Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Shift-Reduce Parsing

Parsing Techniques

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

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

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

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step, a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

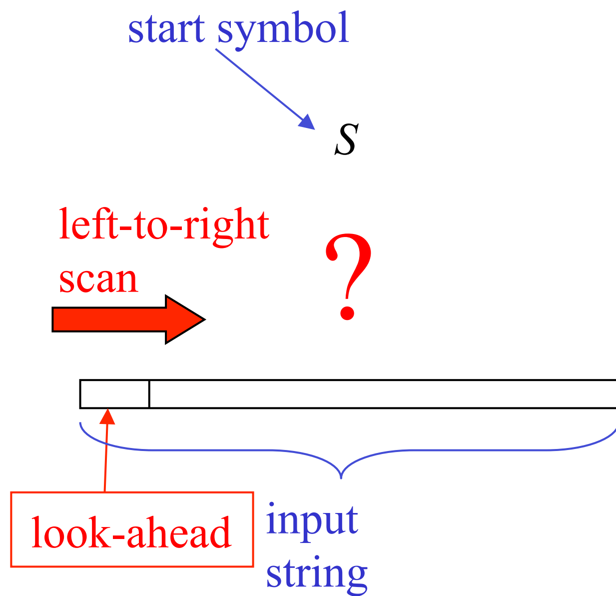
Bottom-Up Parsers

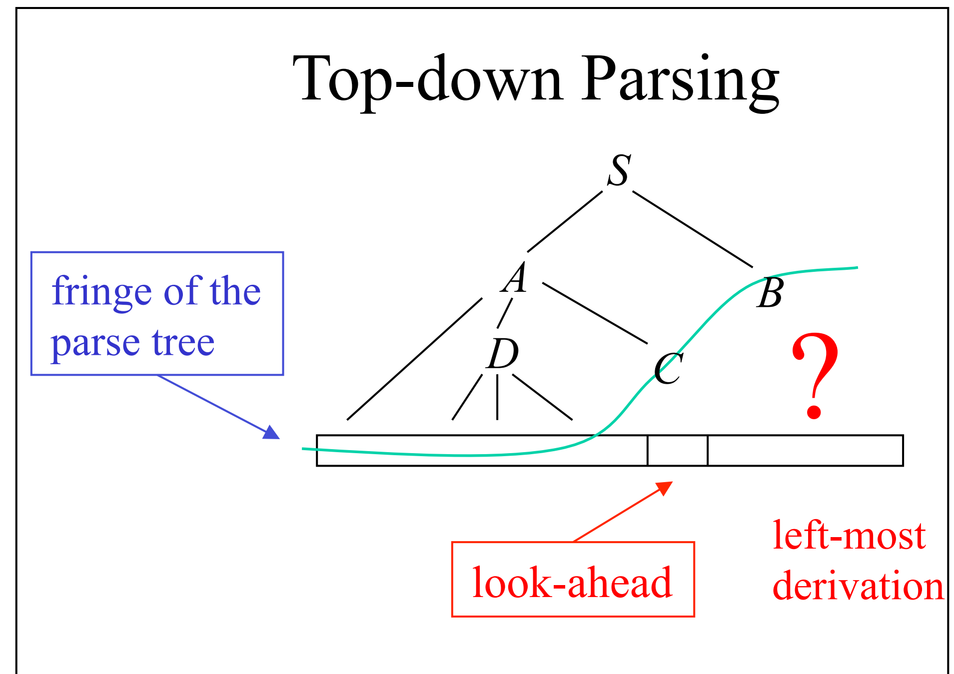
- Bottom up parsers handle a larger class of useful grammars
  Why is that?

- With both techniques, we are trying to build a tree that connects the input string to our start symbol (S) with a parse tree

- Let's look at what what each technique looks like halfway through parsing an input string

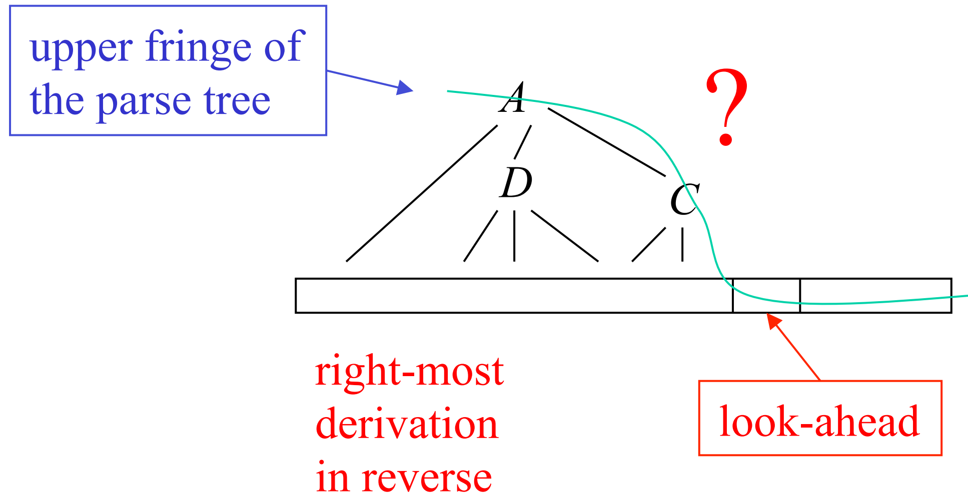
Top-Down Parsing





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

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

From the Bottom Up

- How does a bottom-up parser work
  - The main idea of bottom-up parsing is to find the rightmost-derivation of a sentence, by running the productions backwards from sentence to the start symbol

A rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

- In the above, w is derived from S via the sentential forms $\gamma_0 \ldots \gamma_n$
  What we are going to do is start with w, and figure out what γ_n leads to w, and then we will replace w with γ_n. Then, we will figure out what γ_{n-1} leads to γ_n and so forth until we reach S.

