Limitations of LL(1) Parsing

Limitations of LL(1)

- A grammar is **LL(1)** if every cell of the Predict table contains at most one rule.
- We saw that this ambiguous expression grammar is not LL(1):

$$E \rightarrow E + E \qquad E \rightarrow 3$$

- Ambiguous grammars are *never* LL(1) because the Predict table attempts to include all rules that "could work" in some context.
- If a grammar is ambiguous, there must be a context where two distinct rules would both be valid to use in a leftmost derivation.
- Are there unambiguous grammars that are not LL(1)?

Limitations of LL(1)

- In the previous module, we developed this unambiguous grammar for arithmetic expressions with addition, subtraction, multiplication, division, brackets, and variables.

$$L_2 \rightarrow L_2 + L_1 \mid L_2 - L_1 \mid L_1$$

$$L_1 \rightarrow L_1 * L_0 \mid L_1 / L_0 \mid L_0$$

$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- This is **not LL(1)**. It has the same problem as the ambiguous grammar.
  - If L_2 is on the stack and "a" is the next symbol of input, we don't know whether to apply $L_2 \rightarrow L_2 + L_1$ or $L_2 \rightarrow L_1$.

Limitations of LL(1)

- In the previous module, we developed this unambiguous grammar for arithmetic expressions with addition, subtraction, multiplication, division, brackets, and variables.

$$L_2 \rightarrow L_2 + L_1 \mid L_2 - L_1 \mid L_1$$

$$L_1 \rightarrow L_1 * L_0 \mid L_1 / L_0 \mid L_0$$

$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- Suppose the input string is **a + a** and we haven't read any input. We need to apply $L_2 \rightarrow L_2 + L_1$ then $L_2 \rightarrow L_1$ then $L_1 \rightarrow L_0$ then $L_0 \rightarrow a$ before we can finally read the first **a**.

Limitations of LL(1)

- In the previous module, we developed this unambiguous grammar for arithmetic expressions with addition, subtraction, multiplication, division, brackets, and variables.

$$L_2 \rightarrow L_2 + L_1 \mid L_2 - L_1 \mid L_1$$
$$L_1 \rightarrow L_1 * L_0 \mid L_1 / L_0 \mid L_0$$
$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- Suppose the input string is **a + a + a** and we haven't read any input. We need to apply $L_2 \rightarrow L_2 + L_1$ **twice** before $L_2 \rightarrow L_1$. But the context is the same as before (L_2 on the stack and "a" at the front of input).

Left Recursion

- This is actually an inherent problem with **left recursion** in grammars.

$$A \rightarrow A\alpha \mid \beta$$

- This grammar can derive any string of the form $\beta\alpha\alpha\ldots\alpha$ ($\beta\alpha^*$)
- The predictor has to figure out how many times to apply $A \rightarrow A\alpha$, but the only information it has is the symbols at the start of β.
- In fact, this cannot be handled by **LL(k) parsing** (up to k symbols of lookahead) for any k, because to determine how many α's there are, we potentially need to look at the entire string!
- This is a problem because left recursion is used for left associativity!

Removing Left Recursion

- Removing left recursion from a grammar might mess up our parse tree (e.g., arithmetic operations would no longer be left associative).
- Nonetheless, we can consider the idea of changing the grammar so LL(1) will work, and then somehow fixing the parse tree later.
- If we use right recursion instead, is the grammar LL(1)?

$$L_2 \rightarrow L_1 + L_2 \mid L_1 - L_2 \mid L_1$$
$$L_1 \rightarrow L_0 * L_1 \mid L_0 / L_1 \mid L_0$$
$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- No. For example, consider **a** vs **a + a**. The first rule to apply can't be predicted just by looking at "a".

Removing Left Recursion

- Removing left recursion from a grammar might mess up our parse tree (e.g., arithmetic operations would no longer be left associative).
- Nonetheless, we can consider the idea of changing the grammar so LL(1) will work, and then somehow fixing the parse tree later.
- If we use right recursion instead, is the grammar LL(1)?

$$L_2 \rightarrow L_1 + L_2 \mid L_1 - L_2 \mid L_1$$
$$L_1 \rightarrow L_0 * L_1 \mid L_0 / L_1 \mid L_0$$
$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- No. For example, consider **a** vs **a + a**. The first rule to apply can't be predicted just by looking at "a". However, this is LL(2).

Left Factoring

- Given a grammar with "direct" left recursion:

$$A \rightarrow A\alpha \mid \beta$$

- We can remove left recursion as follows:
  - Introduce a new nonterminal A'.
  - Replace these rules with $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$.
- But this might not produce an LL(1) grammar.
- Consider $A \rightarrow Aab \mid Aac \mid d$.
- We could transform this into $A \rightarrow dA'$, $A' \rightarrow abA' \mid acA' \mid \varepsilon$.
- How do we tell whether to apply $A' \rightarrow abA'$ or $A' \rightarrow acA'$ if the next symbol is a? (Would need 2 lookaheads)

Left Factoring

- If multiple rules with the same left hand side have a *common (non-empty) prefix* on the right hand side, the grammar is not LL(1).

$$A' \rightarrow \underline{a}bA' \qquad A' \rightarrow \underline{a}cA'$$

- **Left factoring** can be used to resolve this.
- If a grammar has a collection of rules with a common left hand since A, and a common right hand side prefix α, as follows:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$$

- Introduce a new nonterminal A' and replace these rules with:

$$A \rightarrow \alpha A' \qquad A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Left Factoring: Example

- Take our right-recursive expression grammar:

$$L_2 \rightarrow L_1 + L_2 \mid L_1 - L_2 \mid L_1$$
$$L_1 \rightarrow L_0 * L_1 \mid L_0 / L_1 \mid L_0$$
$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

- Left-factored version, which is LL(1):

$$L_2 \rightarrow L_1 L_2' \qquad L_2' \rightarrow + L_2 \mid - L_2 \mid \varepsilon$$
$$L_1 \rightarrow L_0 L_1' \qquad L_1' \rightarrow * L_1 \mid / L_1 \mid \varepsilon$$
$$L_0 \rightarrow a \mid b \mid c \mid (L_2)$$

The State of Things

- Left-recursive grammars, which we use for left-associative operations, are incompatible with LL(1) parsing.
- Even increasing the lookahead and using a more complicated "LL(k)" predict table would not solve this.
- We can convert the left recursion to right recursion, but this messes up our parse trees, and the resulting grammar isn't even always LL(1).
- We can sometimes use left factoring to get an LL(1) grammar, but this messes up our parse trees even more.
- Some languages do not permit an LL(1) grammar at all.

Our Solution

- We do not necessarily *need* to give up on top-down parsing.
- There are top-down parsers that overcome the issues we have encountered by using more ad-hoc techniques, as opposed to the formalism of LL(1) or LL(k).
- There are also other formal techniques that expand on LL parsing.
- However, we will instead explore the idea of **bottom-up parsing**.
- We will see that bottom-up parsers, while they are less intuitive, are able to handle left recursion in practical grammars without issues.
- We will ultimately use a bottom-up parser in our compiler.

Bottom-Up Parsing: First Steps

The Idea

- In top-down parsing, we begin the derivation from the **start symbol**.
- At each step, we either **match a terminal** (read input) or **apply a rule** (progress our derivation) until we derive the target string.
- In bottom-up parsing, we find a **reverse derivation**, starting from the **target string** and working backwards to the start symbol.
- At each step, we either **shift a terminal** (read input) or **reduce by a rule** (progress our reverse derivation) until we reach the start symbol.
  - "Reduce" means to apply the rule "backwards": we take part of our current derivation that matches the right-hand side of the rule, and replace that part with the left-hand side.

Bottom-Up Parsing, Informally

`⊢ num * num * num ⊣`

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣        shift ⊢
⊢ num * num * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

`⊢ num * num * num ⊣`	`shift ⊢`
`⊢ num * num * num ⊣`	`shift num`
`⊢ num * num * num ⊣`	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣        shift ⊢
⊢ num * num * num ⊣        shift num
⊢ num * num * num ⊣        reduce N → num
⊢   N * num * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣      shift ⊢
⊢ num * num * num ⊣      shift num
⊢ num * num * num ⊣      reduce N → num
⊢   N * num * num ⊣      reduce T → N
⊢   T * num * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣        shift ⊢
⊢ num * num * num ⊣        shift num
⊢ num * num * num ⊣        reduce N → num
⊢   N * num * num ⊣        reduce T → N
⊢   T * num * num ⊣        shift *
⊢   T * num * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣        shift ⊢
⊢ num * num * num ⊣        shift num
⊢ num * num * num ⊣        reduce N → num
⊢   N * num * num ⊣        reduce T → N
⊢   T * num * num ⊣        shift *
⊢   T * num * num ⊣        shift num
⊢   T * num * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

	⊢ num * num * num ⊣	shift ⊢
⊢	num * num * num ⊣	shift num
⊢ num	* num * num ⊣	reduce N → num
⊢ N	* num * num ⊣	reduce T → N
⊢ T	* num * num ⊣	shift *
⊢ T *	num * num ⊣	shift num
⊢ T * num	* num ⊣	reduce N → num
⊢ T * N	* num ⊣	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

```
⊢ num * num * num ⊣        shift ⊢
⊢ num * num * num ⊣        shift num
⊢ num * num * num ⊣        reduce N → num
⊢   N * num * num ⊣        reduce T → N
⊢   T * num * num ⊣        shift *
⊢   T * num * num ⊣        shift num
⊢   T * num * num ⊣        reduce N → num
⊢   T *   N * num ⊣        reduce T → T * N
⊢         T * num ⊣
```

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

⊢ num * num * num ⊣	shift ⊢
⊢ num * num * num ⊣	shift num
⊢ num * num * num ⊣	reduce N → num
⊢ N * num * num ⊣	reduce T → N
⊢ T * num * num ⊣	shift *
⊢ T * num * num ⊣	shift num
⊢ T * num * num ⊣	reduce N → num
⊢ T * N * num ⊣	reduce T → T * N
⊢ T * num ⊣	shift *
⊢ T * num ⊣	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

	⊢ num * num * num ⊣	shift ⊢
⊢	num * num * num ⊣	shift num
⊢ num	* num * num ⊣	reduce N → num
⊢ N	* num * num ⊣	reduce T → N
⊢ T	* num * num ⊣	shift *
⊢ T *	num * num ⊣	shift num
⊢ T * num	* num ⊣	reduce N → num
⊢ T * N	* num ⊣	reduce T → T * N
⊢ T	* num ⊣	shift *
⊢ T *	num ⊣	shift num
⊢ T * num	⊣	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

⊢ num * num * num ⊣	shift ⊢
⊢ num * num * num ⊣	shift num
⊢ num * num * num ⊣	reduce N → num
⊢ N * num * num ⊣	reduce T → N
⊢ T * num * num ⊣	shift *
⊢ T * num * num ⊣	shift num
⊢ T * num * num ⊣	reduce N → num
⊢ T * N * num ⊣	reduce T → T * N
⊢ T * num ⊣	shift *
⊢ T * num ⊣	shift num
⊢ T * num ⊣	reduce N → num
⊢ T * N ⊣	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

