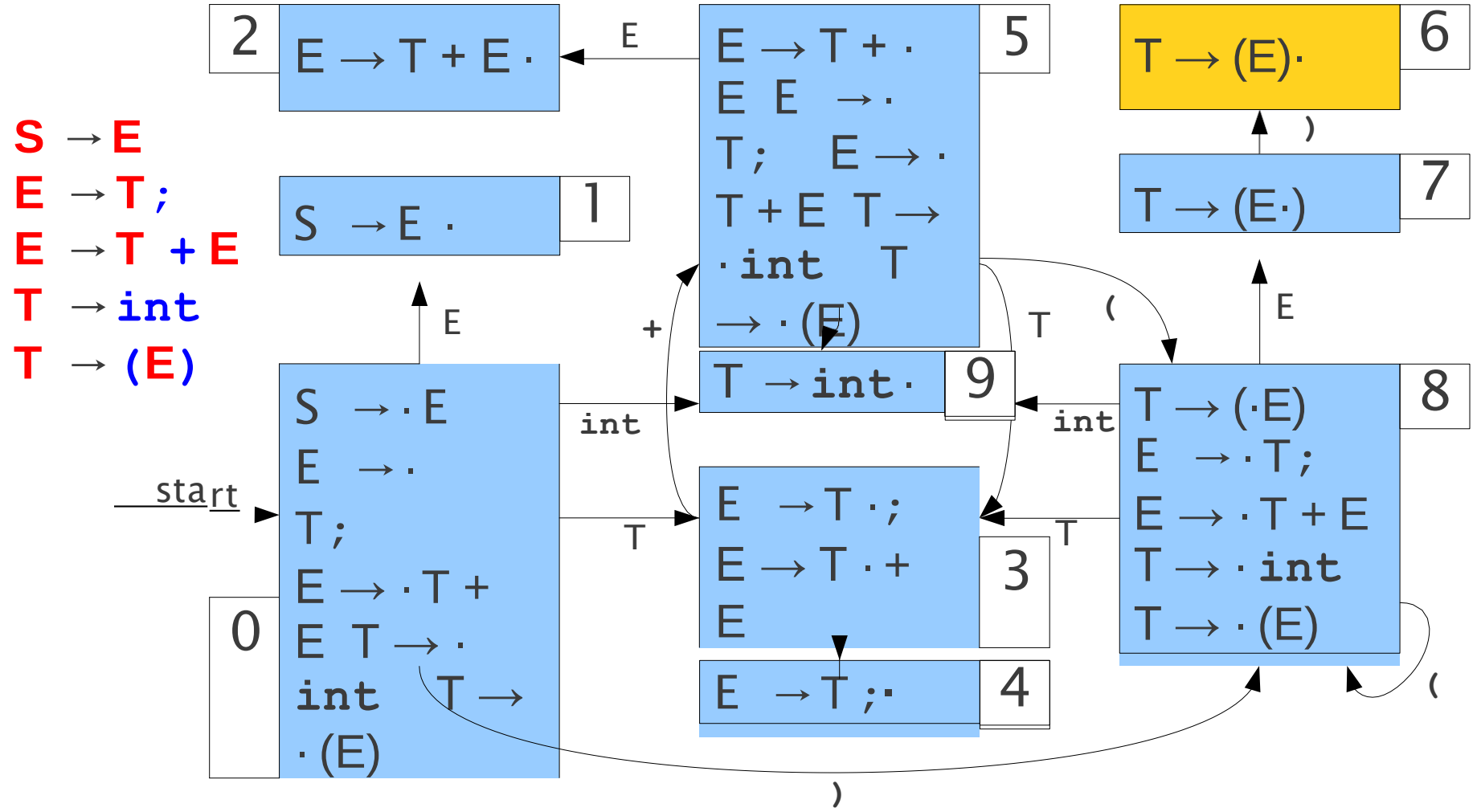
LR(0) Parsing

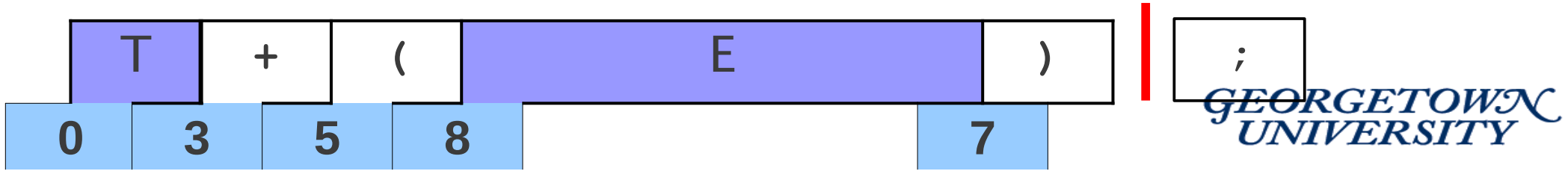




GEORGETOWN UNIVERSITY

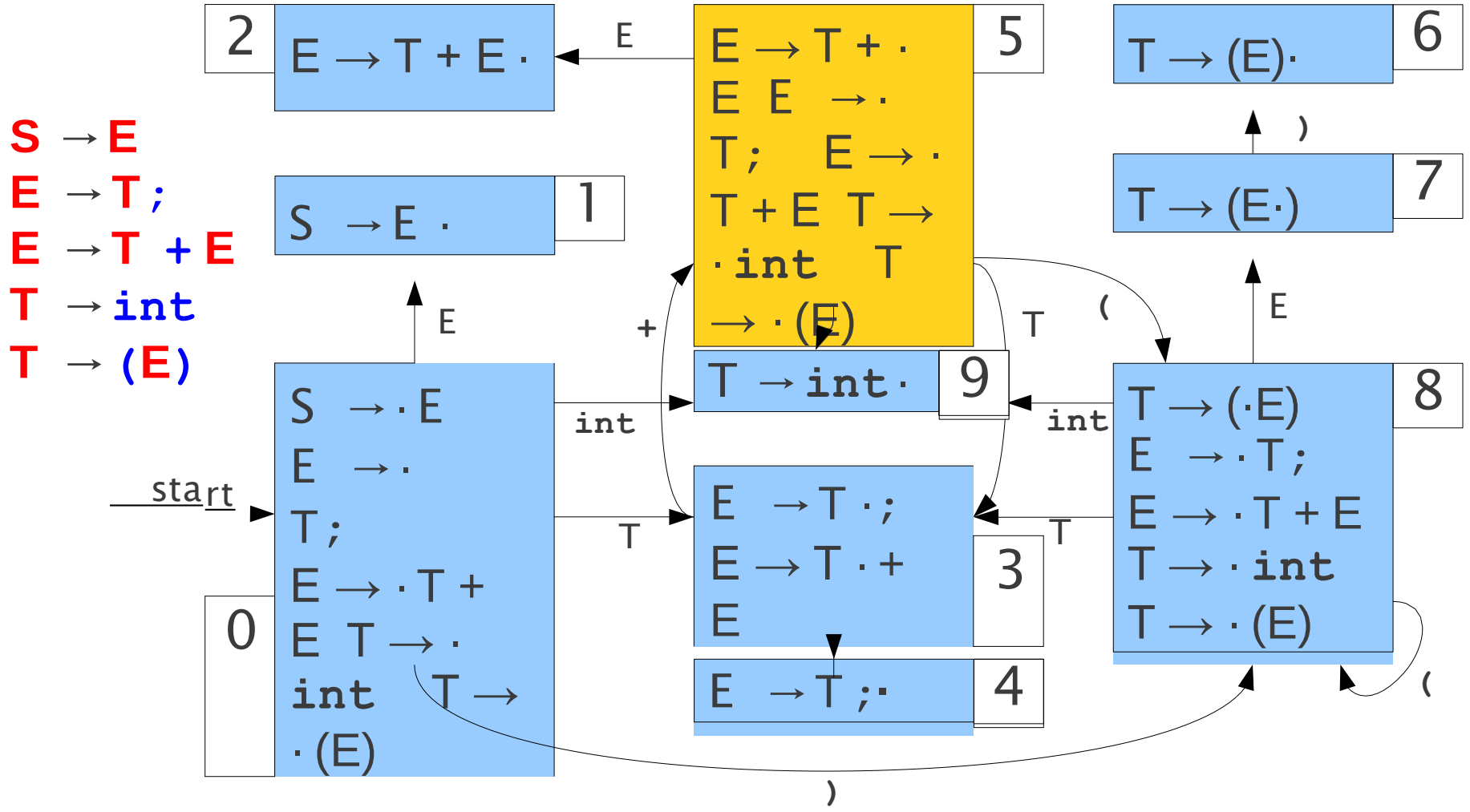
LR(0) Parsing

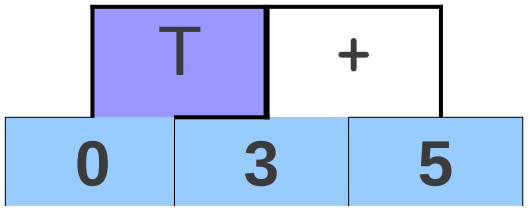




GEORGETOWN UNIVERSITY

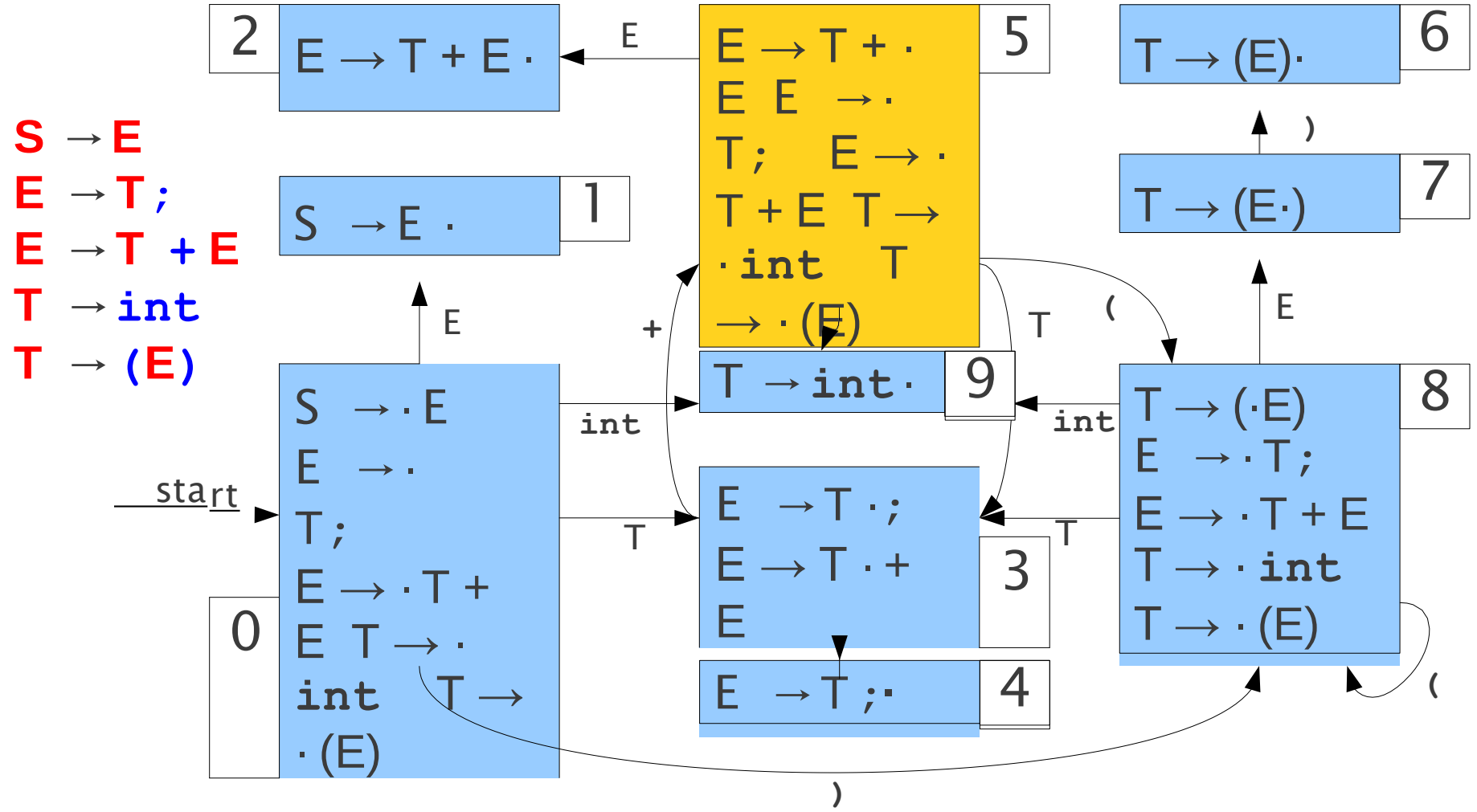
LR(0) Parsing

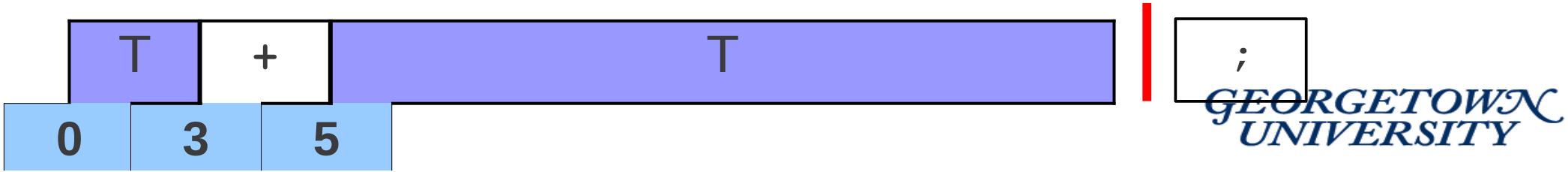




GEORGETOWN UNIVERSITY

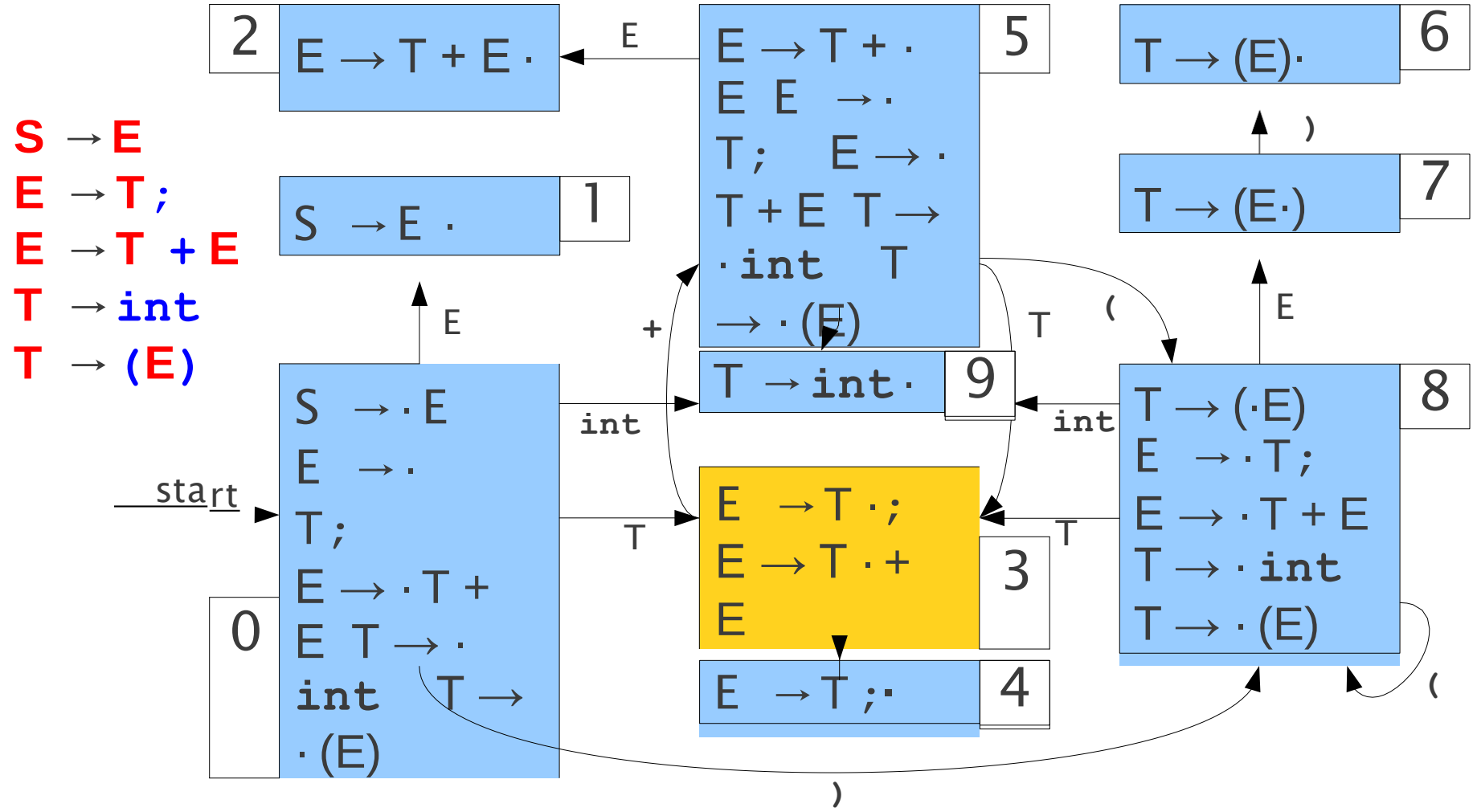
LR(0) Parsing

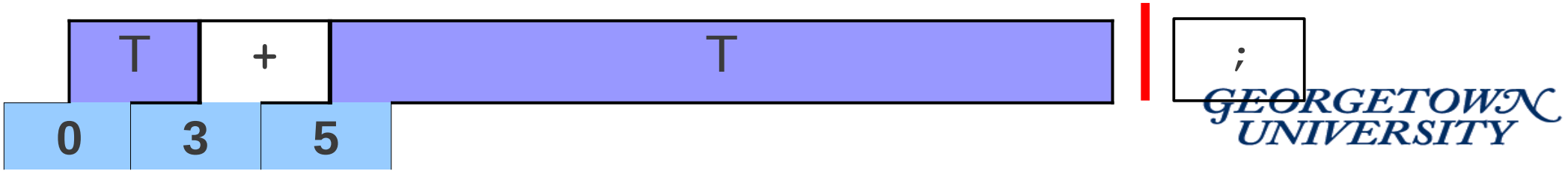




GEORGETOWN UNIVERSITY

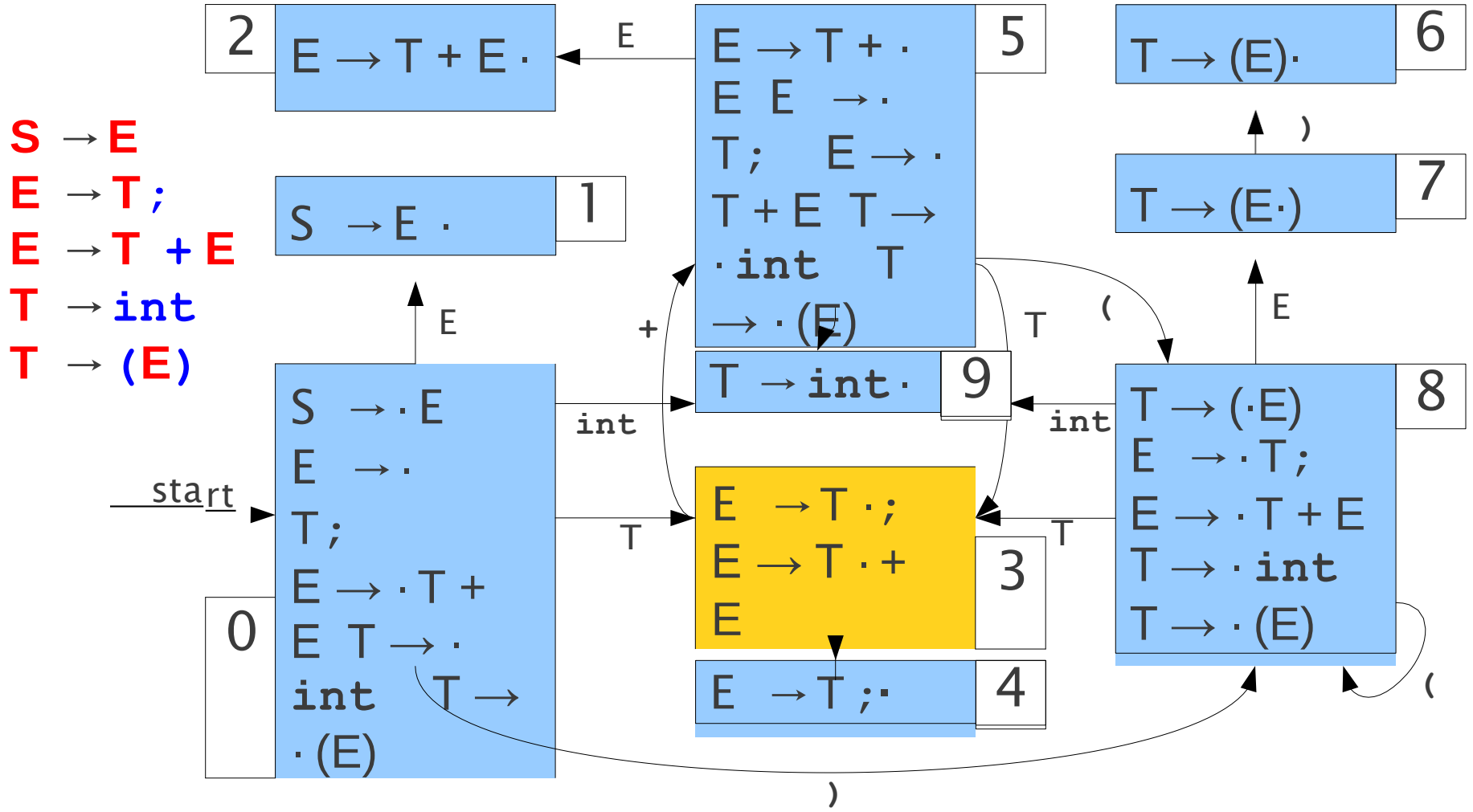
LR(0) Parsing

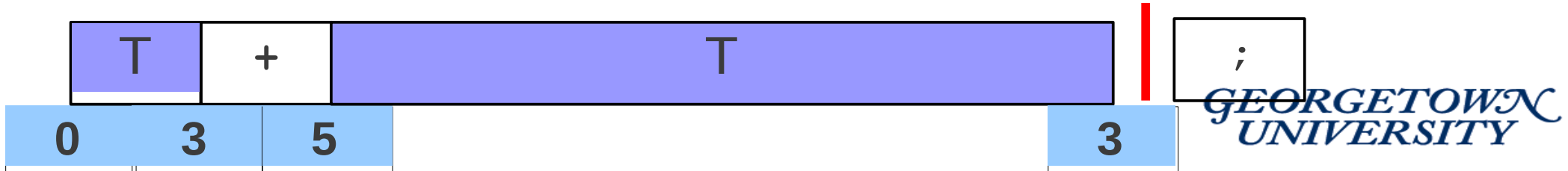




GEORGETOWN UNIVERSITY

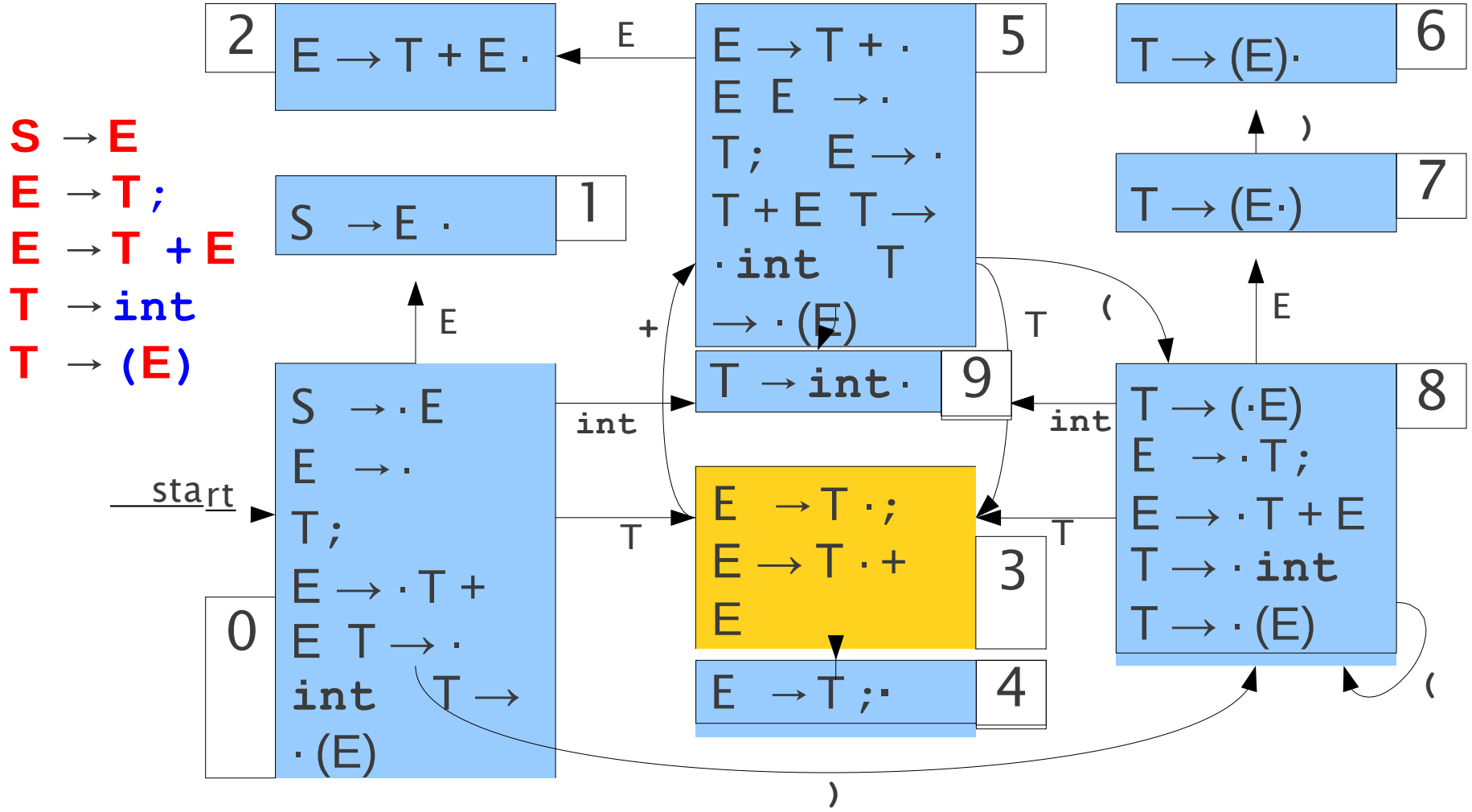
LR(0) Parsing

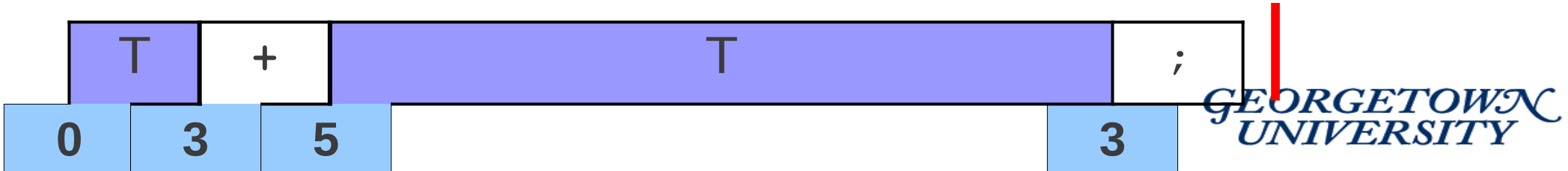




GEORGETOWN UNIVERSITY

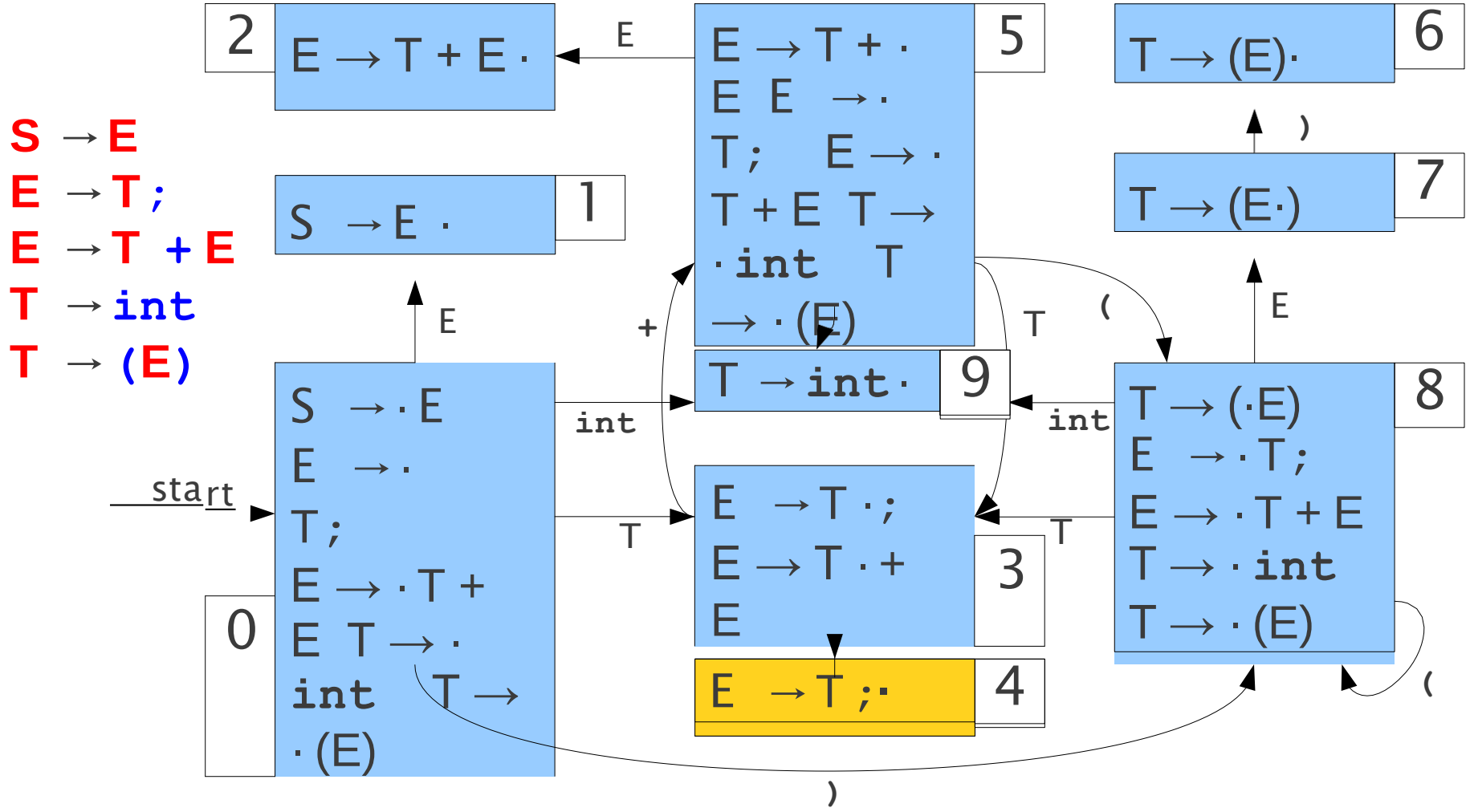
LR(0) Parsing

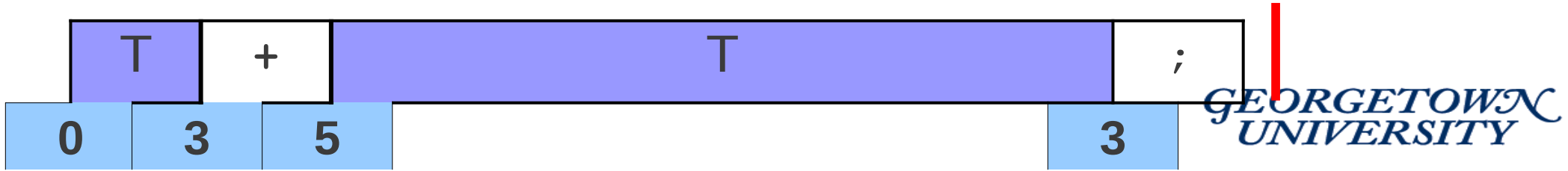




GEORGETOWN UNIVERSITY

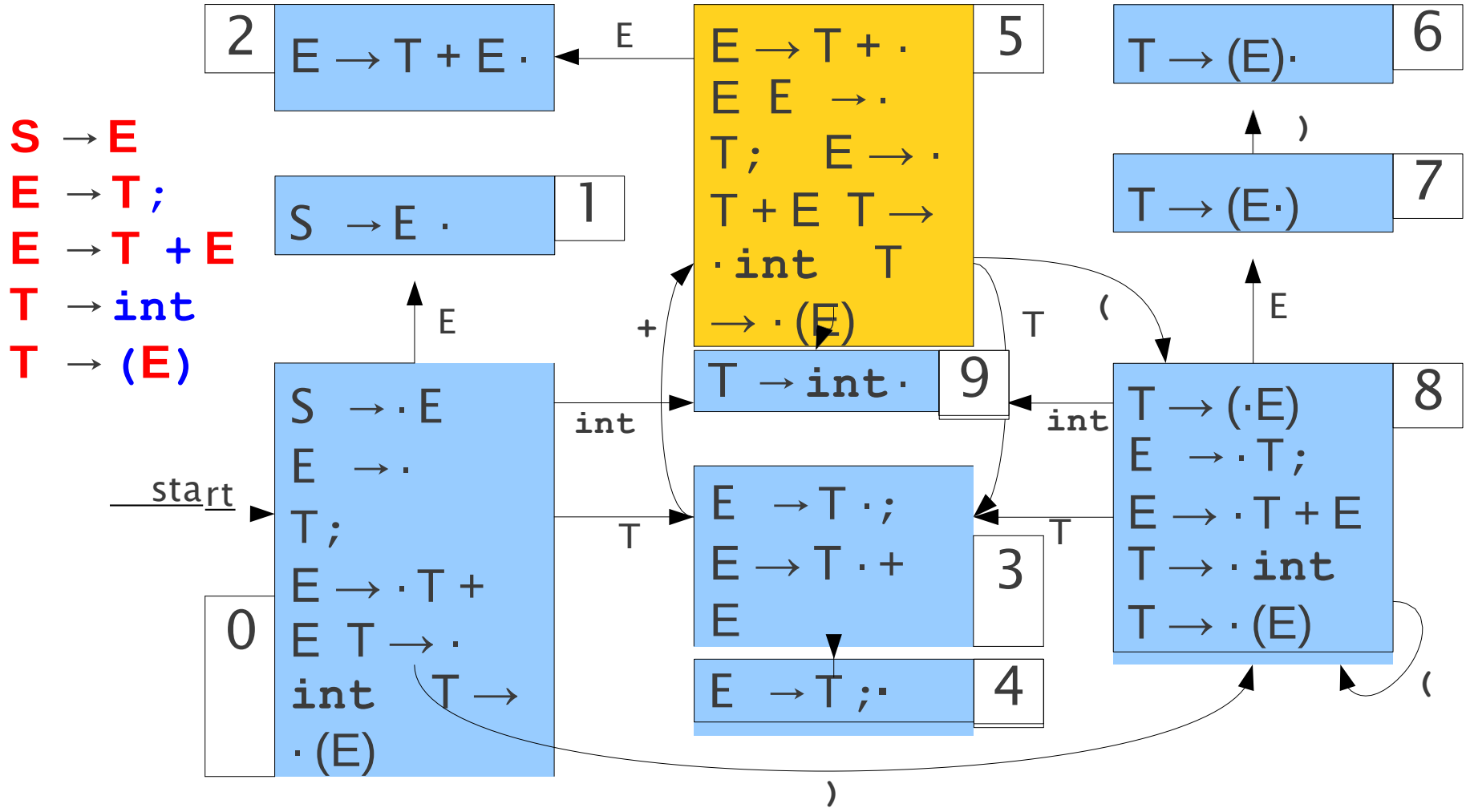
LR(0) Parsing

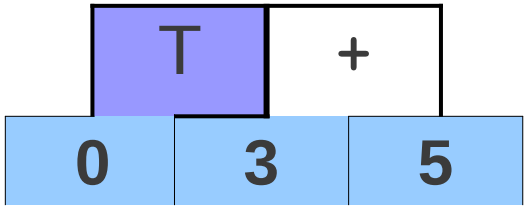




GEORGETOWN UNIVERSITY

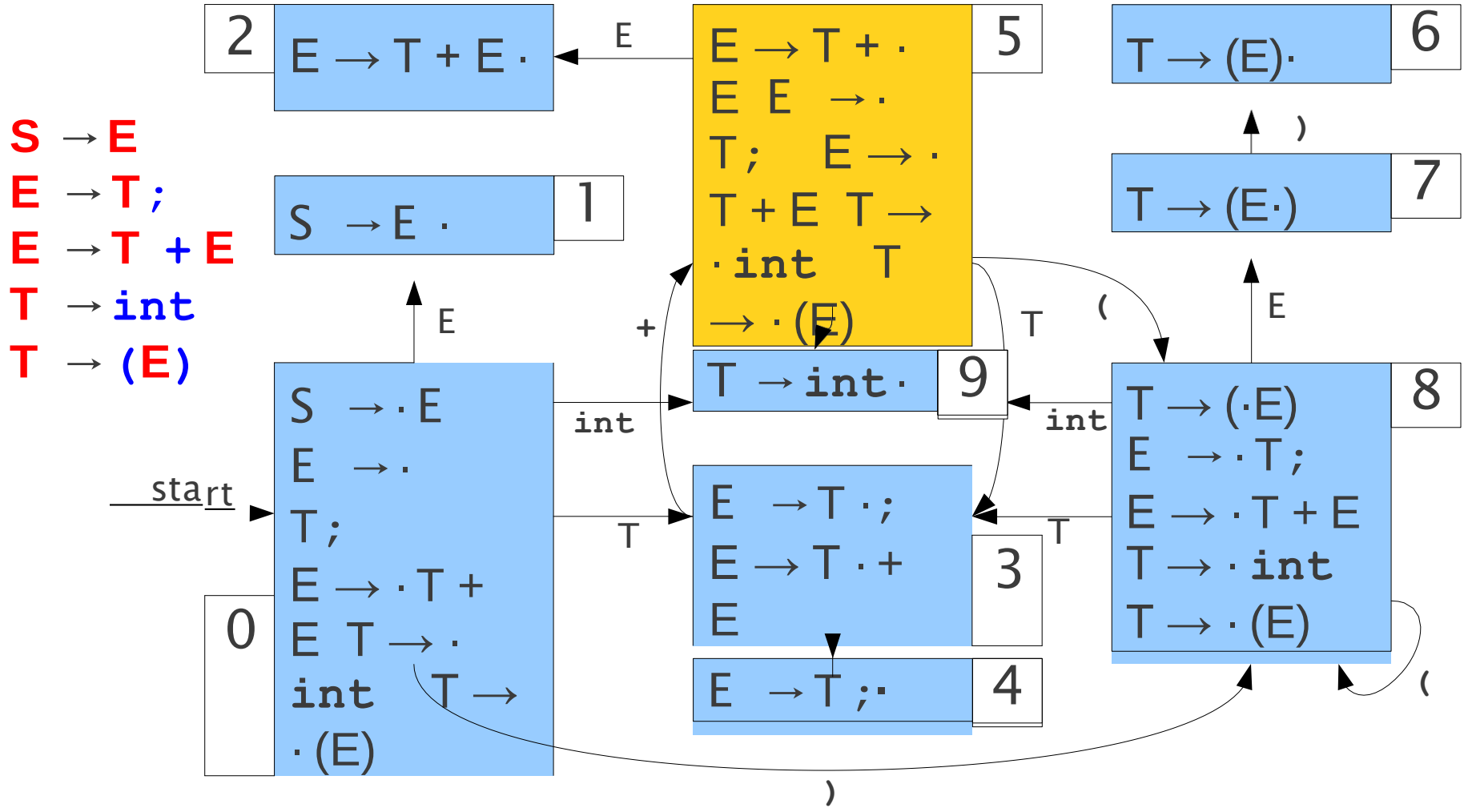
LR(0) Parsing





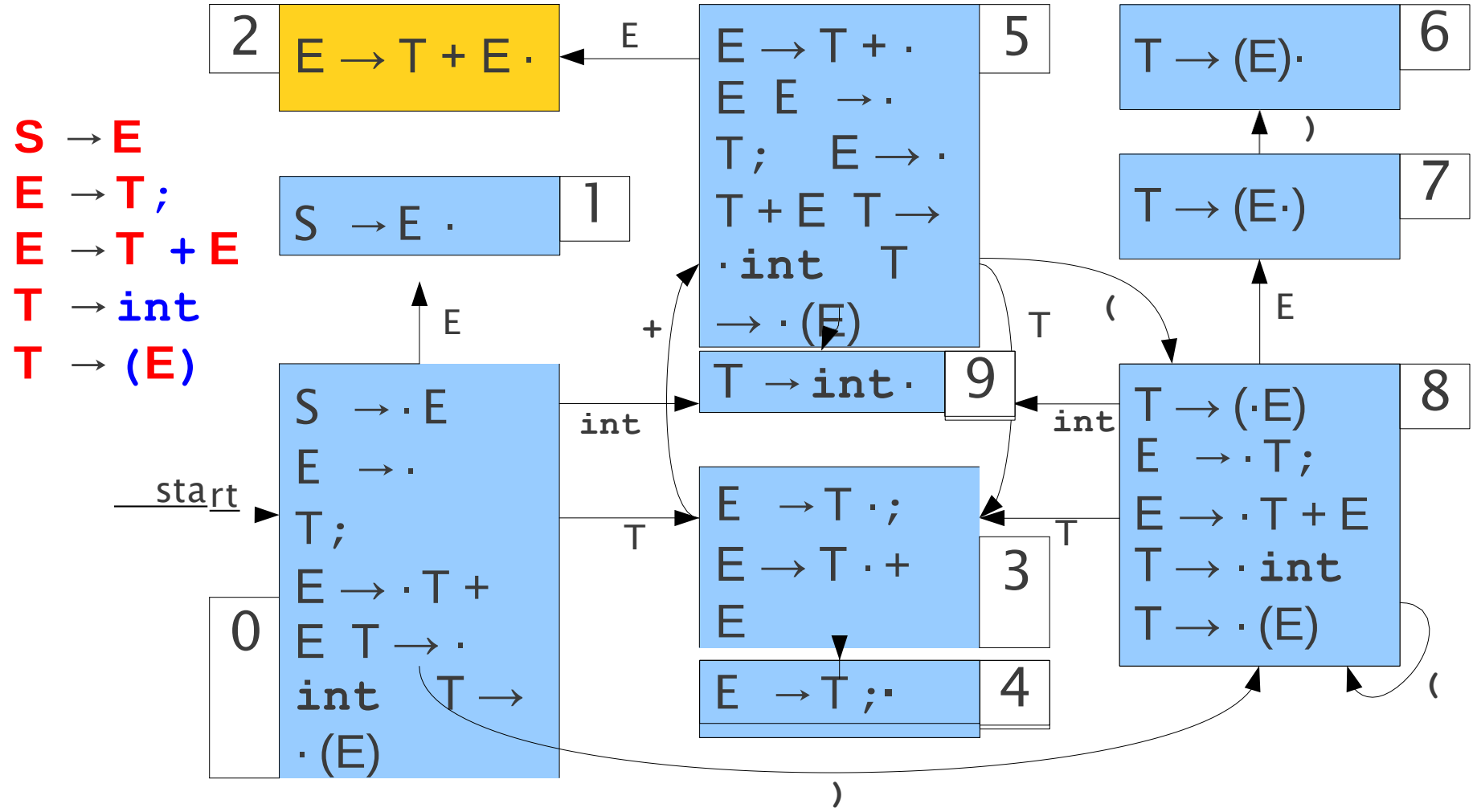
GEORGETOWN UNIVERSITY

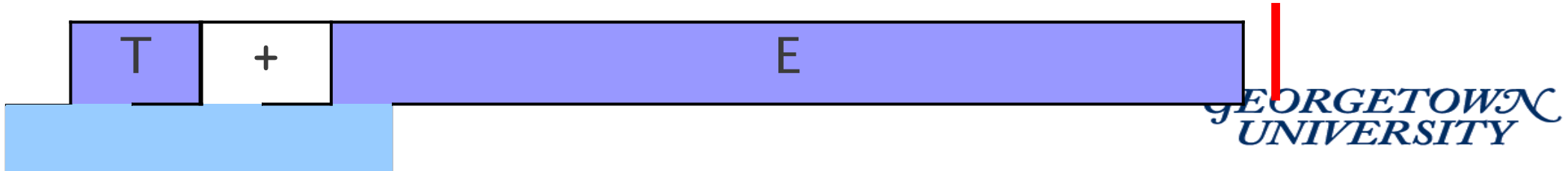
LR(0) Parsing



GEORGETOWN UNIVERSITY

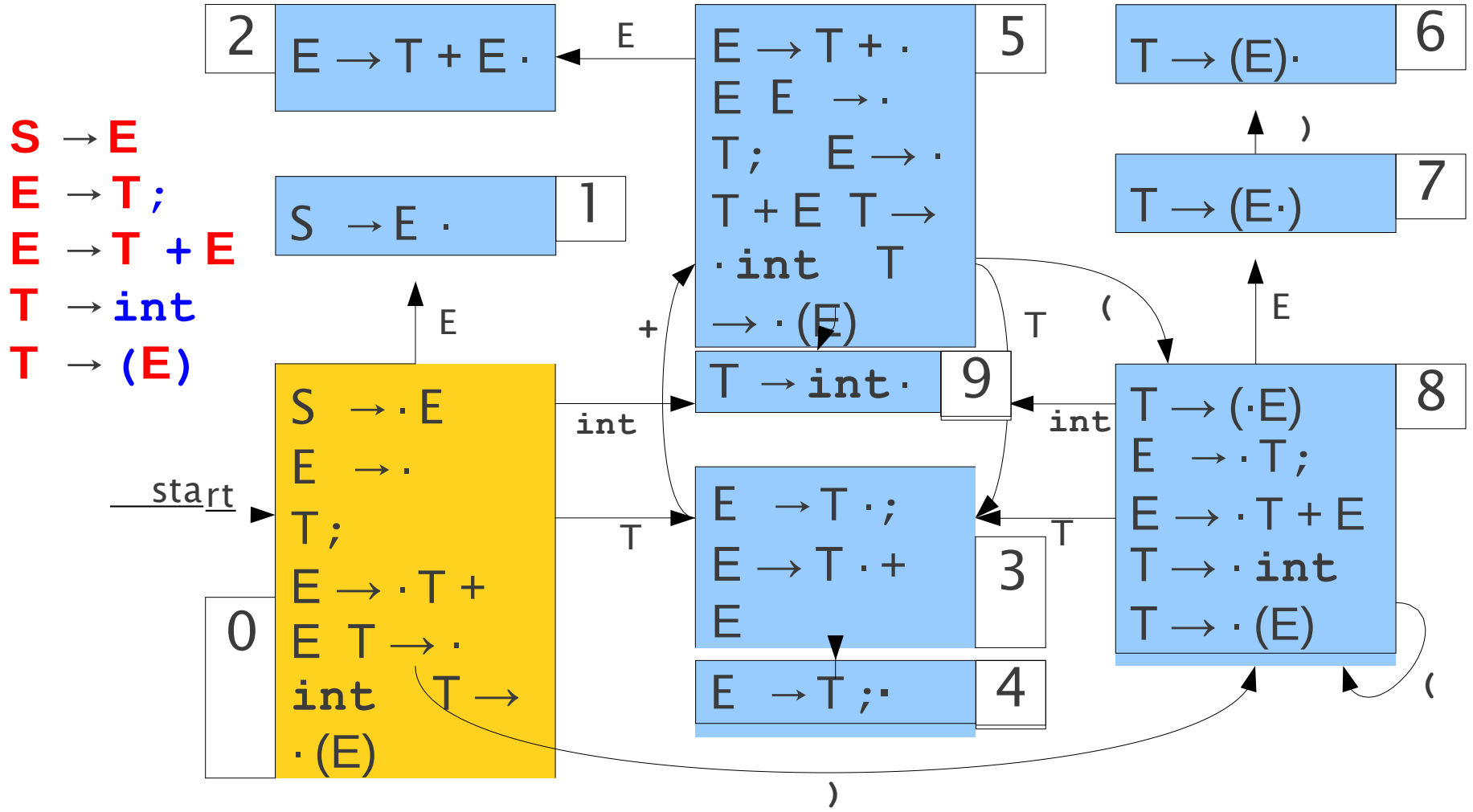
LR(0) Parsing





GEORGETOWN UNIVERSITY

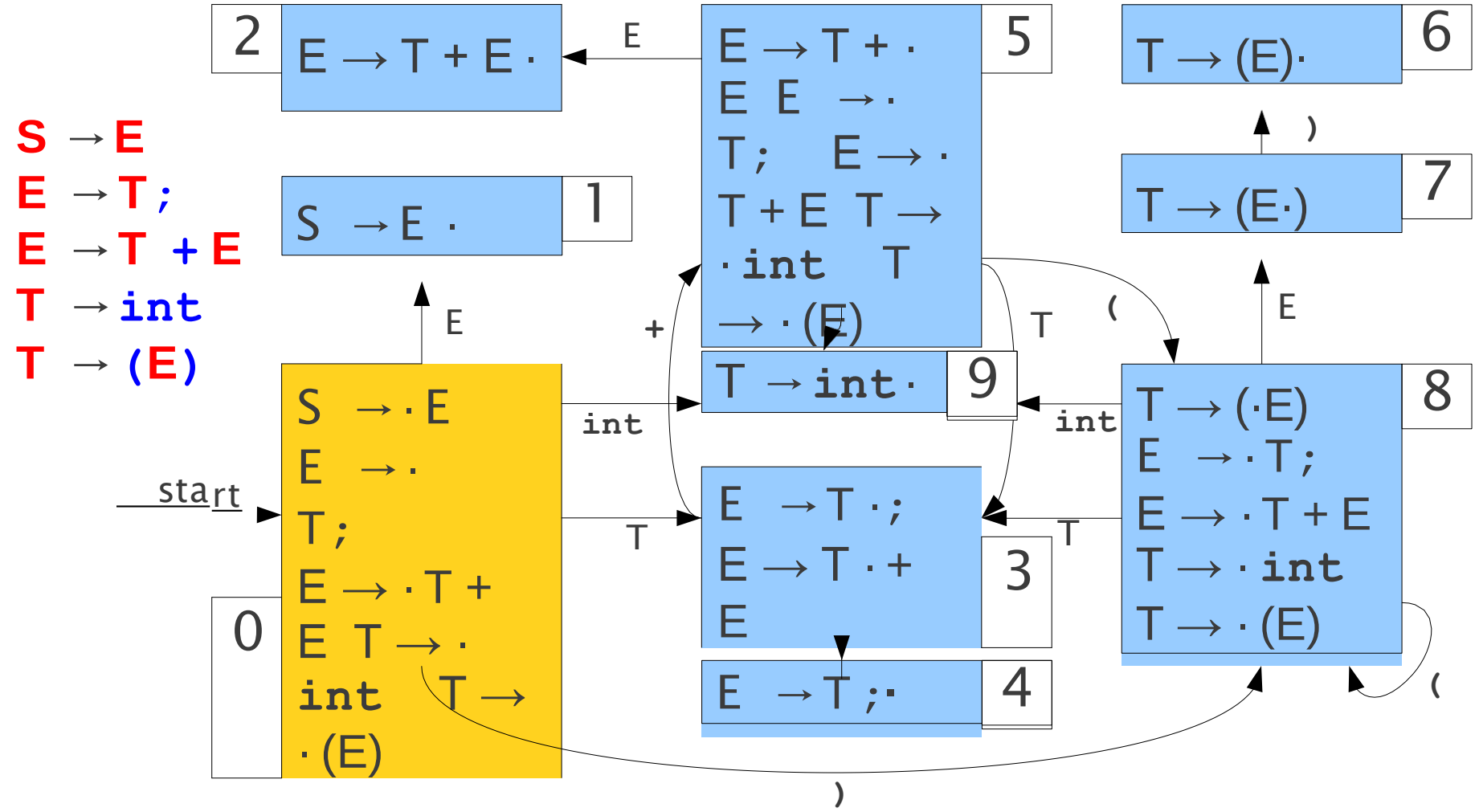
LR(0) Parsing



0

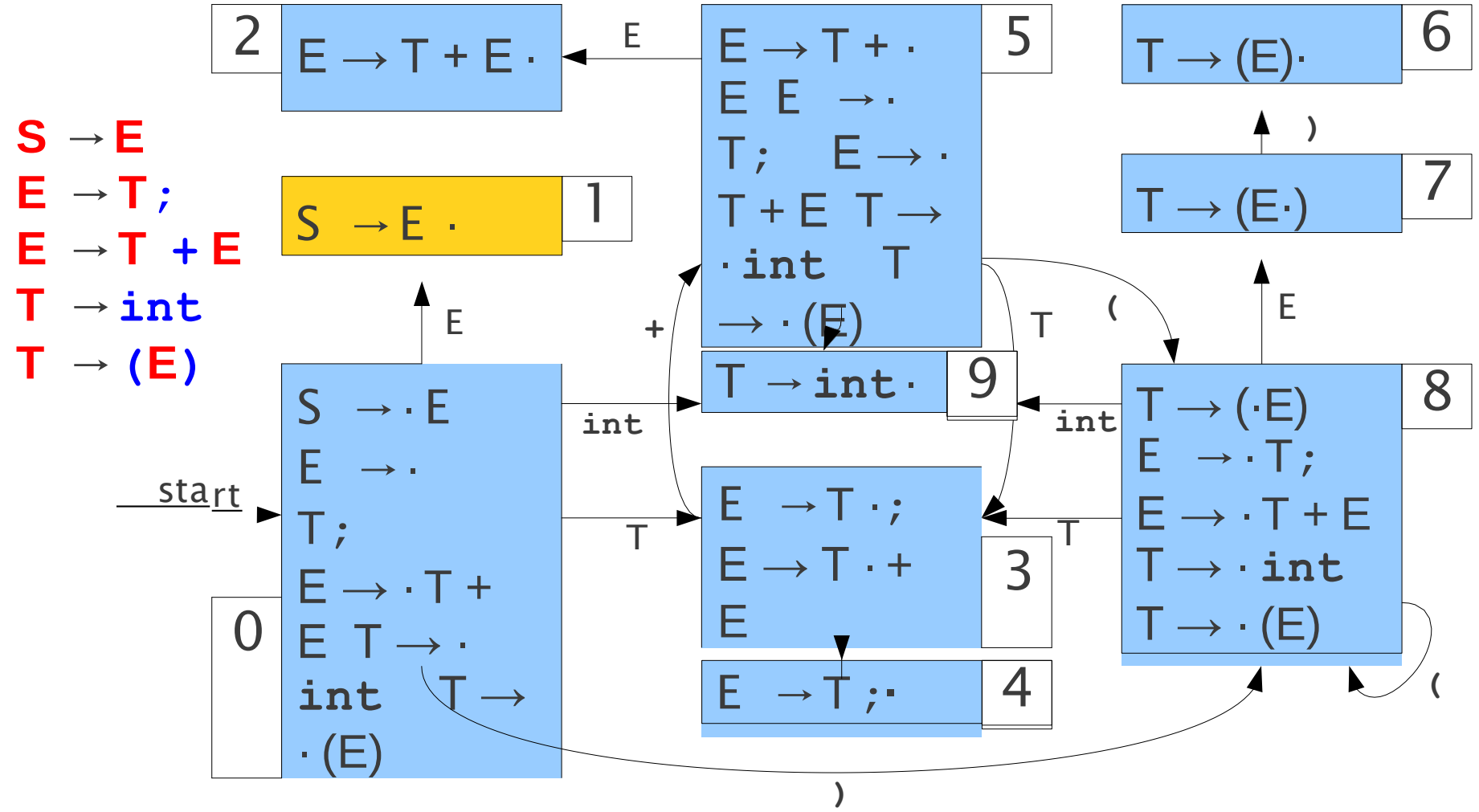
GEORGETOWN UNIVERSITY

LR(0) Parsing



GEORGETOWN UNIVERSITY

LR(0) Parsing

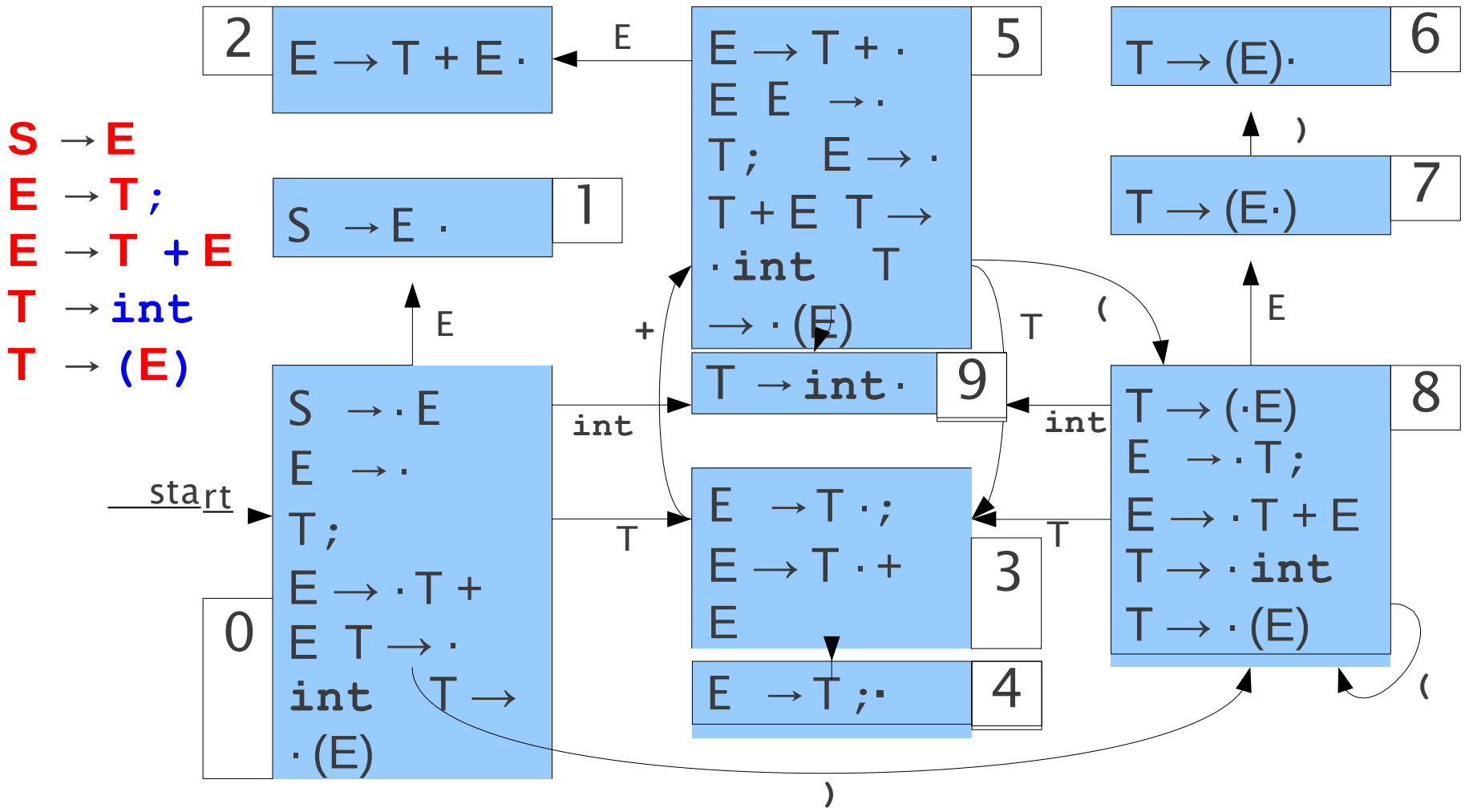


GEORGETOWN UNIVERSITY

Representing the Automaton

- LR(0) parsers are usually represented via two tables: an **action** table and a **goto** table.
- The **action** table maps each state to an action:
  - **shift**, which shifts the next terminal, and
  - **reduce** **A** $\rightarrow \omega$, which performs reduction **A** $\rightarrow \omega$.
  - Any state of the form **A** $\rightarrow \omega$ $\cdot$ does that reduction; everything else shifts.
- The **goto** table maps state/symbol pairs to a next state.
  - This is just the transition table for the automaton.

