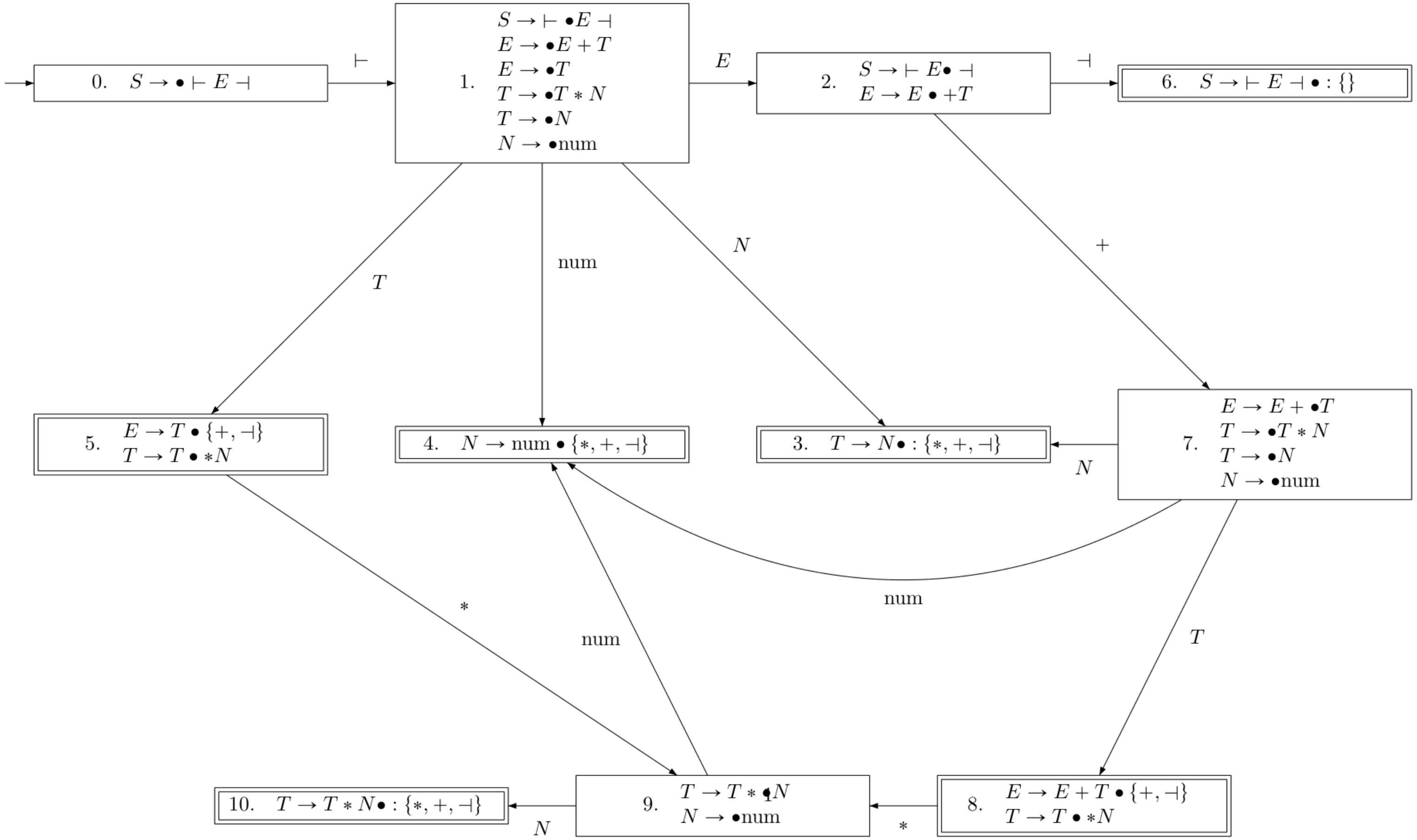
Bottom-Up Parsing:
SLR(1) and LR(1)

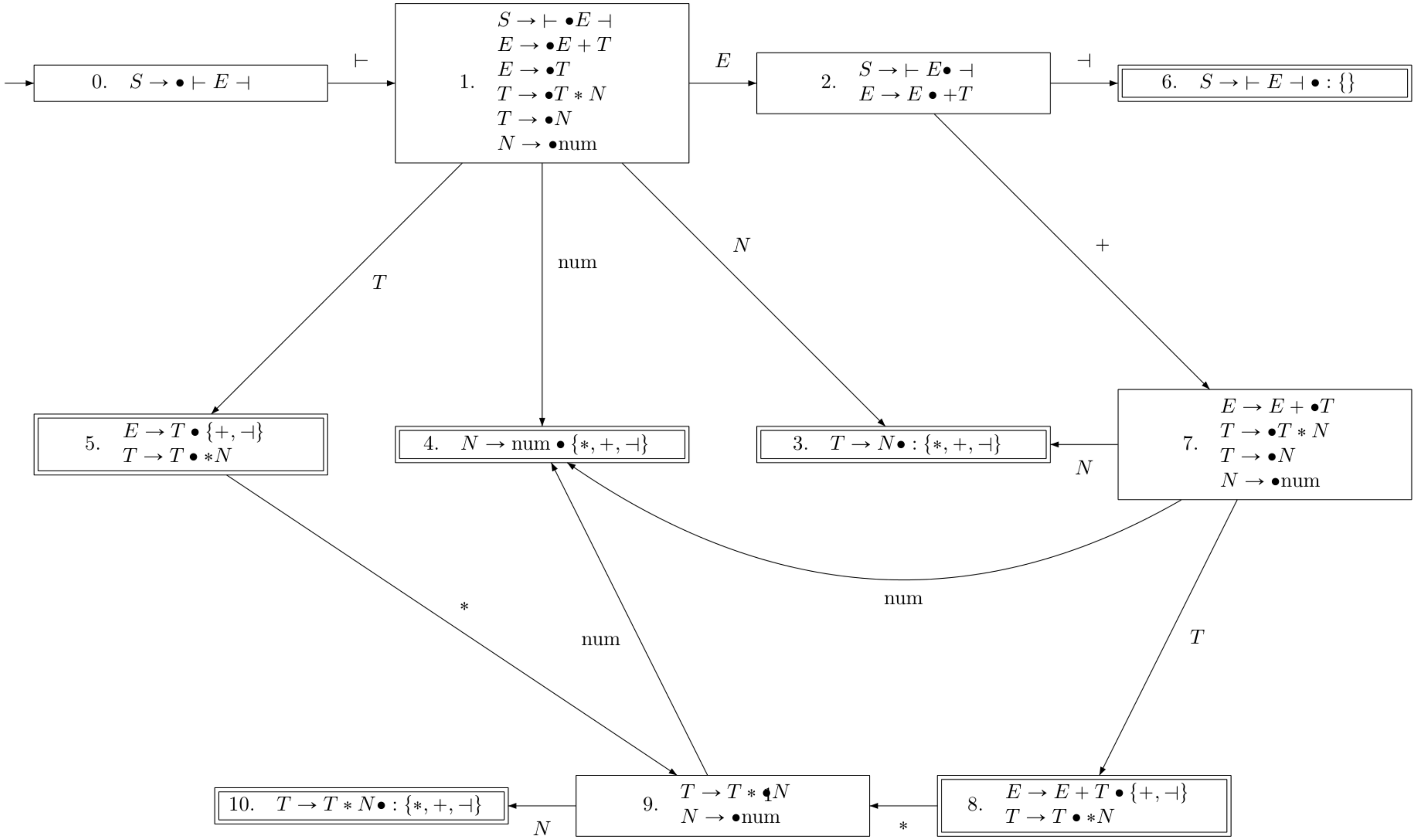
SLR(1): Using Follow Sets to Resolve Conflicts

- The idea of SLR(1) is to use the same DFA construction as LR(0), but every reducible item is "tagged" with the Follow set of the LHS.
- If you are in a reduce state, and the lookahead (first symbol of unread input) is in the "tag" of a reducible item, reduce using that rule.
- Otherwise, shift.



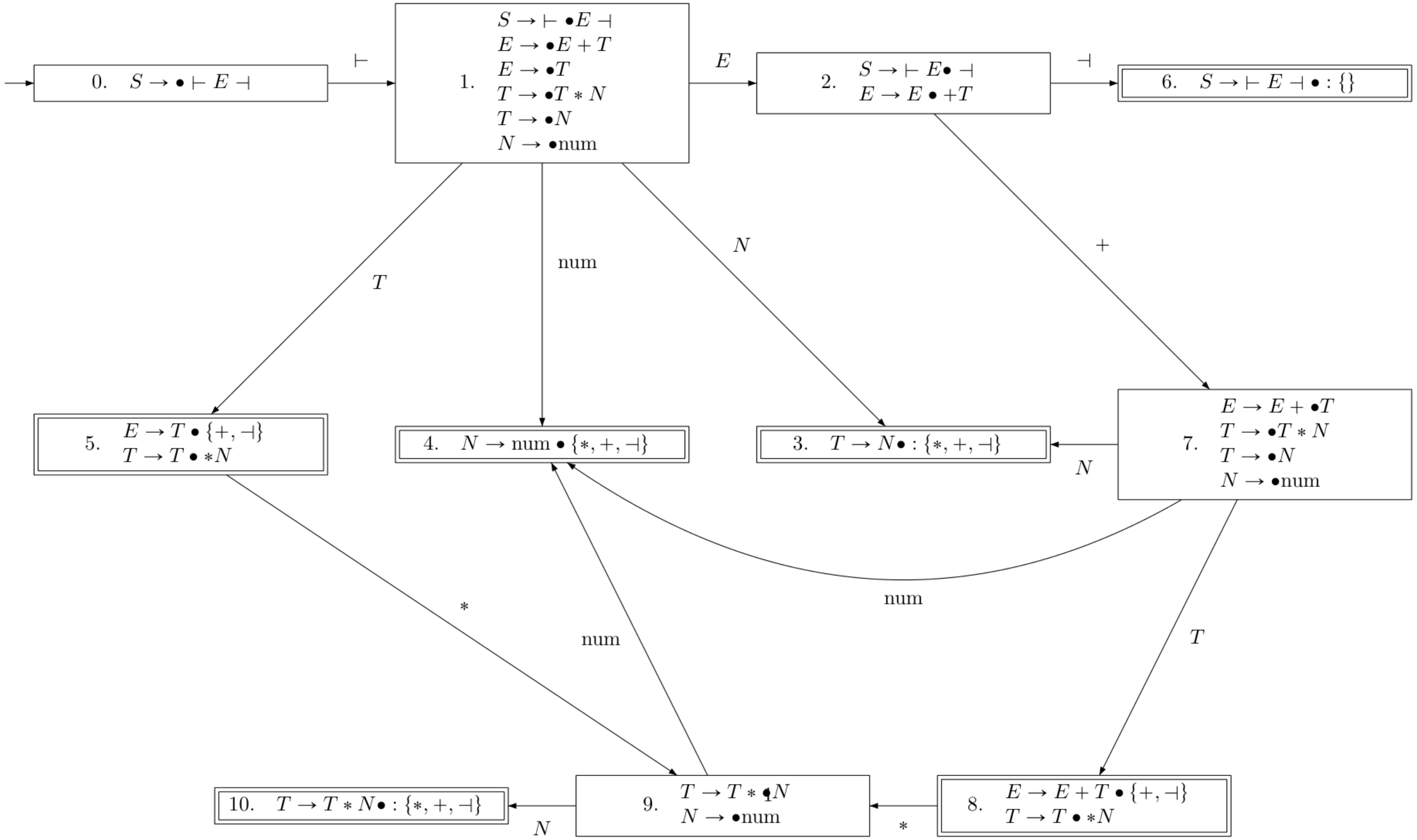
SLR(1): Using Follow Sets to Resolve Conflicts

- The idea of SLR(1) is to use the same DFA construction as LR(0), but every reducible item is "tagged" with the Follow set of the LHS.
- Shift-Reduce conflicts are resolved if the symbol to be shifted is not in the Follow set for the reducible item.
- Reduce-Reduce conflicts are resolved if the Follow sets don't overlap.



SLR(1): Using Follow Sets to Resolve Conflicts

- For example, if the reduction sequence is: **⊢ T**
- If the next symbol of unread input is + or ⊣ we reduce by E → T.
- Otherwise, we shift.
- If the rest of input is
  ⊣
  or
  + T ⊣
  reducing is correct.
- If it's * N ⊣ then shifting is correct.



The LR(0) Parsing Algorithm: Pseudocode

```
initialize symbolStack to an empty stack
initialize stateStack with the initial state of the parsing DFA
while input is not empty:
  a = first symbol of input
  (Reduce until we are no longer in a reduce state)
  while stateStack.top contains a reducible item [A→α•]:
    len = length of α                   (number of symbols on right-hand side)
    pop len symbols from symbolStack    (pop the right-hand side)
    push A to symbolStack               (push the left-hand side)
    pop len states from stateStack      (backtrack in the DFA)
    let newState be obtained by taking the transition from stateStack.top on A
    push newState to stateStack         (state stack is again synchronized with symbol stack)
  (Once we can no longer reduce, shift a symbol from input)
  if there is a transition from stateStack.top on a to newState:
    push a to symbolStack               (push the symbol-to-shift)
    push newState to stateStack         (keep the state stack synchronized)
    consume a from input                (read and remove first symbol from input)
  else
    ERROR                               (no transition on input symbol, parse failed)
```

The **LR(1)** Parsing Algorithm: Pseudocode

The only difference!

```
initialize symbolStack to an empty stack
initialize stateStack with the initial state of the parsing DFA
while input is not empty:
  a = first symbol of input
  (Reduce until we are no longer in a reduce state)
  while stateStack.top contains a reducible item [A→α•] with a in the lookahead tag:
    len = length of α                     (number of symbols on right-hand side)
    pop len symbols from symbolStack      (pop the right-hand side)
    push A to symbolStack                 (push the left-hand side)
    pop len states from stateStack        (backtrack in the DFA)
    let newState be obtained by taking the transition from stateStack.top on A
    push newState to stateStack           (state stack is again synchronized with symbol stack)
  (Once we can no longer reduce, shift a symbol from input)
  if there is a transition from stateStack.top on a to newState:
    push a to symbolStack                 (push the symbol-to-shift)
    push newState to stateStack           (keep the state stack synchronized)
    consume a from input                  (read and remove first symbol from input)
  else
    ERROR                                 (no transition on input symbol, parse failed)
```

SLR(1) vs. LR(1)

- The LR(1) parsing algorithm can be used with any kind of parsing DFA that has "lookahead tags".
- The SLR(1) DFA uses Follow sets as lookahead tags.
- The term **LR(1) DFA** refers to a more complex construction (not covered in this course) where only a subset of the Follow set is used.
  - The LR(1) DFA resolves more LR(0) conflicts than the SLR(1) DFA, but the number of states can be exponentially larger than the SLR(1) DFA.
- There is also something called LALR(1) ("Lookahead LR(1)") which is a compromise and is popular in practice. It resolves more conflicts than SLR(1), and uses less states than LR(1) but resolves fewer conflicts.

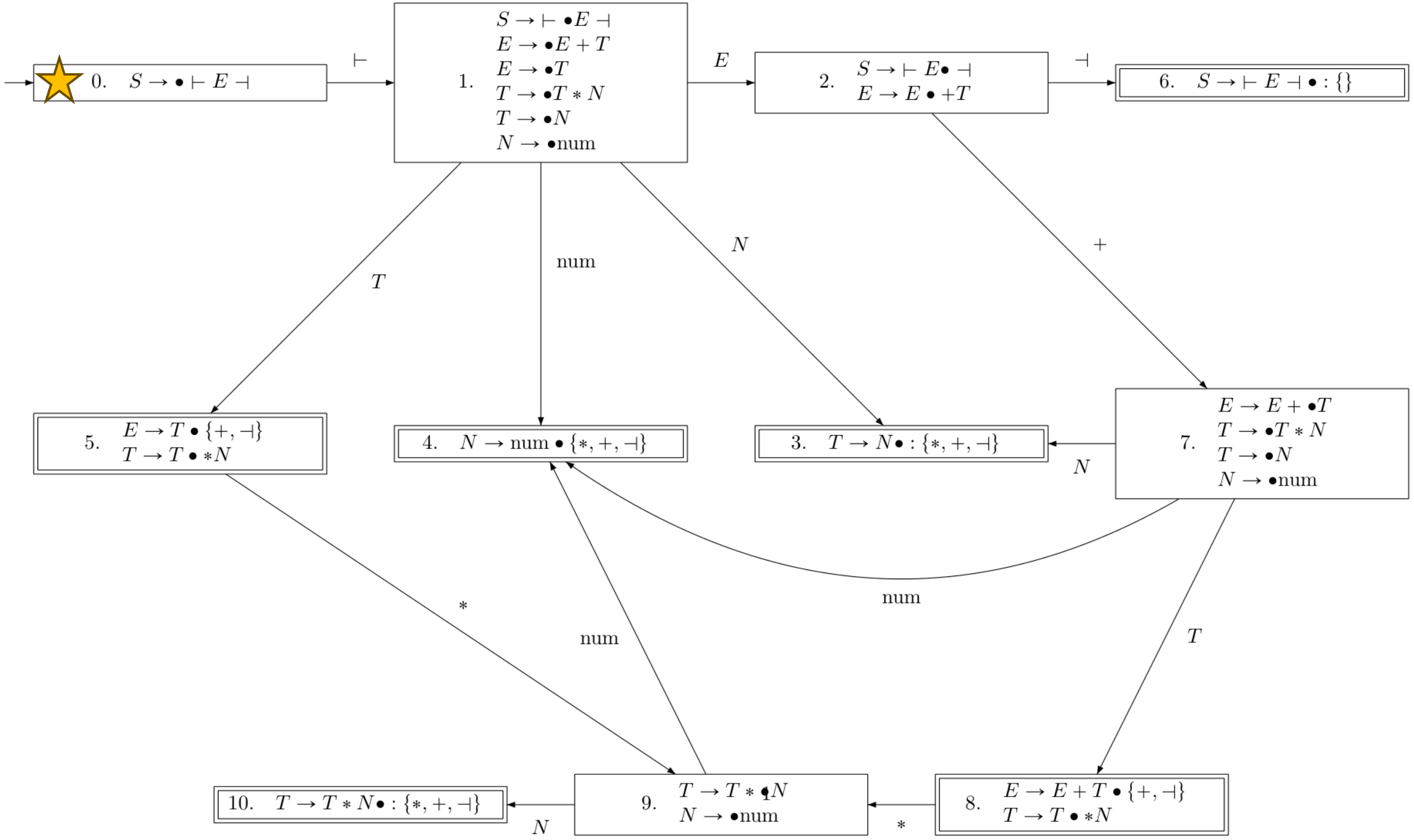
Building a Parse Tree

- The pseudocode on the previous slides doesn't actually produce any result. It either runs to completion, or produces an error.
- To make it produce a **derivation**, we could modify it to output the reduce rule every time we do a reduce step.
  - The derivation would be in reverse order.
- A better option is to make it produce a **parse tree**.
- The idea is to replace the symbol stack with a **tree stack**.
  - When shifting, we add leaf nodes corresponding to the shifted terminal.
  - When reducing, we pop tree nodes corresponding to the rule RHS, make them children of a new node with the LHS, and push this new tree.