Bottom-up Parsing

The point of parsing is to construct a *derivation*

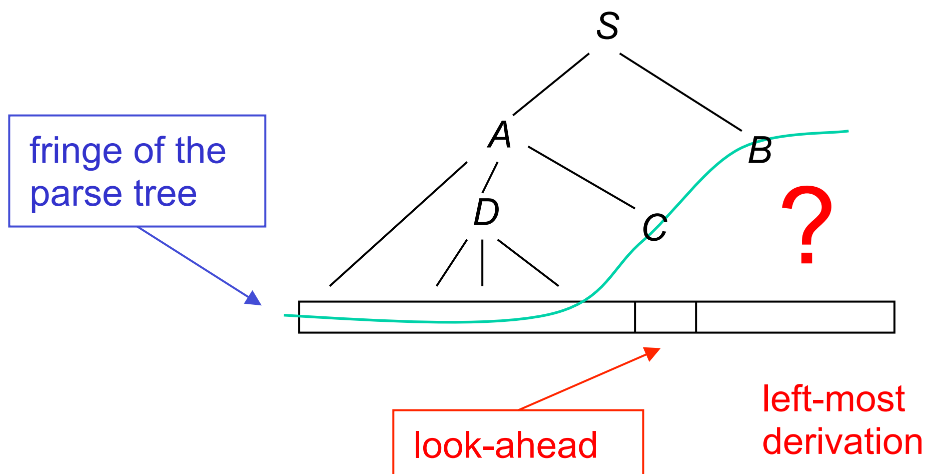
A derivation consists of a series of rewrite steps

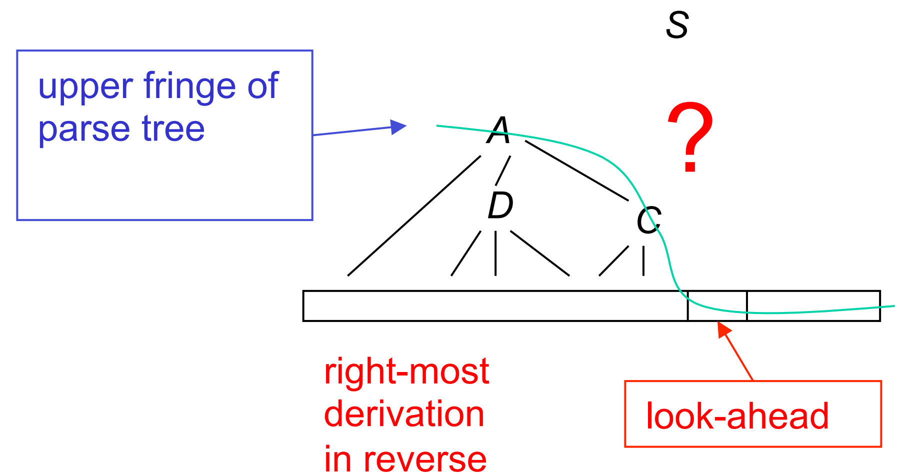
$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow sentence$$

- Each γ_i is a sentential form
  - If α contains only terminal symbols, α is a sentence in $L(G)$
  - If α contains ≥ 1 non-terminals, α is just a sentential form
- To get γ_i from γ_{i-1}, expand non-terminal $\alpha \in \gamma_{i-1}$ by using a production $\alpha \rightarrow \beta$
  - Replace the occurrence of $\alpha \in \gamma_{i-1}$ with β to get γ_i
  - In a leftmost derivation, it would be the first $\alpha \in \gamma_{i-1}$

A *left-sentential form* occurs in a *leftmost* derivation
A *right-sentential form* occurs in a *rightmost* derivation

Parsing Techniques





Finding Reductions (Handles)

The parser must find a substring $\beta\mu\delta$ of the tree's fringe that matches some production $\alpha \rightarrow \beta\mu\delta$ that occurs as one step in the rightmost derivation. Informally, we call this substring $\beta\mu\delta$ a *handle*

Formally, **(IMPORTANT)**

- A *handle* of a right-sentential form γ is a pair $< \alpha \rightarrow \beta\mu\delta , \boldsymbol{k} >$ where $\alpha \rightarrow \beta\mu\delta$ is a production and *k* is the position in γ of $\beta\mu\delta$'s rightmost symbol.
- If $< \alpha \rightarrow \beta\mu\delta , \boldsymbol{k} >$ is a handle, then replacing $\beta\mu\delta$ at *k* with α produces the right sentential form preceding γ in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

$\Rightarrow$ The parser doesn't need to scan past the handle (needs only a look-ahead)

Finding Reductions (Handles)

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

2 $\Rightarrow$ a unique production $\alpha \rightarrow \beta\mu\delta$ applied to take γ_{i-1} to γ_i

