# Bottom Up Parsing

- More general than deterministic top-down parsing
  - Just as efficient
  - Uses the same ideas
- The method used by most compiler generation tools
- +ve:  do not need left factored grammar
- For example the following grammar is OK

$$E \rightarrow T+E \mid T$$
$$T \rightarrow int*T \mid int \mid (E)$$

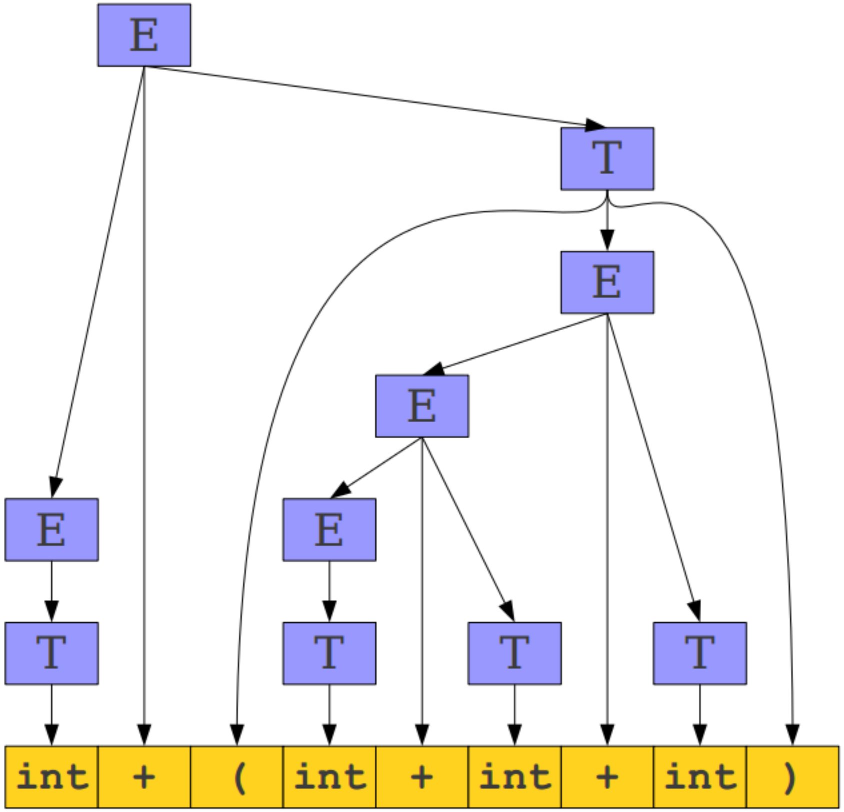
- A grammar is OK provided that it is unambiguous

# What is Bottom-Up Parsing?

- Idea: Apply productions in reverse to convert the user's program to the start symbol.
- A left-to-right, bottom-up parse is a rightmost derivation traced in reverse  (as we will see).

# One View of a Bottom-Up Parse

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



# A Second View of a Bottom-Up Parse

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

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

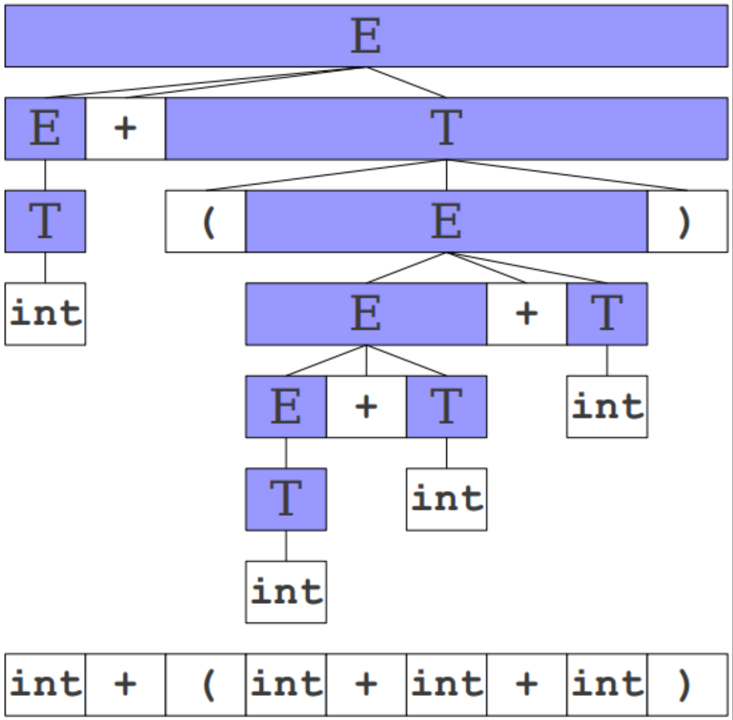
# A Third View of a Bottom-Up Parse

```
  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
```

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

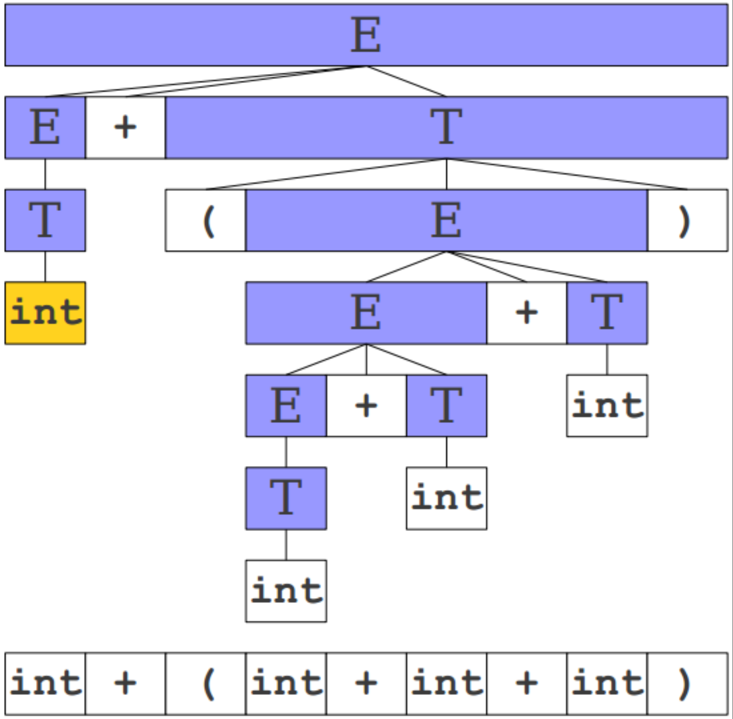
# A Third View of a Bottom-Up Parse

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



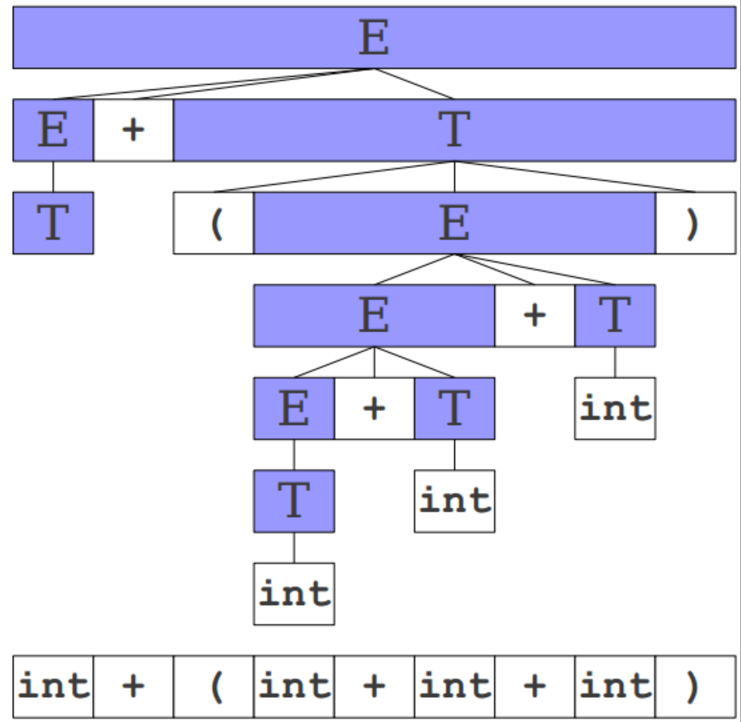
# A Third View of a Bottom-Up Parse

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



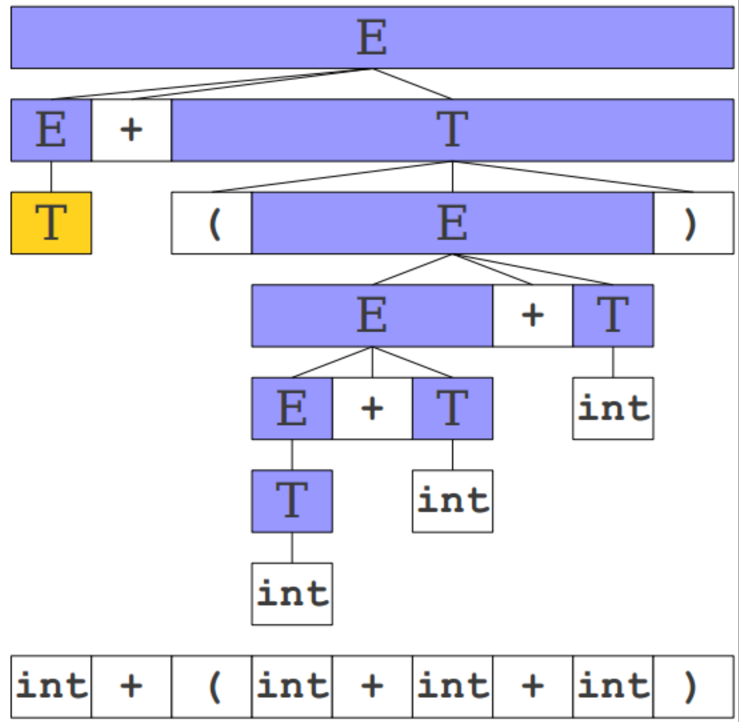
# A Third View of a Bottom-Up Parse

⇒ 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



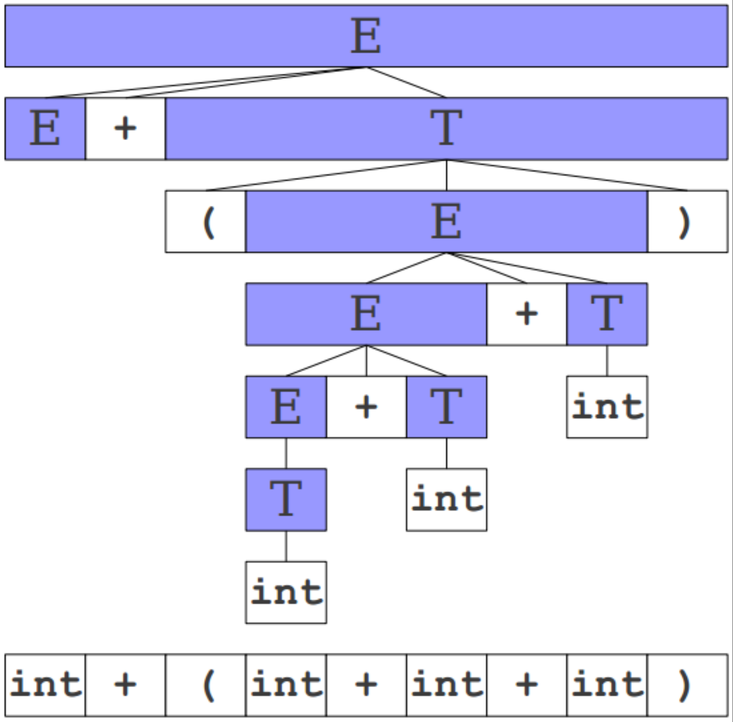
# A Third View of a Bottom-Up Parse

⇒ **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

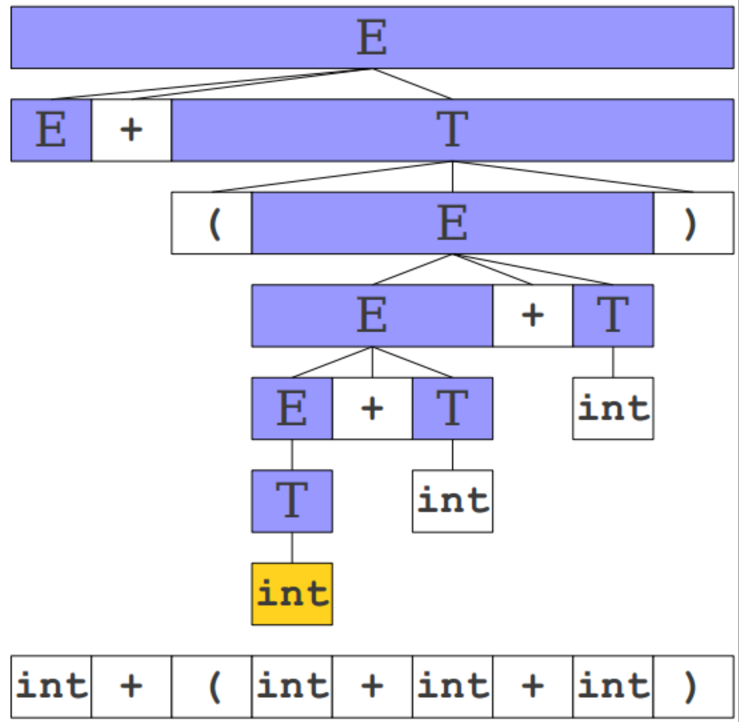
$\Rightarrow$ E + (int + int + int)
$\Rightarrow$ E + (T + int + int)
$\Rightarrow$ E + (E + int + int)
$\Rightarrow$ E + (E + T + int)
$\Rightarrow$ E + (E + int)
$\Rightarrow$ E + (E + T)
$\Rightarrow$ E + (E)
$\Rightarrow$ E + T
$\Rightarrow$ E