⊢ num * num ⊣

State Stack: 0

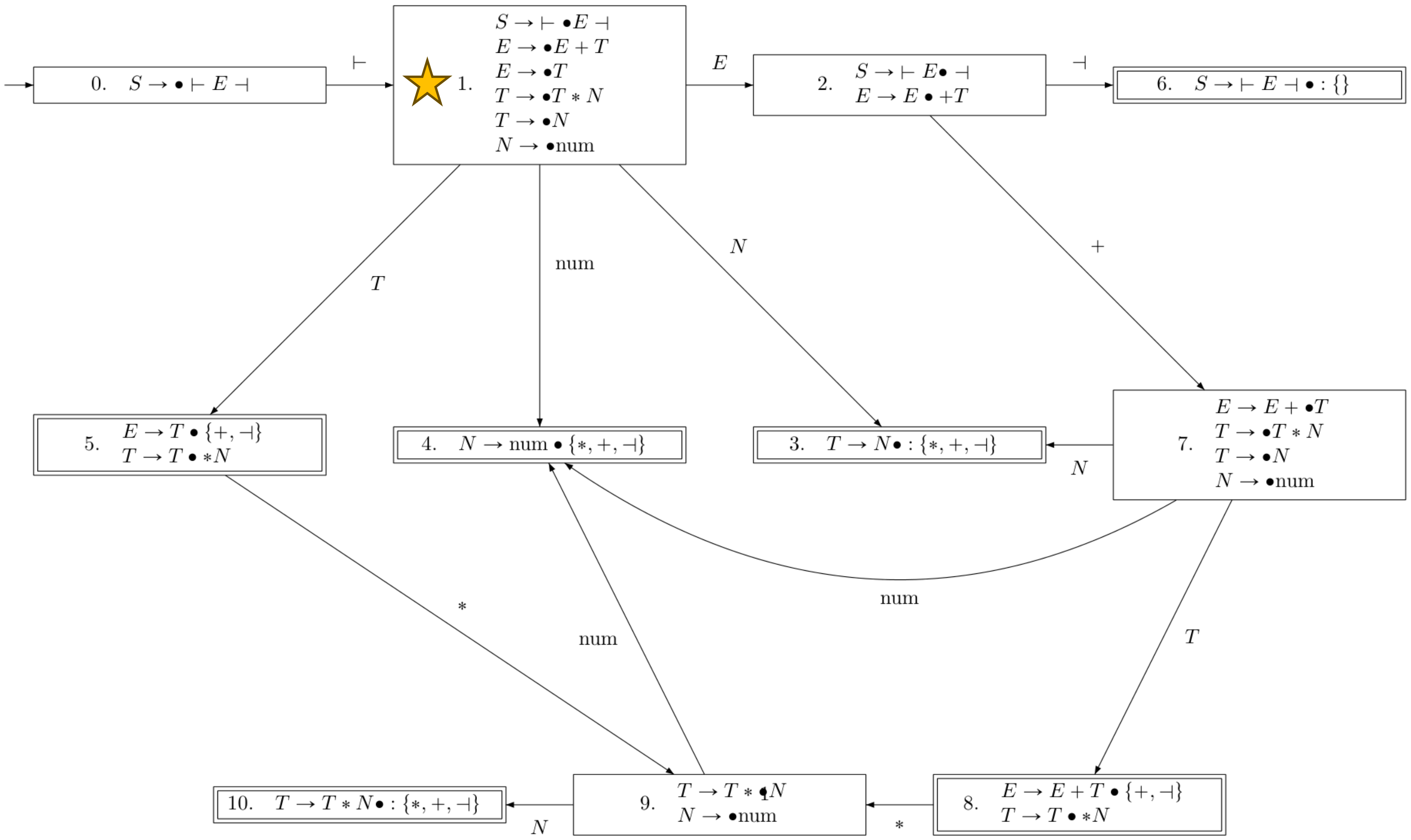
Tree Stack Top



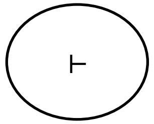
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1



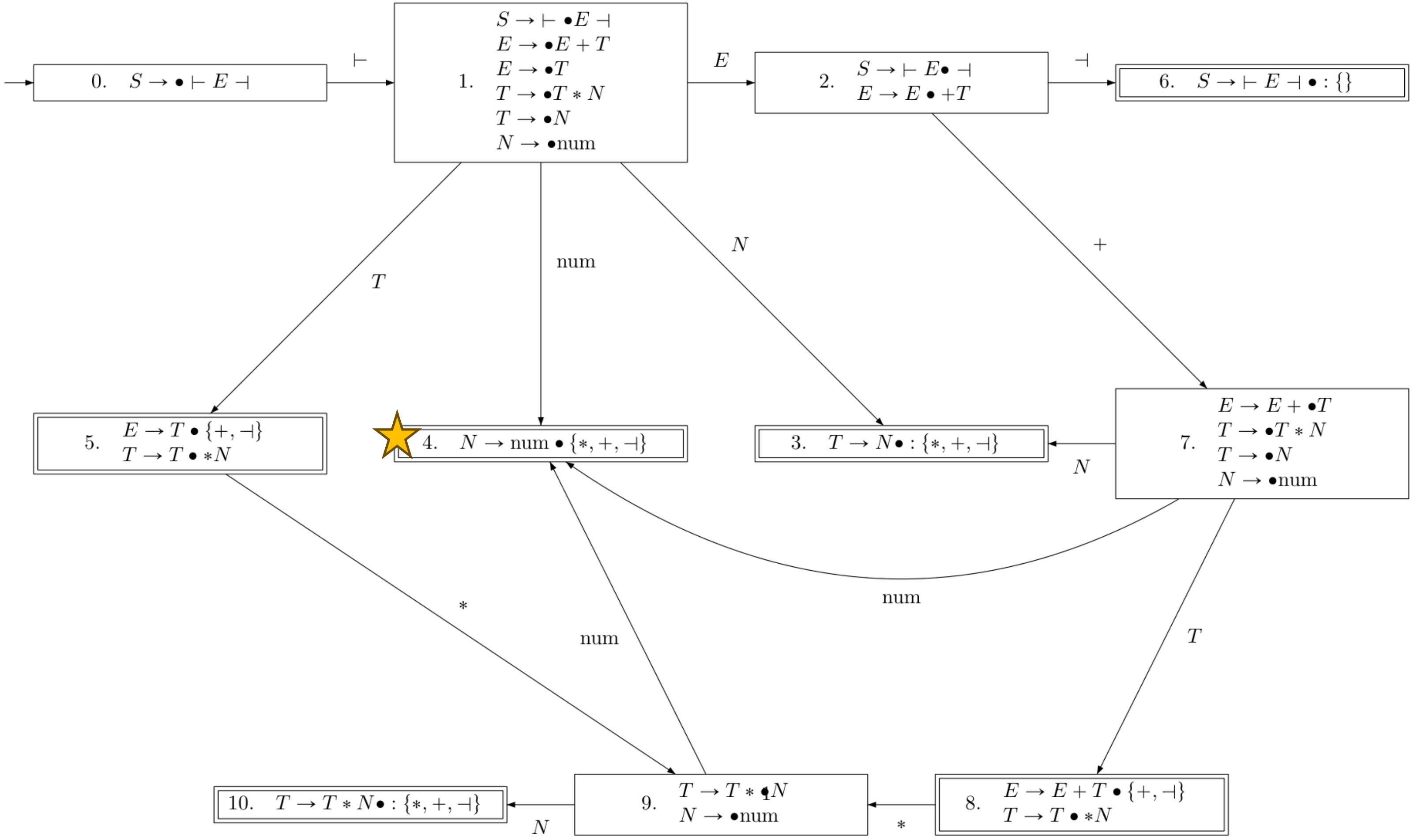
Tree Stack Top



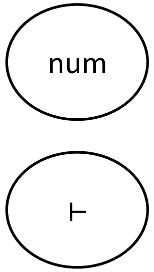
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 4