3 $\Rightarrow$ a unique position ***k*** at which $\alpha \rightarrow \beta\mu\delta$ is applied

4 $\Rightarrow$ a unique handle < $\alpha \rightarrow \beta\gamma\delta$, ***k*** >

This all follows from the definitions

Expression Example

1	*S*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr – Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	num
9		|	id

The expression grammar

Sentential Form	*Handle Prod'n , Pos'n*
S	—
Expr	1,1
Expr – Term	3,3
*Expr – Term * Factor*	5,5
Expr – Term * <id,y>	9,5
Expr – Factor * <id,y>	7,3
Expr – <num,2> * <id,y>	8,3
Term – <num,2> * <id,y>	4,1
Factor – <num,2> * <id,y>	7,1
<id,x> – <num,2> * <id,y>	9,1

Handles for rightmost derivation of input:

x – 2 * y

If we start with our input string, we can work backwards to S

Handle-pruning, Bottom-up Parsers

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

- Handle pruning forms the basis for a bottom-up parsing method

- To construct a rightmost derivation

  $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$

- Apply the following simple algorithm

  for i $\leftarrow$ *n to 1 by -1*

  Find the handle $< \alpha_i \rightarrow \beta_i , \mathbf{k}_i >$ in γ_i

  Replace β_i *with* α_i *to generate* γ_{i-1}

Shift-Reduce Parsing

- Let's say that we have a way of finding the handles (which we will discuss later), how can I build a parser that can will actually do bottom up parsing?
- The easiest way to implement it is with "shift-reduce" parsing
- Shift-reduce parsing makes use of two data structures, a stack and an input buffer
- At each point in time, you have two main choices
  - **Shift** (which eats input terminals and moves them on the stack)
  - **Reduce** (which applies some production in reverse, replacing some right hand side (β) with the corresponding left hand side (α)
- The other two options you have are:
  - **Accept**, in which case you are all done
  - **Error** when something has gone wrong

Shift-Reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  Locate left end of handle within the stack
  Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & Error are simple
Shift is just a push and a call to the scanner
Reduce takes |*rhs*| pops & 1 push

If handle-finding requires state, put it in the stack

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

An Example

- Let us run "x-2*y" through our shift-reduce parser
- We will use the expression grammar from before (which has all of the handles we already listed by hand)
- At each step, we will either shift a new terminal onto the stack (from the input buffer) or we will reduce some right-hand side β to a left-hand side α
- When we are done, we should have nothing left in our input and $S should be left of the stack (which means we successfully reduced our input to our start symbol).

Back to x - 2 * y Example

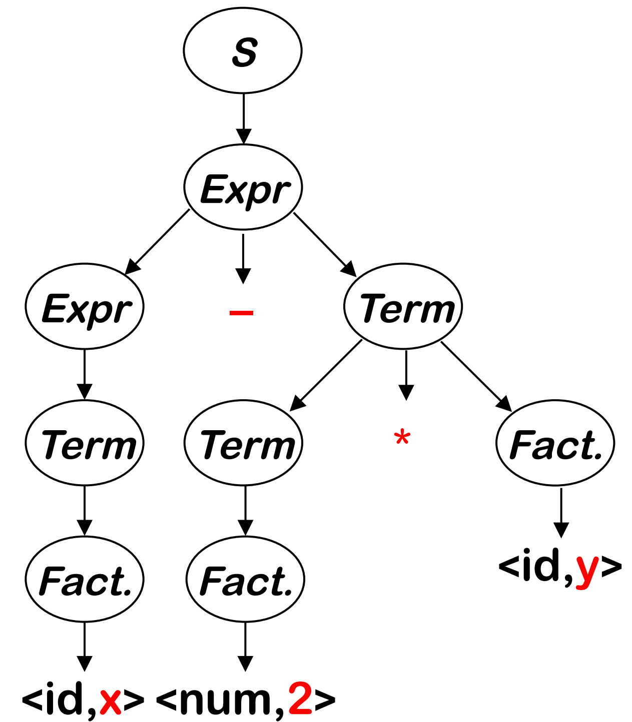
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *S*		*none*	accept

1. Shift until top-of-stack is the right end of a handle

2. Reduce: Pop the right-hand side of the production that corresponds to the handle from the stack, and push the left hand side (to figure out if it is a handle, check the handles we highlighted in the derivation from before)

Example: Corresponding Parse Tree

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *S*		*none*	accept



LR Parsers

- Shift-reduce parsers are very fast and simple to implement
  - They keep moving more input onto the stack
  - All the while they look for handles to reduce
  - The tricky part is recognizing the handles

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

- 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.