Building LR(0) Tables



GEORGETOWN UNIVERSITY

LR(0) Tables

	int	+	;	(	)	E	T	Action
0	9			8		1	3	Shift
1								Accept
2								Reduce E →T + E
3		5	4					Shift
4								Reduce E →T;
5	9			8		2	3	Shift
6								Reduce T →(E)
7					6			Shift
8	9			8		7	3	Shift
9								Reduce T →int

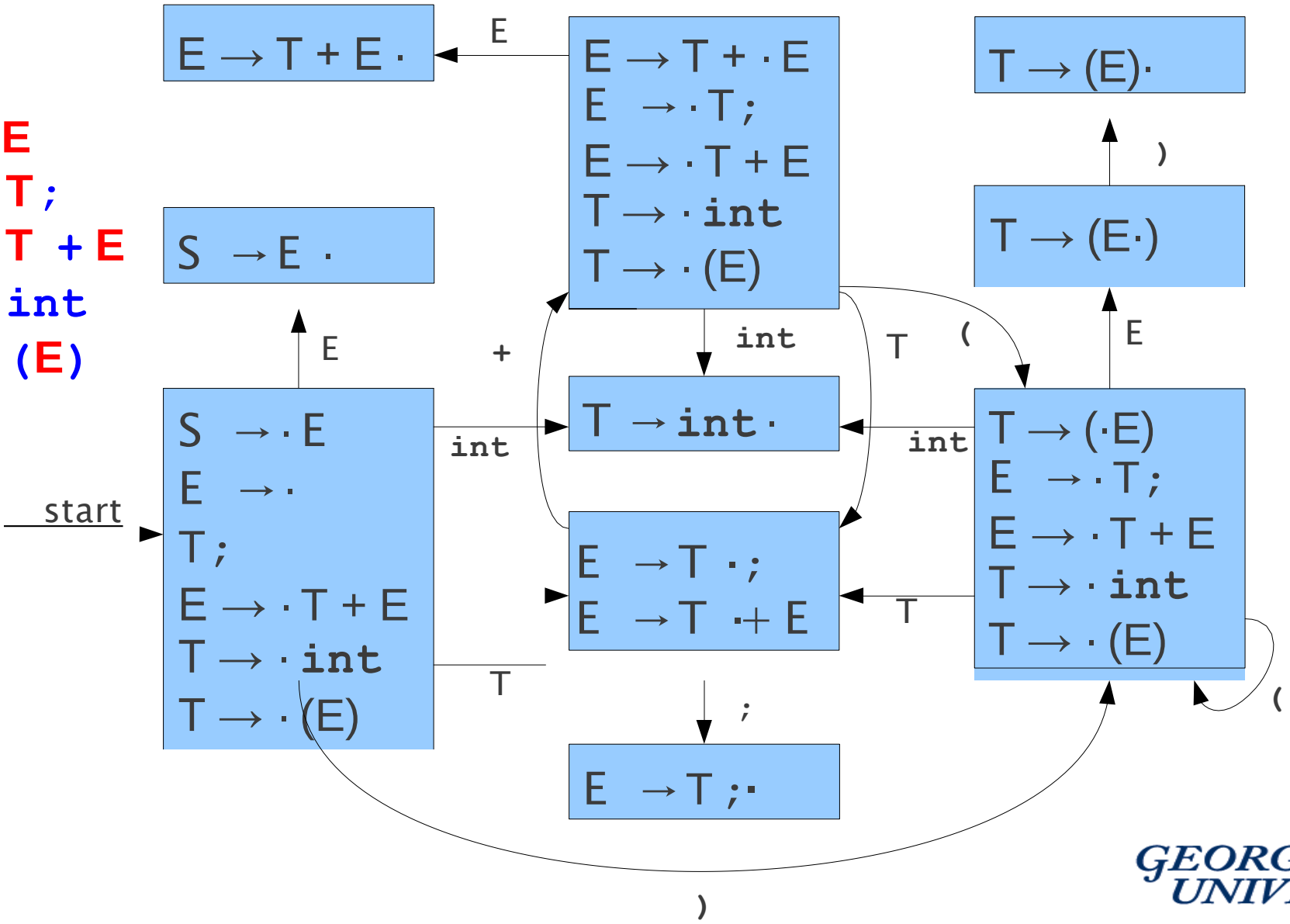
The LR(0) Algorithm

- Maintain a stack of (symbol, state) pairs, which is initially (`?`, 1) for some dummy symbol `?`.
- While the stack is not empty:
  - Let **state** be the top state.
  - If **action[state]** is **shift**:
    - Let `t` be the next symbol in the input.
    - Push (`t`, **goto[state, `t`]**) atop the stack.
  - If **action[state]** is **reduce A** $\rightarrow \omega$:
    - Remove $|\omega|$ symbols from the top of the stack.
    - Let **top-state** be the state on top of the stack.
    - Push (**A**, **goto[top-state, A]**) atop the stack.
  - Otherwise, report an error.

The Limits of LR(0)

A Non-LR(0) Grammar

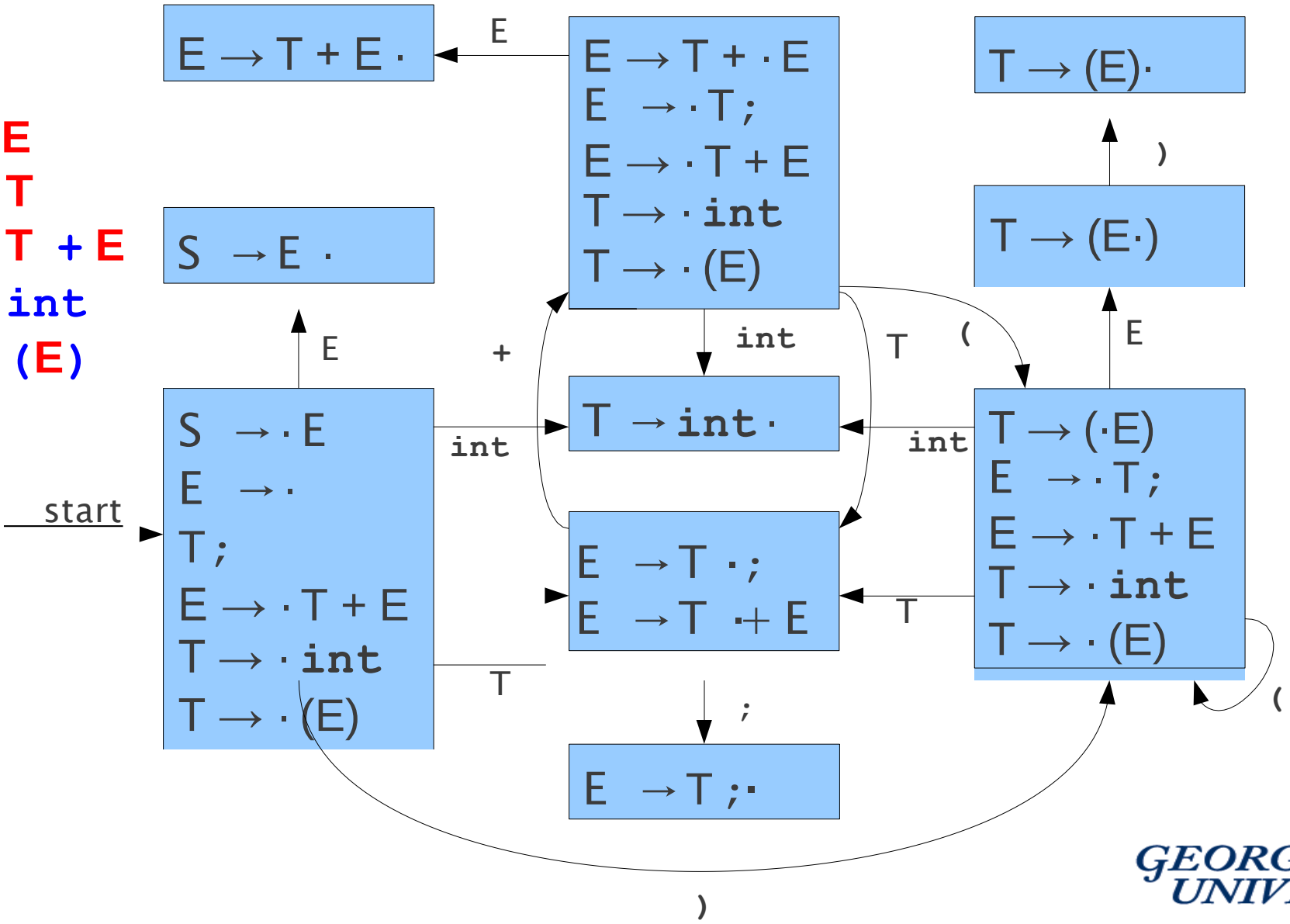
S → E
E → T ;
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

A Non-LR(0) Grammar

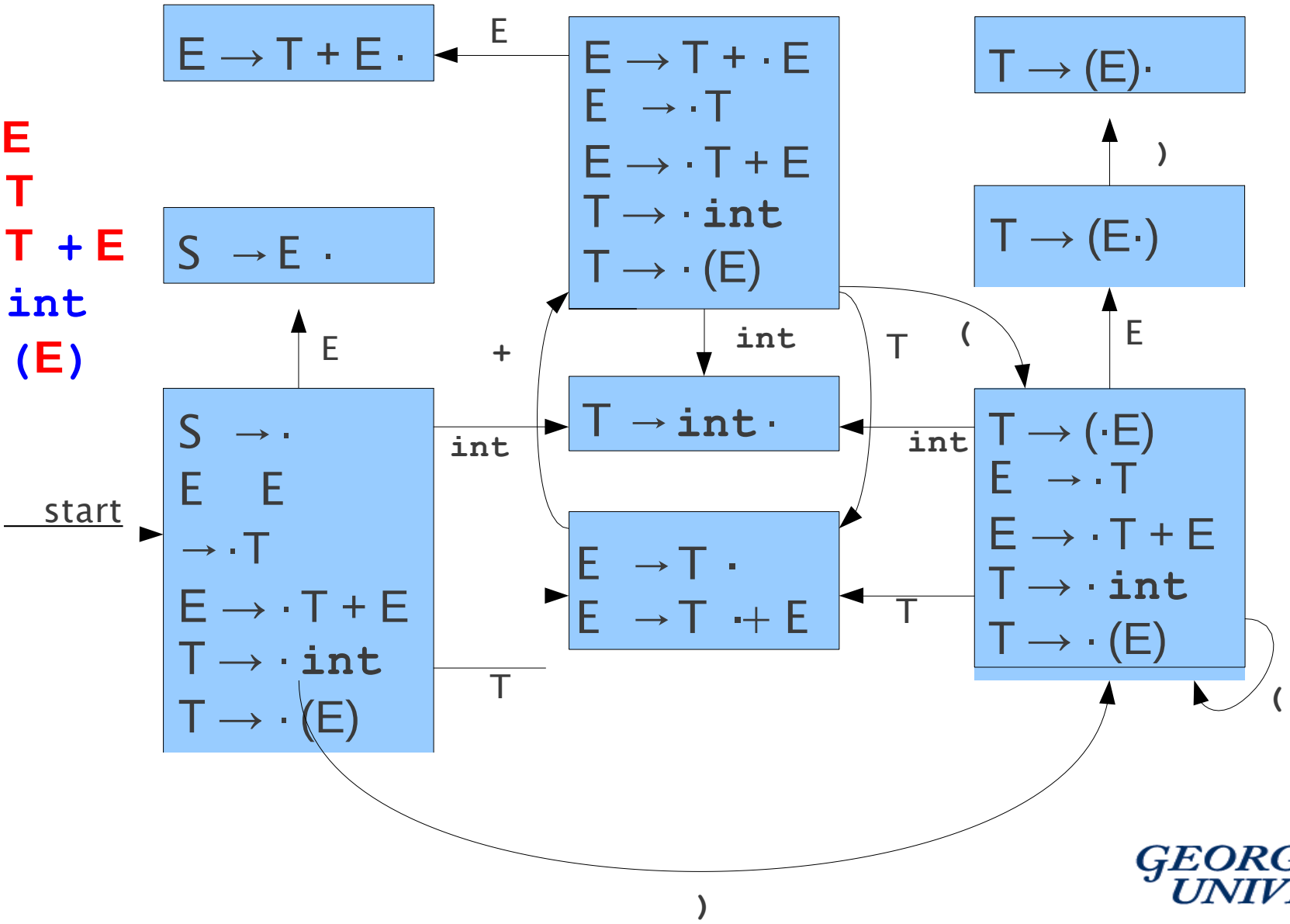
S → E
E → T
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

A Non-LR(0) Grammar

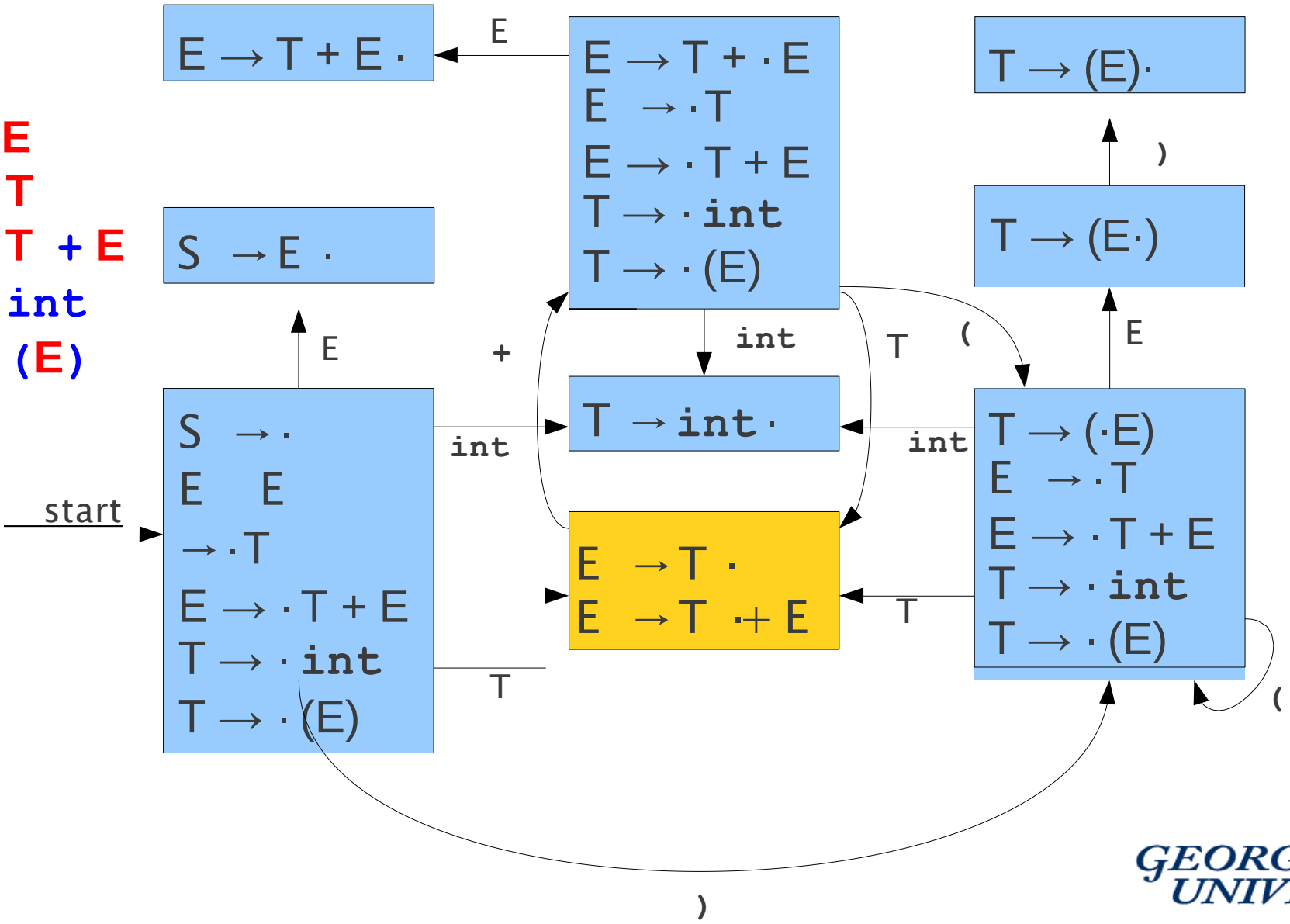
S → E
E → T
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

A Non-LR(0) Grammar

S → E
E → T
E → T + E
T → int
T → (E)



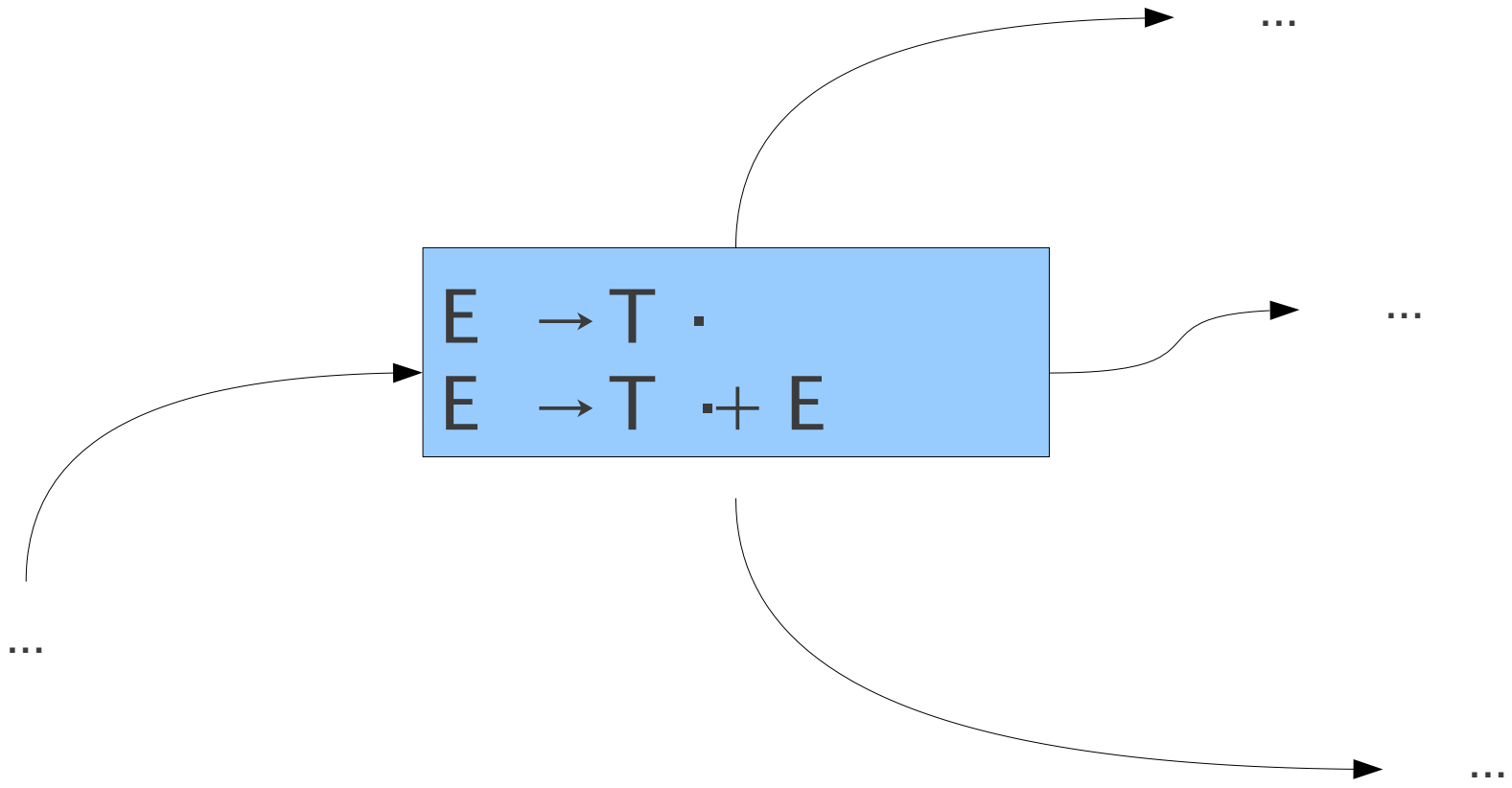
GEORGETOWN UNIVERSITY

LR Conflicts

- A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.
  - Often happens when two productions overlap.
- A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.
  - Often the result of ambiguous grammars.
- A grammar whose handle-finding automaton contains a shift/reduce conflict or a reduce/reduce conflict is not LR(0).
- Can you have a shift/shift conflict?

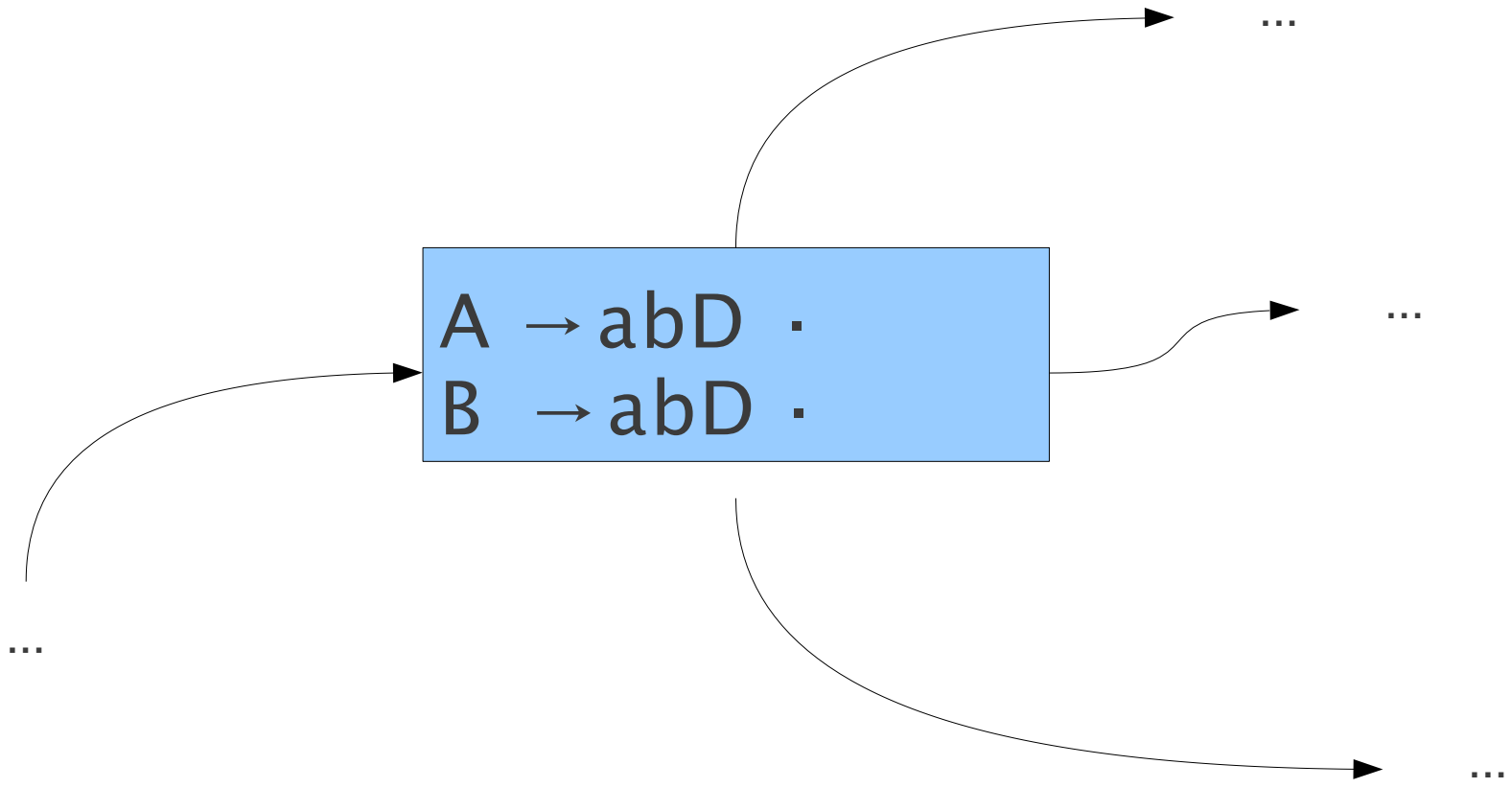
What error is this?

What error is this?



What about this?

What about this?



What do these conflicts mean?

- Recall: our automaton was constructed by looking for viable prefixes.
- Each accepting state represents a point where the handle might occur.
- A **shift/reduce** conflict is a state where the handle might occur, but we might actually need to keep searching.
- A **reduce/reduce** conflict is a state where we know we have found the handle, but can't tell which reduction to apply.

Why LR(0) is Weak

- LR(0) only accepts languages where the handle can be found with no **right context**.
- Our shift/reduce parser only looks to the left of the handle, not to the right.
- How do we exploit the tokens after a possible handle to determine what to do?

A Powerful Parser: **LR(1)**

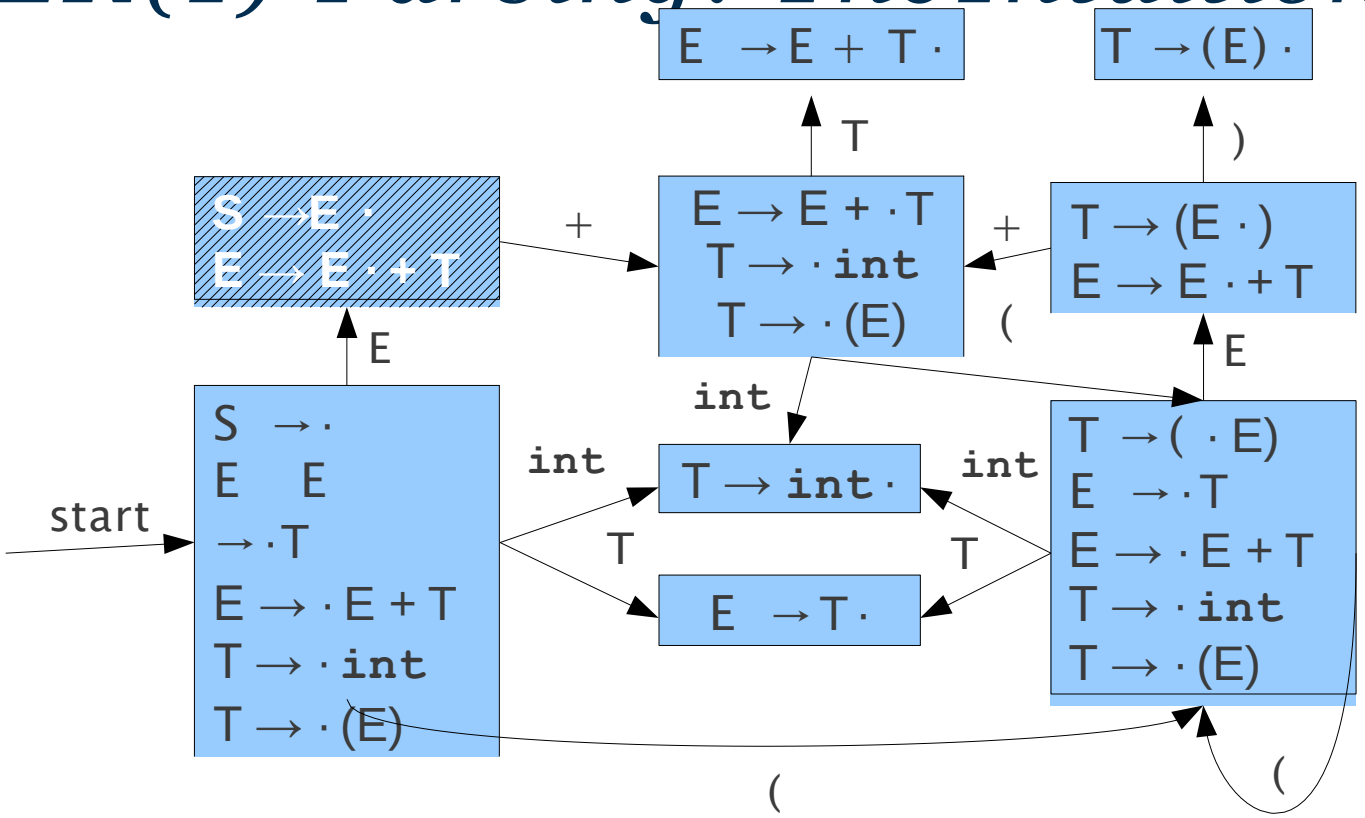
- Bottom-up predictive parsing with
  - **L**: **L**eft-to-right scan
  - **R**: **R**ightmost derivation
  - (**1**): One token lookahead
- *Substantially* more powerful than the other methods we've covered so far (more on that later).
- Tries to more intelligently find handles by using a lookahead token at each step.

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

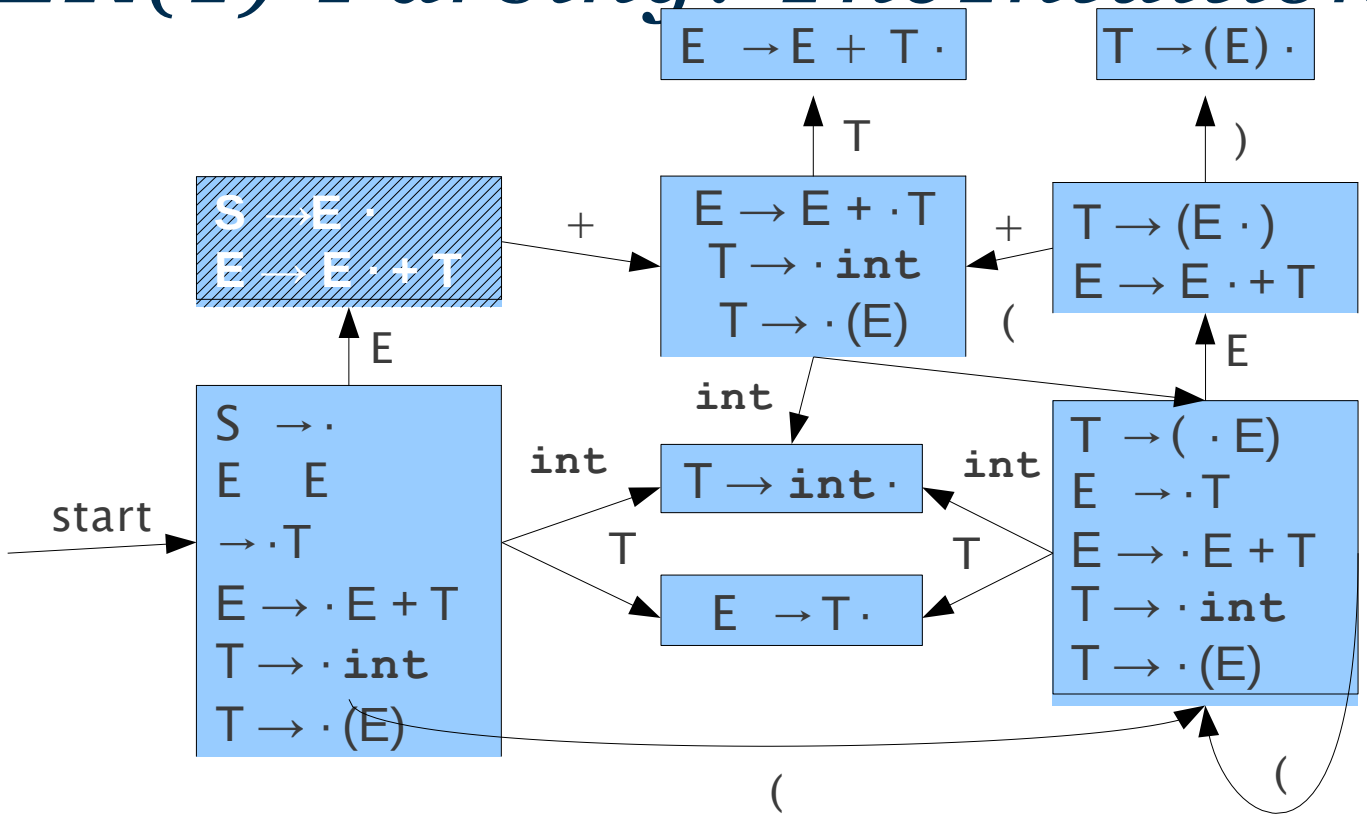
LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



LR(1) Parsing: The Intuition

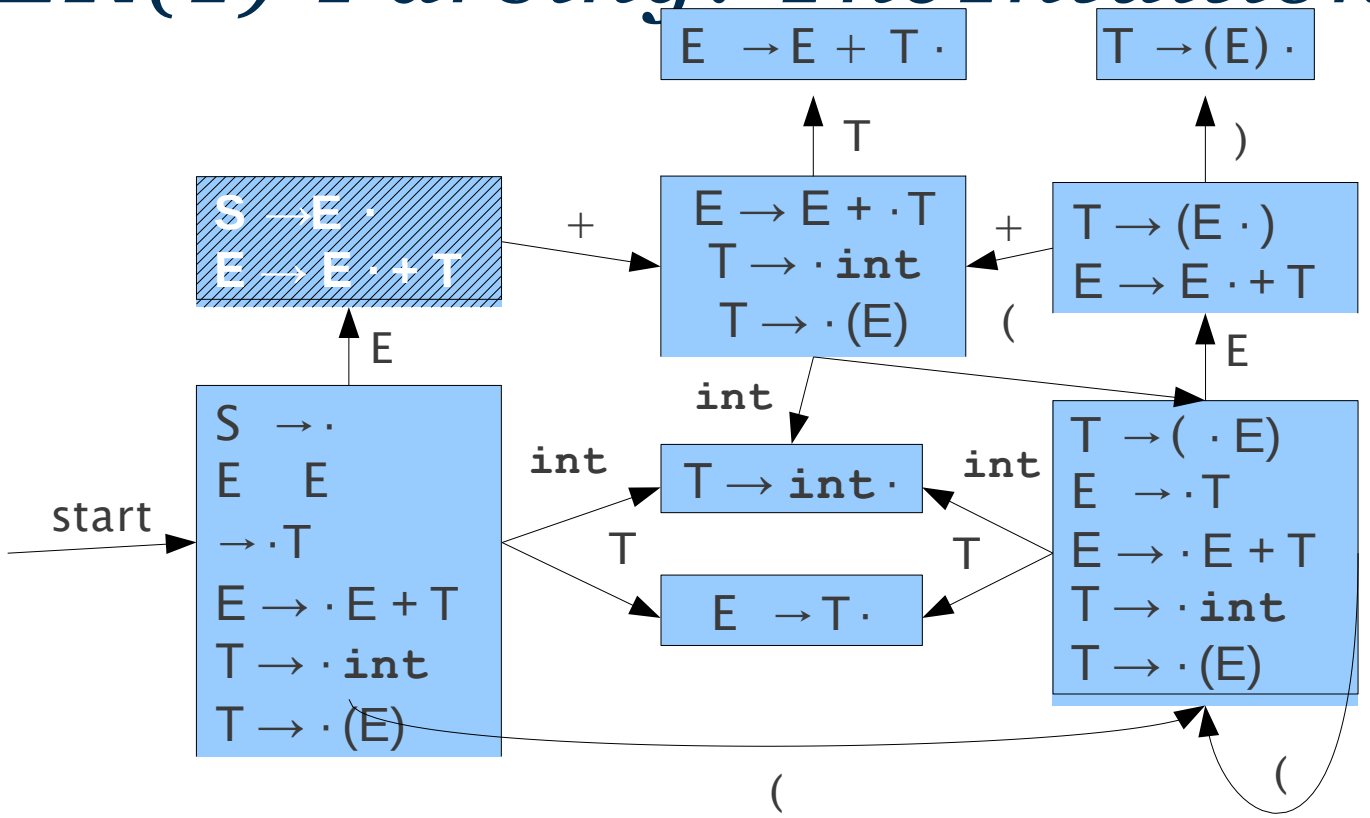
S → E
E → T
E → E + T
T → int
T → (E)



int	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



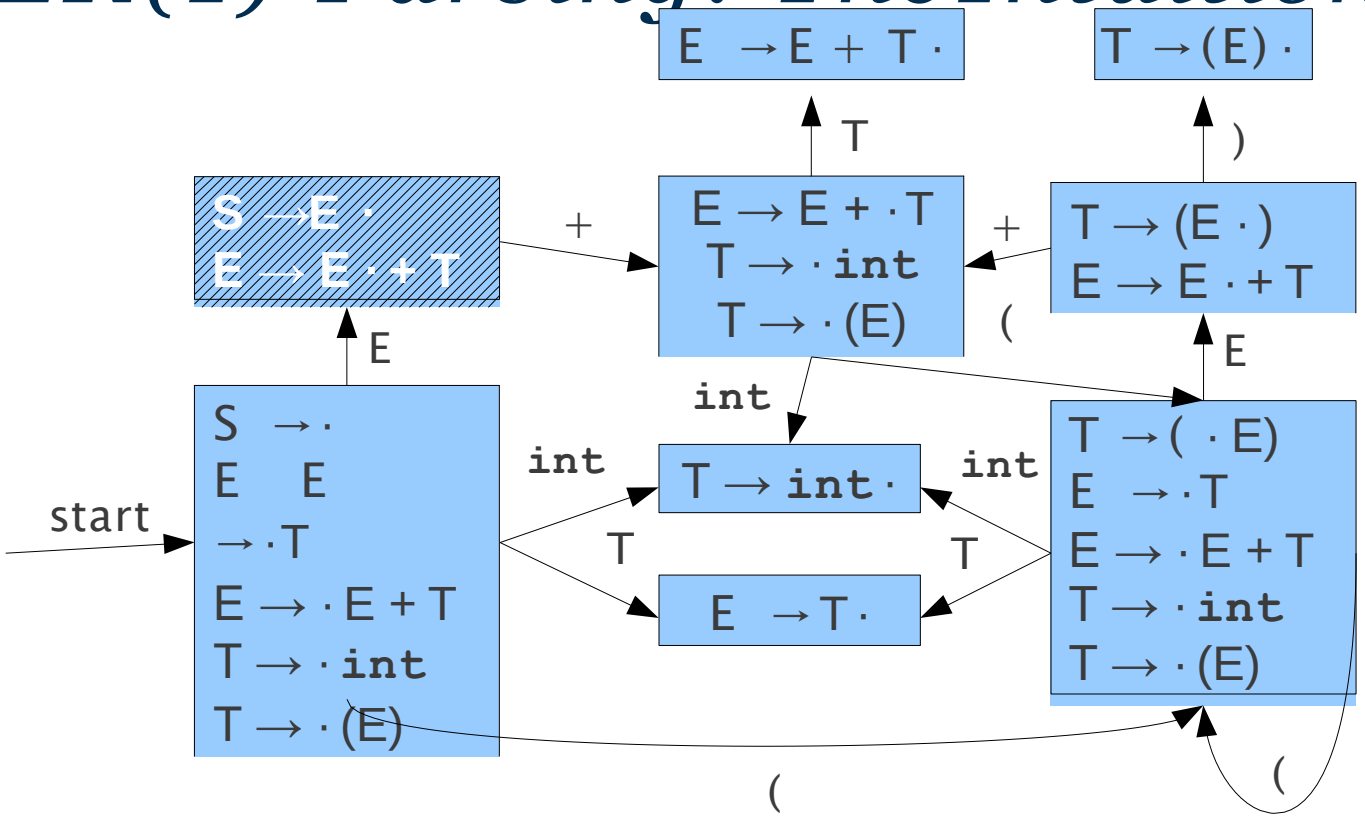
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$



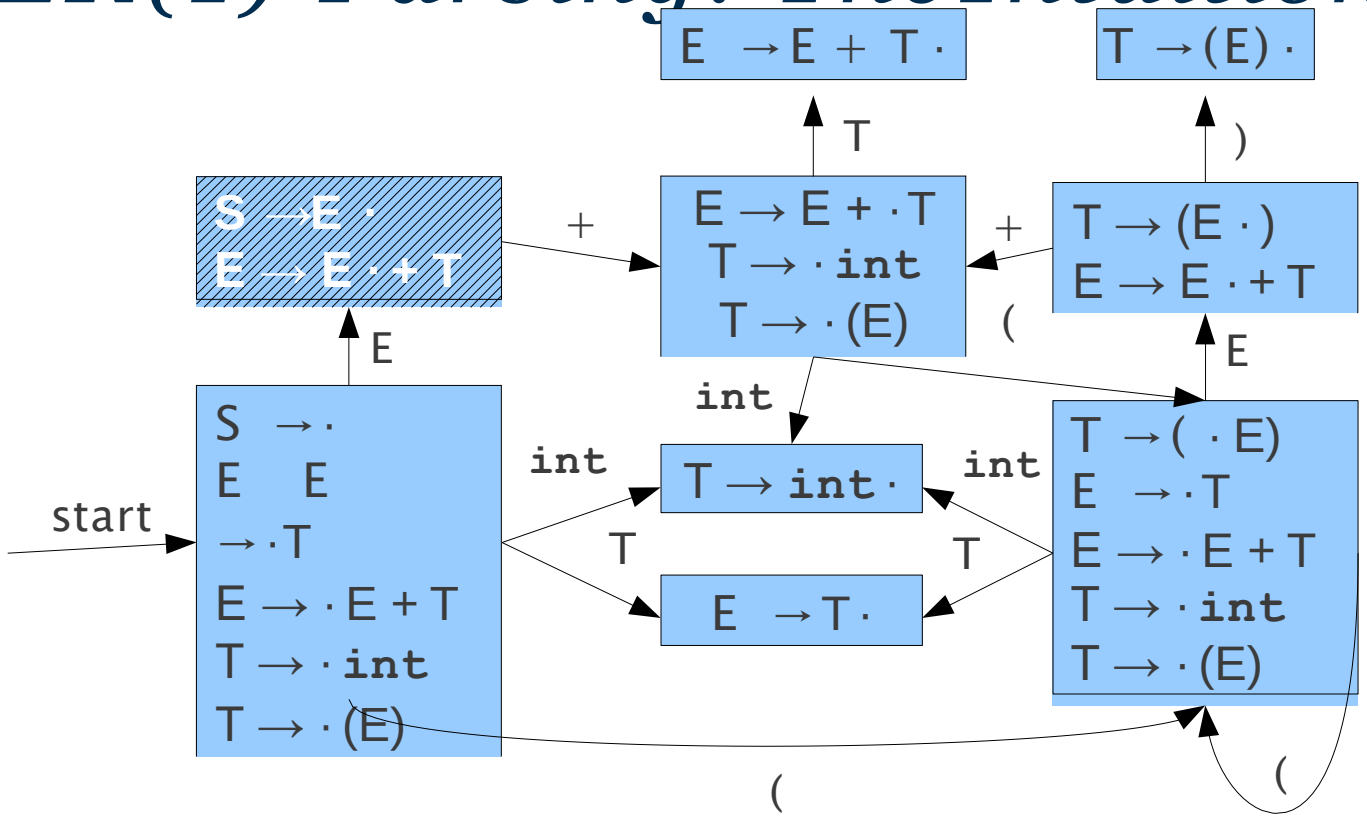
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$
E → · T	+

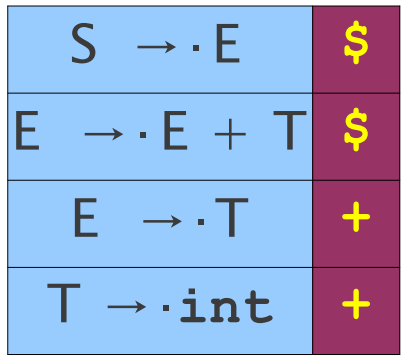


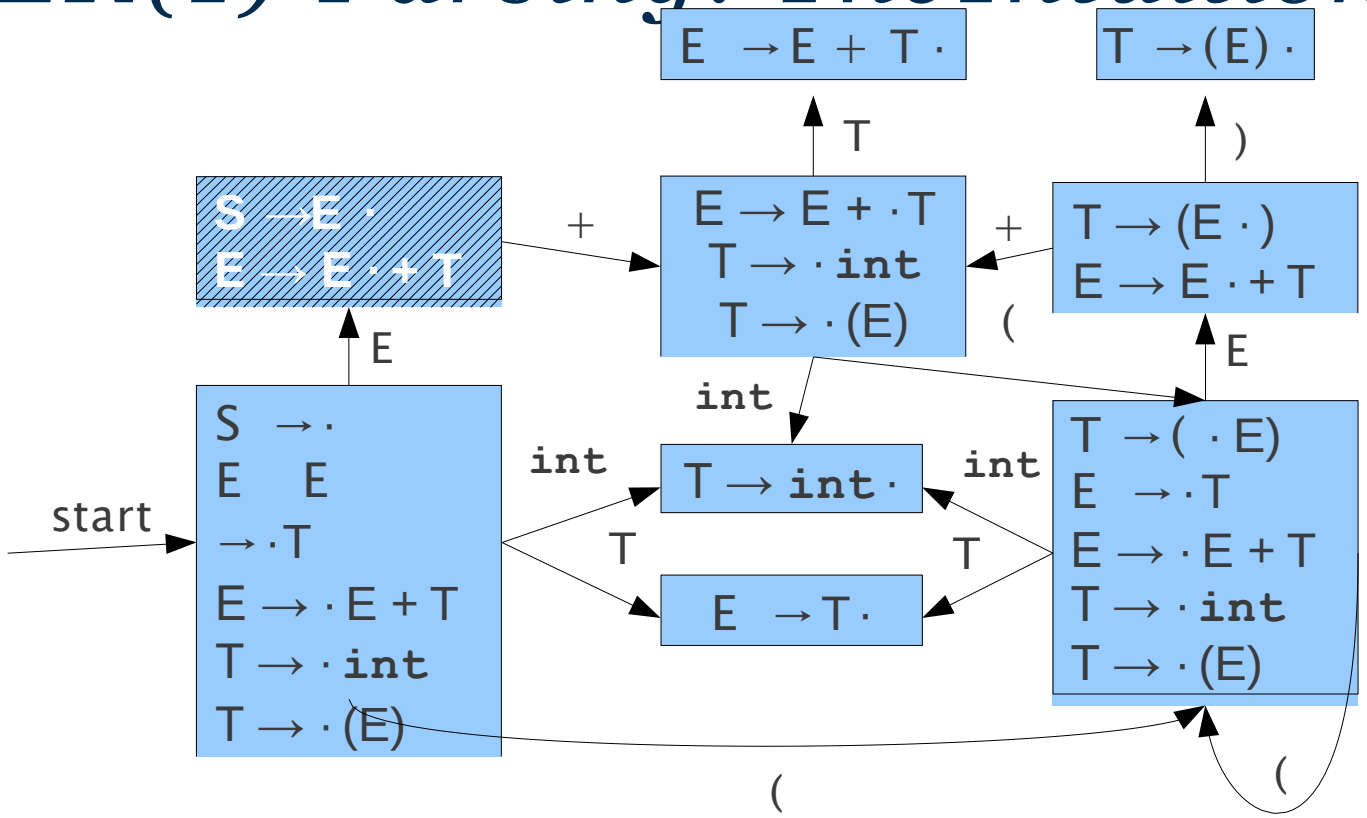
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)