Tree Stack Top

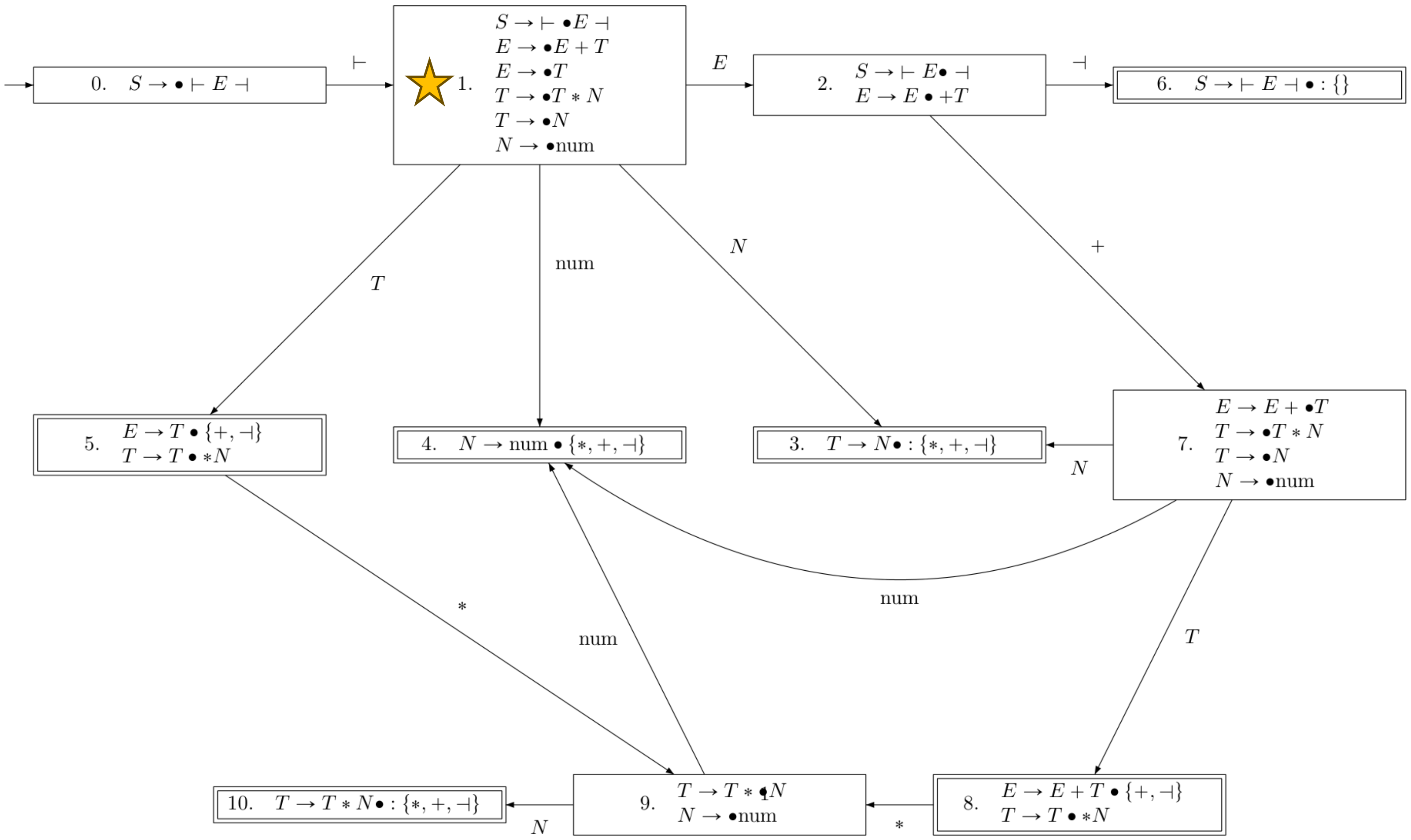


Tree Stack Bottom

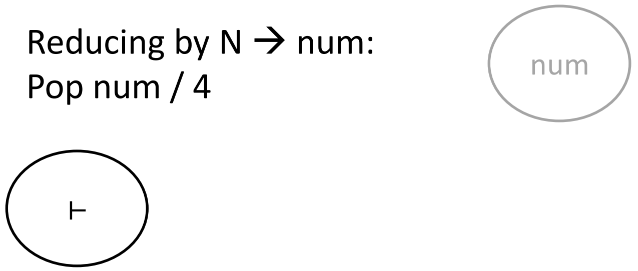
⊢ num * num ⊣

State Stack: 0 1

Tree Stack Top



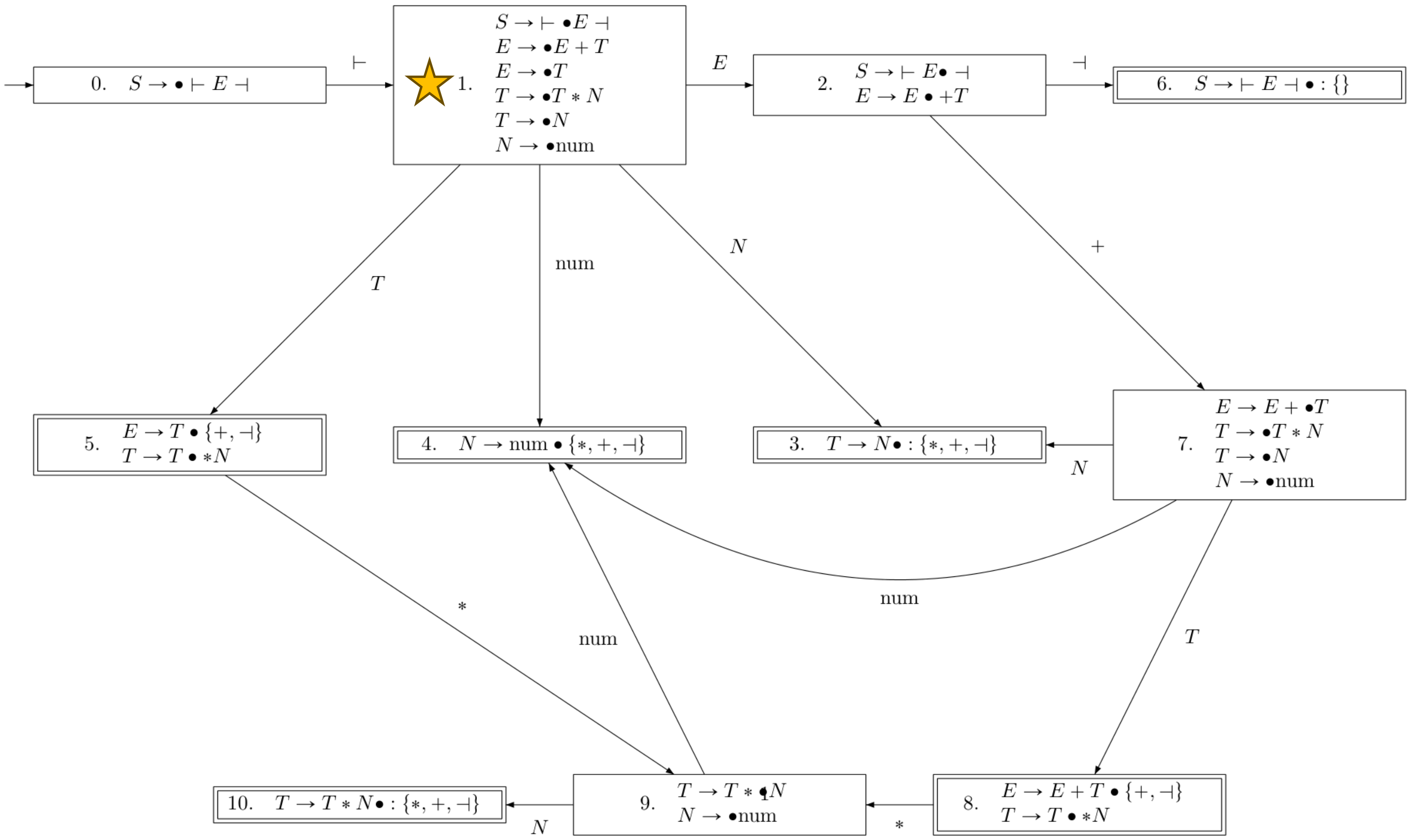
Reducing by N → num:
Pop num / 4



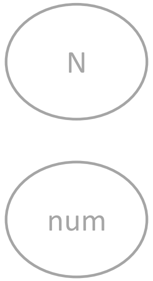
Tree Stack Bottom

⊢ num * num ⊣

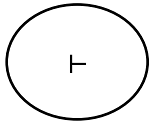
State Stack: 0 1



Tree Stack Top



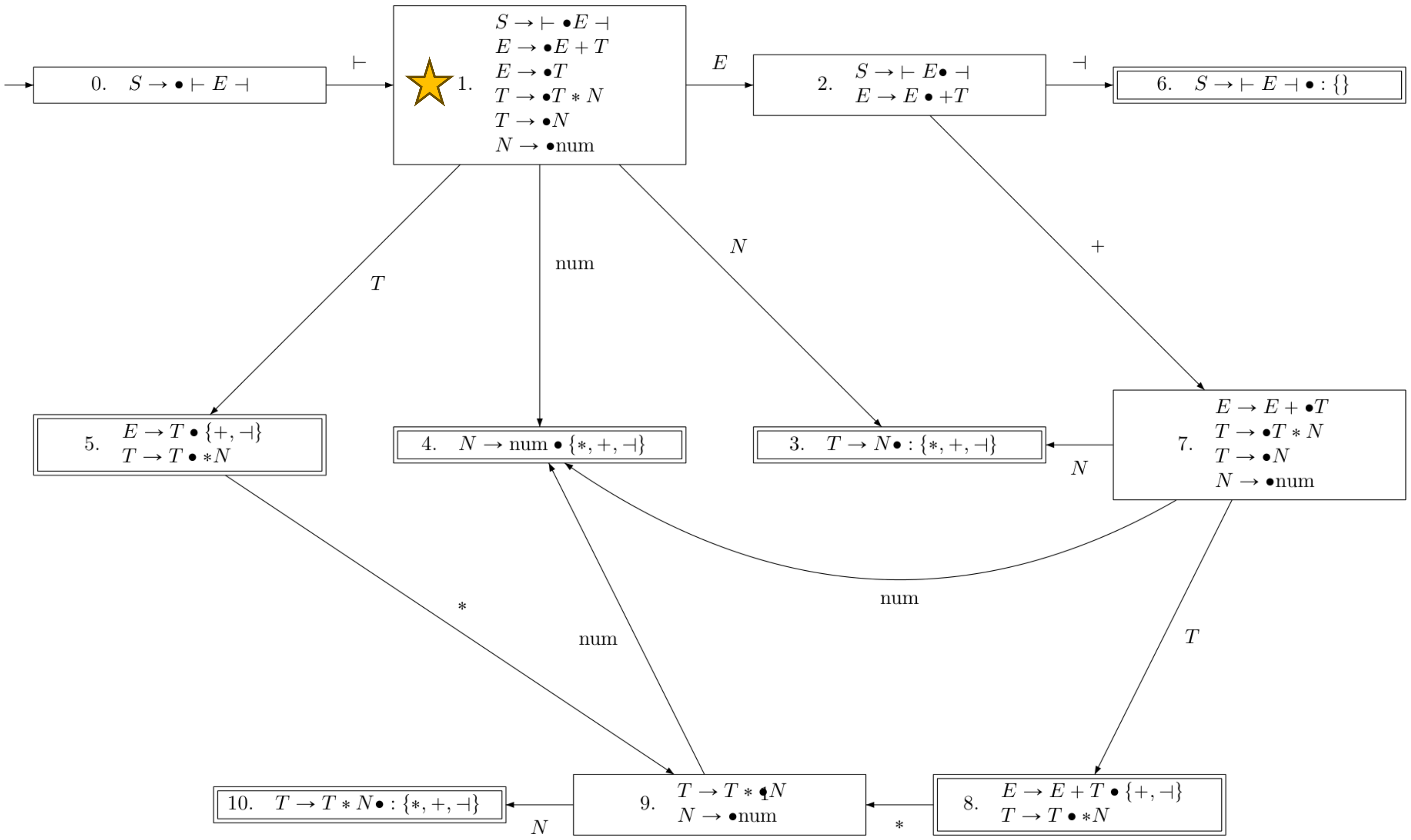
Reducing by N → num:
Create node for N



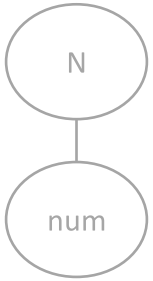
Tree Stack Bottom

⊢ num * num ⊣

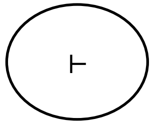
State Stack: 0 1



Tree Stack Top



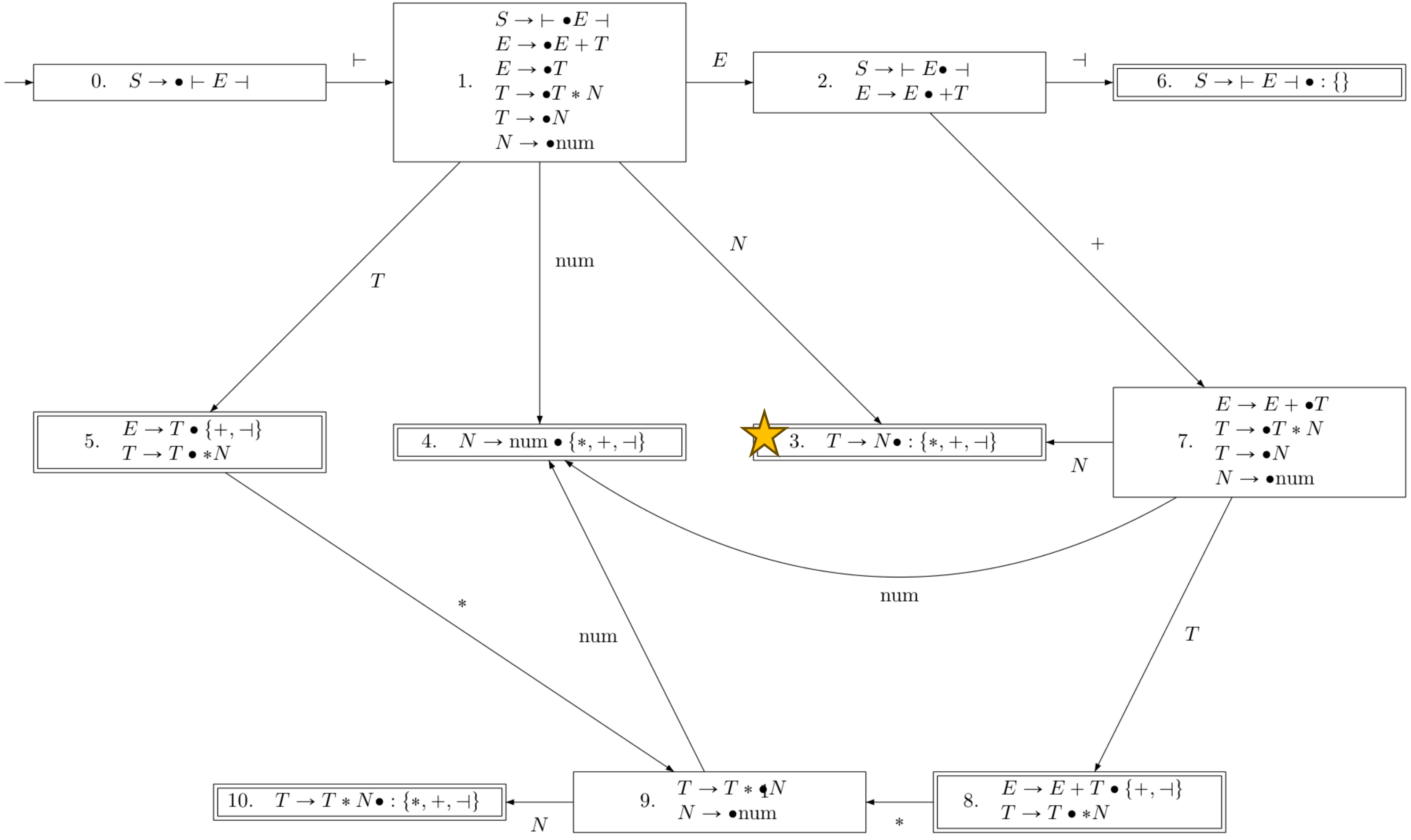
Reducing by N → num:
Add num node as child



Tree Stack Bottom

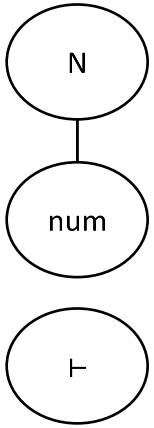
⊢ num * num ⊣

State Stack: 0 1 3



Tree Stack Top

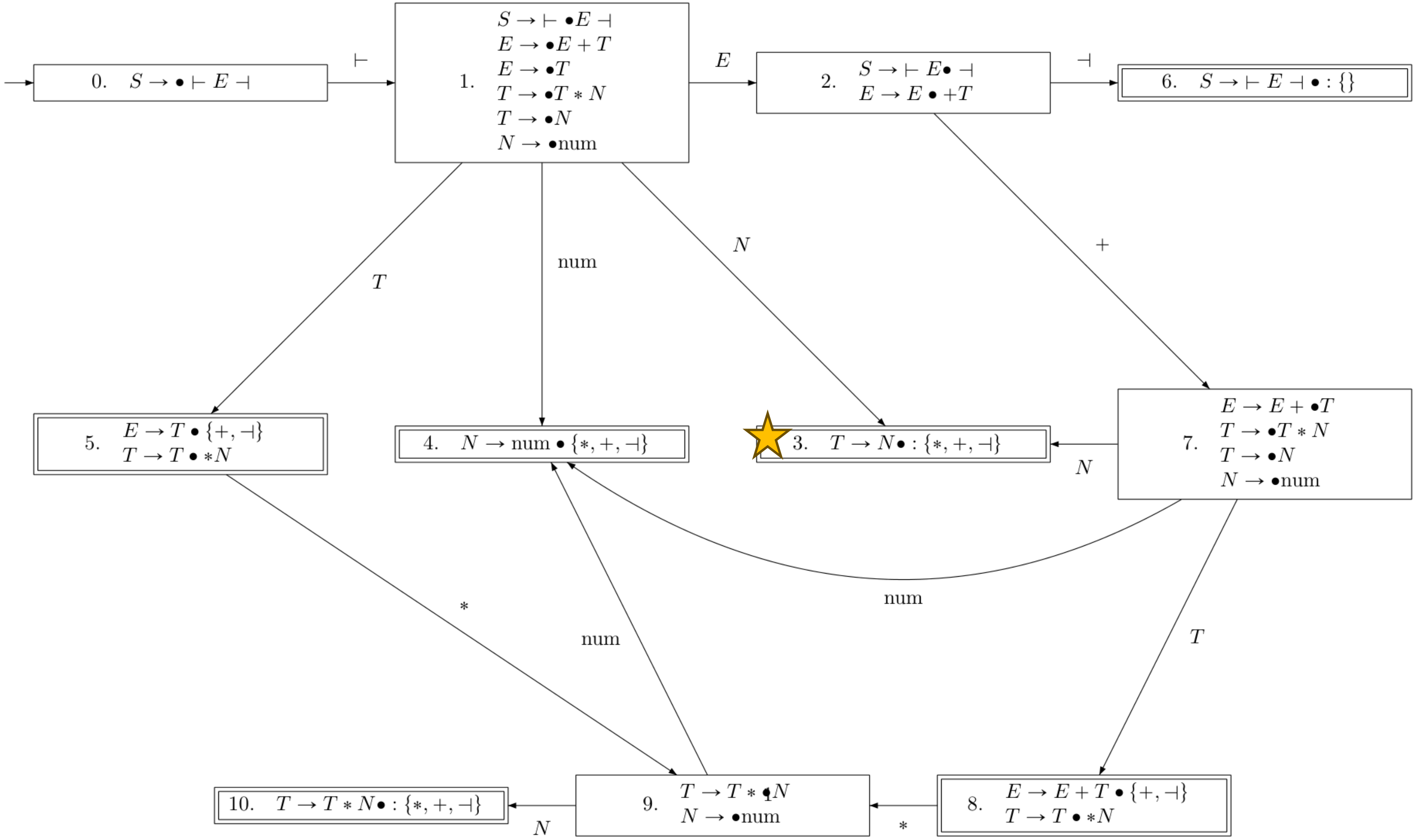
Reducing by N → num:
Push N / 3



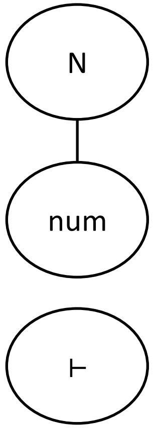
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 3



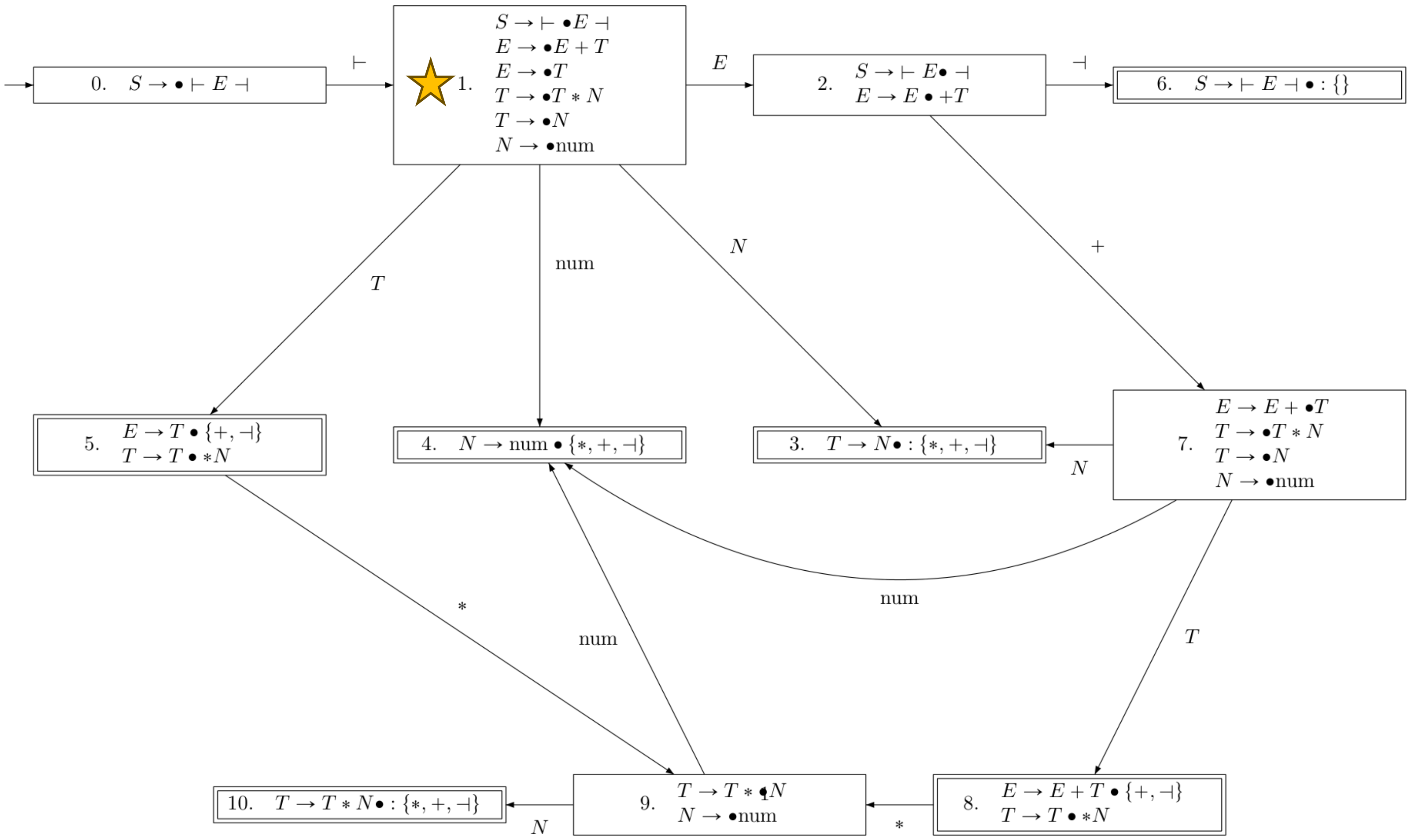
Tree Stack Top



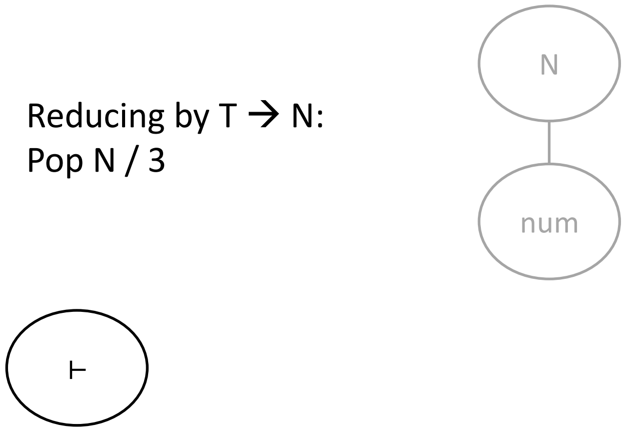
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1