	⊢ num * num * num ⊣	shift ⊢
⊢	num * num * num ⊣	shift num
⊢ num	* num * num ⊣	reduce N → num
⊢ N	* num * num ⊣	reduce T → N
⊢ T	* num * num ⊣	shift *
⊢ T *	num * num ⊣	shift num
⊢ T * num	* num ⊣	reduce N → num
⊢ T * N	* num ⊣	reduce T → T * N
⊢ T	* num ⊣	shift *
⊢ T *	num ⊣	shift num
⊢ T * num	⊣	reduce N → num
⊢ T * N	⊣	reduce T → T * N
⊢ T	⊣	

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

	⊢ num * num * num ⊣	shift ⊢
⊢	num * num * num ⊣	shift num
⊢ num	* num * num ⊣	reduce N → num
⊢ N	* num * num ⊣	reduce T → N
⊢ T	* num * num ⊣	shift *
⊢ T *	num * num ⊣	shift num
⊢ T * num	* num ⊣	reduce N → num
⊢ T * N	* num ⊣	reduce T → T * N
⊢ T	* num ⊣	shift *
⊢ T *	num ⊣	shift num
⊢ T * num	⊣	reduce N → num
⊢ T * N	⊣	reduce T → T * N
⊢ T	⊣	shift ⊣
⊢ T ⊣		

Bottom-Up Parsing, Informally

Here is a simple grammar for expressions with multiplication:

S → ⊢ T ⊣
T → T * N
T → N
N → num

Let's parse this string:
⊢ num * num * num ⊣

⊢ num * num * num ⊣	shift ⊢
⊢ num * num * num ⊣	shift num
⊢ num * num * num ⊣	reduce N → num
⊢ N * num * num ⊣	reduce T → N
⊢ T * num * num ⊣	shift *
⊢ T * num * num ⊣	shift num
⊢ T * num * num ⊣	reduce N → num
⊢ T * N * num ⊣	reduce T → T * N
⊢ T * num ⊣	shift *
⊢ T * num ⊣	shift num
⊢ T * num ⊣	reduce N → num
⊢ T * N ⊣	reduce T → T * N
⊢ T ⊣	shift ⊣
⊢ T ⊣	reduce S → ⊢ T ⊣
S	

Implementing Bottom-Up Parsing

- The idea is to shift symbols until the tail of our current sequence matches the right-hand side of a rule.
- Once we have the right-hand side of a rule, things get difficult.
  - Do we reduce right away, or do we keep shifting more symbols?
  - What if there are multiple rules with the same RHS to reduce by?
- With LL(1) top-down parsing, we dealt with the tough decisions by just saying "if we have to make decisions, it's not an LL(1) grammar".
- We'll start out by looking at **LR(0) parsing** which takes a similar approach: We only worry about how to handle grammars that *don't* require us to make decisions during parsing.

LR(0) Parsing

- **Left-to-right** scan of the input, **Rightmost** derivation produced… and **zero** symbols of lookahead!
- In LR(0) parsing, we don't use the next unread input symbol to make decisions, unlike in LL(1).
- The algorithm, at a high level:
  - Keep shifting until we see the right-hand side of a rule.
  - Keep reducing as long as the tail of our shifted sequence matches the right-hand side of a rule. Then go back to shifting.
- If this algorithm ever has to make decisions about which rule to reduce by, we give up and say "the grammar is not LR(0)".

Recognizing Right-Hand Sides

- Conceptually, the algorithm is simple, but how do we tell whether our sequence ends with the right-hand side of a rule?
- One approach is to **use an NFA!**
- It's easy to create NFAs for "strings that end with (something)".
- We could use ε-transitions to create a big NFA that tells us whether our NFA ends with the right-hand side of any rule in the grammar.

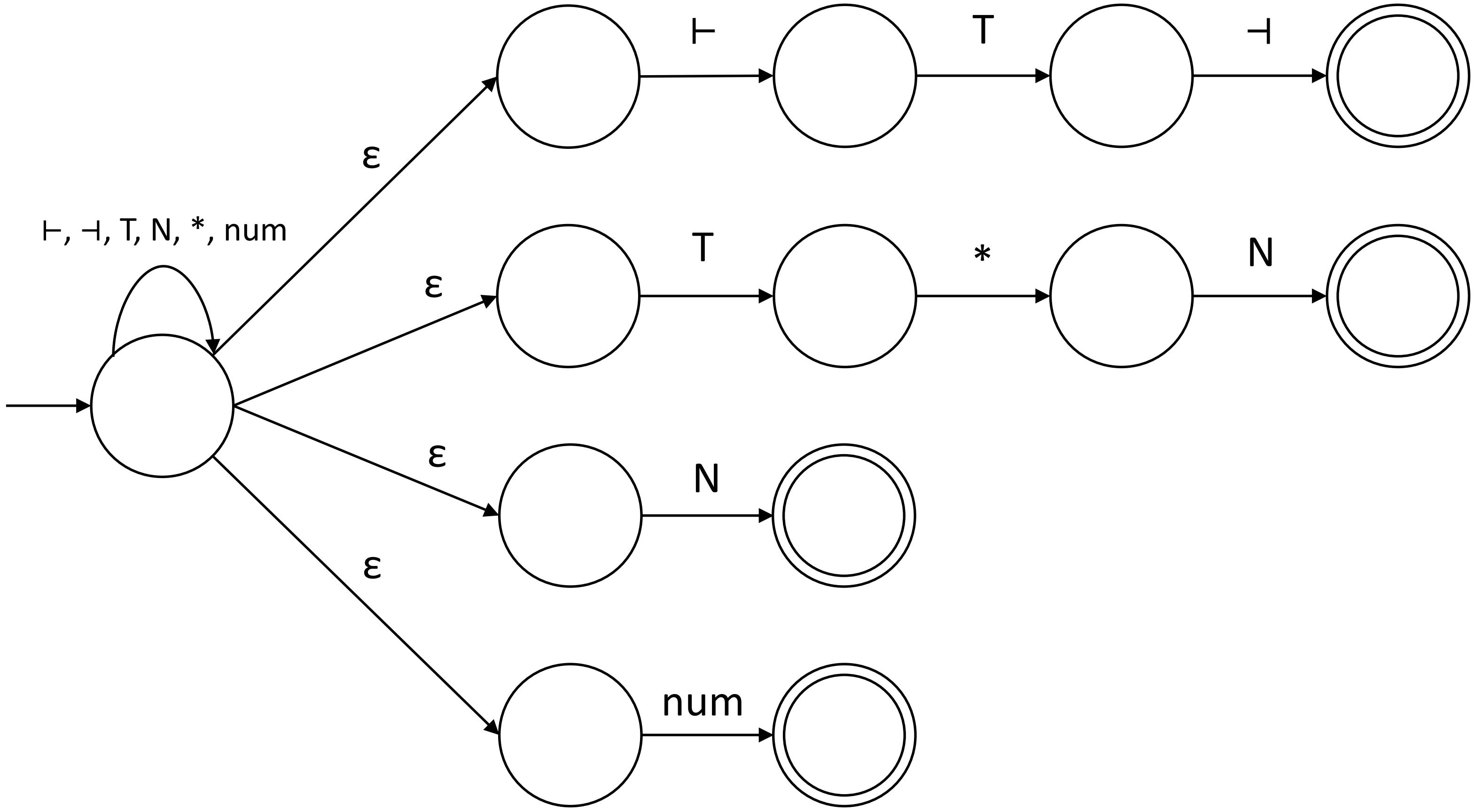
Simple NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num

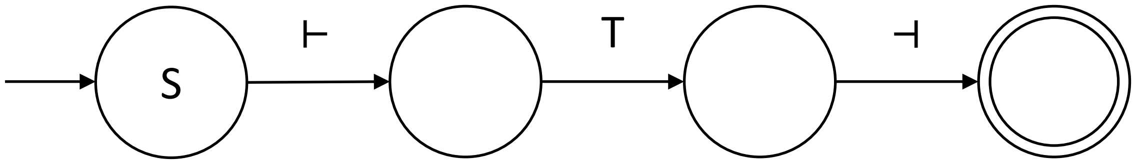


Problems With This NFA

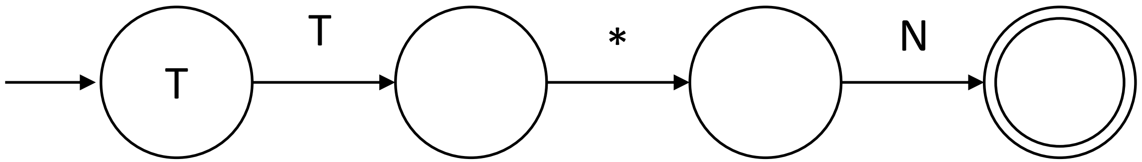
- The purpose of this NFA is just to tell us whether the current step of our "reverse derivation" ends with the right-hand side of a rule.
  - If it does, we reduce by that rule.
  - If not, we continue shifting.
- It accomplishes this goal, but it is a bit too lenient in what it accepts.
- For example, it will accept a nonsense string like ⊢⊣T⊢*N*num because it ends with "num" which is the RHS of N → num.
- This string will clearly not parse correctly, but it will take several reduce steps before we run into an issue.

Better NFA for Right-Hand Sides

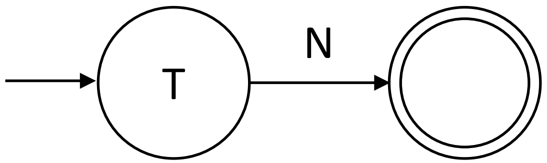
S → ⊢ T ⊣



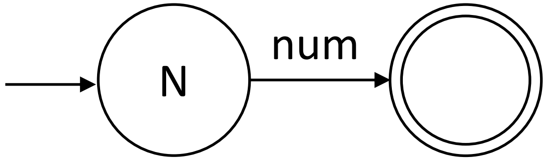
T → T * N



T → N



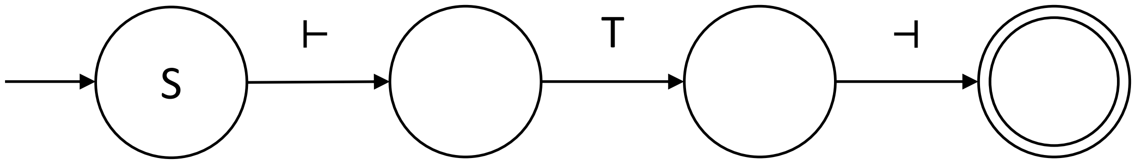
N → num



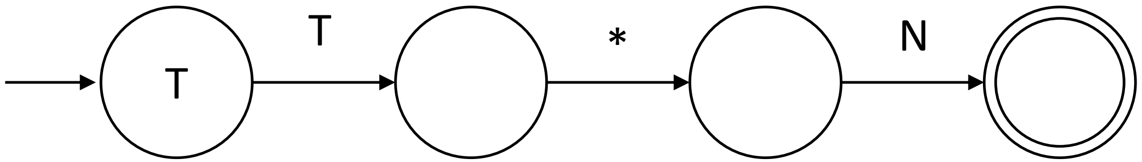
1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.