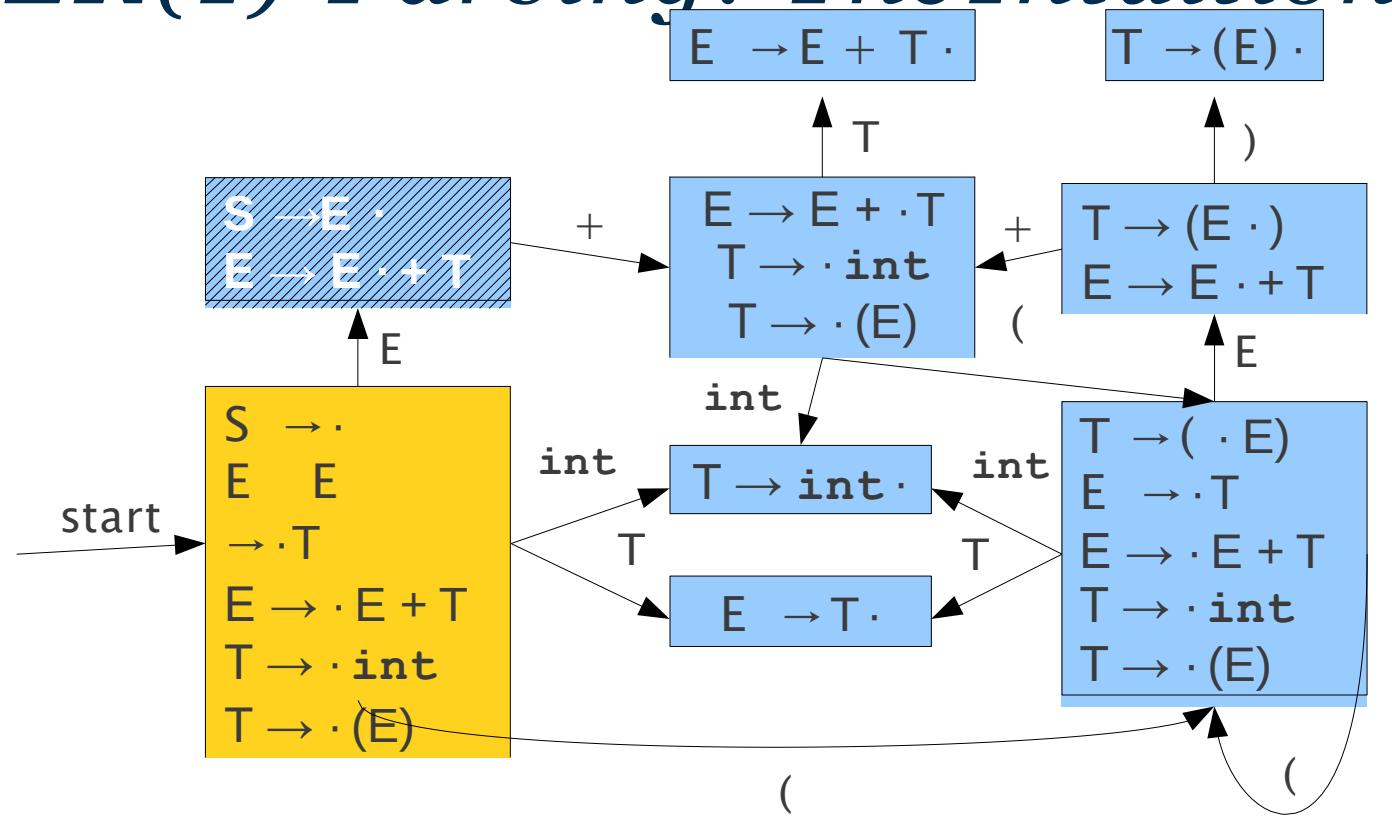
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$
E → · T	+
T → · int	+

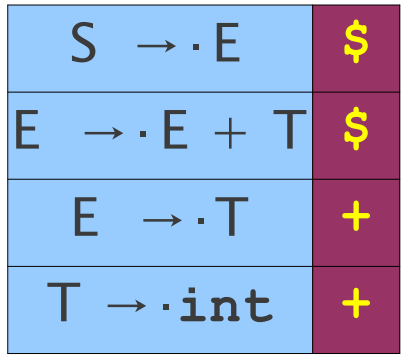


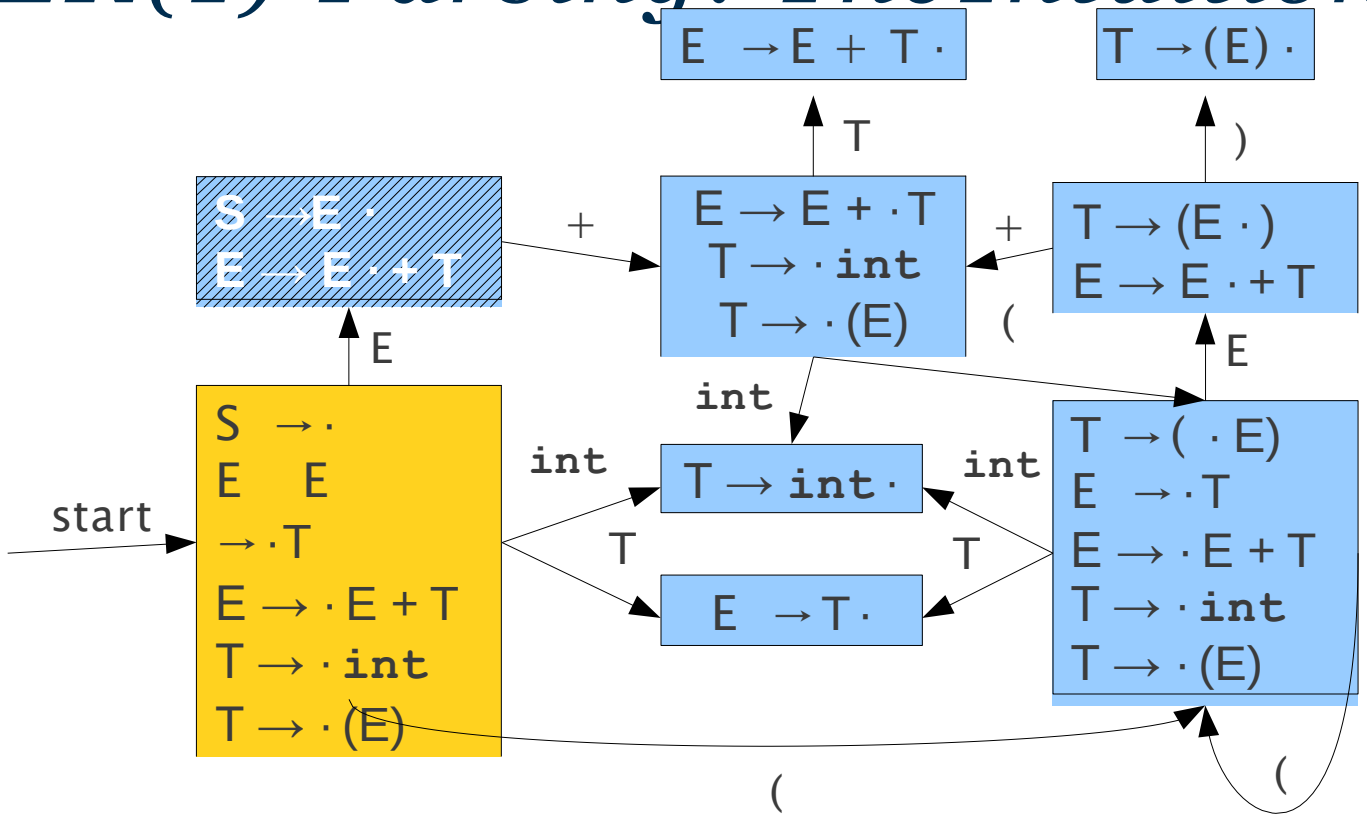
int	+	(	int	+	int	+	int	)	$

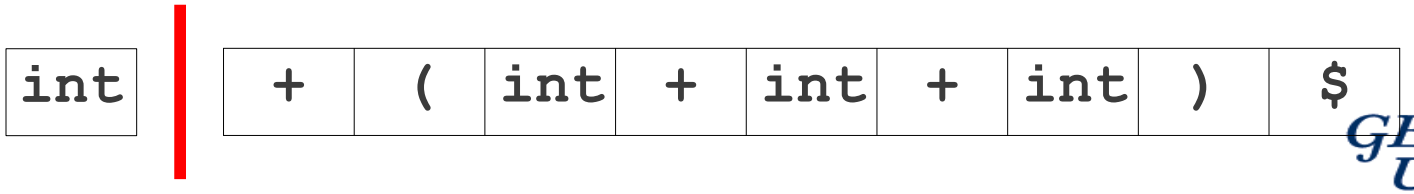
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



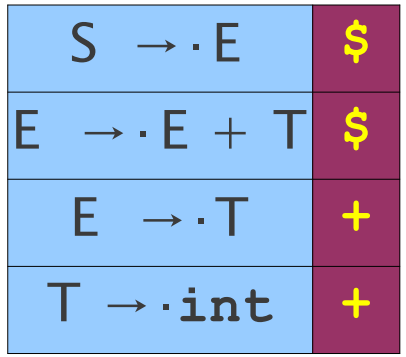


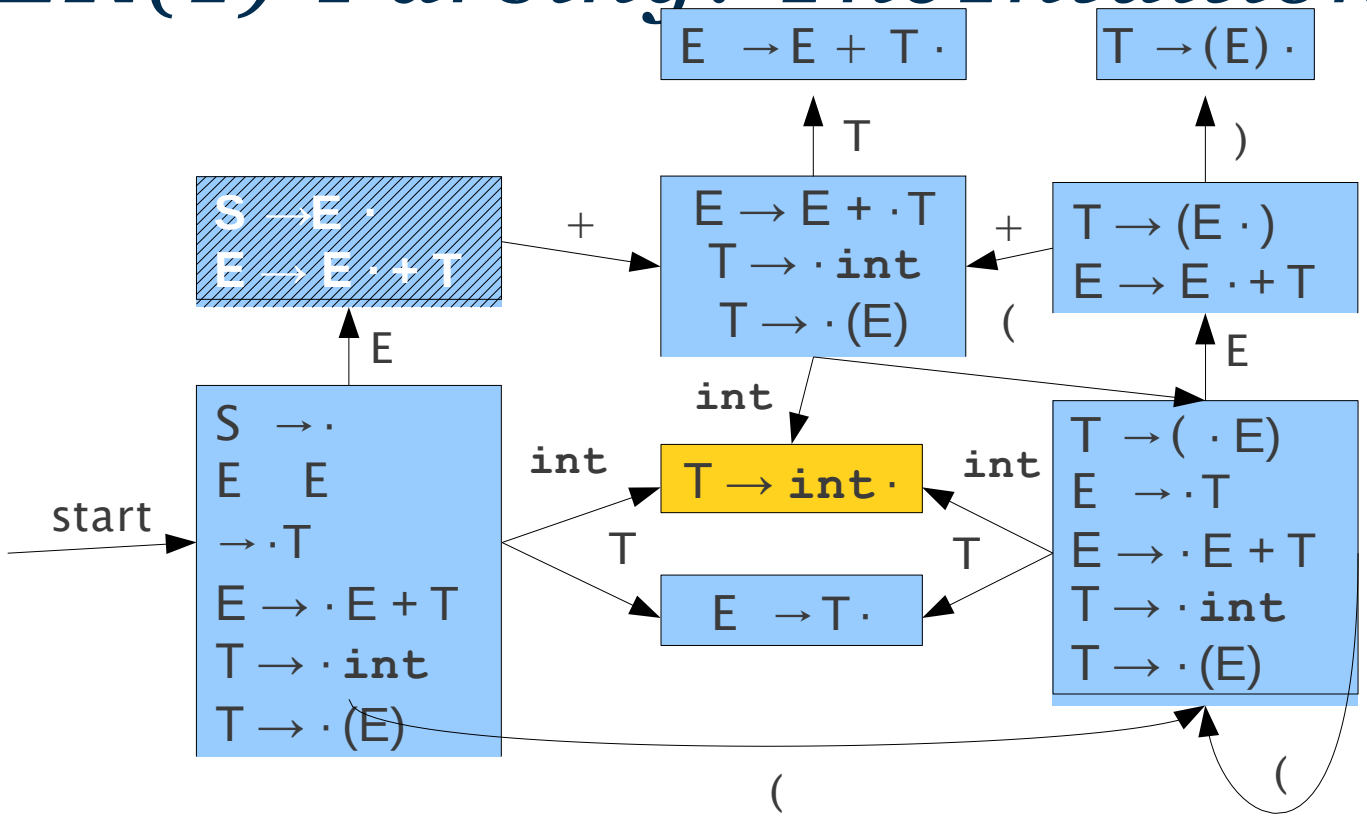


GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



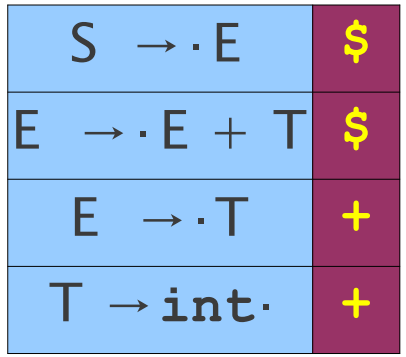


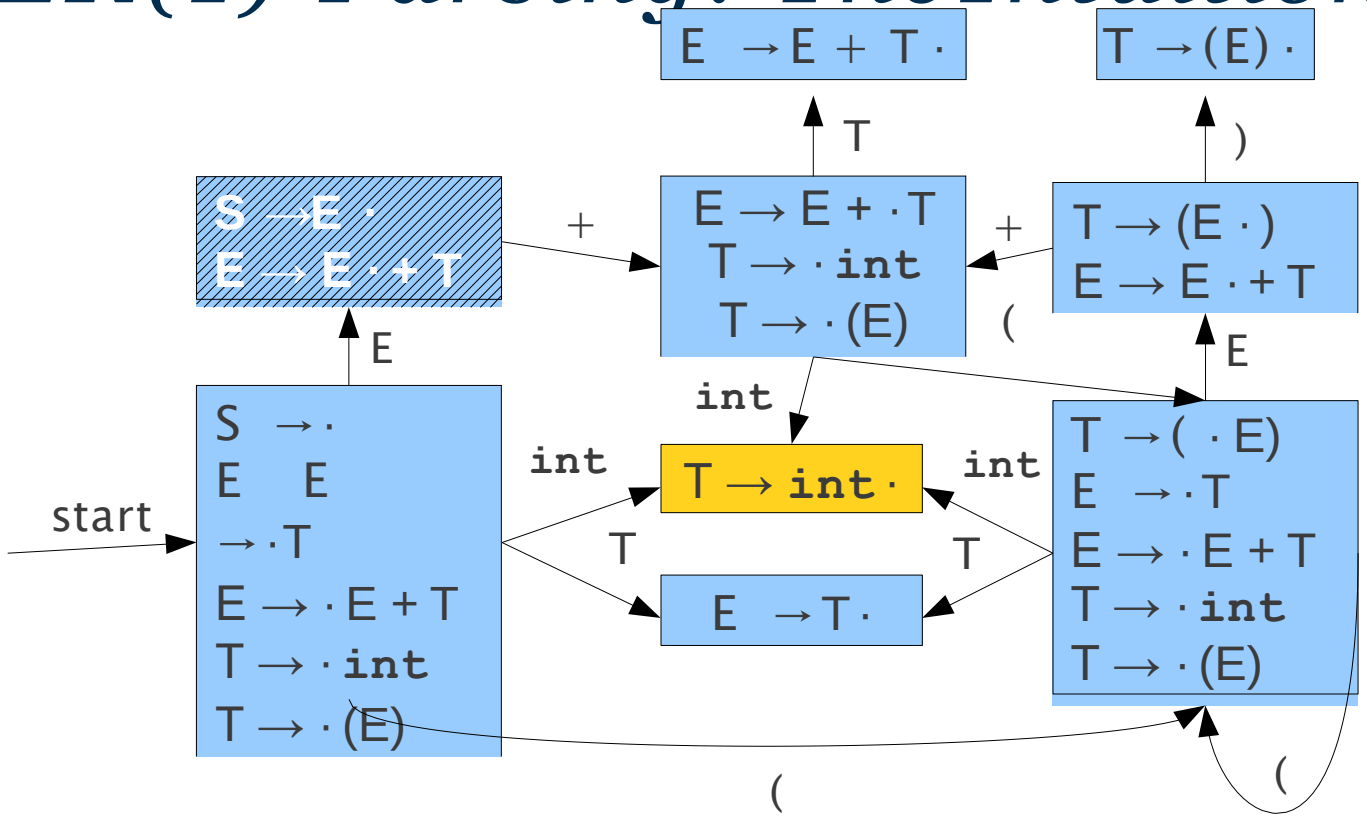
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)





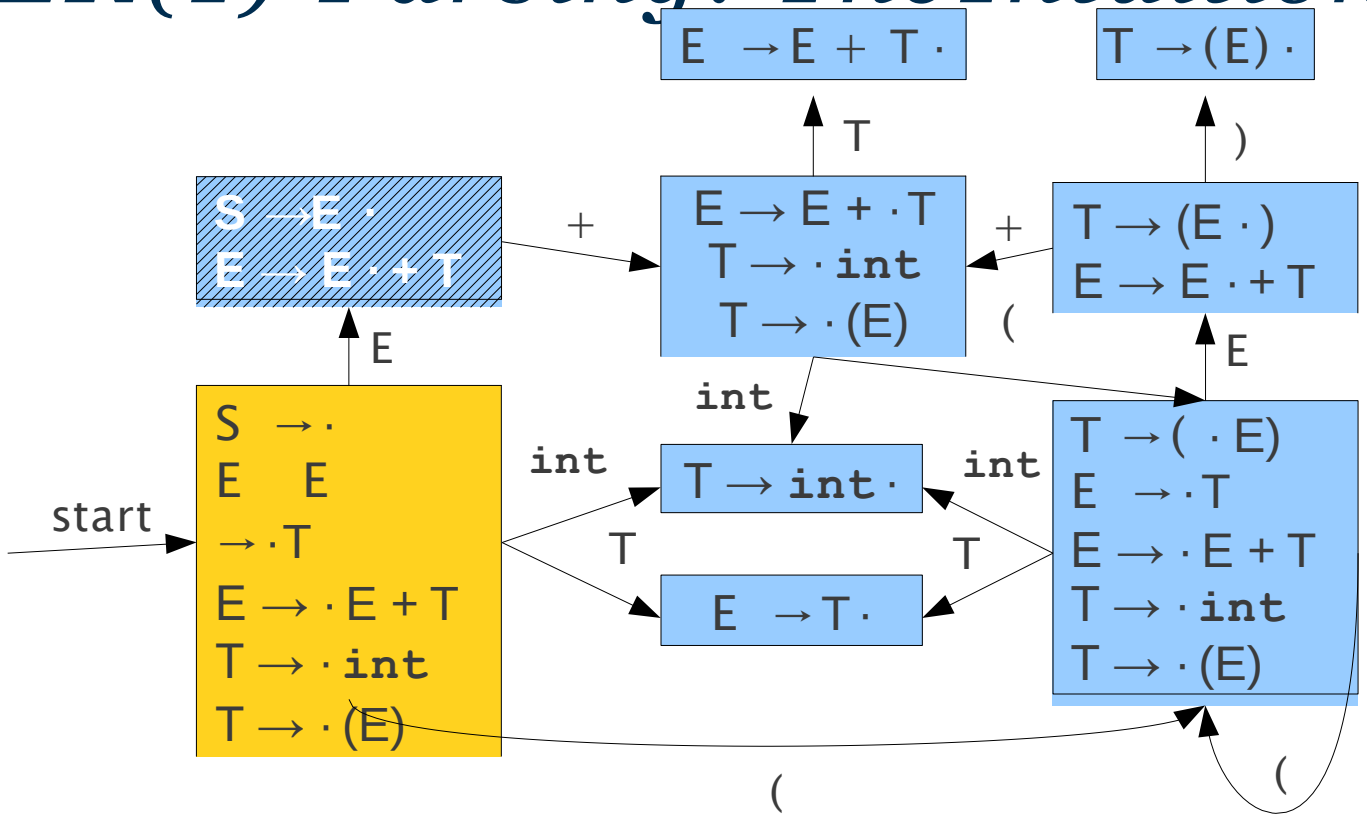
int	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$
E → · T	+

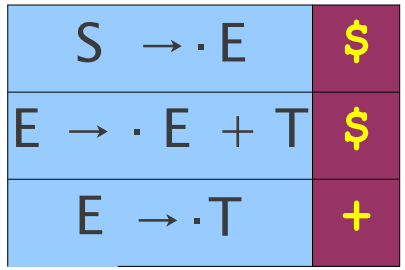


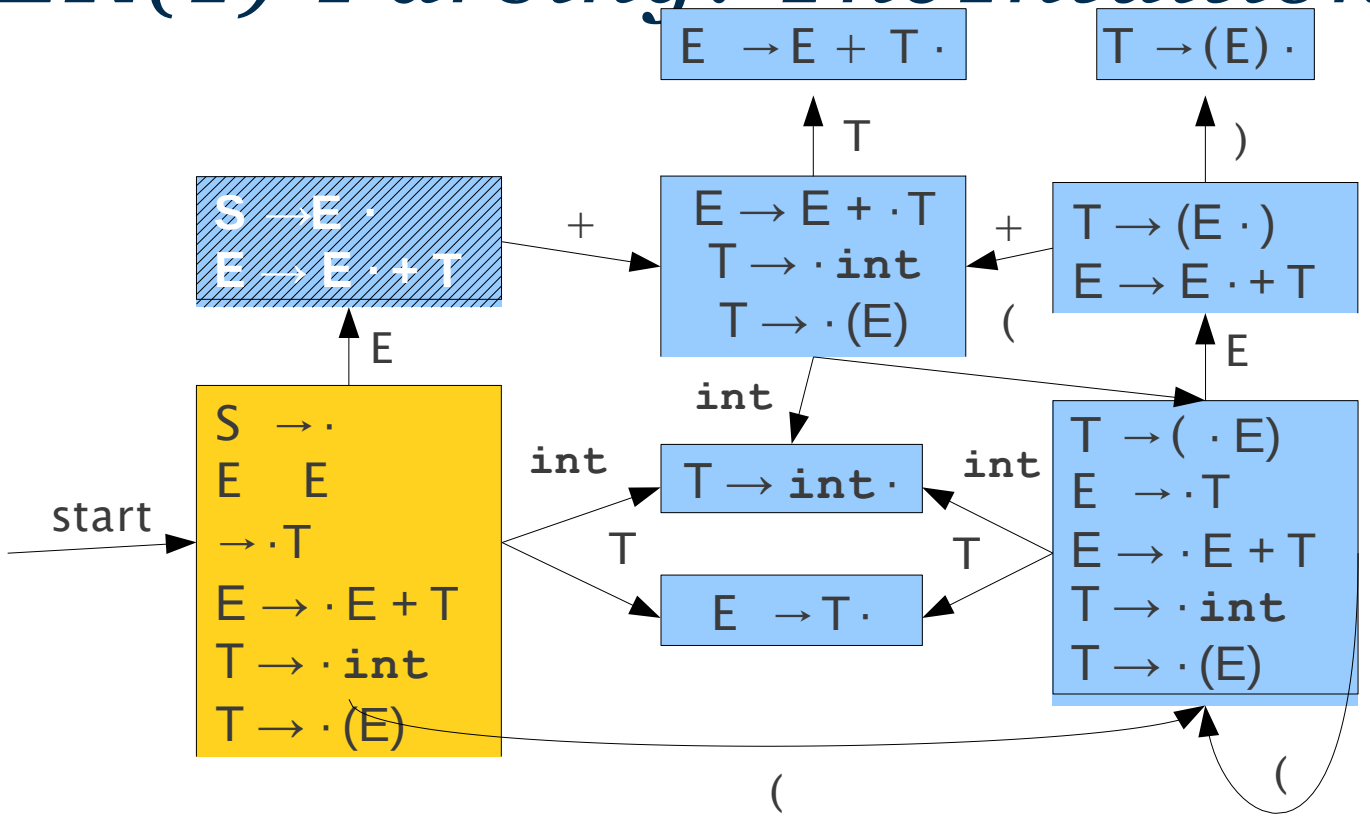
+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



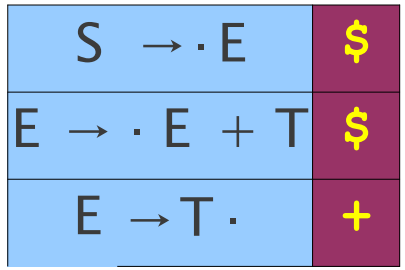


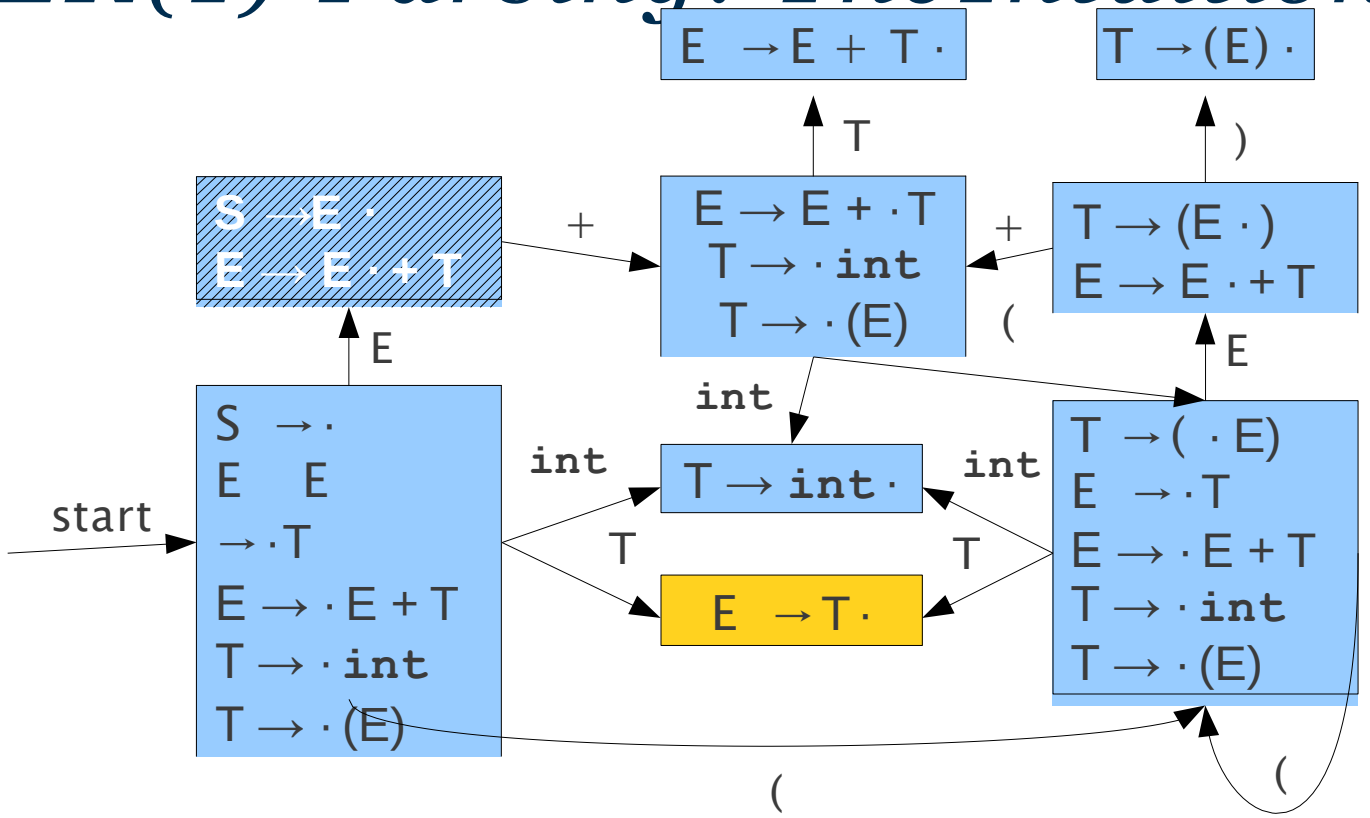
T	+	(	int	+	int	+	int	)	$

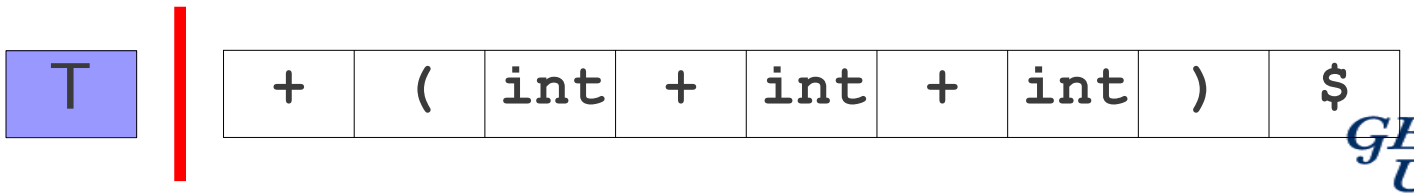
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



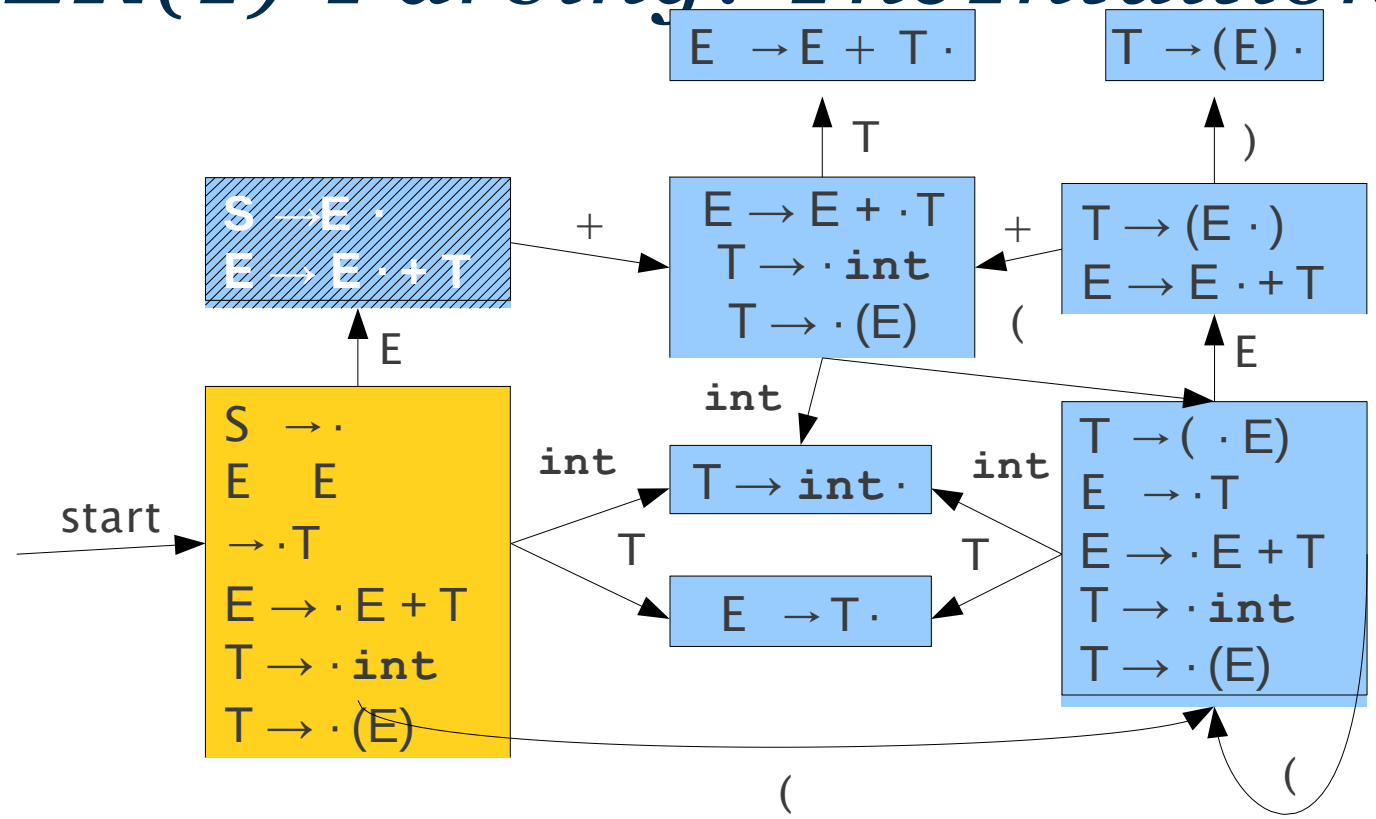




LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

S → · E	$
E → · E + T	$

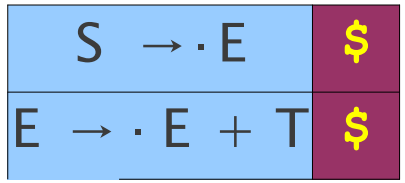


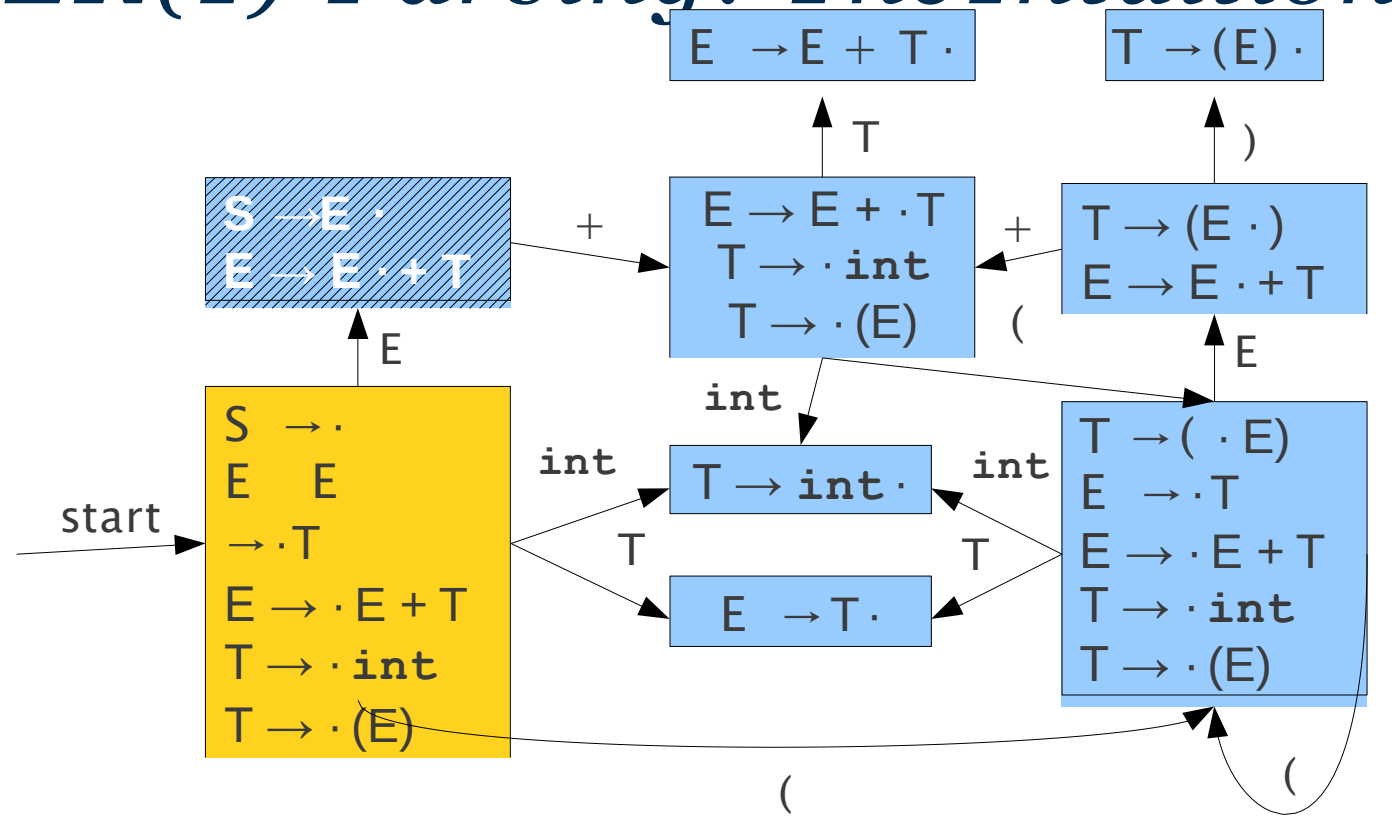
+	(	int	+	int	+	int	)	$

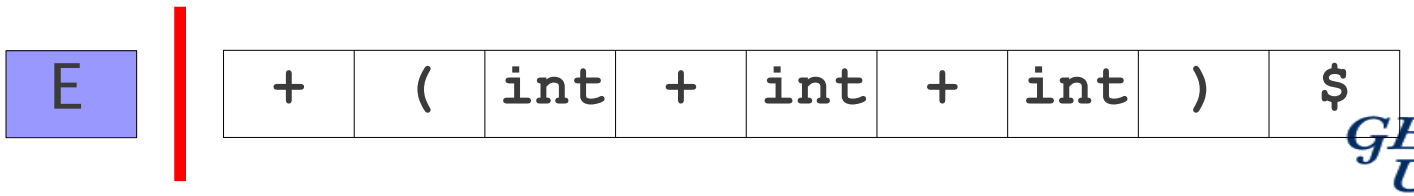
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

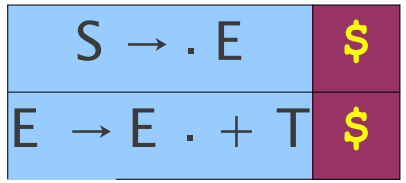


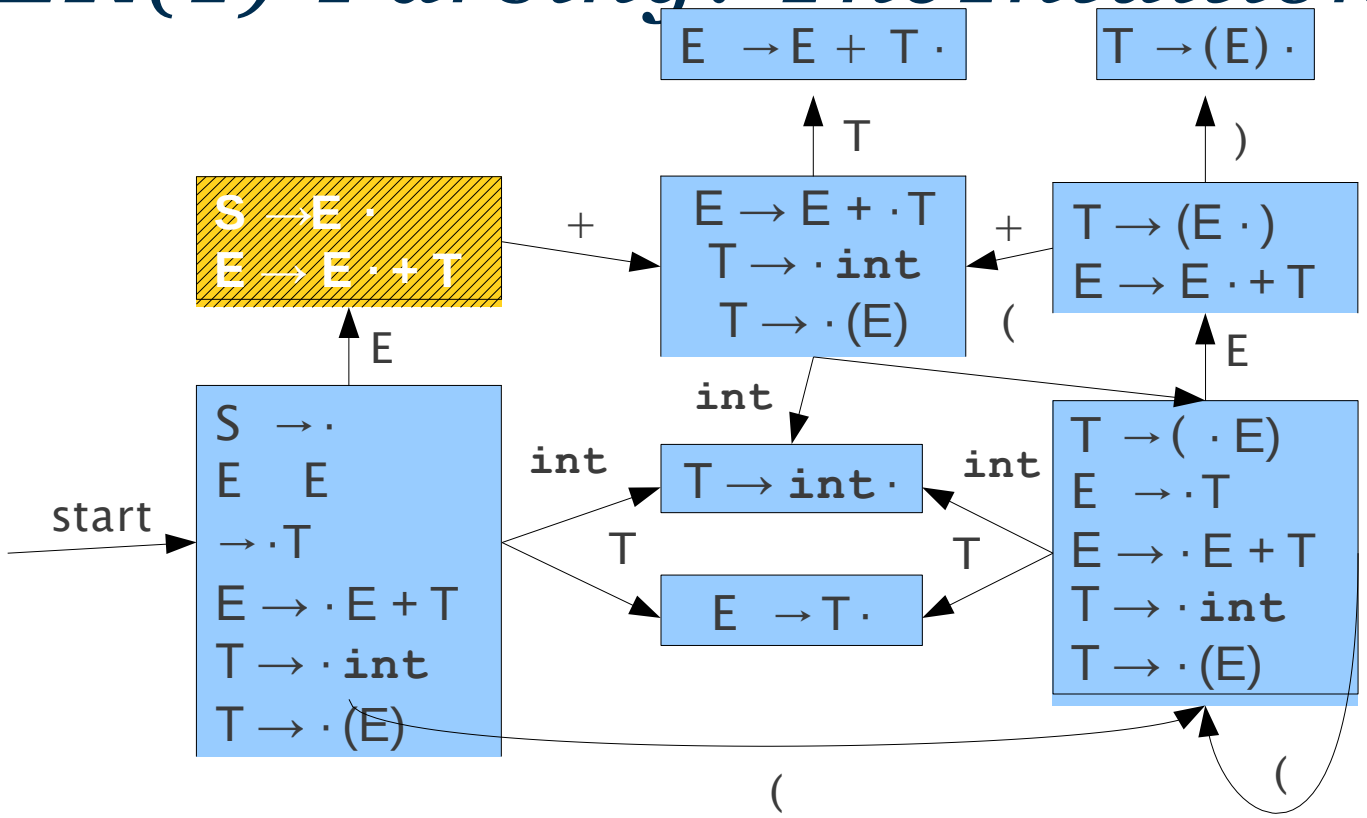


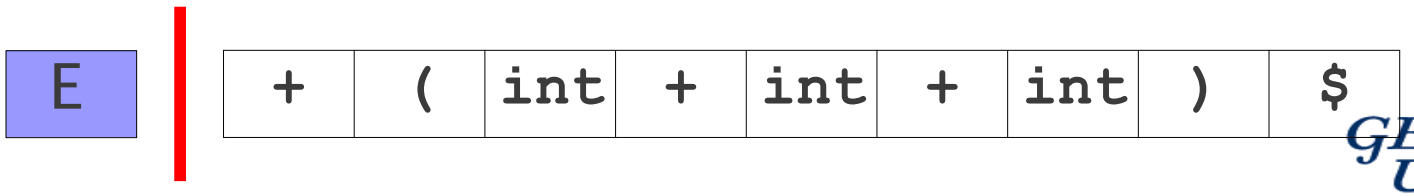


LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

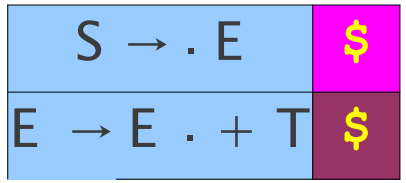


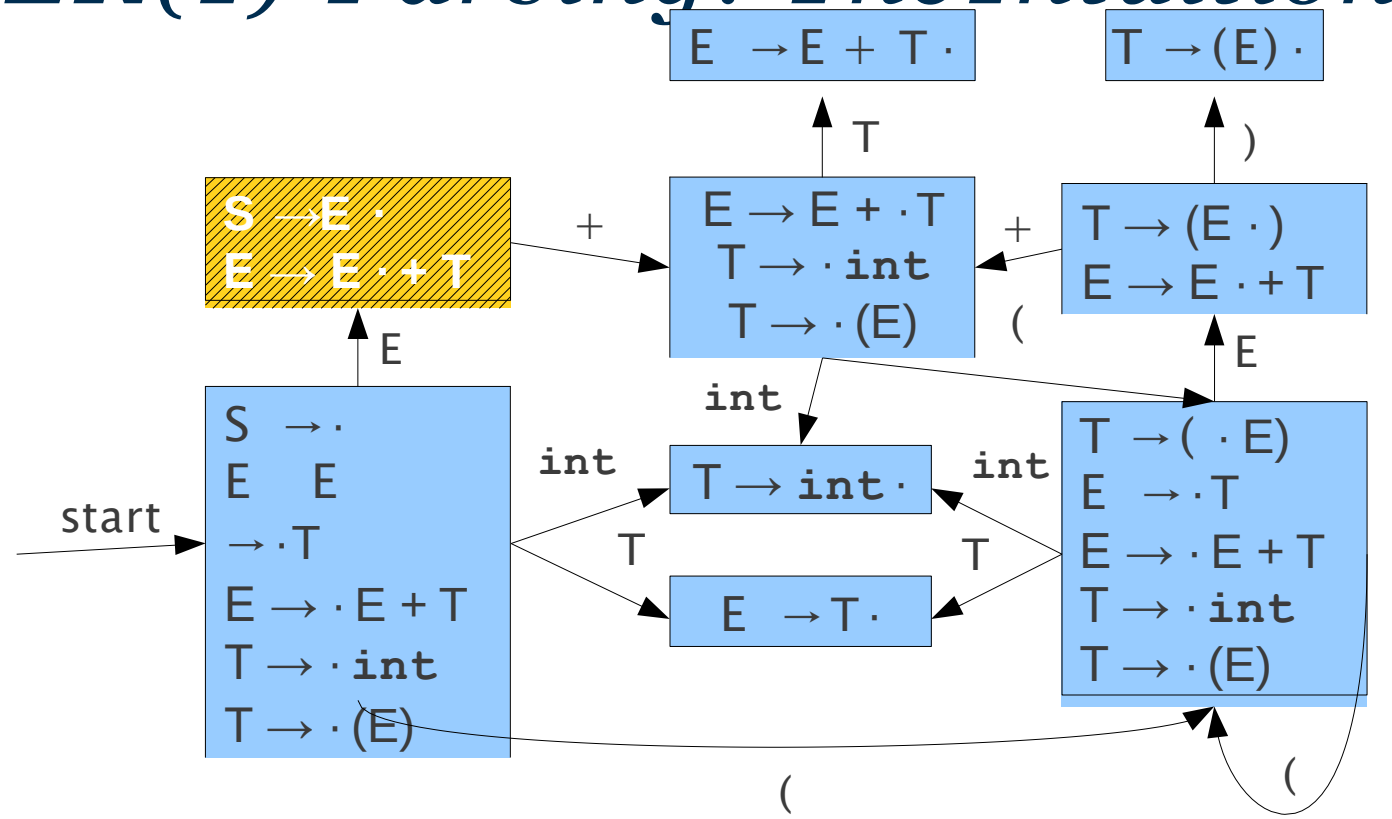


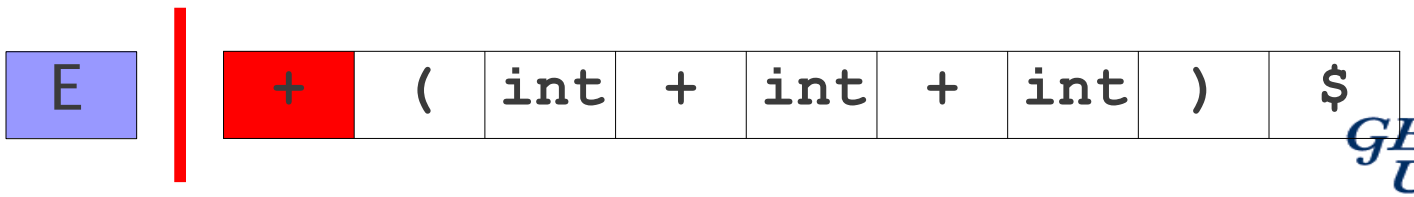


LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



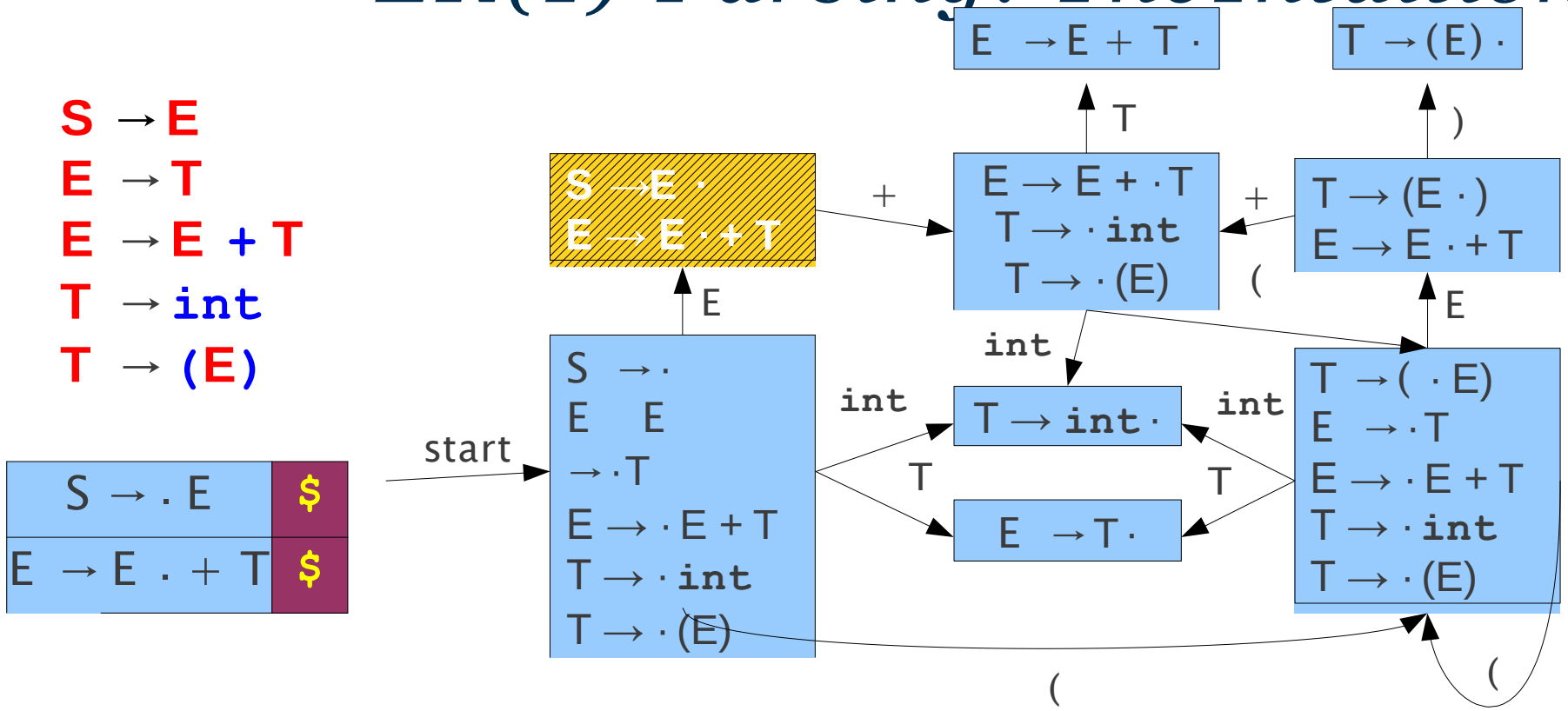




GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



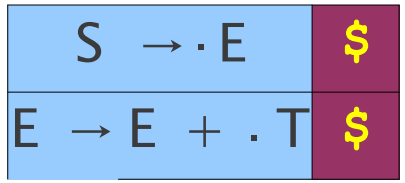
S → · E	$
E → E · + T	$

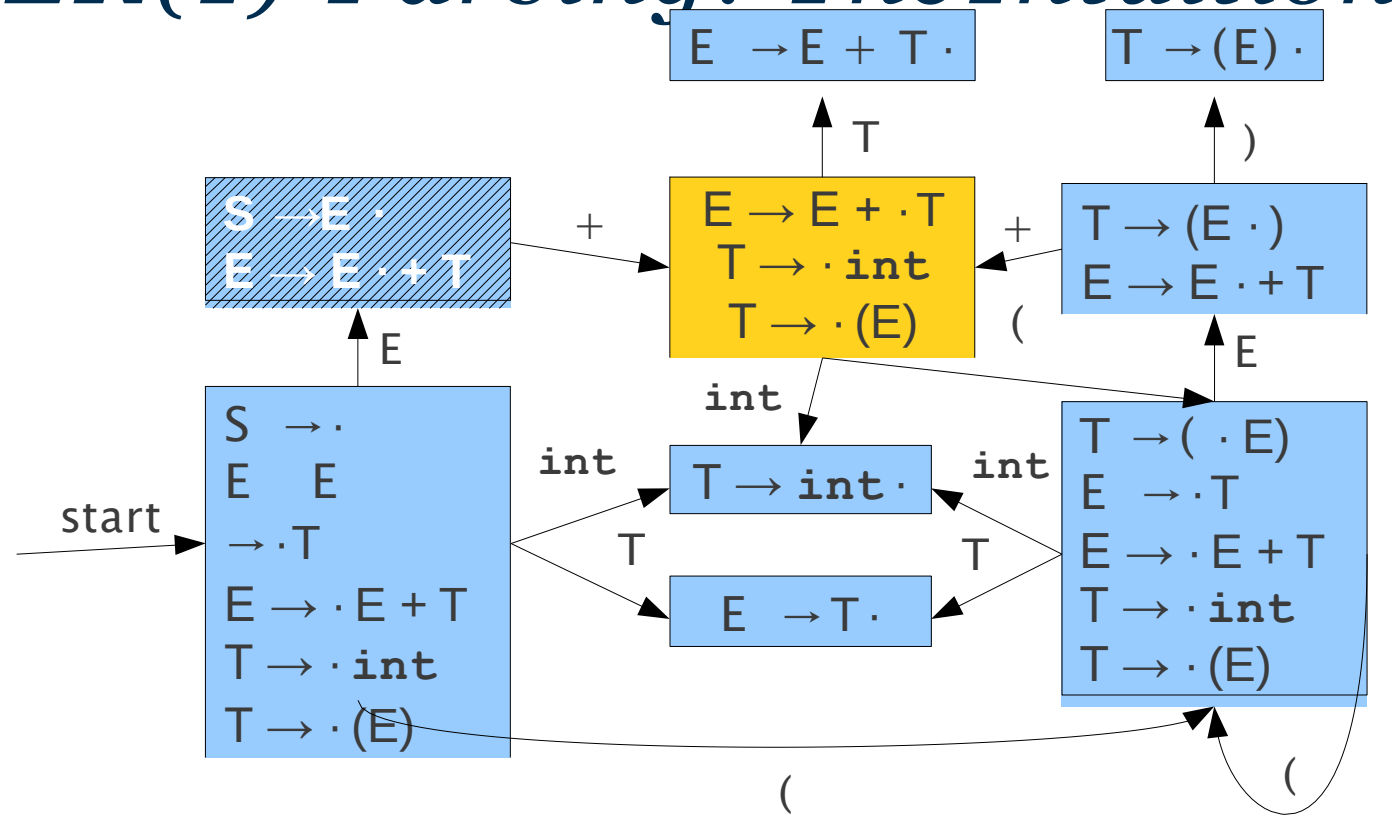
E	+	(	int	+	int	+	int	)	$

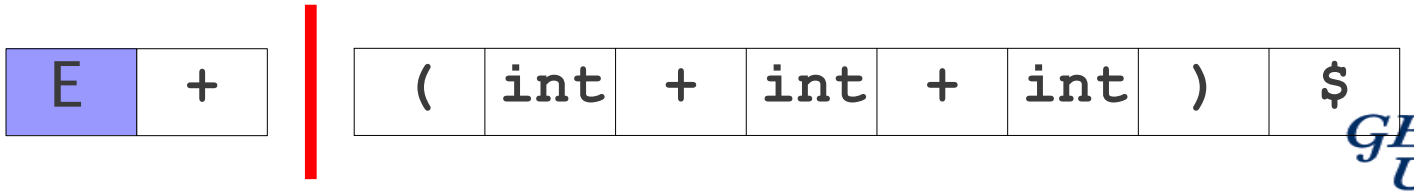
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