Tree Stack Top

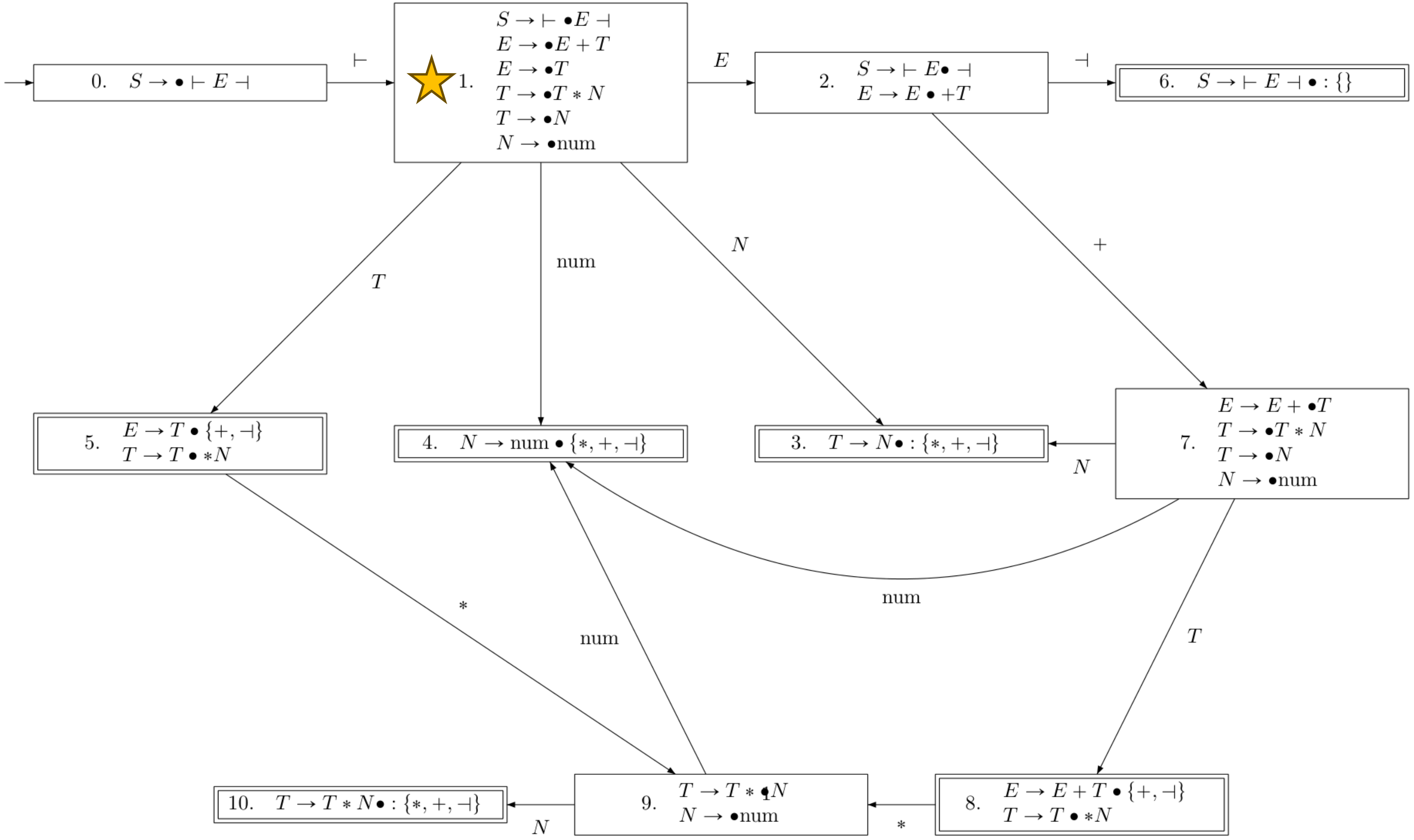


Reducing by T → N:
Pop N / 3

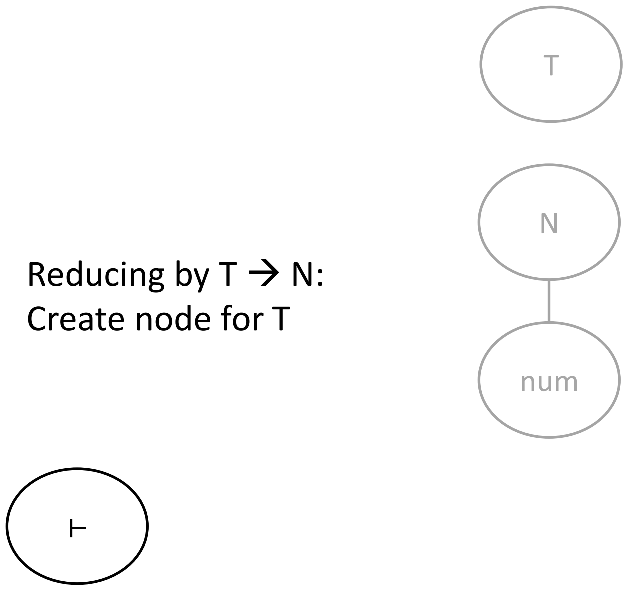
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1



Tree Stack Top

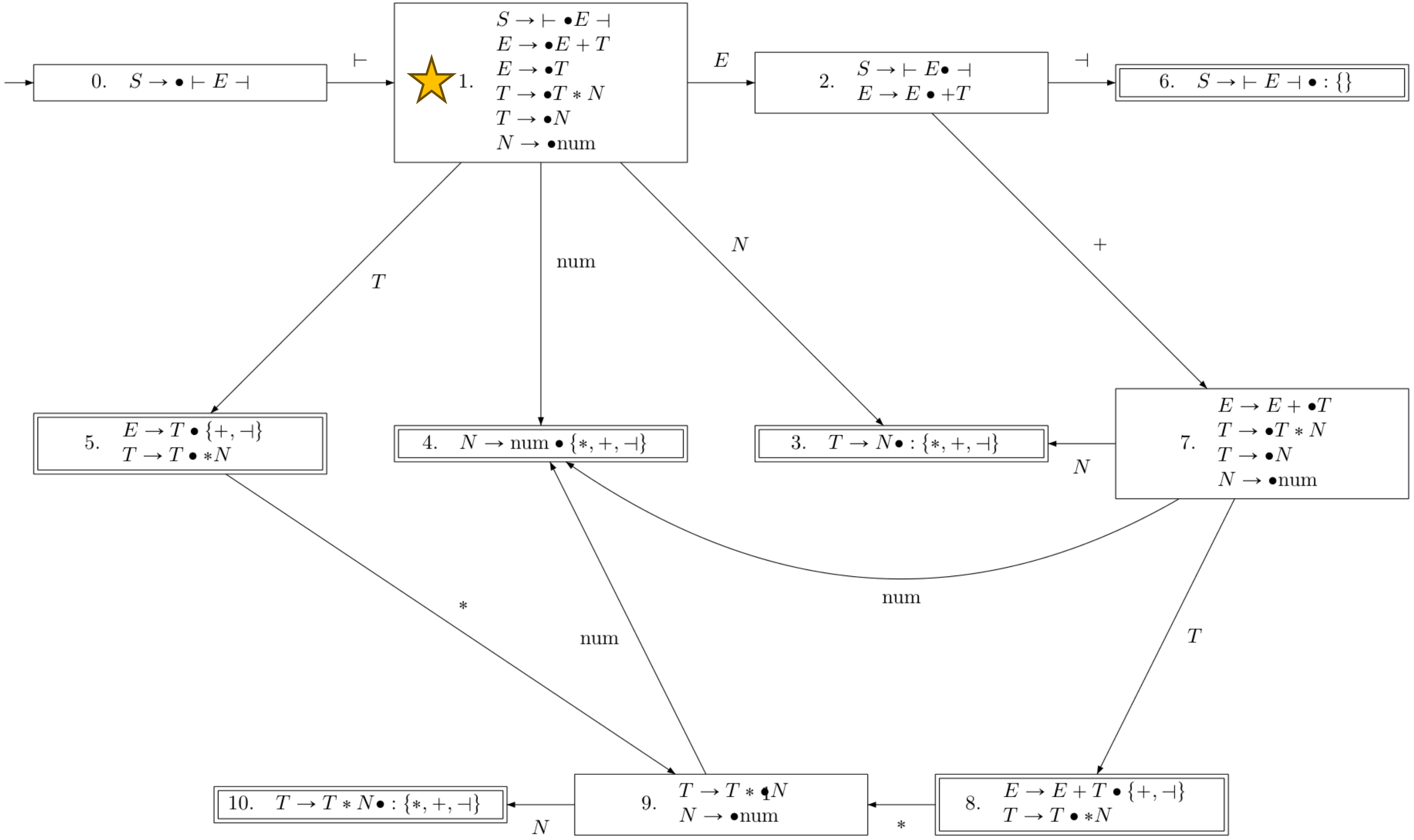


Reducing by T → N:
Create node for T

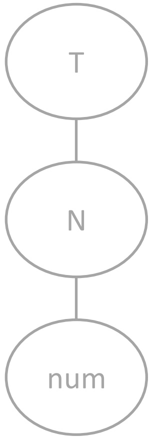
Tree Stack Bottom

⊢ num * num ⊣

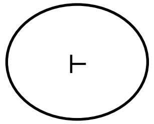
State Stack: 0 1



Tree Stack Top



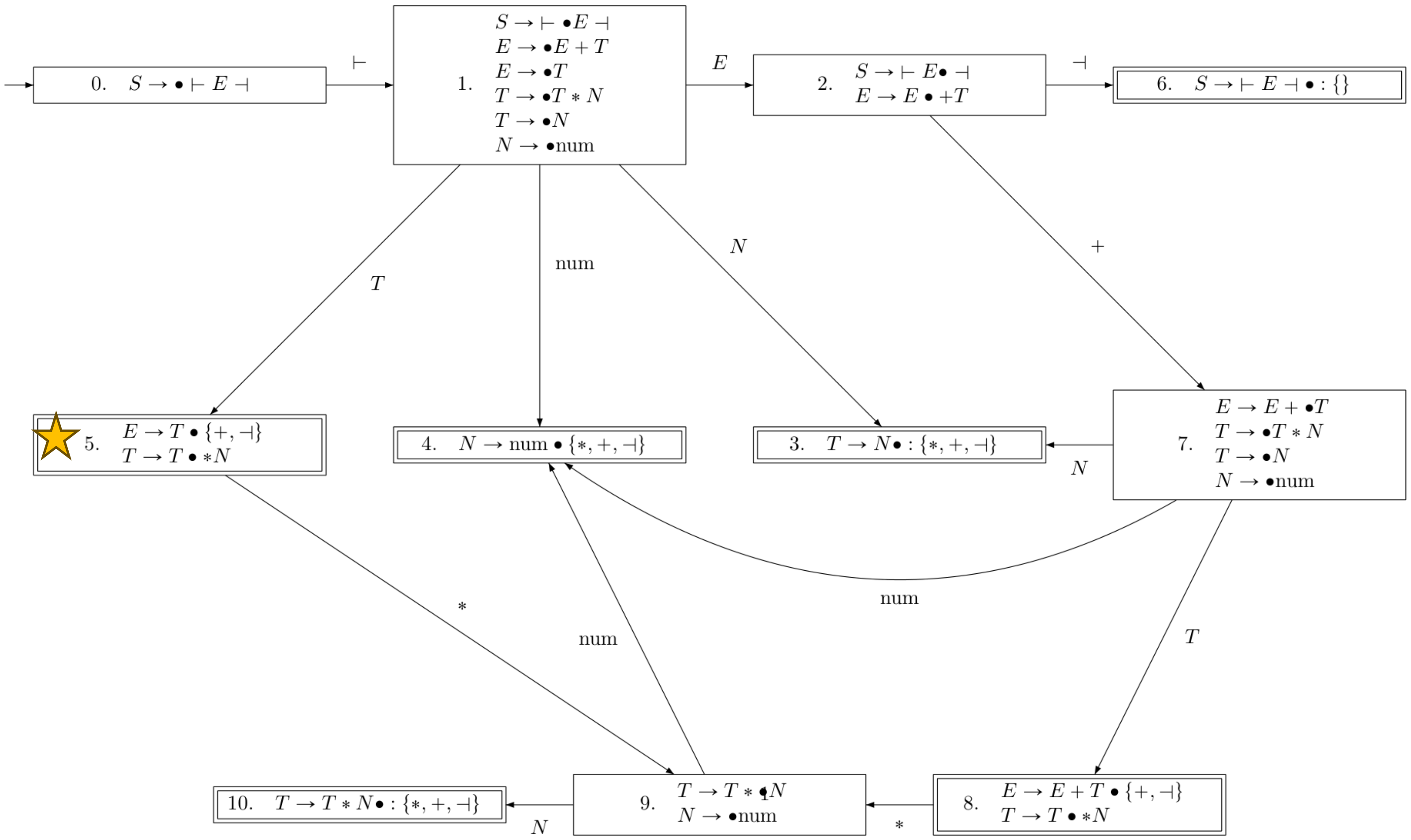
Reducing by T → N:
Add N node as child



Tree Stack Bottom

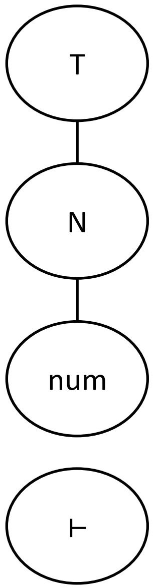
⊢ num * num ⊣

State Stack: 0 1 5



Tree Stack Top

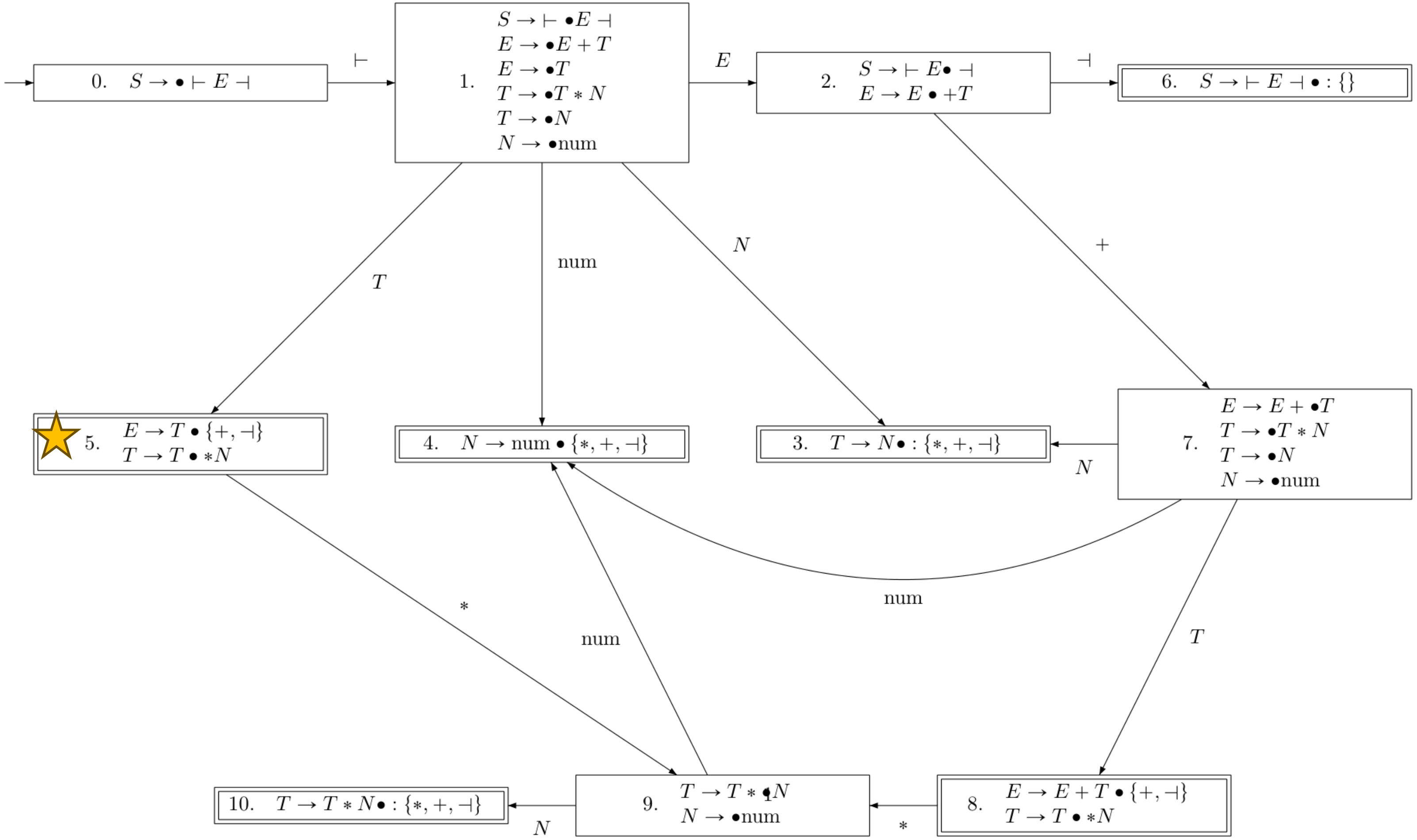
Reducing by T → N:
Push T / 5



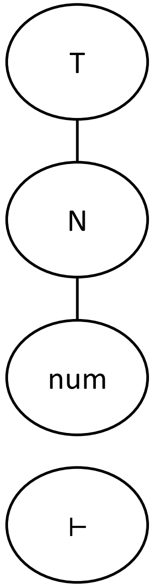
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 5



