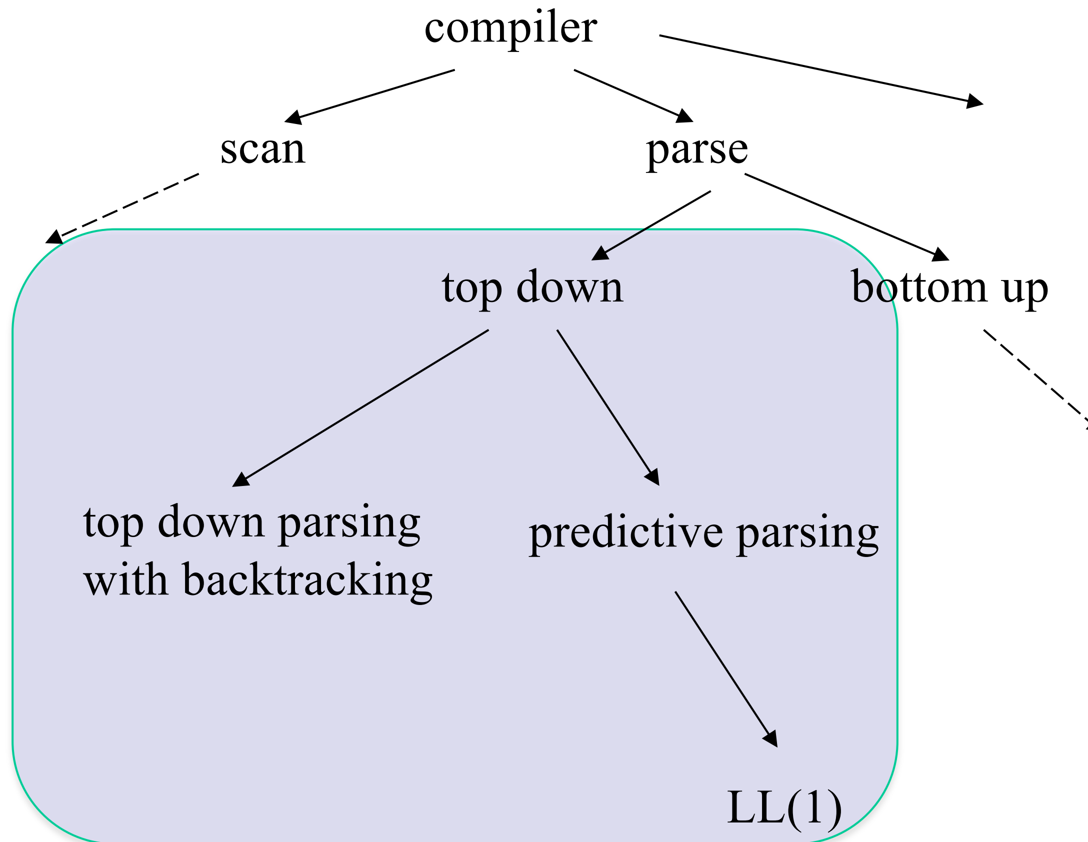
CMPSC 160
Translation of Programming Languages

Lecture 5: Top-down parsing + Introduction to Bottom-up Parsing

Where are we in the process?



Predictive Parsing with LL(1) Grammars

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α and β based on peeking at the next token in the stream

LL(1): **L**eft-to-right scan of the input, **L**eftmost derivation, **1**-token look-ahead

A grammar *G* is LL(1) if for each set of its productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

FIRST(α_1), FIRST(α_2), ..., FIRST(α_n), are all pair-wise disjoint

If $\alpha_i \Rightarrow^* \varepsilon$, then FIRST (α_j) $\cap$ FOLLOW (A) = $\varnothing$ for all $1 \leq i \leq n$, $i \neq j$

In other words, LL(1) grammars

- during leftmost derivation, productions are uniquely predictable with a one token lookahead

General Predictive Parsing

How much look-ahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami, or Earley's algorithm
  - Complexity is $O(|x|^3)$ where x is the input string

Fortunately,

- Large subclasses of context free grammars can be parsed efficiently with limited look-ahead
  - Linear complexity, $O(|x|)$ where x is the input string
- Most programming language constructs fall in those subclasses

Among the interesting subclasses is the *LL(1)* grammars.