11.03 X 8.26 inch

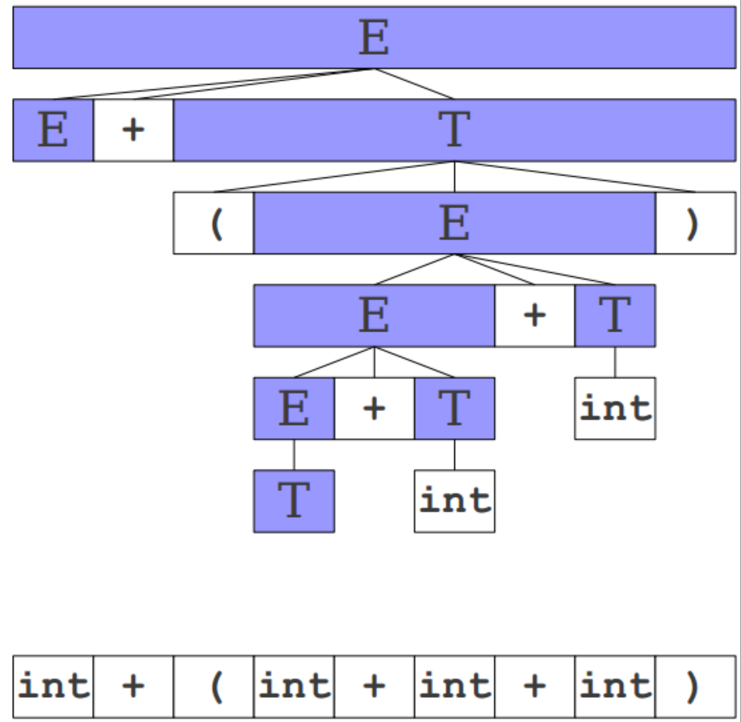
# A Third View of a Bottom-Up Parse

⇒ 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



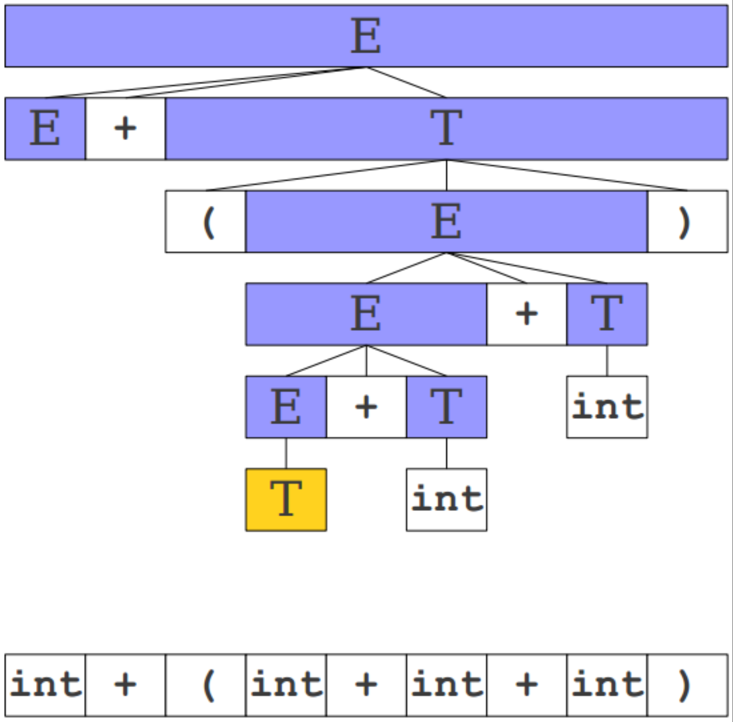
# A Third View of a Bottom-Up Parse

⇒ E + (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

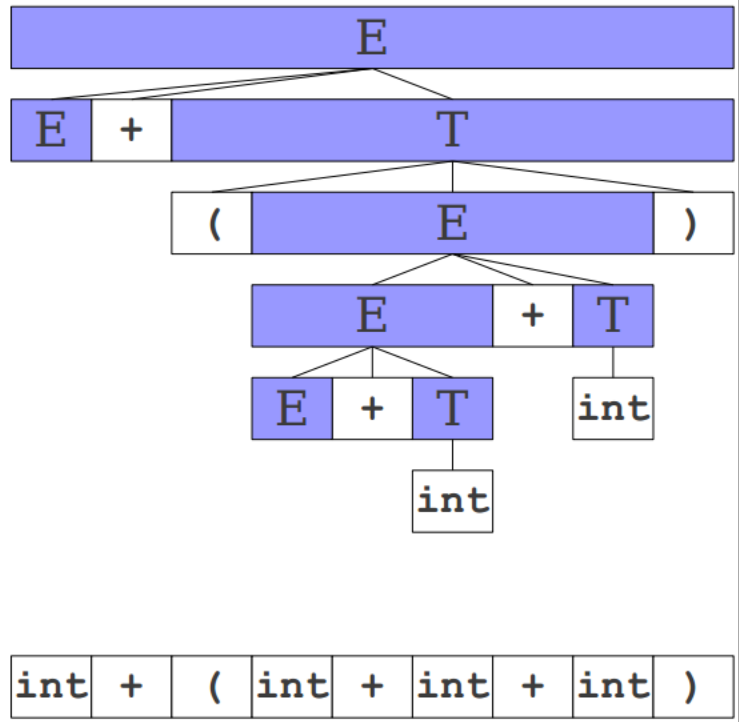
⇒ E + (**T** + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E




11.03 X 8.26 inch

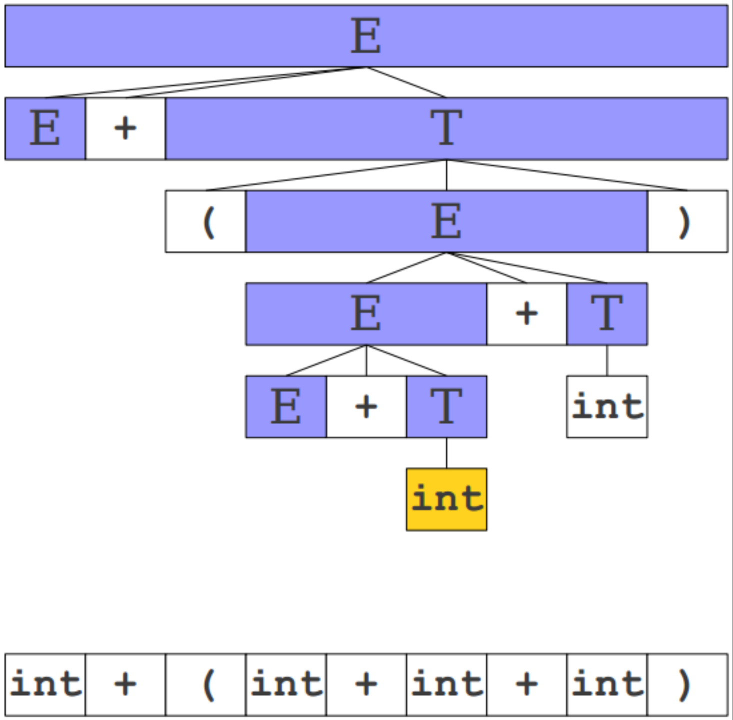
# A Third View of a Bottom-Up Parse

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



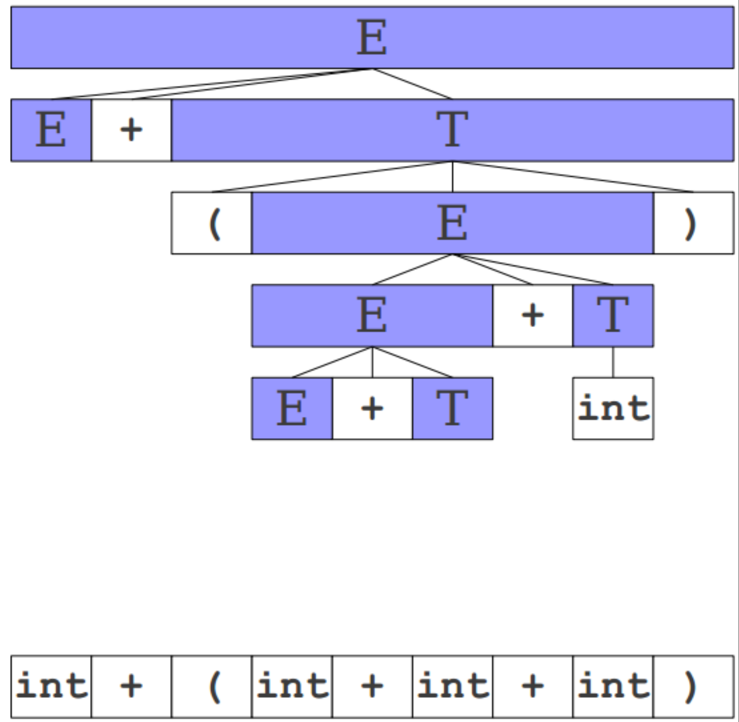
# A Third View of a Bottom-Up Parse

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

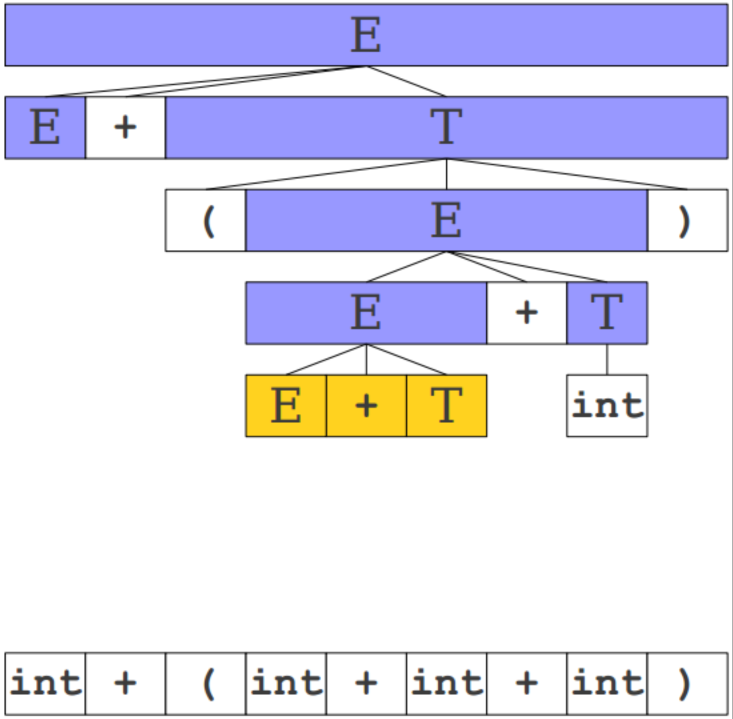


# A Third View of a Bottom-Up Parse

$\Rightarrow$ **E + (E + T + int)**
$\Rightarrow$ **E + (E + int)**
$\Rightarrow$ **E + (E + T)**
$\Rightarrow$ **E + (E)**
$\Rightarrow$ **E + T**
$\Rightarrow$ **E**



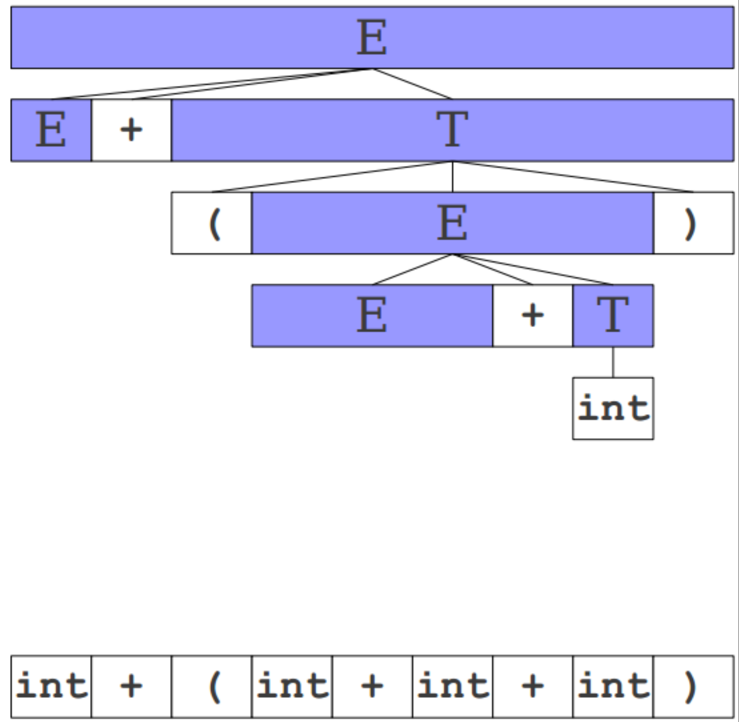
# A Third View of a Bottom-Up Parse



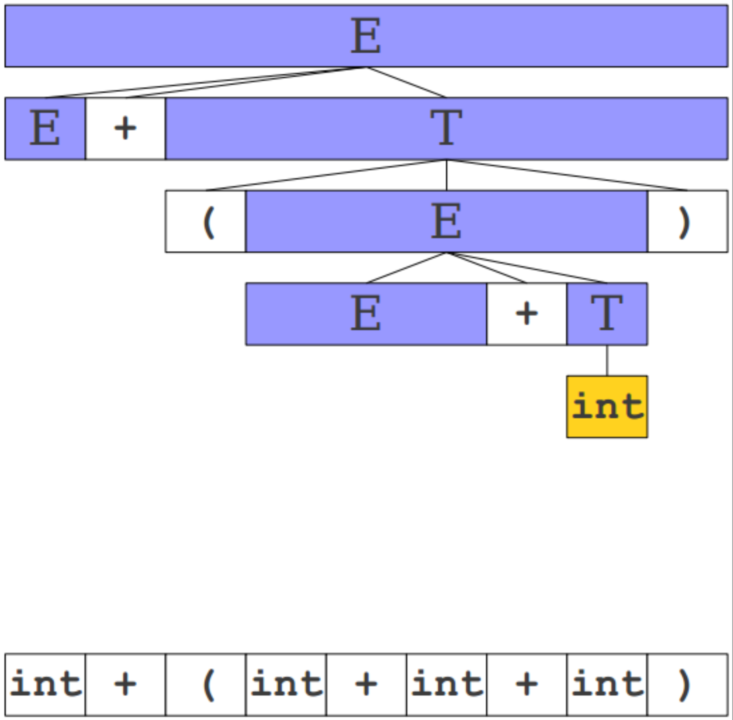

⇒ **E + (E + T + int)**
⇒ **E + (E + int)**
⇒ **E + (E + T)**
⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