LR Parsers

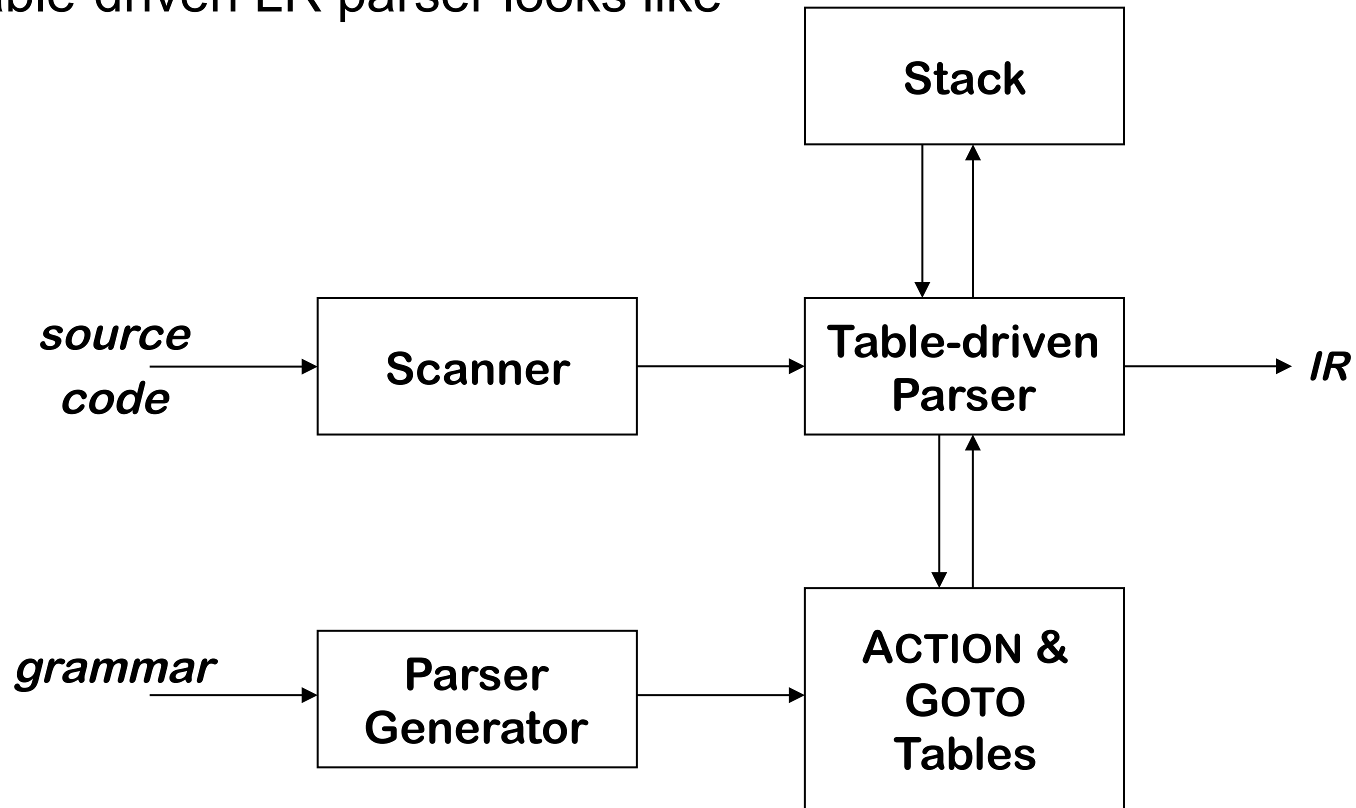
- Then the hard part is figuring out the language of handles that we need to recognize
  - There are several different ways of doing this: SLR, LALR, LR each of varying power and complexity

- At the end of the day, they all generate a state machine

  in the form of a table

LR Parsers

A table-driven LR parser looks like



LR Parser Tables

- SLR, LALR, and LR are all different ways of automatically generating a state machine to capture handles encoded into the form of a table. However, unlike a normal state machine, this one is searching the top of a stack

- 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

- The other major difference is that now we not only push symbols onto the stack, we also push an associated state onto the stack with each symbol (always in pairs: 1 symbol and 1 state (just a number))

LR Shift-Reduce Parsers


```
push($);    // $ is the end-of-file symbol
push(s0);   // s0 is the start state of the DFA that recognizes handles
lookahead = get_next_token();
repeat forever
  s = top_of_stack();
  if ( ACTION[s,lookahead] == reduce α→β ) then
       pop 2*|β| symbols;
       s = top_of_stack();
       push(α);
       push(GOTO[s,α]);
  else if ( ACTION[s,lookahead] == shift si ) then
       push(lookahead);
       push(si);
       lookahead = get_next_token();
  else if ( ACTION[s,lookahead] == accept and lookahead == $ )
     then return success;
  else error();
```

The skeleton parser

- **uses ACTION & GOTO**
- **does |*words*| shifts**
- **does |derivation| reductions**
- **does 1 accept**

LR Shift-Reduce Algorithm

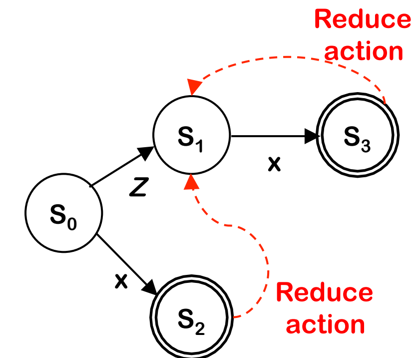
- First part: Initialization
  - Push $ and the specified init state (usually called s_0)
- Clause: Shift
  - We consult the ACTION table, and see if it tells us to do a shift
  - If it does, then we move the look-ahead onto the stack and update the state with the value read from the ACTION table
- Last Clauses:
  - The final part of the algorithm checks for accept and error