Recursive Descent Parsing

1	S	$\rightarrow$	if E then S else S
2		|	begin S L
3		|	print E
4	L	$\rightarrow$	end
5		|	; S L
6	E	$\rightarrow$	num = num

```
void match(int token) {
        if (lookahead==token)
                lookahead=getNextToken();
        else
                error();
}
```

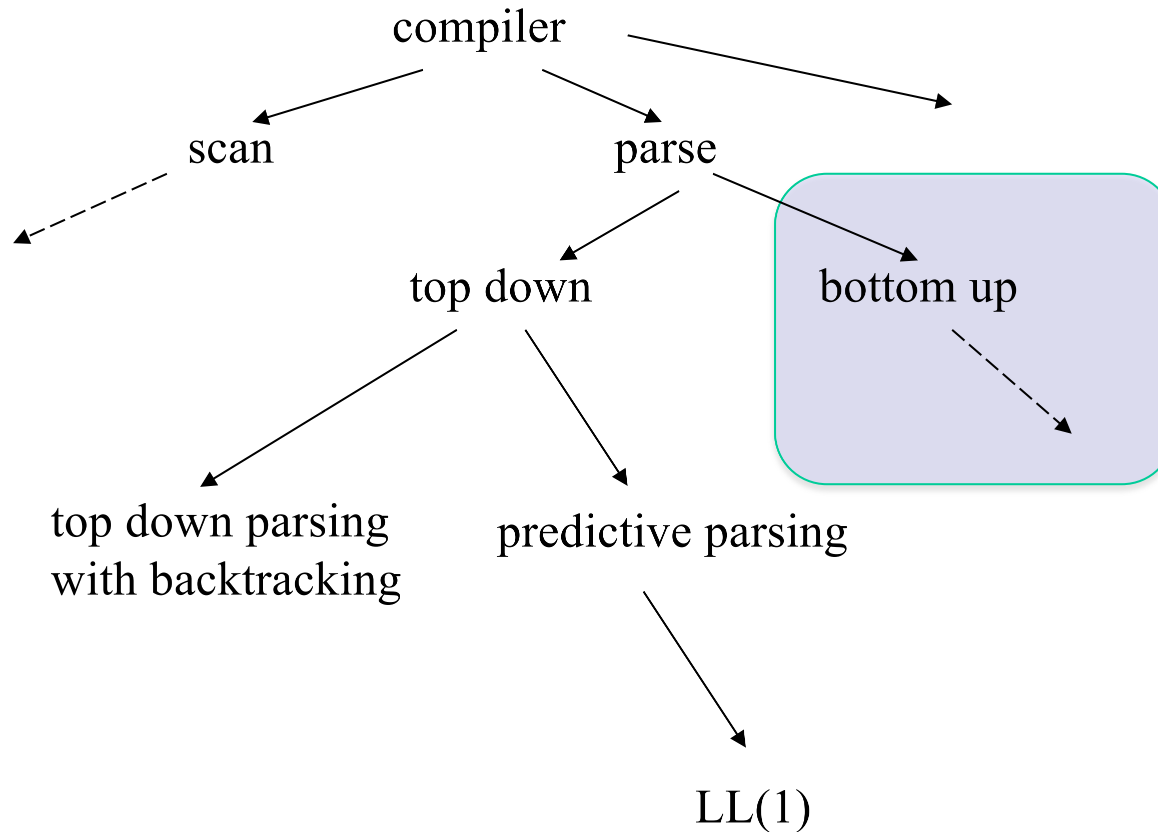
```
void S() {
  switch(lookahead) {
    case IF: match(IF); E(); match(THEN); S();
             match(ELSE); S(); break;
    case BEGIN: match(BEGIN); S(); L(); break;
    case PRINT: match(PRINT); E(); break;
    default: error();
  }
}

void E() { match(NUM); match(EQ); match(NUM); }
```

```
void L() {
  switch(lookahead) {
    case END: match(END); break;
    case SEMI: match(SEMI); S();
               L(); break;
    default: error();
  }
}

void main() {
  lookahead=getNextToken();
  S();
  match(EOF);
}
```