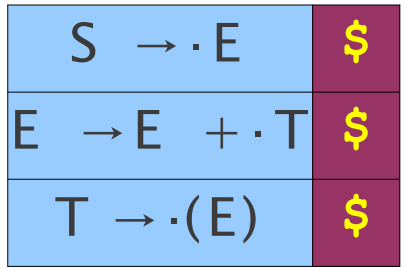


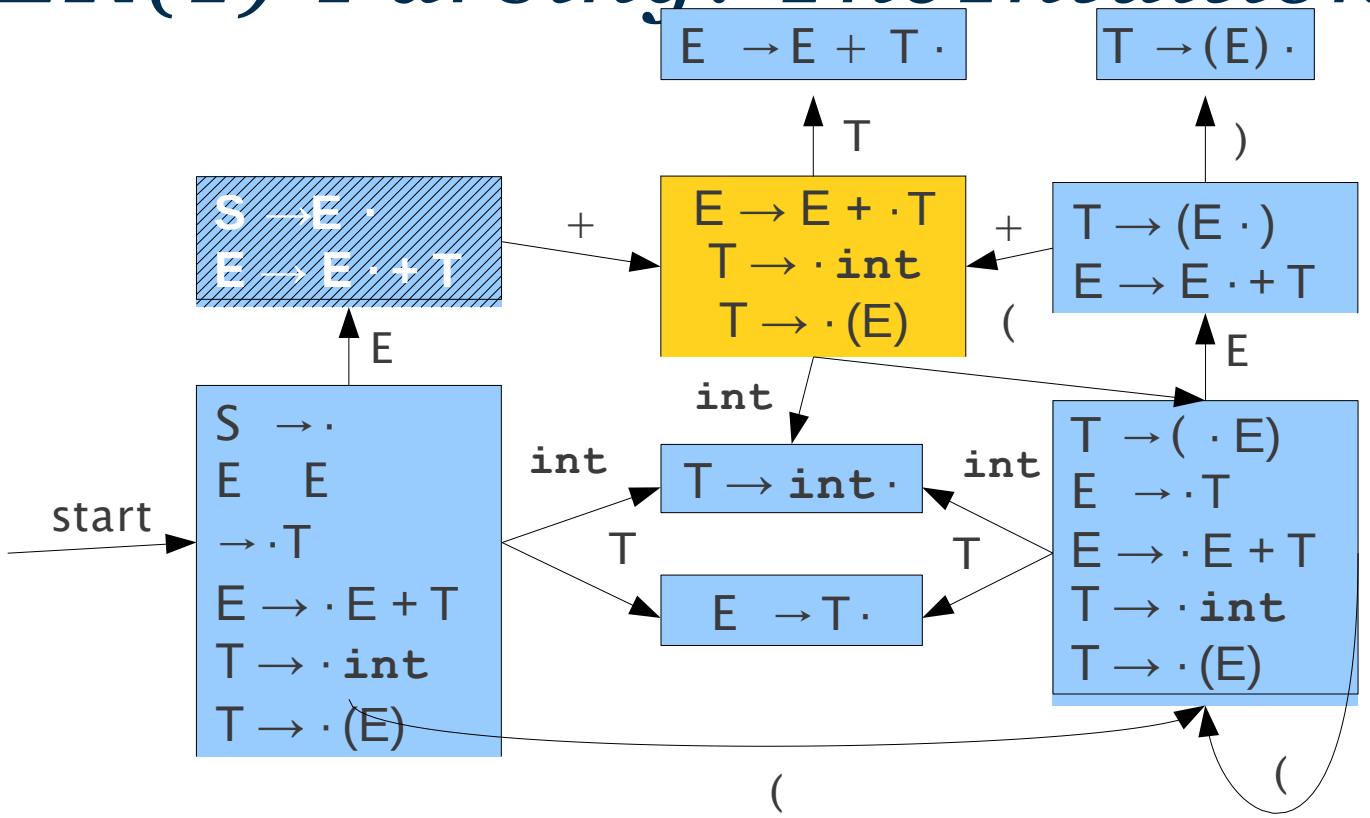


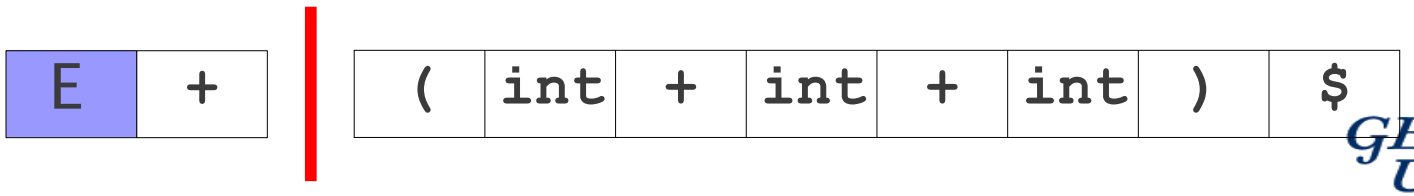


LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



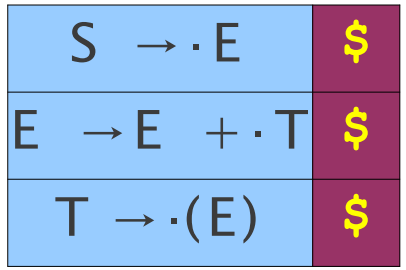


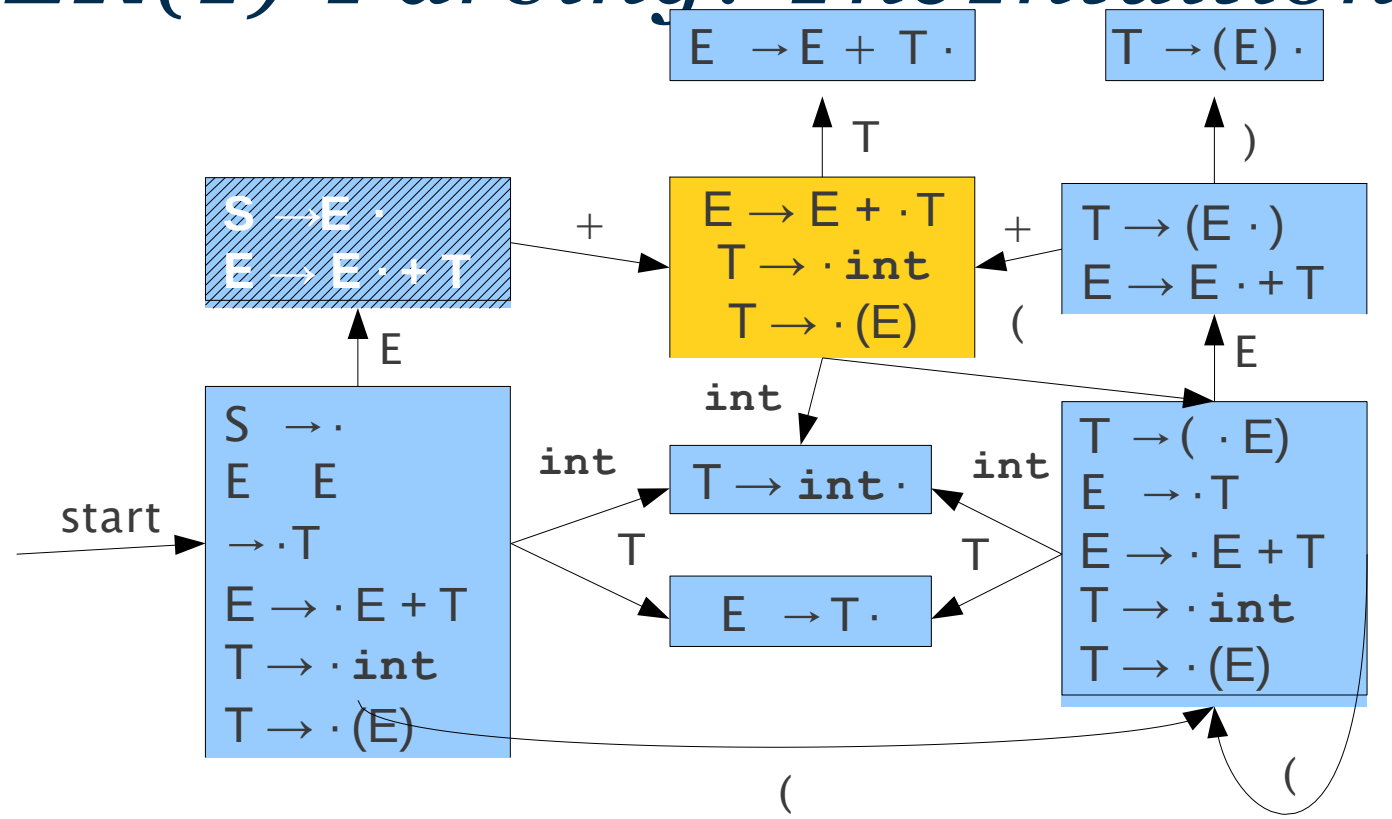


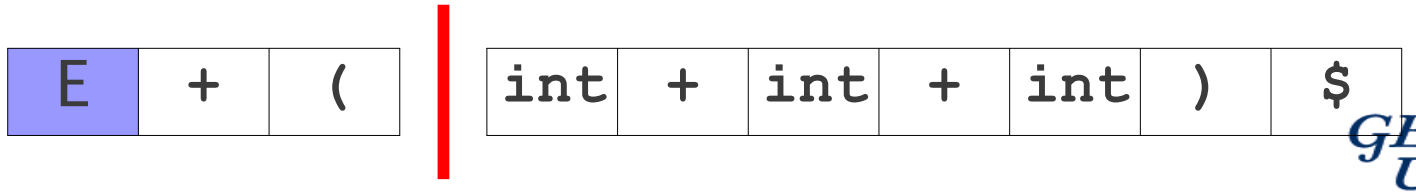
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



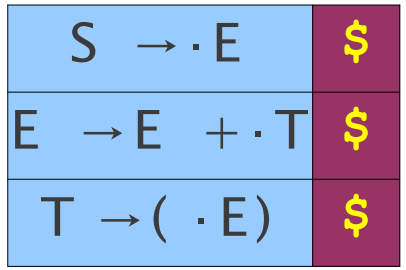


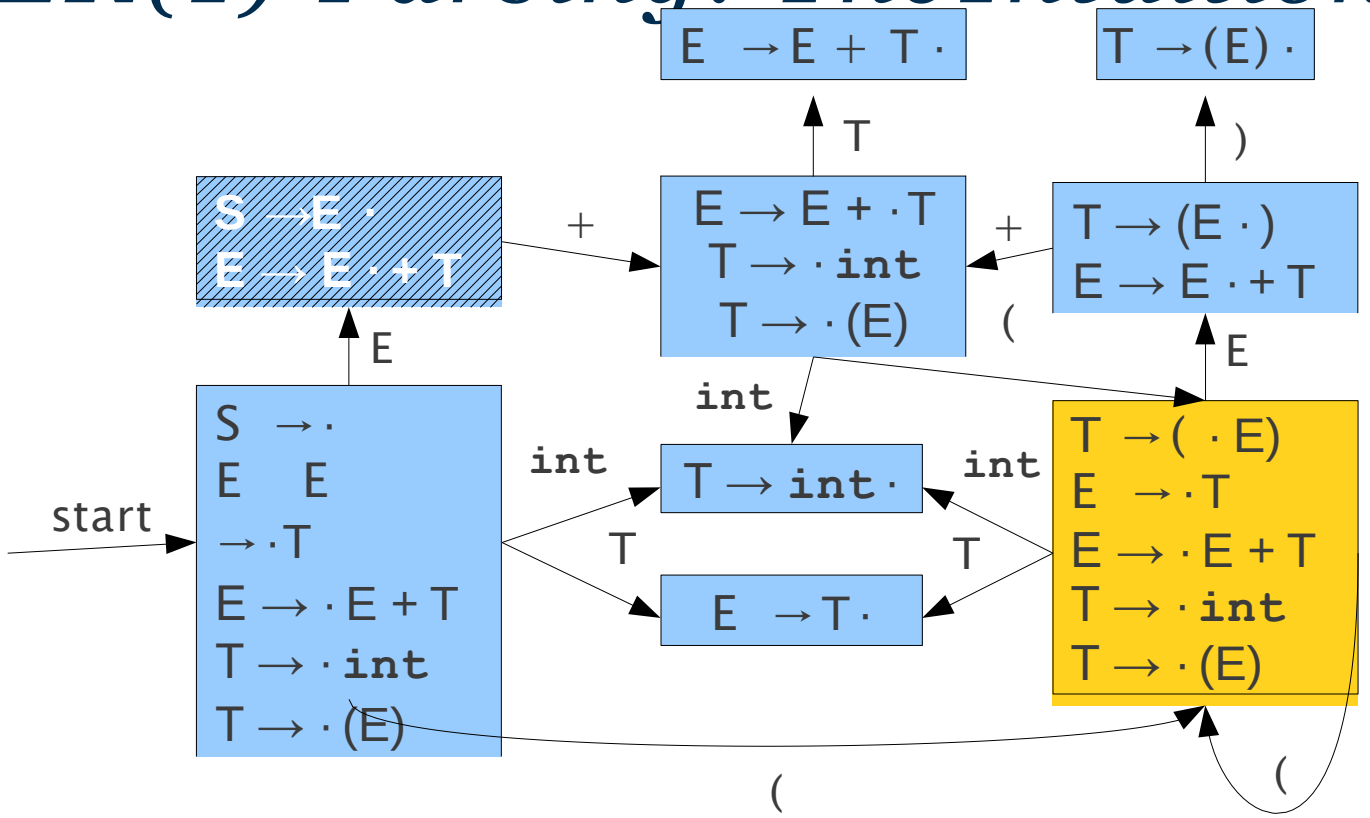


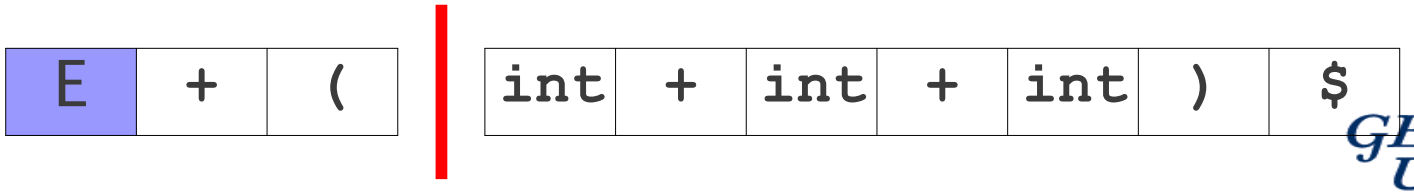
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



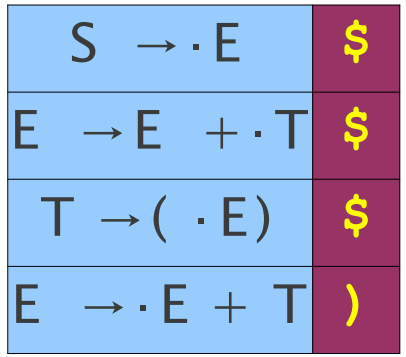


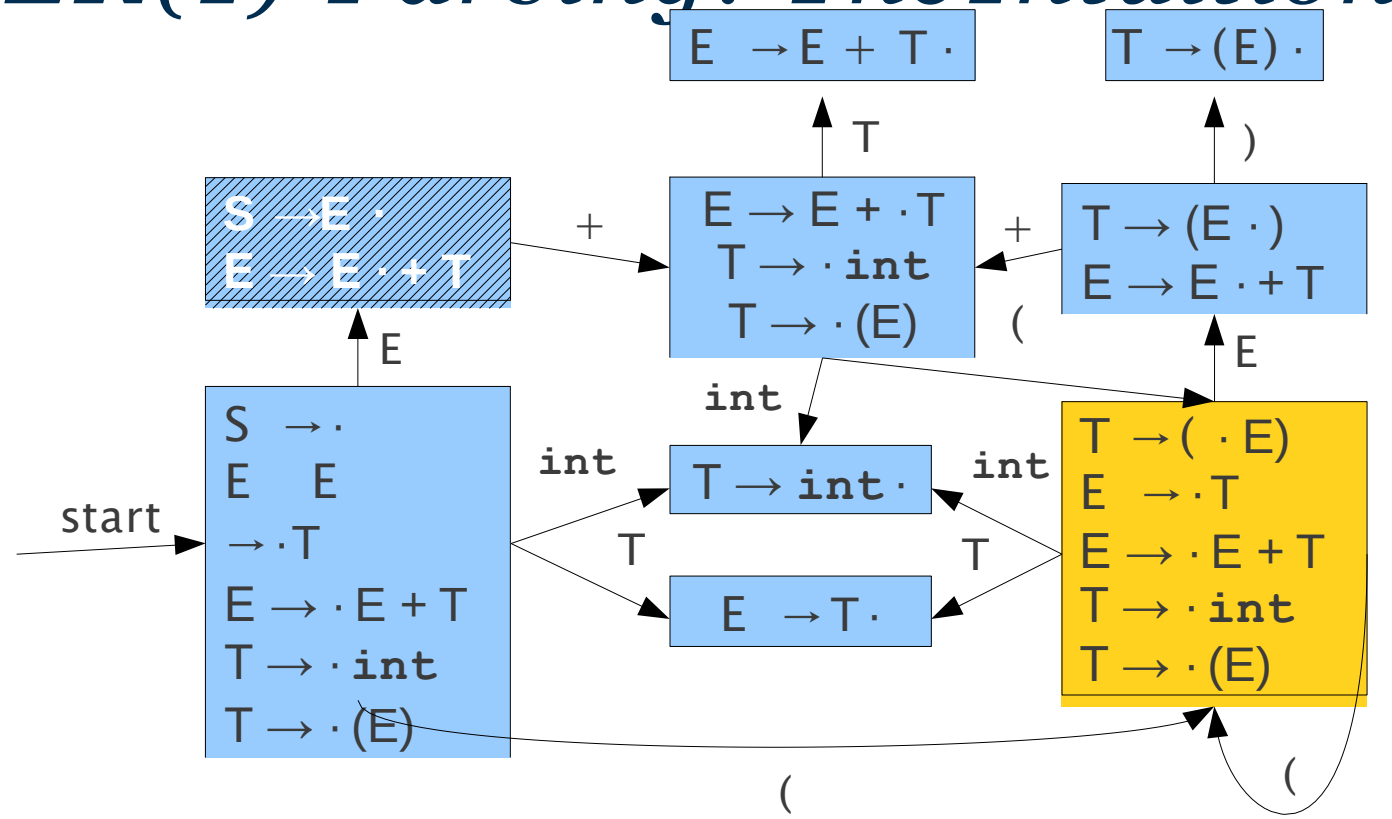


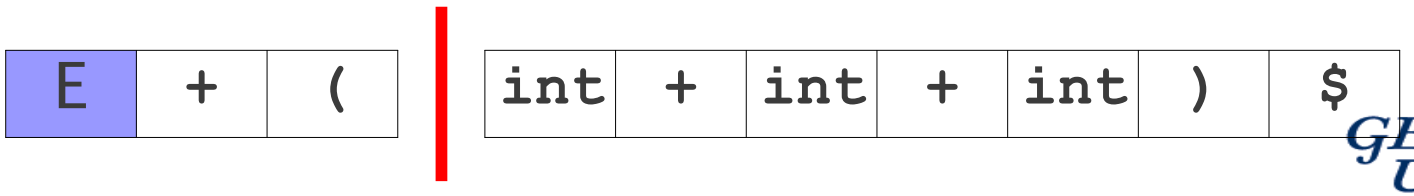
GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



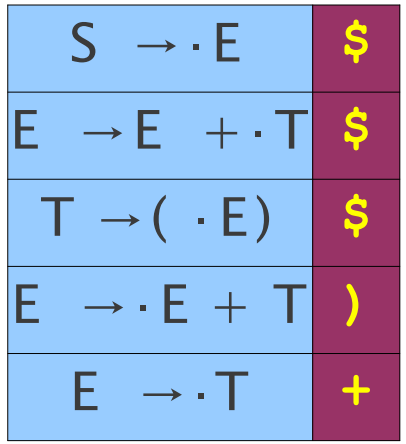


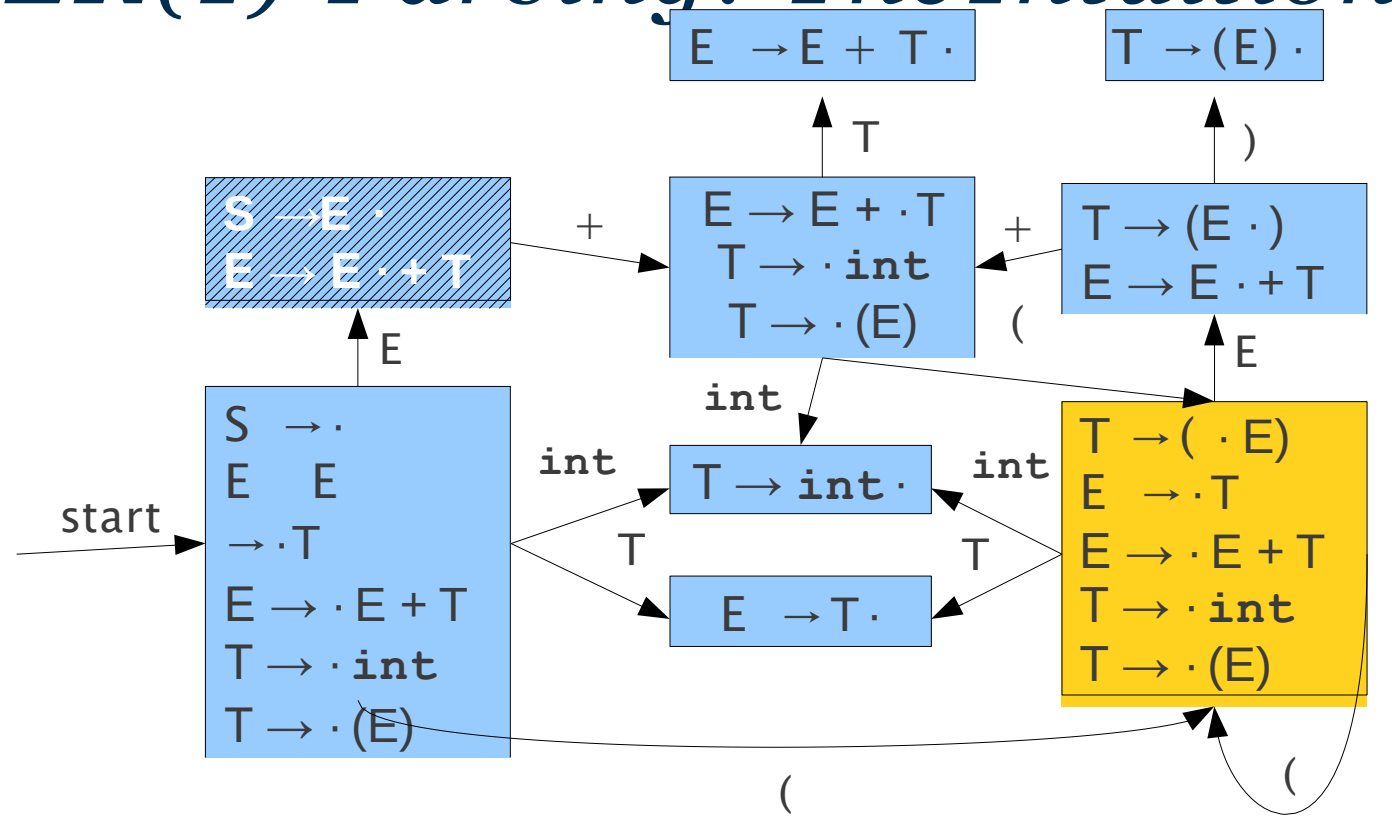


GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



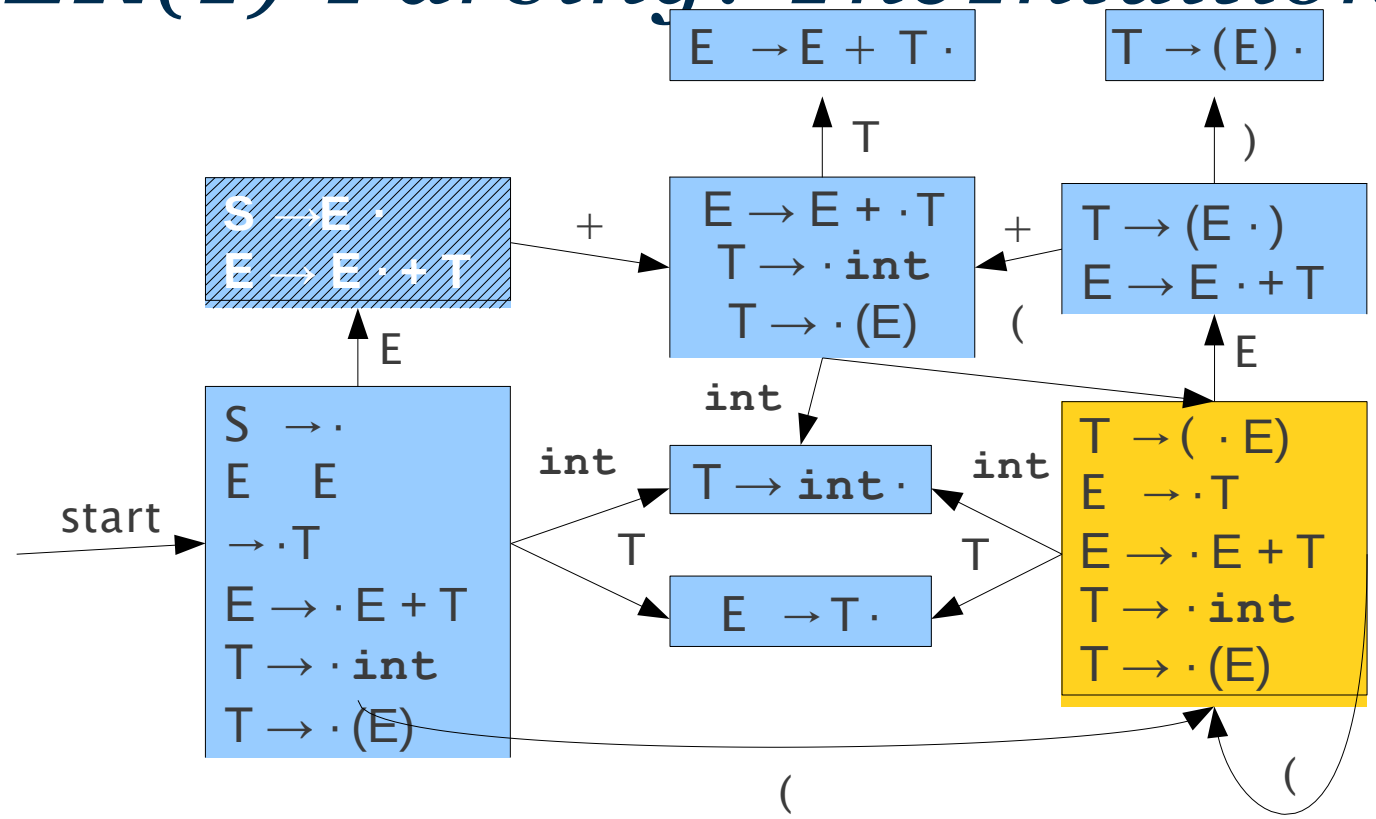


E	+	(	int	+	int	+	int	)	$

GEORGETOWN UNIVERSITY

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)

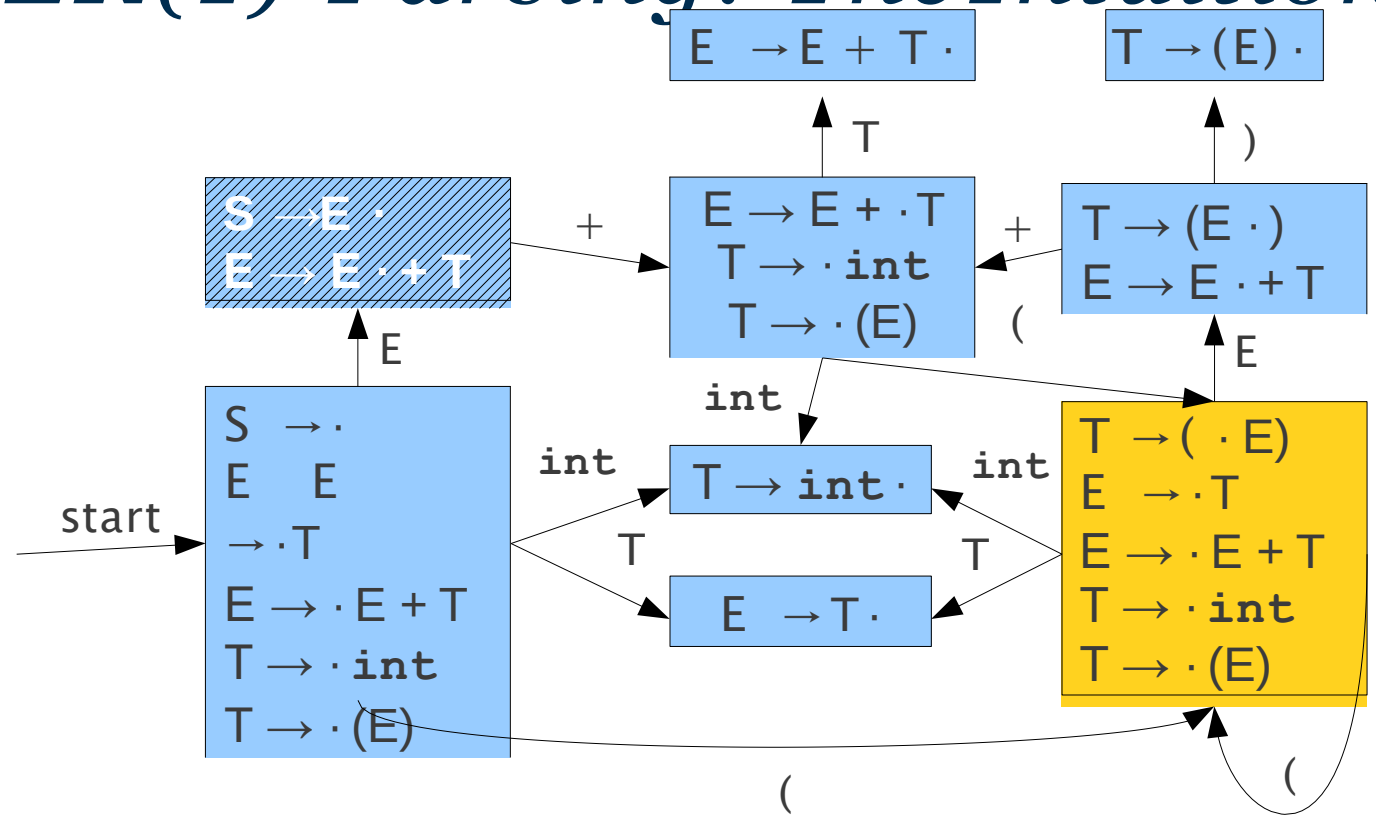


S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)
E → · T	+
T → · int	+

E	+	(		int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

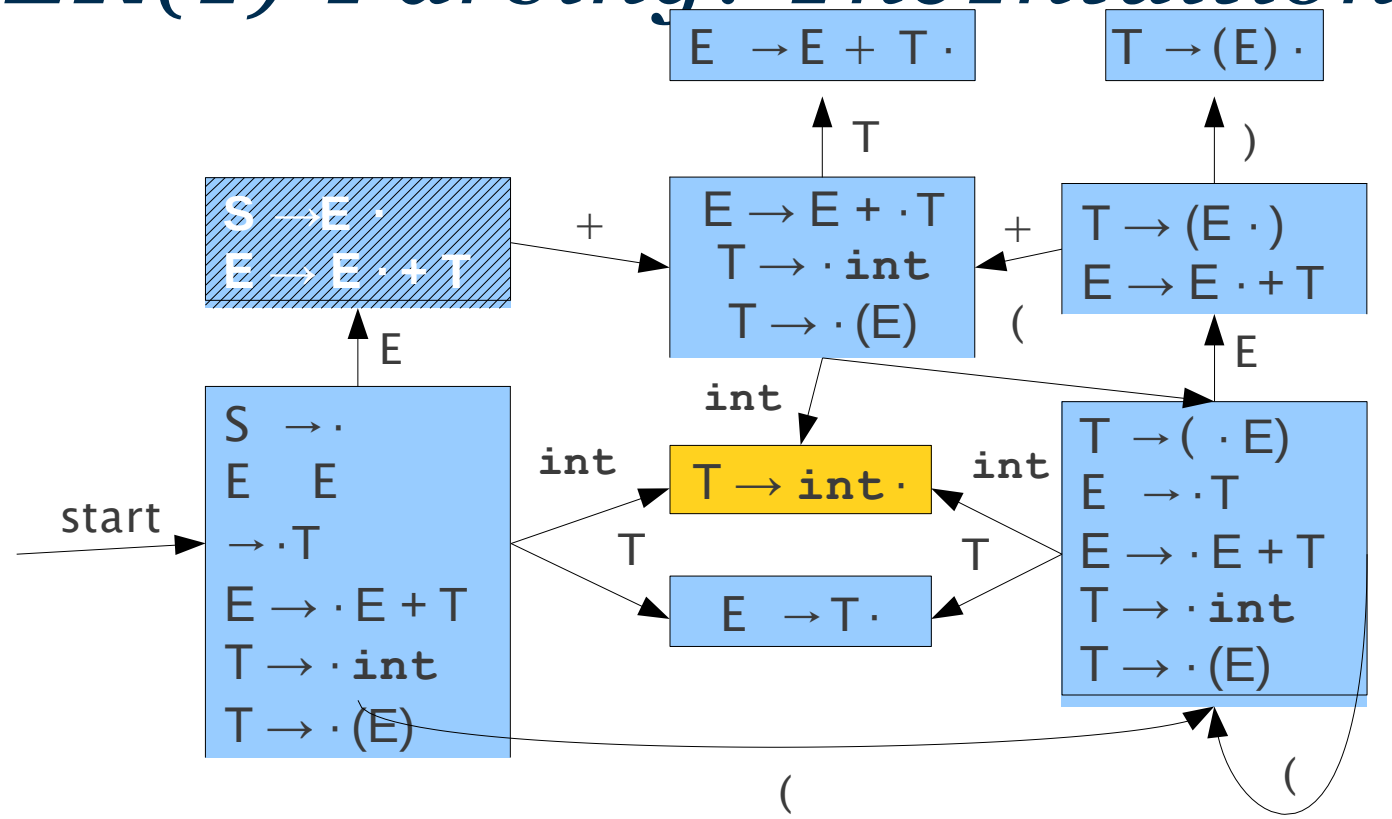
S → E
E → T
E → E + T
T → int
T → (E)



E	+	(	int	+	int	+	int	)	$

LR(1) Parsing: The Intuition

S → E
E → T
E → E + T
T → int
T → (E)



S → · E	$
E → E + · T	$
T → (· E)	$
E → · E + T	)
E → · T	+
T → int ·	+

E	+	(	int	+	int	+	int	)	$

The Intuition behind LR(1)

- Guess which series of productions we are reversing.
- Use this information to maintain information about what lookahead to expect.
- When deciding whether to shift or reduce, use lookahead to disambiguate.

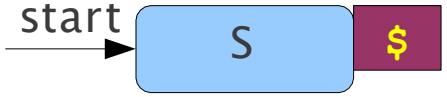
Tracking Lookaheads

- How do we know what lookahead to expect at each state?
- Observation:
  - There are only finitely many productions we can be in at any point.
  - There are only finitely many positions we can be in each production.
  - **There are only finitely many lookahead sets** at each point.
- Construct an automaton to track lookaheads!

Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)

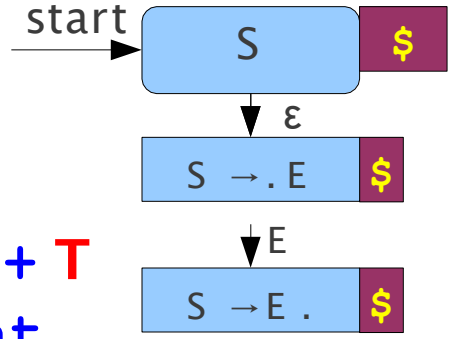
Constructing LR(1) Automata



S → E
E → T
E → E + T
T → int
T → (E)

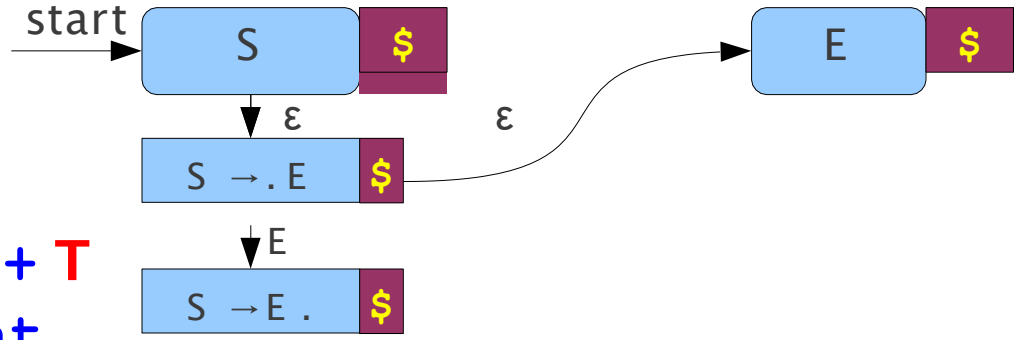
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)



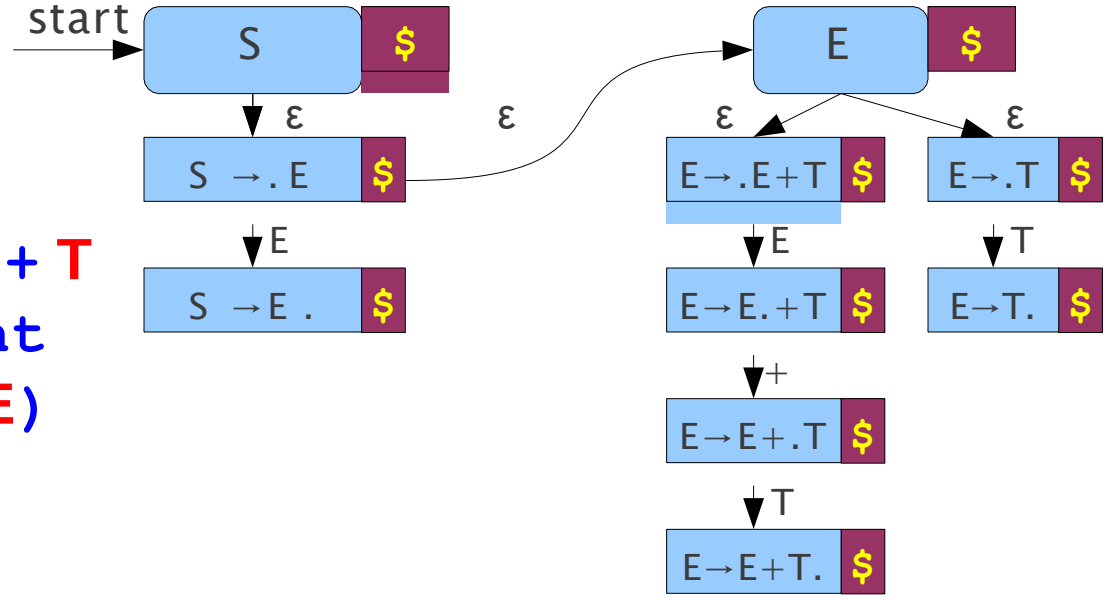
Constructing LR(1) Automata

S → E
E → T
E → E + T
T → int
T → (E)

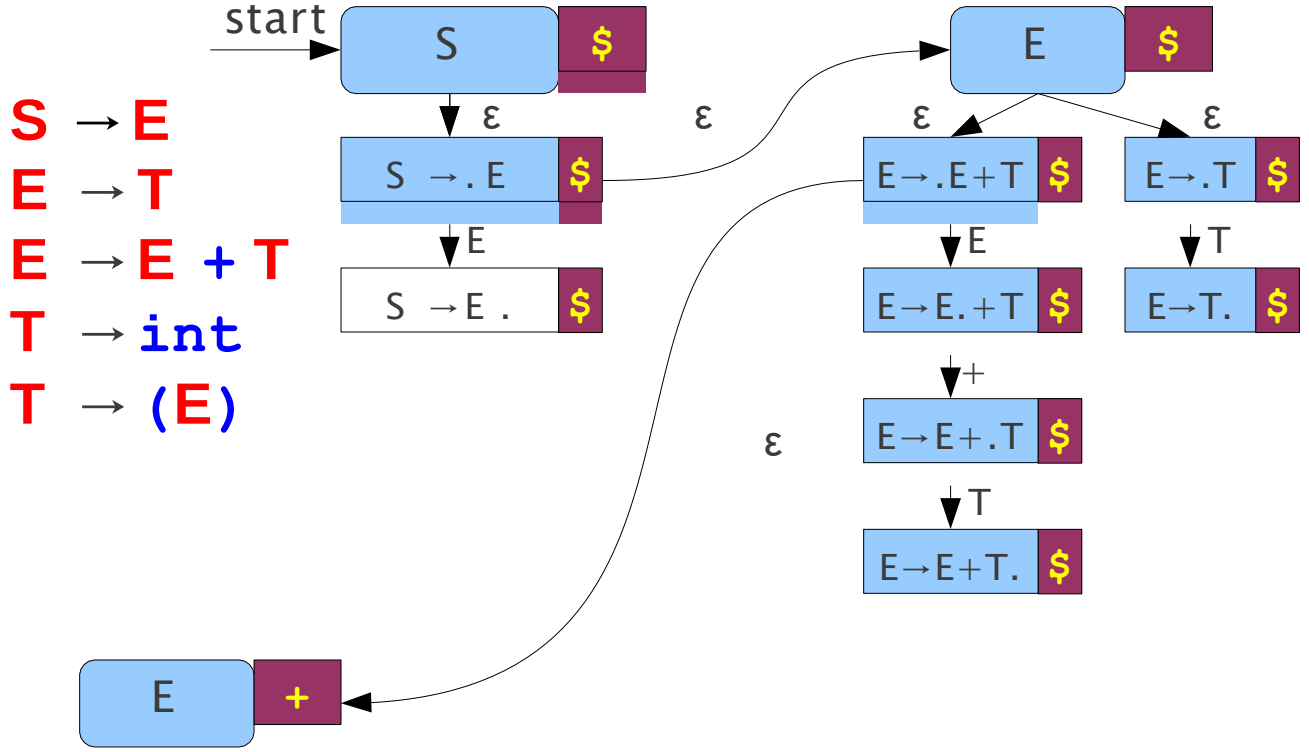


Constructing LR(1) Automata

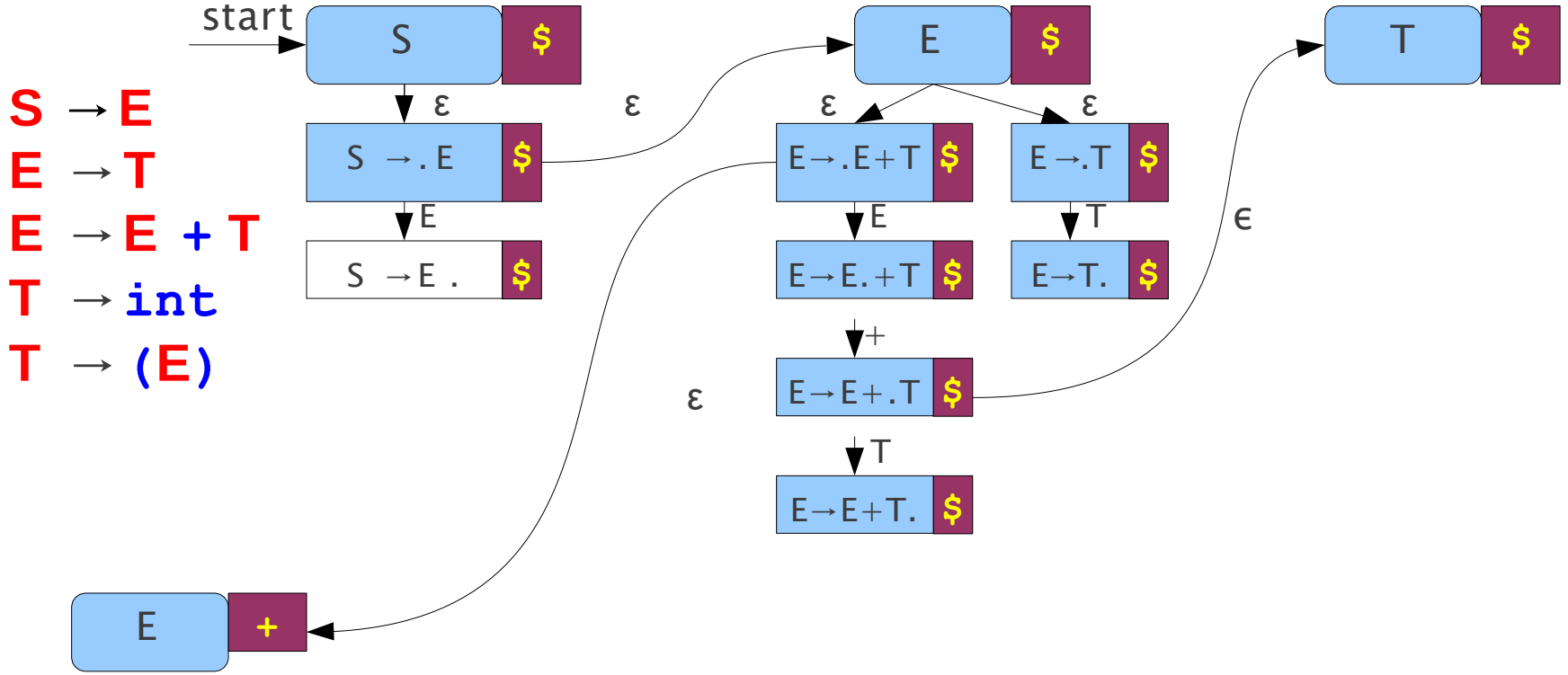
S → E
E → T
E → E + T
T → int
T → (E)