Better NFA for Right-Hand Sides

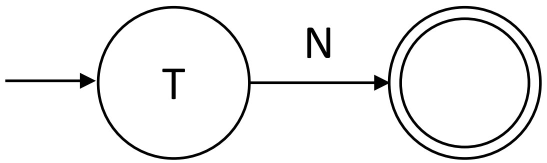
S → ⊢ T ⊣



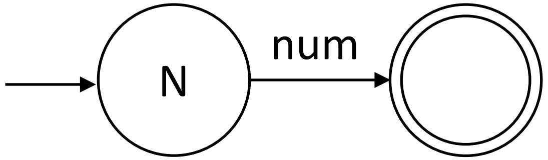
T → T * N



T → N



N → num



1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

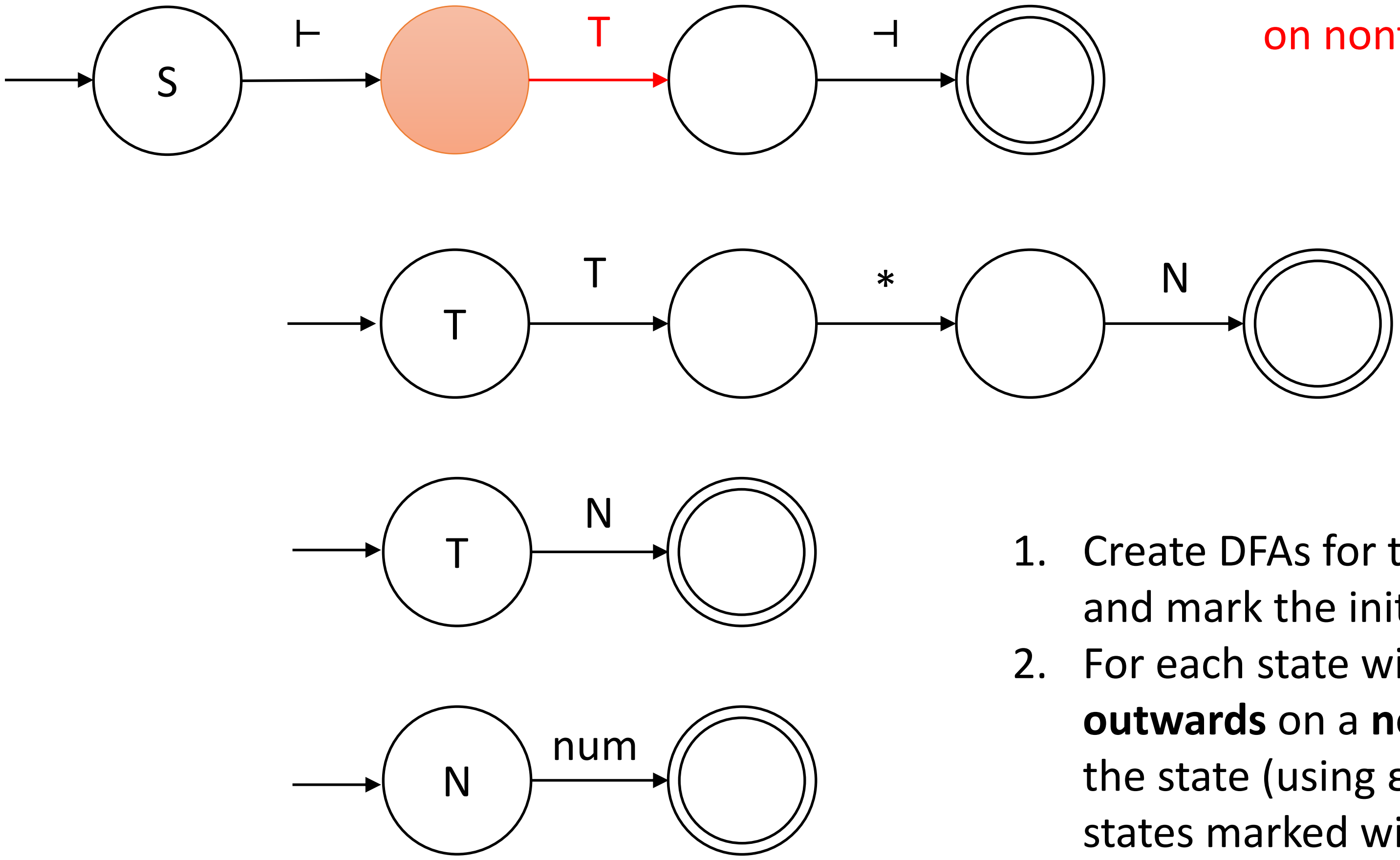
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



Outwards transition on nonterminal T

1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

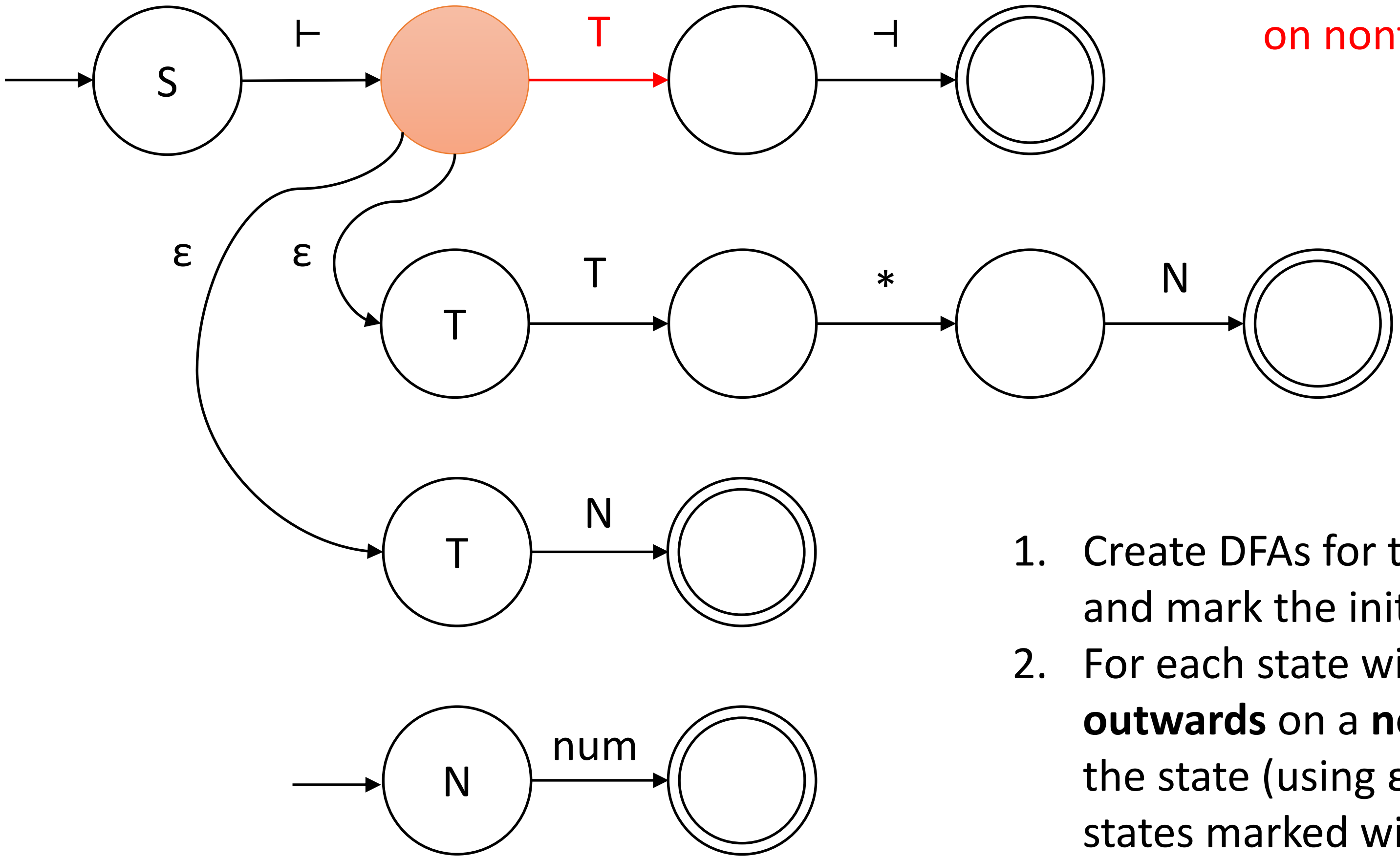
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