# A Third View of a Bottom-Up Parse

$\Rightarrow$ **E + (E + int)**
$\Rightarrow$ **E + (E + T)**
$\Rightarrow$ **E + (E)**
$\Rightarrow$ **E + T**
$\Rightarrow$ **E**

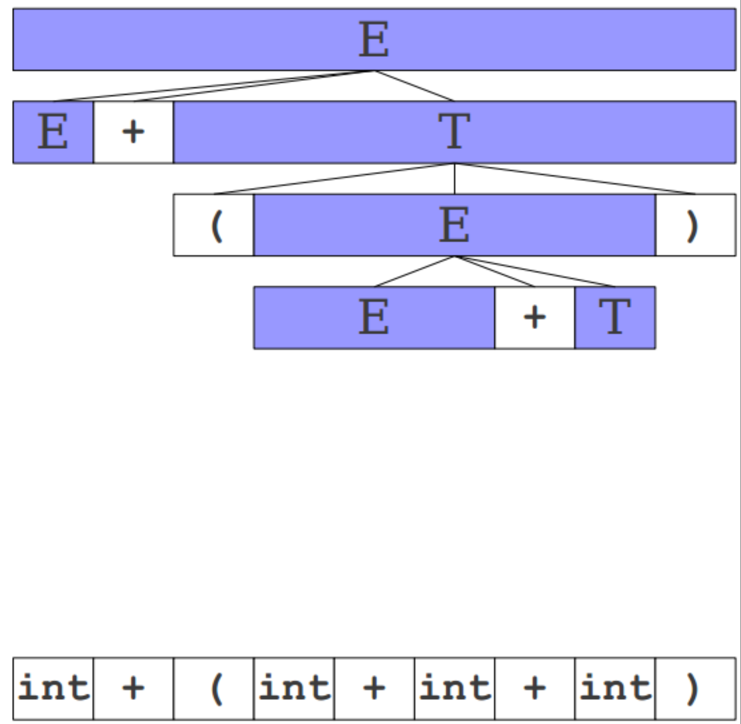


# A Third View of a Bottom-Up Parse



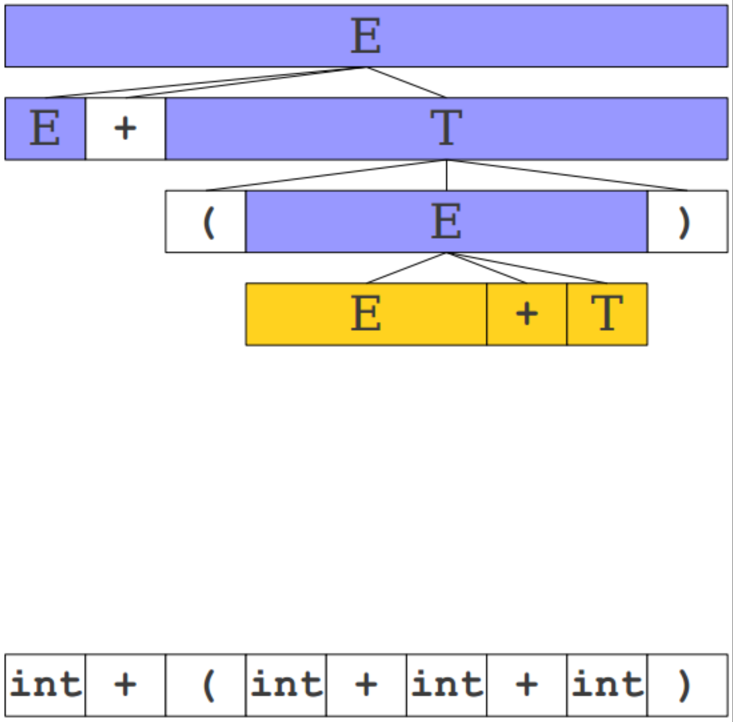

⇒ **E + (E + int)**
⇒ **E + (E + T)**
⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

# A Third View of a Bottom-Up Parse




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

# A Third View of a Bottom-Up Parse



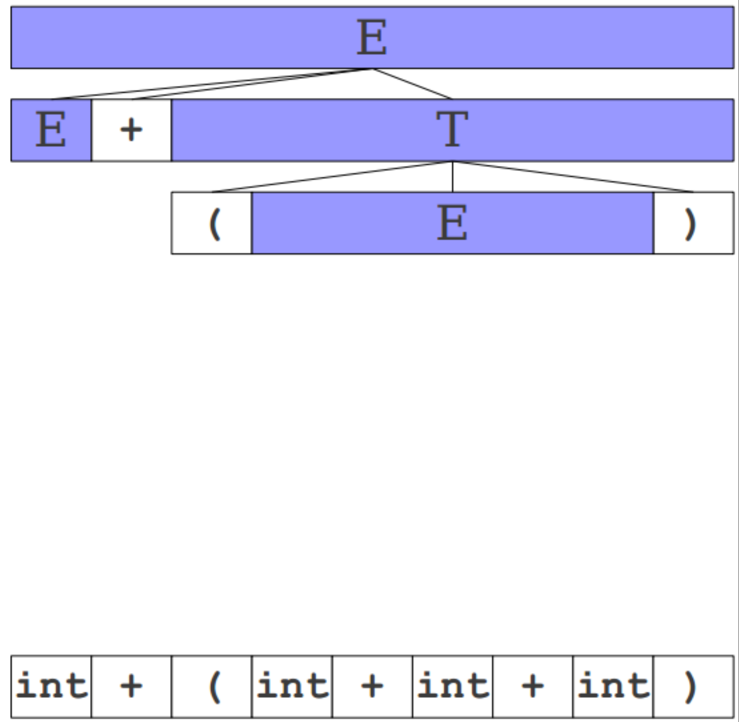

$\Rightarrow$ **E + (E + T)**

$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

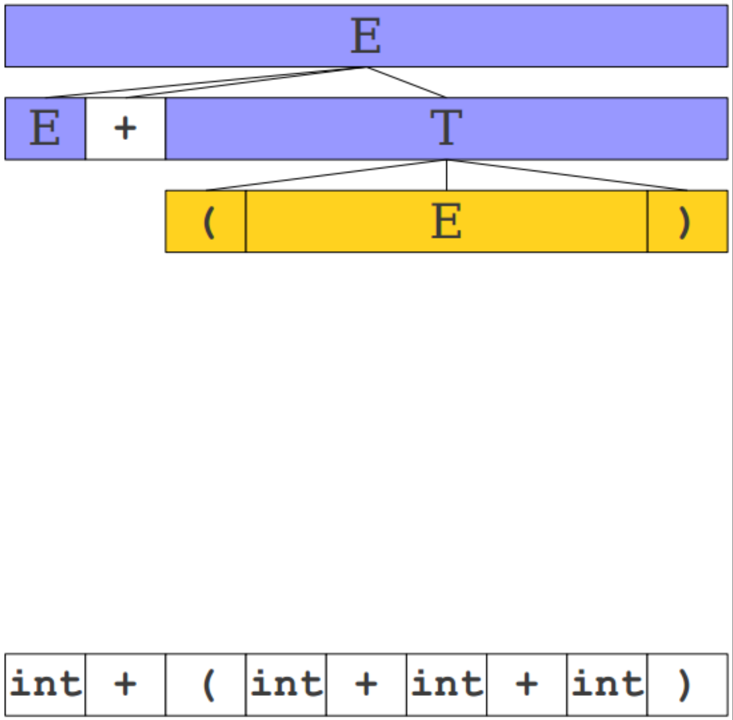
$\Rightarrow$ **E**

# A Third View of a Bottom-Up Parse




$\Rightarrow$ **E + (E)**
$\Rightarrow$ **E + T**
$\Rightarrow$ **E**

# A Third View of a Bottom-Up Parse

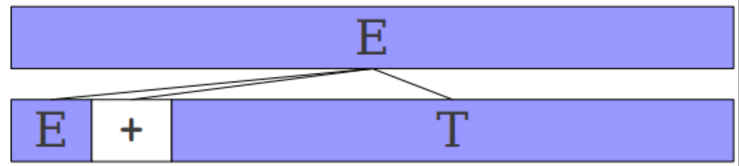



$\Rightarrow$ **E + (E)**

$\Rightarrow$ **E + T**

$\Rightarrow$ **E**

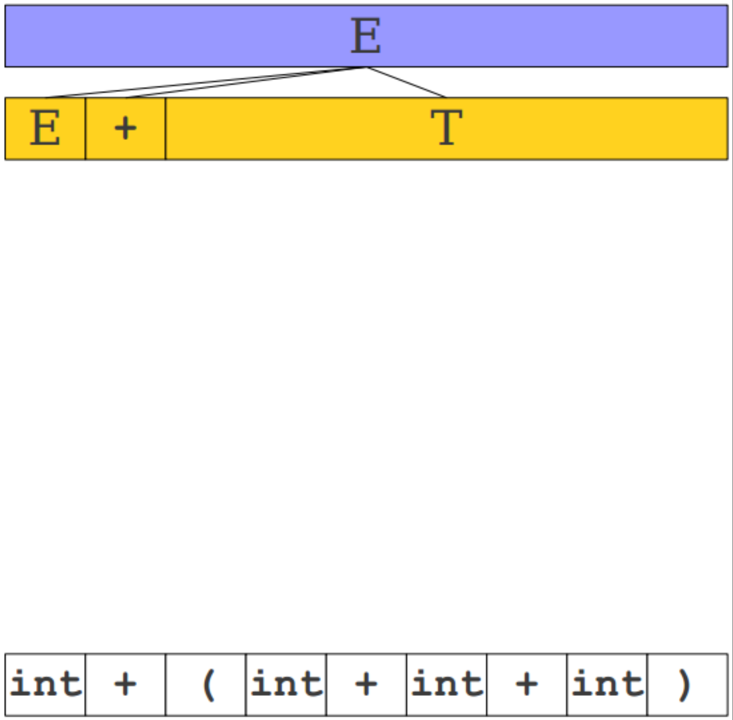
# A Third View of a Bottom-Up Parse




⇒ **E + T**
⇒ **E**

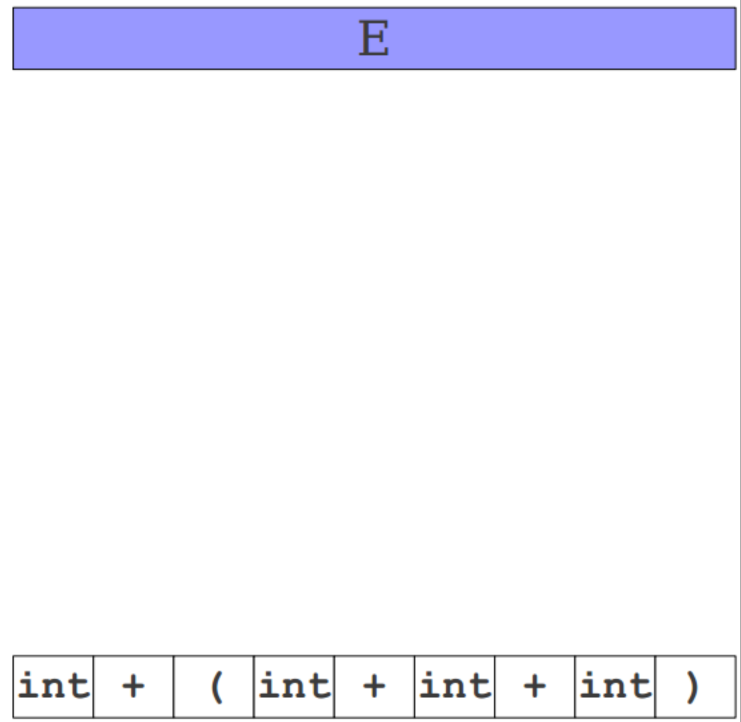
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| int | + | ( | int | + | int | + | int | ) |

# A Third View of a Bottom-Up Parse




⇒ **E + T**
⇒ **E**

# A Third View of a Bottom-Up Parse




⇒ **E**

## Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

| | |
|---|---|
| int * int + int | $T \rightarrow$ int |
| int * T + int | $T \rightarrow$ int * T |
| T + int | $T \rightarrow$ int |
| T + T | $E \rightarrow T$ |
| T + E | $E \rightarrow T + E$ |
| E | |

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

# A Bottom-up Parse in Detail (1)

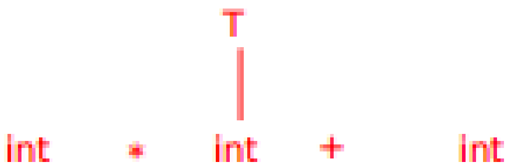
int * int + int

int * int + int

# A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

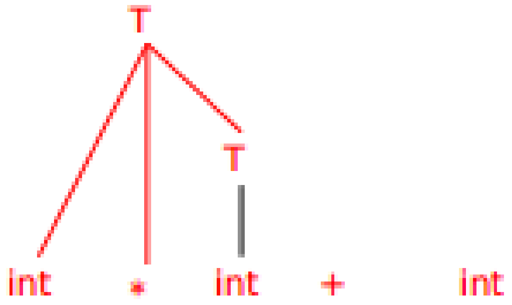


# A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



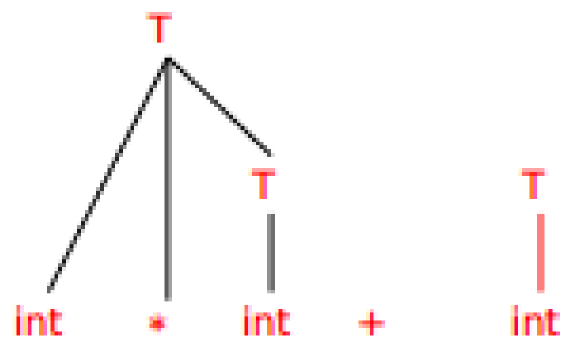
# A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