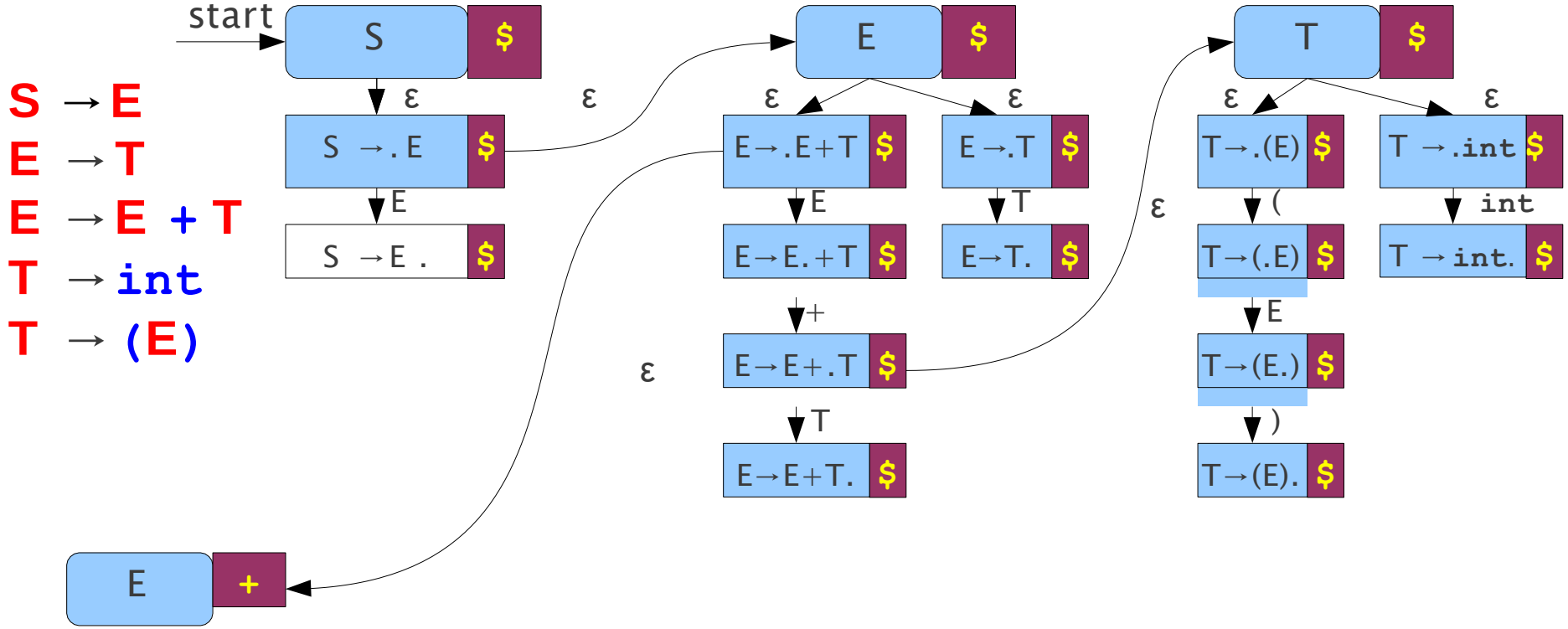
Constructing LR(1) Automata



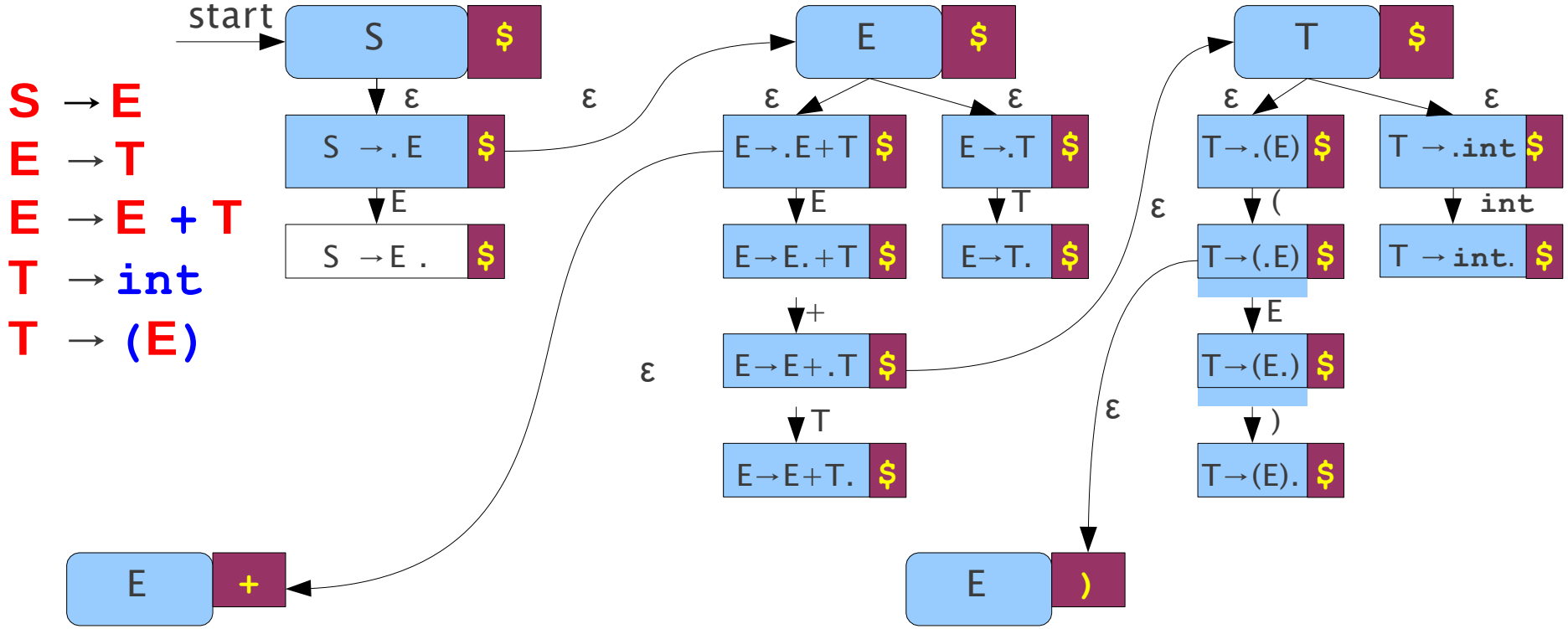
Constructing LR(1) Automata



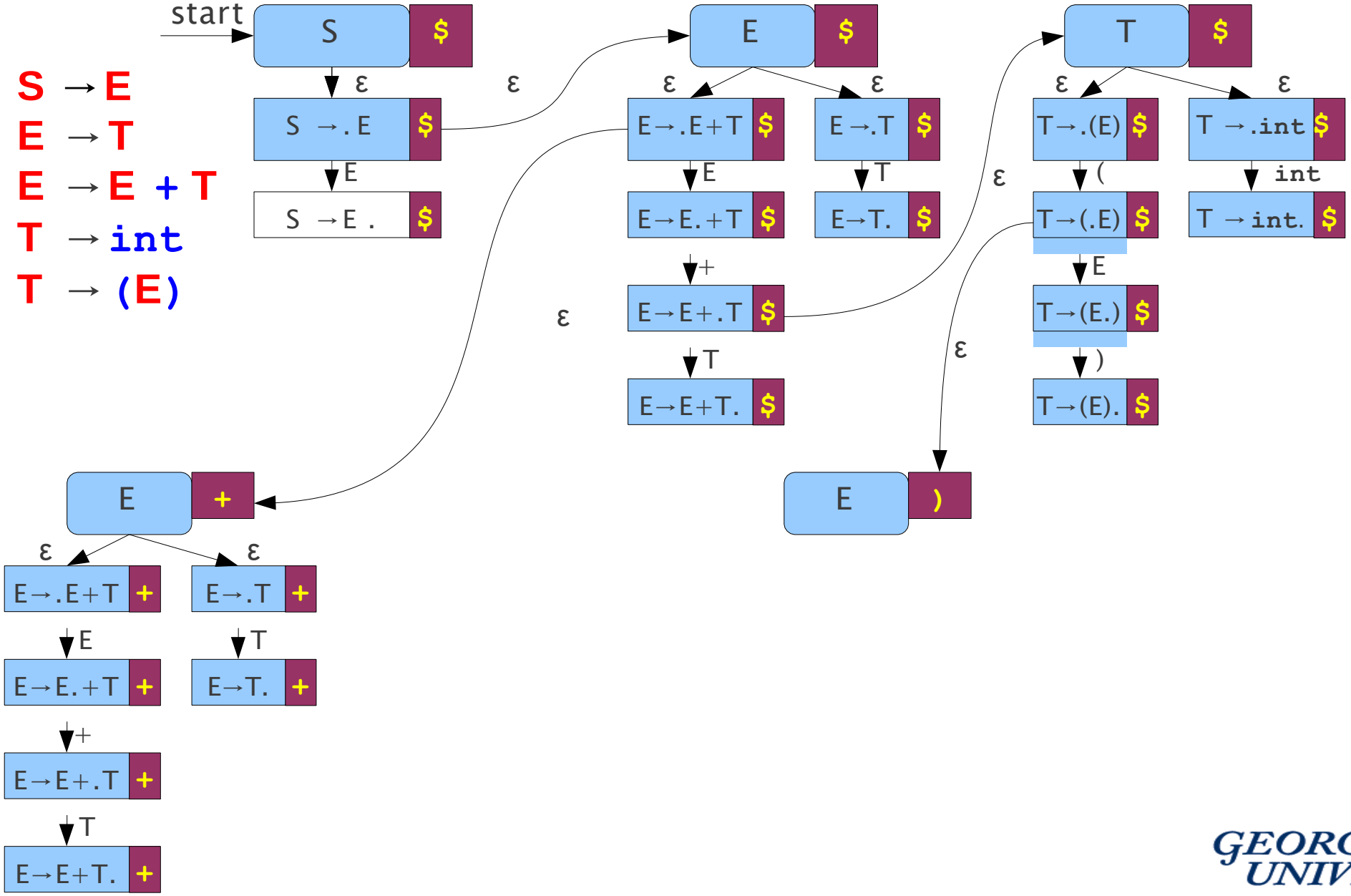
Constructing LR(1) Automata



Constructing LR(1) Automata

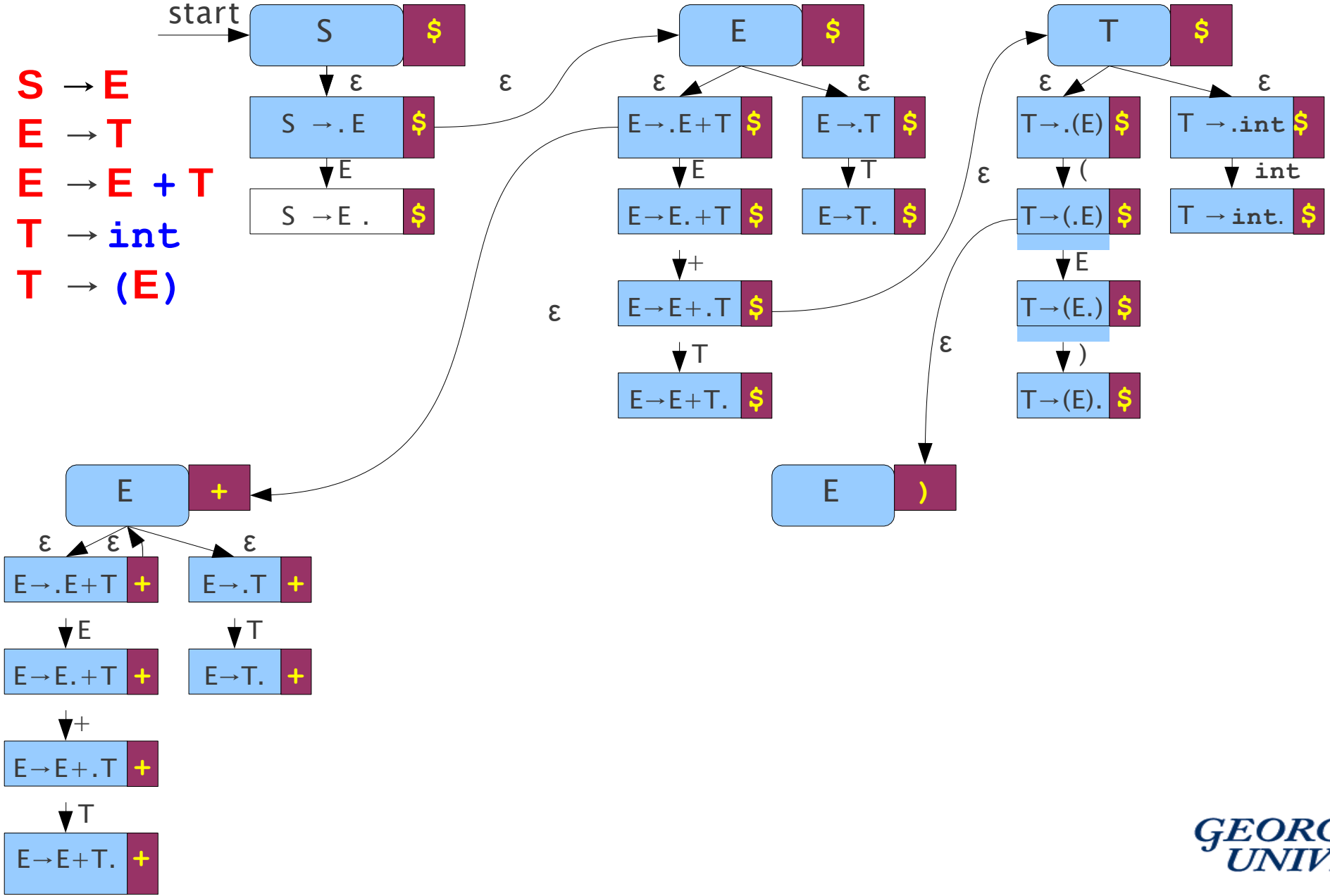


Constructing LR(1) Automata



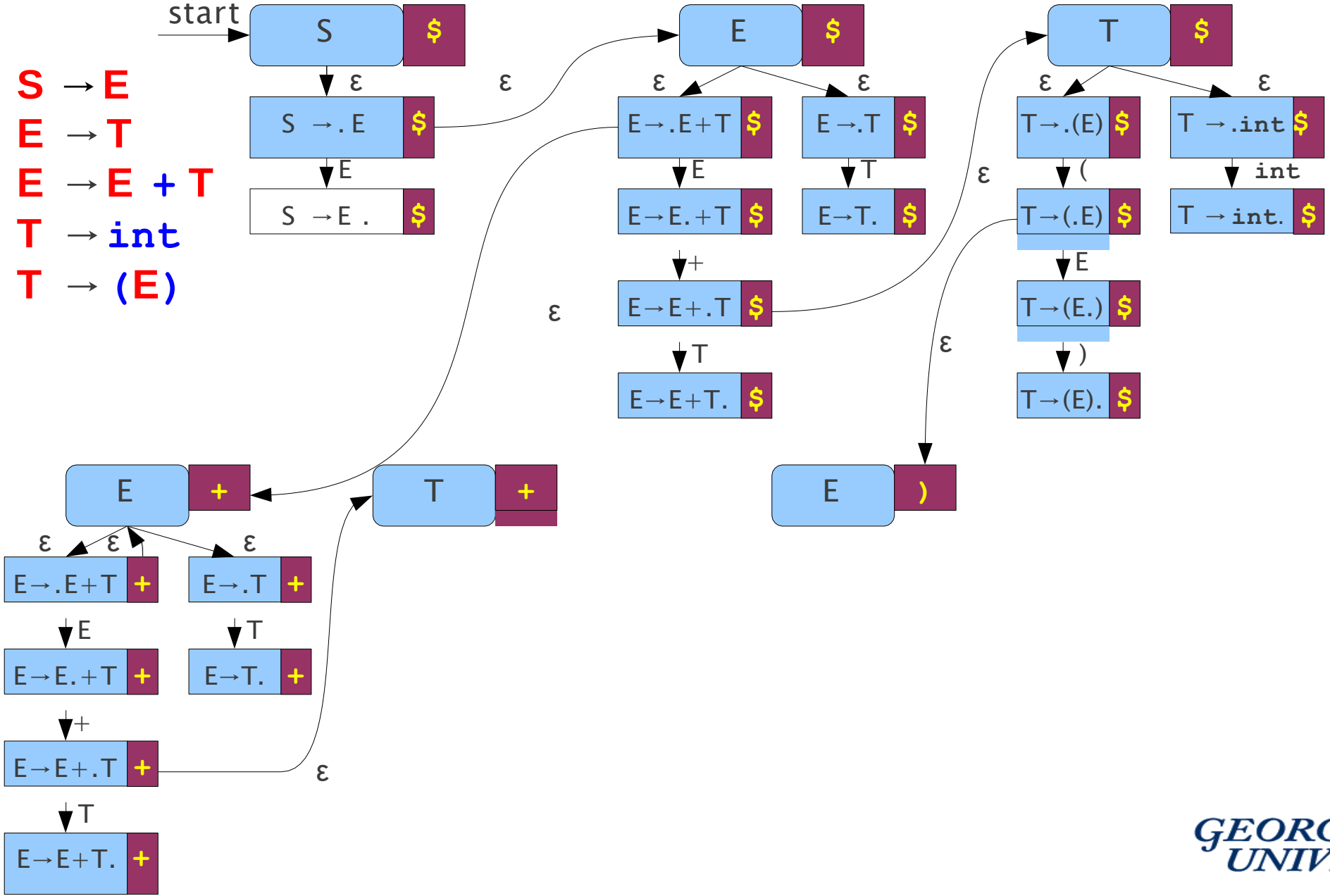
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



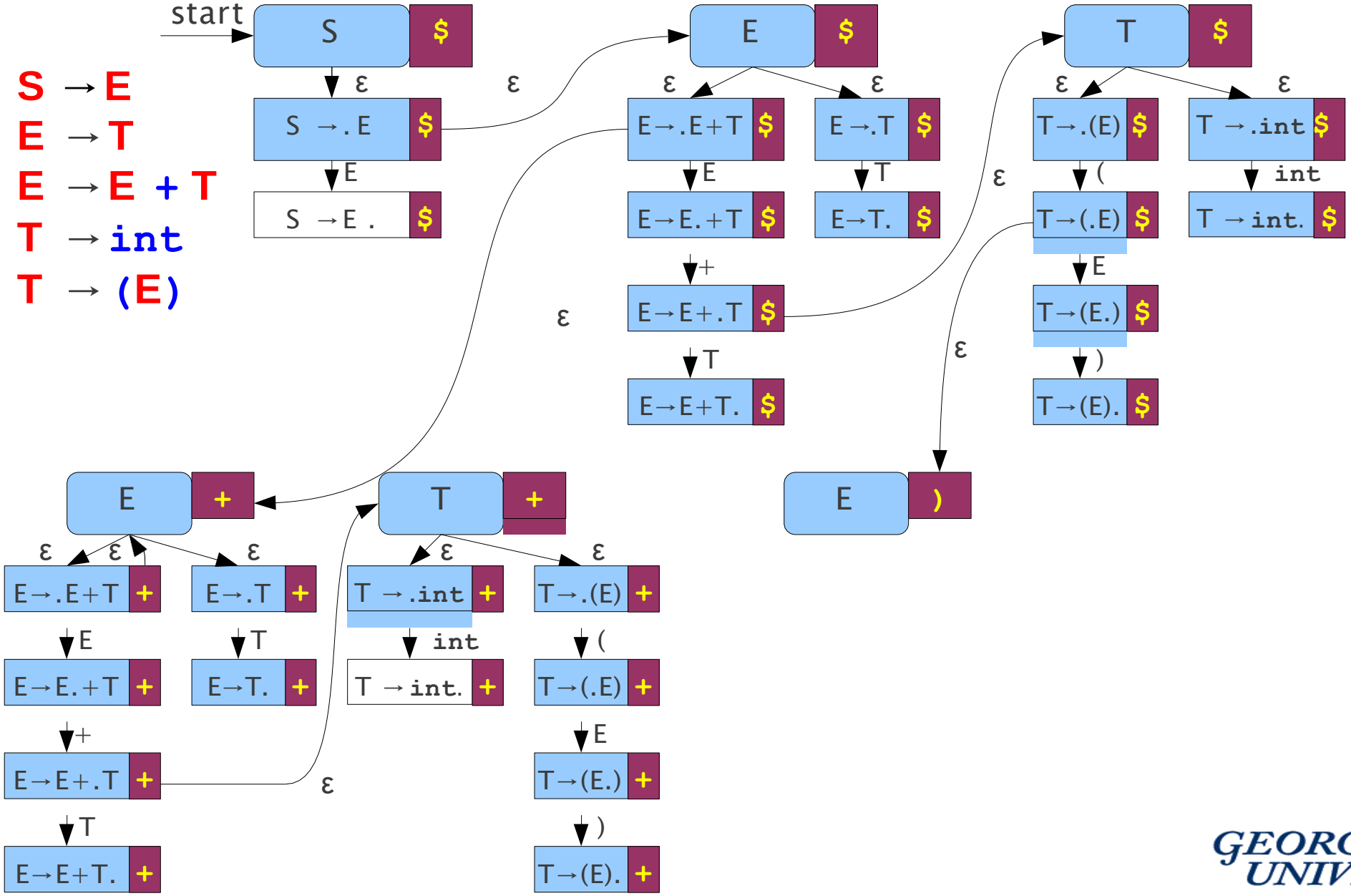
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



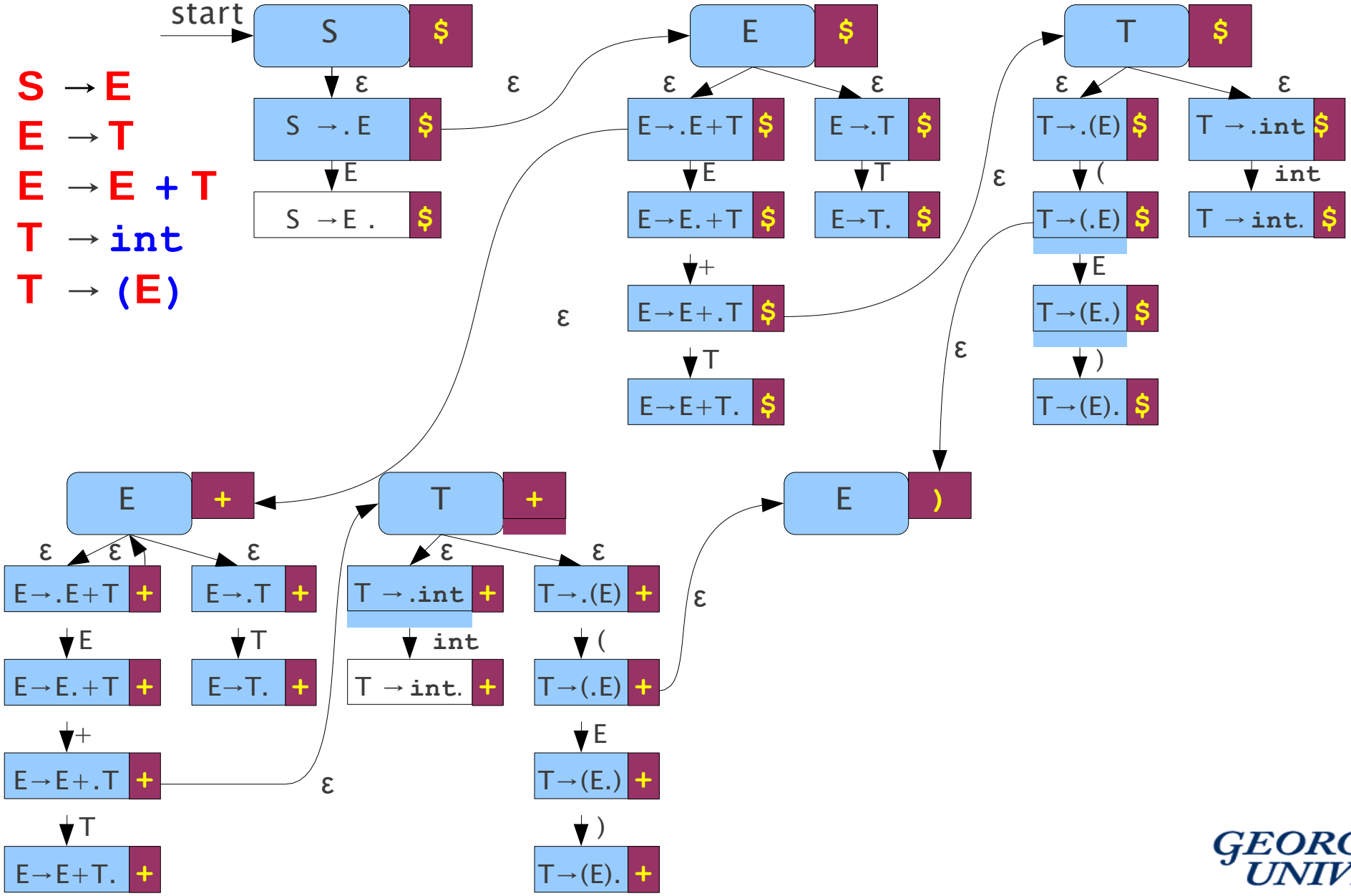
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



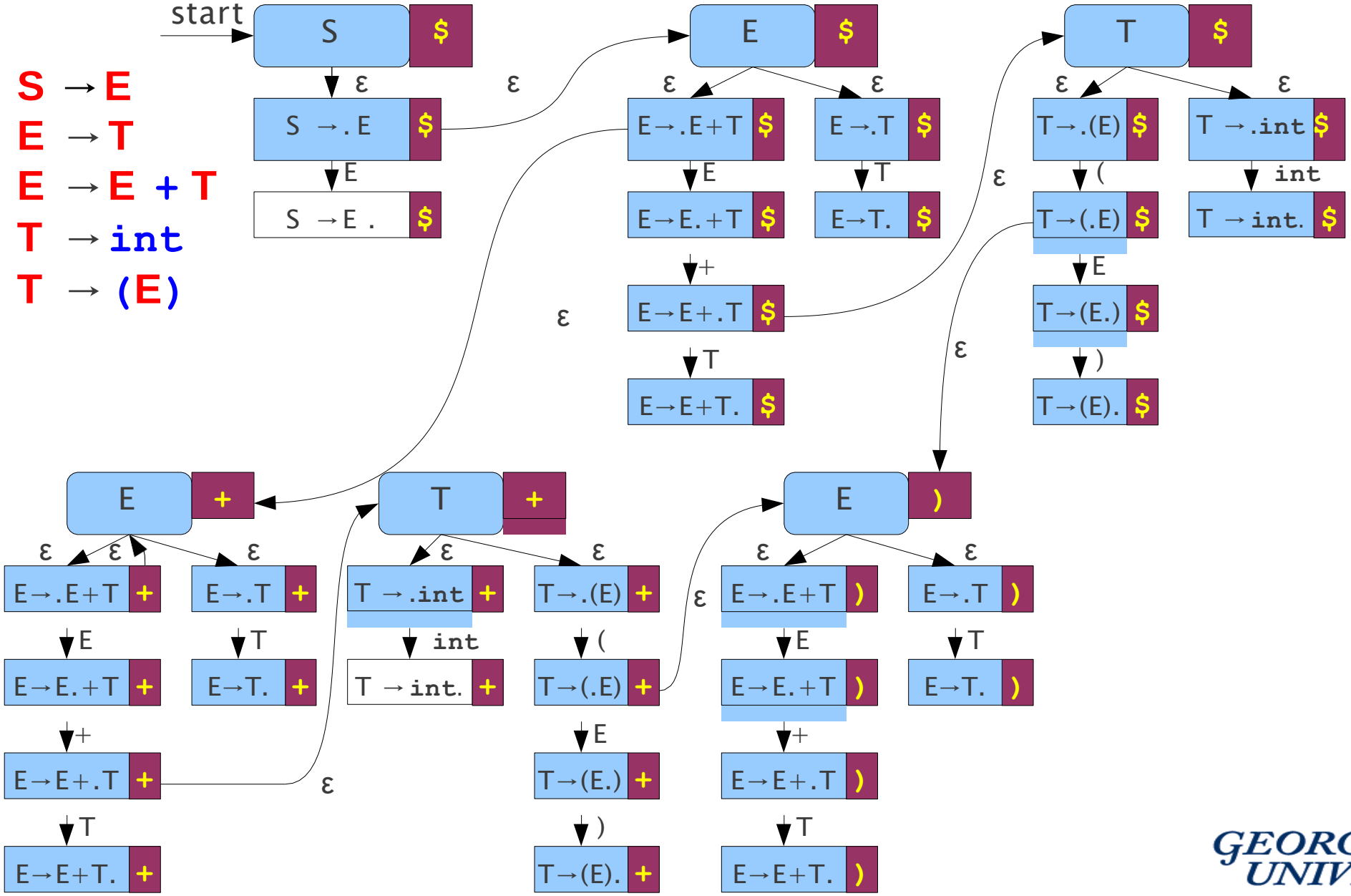
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



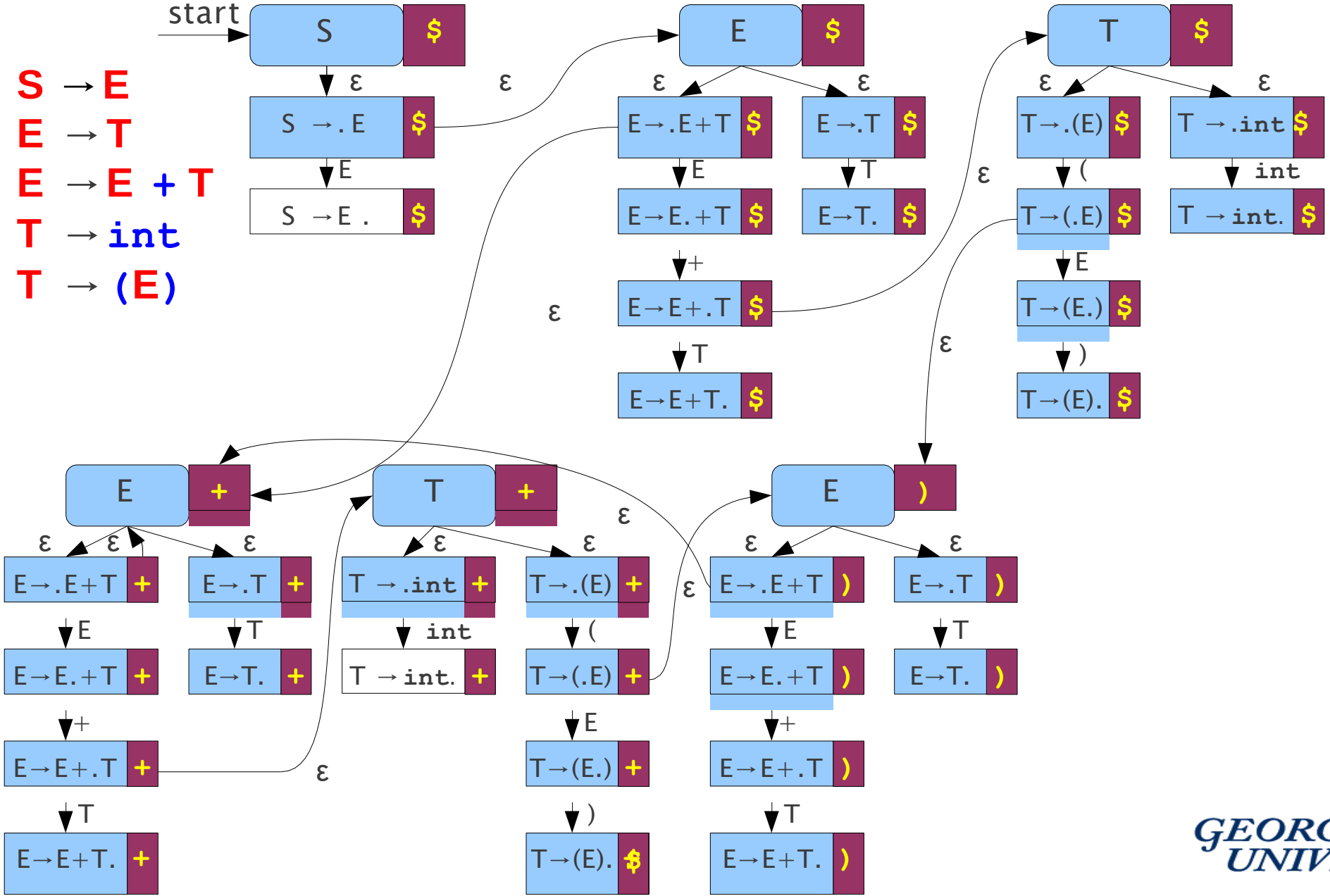
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



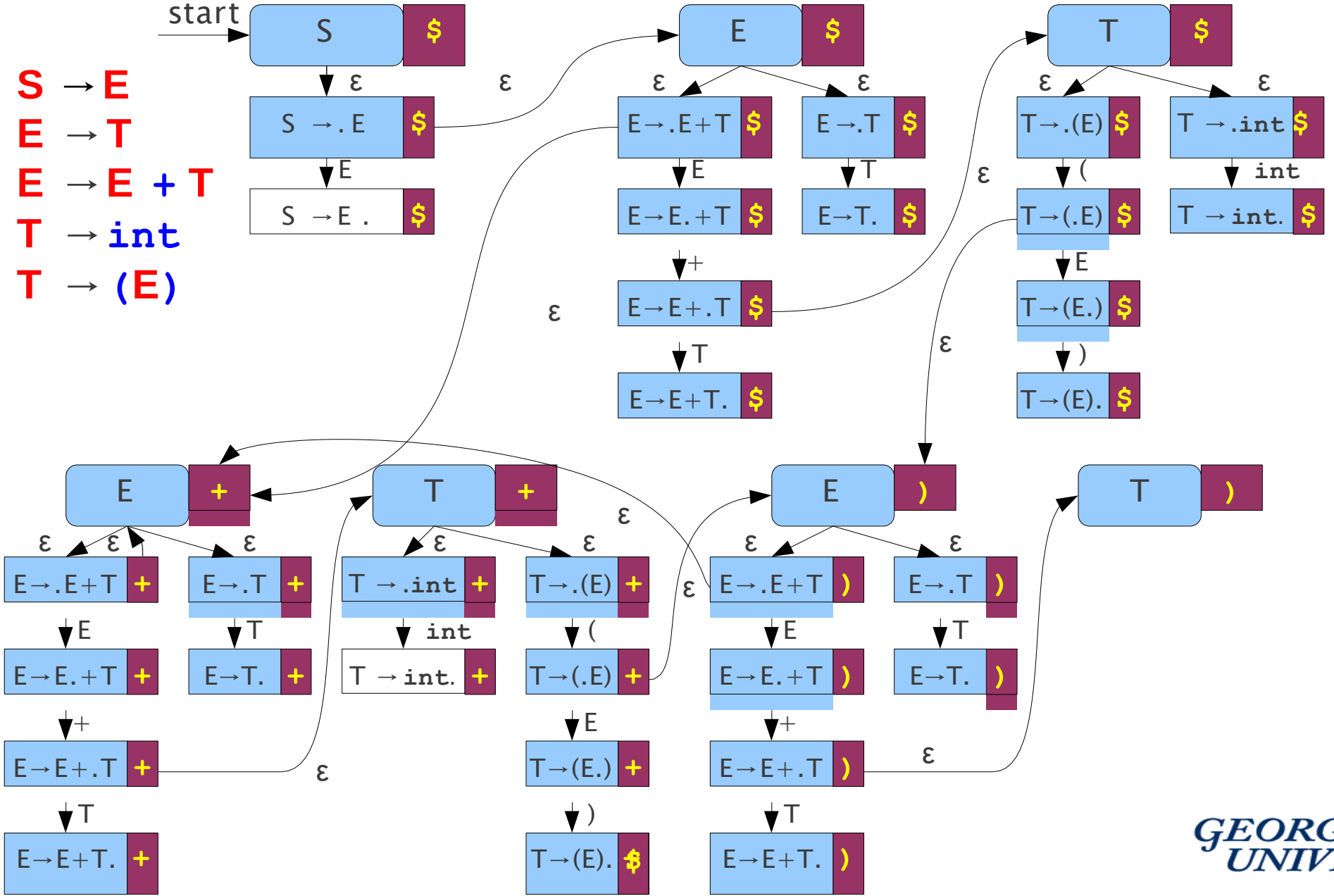
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



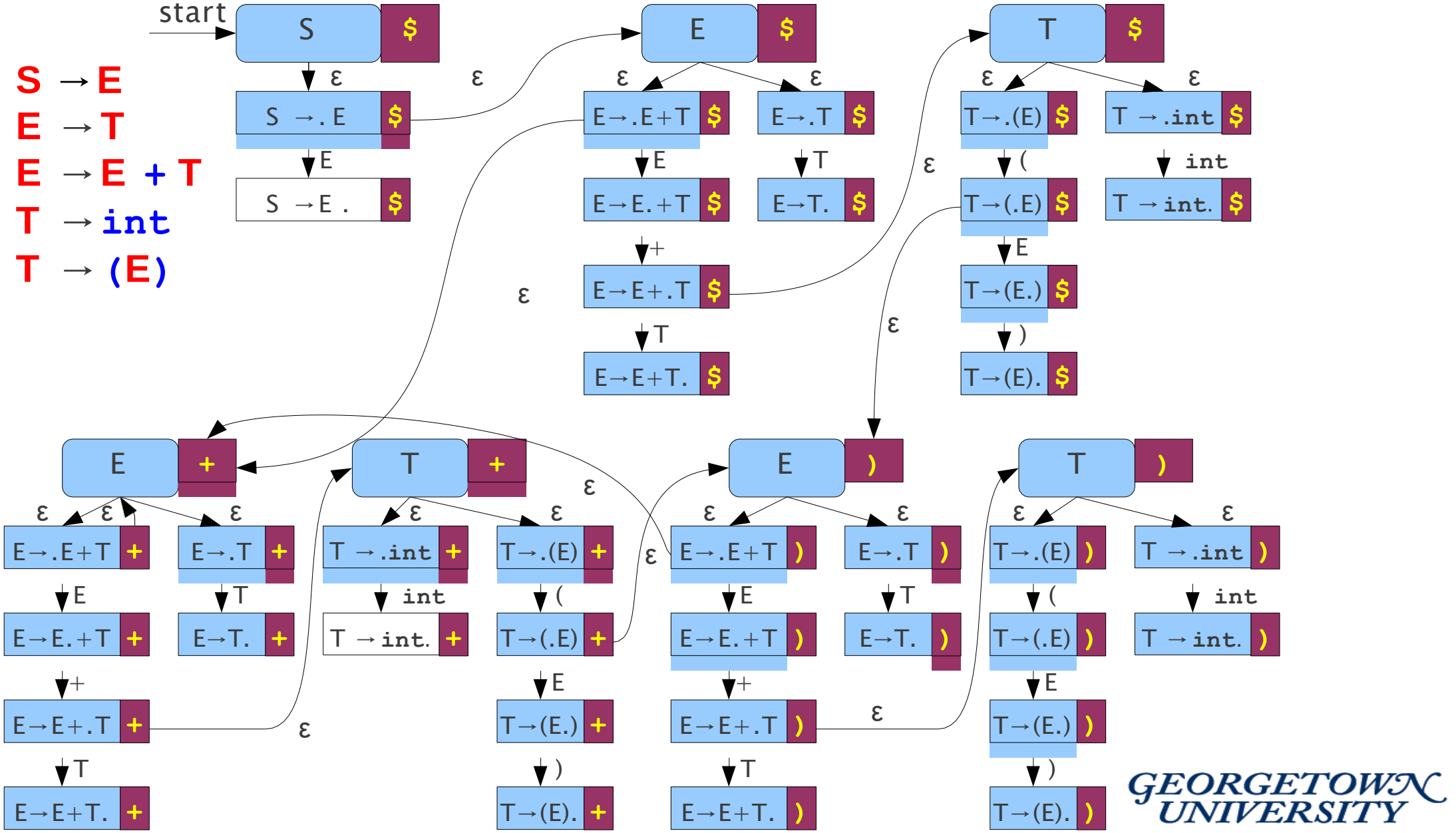
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



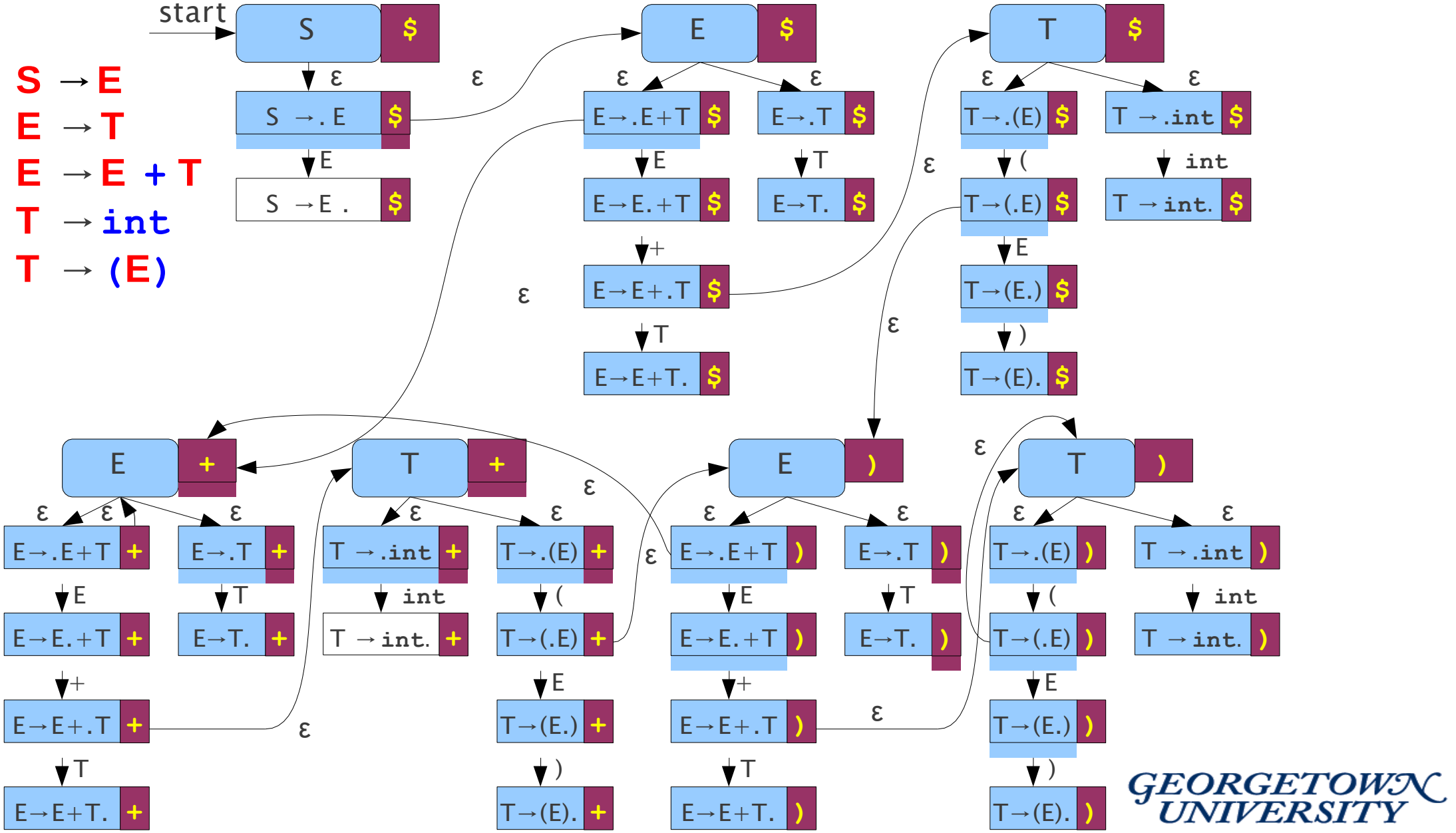
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



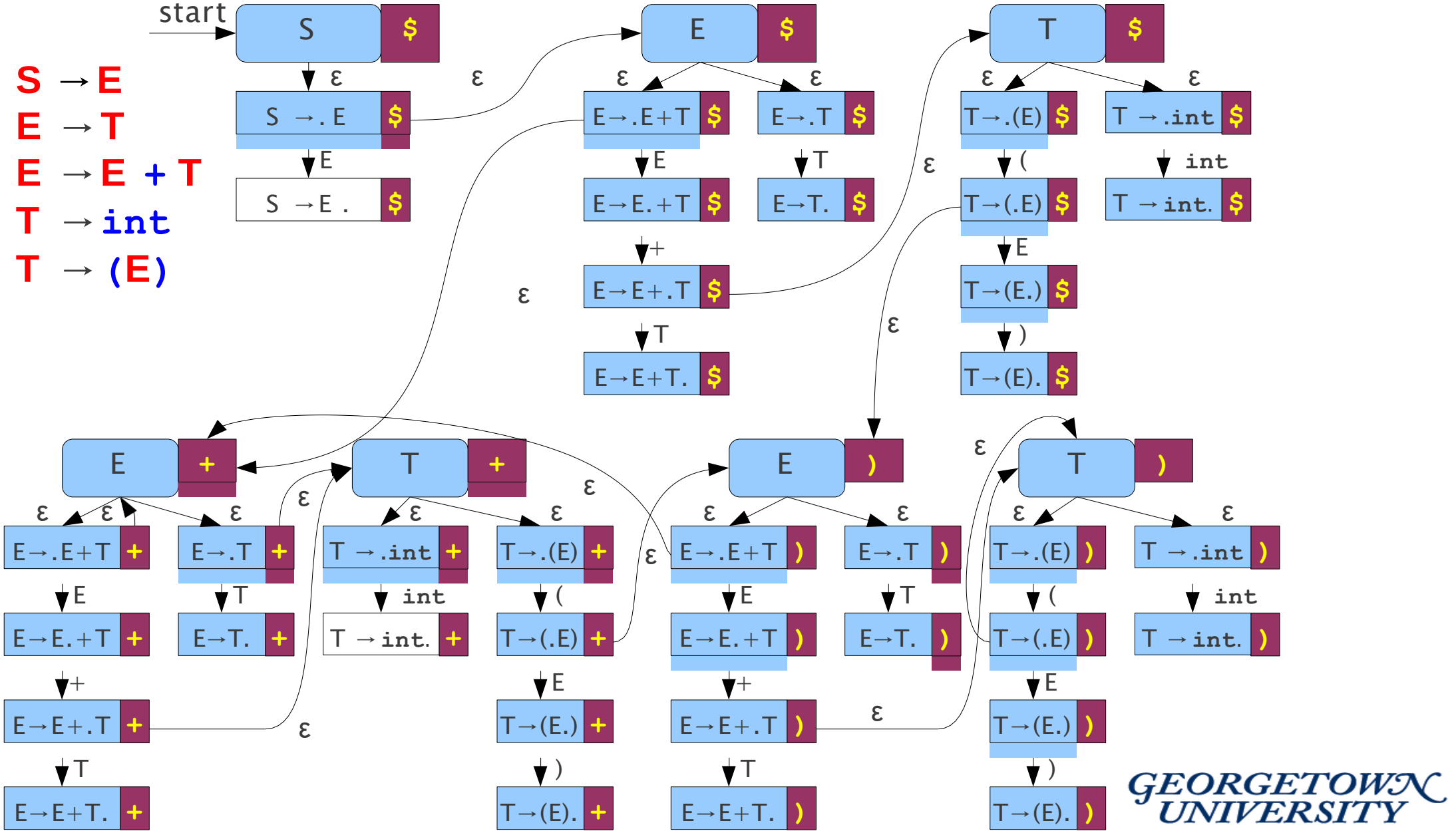
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



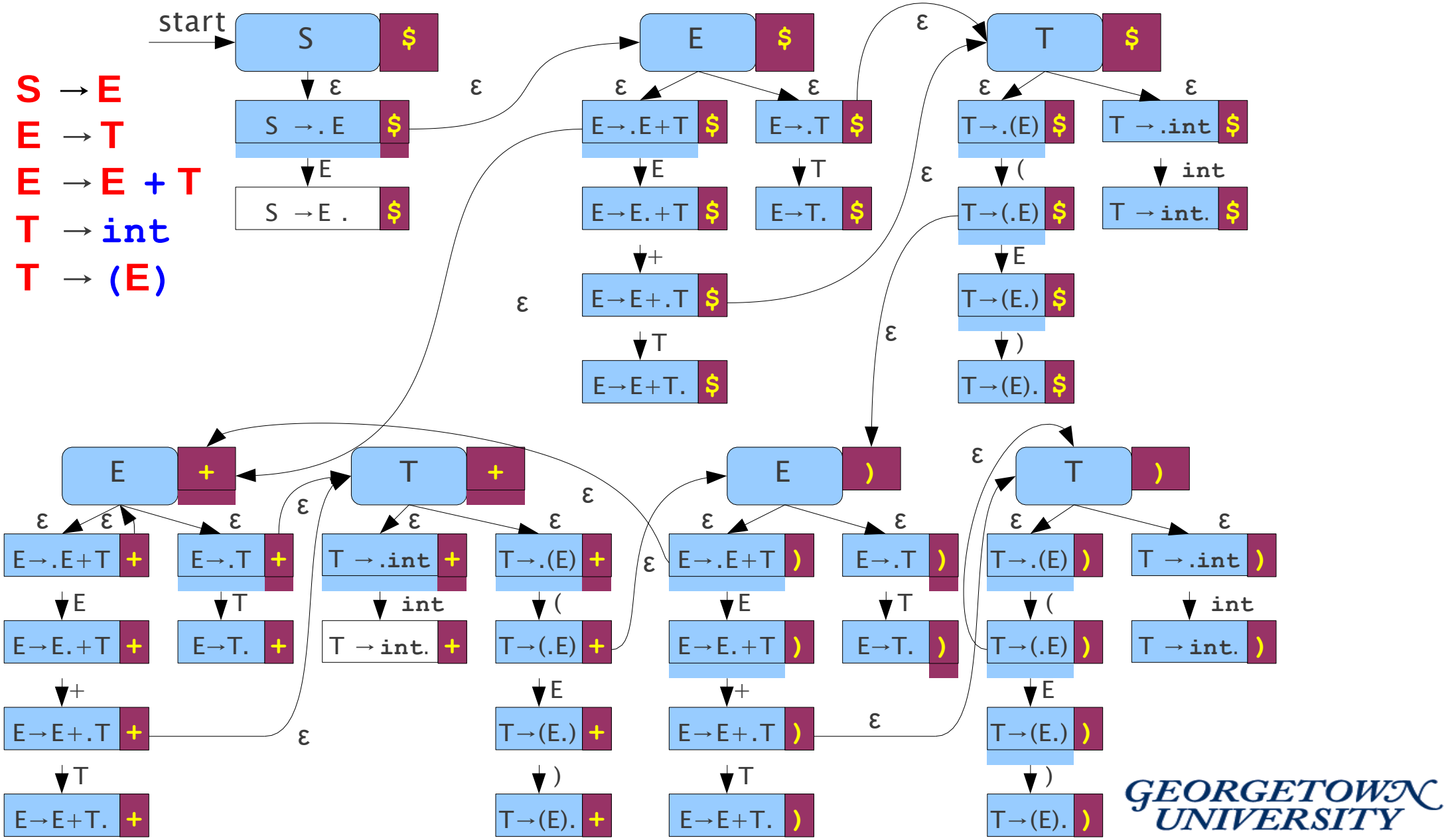
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



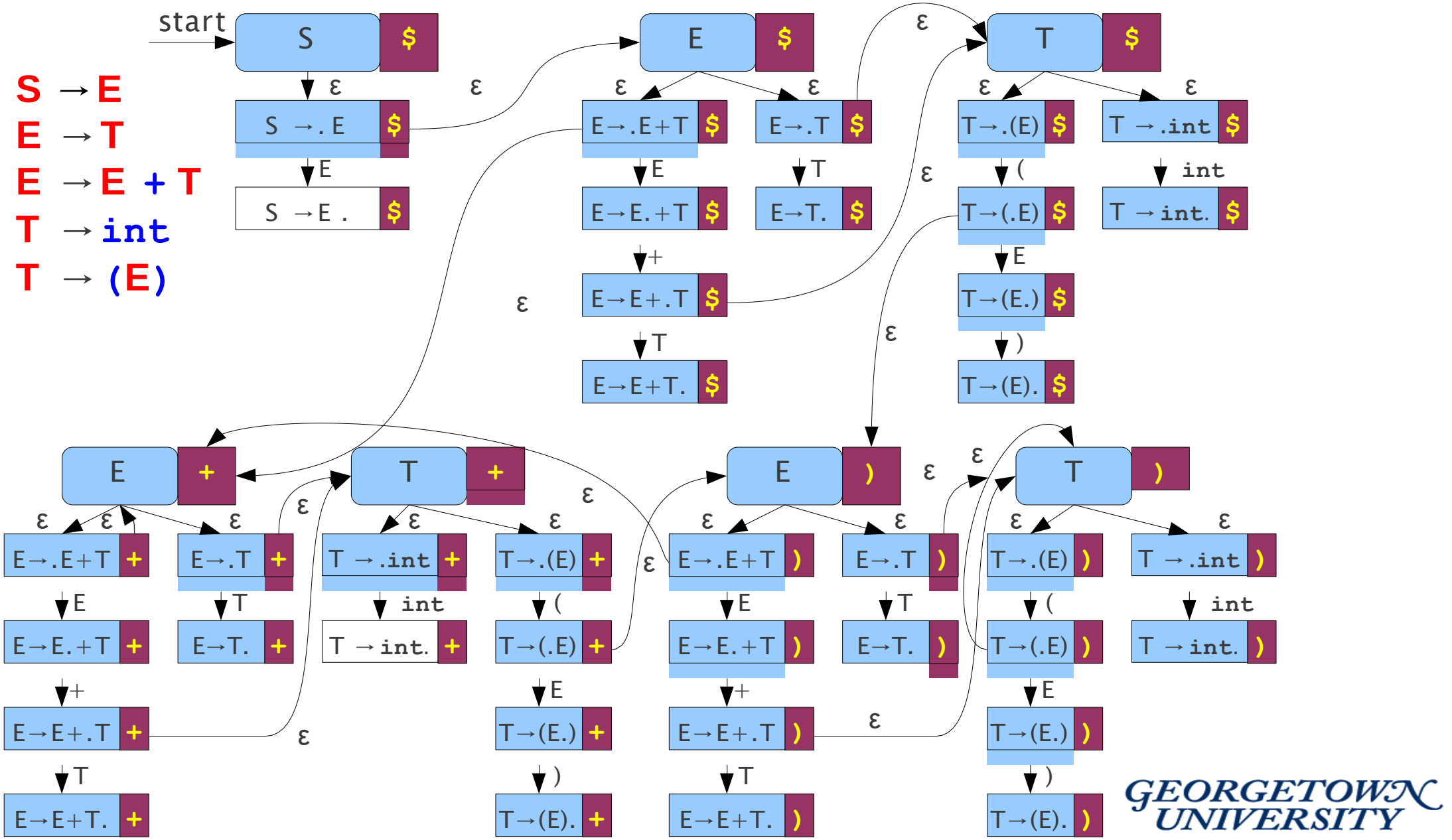
GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



GEORGETOWN UNIVERSITY

Constructing LR(1) Automata



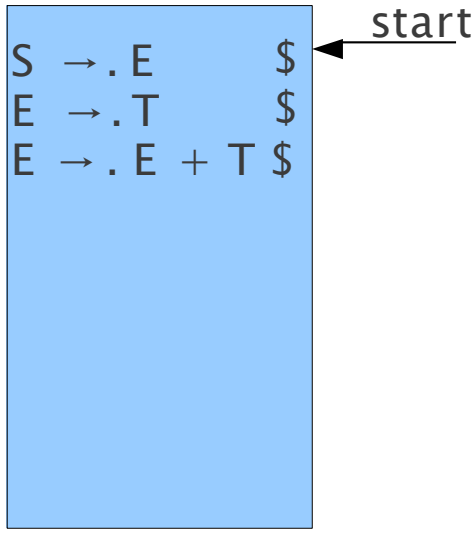
Constructing LR(1) Automata

- Begin with a state **S** [**$**].
- For each state **A** [**t**], for each production **A** → γ:
  - Construct states **A** → $\alpha \cdot \omega$ [**t**] for all possible ways of splitting $\gamma = \alpha\omega$.
  - Add an ϵ-transition from **A** [**t**] to each of these states.
  - Add transitions on x between **A** → $\alpha \cdot x\omega$ [**t**] and **A** → $\alpha x \cdot \omega$ [**t**]
- For each state **A** → $\alpha \cdot$ **B** ω [**t**], add an ϵ-transition from **A** → $\alpha \cdot$ **B** ω [**t**] to **B** [**r**] for each terminal **r** $\in$ FIRST*(ω**t**).