LR Shift-Reduce Algorithm

- Clause: Reduce
  - We consult the ACTION table, and see if it tells us to do a reduce
  - We index the ACTION table with the state on top of the stack and our look-ahead symbol
  - If we are to do a reduction we pop all the symbols that match β along with all the paired state variables (hence the 2*). For example if we are doing the reduction $X \rightarrow YZ$ then $|\beta| = 2$ and the top of the stack would look like "Y s5 Z s6" so we pop 2*2=4 things off the stack
  - We then push α onto the stack (which would be X in $X \rightarrow YZ$) and the some new state that is found in the GOTO table

LR Parsers (Parse Tables) Simple Example

To make a parser for *L(G)*, we need a set of tables



Control DFA for the simple example

The grammar

1	*S*	→	*Z*
2	*Z*	→	*Z* **x**
3		|	**x**

The tables

ACTION

State	$	x
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO

State	*Z*
0	1
1	
2	
3	

Simple Example Parses

The string "x"

Stack	Input	Action
$\$ s_0$	x \$	shift 2
$\$ s_0 \text{ x } s_2$	\$	reduce 3
$\$ s_0 Z s_1$	\$	accept

The string "xx"

Stack	Input	Action
$\$ s_0$	x x \$	shift 2
$\$ s_0 \text{ x } s_2$	x \$	reduce 3
$\$ s_0 Z s_1$	x \$	shift 3
$\$ s_0 Z s_1 \text{ x } s_3$	\$	reduce 2
$\$ s_0 Z s_1$	\$	accept

A More Complex Example

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

S8 means shift and add state 8 to the top of stack

R4 means reduce by production number 4

A means accept, and an empty square should indicate an error

	ACTION				GOTO		
	X	*	=	$	S	E	V
1	S8	S6			2	5	3
2				A			
3			S4	R3			
4	S8	S6				9	7
5				R2			
6	S8	S6				10	7
7			R3	R3			
8			R4	R4			
9				R1			
10			R5	R5			

A more complex example

x = * x $

look-ahead	Stack:
	$ s1
x	$ s1 x s8
=	$ s1 V s3
=	$ s1 V s3 = s4
*	$ s1 V s3 = s4 * s6
x	$ s1 V s3 = s4 * s6 x s8
$	$ s1 V s3 = s4 * s6 V s7
$	$ s1 V s3 = s4 * s6 E s10
$	$ s1 V s3 = s4 V s7
$	$ s1 V s3 = s4 E s9
$	$ s1 S s2

s8 comes from ACTION[s1,x]

s3 comes from GOTO[s1,V]

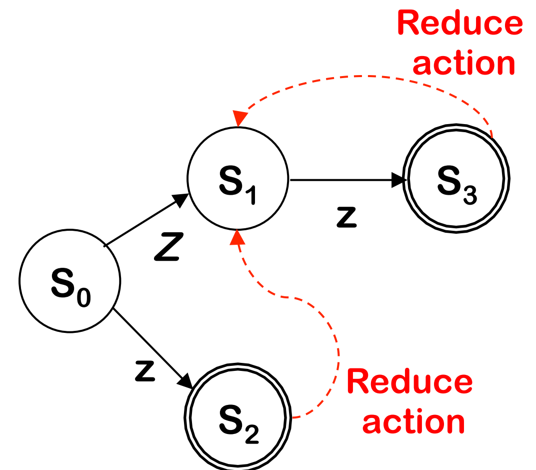
Accept!

Make, sure that you understand when it is shifting, when in is reducing and where the states come from.

LR Parsers

How does this LR stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include TOS
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular
  - Build a handle-recognizing DFA
  - ACTION & GOTO tables encode the DFA
- To match sub-terms, recurse and leave DFA's state on stack
- Final states of the DFA correspond to *reduce* actions
  - New state is GOTO[*lhs* , state at TOS]
  - For *Z*, this takes the DFA to S_1



Control DFA for the simple example

How do we recognize Handles?

- The complete left context in conjunction with the look-ahead tell us if we have recognized a handle or not

Stack	Input	Handle	Action
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *S*		*none*	accept

- This grammar requires us to look at just the next one terminal to make our decision if the TOS is a handle or not.
- This is an LR(1) grammar

How do we recognize Handles?

- We need to enumerate the list of handles based on the look-ahead (this list is finite), and then we need to build a recognizer that can check for this list on the TOS

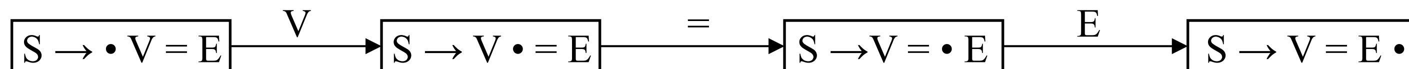
- The problem is that we need to recognize something on the top of a stack, NOT on a "stream" of tokens as before
  - We don't want to have to re-run the recognizer on the entire left context each and every time that we do a reduction
  - We need to 'save our work' on the stack

Building a Handle Recognizing Machine:
[for now, without look-ahead, which is LR(0)]

Understanding a Handle Recognizing Machine

Let's build an NFA (nondeterministic finite automaton) to recognize the right hand sides of our rule assuming that our machine will read both terminal **and** non-terminals

For example: Let's start with production 1 from our grammar



example

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

Each one of the squares above is an "item" which is a production with a special placeholder (the "•") that says how far along we are in matching this production. If the dot is at the start of the right-hand-side (RHS) then it means we should have the entire RHS on the top of the stack.