Outwards transition on nonterminal T

1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

Better NFA for Right-Hand Sides

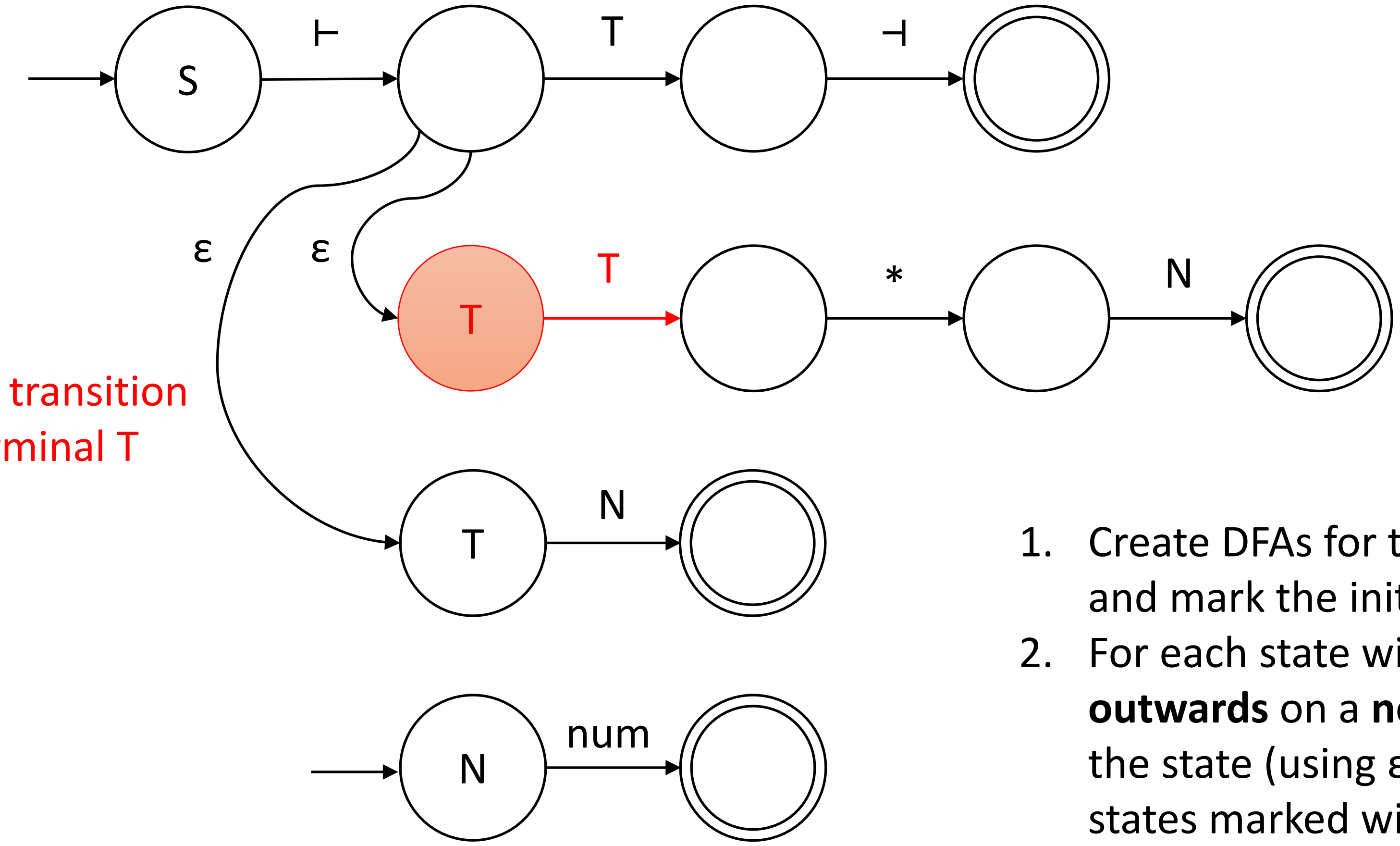
S → ⊢ T ⊣

T → T * N

Outwards transition on nonterminal T

T → N

N → num



1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

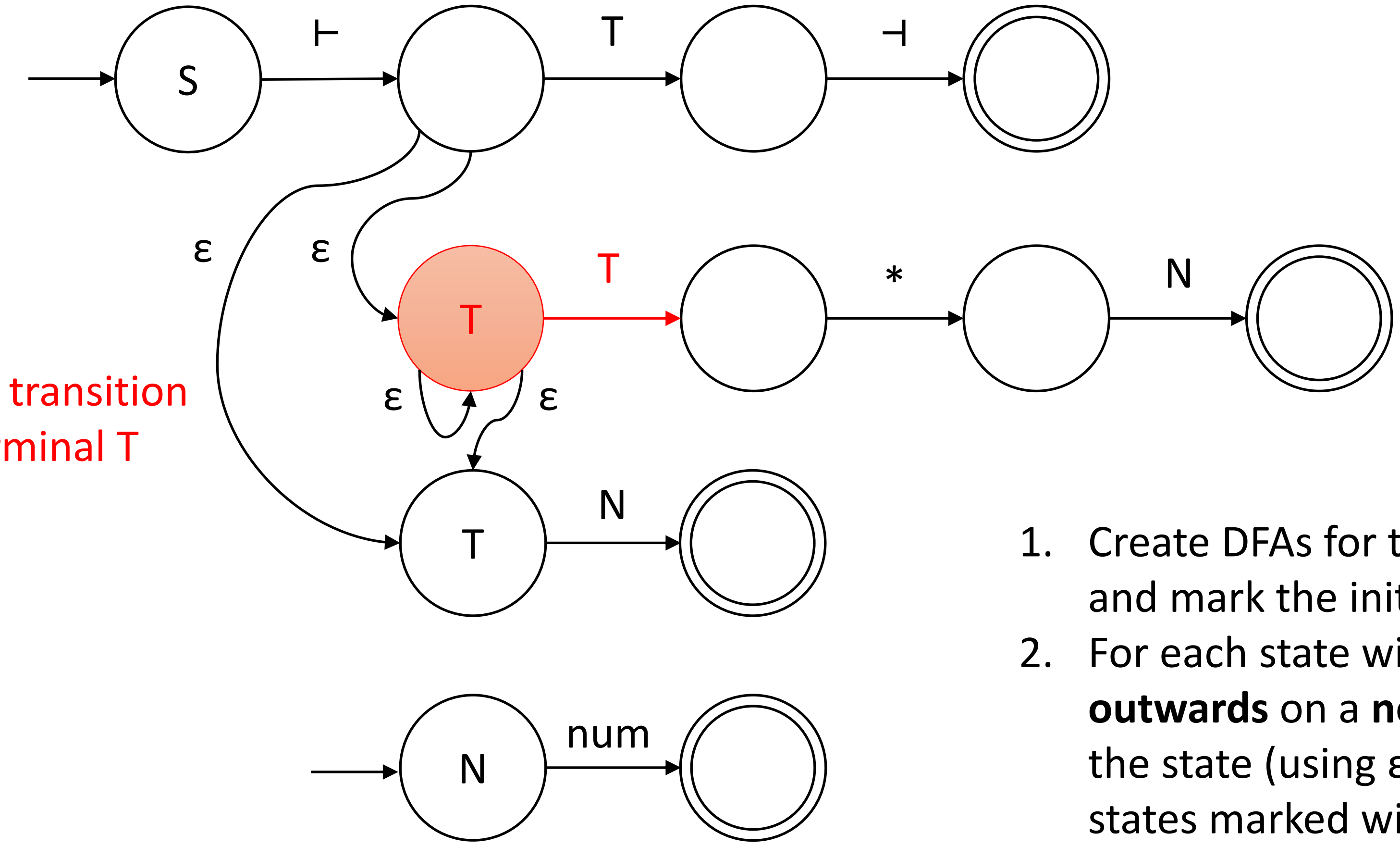
Better NFA for Right-Hand Sides

$S \rightarrow \vdash T \dashv$

$T \rightarrow T * N$

$T \rightarrow N$

$N \rightarrow num$



Outwards transition on nonterminal T

1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

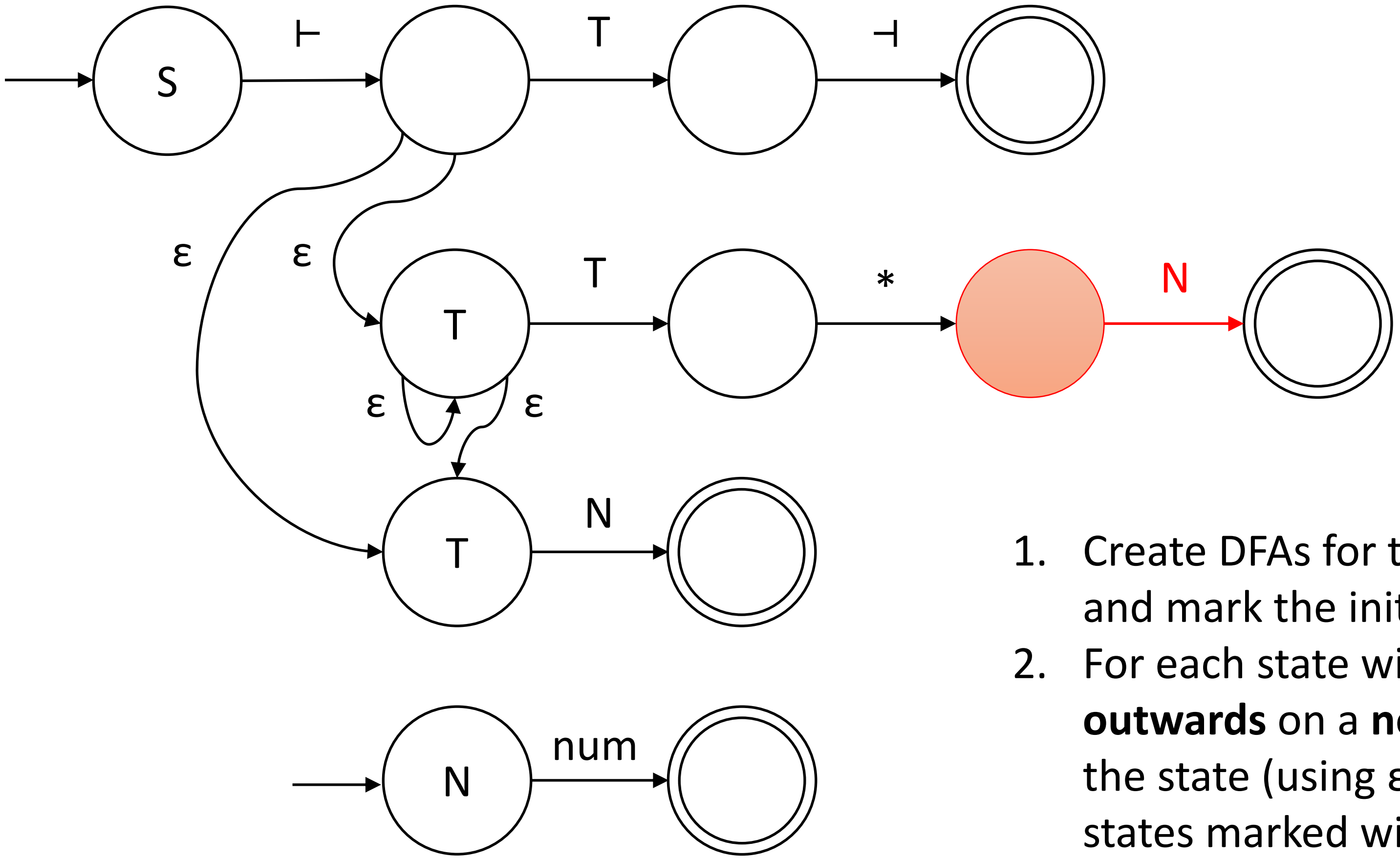
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



Outwards transition on nonterminal N

1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

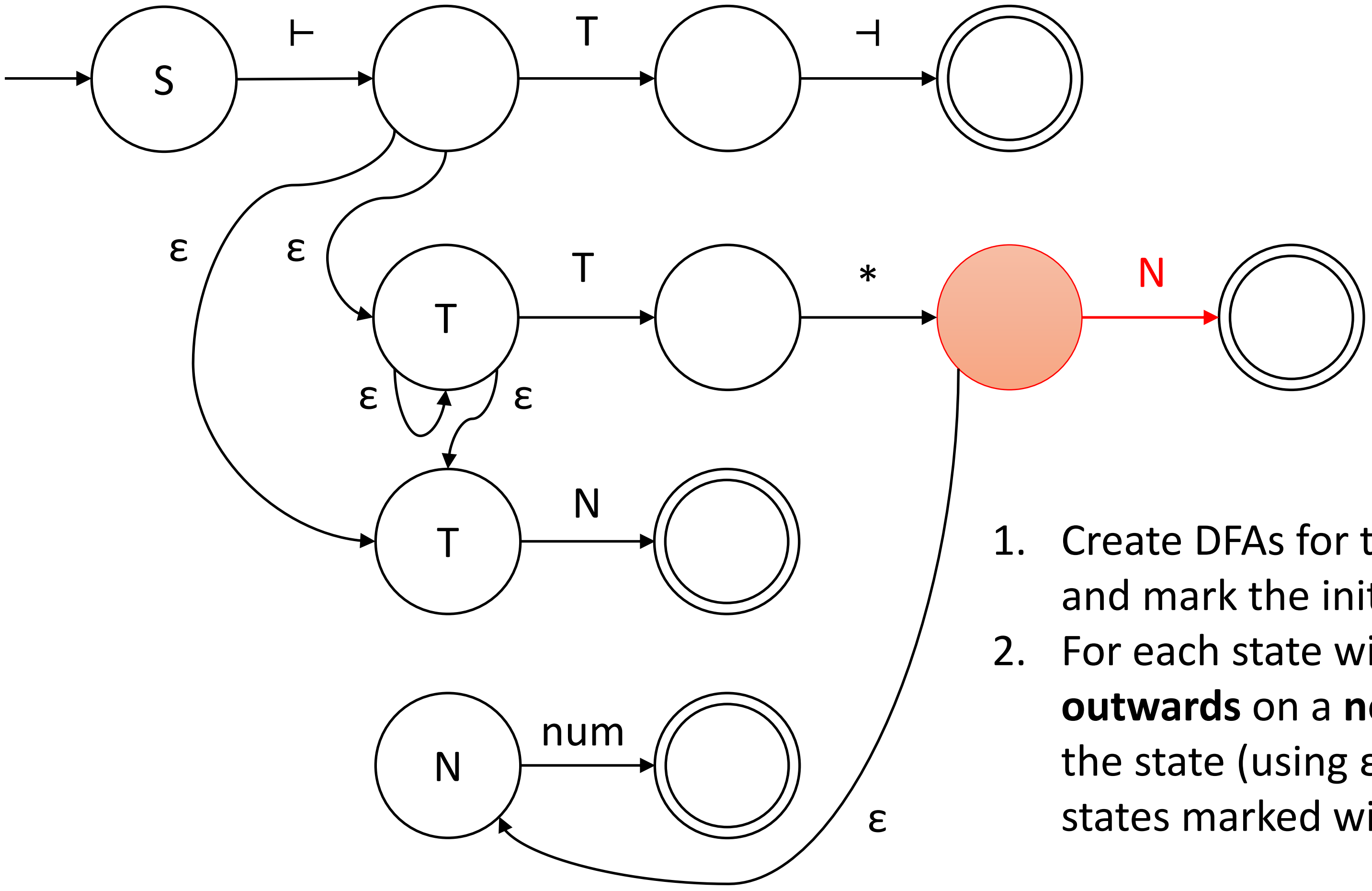
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