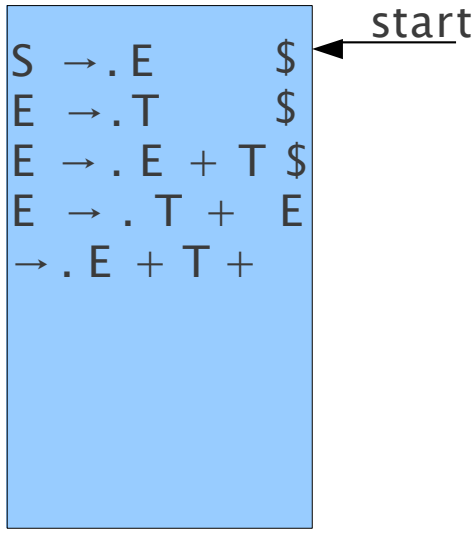
Deterministic LR(1) Automata

Deterministic LR(1) Automata

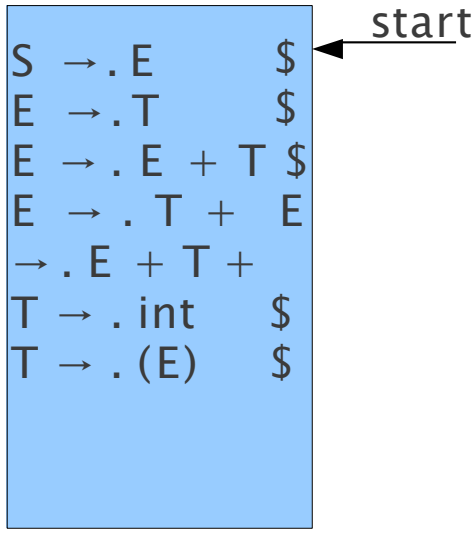
Deterministic LR(1) Automata



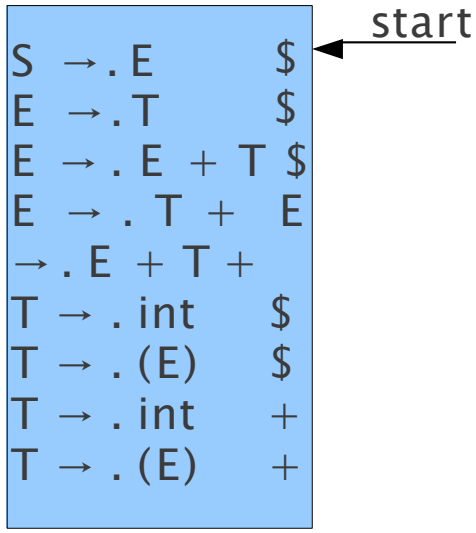
Deterministic LR(1) Automata



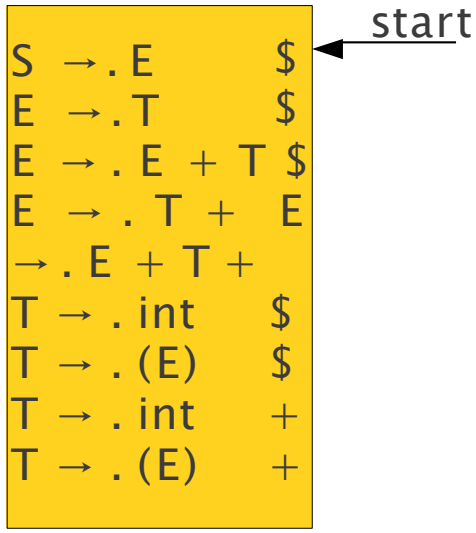
Deterministic LR(1) Automata



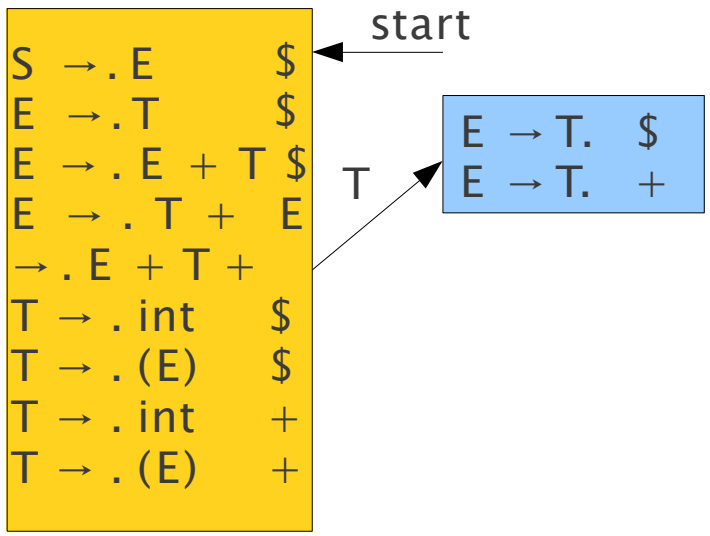
Deterministic LR(1) Automata



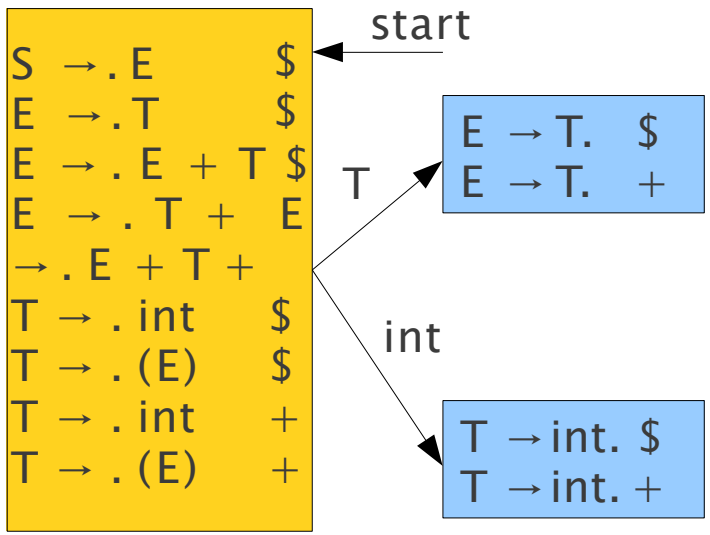
Deterministic LR(1) Automata



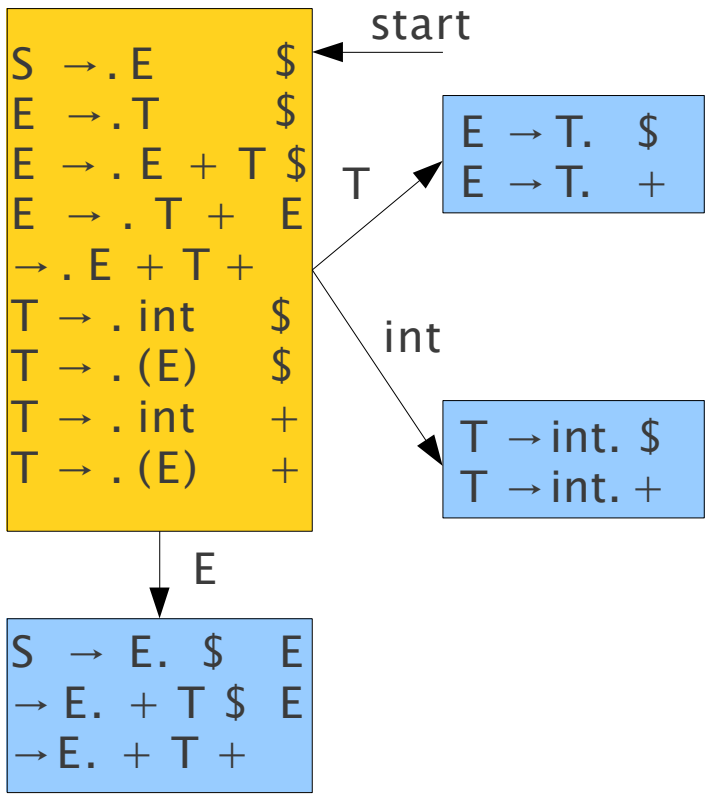
Deterministic LR(1) Automata



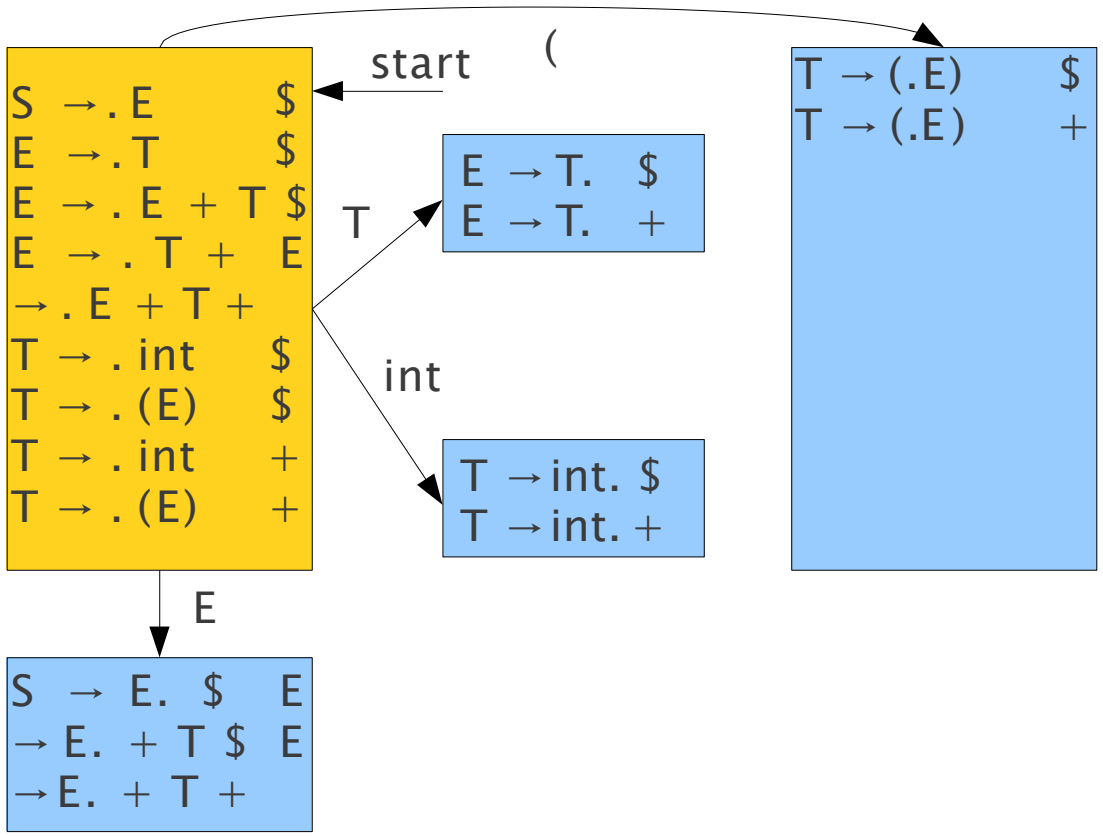
Deterministic LR(1) Automata



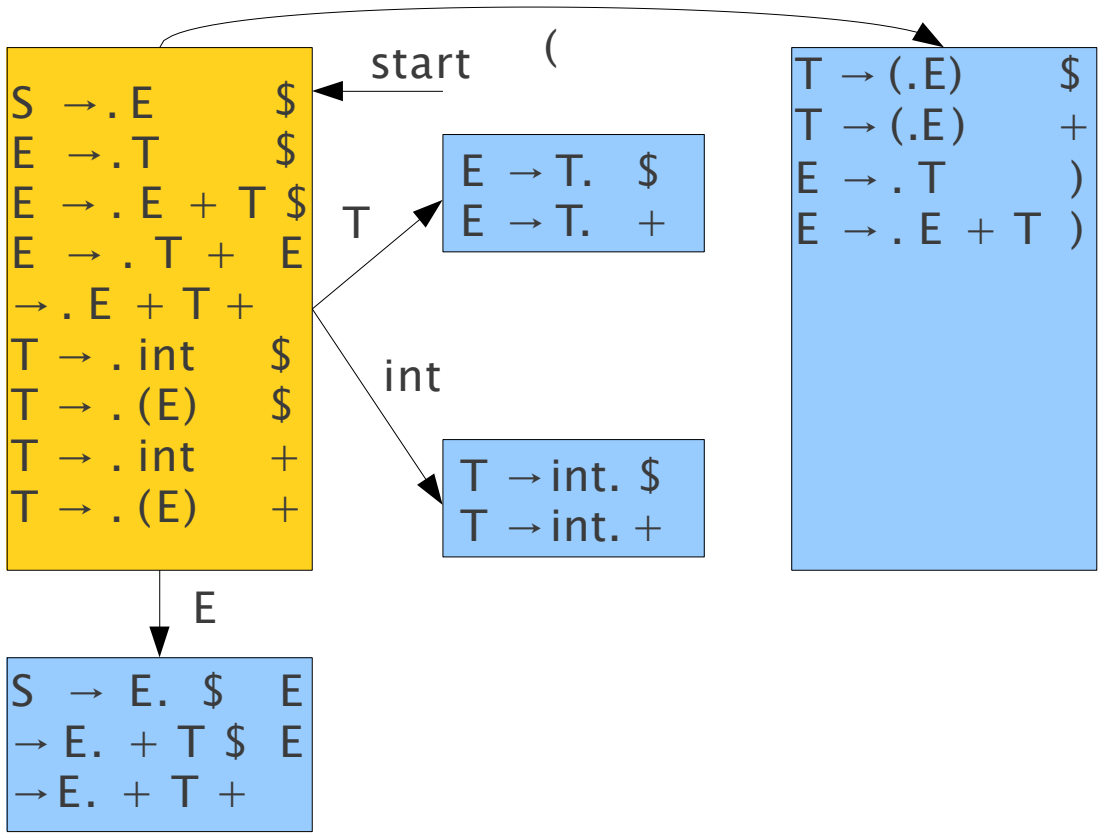
Deterministic LR(1) Automata



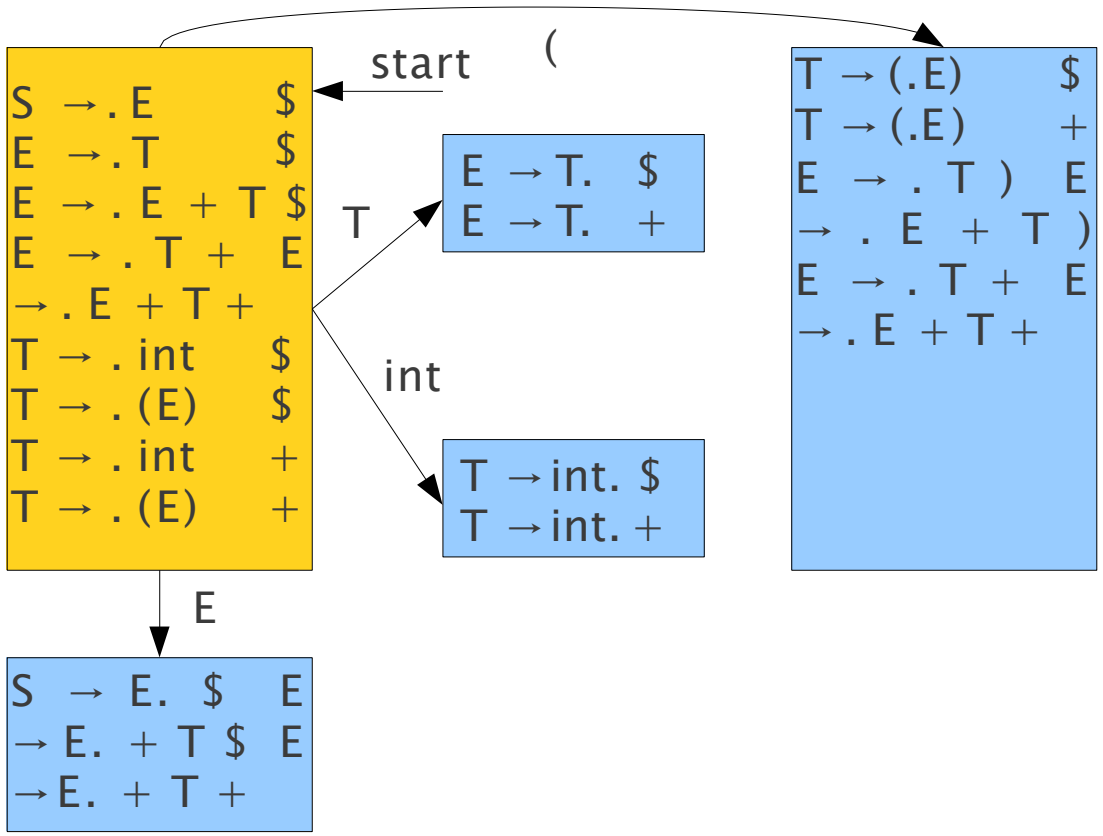
Deterministic LR(1) Automata



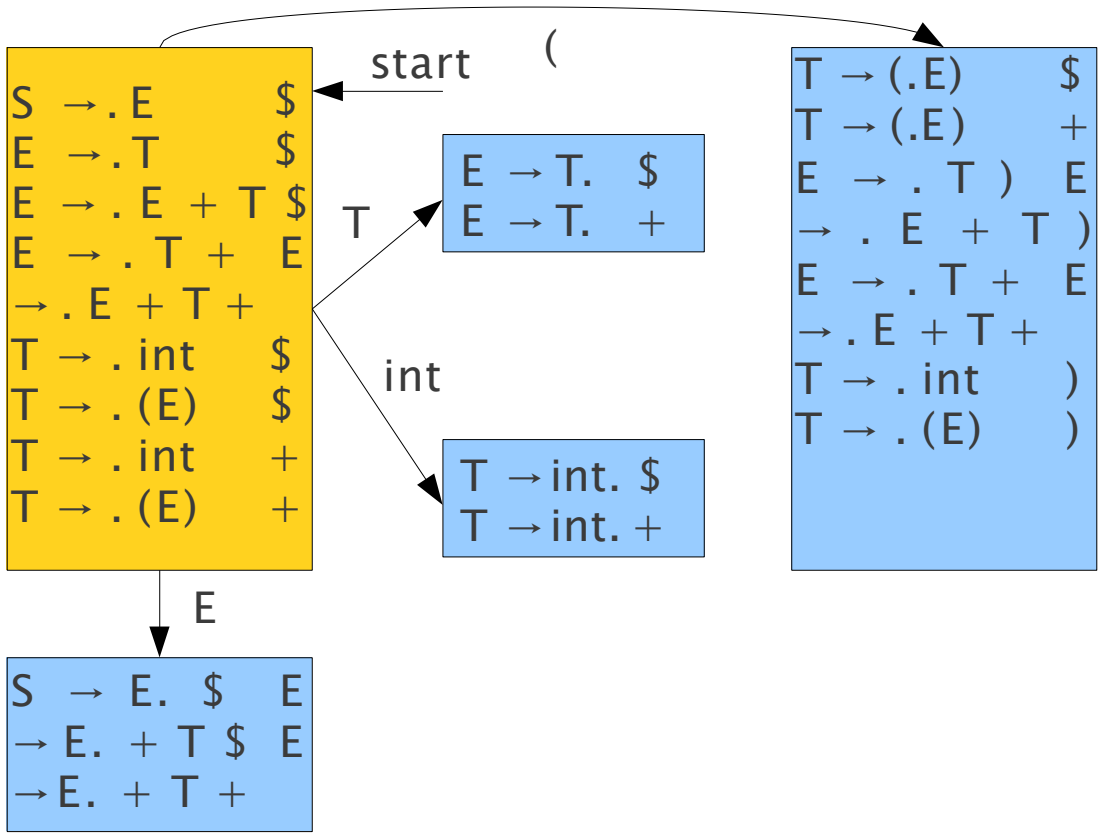
Deterministic LR(1) Automata



Deterministic LR(1) Automata

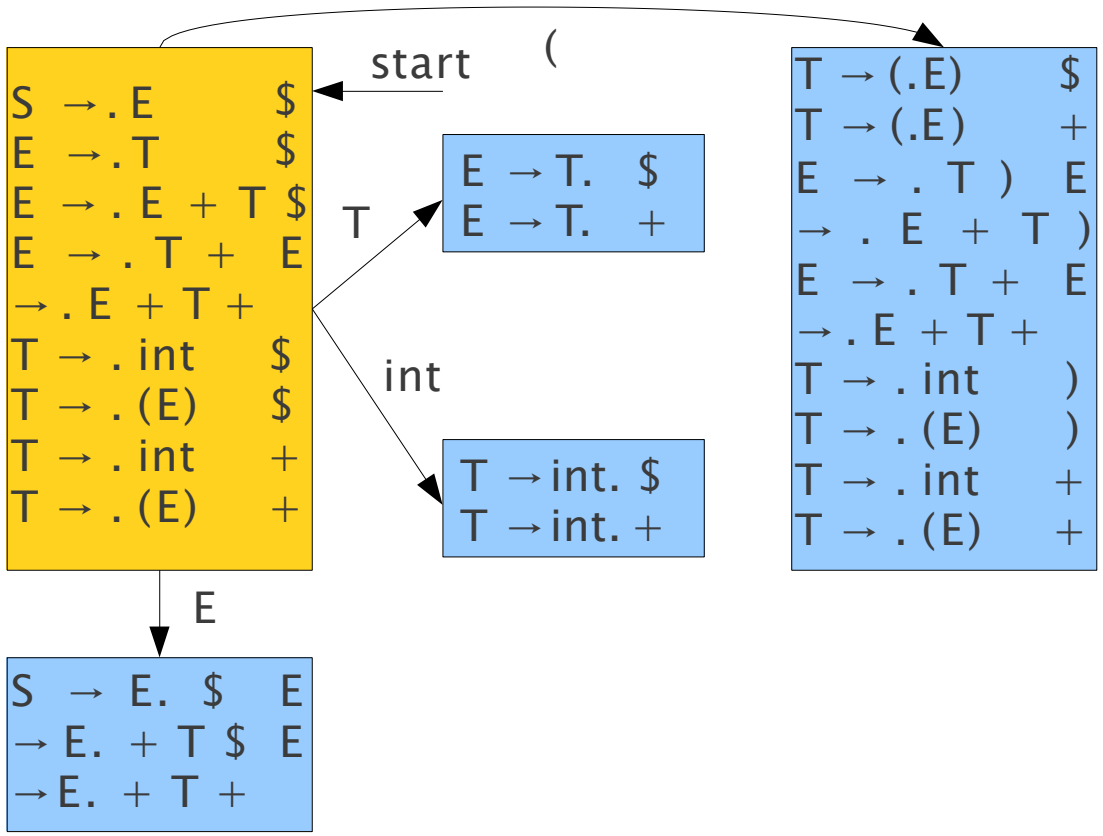


Deterministic LR(1) Automata



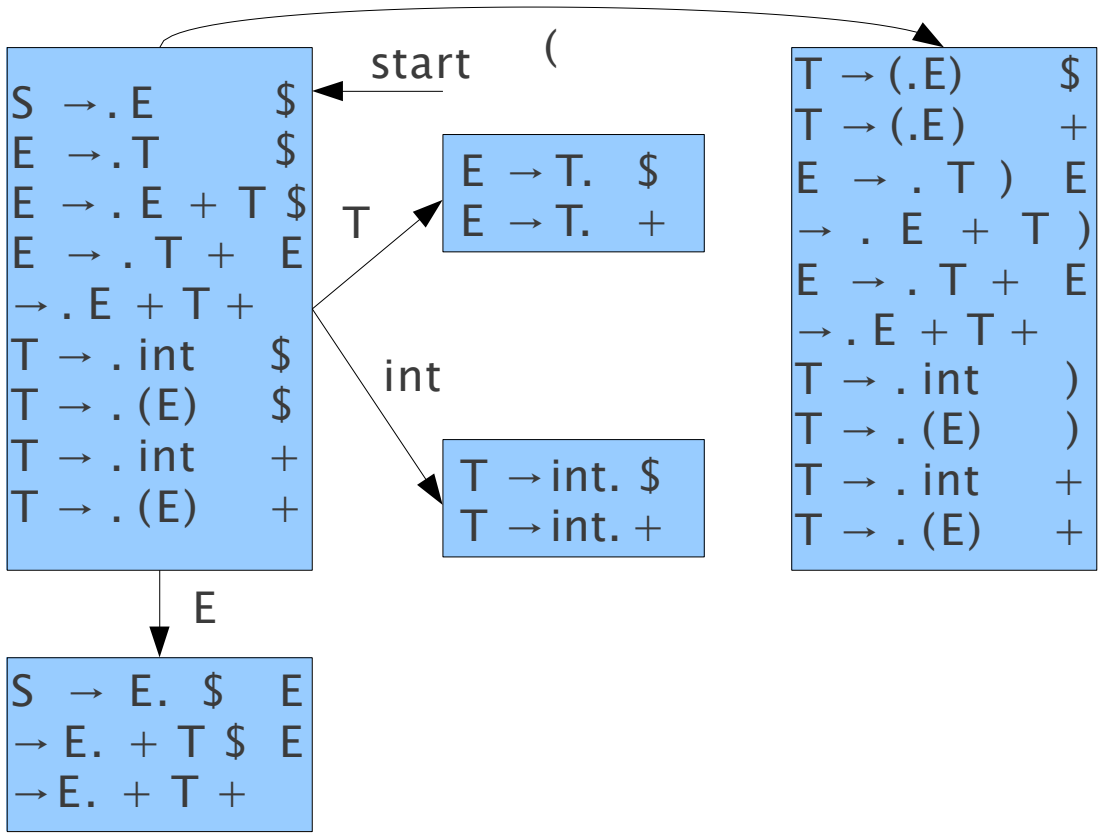
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



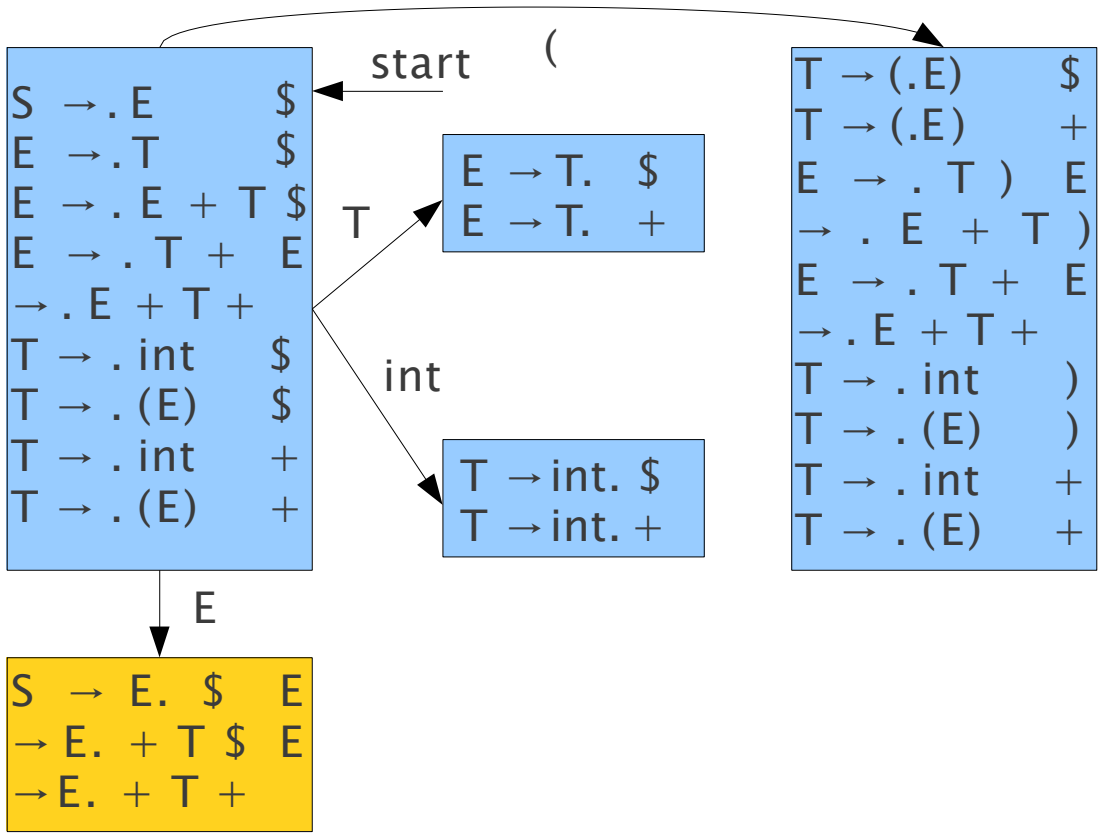
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