# A Bottom-up Parse in Detail (5)

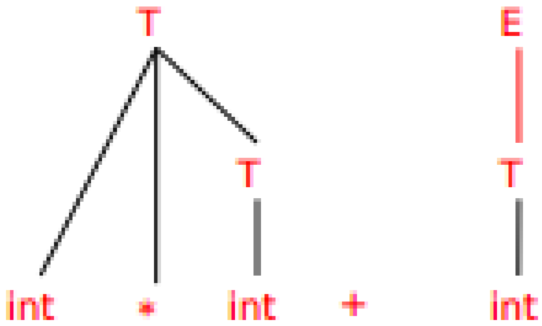
int * int + int

int * T + int

T + int

T + T

T + E



## A Bottom-up Parse in Detail (6)

int * int + int

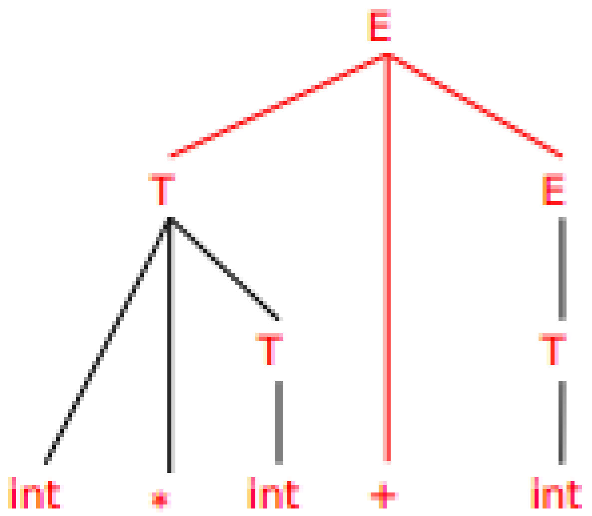
int * T + int

T + int

T + T

T + E

E



# Shift Reduce Parsing

- The main strategy used by bottom up parsers
- Recall that a bottom-up parser traces a rightmost derivation in reverse
- An important consequence
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \rightarrow \beta$
  - Then $\omega$ is a string of terminals, otherwise the reduction we just did was not for the rightmost terminal.
- The general idea: split string into two substrings
  - Right substring is yet unexamined by parsing
  - The Left substring has terminals and non-terminals
  - The left substring is our work area (where we should search for handles)
  - The dividing point is marked by a |

# Two Main kinds of actions

1. Shift: Move | one place to the right
    - i.e., shifts a terminal to the left string
    - e.g. ABC|xyz➔ ABCx|yz

2. Reduce: apply an inverse production at the right end of the left string
    - If A→ xy is a production, then
      Cbxy | ijk ➔ CbA| ijk

- When to shift and when to reduce is another story

## The Example with Shift-Reduce Parsing

<table>
<tr><td>|int * int + int</td><td>shift</td></tr>
<tr><td>int | * int + int</td><td>shift</td></tr>
<tr><td>int * | int + int</td><td>shift</td></tr>
<tr><td>int * int | + int</td><td>reduce T → int</td></tr>
<tr><td>int * T | + int</td><td>reduce T → int * T</td></tr>
<tr><td>T | + int</td><td>shift</td></tr>
<tr><td>T + | int</td><td>shift</td></tr>
<tr><td>T + int |</td><td>reduce T → int</td></tr>
<tr><td>T + T |</td><td>reduce E → T</td></tr>
<tr><td>T + E |</td><td>reduce E → T + E</td></tr>
<tr><td>E |</td><td></td></tr>
</table>

# Bottom-Up Parsing Using a Stack

- Left string can be implemented by a stack
  - Shift
    - pushes a terminal on the stack
  - Reduce
    - Pop symbols off of the stack (the right hand side of a production)
    - Pushes a non-terminal on the stack (the left hand side of a production)

# Conflicts

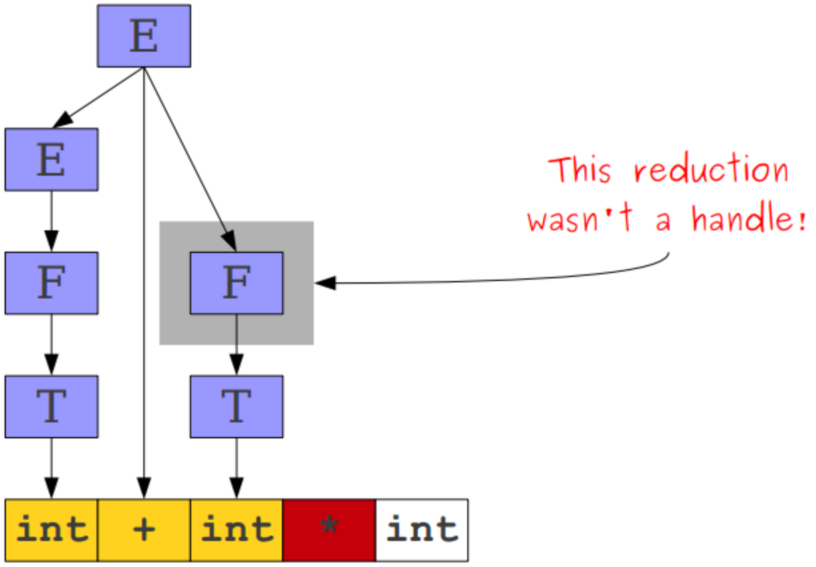
- In a given state, more than one action (shift or reduce) may lead to a parse tree
- Two main kinds of conflicts:
  - If it is legal to shift or reduce, there is a shift-reduce conflict
    - Not very good, but it is easy to rewrite the grammar to remove it.
  - If it is legal to reduce by two different productions, there is a reduce-reduce conflict.
    - This is bad because it indicates that something is wrong with the grammar

# Handles

- How do we decide when to shift or reduce?
- The leftmost reduction is not always the best thing to do
- 2 Examples

# A Detail about Handles

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



# Another Example

- Example grammar:
  - E→ T+E | T
  - T → int * T | int | (E)
  - Consider step int | * int  + int
    - We could reduce by T → int giving T| *int+int
    - A fatal mistake
      - Since no production can handle T*
      - There would be no way to reduce to the start symbol E

# Handles

- A handle is a reduction that **allows further reductions back to the start symbol**
- $S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$,     then $\alpha\beta$ is the handle simply because it is **not a mistake it allowed us to go back to the start symbol**.
- The handle of a parse tree T is **the leftmost complete cluster of leaf nodes.**

- A left-to-right, bottom-up parse works by
  - iteratively searching for a handle, then
  - reducing 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?
  - What algorithm do we use to try to discover a handle?
- **How do we recognize handles**?
  - Once we've found a candidate handle, how do we confirm that it is  correct (i.e., the handle?)

# Recognizing Handles

- There are **no known efficient algorithms** to recognize handles

- But, there are **good heuristics for guessing handles**
- **On some CFGs, the heuristics always guess correctly**

- It is **not obvious how to detect** handles
- At each step the parser **sees only the stack**, not the entire input;
- It sees α where **α is a viable prefix** if there is an ω such that **α| ω is a state of a shift-reduce** parser.
- Recall that α is on the stack while ω is the unseen input.

# Viable Prefix

- A viable prefix because is a **prefix of the handle**
- In other words: it **does not extend past the right end** of the handle
- As long as a parser has viable prefixes on the stack **no parsing error has been detected**.

# Important Fact

- For any grammar, **the set of viable prefixes is a regular language**.
- Therefore they **can be recognized by a finite automata**
- The basis for many **compiler generation tools**.
- We will see **how to construct such a FA**.
- But first we need **a few more definitions**

# An item

- An item is **a production with a "." somewhere on the rhs.**
- The items for T → (E) are

  T → .(E)

  T → (.E)

  T → (E.)

  T → (E).
- The only item for X → ξ is X → .
- **Items are often called "LR(0) items"**

# The problem in recognizing viable prefixes

- is that the stack has only **bits and pieces** of the rhs of productions
  - If it had a complete rhs, we could reduce
- In any successful parse **theses bits and pieces are always prefixes of rhs of productions**.

- Consider the input (int)

  $E \rightarrow T + E \mid T$

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

  – Then (E|) is a **valid state of a shift-reduce** parse
  – (E is a prefix of the rhs of T → (E)
    - Notice that it will be reduced after the next shift
  – Item T → (E.) says that so far we have seen (E of this production and hope to see )
    - i.e. **no parsing errors so far**

# The structure of the stack

- The stack does not contain an arbitrary string of symbols.
- The stack may have many prefixes or rhs's
- $Prefix_1$ $Prefix_2$ … $Prefix_{n-1}$ $Prefix_n$

- Let $Prefix_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - $Prefix_i$ will eventually reduce to $X_i$
  - The missing part of $\alpha_{i-1}$ starts with $X_i$
  - i.e., there is an $X_{i-1} \rightarrow Prefix_{i-1}\ X_i\ \beta$ for some $\beta$
  - Recursively, $Prefix_{k+1} \ldots Prefix_n$ eventually reduces to the missing part of $\alpha_k$

# An Example

- Consider the grammar

  $E \rightarrow T + E \mid T$

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

- And the string (int * int)
- (int * | int) is a state of a shift-reduce parse
- The stack contents from bottom-to-top is

  “(”  which is a prefix of the rhs of $T \rightarrow (E)$

  “ξ”  which is prefix of the rhs of $E \rightarrow T$

  “int *”  which is a prefix of the rhs of $T \rightarrow \text{int} * T$

- The stack of items

  $T \rightarrow (.E)$

  $E \rightarrow .T$

  $T \rightarrow int * .T$

- Which says
  - We have seen "(" of $T \rightarrow (E)$
  - We have seen $\xi$ of $E \rightarrow T$
  - We have seen int* of $T \rightarrow int * T$

- **To recognize viable prefixes, we must**
  - **Recognize a sequence** of partial rhs's of productions, where
  - Each partial rhs can eventually **reduce to part of the missing suffix of its predecessor**

# An Algorithm for recognizing Viable prefixes

- Recall that the set of **viable prefixes are regular**, so what we are going to do is to **construct a NFA that recognizes** them.
- The input of the NFA is the stack.
  - **It will be read bottom-up**
- The **output is**
  - **yes if it is a viable prefix and**
  - **no if it is not**.
- **The states of the NFA are the items of the grammar.**

# Algorithm:

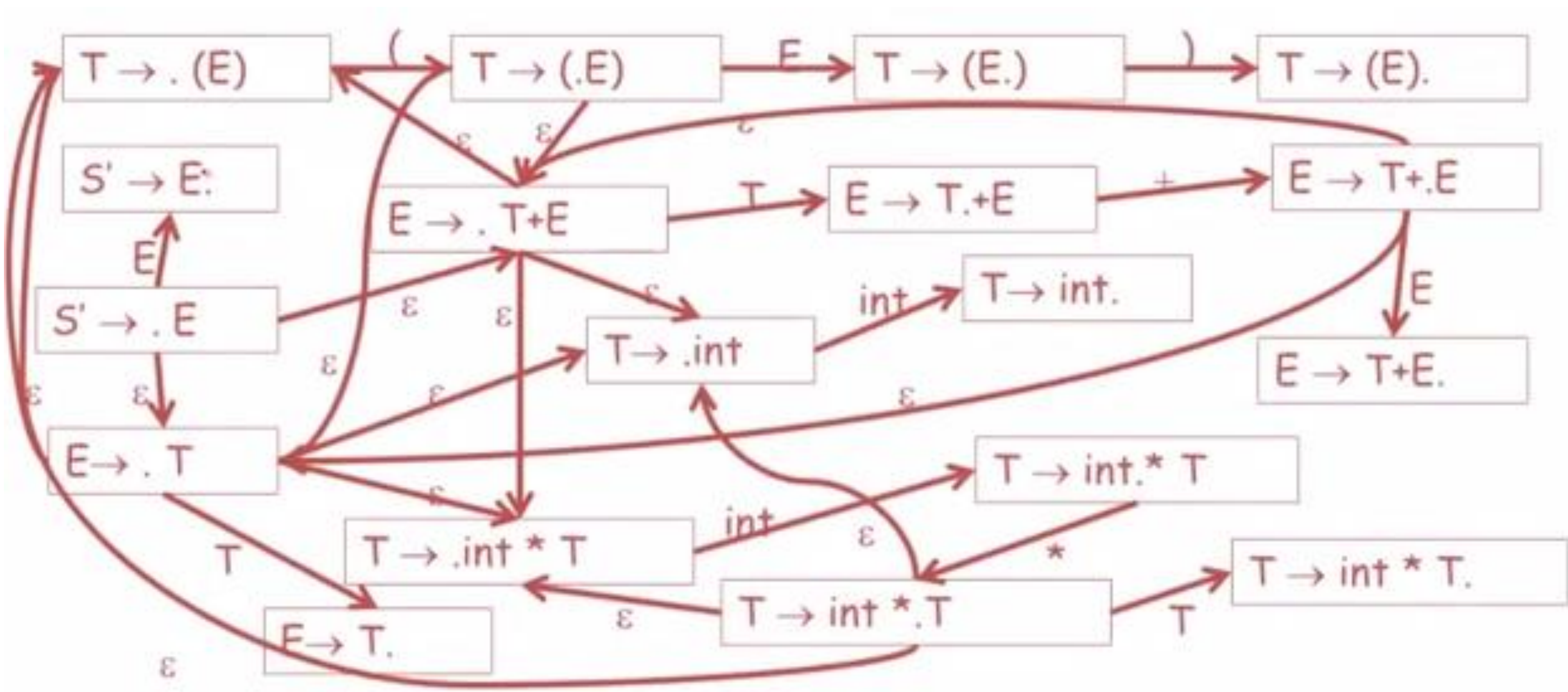
1. Add a dummy production S’ → S to G
   - this makes S’ the new start symbol and
   - makes sure that there is one production for the new start symbol
2. The NFA **states are the items of G**
   - Including the extra production
3. For item E → α.Xβ add transition
   - (i.e. so far we have seen α on the stack)
   - So if x is the next symbol on the stack (above α) then we can make this transition
   - $E \rightarrow \alpha.X\beta \xrightarrow{x} E \rightarrow \alpha X.\beta$ where X a terminal or non terminal (i.e., a move the NFA can make)
4. For item E → α.Xβ and for every production X →ϒ
   - where X is a non-terminal
   - and what is on the stack can eventually be reduced to x
   - So we can make the transition
   - $E \rightarrow \alpha.X\beta \xrightarrow{\xi} X \rightarrow .\Upsilon$
5. Every state is an accepting state
6. Start State is S’ → .S