Outwards transition on nonterminal N

1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

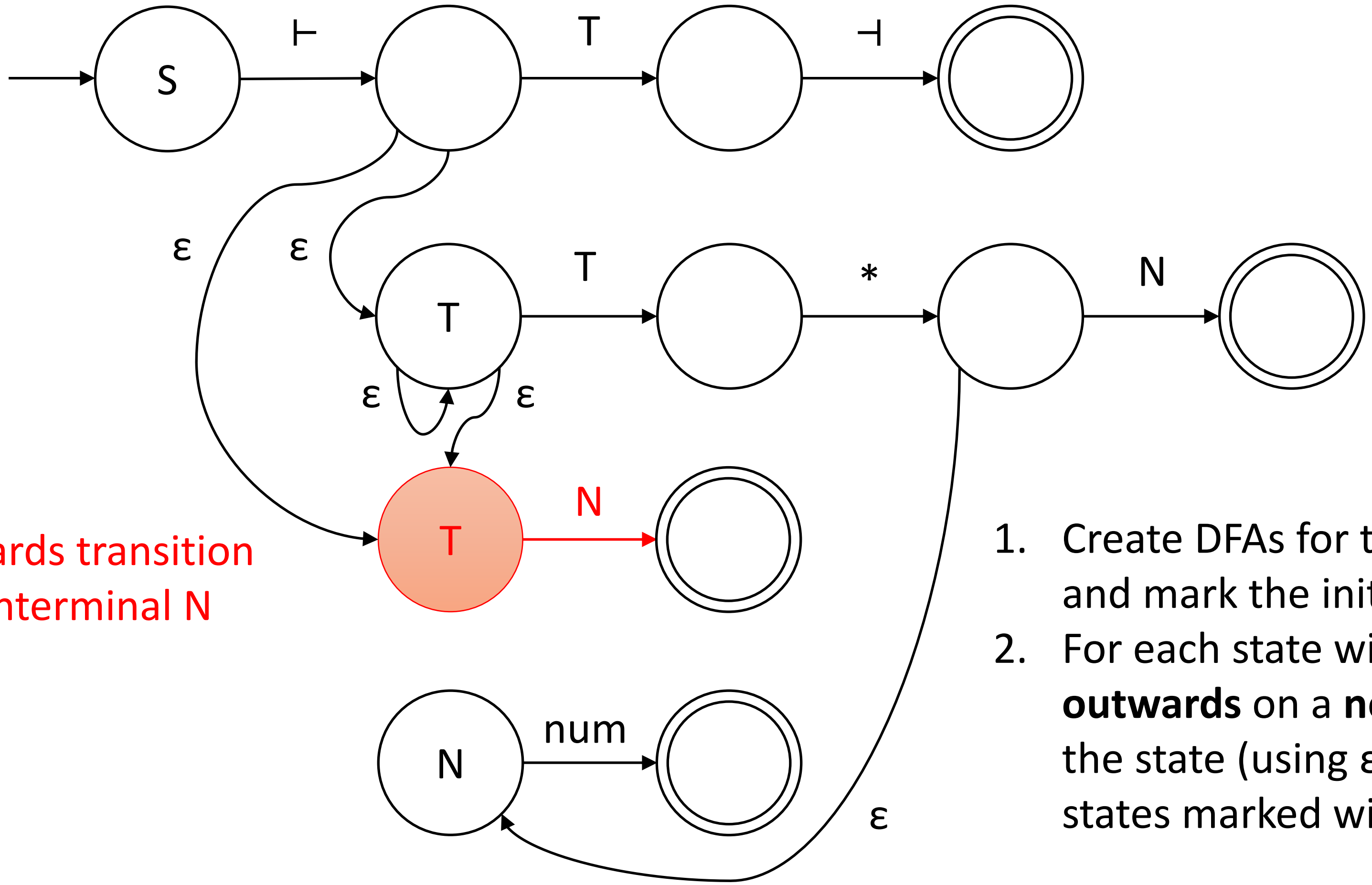
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

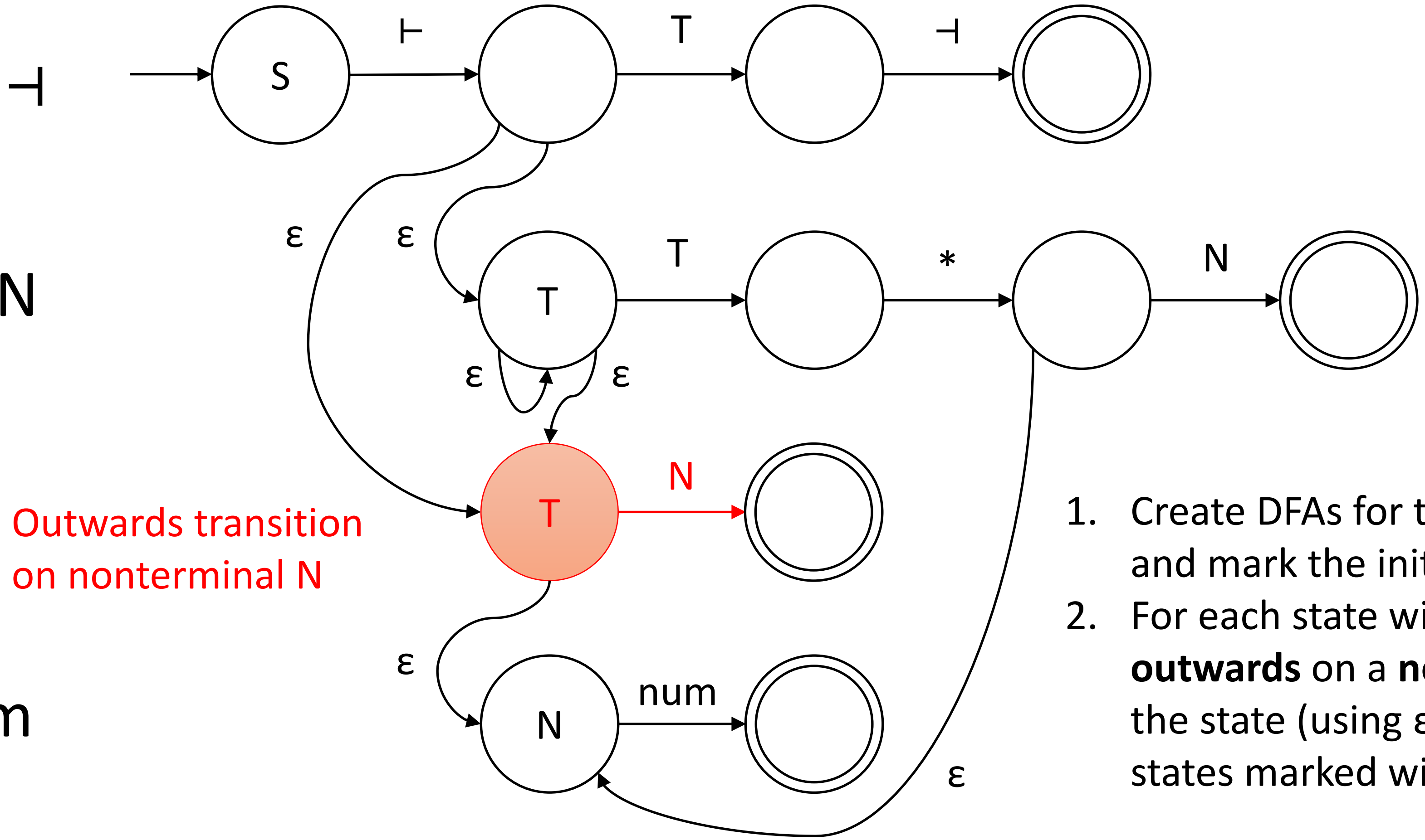
Better NFA for Right-Hand Sides

$S \rightarrow \vdash T \dashv$

$T \rightarrow T * N$

$T \rightarrow N$

$N \rightarrow num$



1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

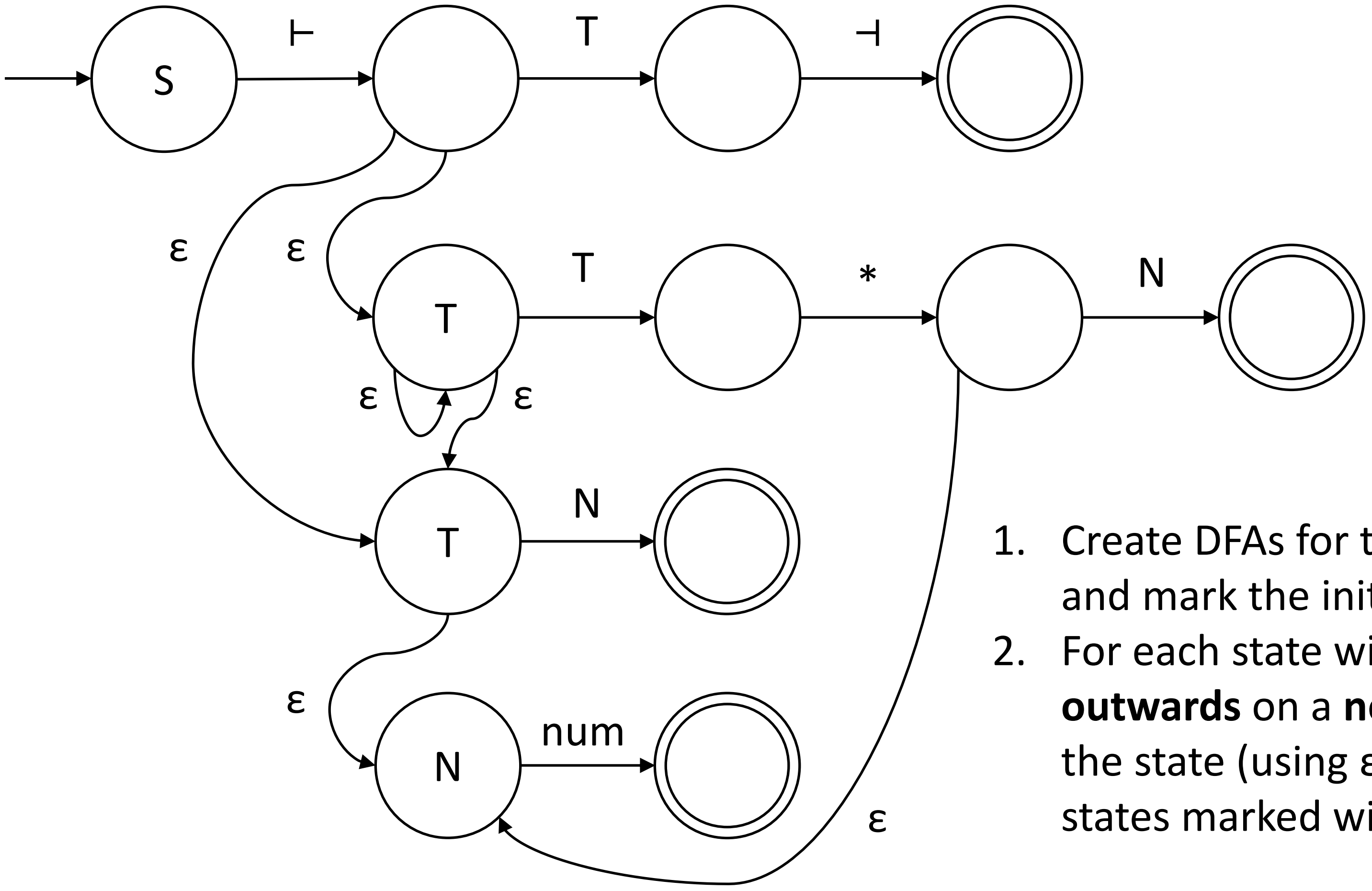
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