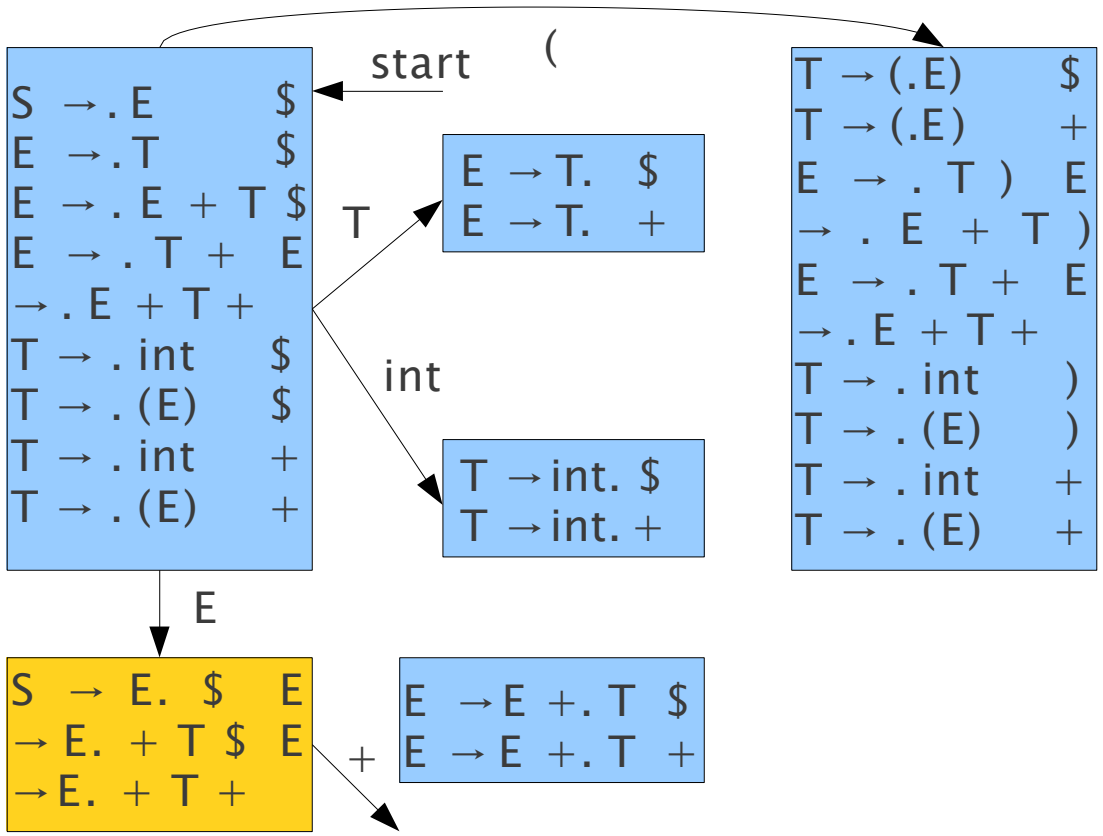


GEORGETOWN UNIVERSITY

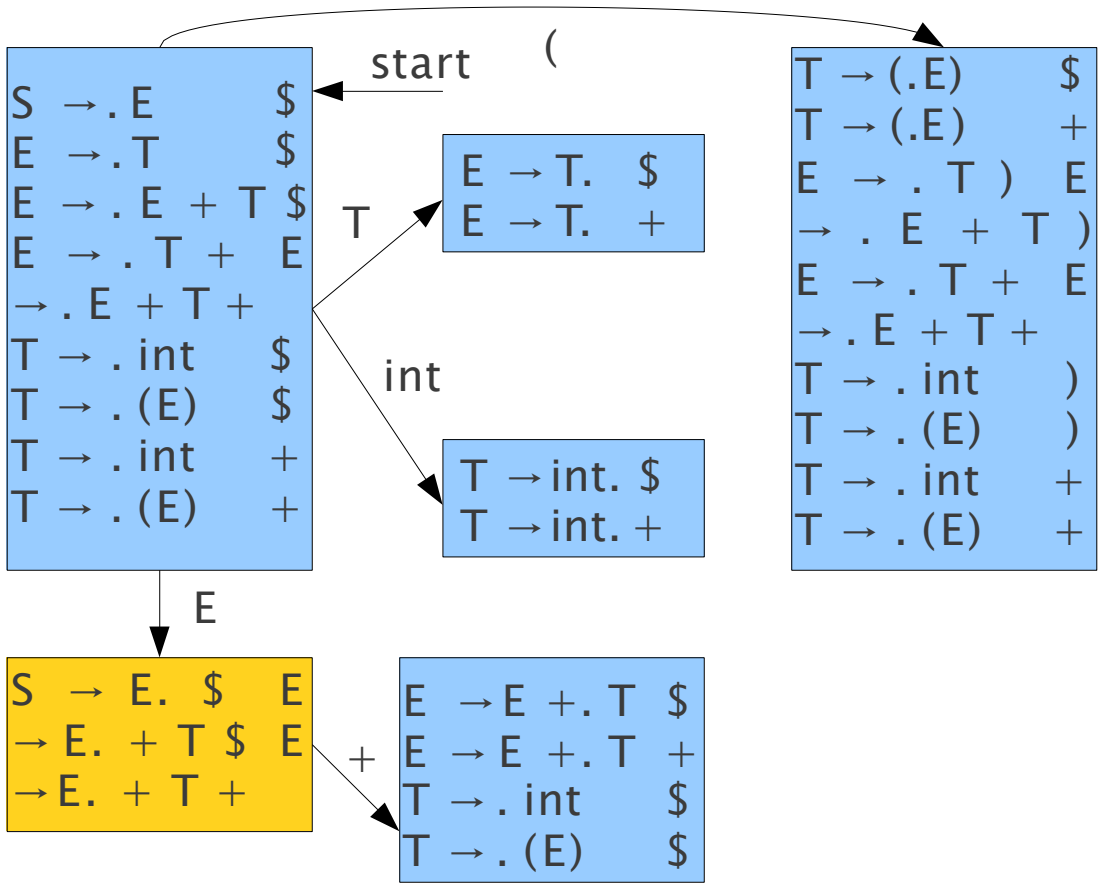
Deterministic LR(1) Automata



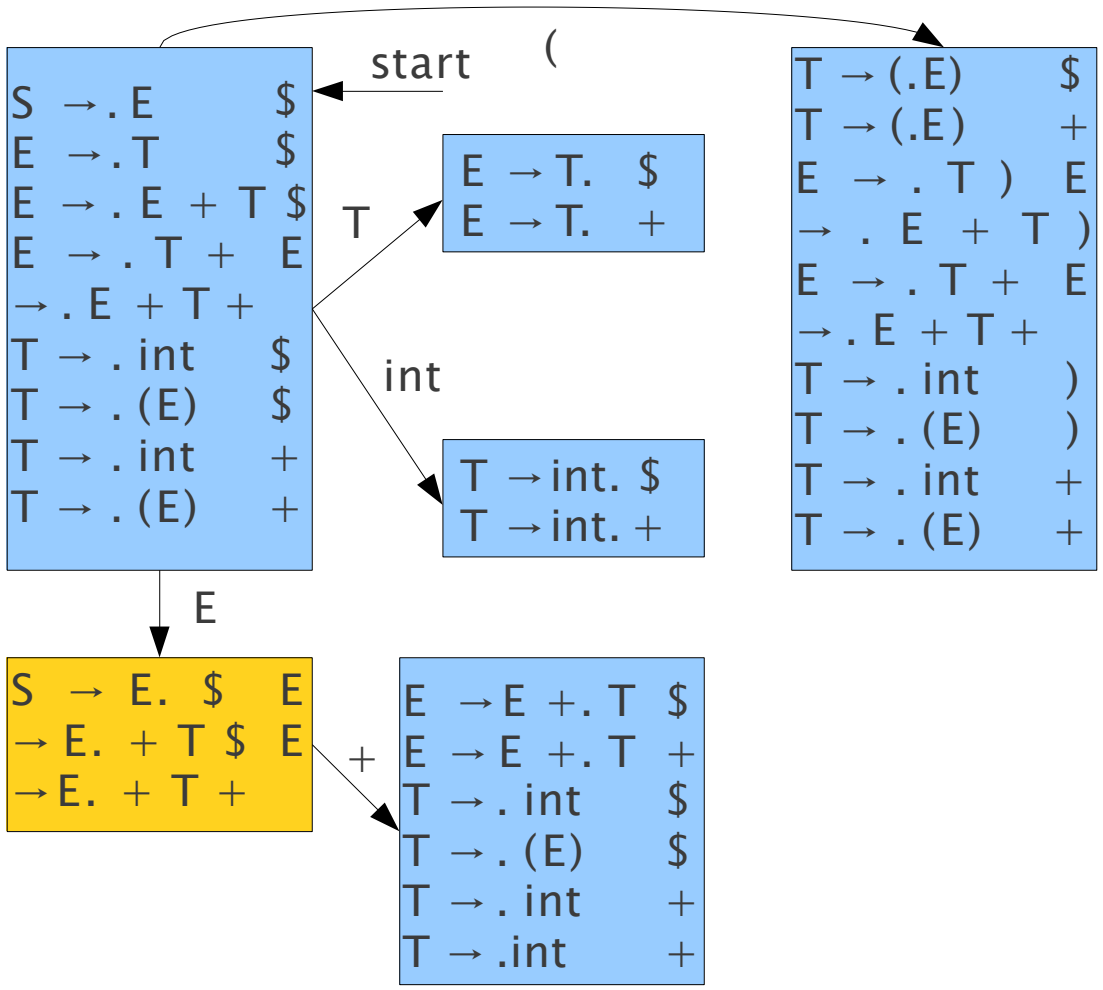
Deterministic LR(1) Automata



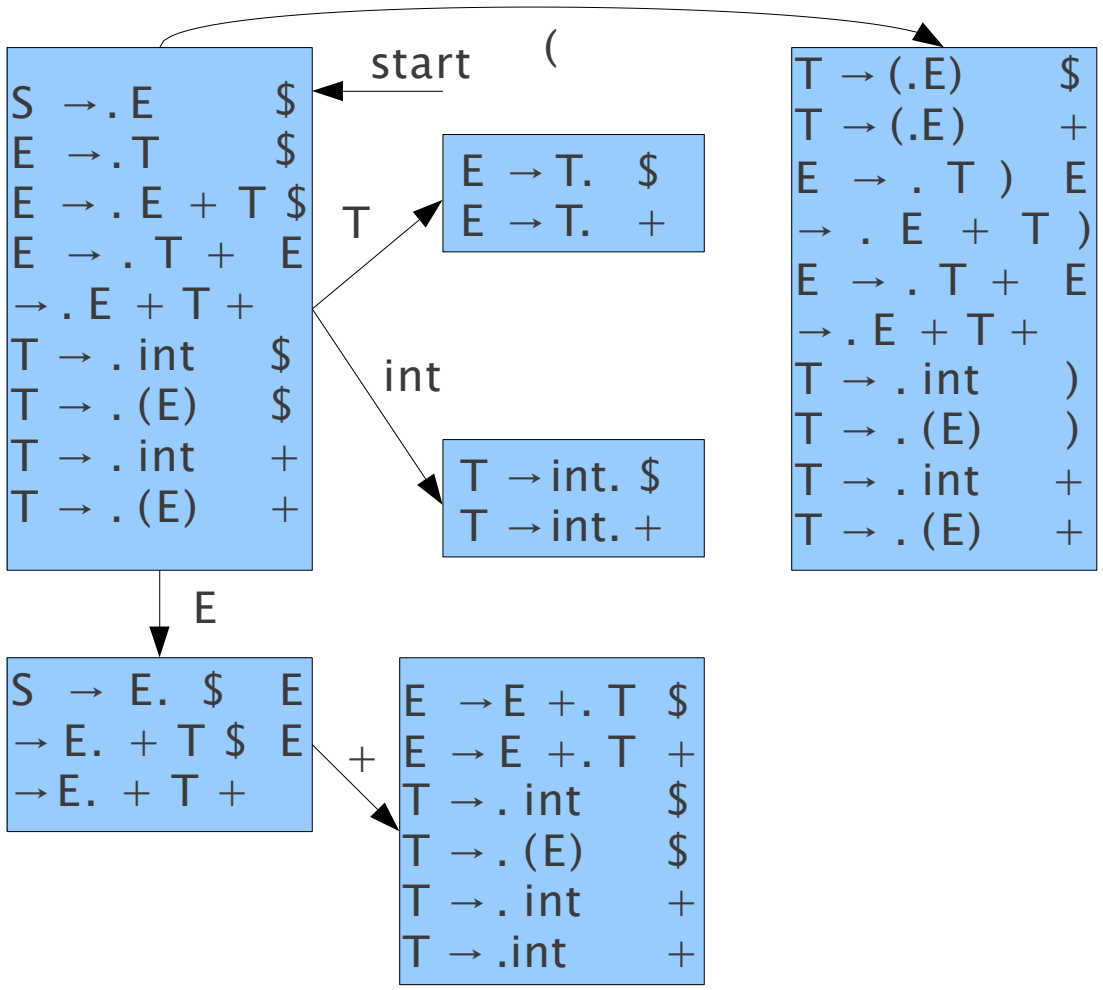
Deterministic LR(1) Automata



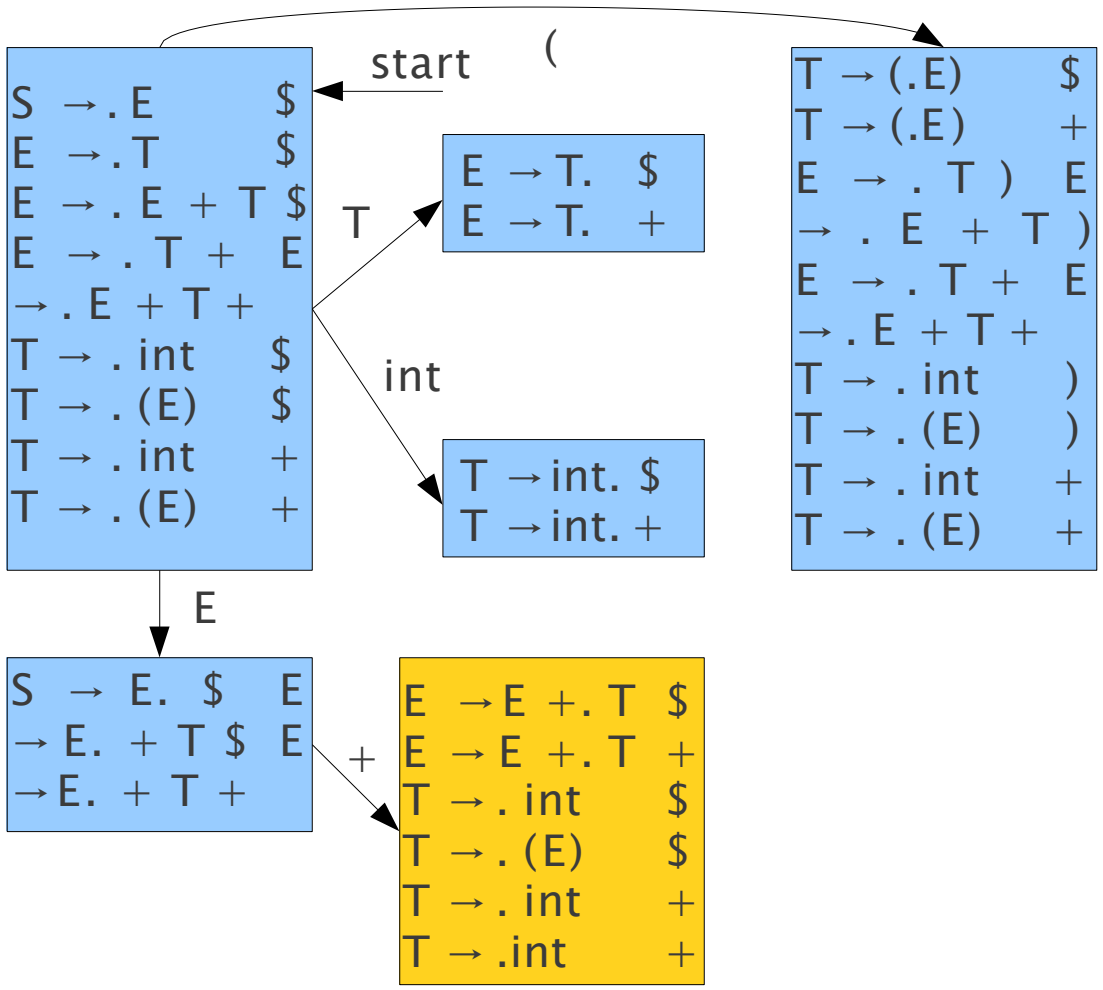
Deterministic LR(1) Automata



Deterministic LR(1) Automata

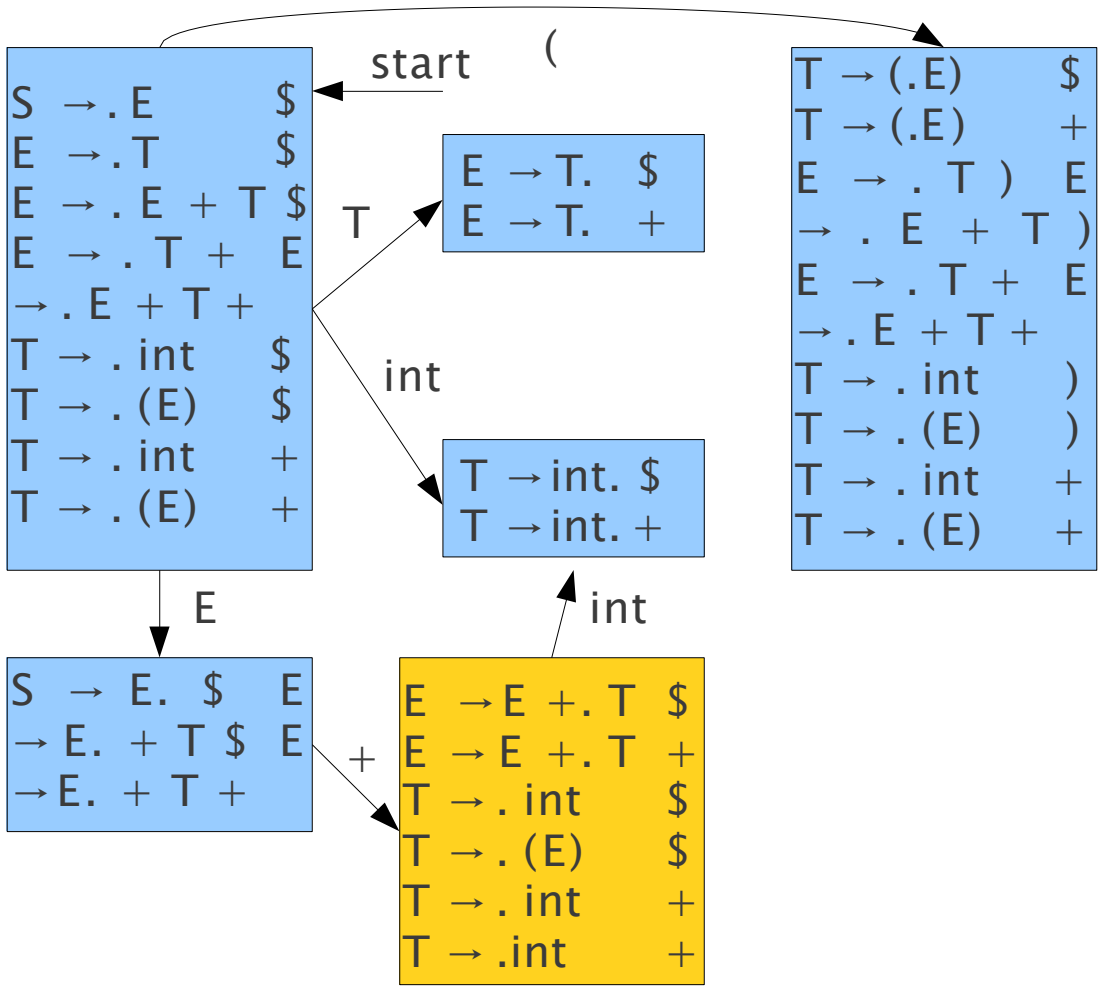


Deterministic LR(1) Automata

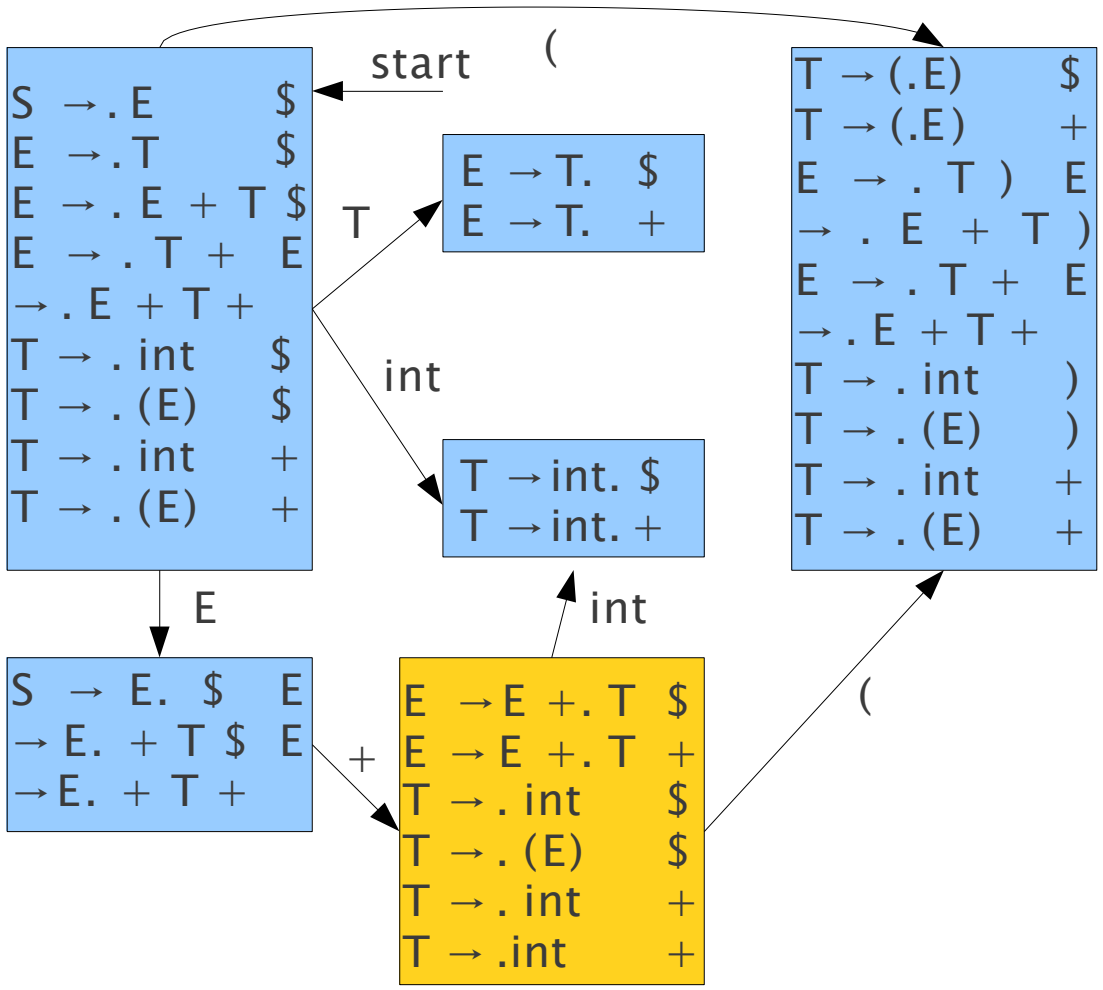


GEORGETOWN UNIVERSITY

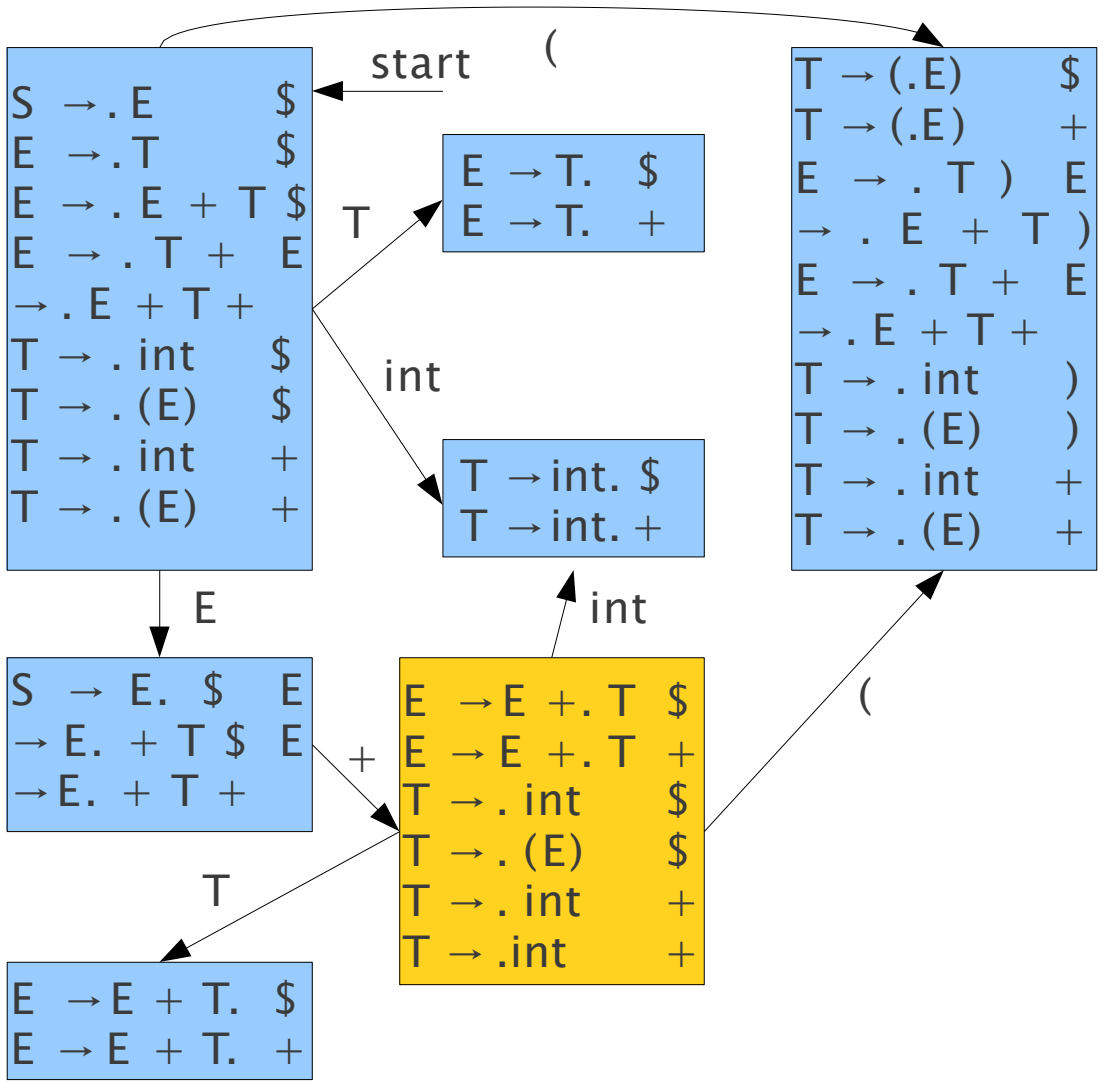
Deterministic LR(1) Automata



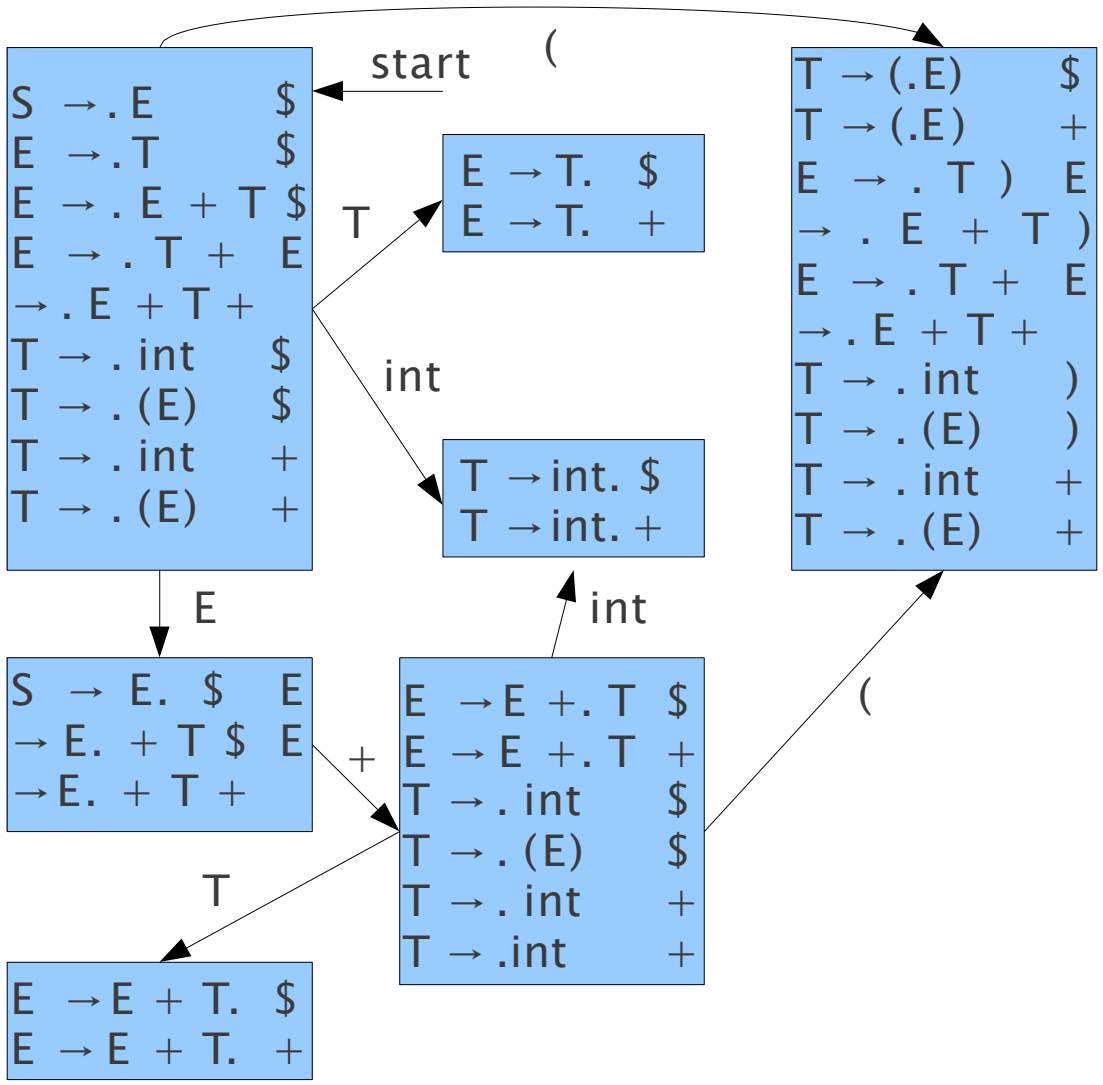
Deterministic LR(1) Automata



Deterministic LR(1) Automata

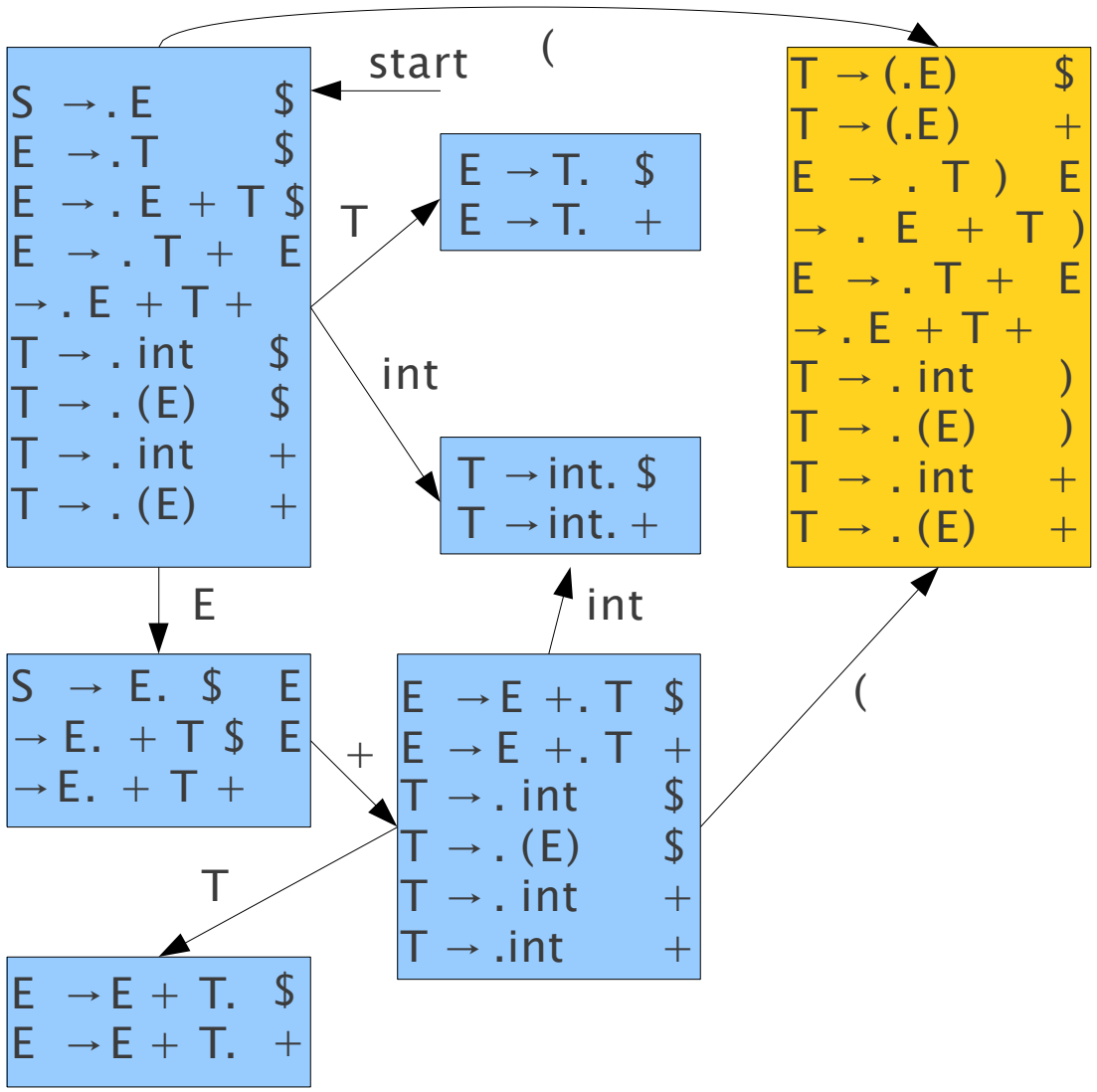


Deterministic LR(1) Automata

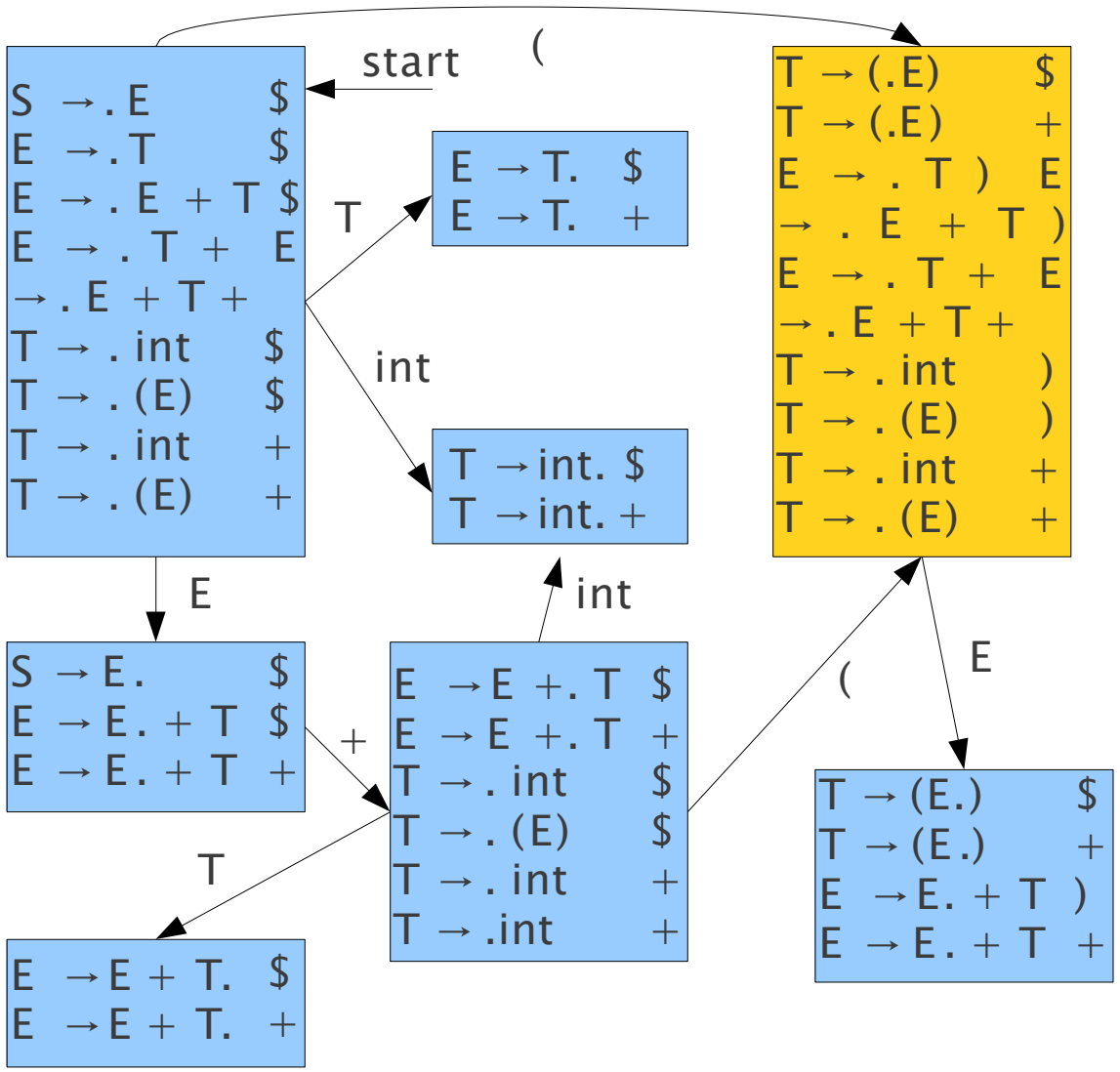


GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

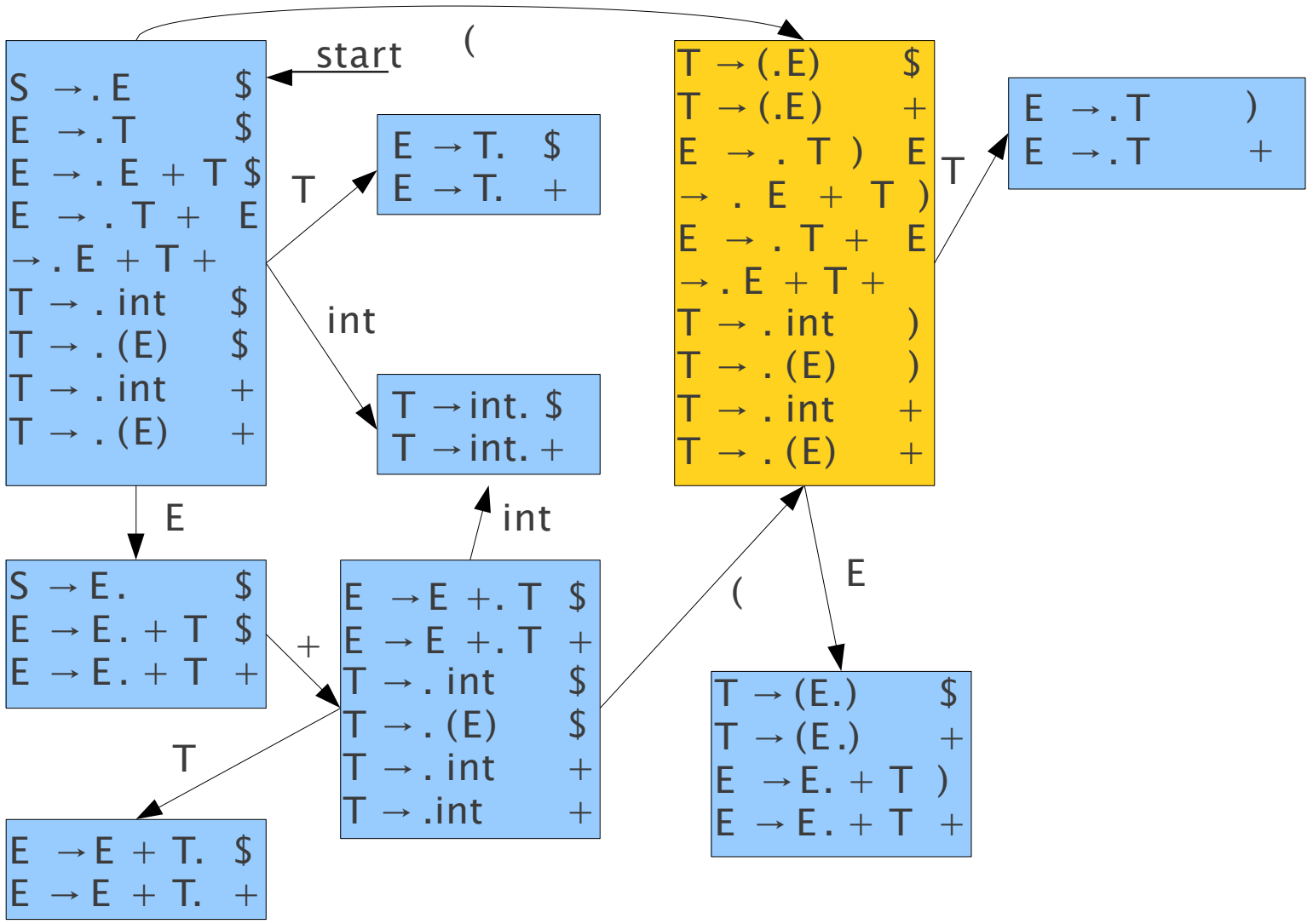


Deterministic LR(1) Automata



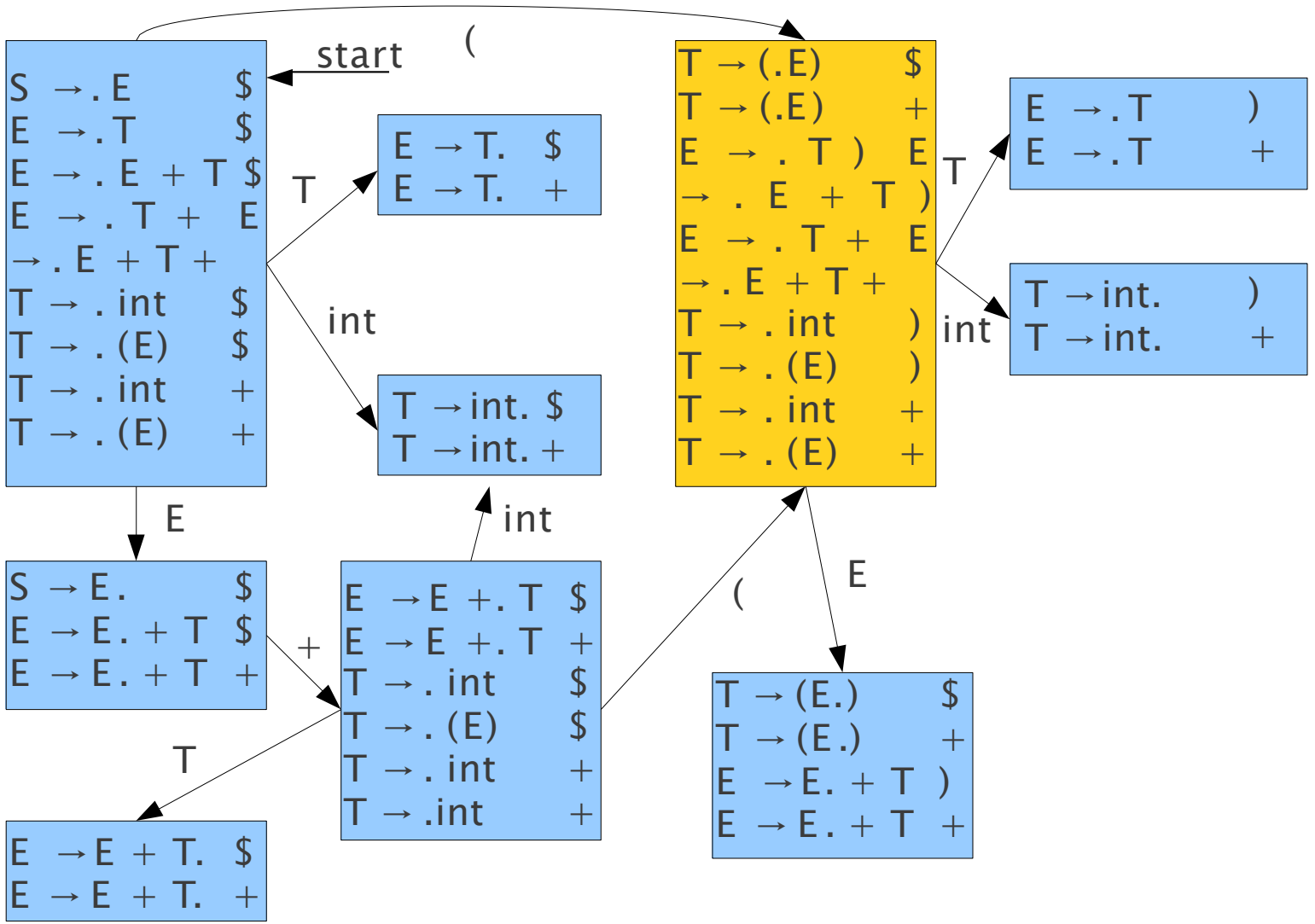
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

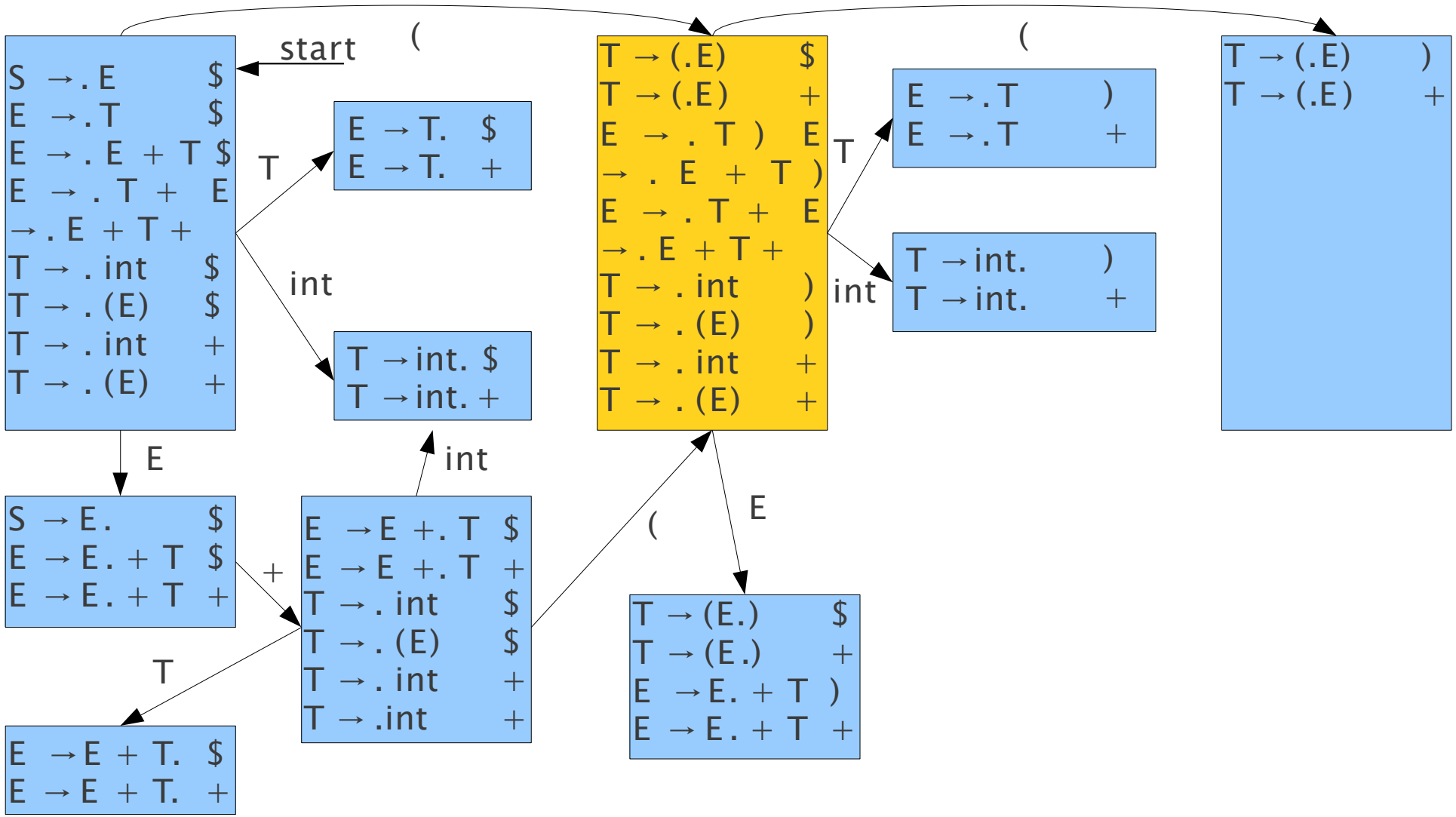


GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

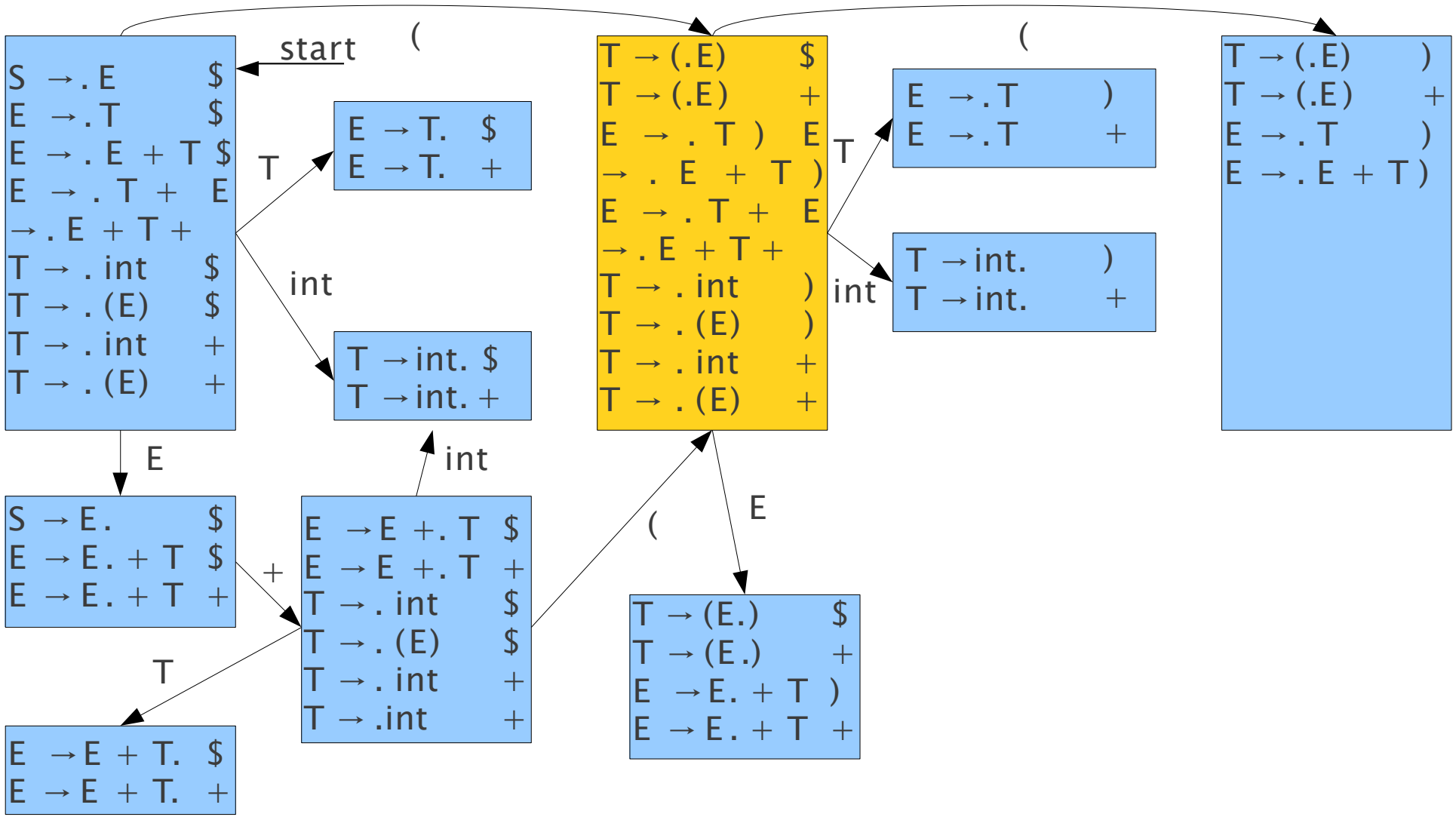


Deterministic LR(1) Automata



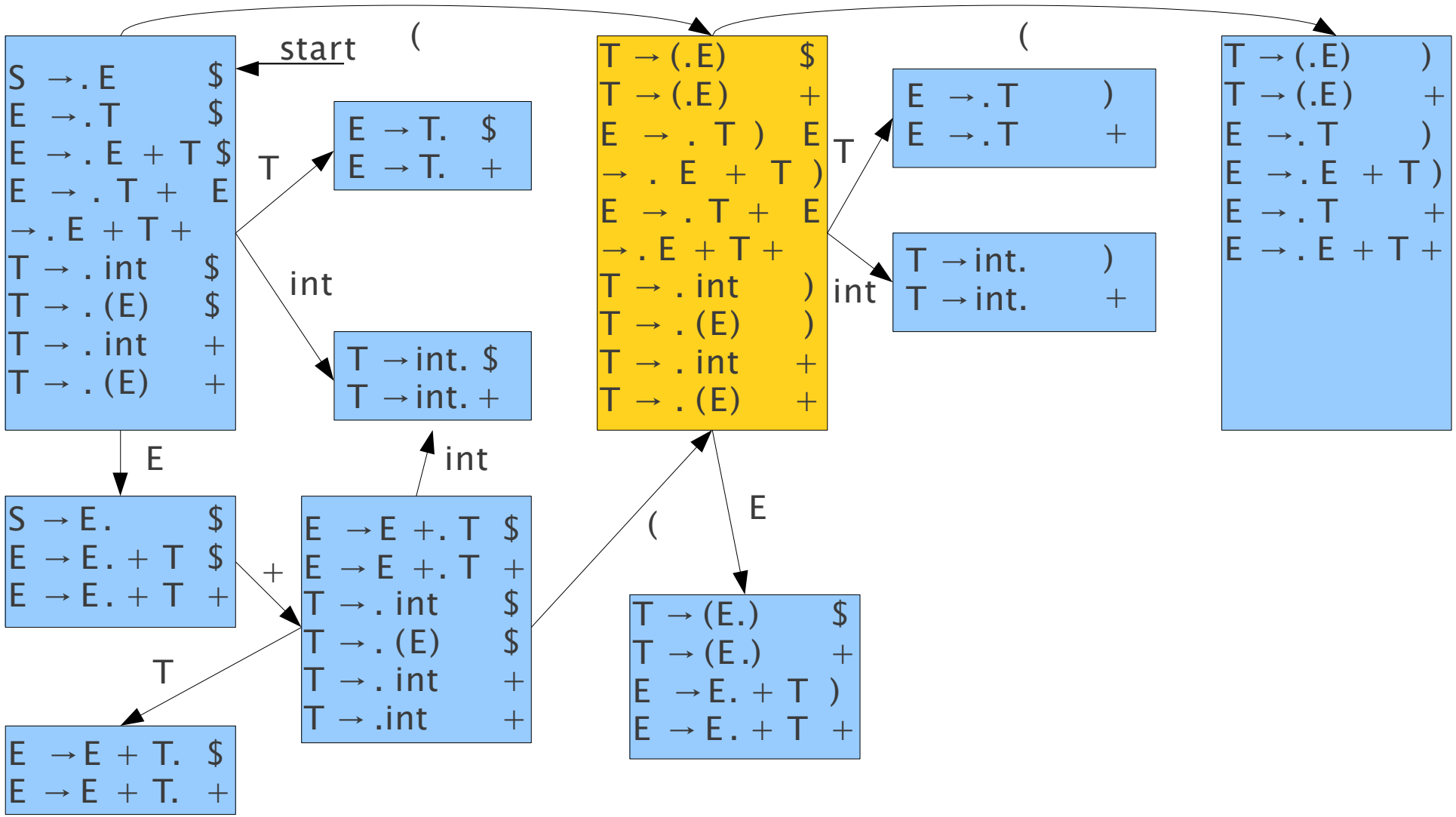
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



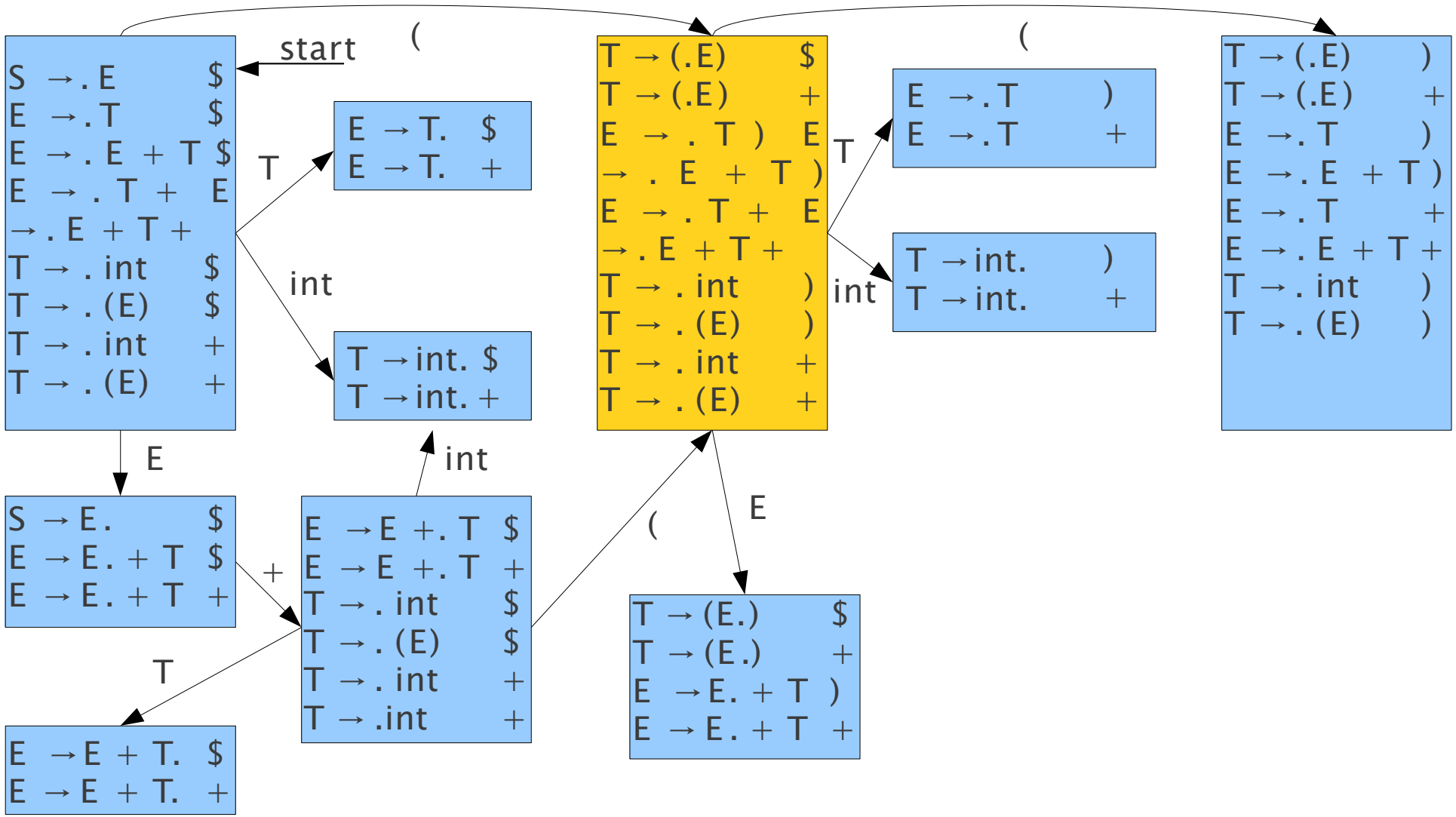
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



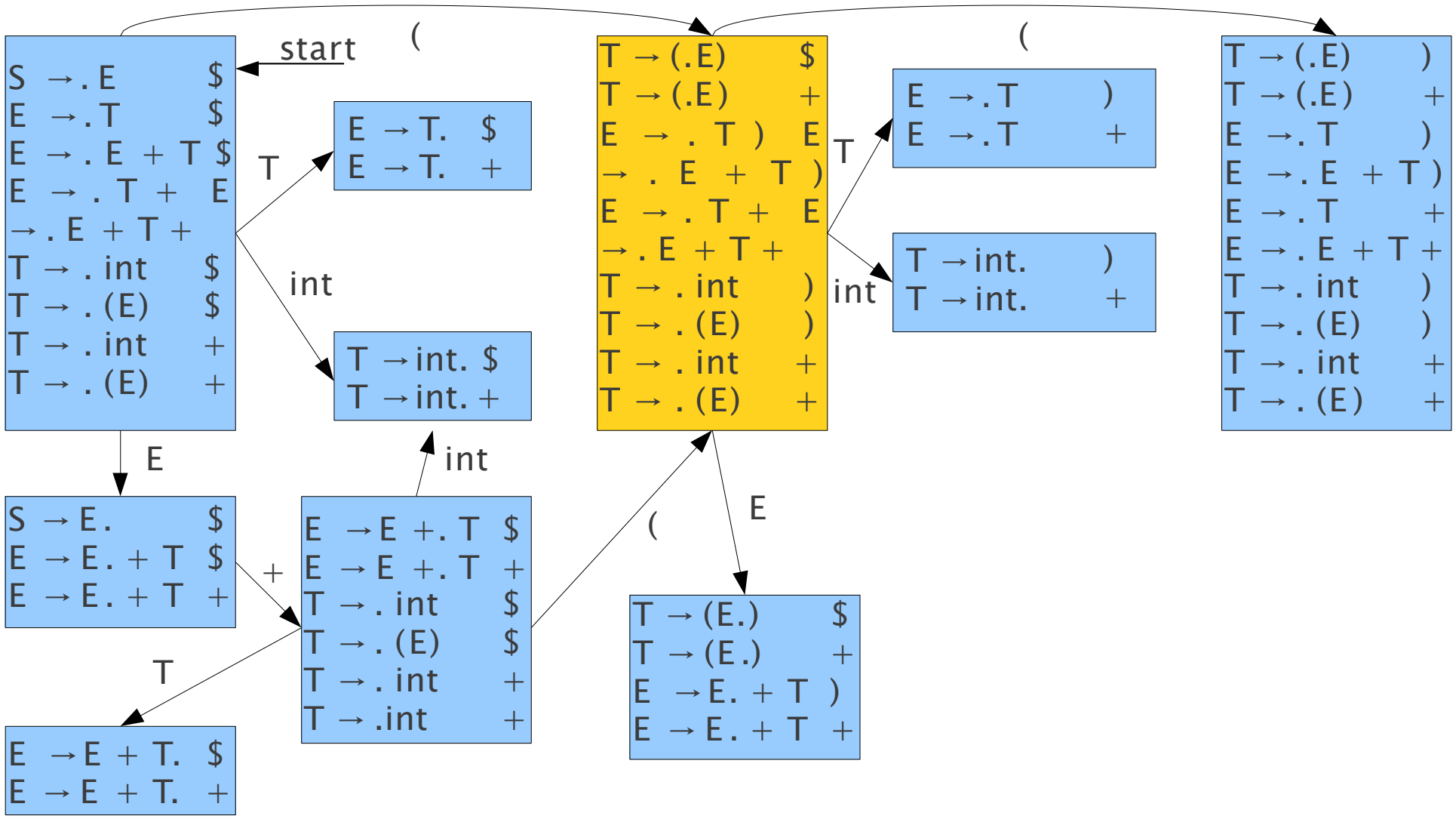
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



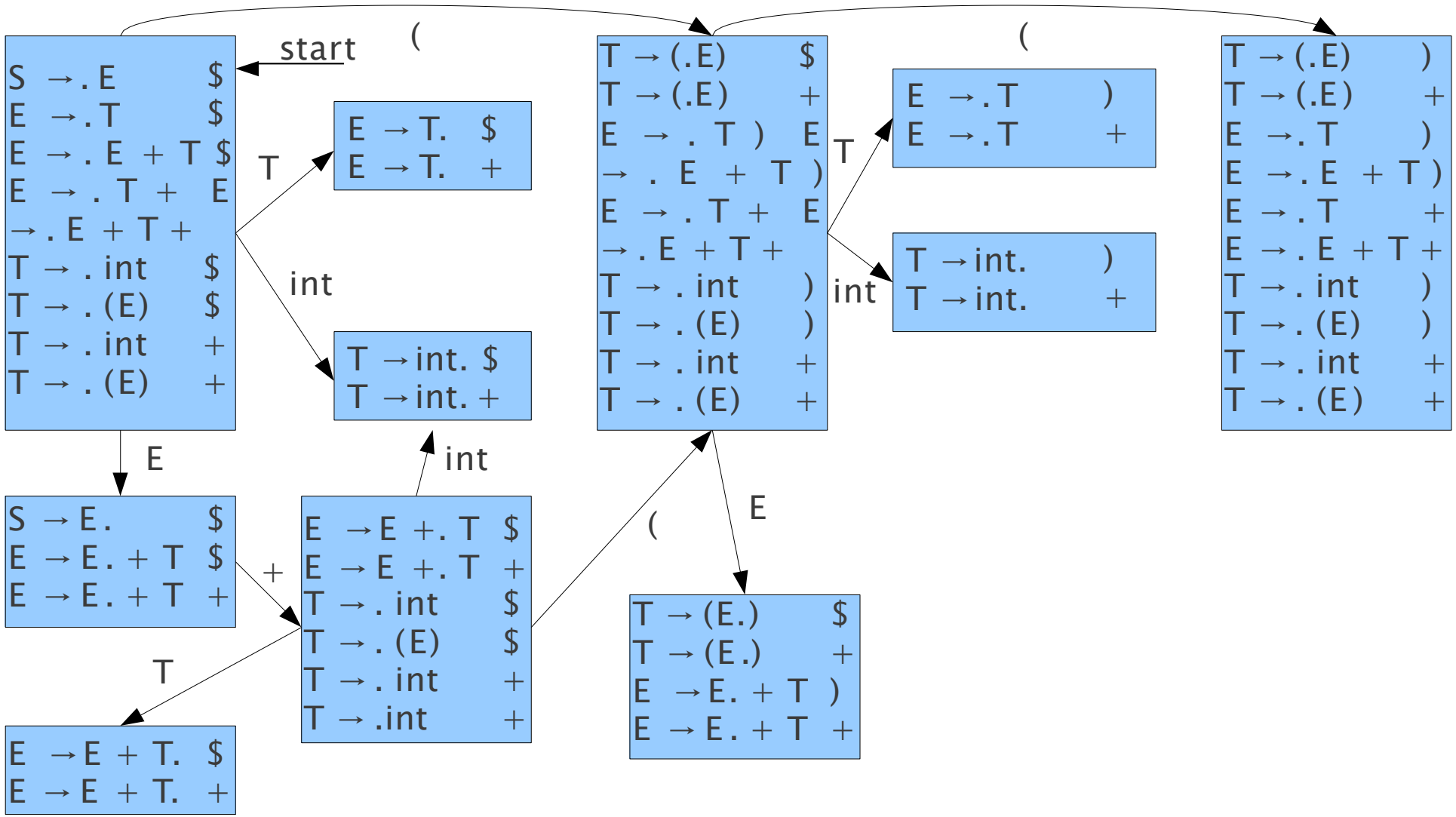
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



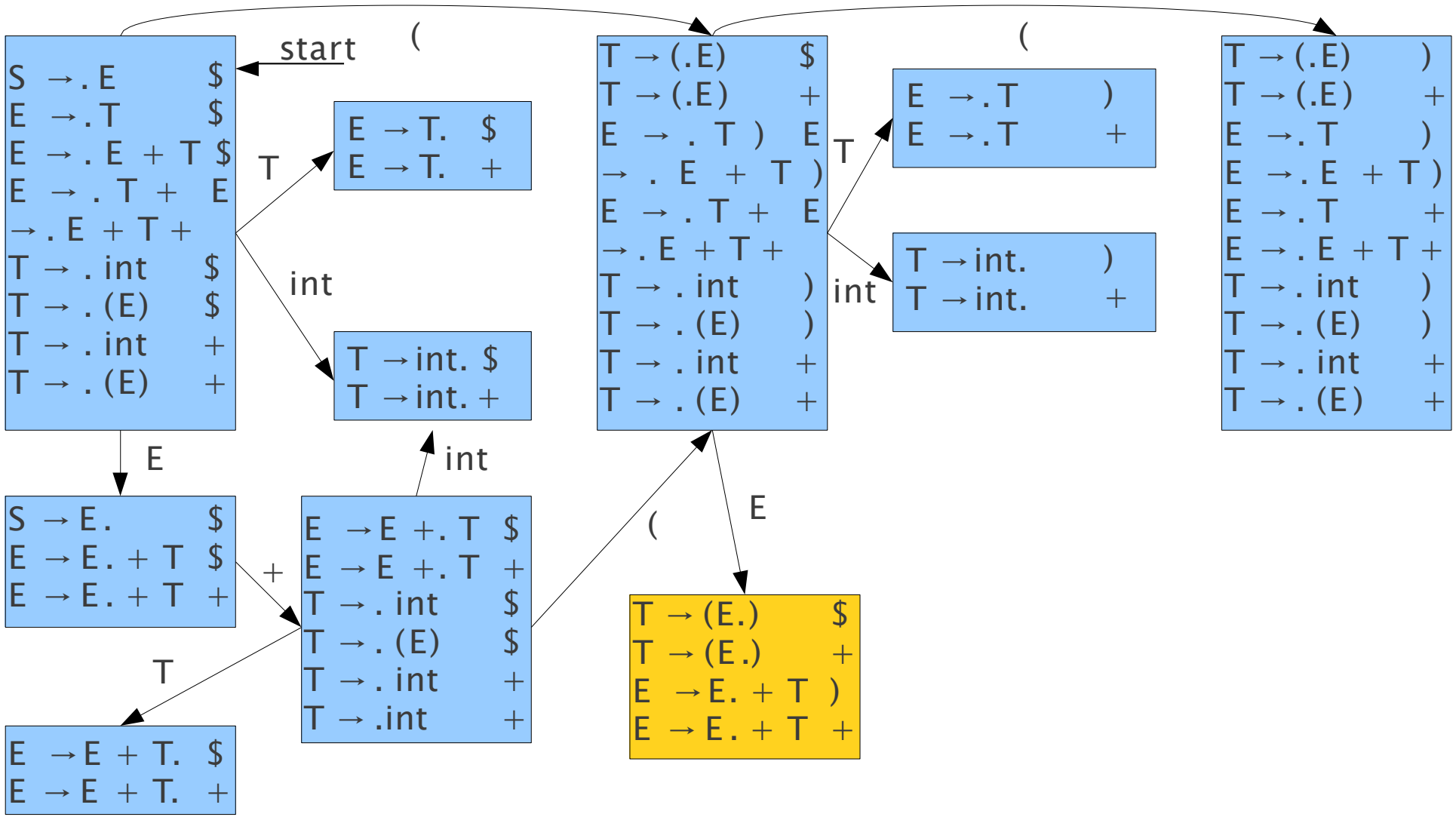
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

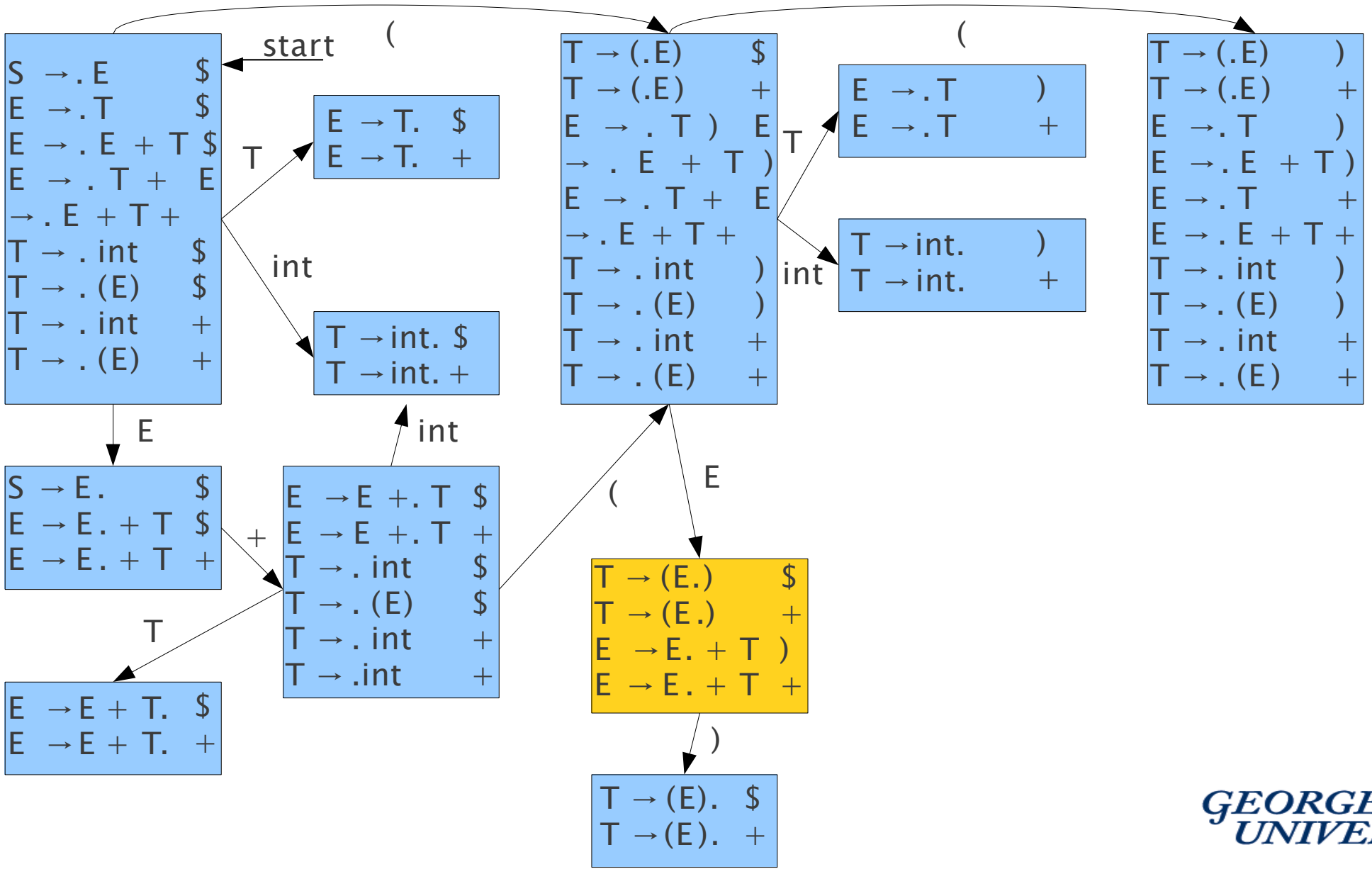


GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

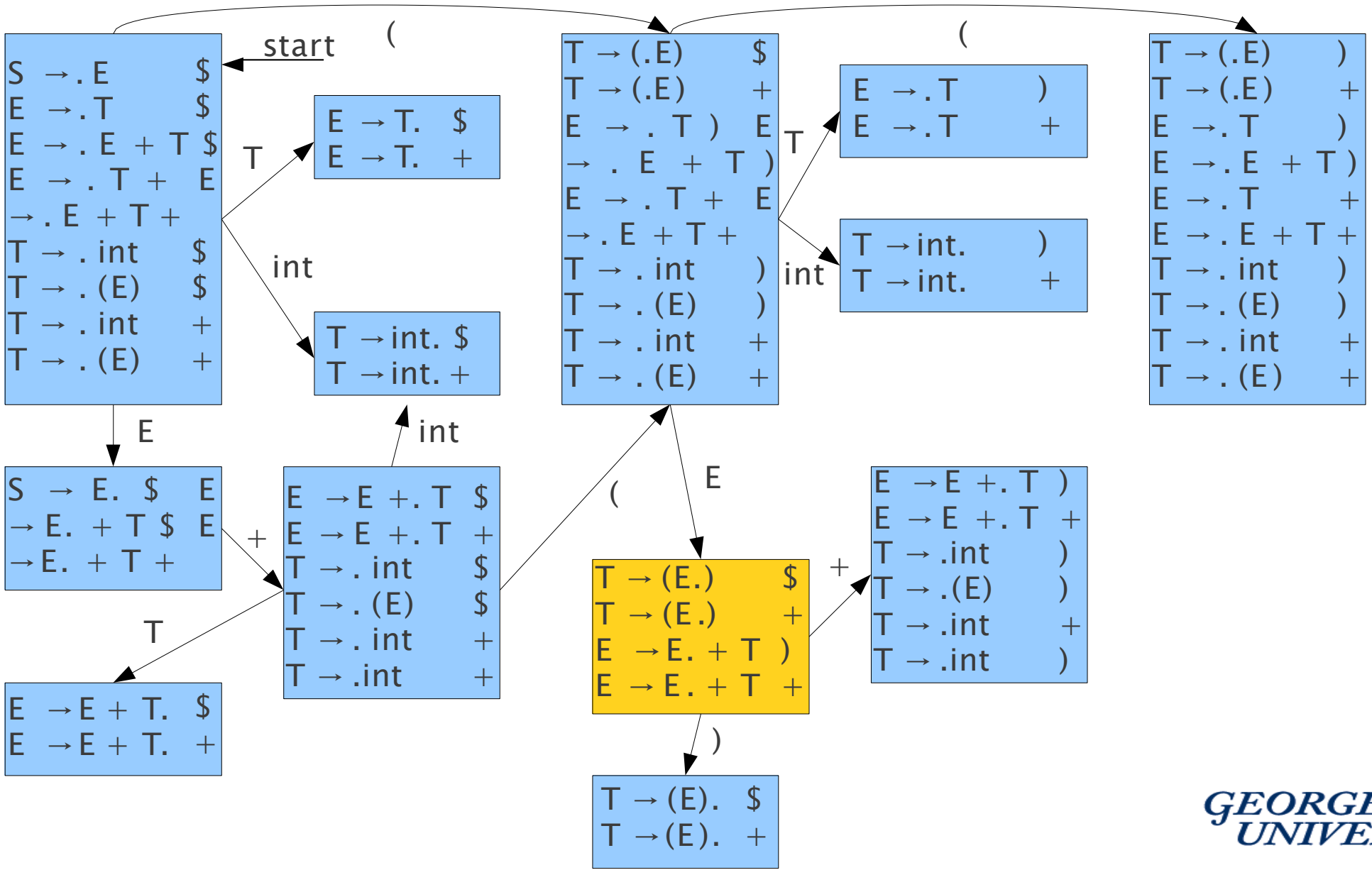


Deterministic LR(1) Automata



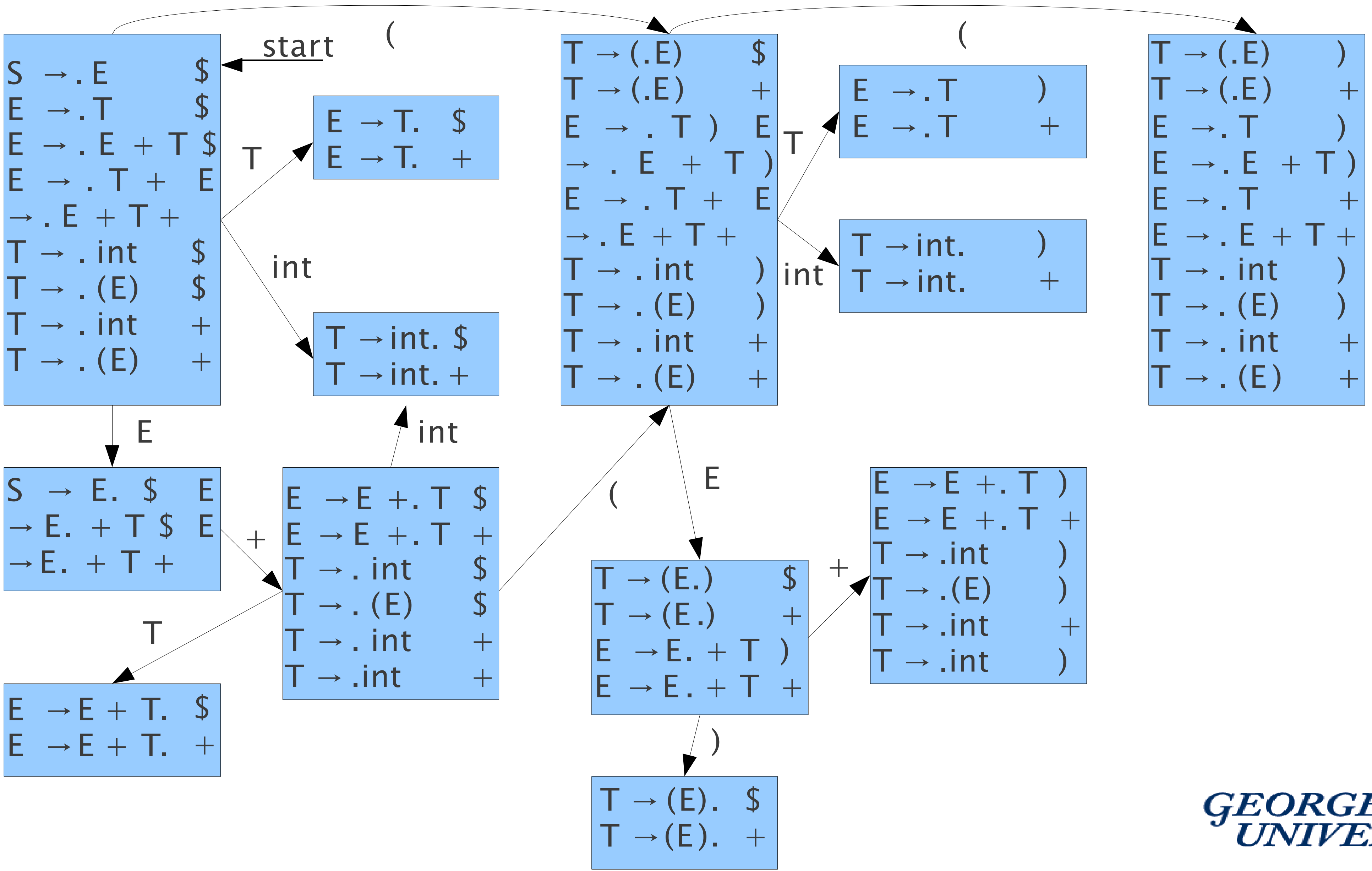
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