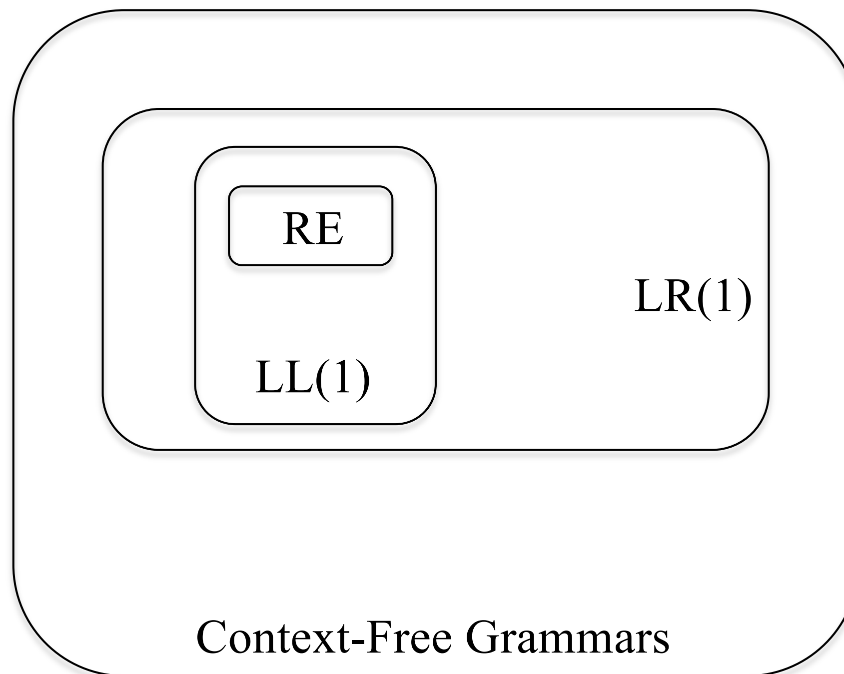
Where are we in the process?