# Example

S’ → E (the extra production)

E → T + E | T

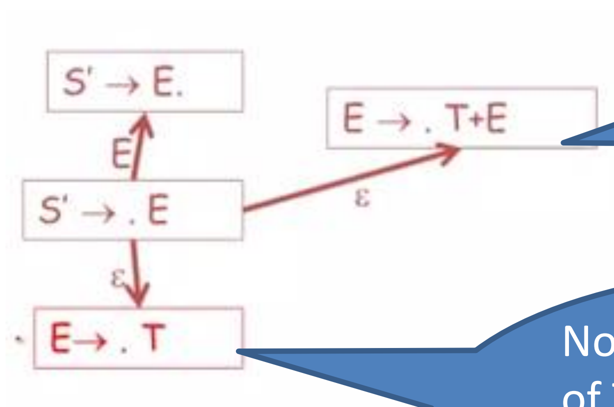
T → int*T | int | (E)



The start state is the extra production


What transitions can we make depends on what can be on the stack:

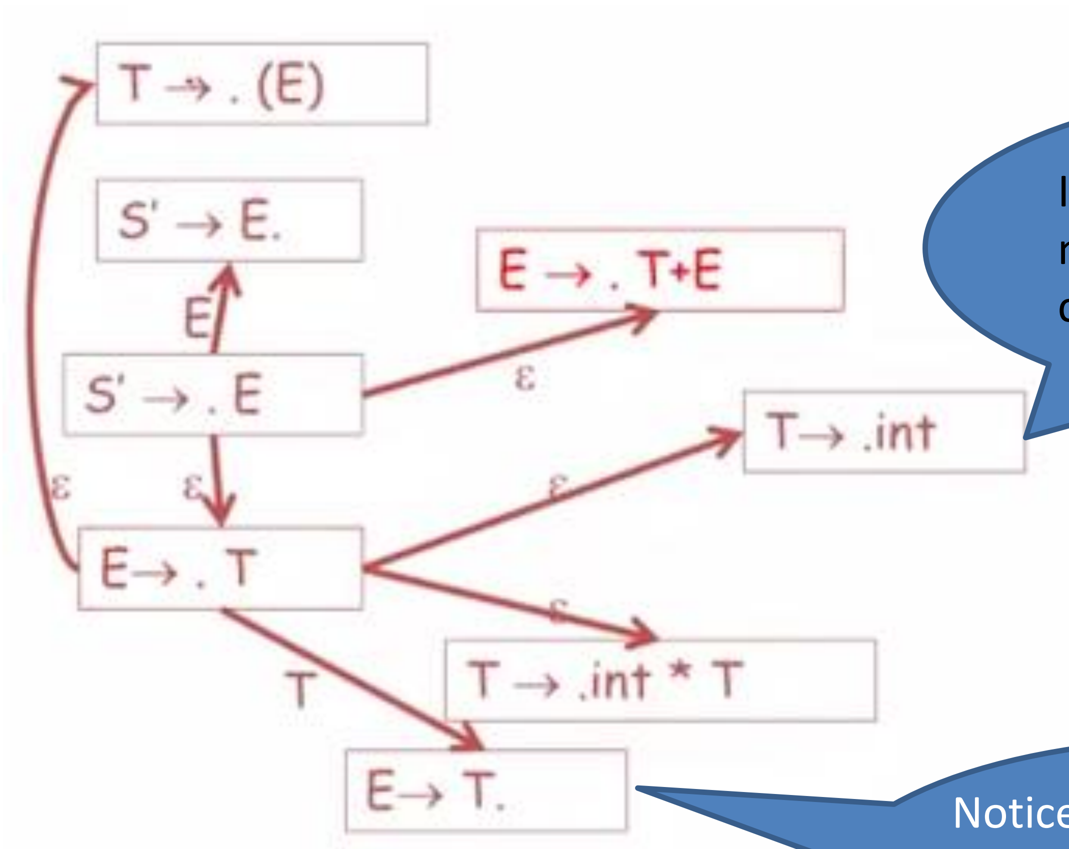
There can be an E on the stack or something derived from E so we need three transitions




Notice the dot on the left of T, indicating that we are hoping to see T on the stack next.

Notice the dot on the left of T, indicating that we are hoping to see T on the stack next.

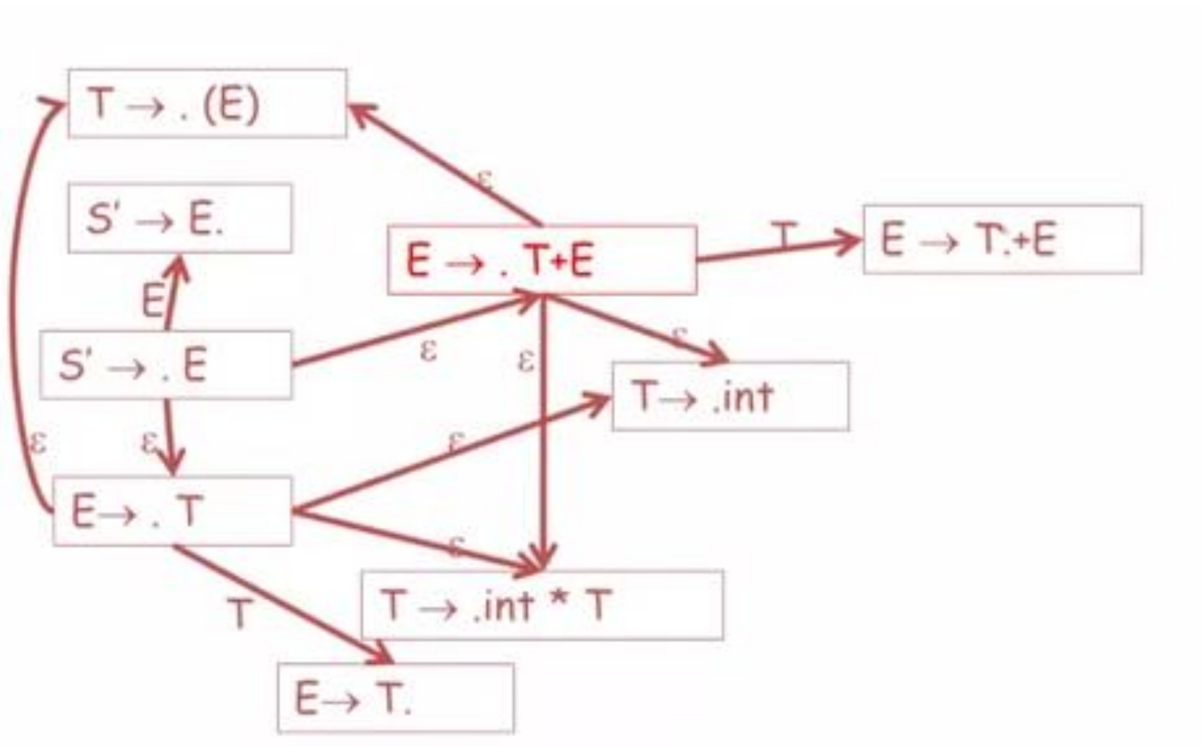
# The transitions for E→.T




If we do not see a T, we may see something that can be derived from T

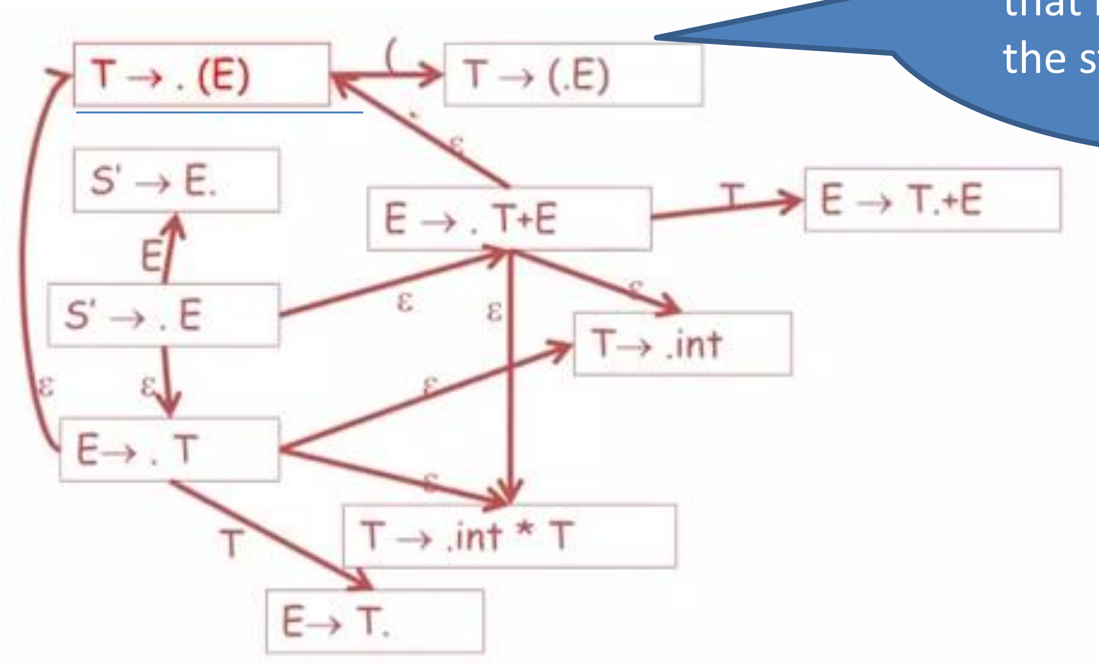
Notice the dot on the right indicates that we are ready to perform a reduction.

# The transitions for E → .T+E



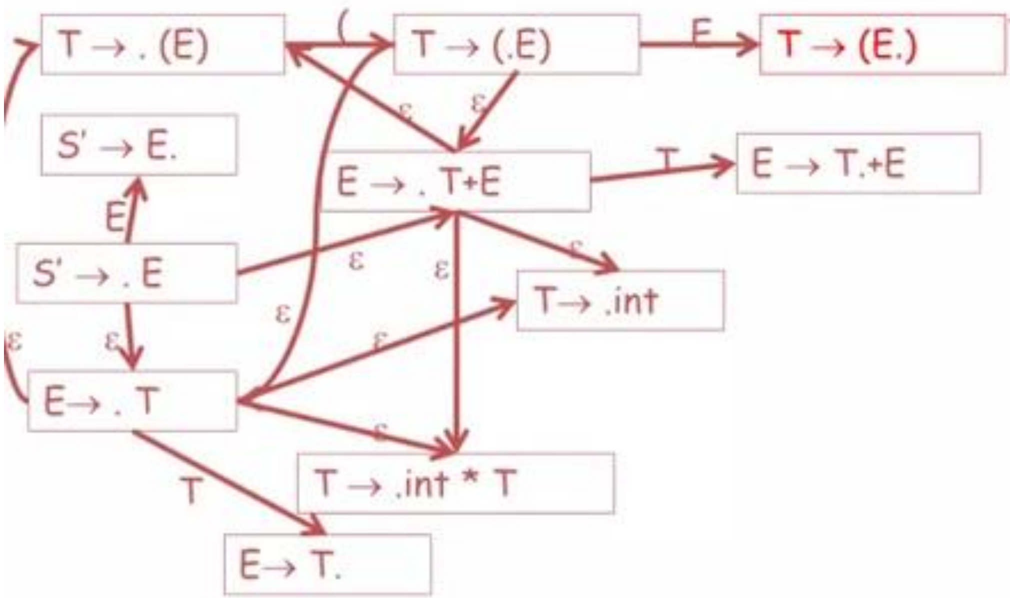
The transitions for T → .(E):

One possible transition that is if we see “(“ on the stack.