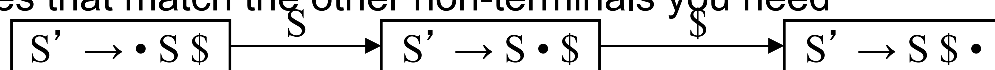
Understanding a Handle Recognizing Machine

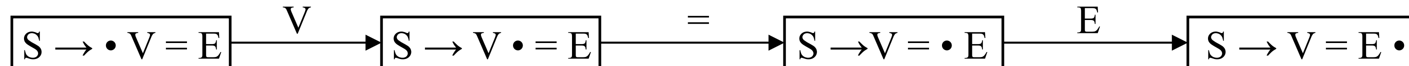
Now let's expand our example to look for each right-hand side we might see. If we hit the state [S → S $ •] then I know I am done.

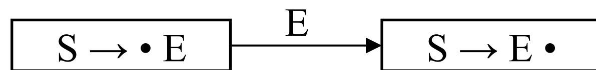
The big problem we are left with is that, we will never see a non-terminal (such as S) on the input stream. The solution to this is to make "subroutine" calls to the other state machines that match the other non-terminals you need

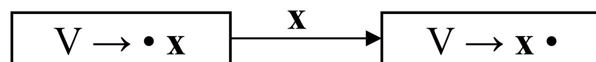
example

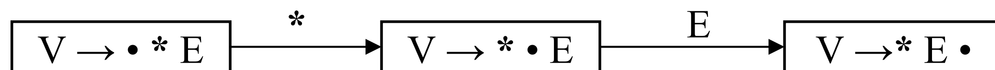
0 S' → S $
1 S → V = E
2 S → E
3 E → V
4 V → **x**
5 V → * E









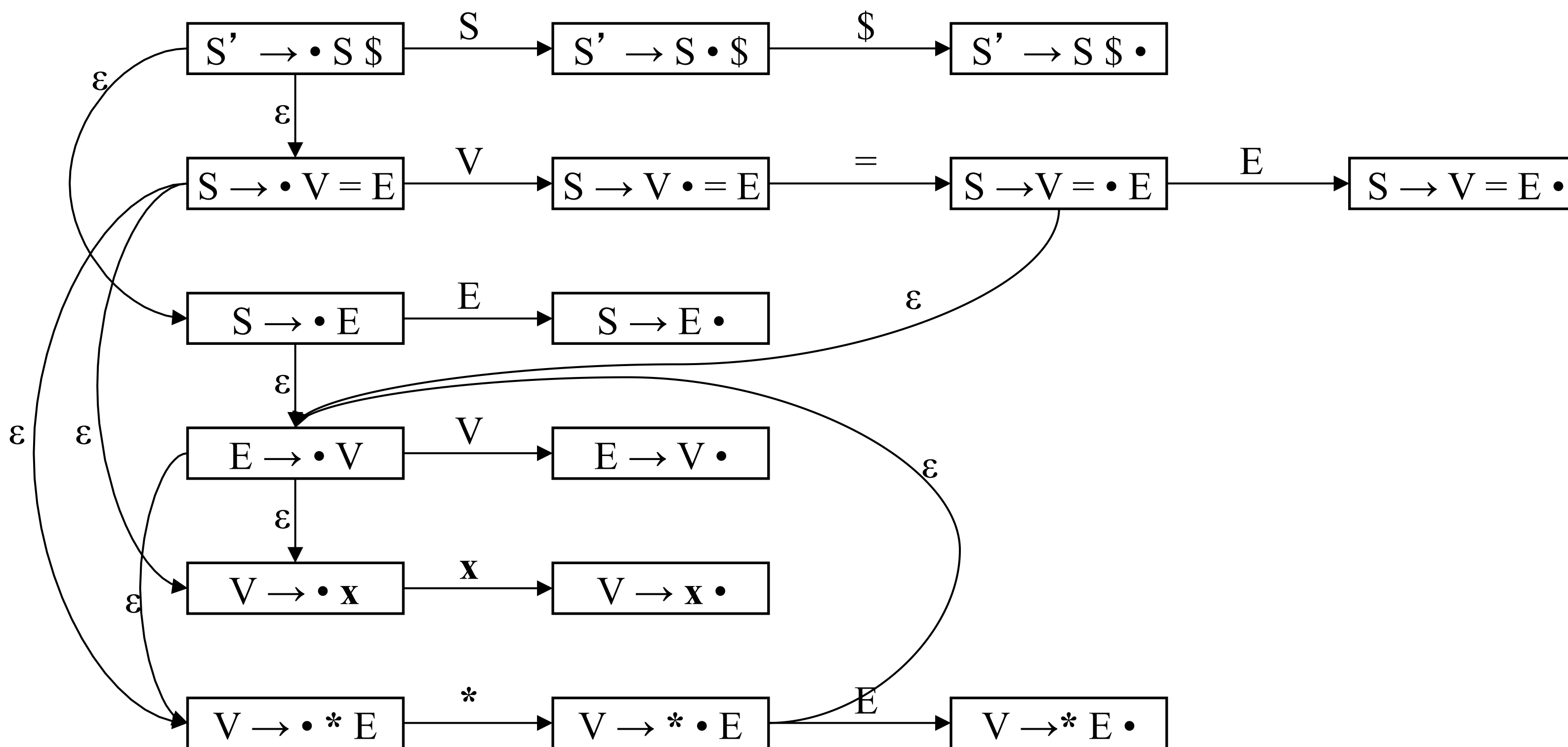


Understanding a Handle Recognizing Machine

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. If you need to review what an ε-transition is and how an NFA works, now is a good time.