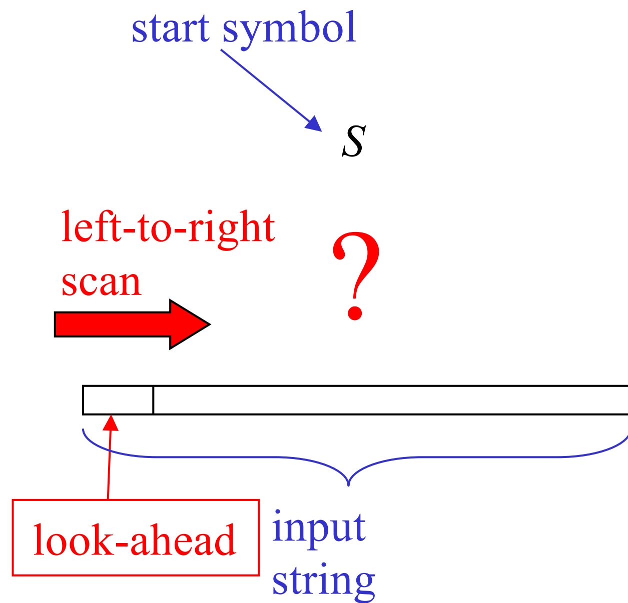
Parsing Techniques

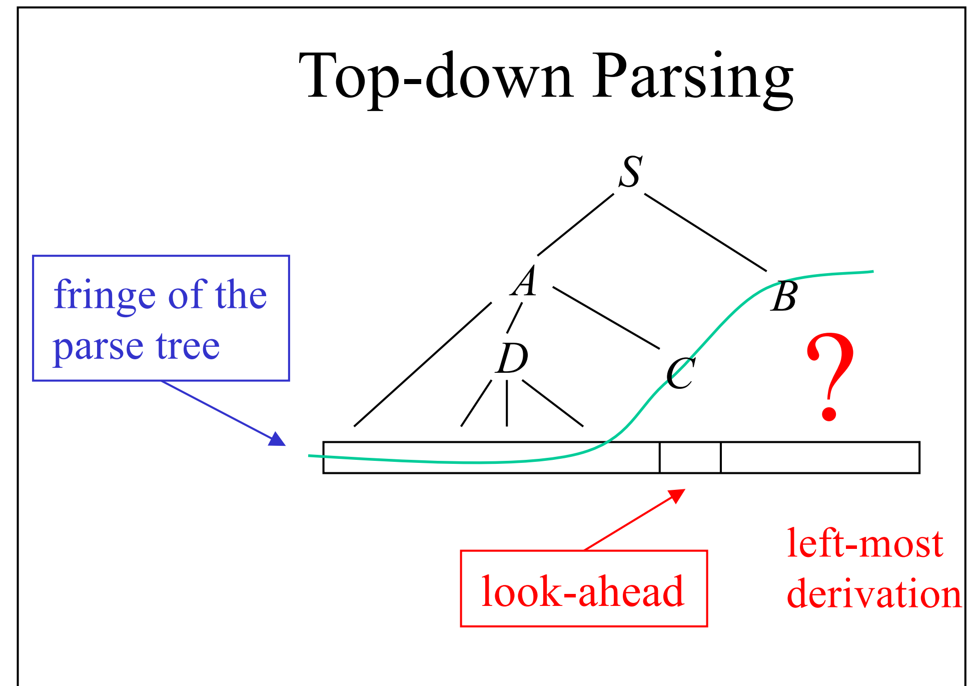
Top-down parsers *(LL(1), recursive descent parsers)*

Bottom-up parsers *(LR(1), shift-reduce parsers)*



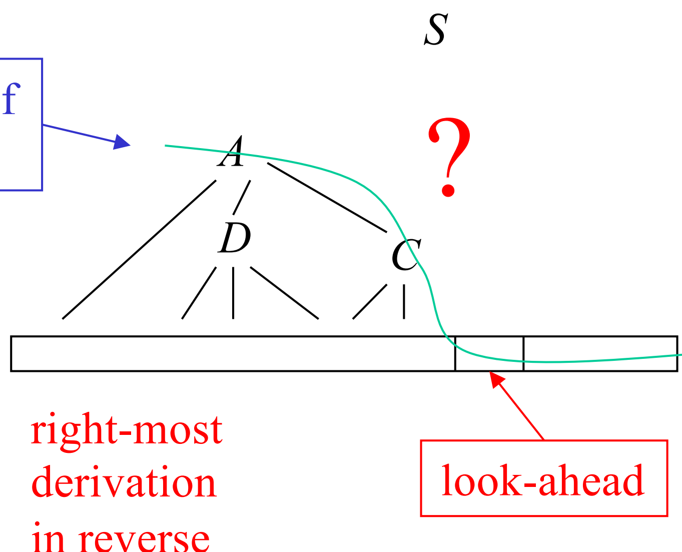
Top-Down Parsing v.s. Bottom-Up Parsing





When we start with a predictive top-down parser, the look-ahead symbol we read from our input string MUST fully specify the parse tree from S to the input symbol. In the example, **we have to know that S→AB before we even see any of B**

Top-Down Parsing v.s. Bottom-Up Parsing



In a bottom-up parser, we can **delay this decision** because we only need to build the tree up above the part of the input string we have examined so far.

In the graphical example on the left, you can see that even though we are at the same point in the input string, the production S→AB has not been specified yet. **This delayed decision allows us to parse more grammars than predictive top-down parsing (LL).**

As a nice side effect, bottom-up parsing allows us to handle **left-recursive** grammars without modification

Bottom-Up Parsers

- Given a stream of tokens w, reduce it to the start symbol.

$$
\begin{array}{lcl}
E & \longrightarrow & E+T \\
E & \longrightarrow & T \\
T & \longrightarrow & \texttt{id}
\end{array}
$$

Parsing for **id + id**:

$$
\begin{array}{c}
\texttt{id} + \texttt{id} \\
T + \texttt{id} \\
E + \texttt{id} \\
E + T \\
E
\end{array}
$$