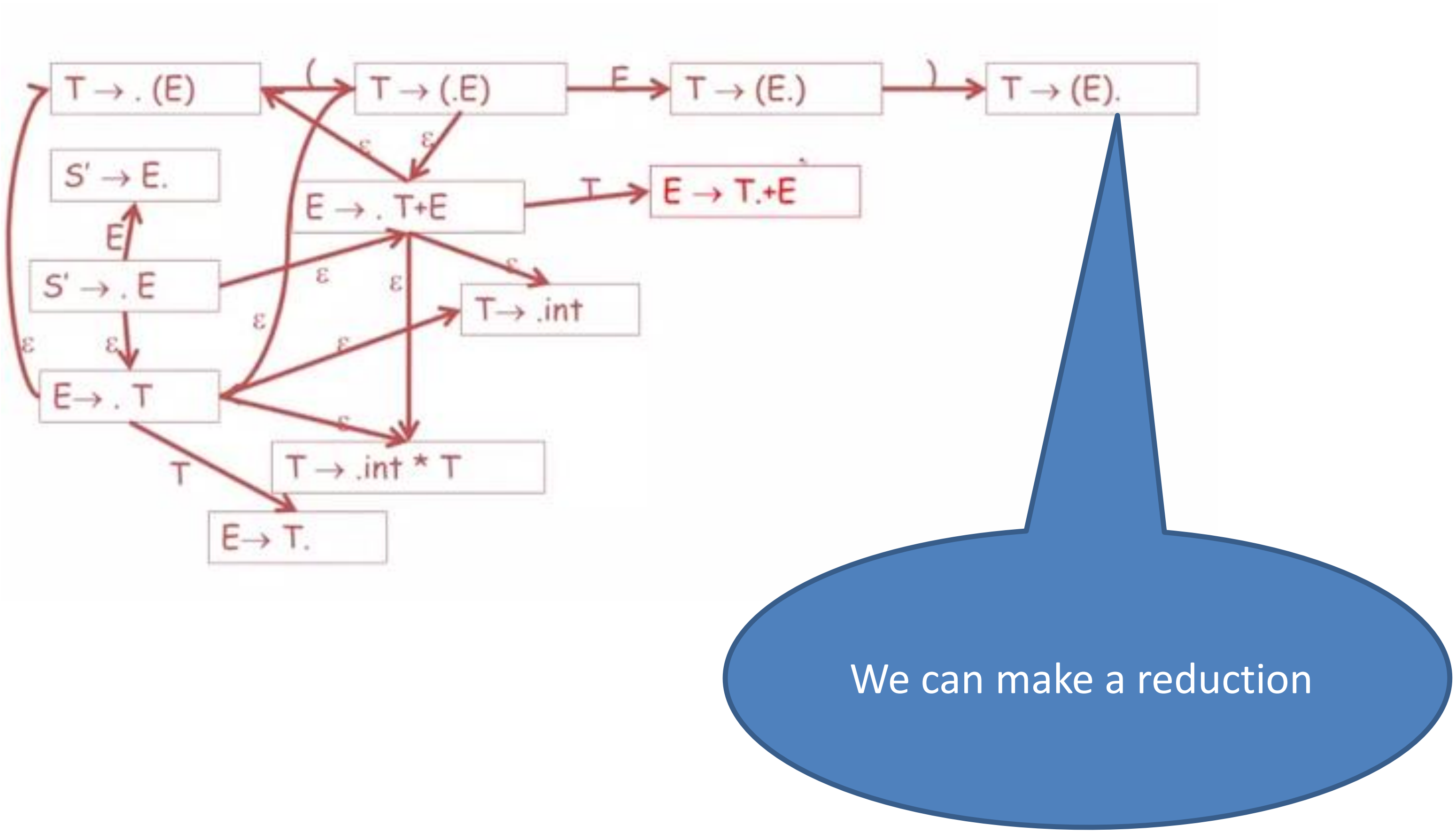



One possible transition that is if we see “(“ on the stack.

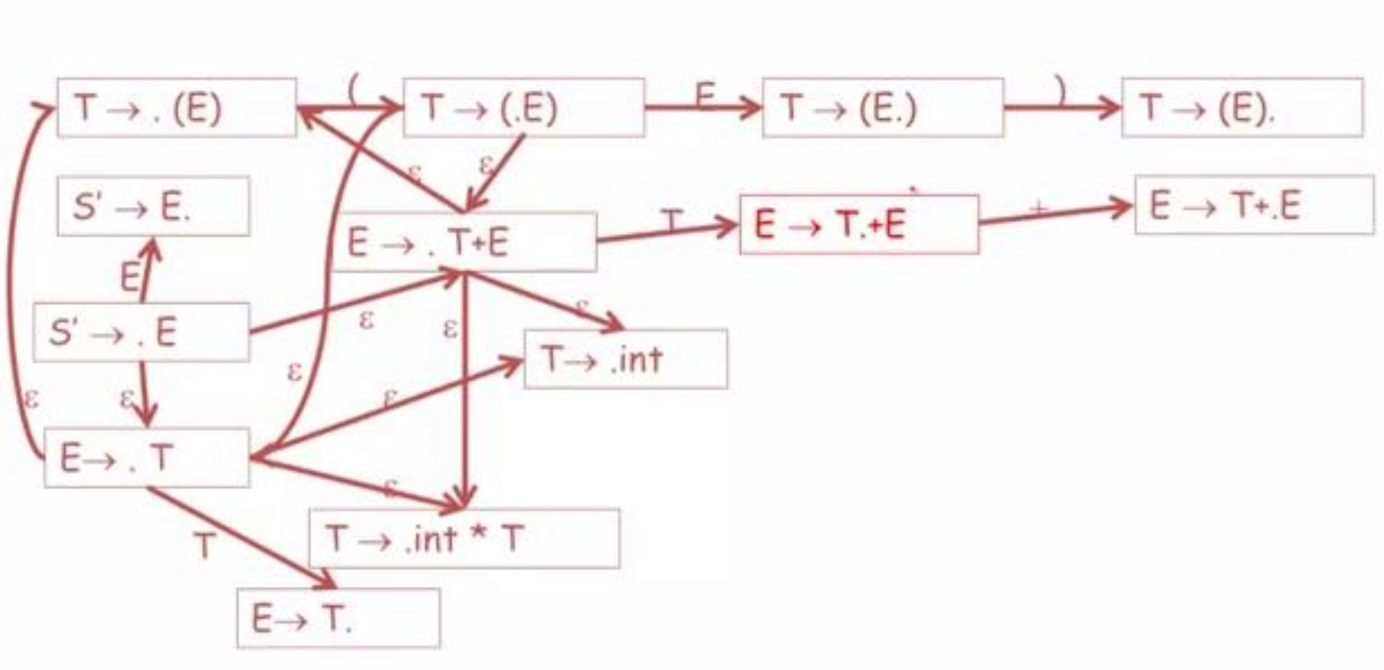
# The transition for T → (.E)



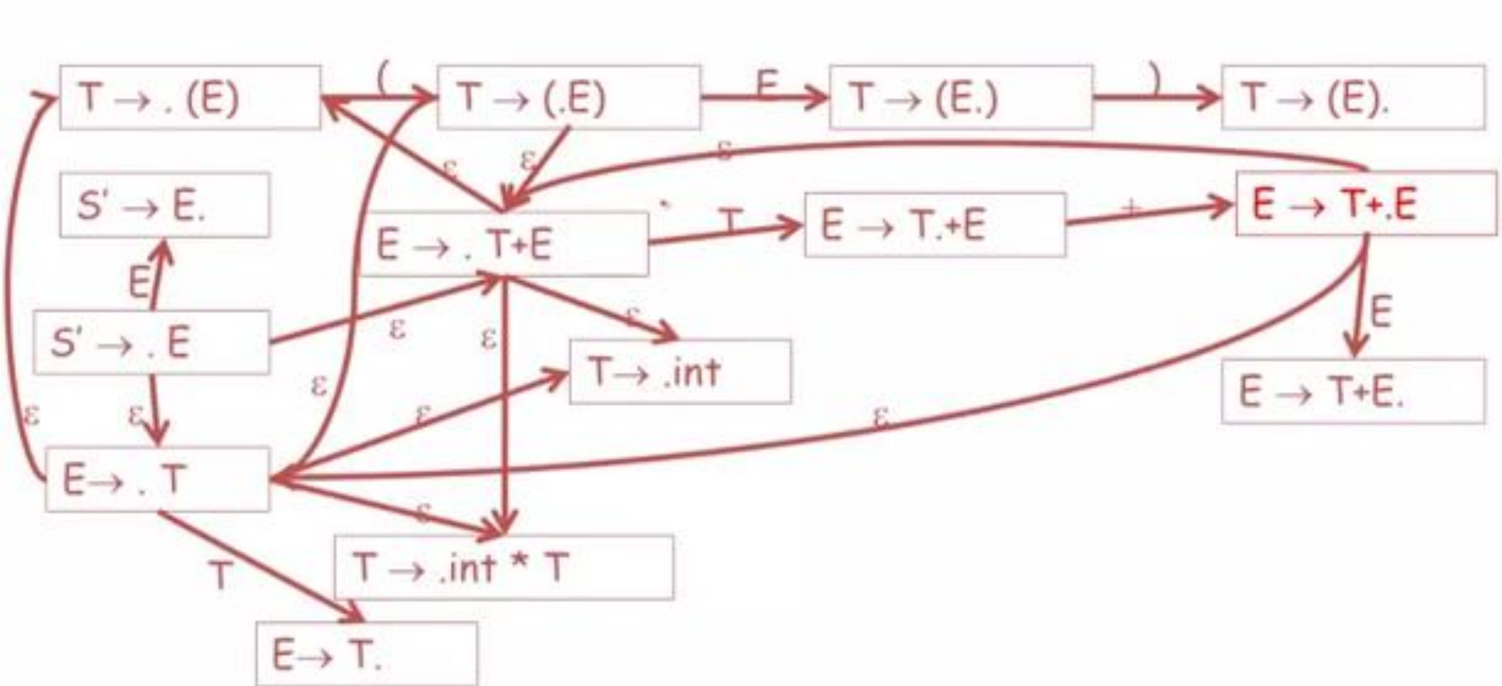
# The transition of T → (E.)