1. Create DFAs for the RHS of each rule and mark the initial states with the LHS.
2. For each state with a transition leading **outwards** on a **nonterminal**, connect the state (using ε-transitions) to all the states marked with that nonterminal.

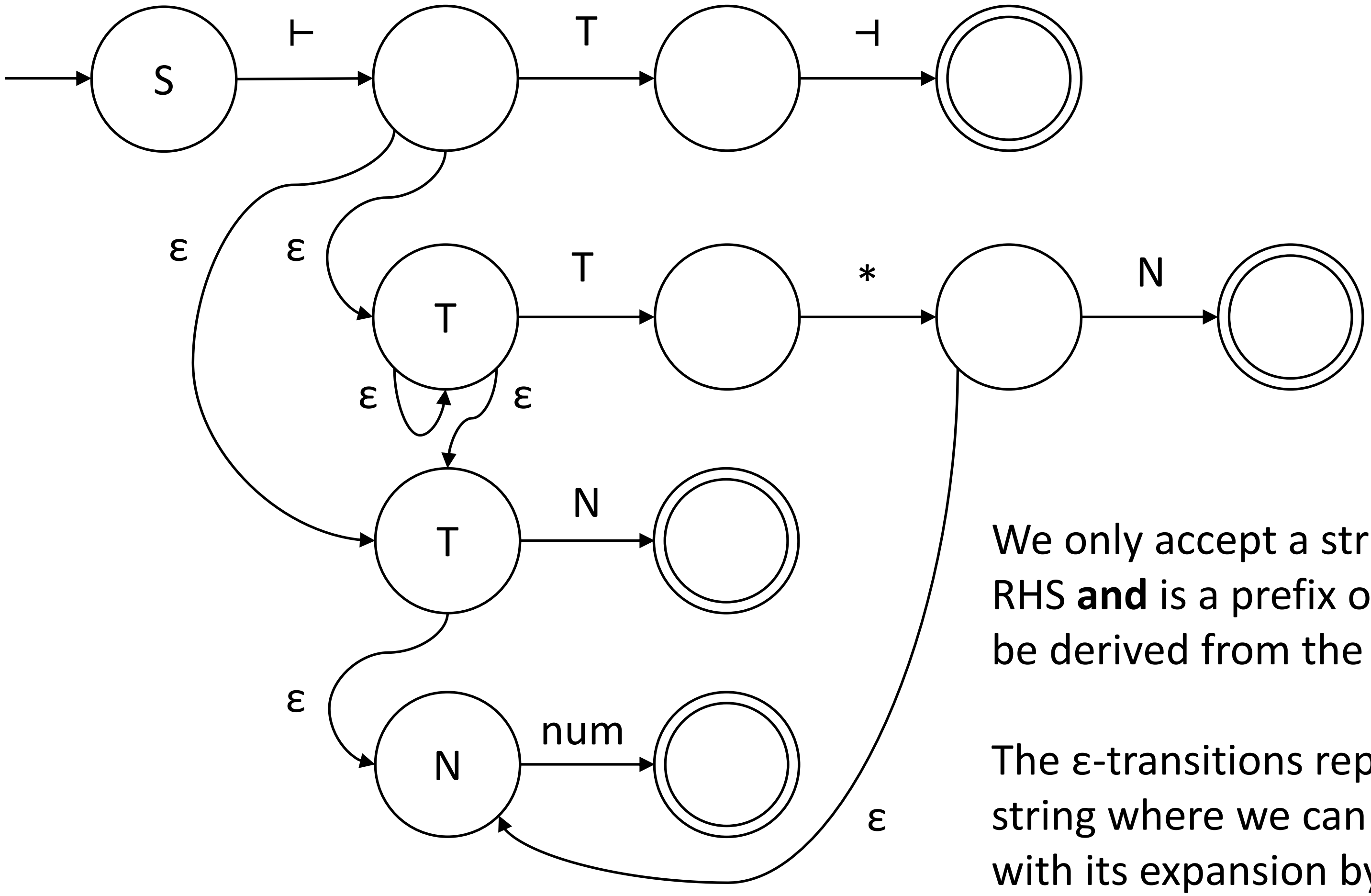
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



We only accept a string if it ends with an RHS **and** is a prefix of something that can be derived from the start symbol.

The ε-transitions represent places in the string where we can replace a nonterminal with its expansion by a rule.

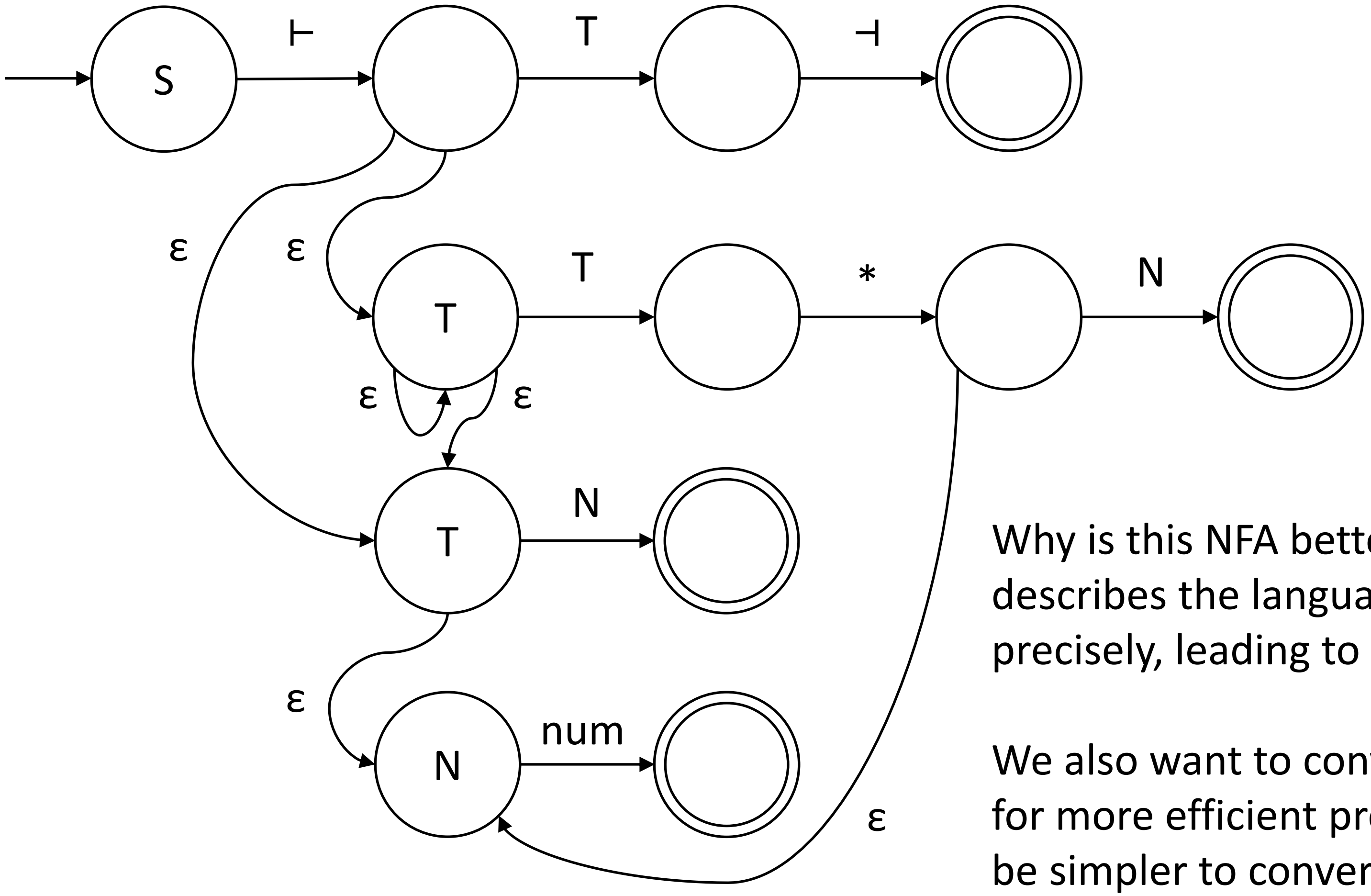
Better NFA for Right-Hand Sides

S → ⊢ T ⊣

T → T * N

T → N

N → num



Why is this NFA better? For one, it describes the language we want more precisely, leading to better error reporting.

We also want to convert the NFA into a DFA for more efficient processing. This NFA will be simpler to convert.