Reduction ≡ Derivation in Reverse (Rightmost Derivation)

LR: Left-to-right scanning, Rightmost Derivation

Shift-Reduce Parsing: An Example

E	$\longrightarrow$	$E+T$
E	$\longrightarrow$	T
T	$\longrightarrow$	$\texttt{id}$

STACK	INPUT STREAM	ACTION
$	id + id $	**shift**
$ id	+ id $	**reduce by $T \longrightarrow \texttt{id}$**
$ T	+ id $	reduce by $E \longrightarrow T$
$ E	+ id $	shift
$ E +	id $	shift
$ E + id	$	reduce by $T \longrightarrow \texttt{id}$
$ E + T	$	reduce by $E \longrightarrow E+T$
$ E	$	**ACCEPT**

Stack: sentential form + rightmost derivation in reverse
Input stream: from left to right
Two key actions: shift or reduce

Key question: when to reduce? which production rule to choose?

Handles

- **Handle:** A structure that furnishes a means to perform reductions

E	$\longrightarrow$	$E+T$
E	$\longrightarrow$	T
T	$\longrightarrow$	`id`

- Handles are substrings of sentential forms:
  - A substring that matches the right-hand side of a production
  - Reduction using that rule can lead to the start symbol

- **Handle Pruning**: The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning.

Shift-Reduce Parsing

Key operations (based on the stack and input stream):

- **Shift:** Construct leftmost handle on top of stack
- **Reduce:** Identify handle and replace by corresponding RHS
- **Accept:** Continue until string is reduced to start symbol and input token stream is empty
- **Error:** Signal parse error if no handle is found.

Implementations:

- Stack to hold grammar symbols (corresponding to tokens seen thus far).
- Stack is initially empty (denoted by $).
- Input stream of yet-to-be-seen tokens.
- Start with the key operations above.
- **Handles appear on top of stack.**
- Parse is successful if stack contains only the start symbol when the input stream ends.

Handle-pruning, Bottom-up Parsers

One implementation of a simple *shift-reduce parser*

```
push $
lookahead  = get_ next_token( )
repeat until (top of stack == start symbol and lookahead == $)
    if the top of the stack is a handle α→β
        then  /* reduce β to α */
            pop |β| symbols off the stack
            push α onto the stack
        else if (token ≠ $)
            then /* shift */
                push lookahead
                lookahead = get_next_token( )
```

How do errors show up?

- **failure to find a handle**
- **hitting $ and needing to shift (final else clause)**

Either generates an error

"pop |β| symbols off the stack" means that if we have some production A→BCx then we need to pop B, C, and x off the stack (|b| = 3 in this case) before we push A

Keys for Shift-Reduce Parsing

- Identify a handle in string.
  - Top of stack is the rightmost end of the handle.

- What is the leftmost end?
  - If there are **multiple productions** with the handle on the RHS, which one to choose?

Existence of a Unique Handler for Rightmost Derivation

Insight

> *If G is unambiguous, then every right-sentential form has a unique handle.*

If we can find those handles, we can build a derivation!

Sketch of Proof:

1. *G* is unambiguous $\Rightarrow$ rightmost derivation is unique (through a list of sentential forms $\gamma_{1, \ldots} \gamma_n$)
2. $\Rightarrow$ a unique production $\alpha \rightarrow \beta$ applied to take sentential form γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position ***k*** at which $\alpha \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $< \alpha \rightarrow \beta , \boldsymbol{k} >$: replacing β at k with α.

This all follows from the definitions

LR Parsers

- LR(k) parsers could parse a set of CFGs, for which the **handles can be deterministically indented with k lookahead of the input stream**.
  - The key part is recognizing the handles

- **Wait a minute… recognizing… we studied something that recognizes strings: State machines (aka automata)**!

- The big picture is that LR parsers use a state machine (to recognize handles) in coordination with a stack (to handle the recursive nature of grammars) to parse an input.