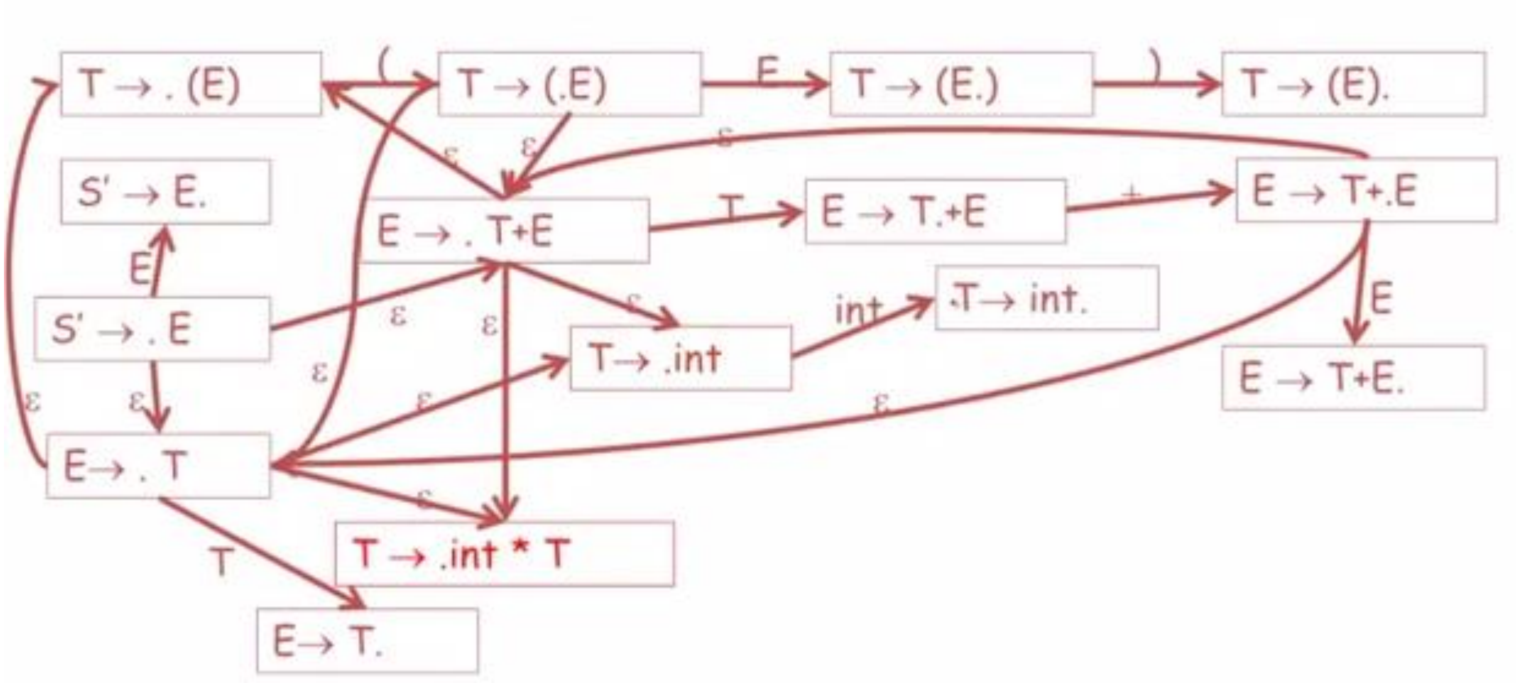
# The transition for E → T.+E



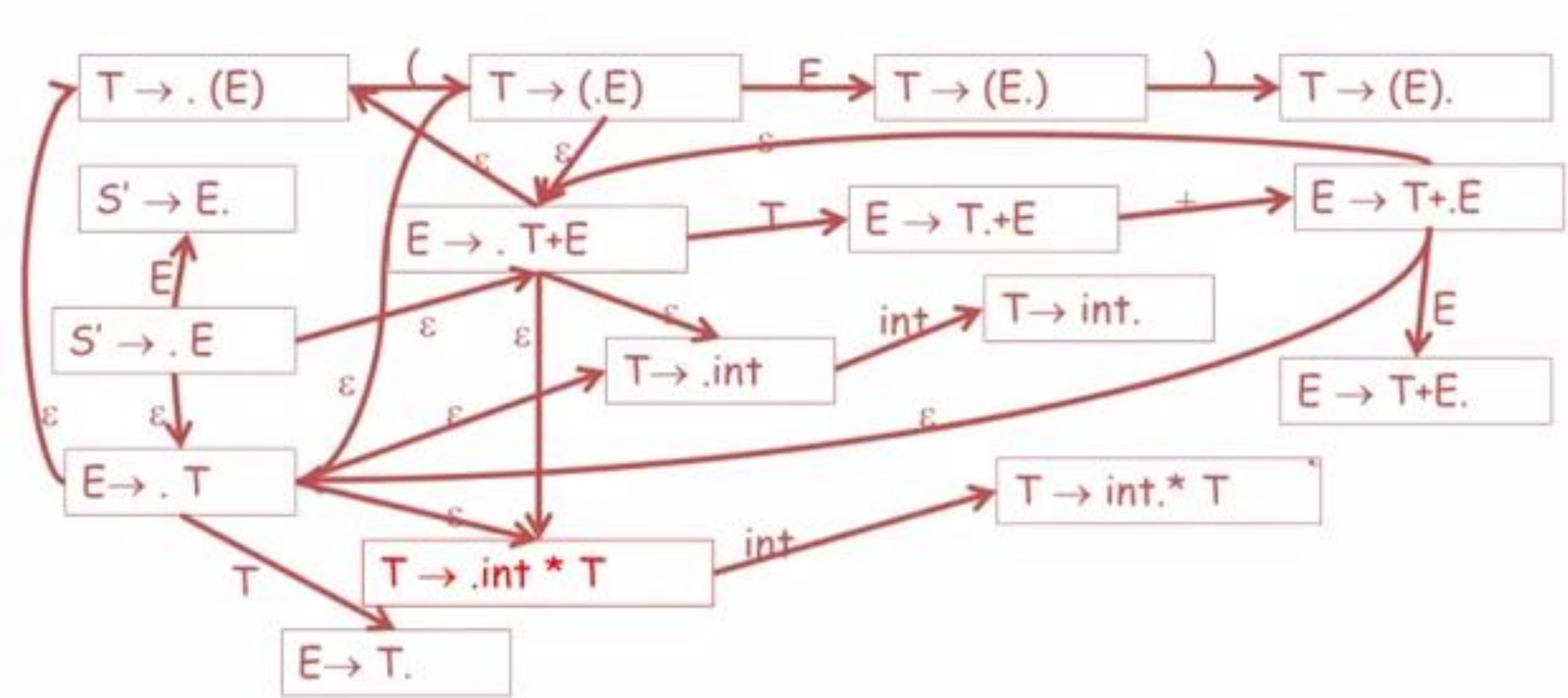
# Transitions for E → T+.E



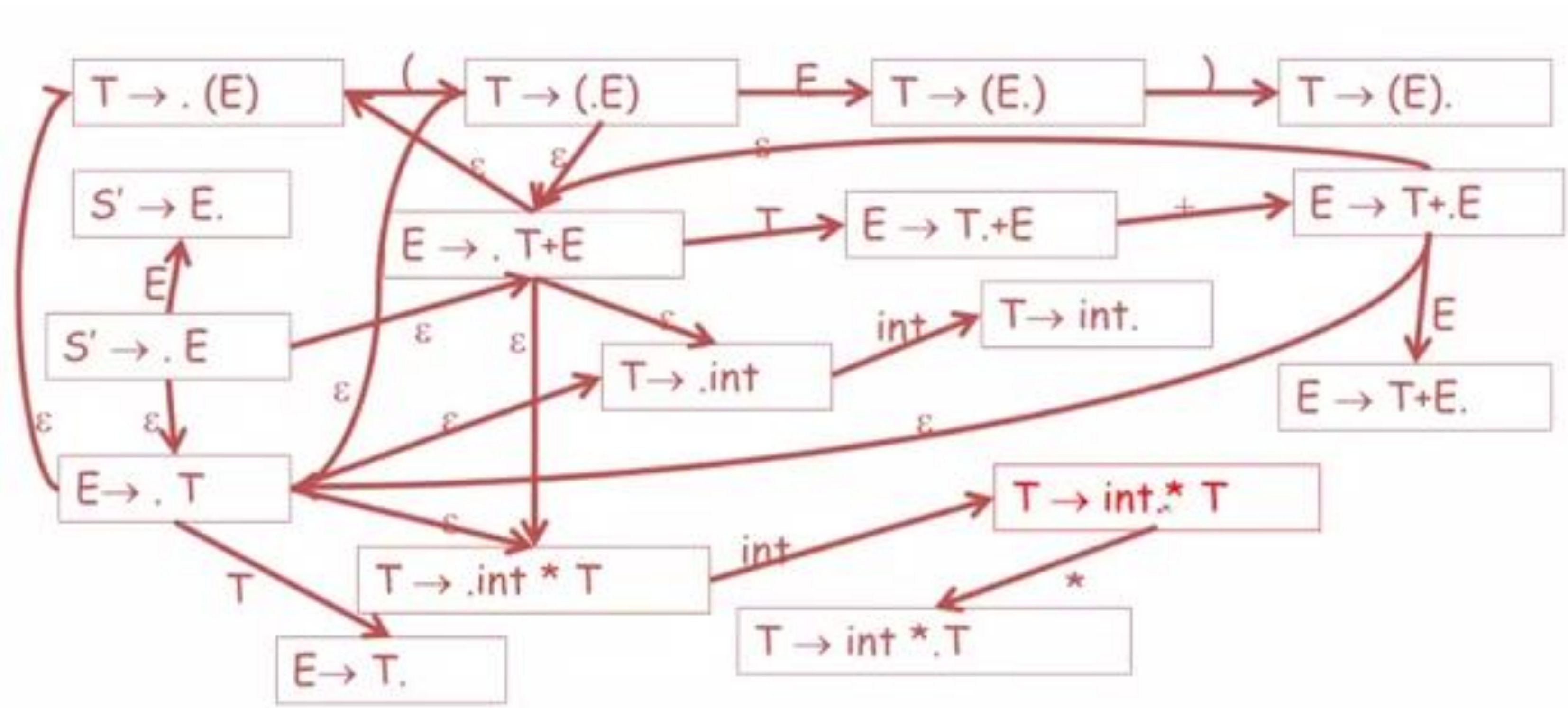
# Transitions for T → .int



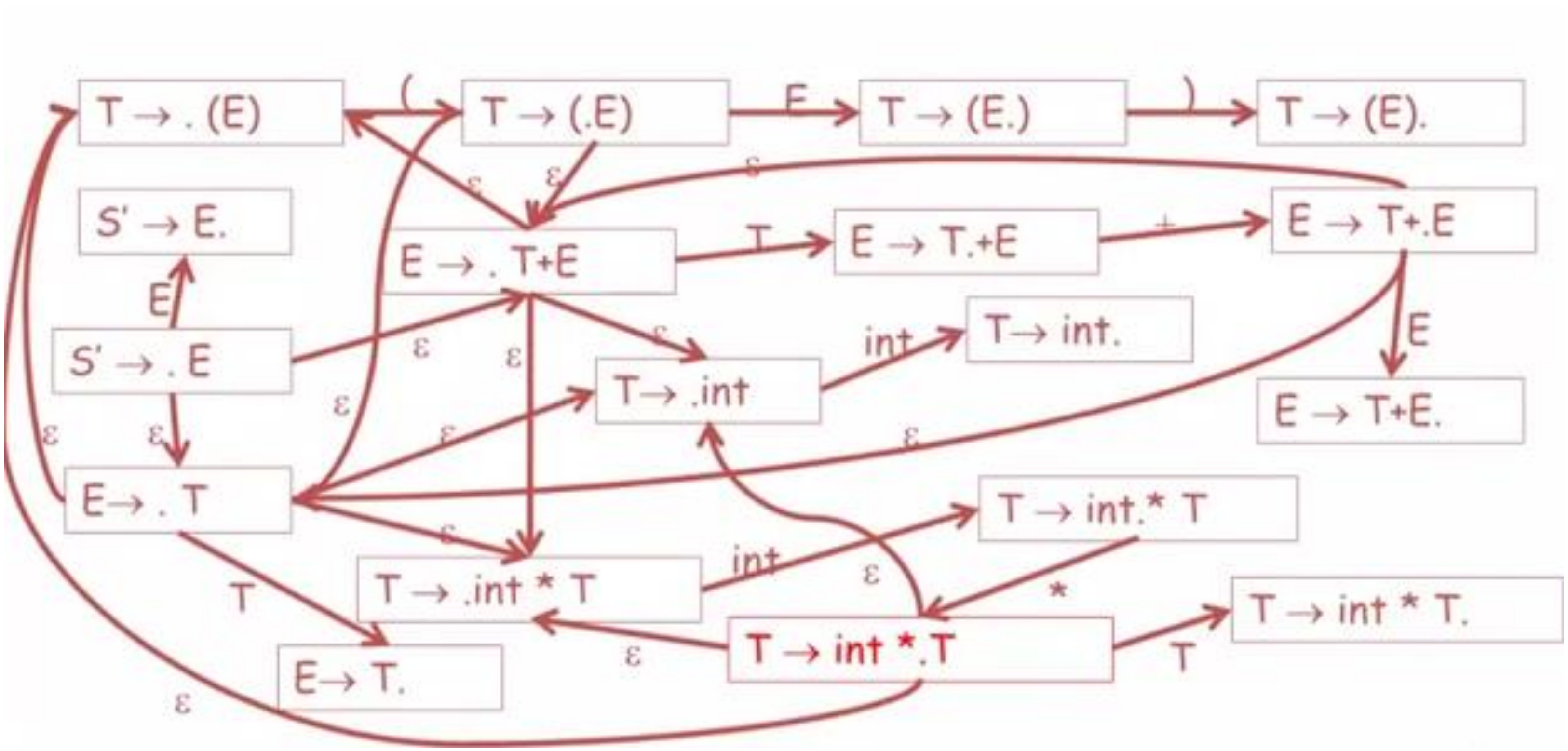
# Transitions for T → .int * T



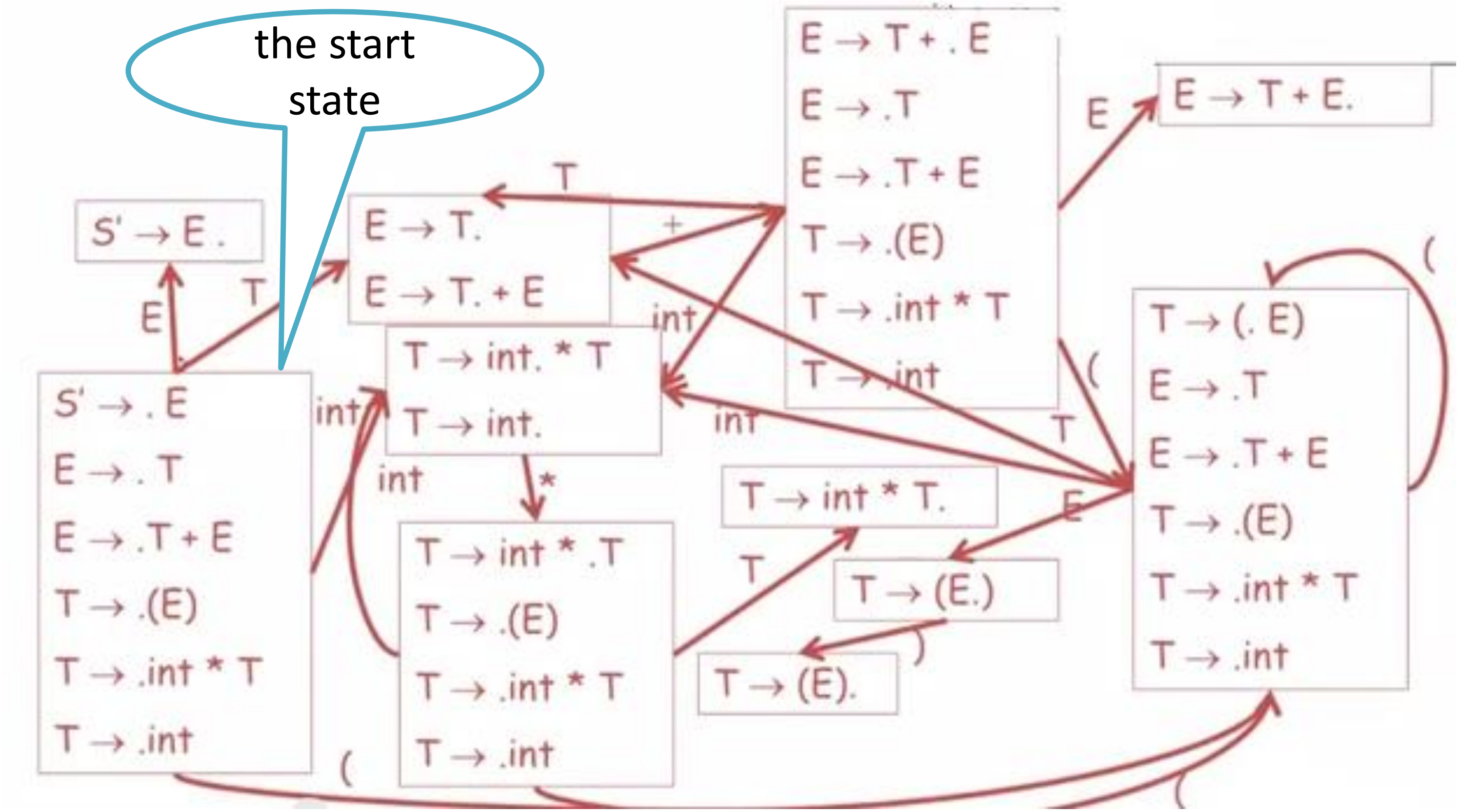
# Transitions for T → int. * T



# Transitions for T→ int *.T



# An Equivalent DFA




•Notice: that each item is state and
•the NFA can be in any of these states

- The states of the DFA are "canonical collections of items"
- Or "canonical collections of LR(0) items"
- Item $X \rightarrow \beta.\Upsilon$ is valid for a viable prefix $\alpha\beta$ if

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\Upsilon\omega$$

  by a rightmost derivation
- After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items.