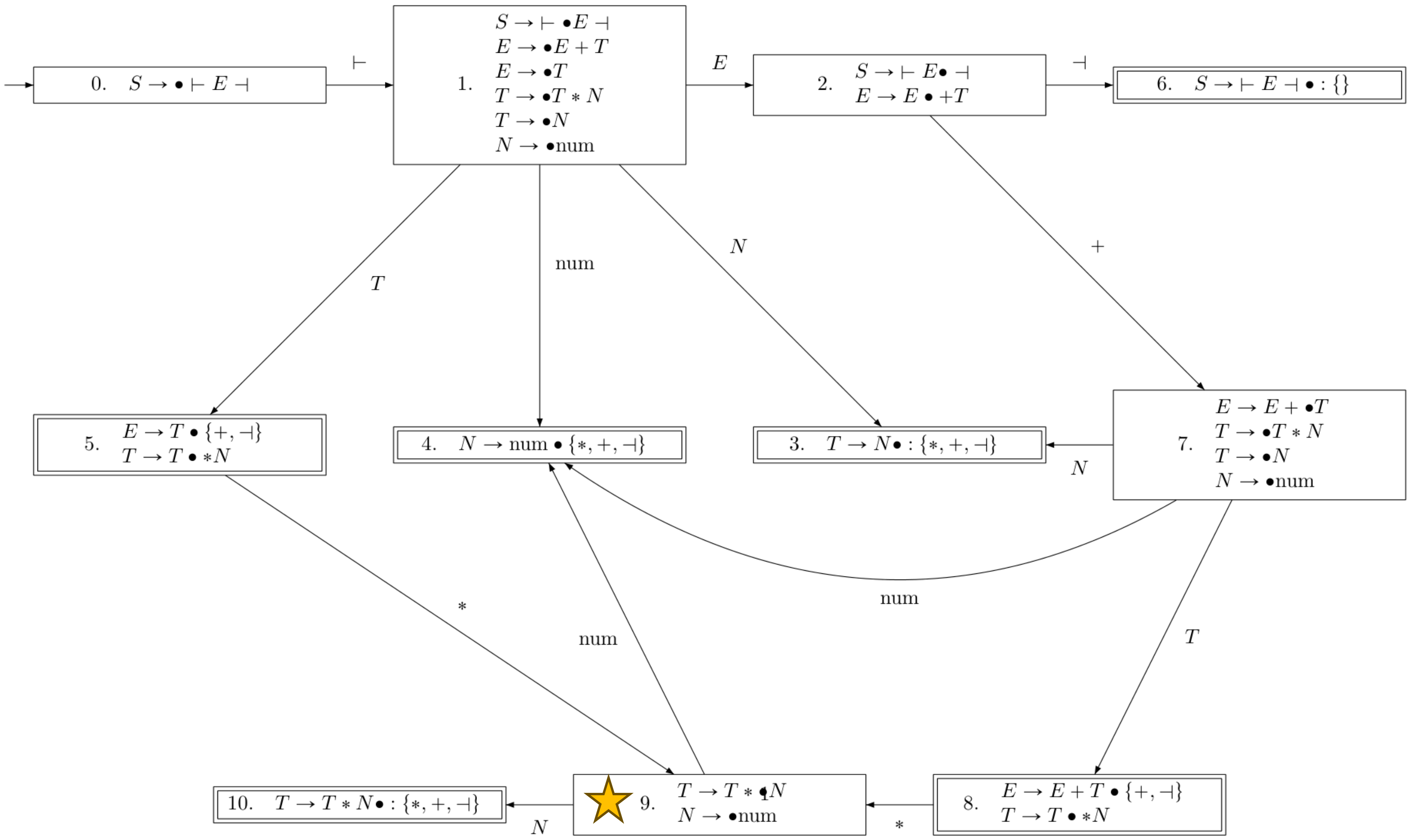
Tree Stack Top



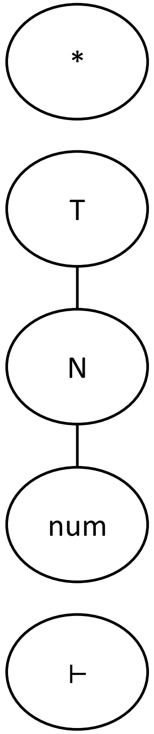
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 5 9



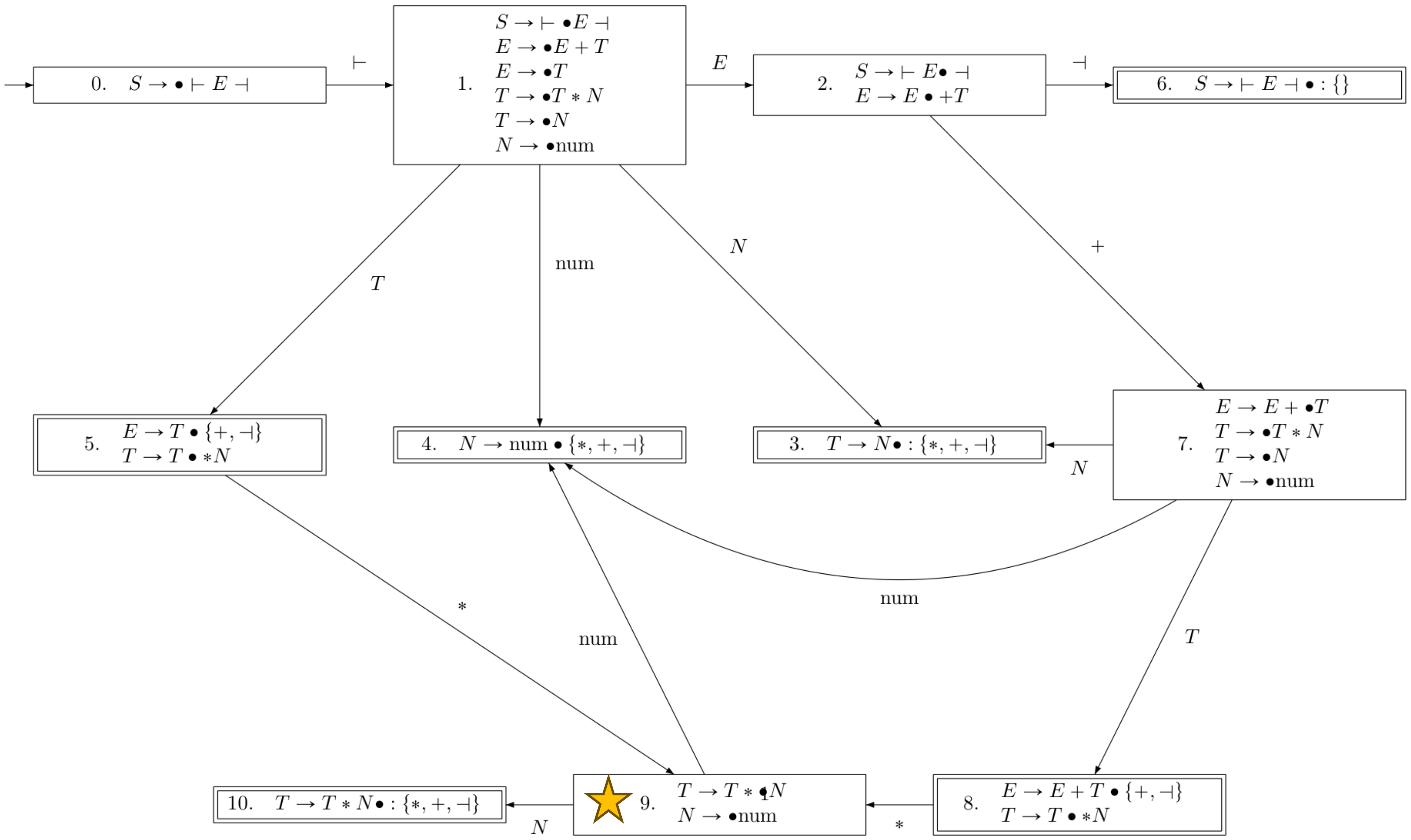
Tree Stack Top



Tree Stack Bottom

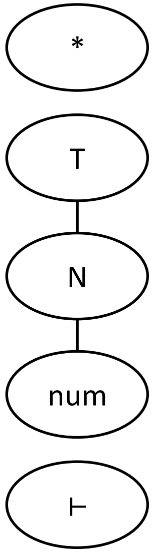
⊢ num * num ⊣

State Stack: 0 1 5 9



Tree Stack Top

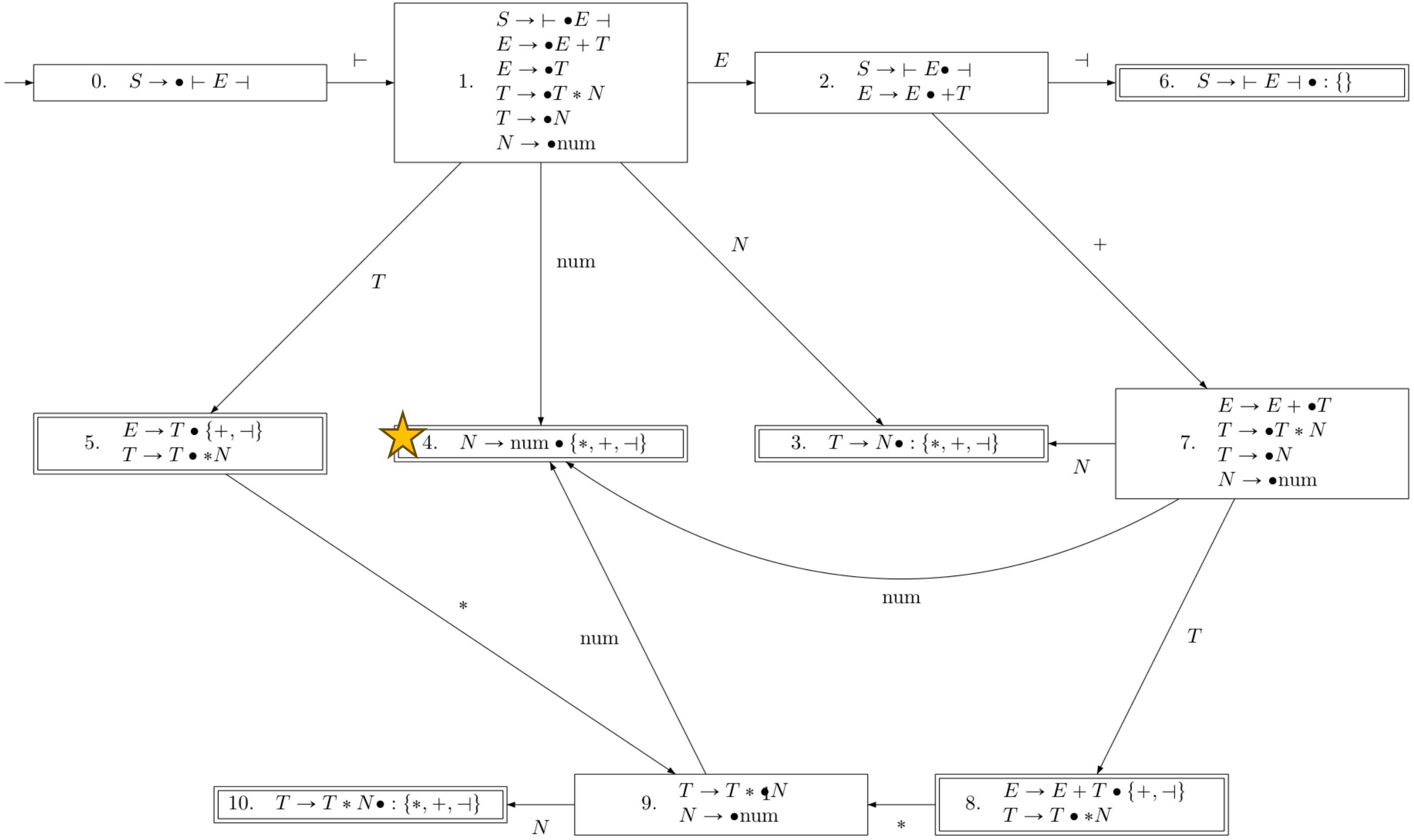
(No change to stack contents, just squishing it to fit more trees on the slide)



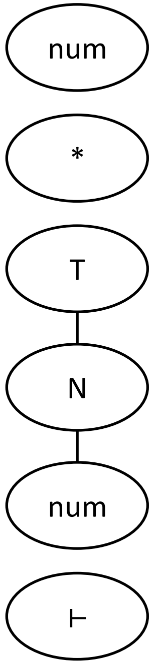
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 5 9 4