A Simple Example of LR Parsing

- For input stream: aa

$$S \longrightarrow BC$$
$$B \longrightarrow \texttt{a}$$
$$C \longrightarrow \texttt{a}$$

Stack	Input Stream	Action
$	a a $	shift
$ a	a $	reduce by $B \longrightarrow$ a
$ B	a $	shift
$ B a	$	reduce by $C \longrightarrow$ a
$ B C	$	reduce by $S \longrightarrow BC$
$ S	$	**ACCEPT**

Any insights about why we should select these handles respectively???
Can you turn these insights into general principles???

A Simple Example of LR(0) Parsing

1	S'	$\longrightarrow$	S
2	S	$\longrightarrow$	$B\ C$
3	B	$\longrightarrow$	a
4	C	$\longrightarrow$	a

STACK	INPUT	STATE	ACTION
\$	a a \$	$S' \longrightarrow \bullet\ S$ $S \longrightarrow \bullet\ BC$ $B \longrightarrow \bullet$ a	shift
\$ a	a \$	$B \longrightarrow$ a$\bullet$	reduce by 3
\$ B	a \$	$S \longrightarrow B\bullet\ C$ $C \longrightarrow \bullet$ a	shift
\$ B a	\$	$C \longrightarrow$ a$\bullet$	reduce by 4
\$ $B\ C$	\$	$S \longrightarrow BC\bullet$	reduce by 2
\$ S	\$	$S' \longrightarrow S\bullet$	**ACCEPT**

- **States**: Productions with "• " somewhere on the RHS.
  - Grammar symbols before the "• " are on stack
  - Grammar symbols after the "• " represent symbols in the input stream.
- **Closure:** What other items are "equivalent" to the given item?

Given an item $A \rightarrow \alpha\bullet B\beta$, closure($A \rightarrow \alpha\bullet B\beta$) is the smallest set that contains the item **$A \rightarrow \alpha\bullet B\beta$**, and every item in closure(**$B \rightarrow \bullet\ \gamma$**) for every production $B \rightarrow \gamma \in$ CFG

Example: States of an LR(0) parser

E'	$\longrightarrow$	E	E	$\longrightarrow$	T
E	$\longrightarrow$	$E+T$	T	$\longrightarrow$	$\texttt{id}$

I_0	$= closure(\{E' \longrightarrow \bullet\ E\})$	$E' \longrightarrow \bullet\ E$ $E \longrightarrow \bullet\ E+T$ $E \longrightarrow \bullet\ T$ $T \longrightarrow \bullet\ \texttt{id}$
I_1	$= goto(I_0, E)$	$E' \longrightarrow E\bullet$ $E \longrightarrow E\bullet\ +T$
I_2	$= goto(I_0, T)$	$E \longrightarrow T\bullet$
I_3	$= goto(I_0, \texttt{id})$	$T \longrightarrow \texttt{id}\bullet$
I_4	$= goto(I_1, +)$	$E \longrightarrow E+\bullet\ T$ $T \longrightarrow \bullet\ \texttt{id}$
I_5	$= goto(I_4, T)$	$E \longrightarrow E+T\bullet$

- Instead of generating the states along with the inputs to parse, we could do a preprocessing and store all possible states (along with their transition) in a table before the parsing process.
- The parsing process becomes largely a table lookup process (just like the scanner part).

States to Table

I_0	$= closure(\{E' \longrightarrow \bullet\ E\})$	$E' \longrightarrow \bullet\ E$ $E \longrightarrow \bullet\ E+T$ $E \longrightarrow \bullet\ T$ $T \longrightarrow \bullet\ \mathtt{id}$
I_1	$= goto(I_0, E)$	$E' \longrightarrow E\bullet$ $E \longrightarrow E\bullet +T$
I_2	$= goto(I_0, T)$	$E \longrightarrow T\bullet$
I_3	$= goto(I_0, \mathtt{id})$	$T \longrightarrow \mathtt{id}\bullet$
I_4	$= goto(I_1, +)$	$E \longrightarrow E+\bullet\ T$ $T \longrightarrow \bullet\ \mathtt{id}$
I_5	$= goto(I_4, T)$	$E \longrightarrow E+T\bullet$

Action Table / Goto table

	id	+	$	E	T
0	$S, 3$			1	2
1		$S, 4$	A		
2	$R3$	$R3$	$R3$		
3	$R4$	$R4$	$R4$		
4	$S, 3$				5
5	$R2$	$R2$	$R2$		

There are in fact two different tables

The ACTION table tells you if you should shift, reduce, accept, or throw and error.