# Valid Items

- An item is often valid for many prefixes
- Example: The item T → (.E) is valid for prefixes

  (

  ((

  (((

  ((((

  (((((

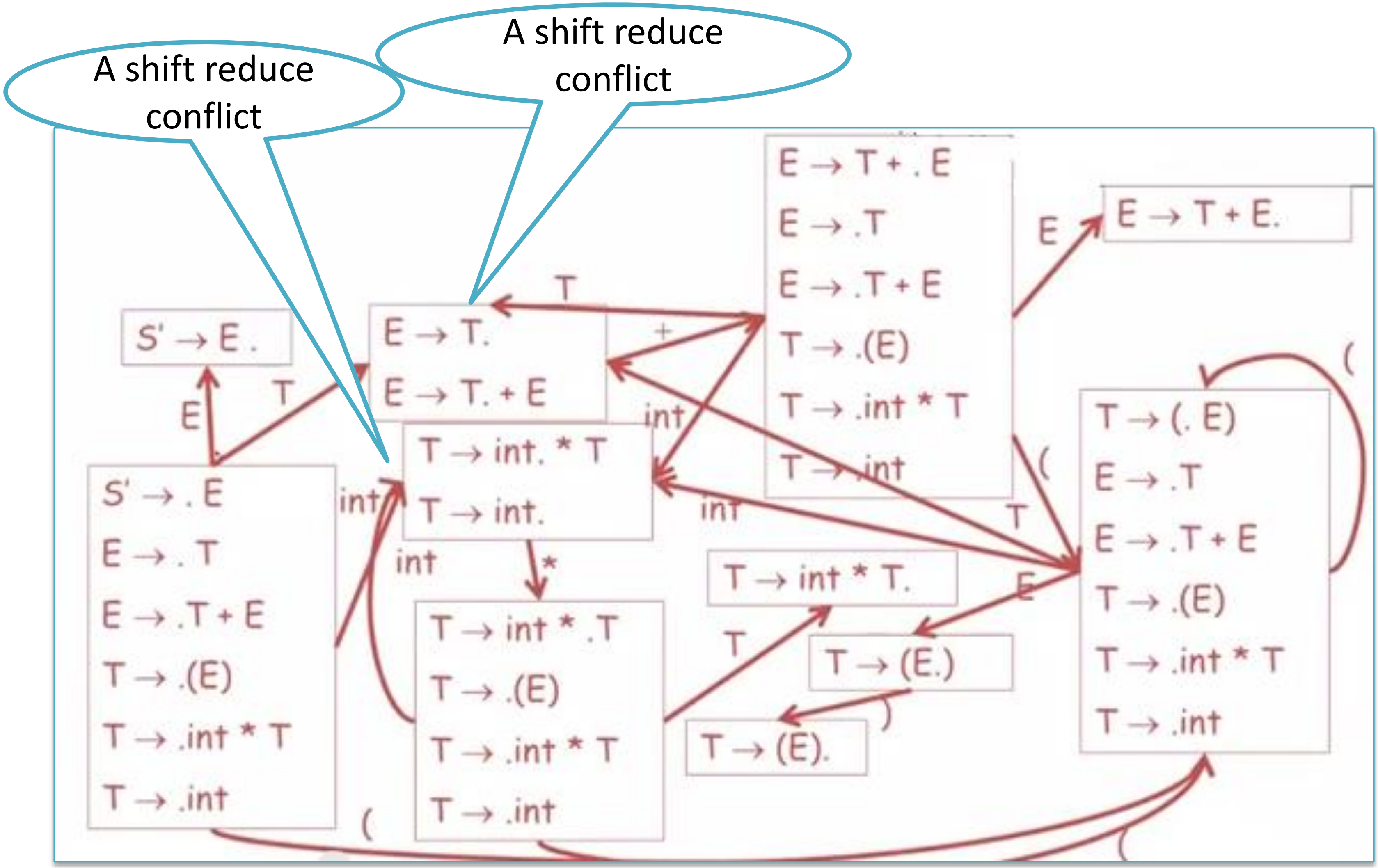
  …

# SLR Parsing Algorithm: Simple LR parsing

- LR(0) parsing: Assume
  - Stack contains α
  - Next input is t
  - DFA on input α terminates in state s
- Reduce by X → β if
  - S contains item X → β.   (i.e. we have seen a complete rhs)
- Shift if
  - S contains item X → β.tω
  - i.e. s has a transition labeled t

# 2 kinds of problems

- LR(0) may not be able to decide what to do in two situations
- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.$ (two possible reduce actions)
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$
    - (i.e. a reduce is possible and a shift is also possible)



A shift reduce conflict
A shift reduce conflict
S' → E .
E → T.
E → T. + E
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
E → T + E.
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int
T → int. * T
T → int.
T → int * .T
T → .(E)
T → .int * T
T → .int
T → int * T.
T → (E.)
T → (E).
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

# SLR Parsing

- SLR = "Simple LR"
- SLR improves on LR(0) by adding shift/reduce heuristics that
- will help us determine when to reduce and when to shift
  - Fewer states have conflicts

# SLR Parsing

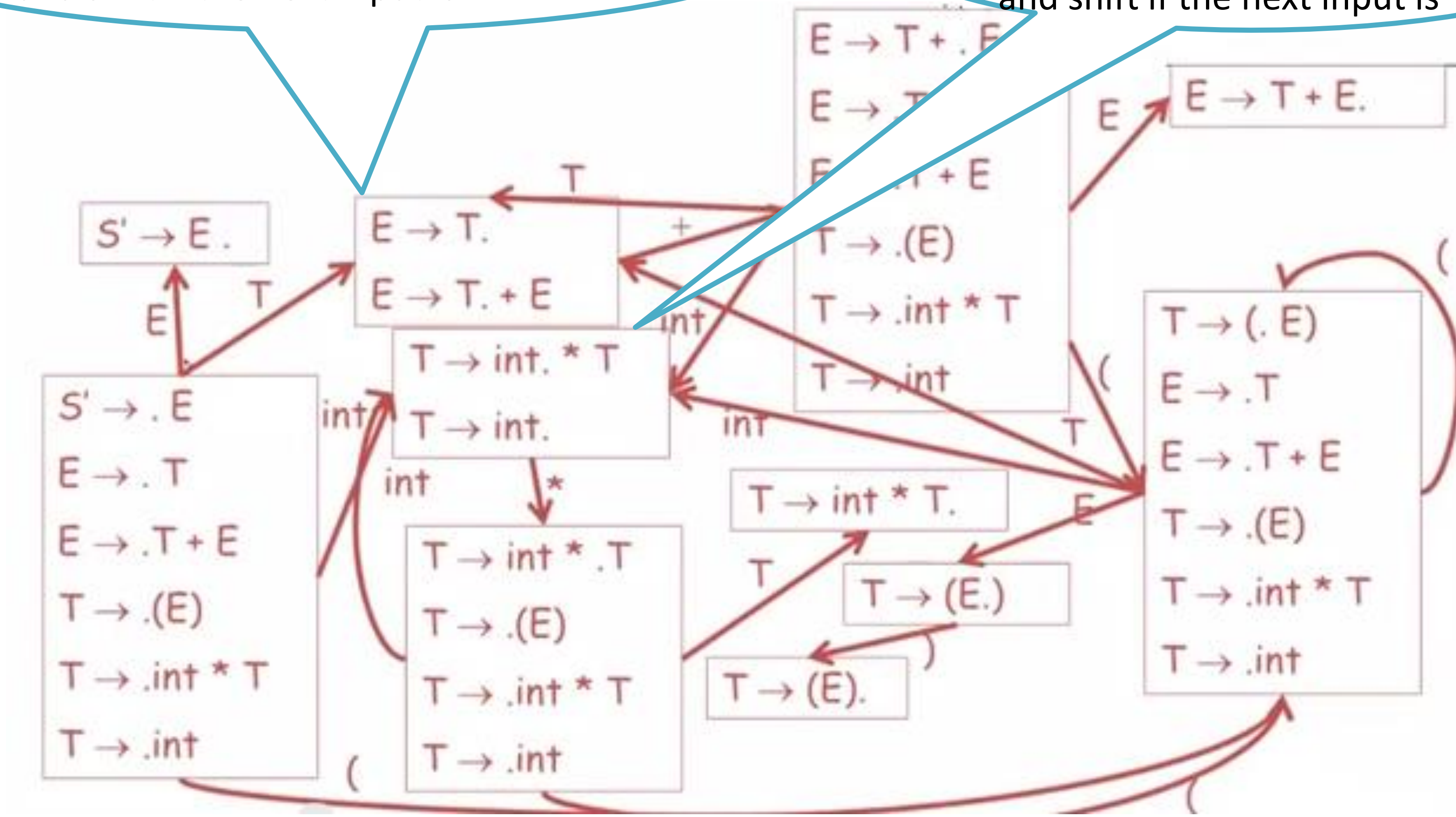
- Assumptions
  - Stack contains α
  - Next input is t
  - DFA on input α terminates in state s.
- Reduce by X → β if
  - S contains X → β.
  - And
  - t ϵ Follow(X)                (where t is the next input)
- Shift if
  - s contains item X → β.tω

- If we still have conflicts after applying these rules, then the grammar is not SLR.
- The rules are heuristics for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles.

# Parsing Example

We will reduce if the next input is in the Follow(E) which is {$,)}
and shift if the next input is +

We will reduce if the next input is in the Follow(T) which is {$,),+}
and shift if the next input is *



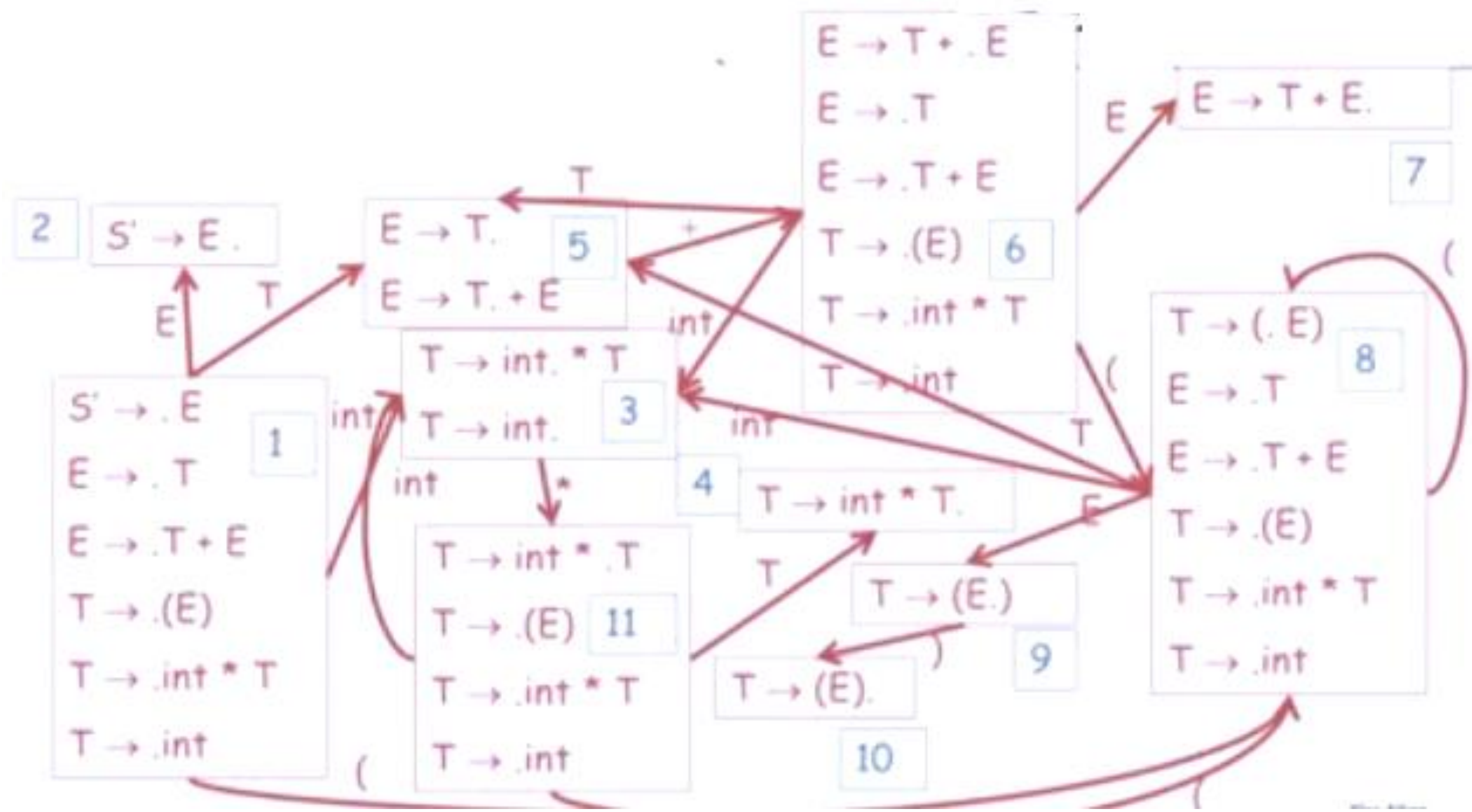
- Notice that all conflicts are resolved in the above grammar so it is an SLR grammar.
- But many grammars are not SLR
  – These include all ambiguous grammar.
- We can parse more grammars by using precedence declarations
  – Instructions for resolving conflicts

- Consider the grammar
- $E \rightarrow E+E \mid E*E \mid (E) \mid int$
- The DFA for this grammar contains a state with the following items:
  - $E \rightarrow E*E.$   And $E \rightarrow E.+E$
  - Shift/reduce conflict.
- Declaring that * has higher precedence than + resolves this conflict in favor of reducing.
- So we will not do the shift.

# SLR Parsing algorithm

- Let M be a DFA for viable prefixes of G
- Let $|x_1 \ldots X_n\$$ be initial configuration  (the stack is empty)
- Repeat until configuration is S|$      (i.e. until all input has been consumed)
  - Let $\alpha|\omega$ be current configuration
  - Run M on current stack $\alpha$
  - If M rejects $\alpha$, report parsing error
    - Stack $\alpha$ is not a viable prefix
  - If M accepts $\alpha$ and ends in a state with items I, let a be next input
    - Shift if $X \rightarrow \beta.a\Upsilon \in I$
    - Reduce if $X \rightarrow \beta. \in I$ and $a \in Follow(X)$
    - Report parsing error if neither applies

# A Parsing Example: int * int$



| Configuration | DFA Halt State | Action |
|---|---|---|
| \|int*int$ | 1 | Shift |
| int\| * int$ | 3 (because * is not in Follow(T)) | Shift |
| int * \| int $ | 11 | shift |
| int * int \|$ | 3 Because $ϵ Follow(T) | Reduce T → int |
| int * T \|$ | 4 Because $ϵ Follow(T) | Reduce T → int*T |
| T\|$ | 5 Because $ϵ Follow(E) | Reudce E → T |
| E\|$ | | accept (since E is the start symbol) |