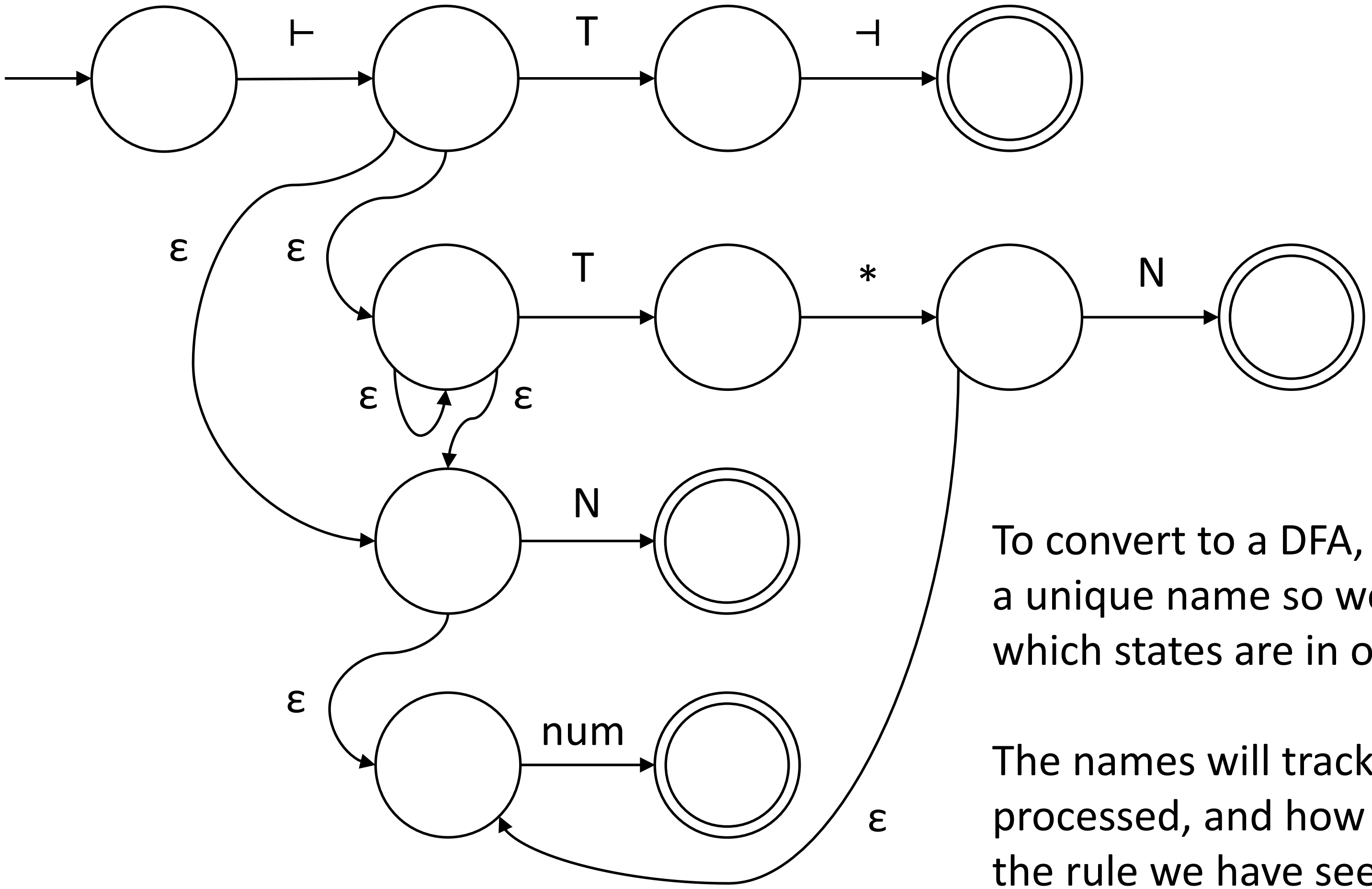
Converting to a DFA

S → ⊢ T ⊣

T → T * N

T → N

N → num



To convert to a DFA, we will give each state a unique name so we can keep track of which states are in our subsets.

The names will track which rule is being processed, and how much of the RHS of the rule we have seen.

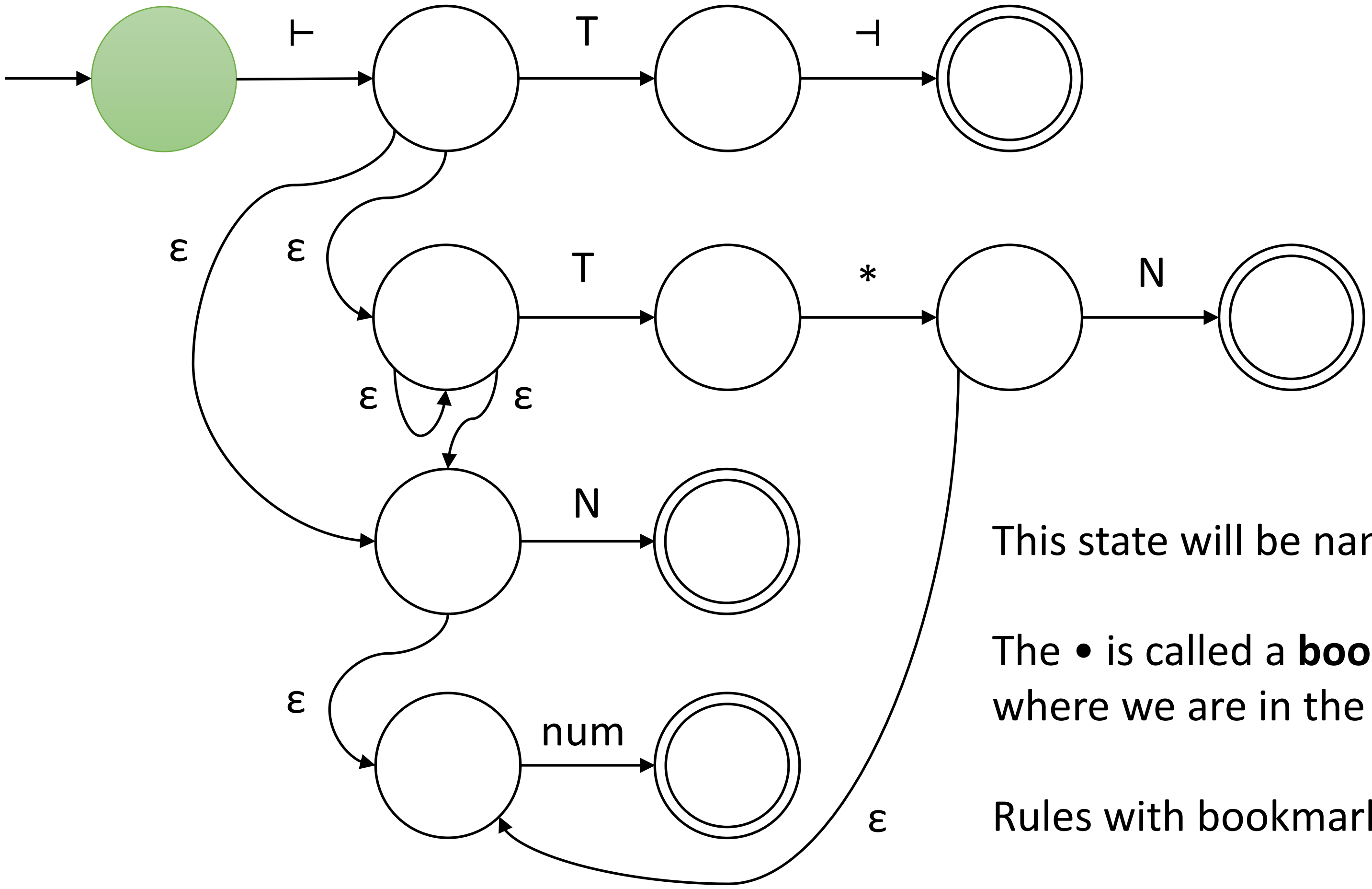
Converting to a DFA

$S \to \vdash T \dashv$

$T \to T * N$

$T \to N$

$N \to num$



This state will be named: $S \to \bullet \vdash T \dashv$

The • is called a **bookmark** and indicates where we are in the right hand side.

Rules with bookmarks are called **items**.

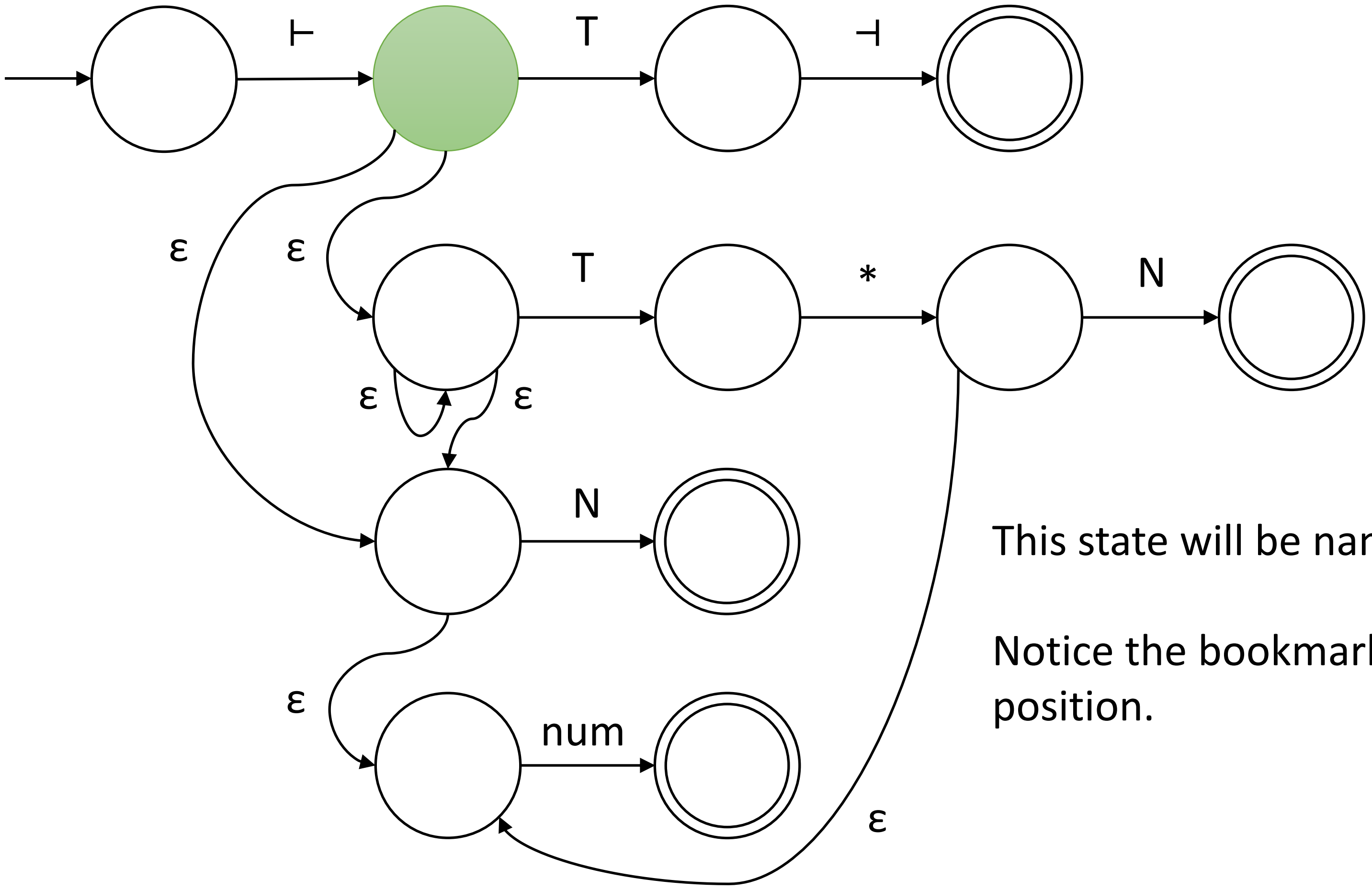
Converting to a DFA

S → ⊢ T ⊣

T → T * N

T → N

N → num



This state will be named: S → ⊢ • T ⊣

Notice the bookmark advanced one position.