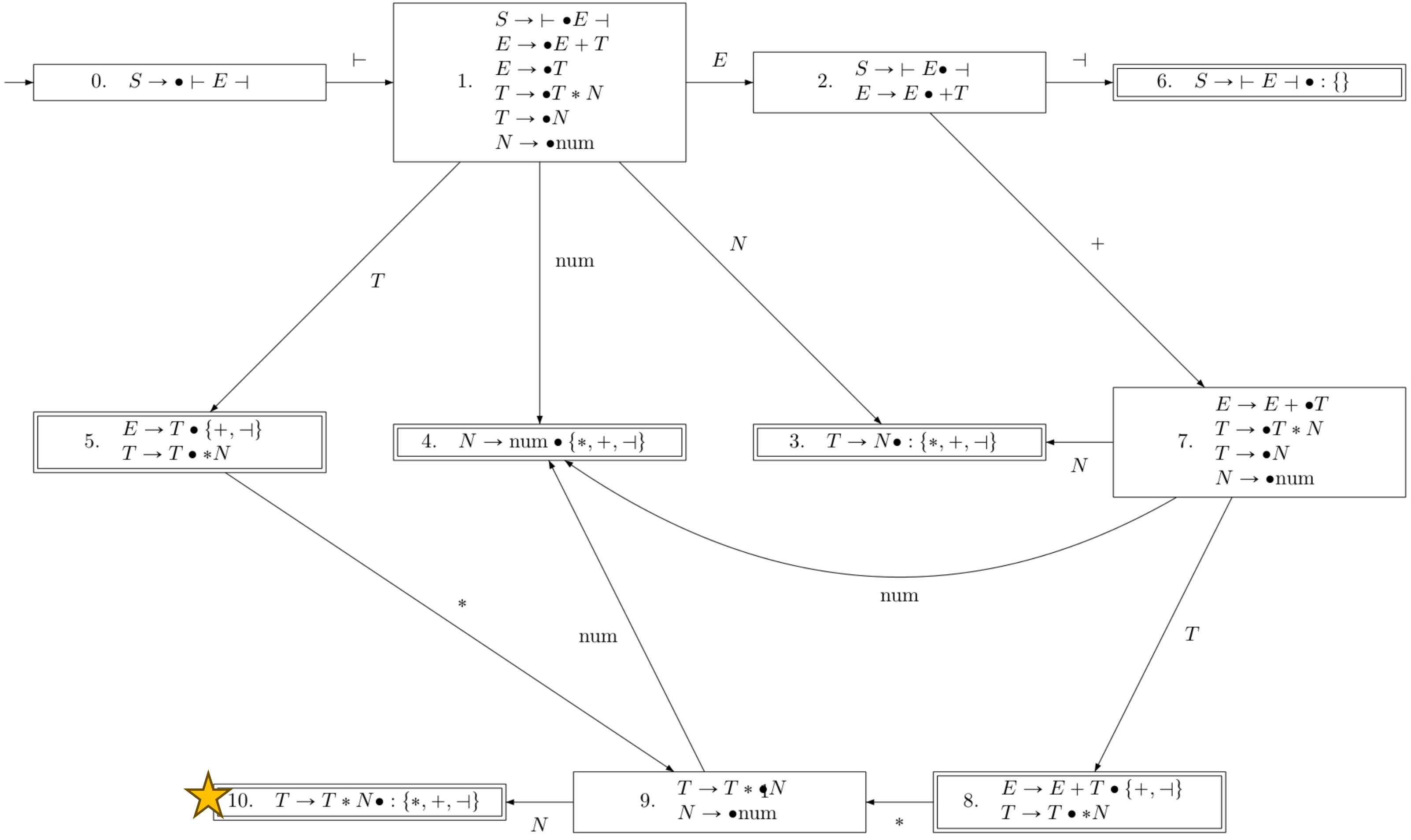
Tree Stack Top



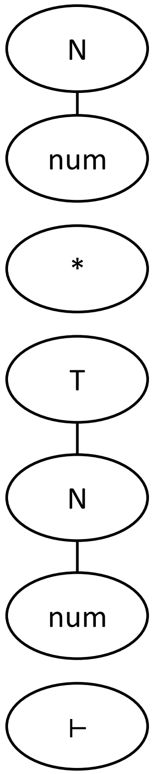
Tree Stack Bottom

$\underline{\vdash \text{num} * \text{num}}$ $\dashv$

State Stack: 0 1 5 9 10



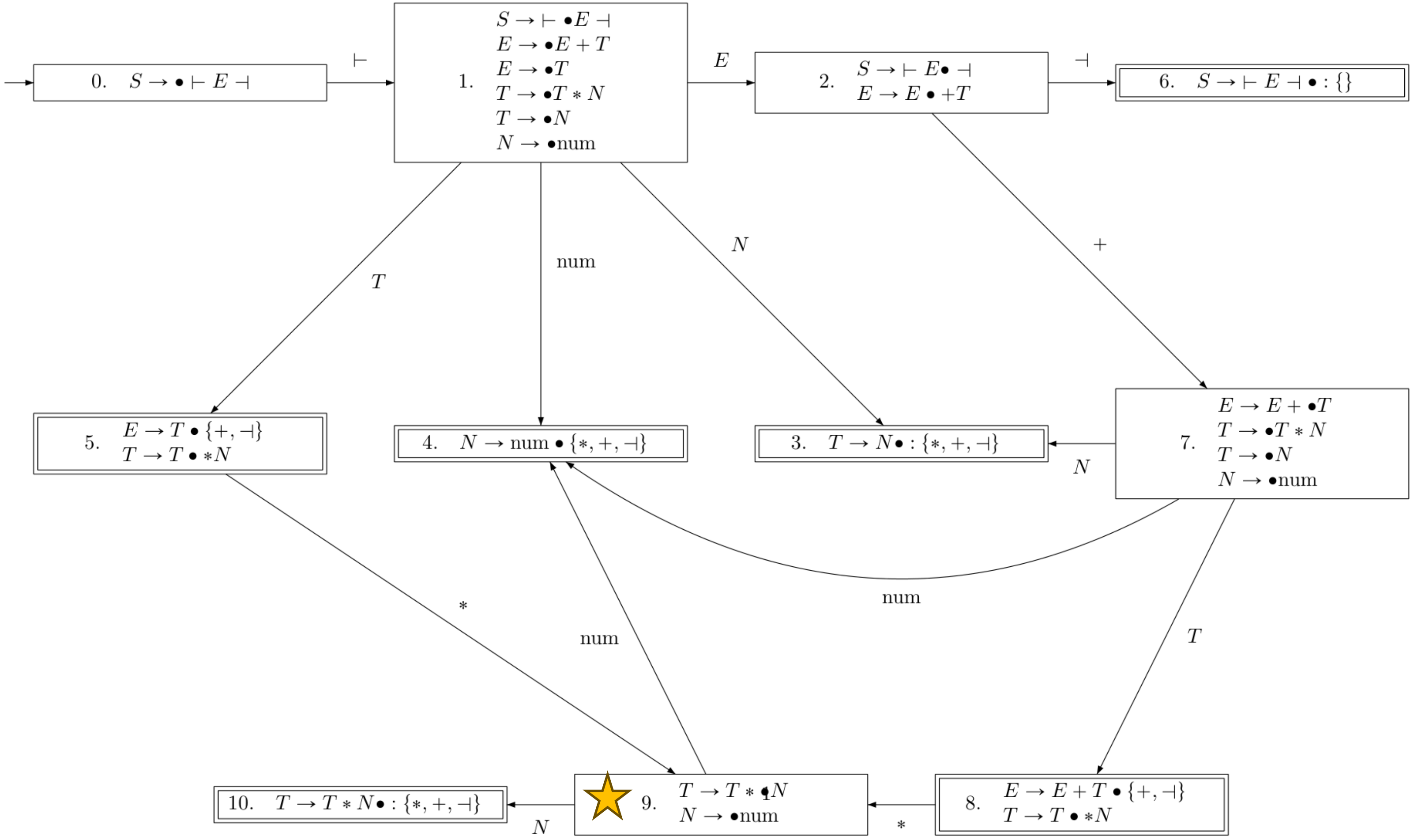
Tree Stack Top



Tree Stack Bottom

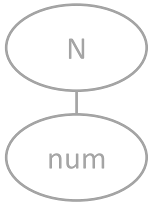
⊢ num * num ⊣

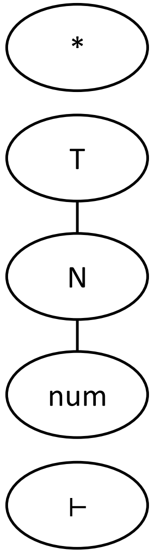
State Stack: 0 1 5 9



Tree Stack Top

Reducing by T → T * N:
Pop N / 10

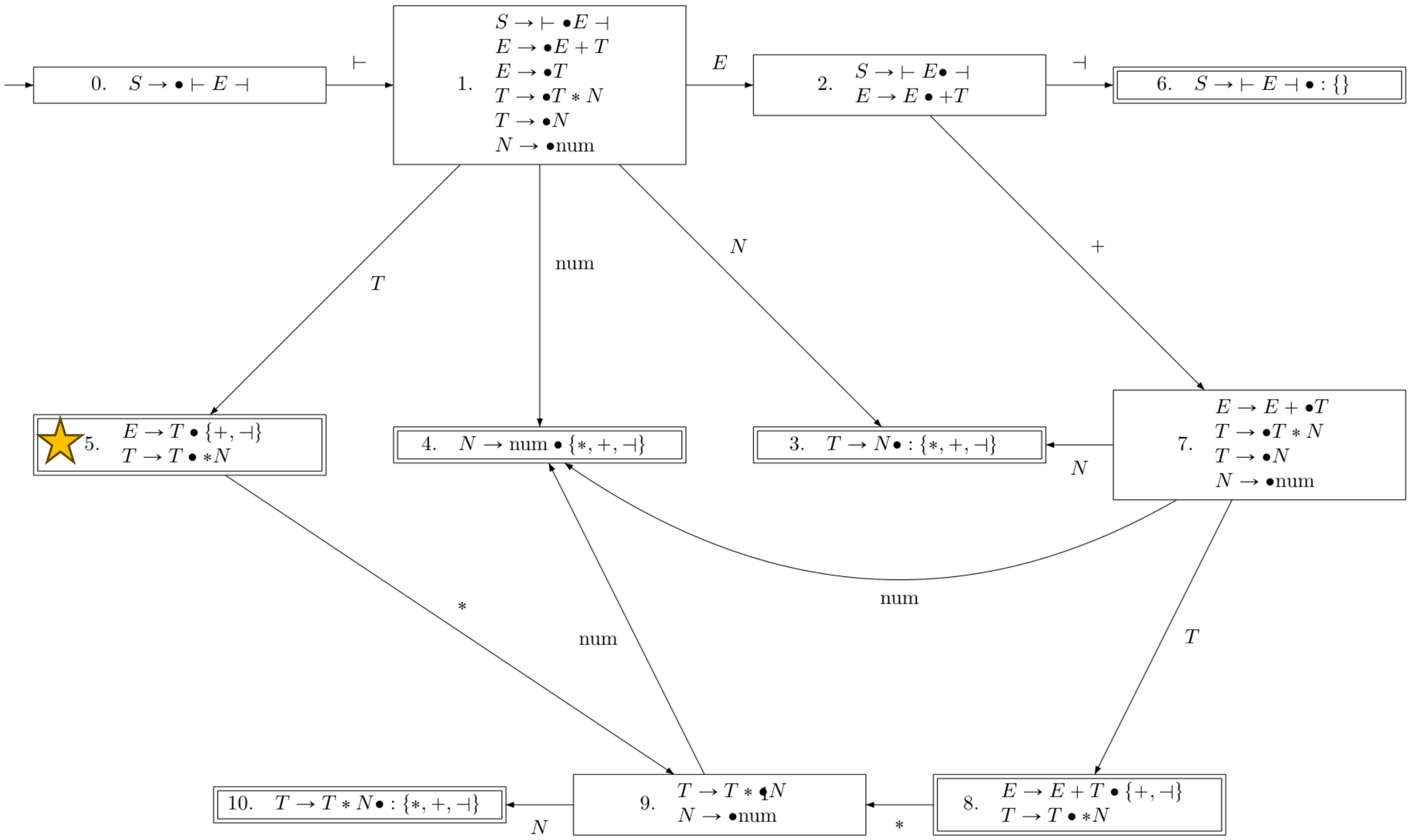




Tree Stack Bottom

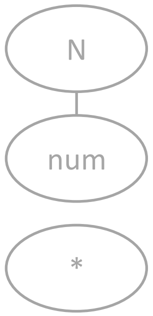
⊢ num * num ⊣

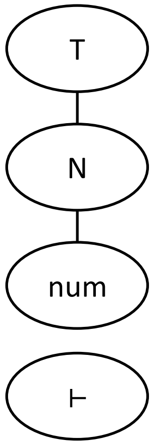
State Stack: 0 1 5



Tree Stack Top

Reducing by T → T * N:
Pop * / 9

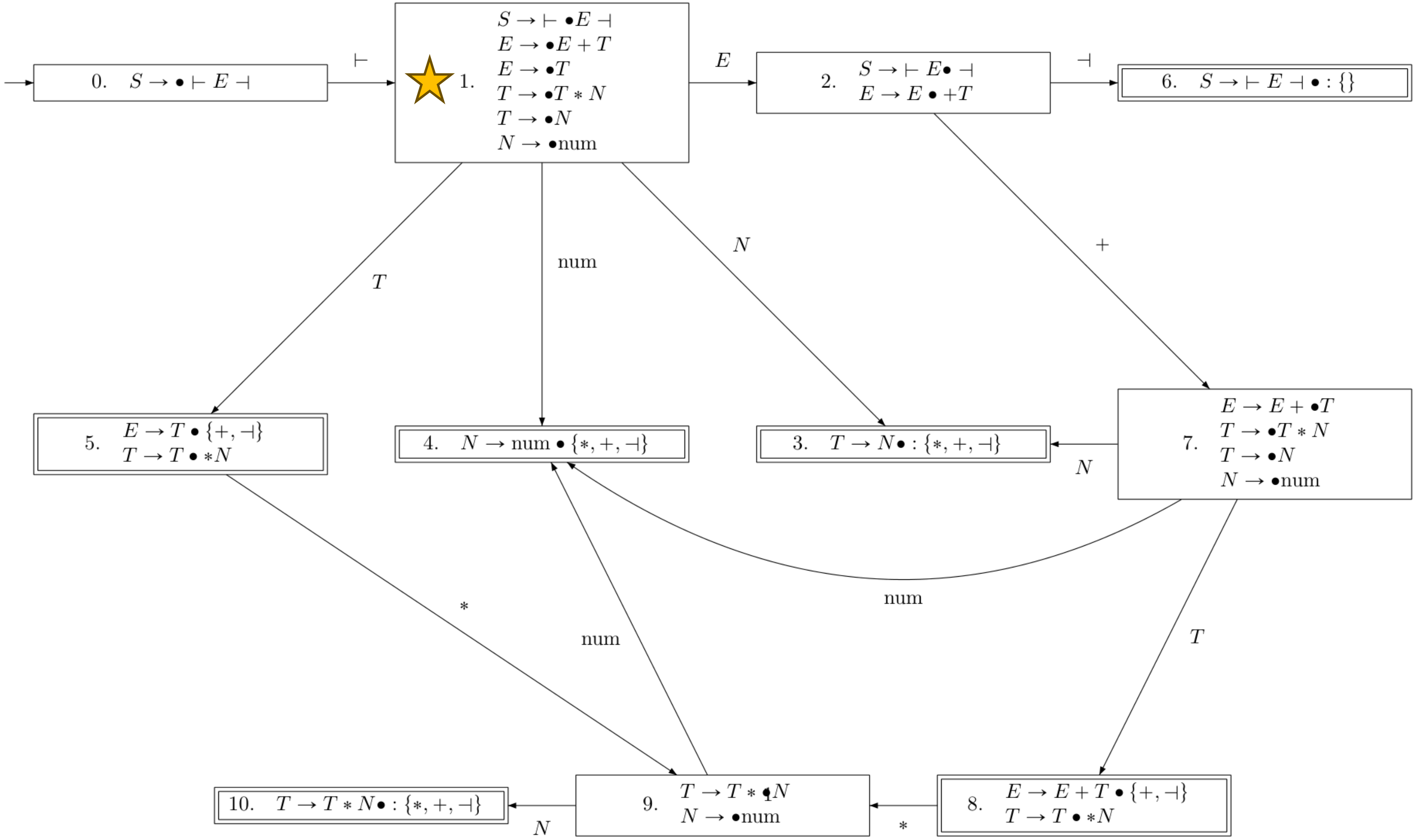




Tree Stack Bottom

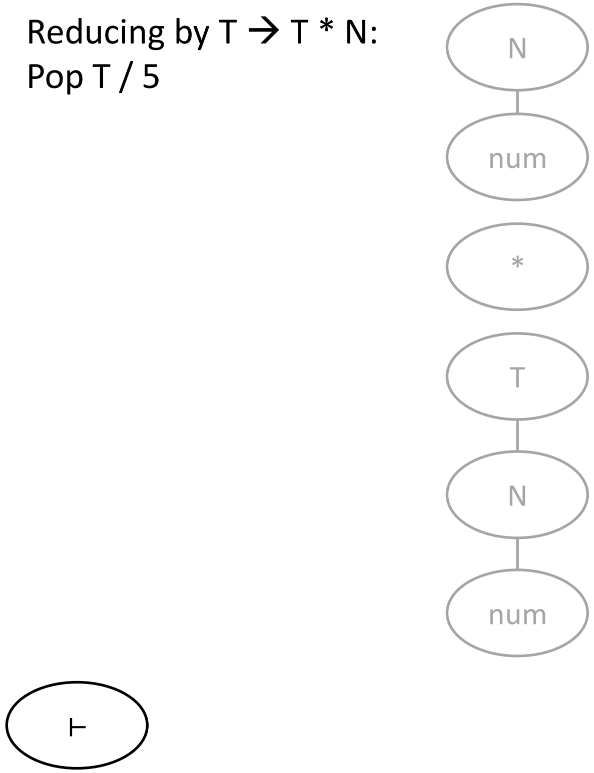
⊢ num * num ⊣

State Stack: 0 1



Tree Stack Top