example

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

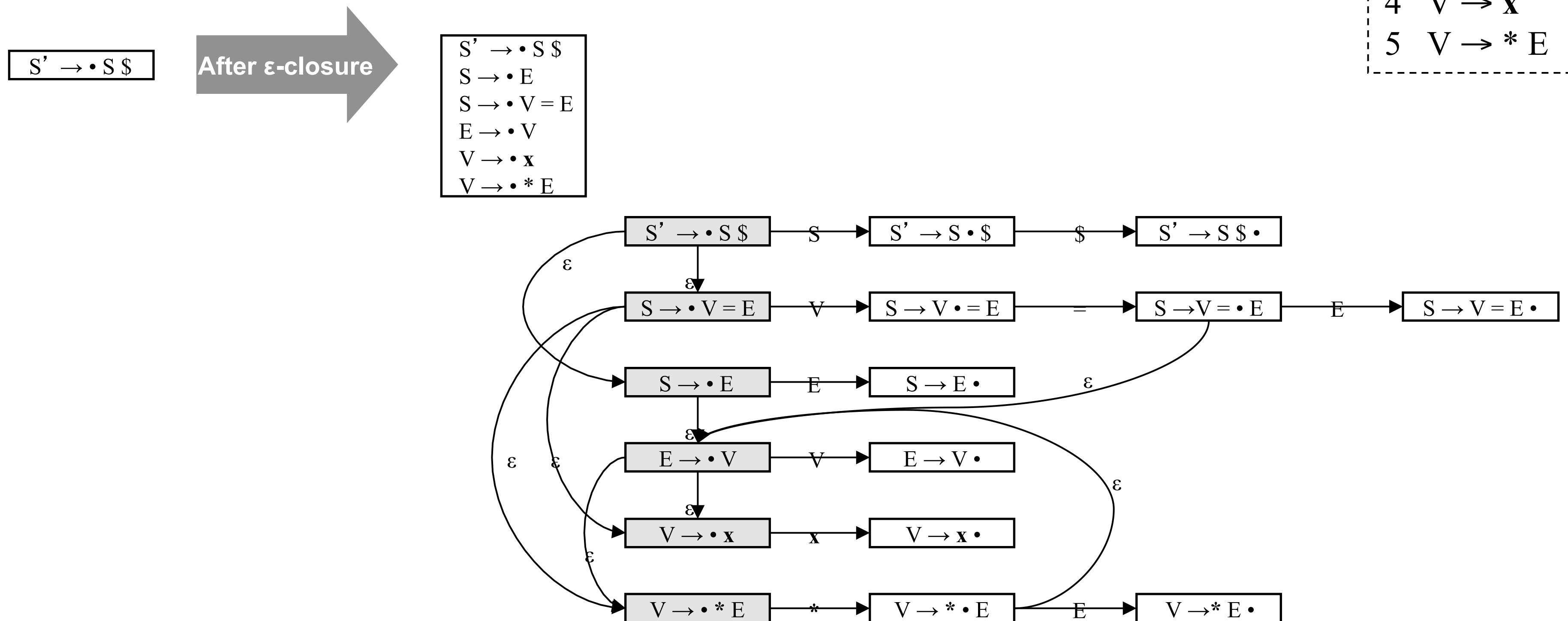


Understanding a Handle Recognizing Machine

Now we are going to build a real state machine from our NFA. Let's compute it as needed as we process the sentence "x=*x$" as we had before. Our first state starts with [S' → • S $] where we have not started parsing our start symbol yet.

example

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



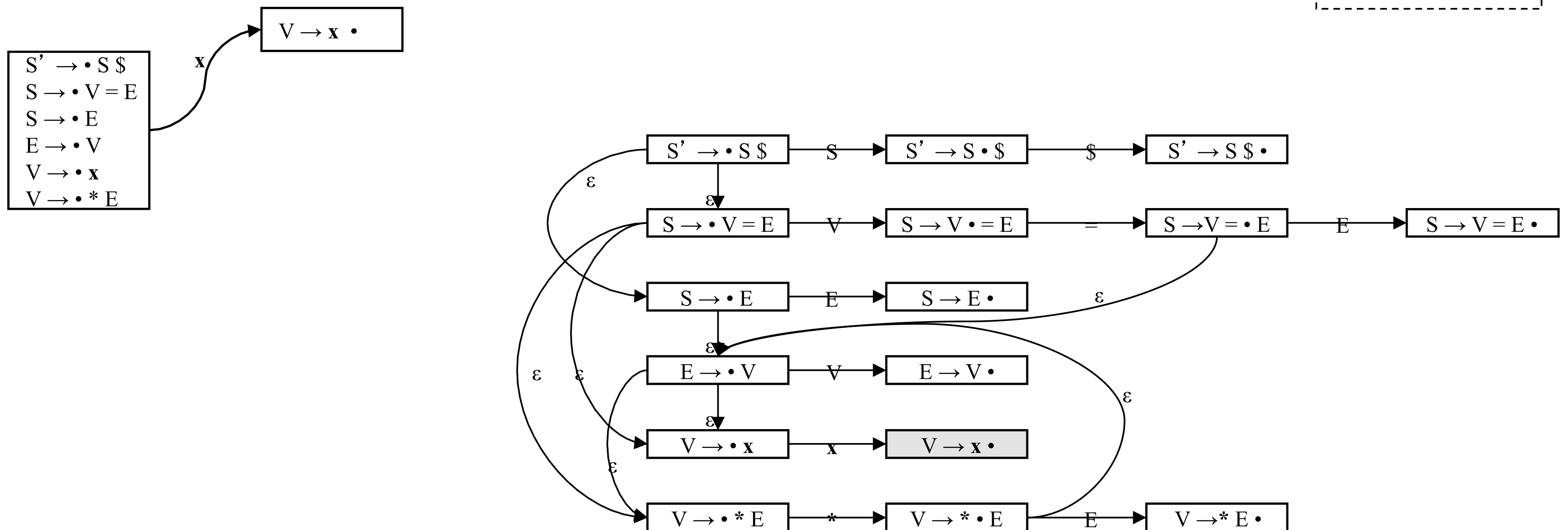
Understanding a Handle Recognizing Machine