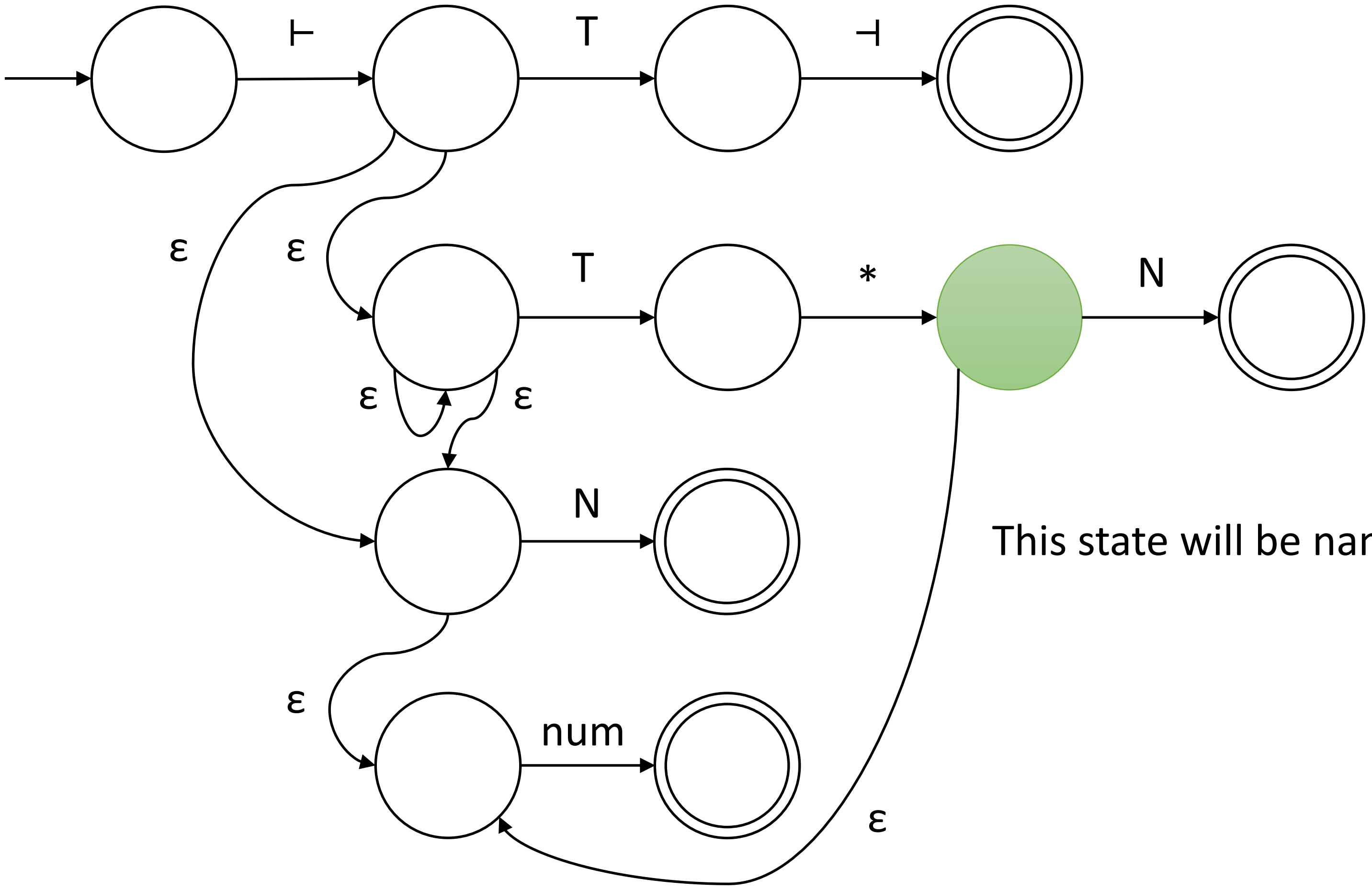
Converting to a DFA

S → ⊢ T ⊣

T → T * N

T → N

N → num



This state will be named: T → T * • N

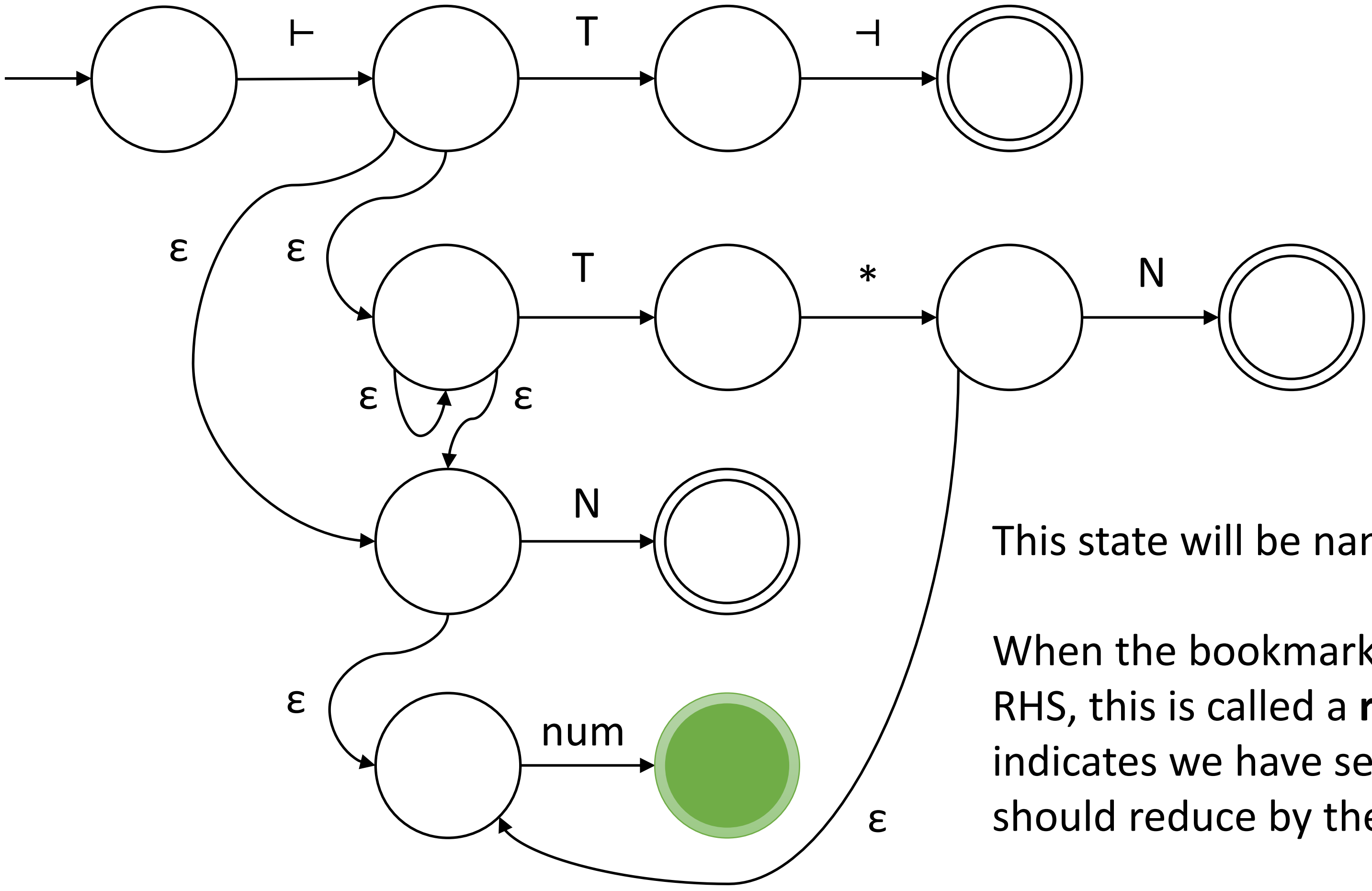
Converting to a DFA

S → ⊢ T ⊣

T → T * N

T → N

N → num



This state will be named: N → num •

When the bookmark is at the end of the RHS, this is called a **reducible item** since it indicates we have seen an entire RHS and should reduce by the associated rule.

Converting to a DFA

S → ⊢ T ⊣

T → T * N

T → N

N → num

Use the subset construction with this table.

State	⊢	⊣	T	N	*	num	ε
S→•⊢T⊣	S→⊢•T⊣						
S→⊢•T⊣			S→⊢T•⊣				T→•T*N T→•N
S→⊢T•⊣		S→⊢T⊣•					
S→⊢T⊣•							
T→•T*N			T→T•*N				T→•T*N T→•N
T→T•*N					T→T*•N		
T→T*•N				T→T*N•			N→•num
T→T*N•							
T→•N				T→N•			N→•num
T→N•							
N→•num						N→num•	
N→num•							

Aside: Augmented Grammars

- Our grammar has an unusual rule S → ⊢ T ⊣ which forces every string in the language to be surrounded with ⊢ and ⊣ symbols.
- We can add these symbols to any CFG:
  - If S is the old start symbol, add a new start symbol S' and rule: S' → ⊢ S ⊣
- This is called **augmenting** the grammar. The augmented CFG is essentially the same but all strings are "wrapped" with ⊢ and ⊣.
  - ⊢ is often called "beginning of file" and ⊣ is often called "end of file".
- This can make algorithms easier to describe. For example, in the LR(0) DFA construction, it forces the DFA to have a simple starting state that only contains one item.

Constructing LR(0) DFAs

- LR(0) DFAs can also be constructed directly without creating the NFA and using the subset construction. We describe an algorithm for this (intended for augmented grammars) on the next slide.
- Some terminology and notation:
  - A **fresh item** is an item with the bookmark on the very left of the RHS.
  - All states are initially **incomplete** and will be marked as **complete** over the course of the algorithm. The algorithm is done when every state is complete.
  - We will write a generic non-reducible item as $A \rightarrow \alpha \bullet \sigma \beta$. Here α and β are strings (possibly empty) of terminals and nonterminals, and σ is a single symbol, which can be a terminal or a nonterminal.

Constructing LR(0) DFAs

1. Create an initial state which contains the fresh item $S' \rightarrow \bullet \vdash S \dashv$, corresponding to the starting rule of the augmented grammar.
2. For each incomplete state X, fill out the transitions as follows.
   - For each non-reducible item $A \rightarrow \alpha \bullet \sigma \beta$ in X, create a set of items Y that initially contains the item $A \rightarrow \alpha\sigma \bullet \beta$.
   - Expand the set of items Y as follows:
     - For each non-reducible item $A \rightarrow \alpha \bullet \sigma \beta$ in Y such that σ is a **nonterminal**, add fresh items to Y for every rule with σ on the left-hand side.
     - Repeat the process in the above bullet point with the newly added fresh items. Keep repeating until there are no more new items to add.
   - If Y already matches an existing state, add a transition from X to the existing state on σ.
   - Otherwise, create a new state for Y, and add a transition from X to the new state on σ.
   - Mark the state X as complete.

Repeat Step 2 until all states are complete.

Coming Up Next

- We'll learn how to use an LR(0) DFA to parse strings efficiently.
- We'll learn about **parsing conflicts** and get a sense of the limitations of the LR(0) technique.
- We'll learn about **SLR(1)** (short for Simple LR(1)), a way of resolving some (but not all) conflicts using lookahead and Follow sets.
- We'll learn how to build a **parse tree** as we parse (as opposed to just finding a "reverse derivation").