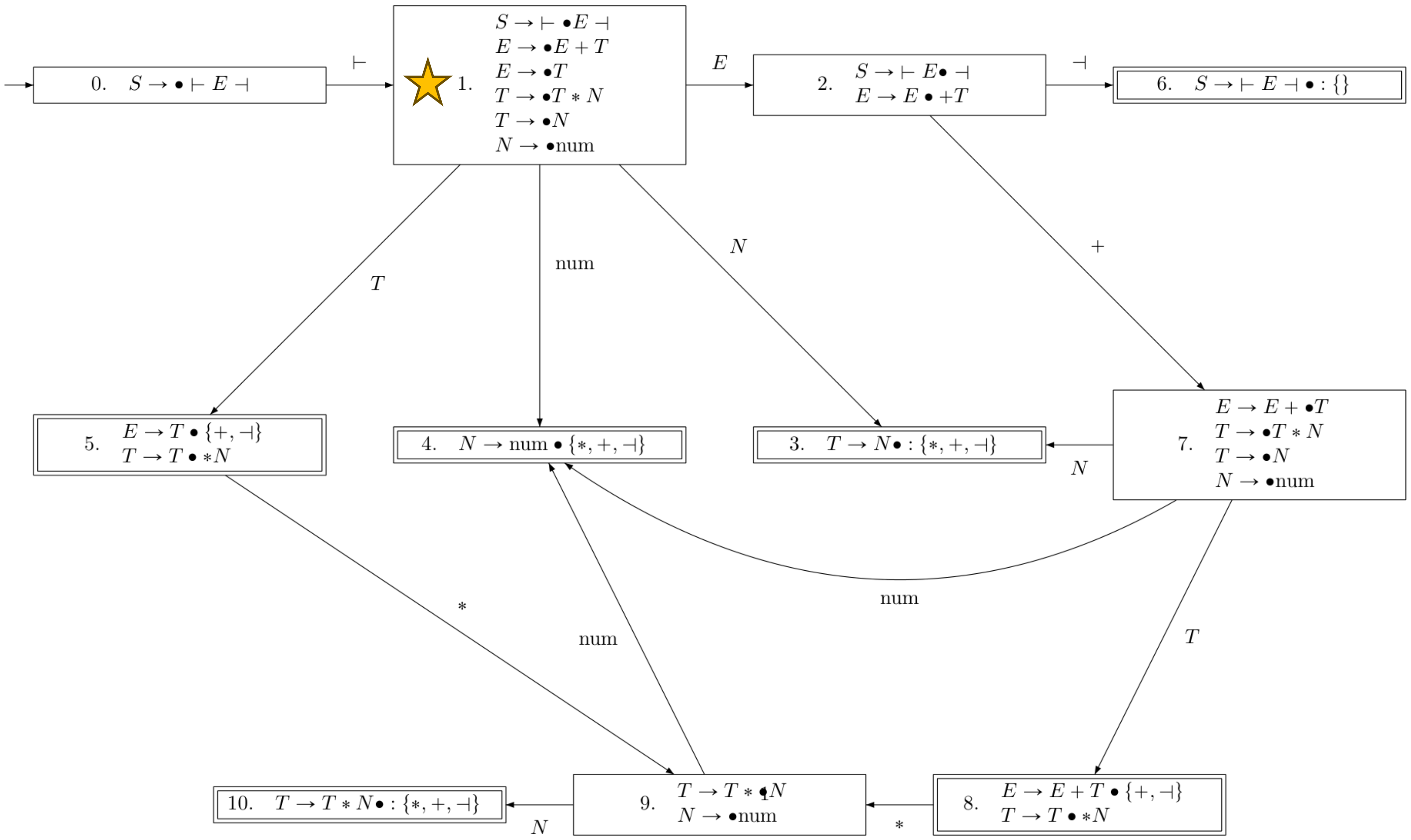
Reducing by T → T * N:
Pop T / 5



Tree Stack Bottom

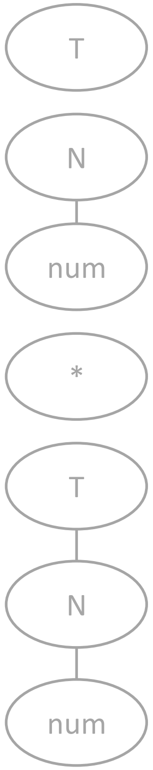
⊢ num * num ⊣

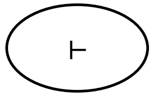
State Stack: 0 1



Tree Stack Top

Reducing by T → T * N:
Create node for T

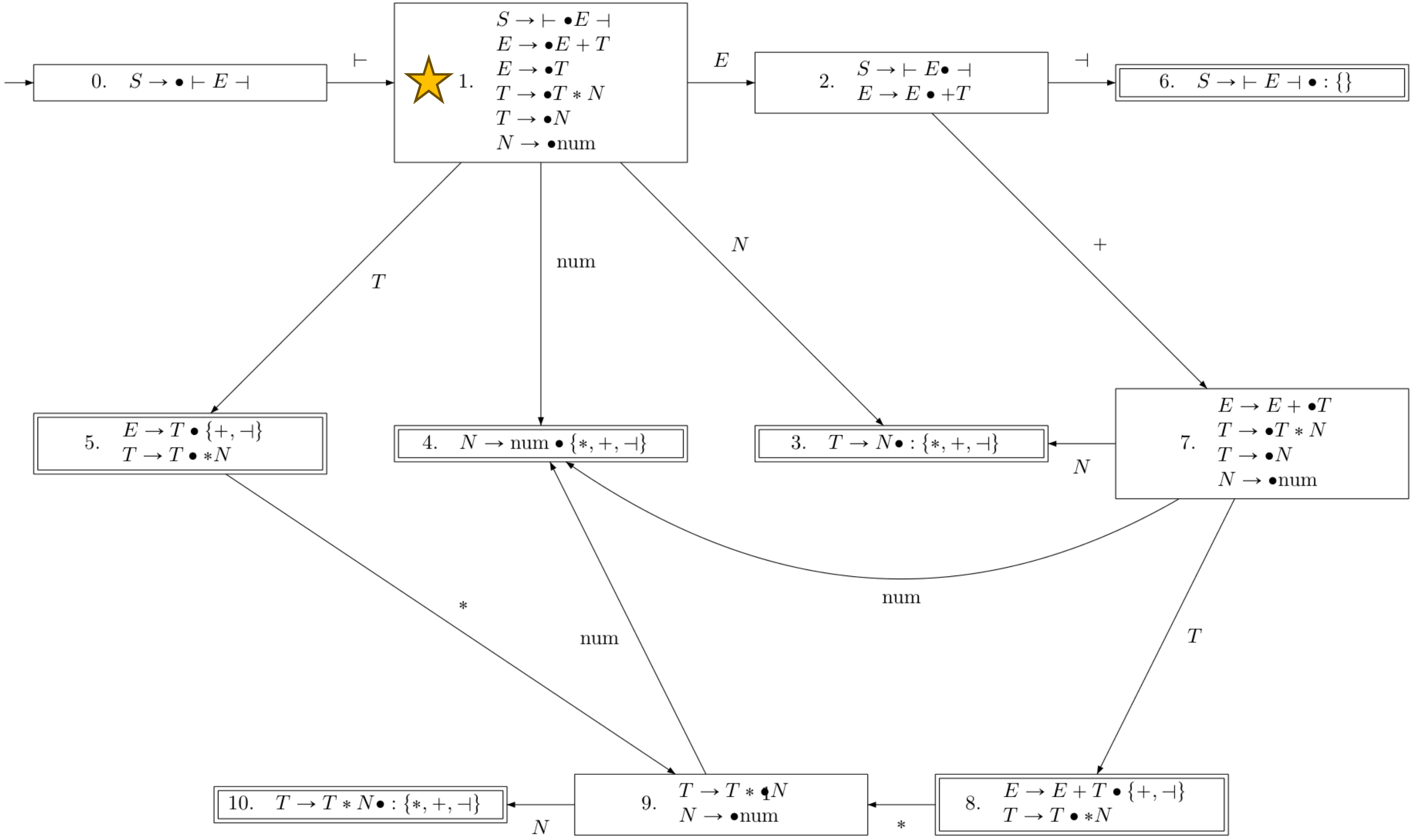




Tree Stack Bottom

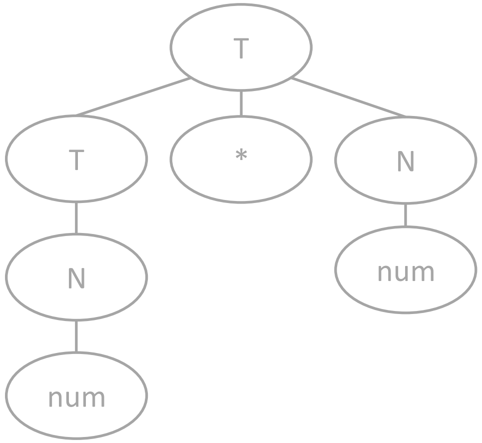
⊢ num * num ⊣

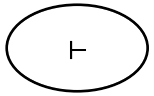
State Stack: 0 1



Tree Stack Top

Reducing by T → T * N:
Add T * N as children

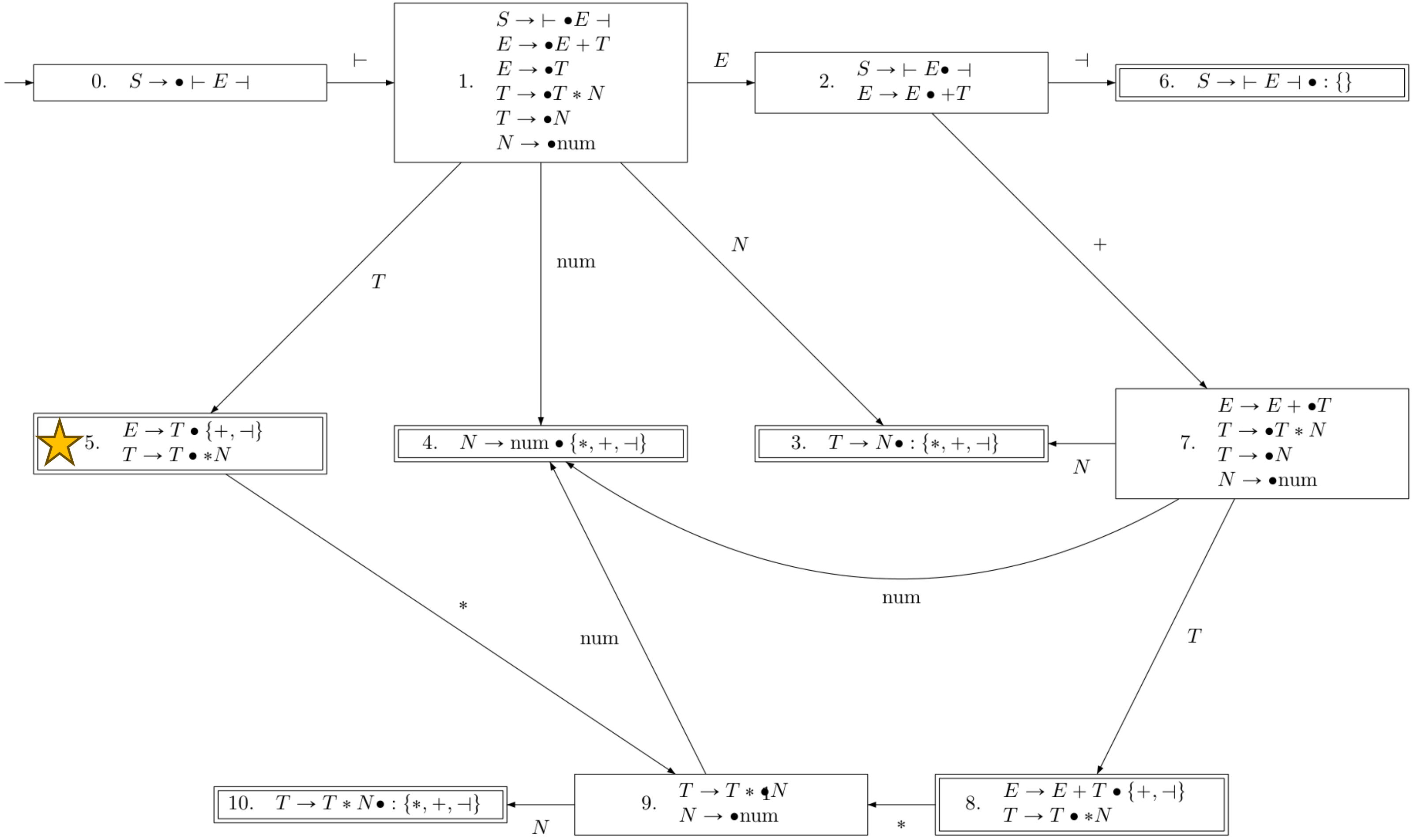




Tree Stack Bottom

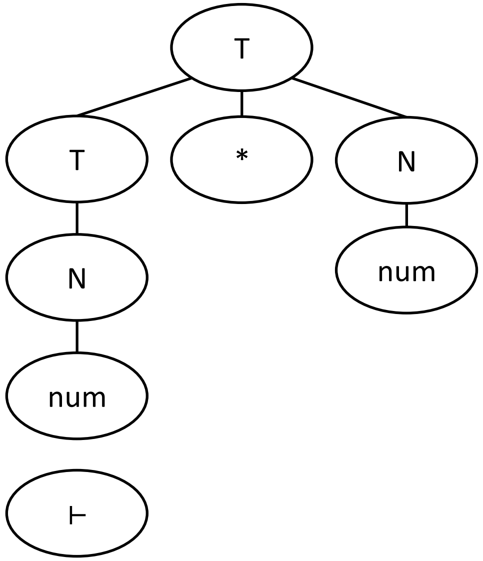
⊢ num * num ⊣

State Stack: 0 1 5



Tree Stack Top

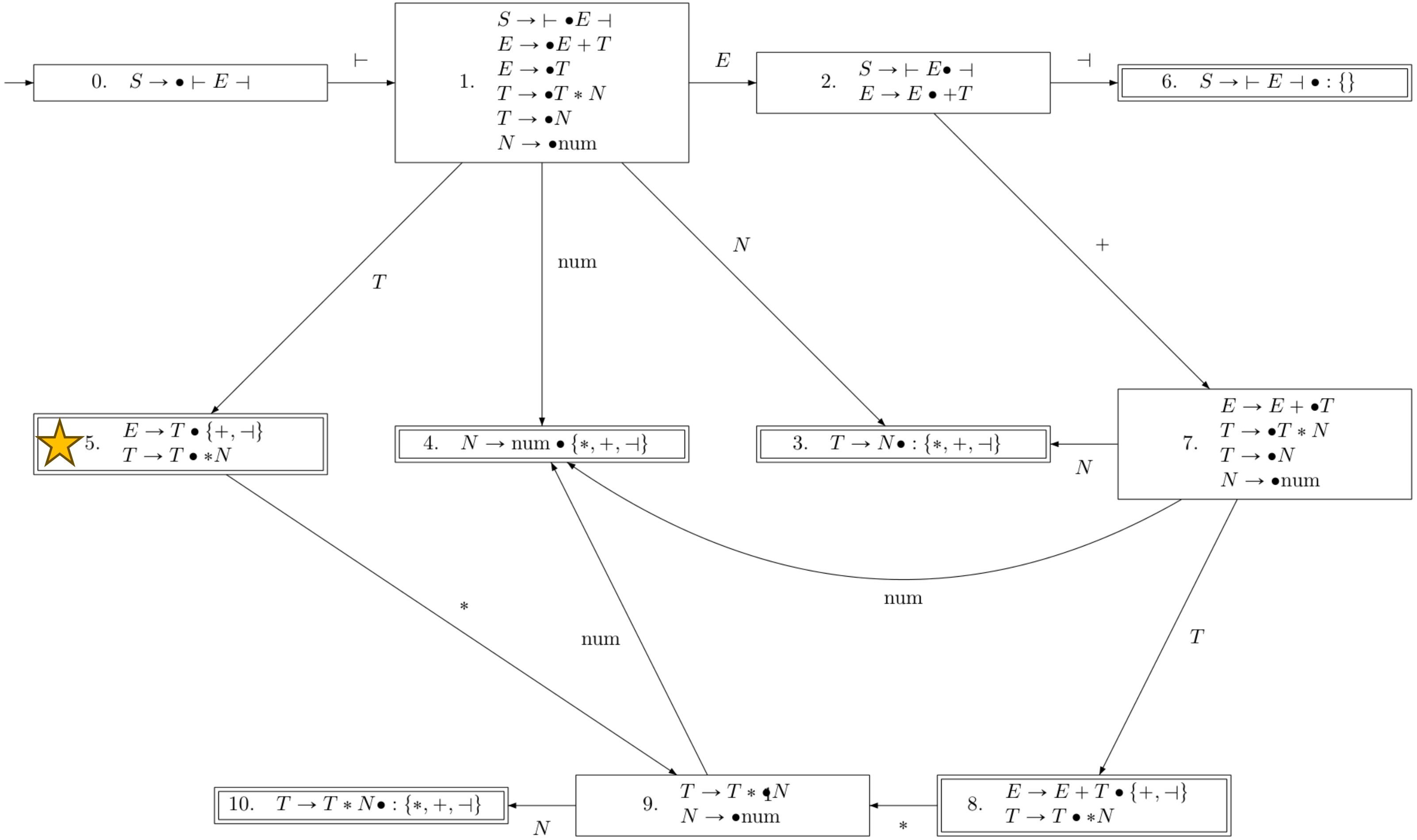
Reducing by T → T * N:
Push T / 5



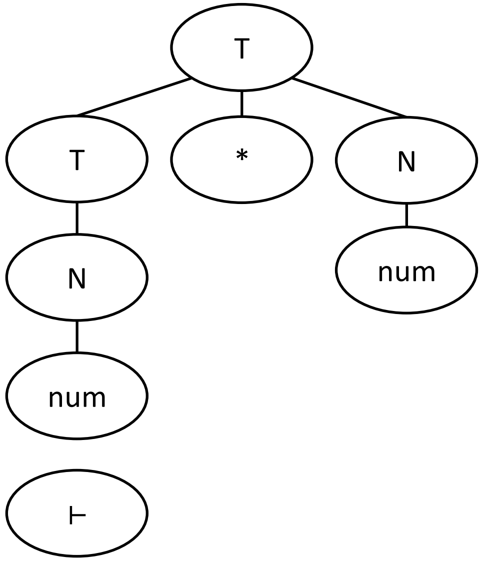
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 5