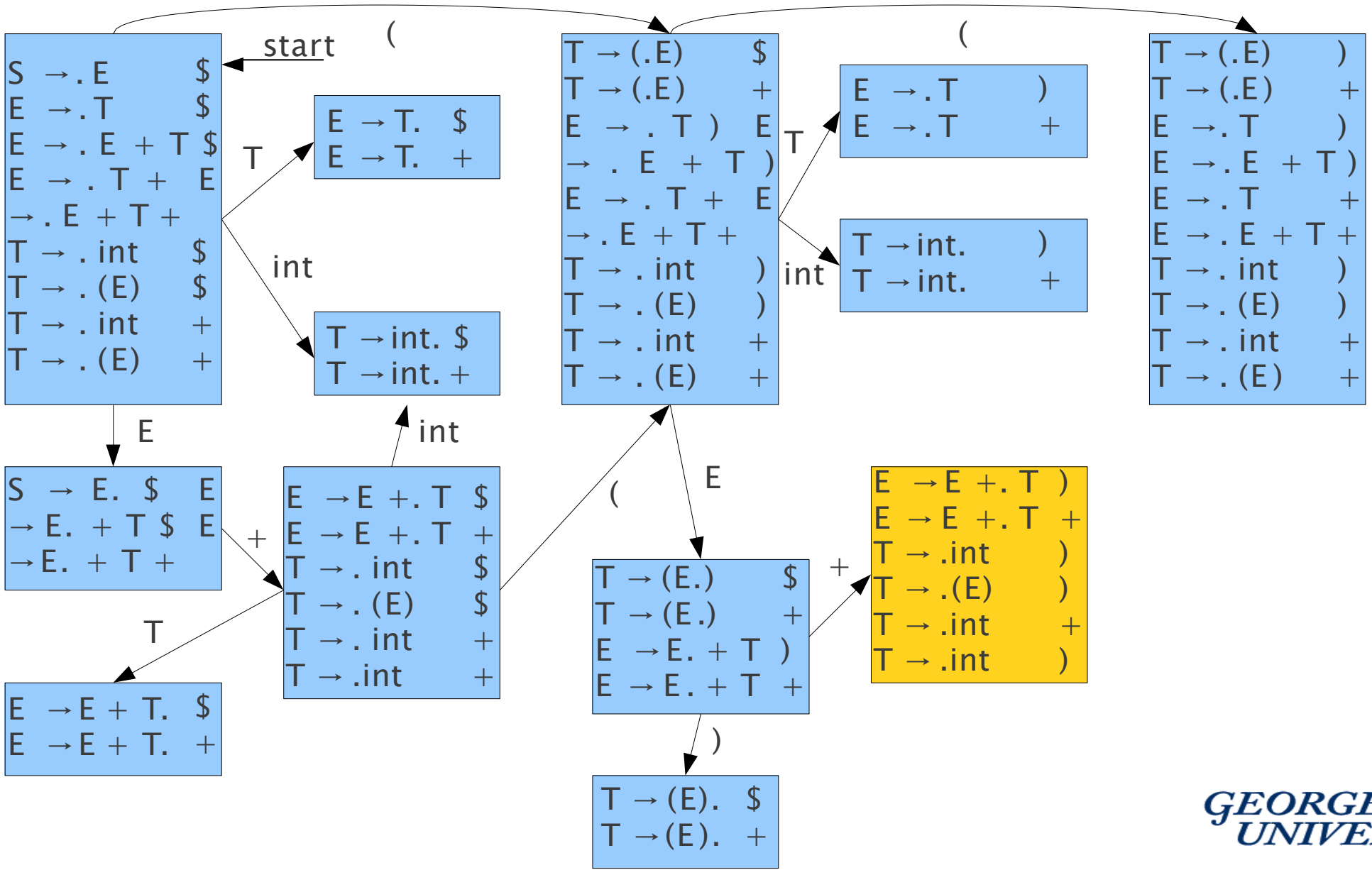
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



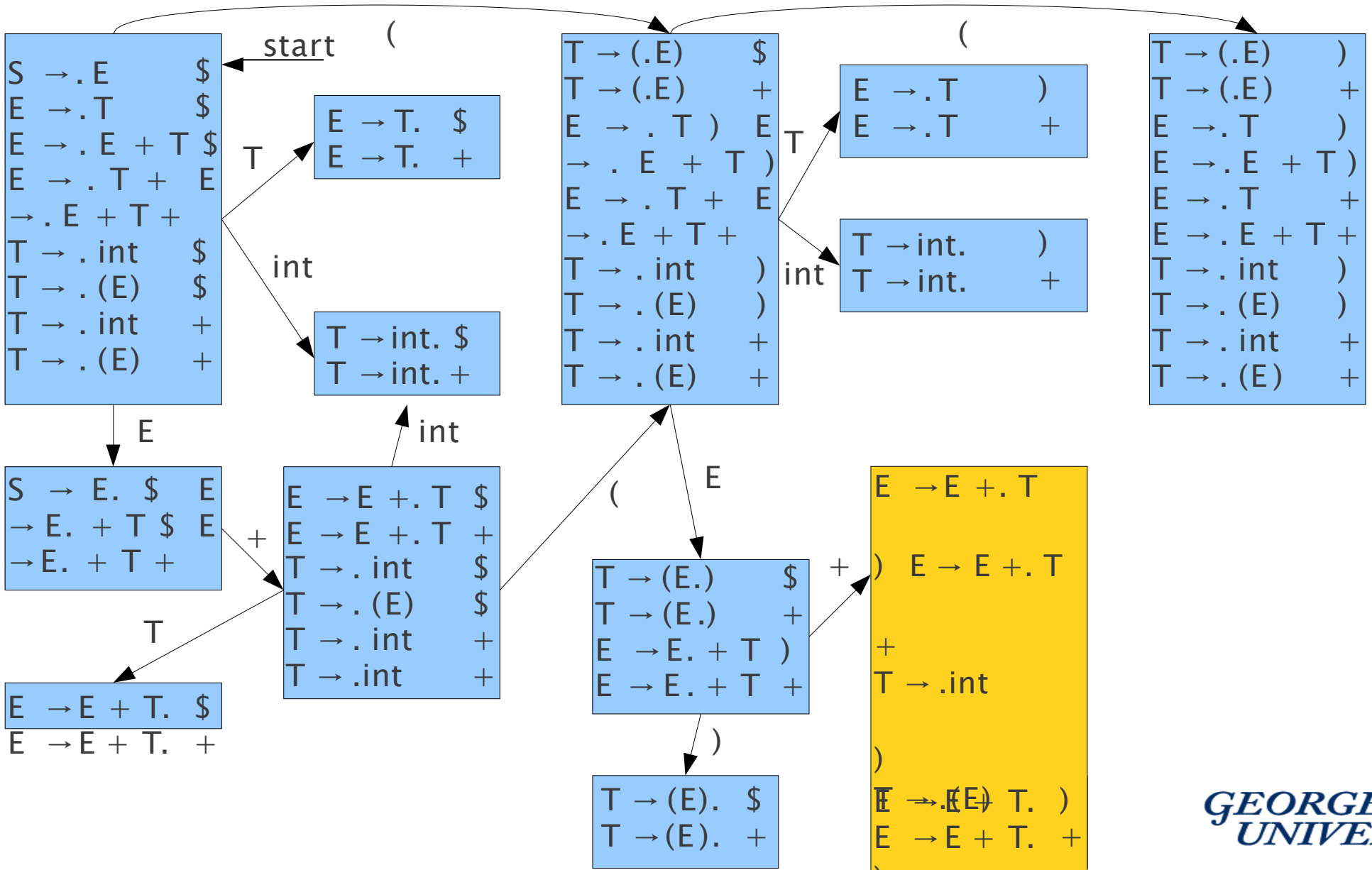
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



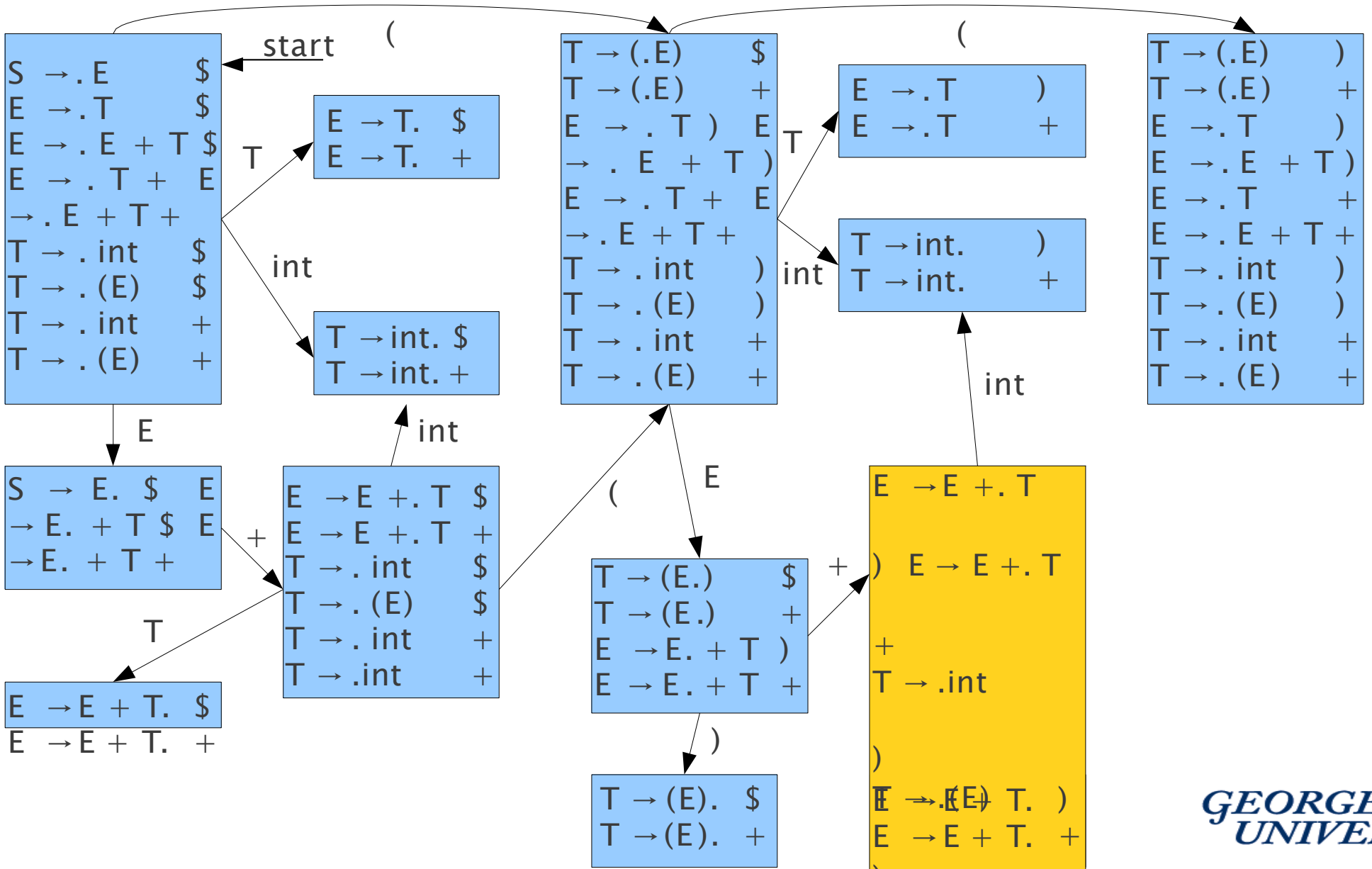
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



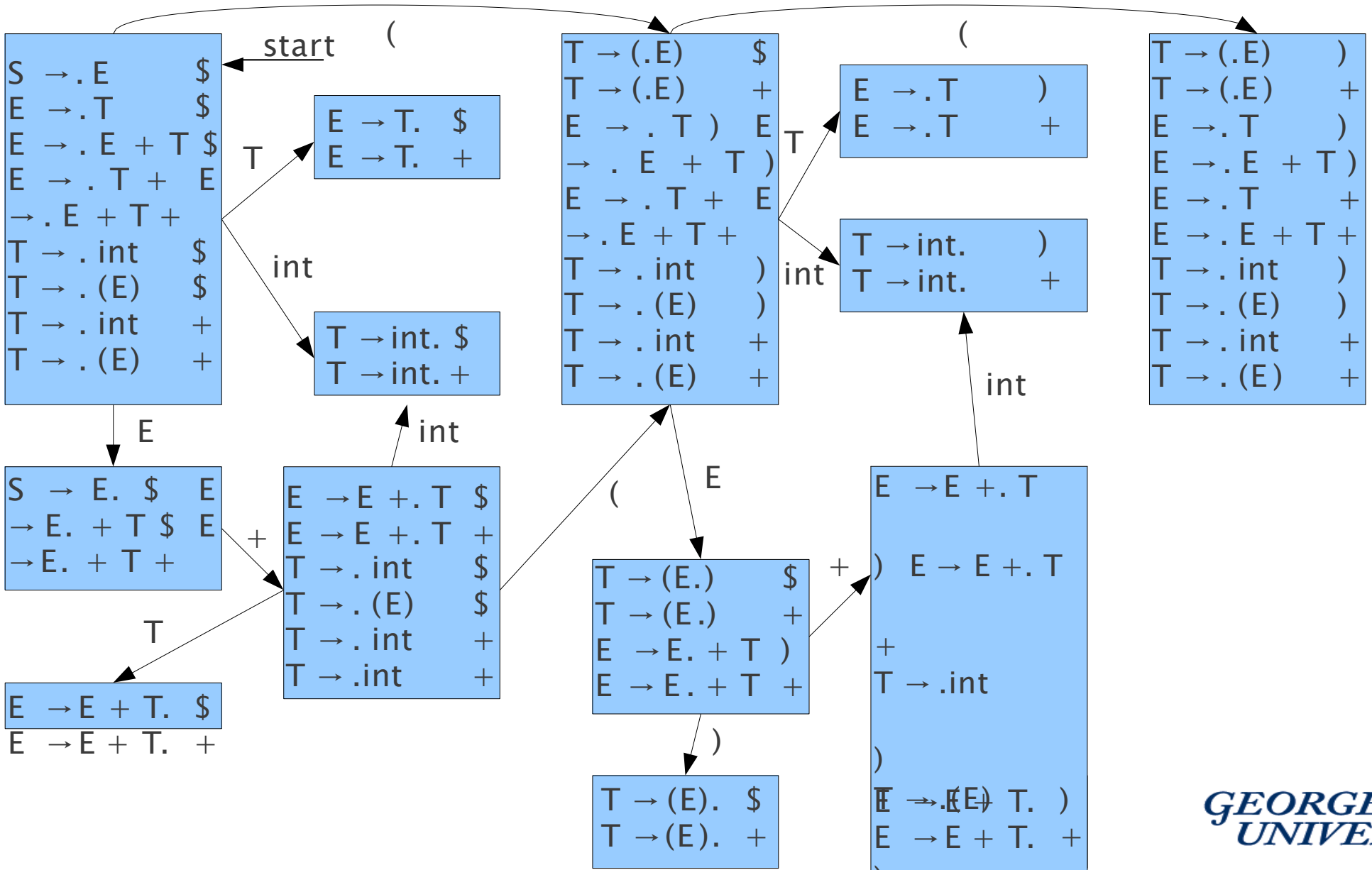
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



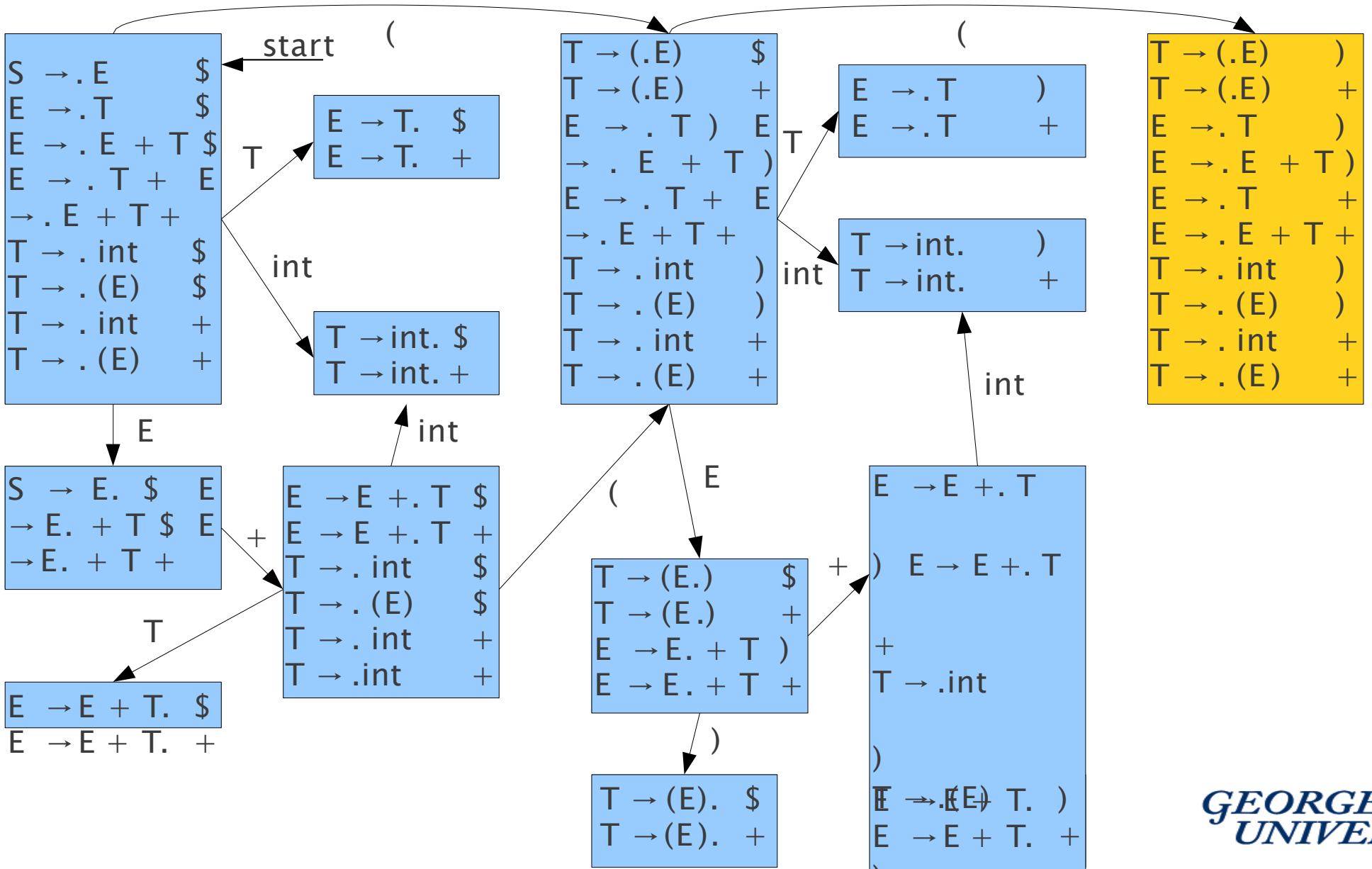
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



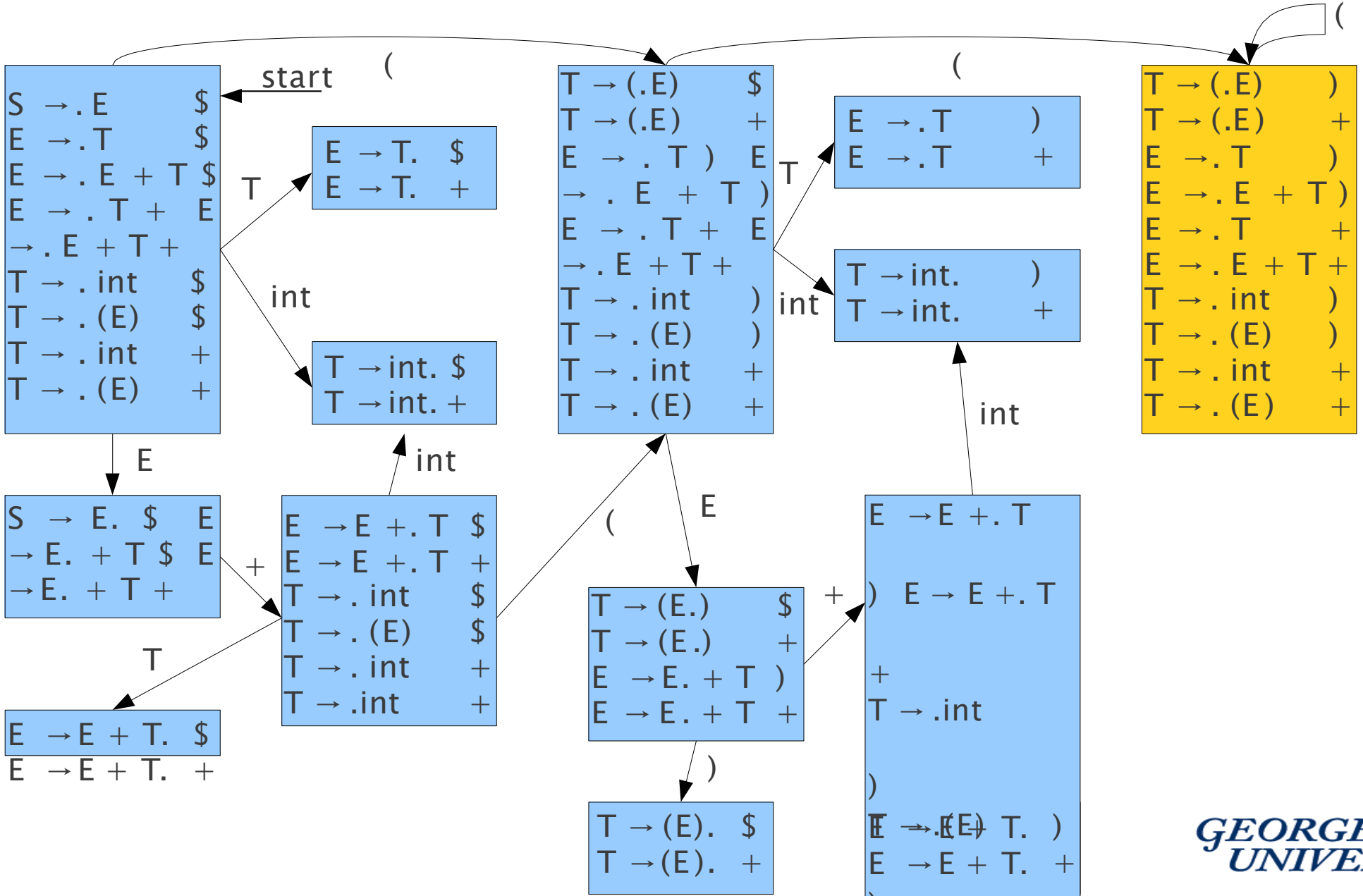
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



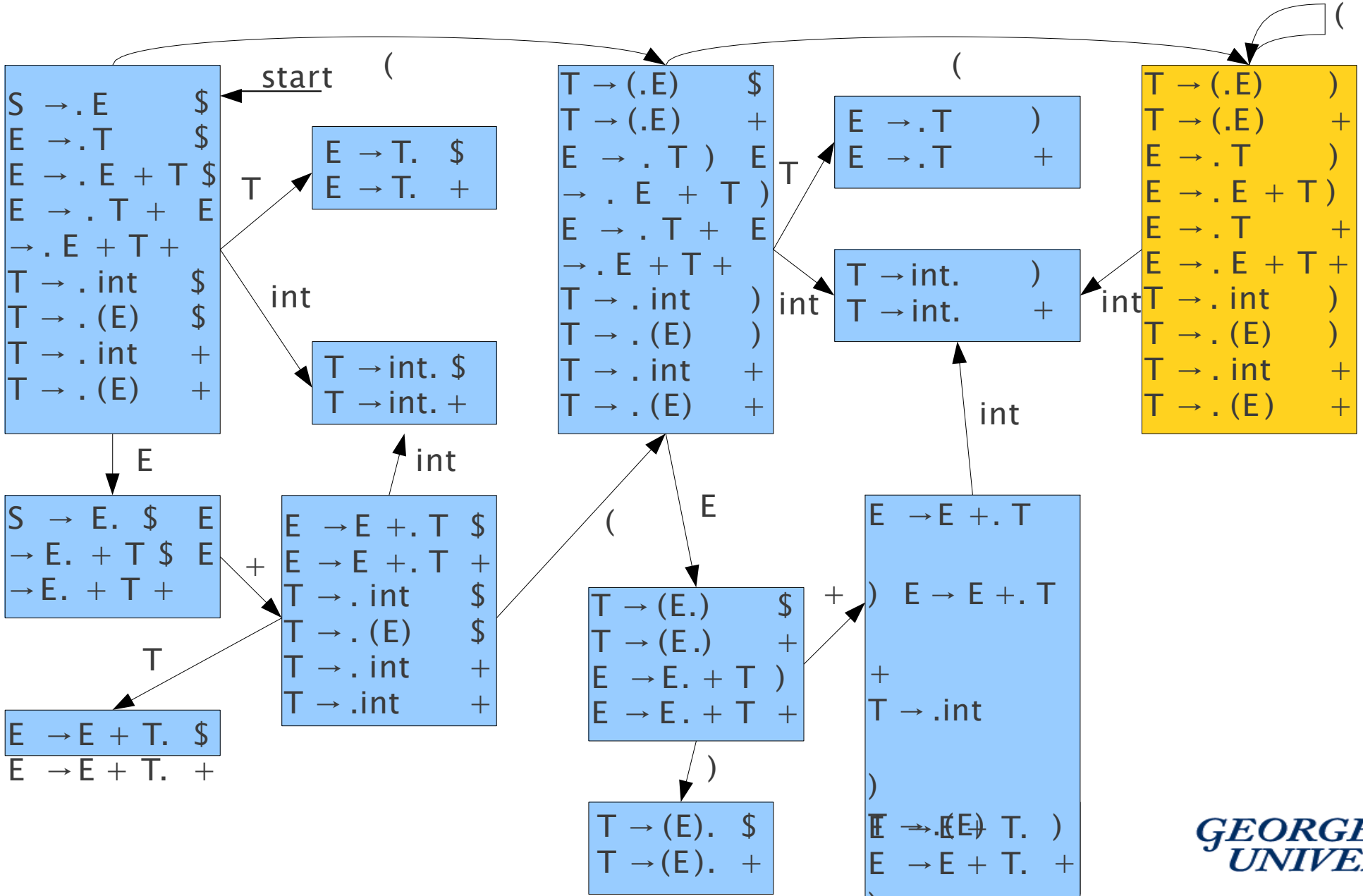
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



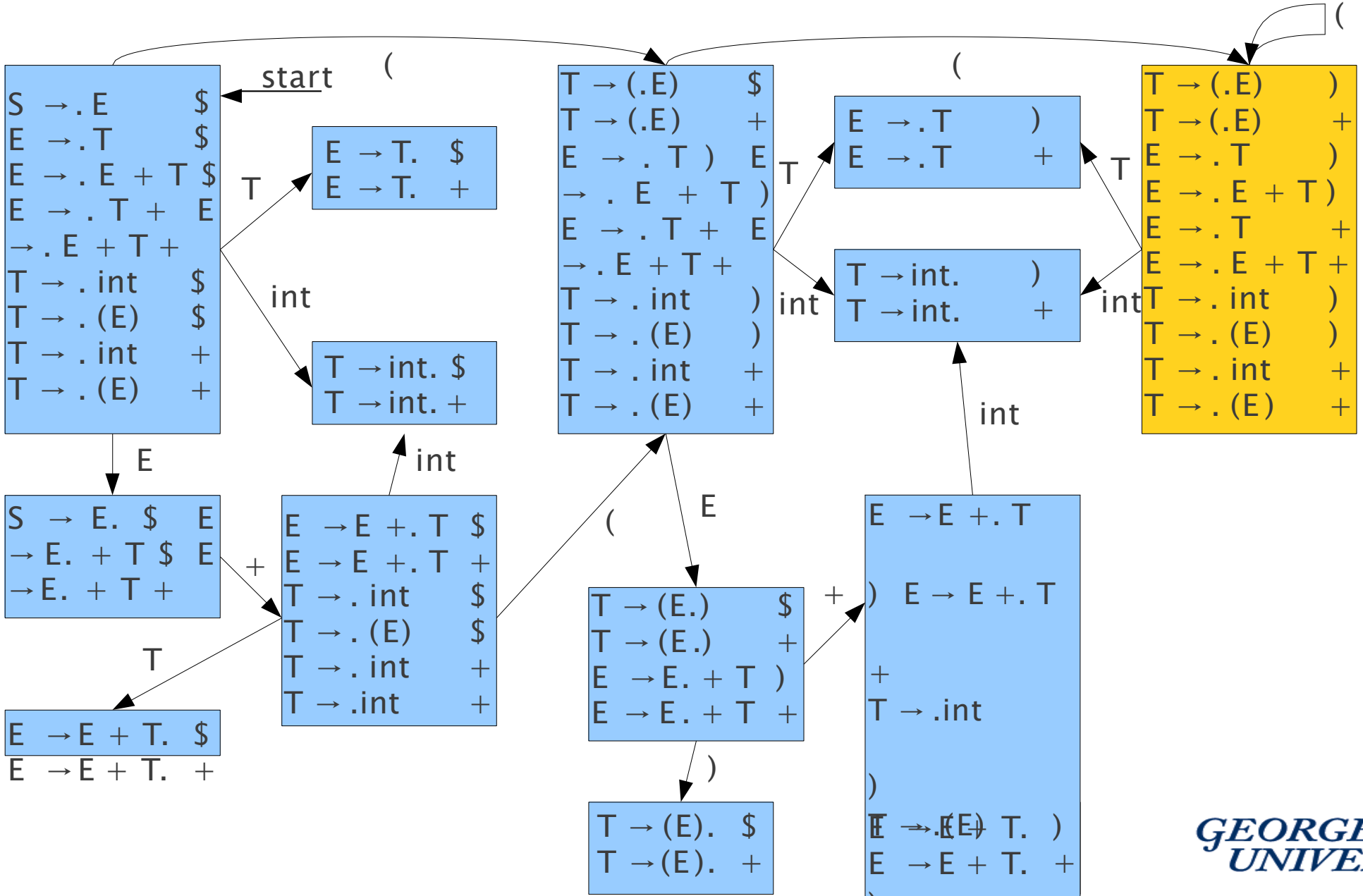
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



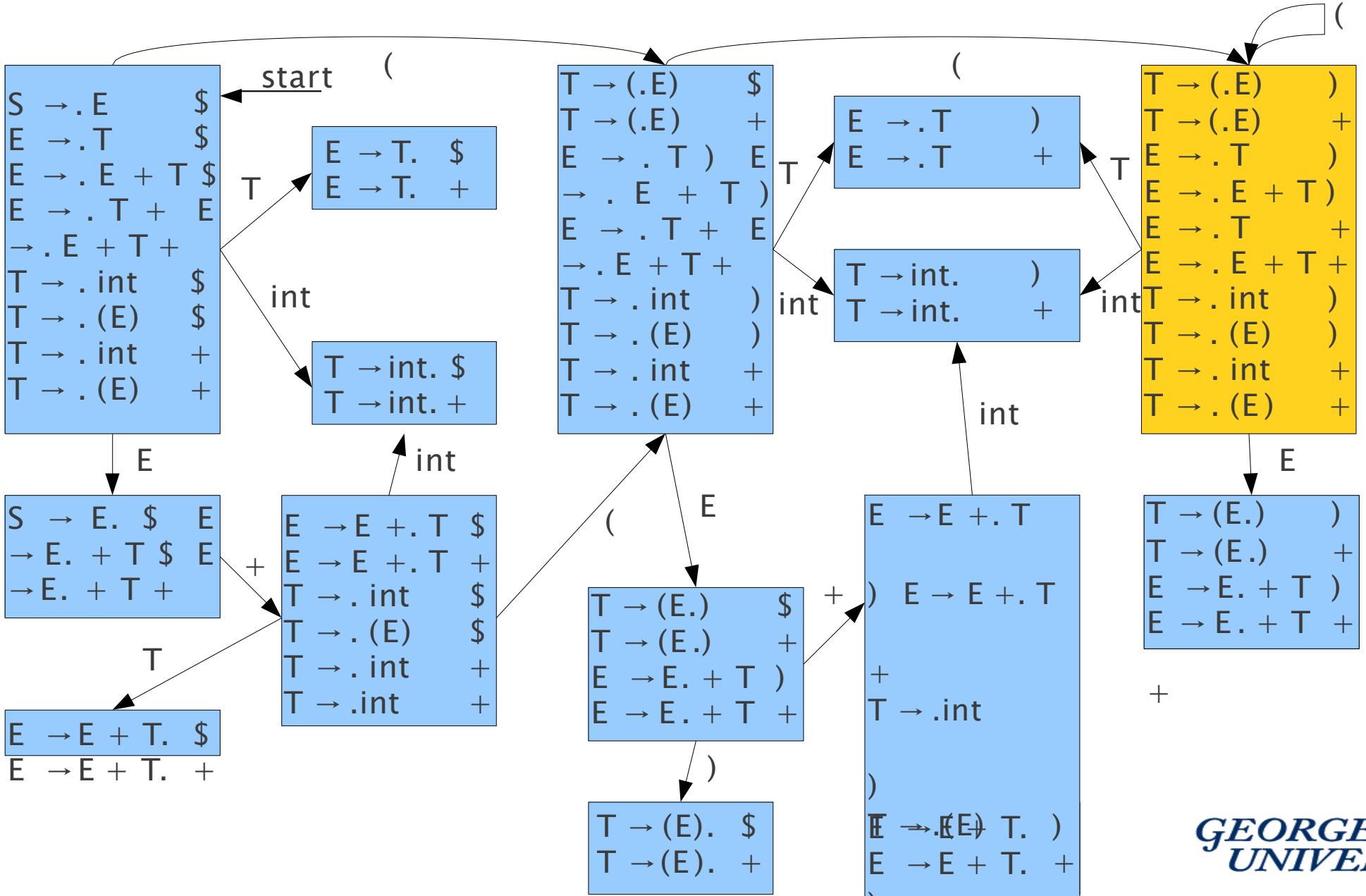
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



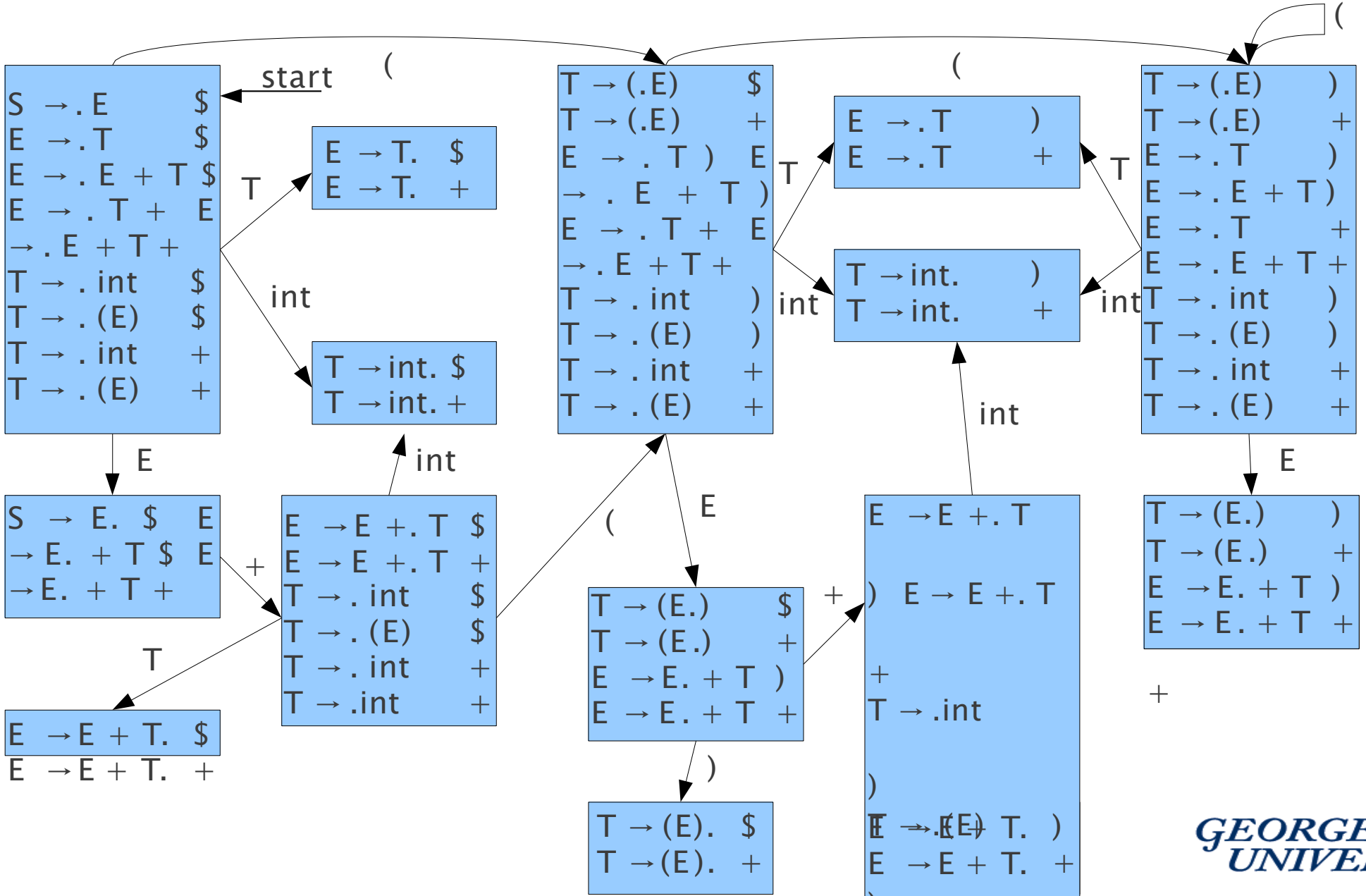
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



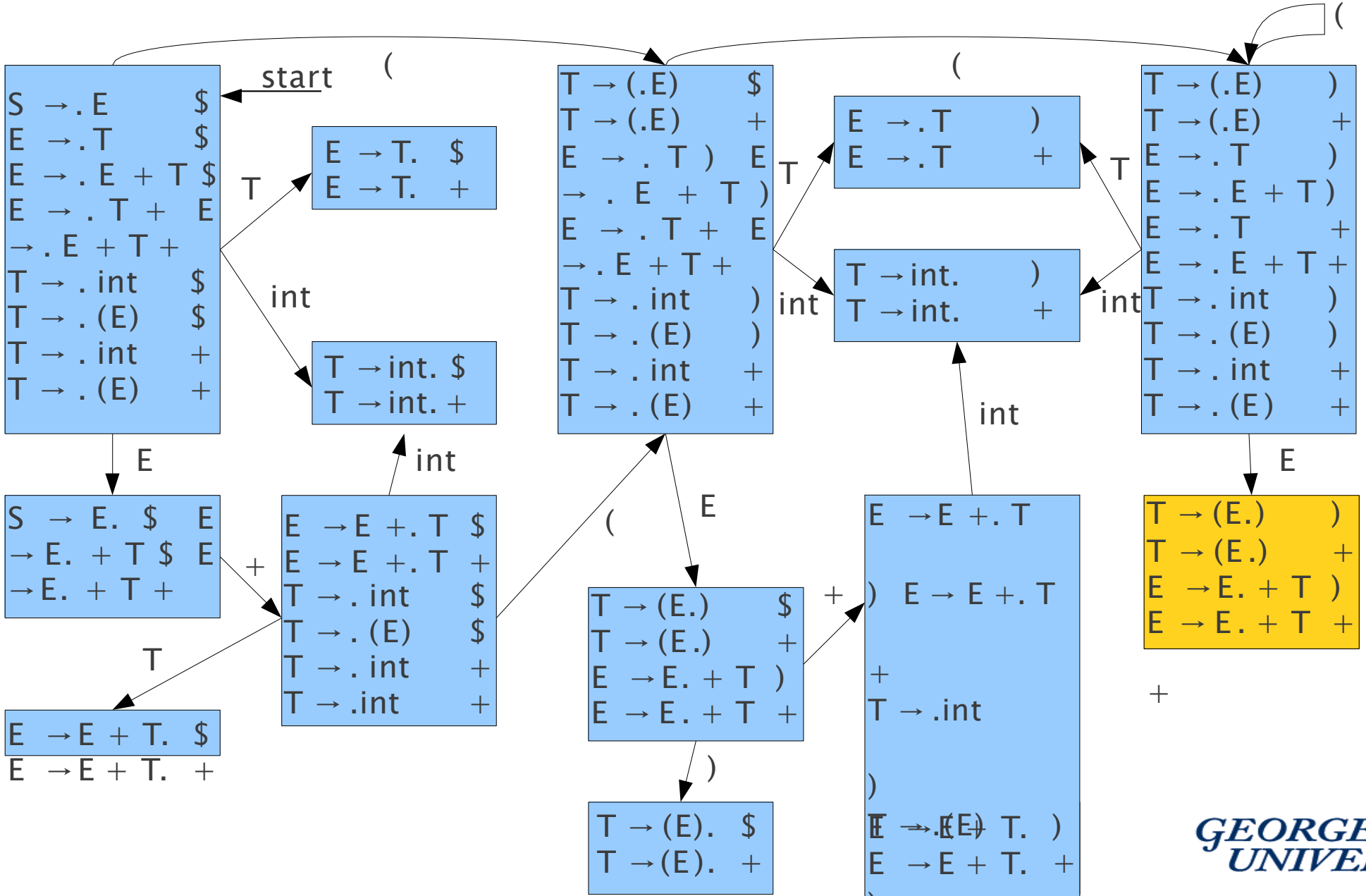
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



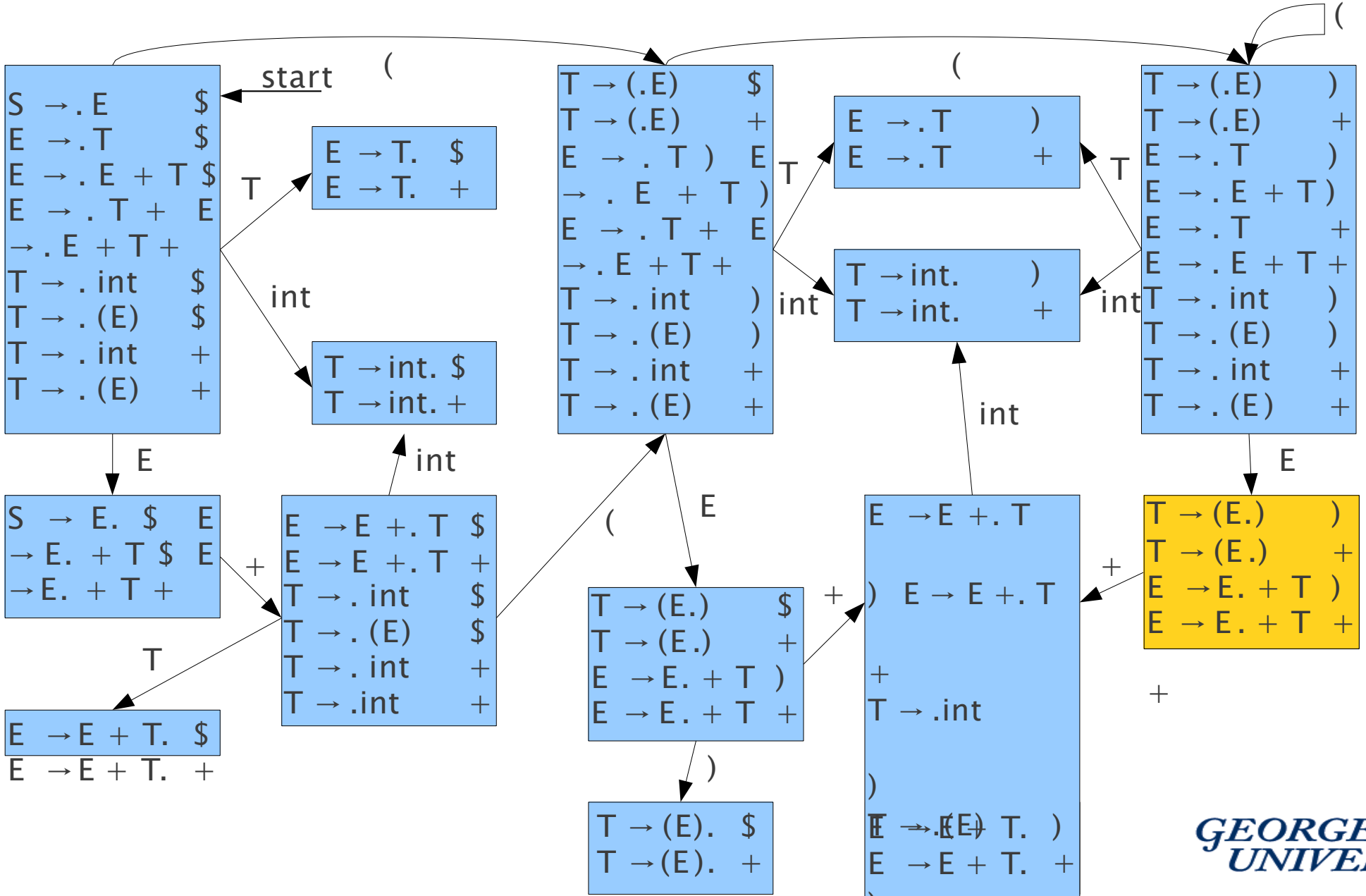
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



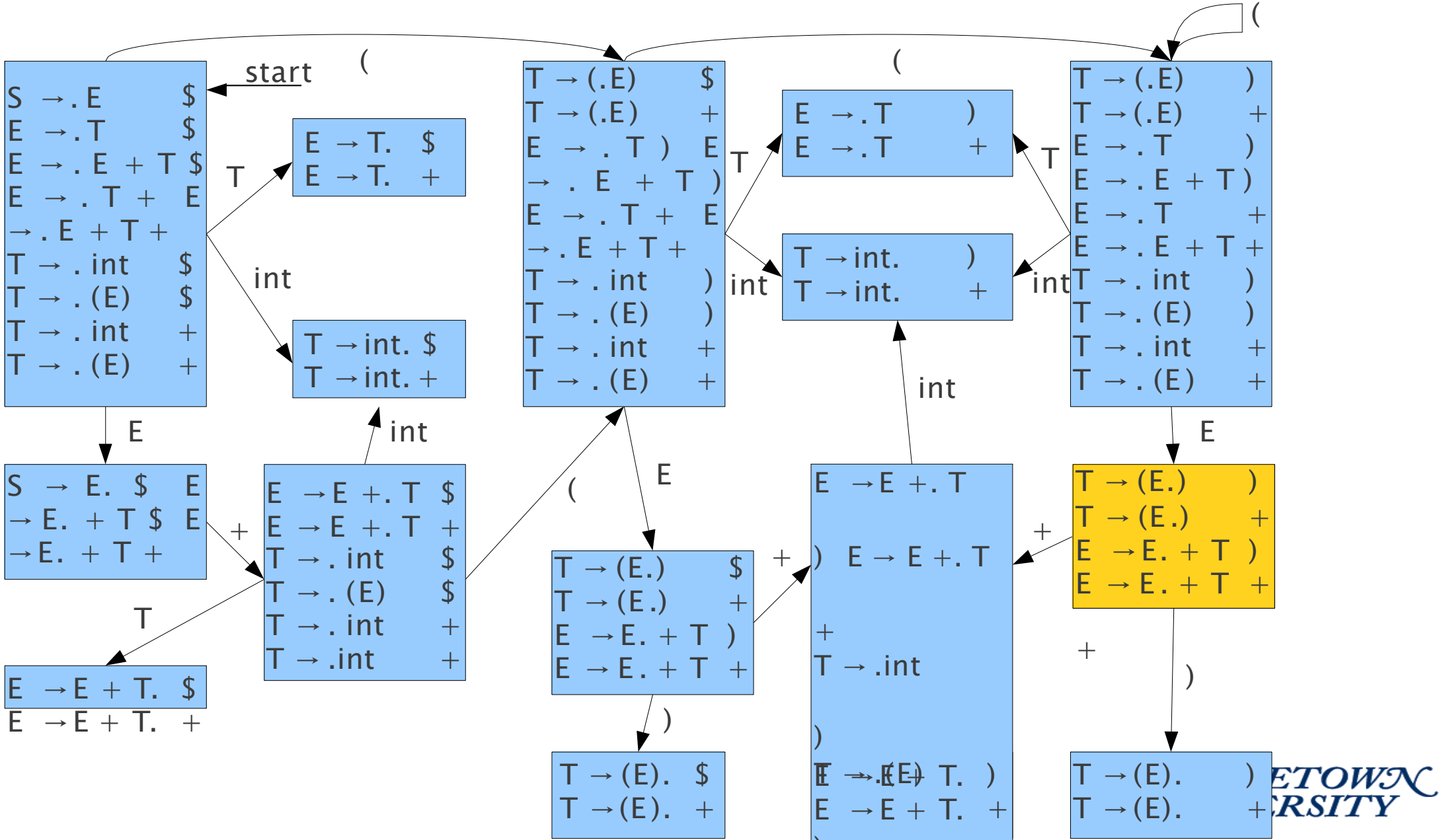
GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata

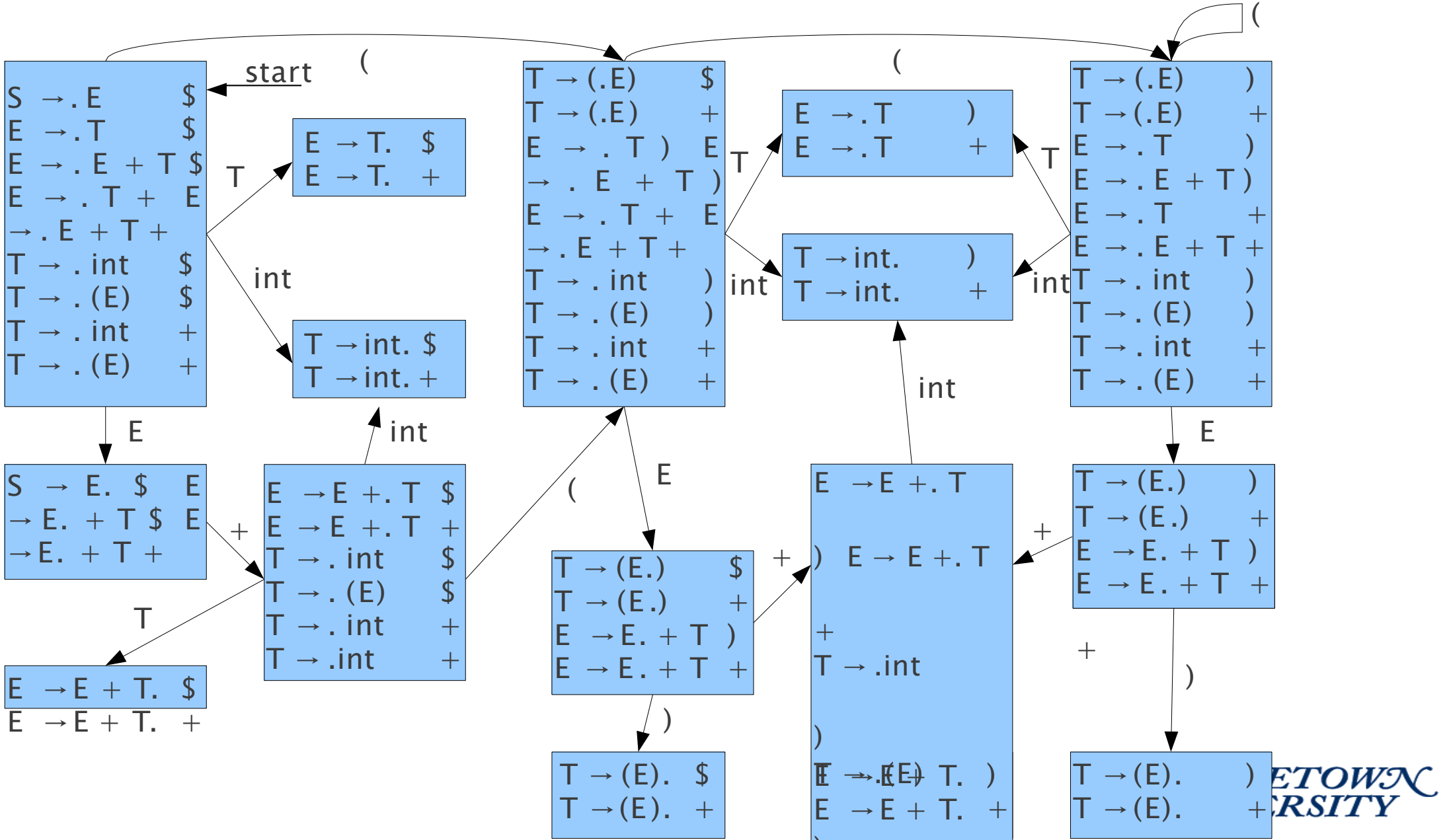


GEORGETOWN UNIVERSITY

Deterministic LR(1) Automata



Deterministic LR(1) Automata



Constructing LR(1) Automata II

- Begin in a state containing **S** → · **E** [**$**], where **S** is the start symbol.
- Compute the **closure** of the state:
  - If **A** → *α* · **B** *ω* [**t**] is in the state, add **B** → · *γ* [**t**] to the state for each production **B** → *γ* and for each terminal **t** ∈ FIRST*(*ω* **t**)
- Repeat until no new states are added:
  - If a state contains a production **A** → *α* · **x** *ω* [**t**], add a transition on **x** from that state to the state containing the closure of **A** → *α* **x** · *ω* [**t**].

Structure of LR(1) Automata

- Every LR(1) automaton simulates two processes simultaneously:
  - An **LR(0) automaton** for finding handles.
  - A **lookahead tracker** for determining what the lookahead is.
- Removing the lookaheads from an LR(1) automaton results in a (much larger) LR(0) automaton for the same grammar.