It additionally tells you how to update the state on a shift

The GOTO table tells you how to update the state on a reduce action

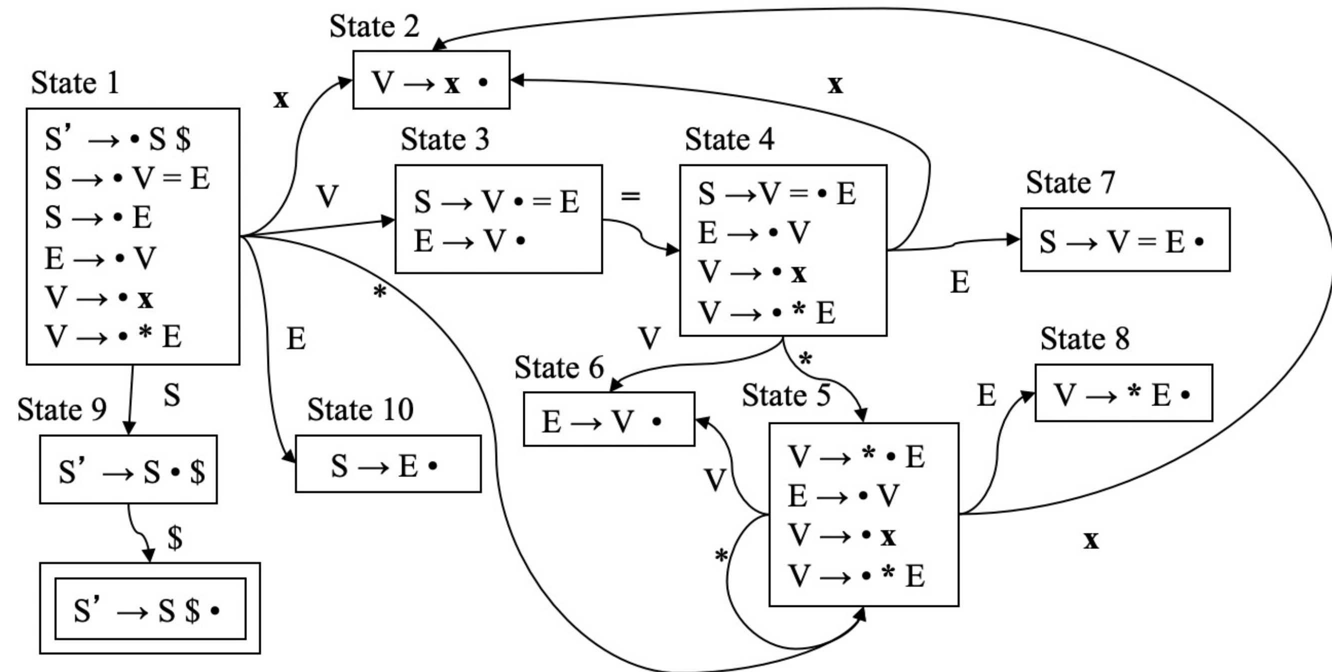
Quiz: States of an LR(0) parser

	example
0	$S' \rightarrow S\ \$$
1	$S \rightarrow V = E$
2	$S \rightarrow E$
3	$E \rightarrow V$
4	$V \rightarrow \mathbf{x}$
5	$V \rightarrow * E$

Example: States of an LR(0) parser

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E



Closure: What other items are "equivalent" to the given item?
Given an item A → α• Bβ, closure(A → α• Bβ) is the smallest set that contains the item **A → α• Bβ**, and every item in closure(**B → • γ**) for every production B → γ ∈ CFG

Things behind the automation: NFA to DFA

For example, to match [S → • V = E] we need to first match V. Let us connect the NFA states together with ε-transitions whenever one state needs to make a "subroutine" call to another state.

example

0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E