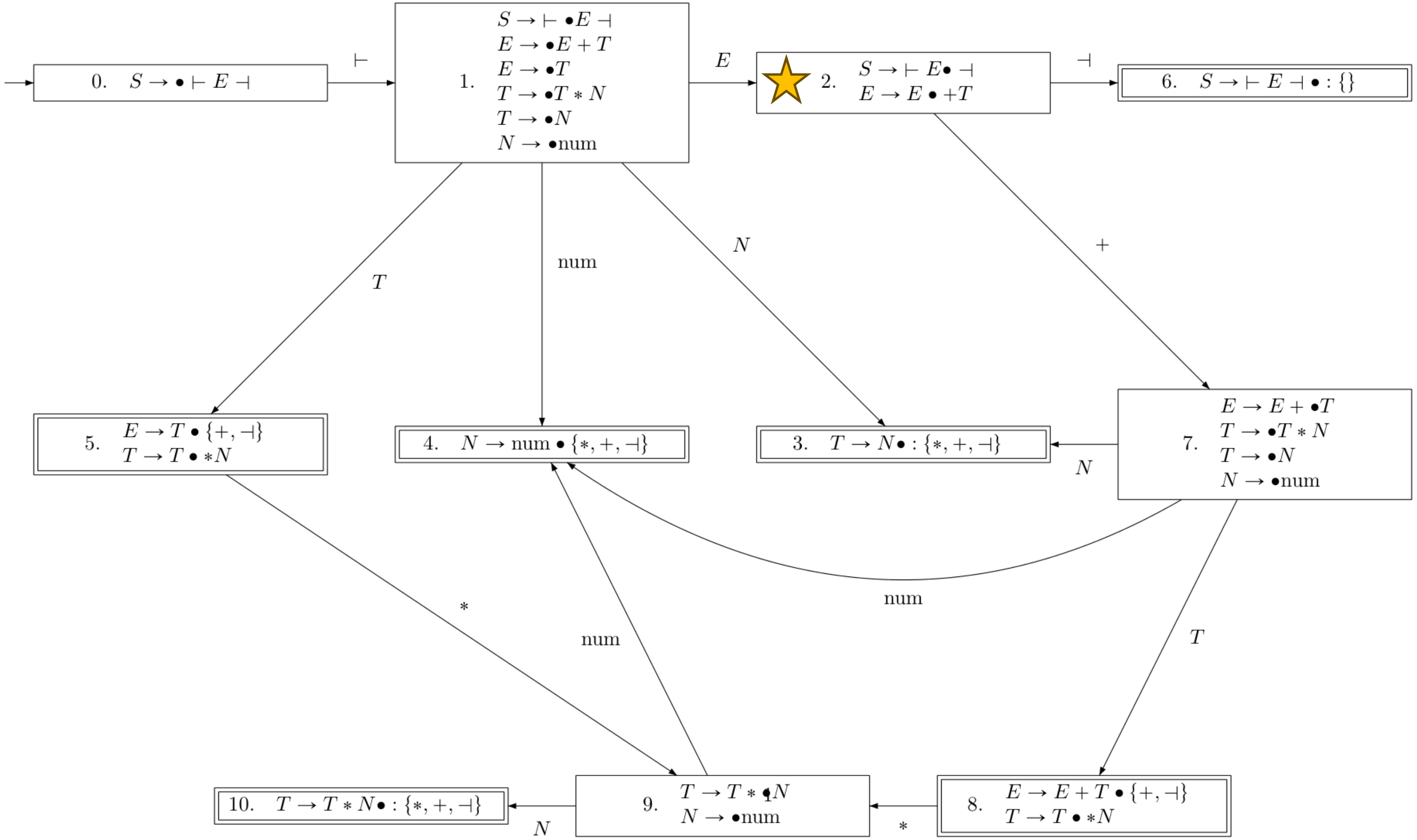
Tree Stack Top



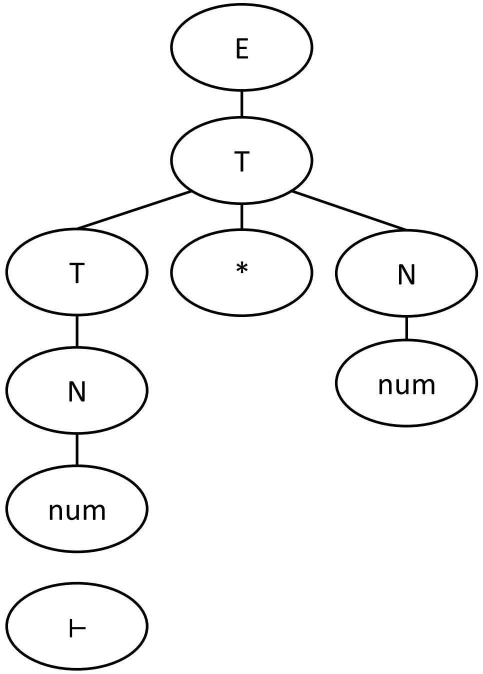
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 2



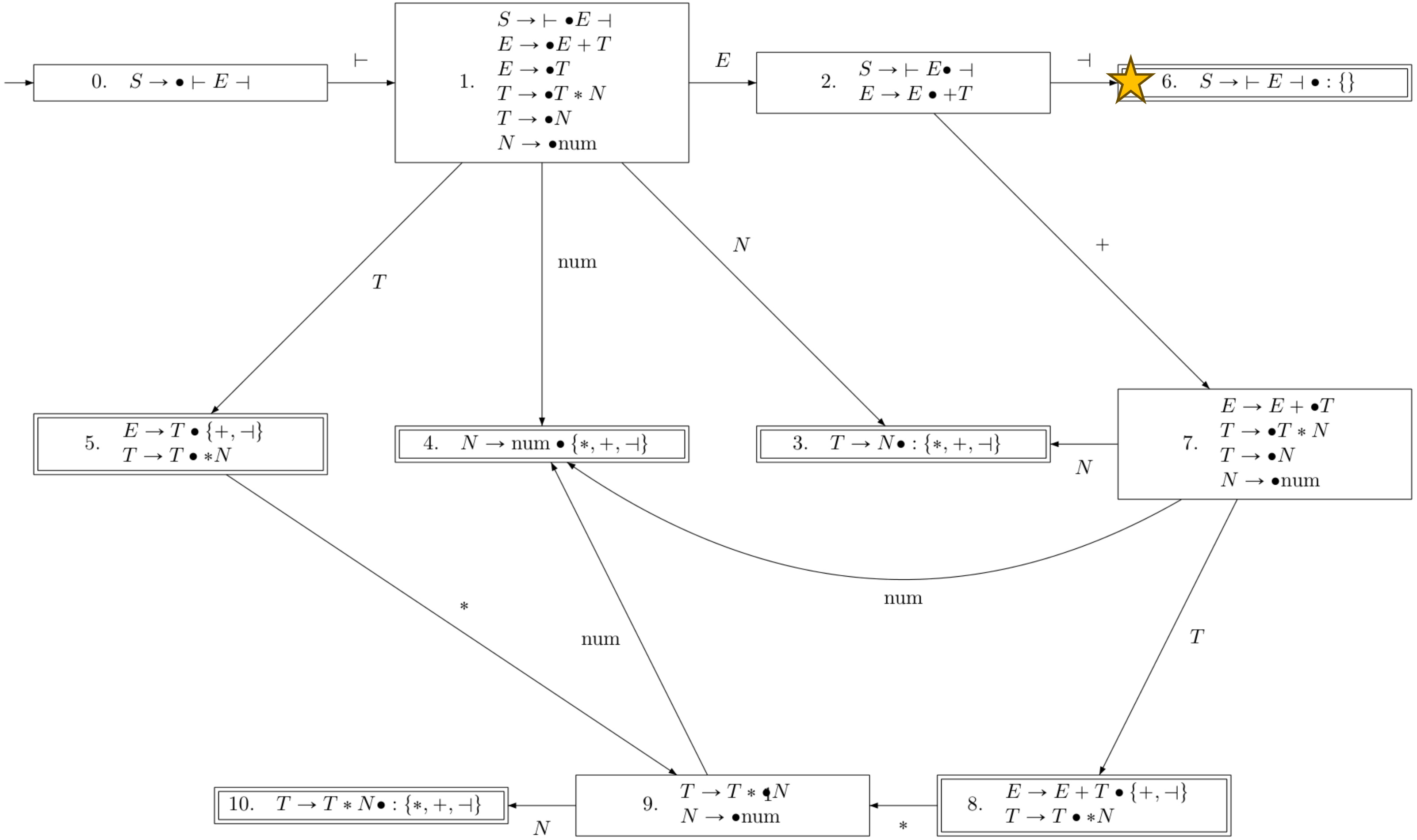
Tree Stack Top



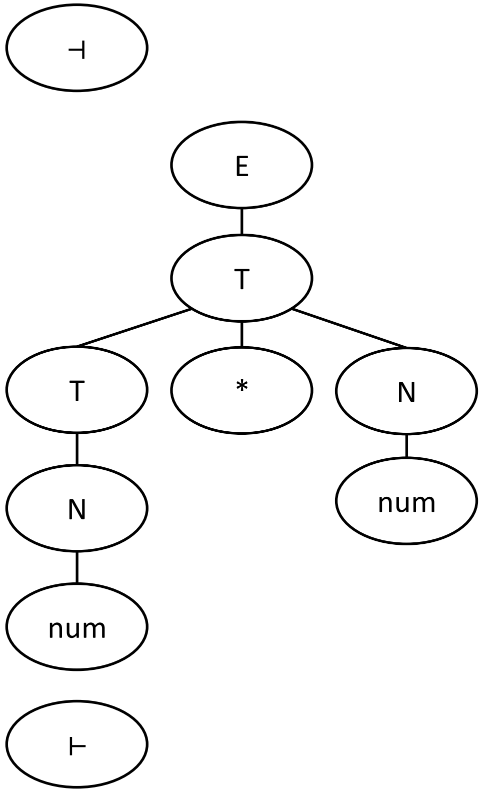
Tree Stack Bottom

⊢ num * num ⊣

State Stack: 0 1 2 6

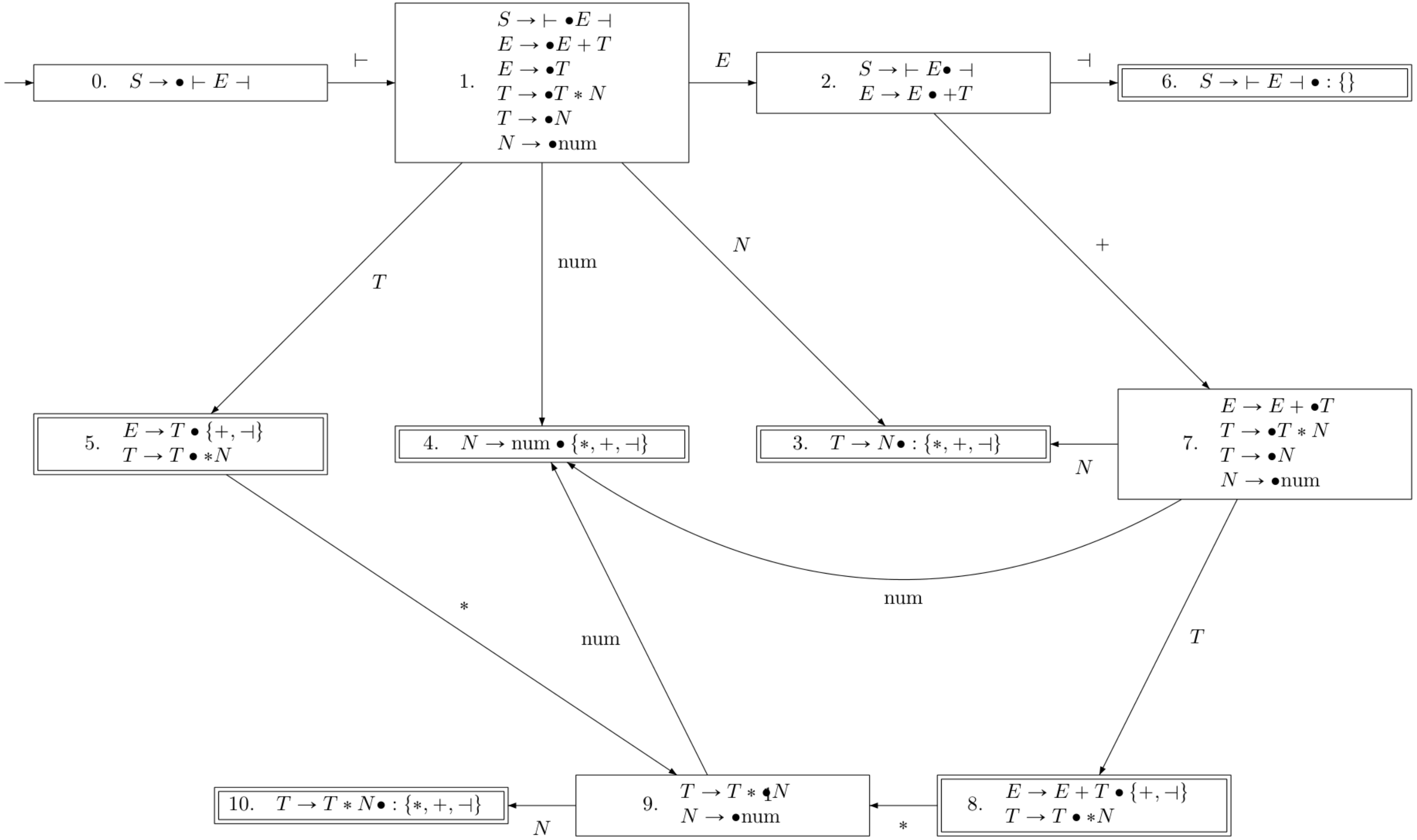


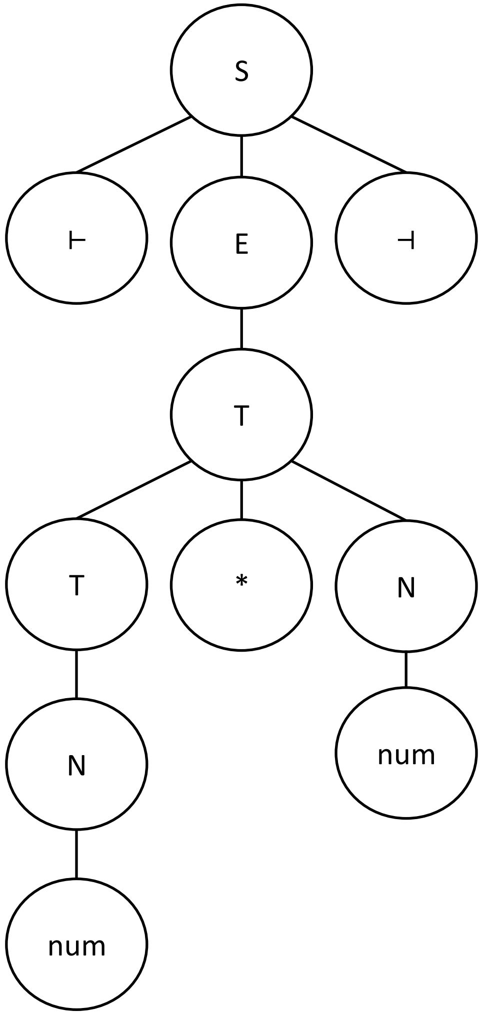
Tree Stack Top



Tree Stack Bottom

⊢ num * num ⊣


0. $S \to \bullet \vdash E \dashv$
$\vdash$
1. $S \to \vdash \bullet E \dashv$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * N$
$T \to \bullet N$
$N \to \bullet \text{num}$
E
2. $S \to \vdash E \bullet \dashv$
$E \to E \bullet + T$
$\dashv$
6. $S \to \vdash E \dashv \bullet : \{\}$
T
num
N
+
5. $E \to T \bullet \{+, \dashv\}$
$T \to T \bullet * N$
4. $N \to \text{num} \bullet \{*, +, \dashv\}$
3. $T \to N \bullet : \{*, +, \dashv\}$
N
7. $E \to E + \bullet T$
$T \to \bullet T * N$
$T \to \bullet N$
$N \to \bullet \text{num}$
num
*
num
T
10. $T \to T * N \bullet : \{*, +, \dashv\}$
N
9. $T \to T * \bullet N$
$N \to \bullet \text{num}$
*
8. $E \to E + T \bullet \{+, \dashv\}$
$T \to T \bullet * N$


S
⊢
E
⊣
T
T
*
N
N
num
num

Semantic Analysis

Also known as Context-Sensitive Analysis

The Stages of Compilation

- The compilation process can be broadly divided into four stages.
  - **Scanning:** Group the individual characters in the source into meaningful chunks called tokens, and detect errors related to syntax of tokens.
  - **Parsing:** Group the tokens into meaningful high-level structures like statements and expressions, and detect errors related to syntax of structures.
  - **Semantic Analysis:** Gather further information about the semantics (meaning) of the program, e.g. scope of identifiers and types of expressions, and detect errors related to semantics.
    - The program should be free of compile-time errors after this stage.
  - **Code Generation:** Translate each structural component of the program into the target language using the information obtained in the previous stages.

The Stages of Compilation

- The compilation process can be broadly divided into four stages.
  - **Scanning:** Group the individual characters in the source into meaningful chunks called tokens, and detect errors related to syntax of tokens.
  - **Parsing:** Group the tokens into meaningful high-level structures like statements and expressions, and detect errors related to syntax of structures.
  - **Semantic Analysis:** Gather further information about the semantics (meaning) of the program, e.g. scope of identifiers and types of expressions, and detect errors related to semantics.
    - The program should be free of compile-time errors after this stage.
  - **Code Generation:** Translate each structural component of the program into the target language using the information obtained in the previous stages.

The WLP4 Programming Language

- WLP4 (Waterloo Language Plus Pointers Plus Procedures) is the programming language we are writing a compiler for in this course.
- It is a (very small) subset of C++ that includes the following:
    - Variables of int (32-bit signed integer) or int* (pointer to int) type
    - Arithmetic expressions with brackets and the operations: + - * / %
    - Printing the value of an int variable
    - If/else statements and while loops, with conditions using the comparison operators: == != < > <= >=
    - Null pointers, pointer operations (dereference/address-of), pointer arithmetic
    - Dynamic memory allocation for int arrays (new/delete)
    - Procedures that take any amount/type of arguments and return an int value (and a special "wain" procedure which works like the C/C++ "main" function)

Semantic Errors in WLP4

- The semantic errors one needs to check for depend on the language.
- Many errors broadly fall into one of two categories.
- **Name errors** are errors related to identifiers and their meanings.
  - A name is used but a definition of the name cannot be found.
  - A name is defined multiple times and there is no way to disambiguate.
- **Type errors** are errors related to the types of expressions.
  - Adding two integers is valid, but adding two pointers is invalid.
  - Calling "delete" on an expression that is not a pointer is invalid.
  - If a procedure expects an integer parameter, passing a pointer is invalid.

Detecting Semantic Errors

- To parse programming languages, we had to move from regular languages to the wider class of context-free languages.
- Technically, there is a class called **context-sensitive languages** that we could use to describe semantically correct programs.
- Semantic analysis is sometimes called **context-sensitive analysis**.
- However, writing context-sensitive grammars and context-sensitive parsers is difficult and nobody does it.
- It is much easier to just analyze the **parse tree** obtained from the parsing phase than to approach this in a language-theoretic way.

Working with Parse Trees

- You can tell what kind of feature or aspect of the program you are looking at by examining the rule that defines the parse tree node.
- For example, the rule for the main (wain) function looks like:

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The rule for a while loop looks like:

statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

- When drawing parse trees, we usually just draw one symbol (terminal or nonterminal) in each node.
- Project 3 asks you to store the corresponding **CFG rule** in each parse tree node that corresponds to a nonterminal.