example

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

What state will our machine be in after we read the “x” in “x=*x$”? If we look to the NFA, it should now be in a state [V → **x** •] which means that we have seen the complete right hand side of our first non-terminal!

This means we have hit a terminal, and we need to “pop” x off the stack and end up back in State 0.

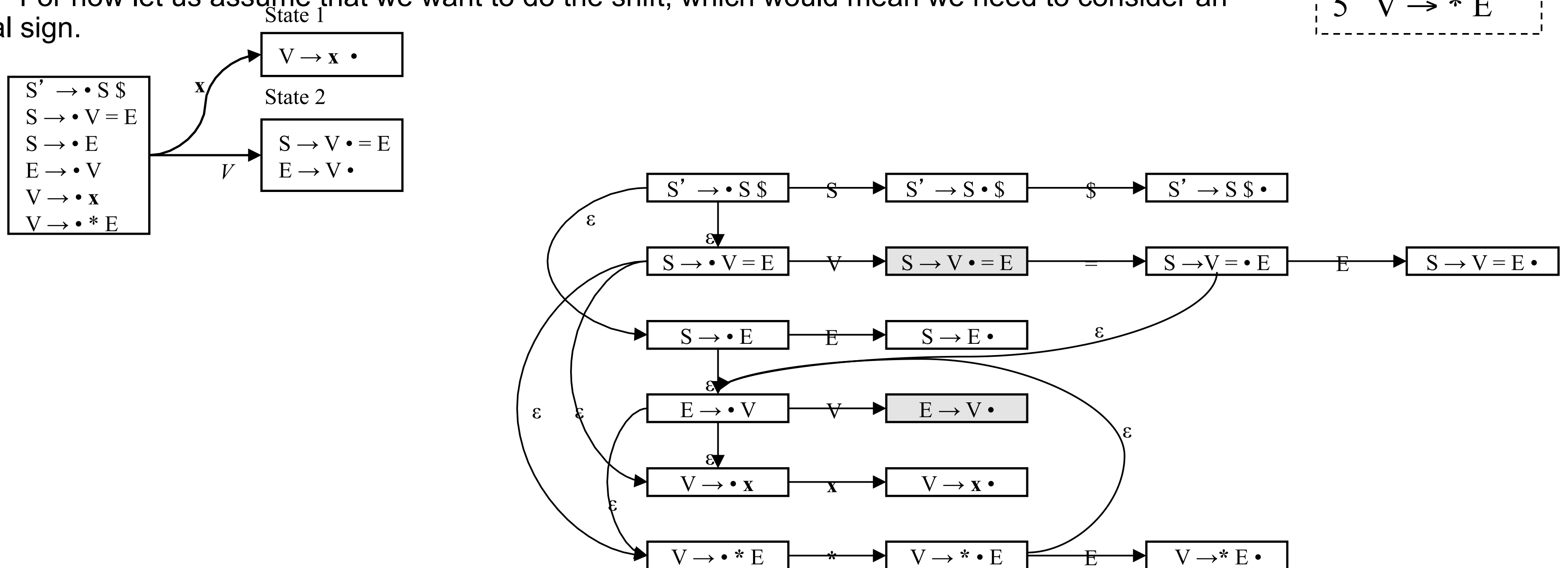


Understanding a Handle Recognizing Machine

After we pop “x” off the stack we end up in state 0, but now we recognize a “V” which comes from the left hand side of the production “V → **x**”.
To handle this we need to add a new edge from State 0, that is traversed when we found our “V”. Which of the items from the NFA below for State 0 were looking for a “V”? There are exactly 2: “S → • V = E” and “E → • V”. After following the V edge of these two states we end up in a new state. This state is shown below. Now we have two choices, should we reduce (E →V) or should we shift the “=“? For now let us assume that we want to do the shift, which would mean we need to consider an equal sign.

example

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



Understanding a Handle Recognizing Machine

If we consider the last state (2) and then consider what happens if we find a "=", we will need a new state. Following the equal sign gives us the item "S →V = • E", and taking the ε-closure of this will

give us our new state 3.

example

0 S' → S $

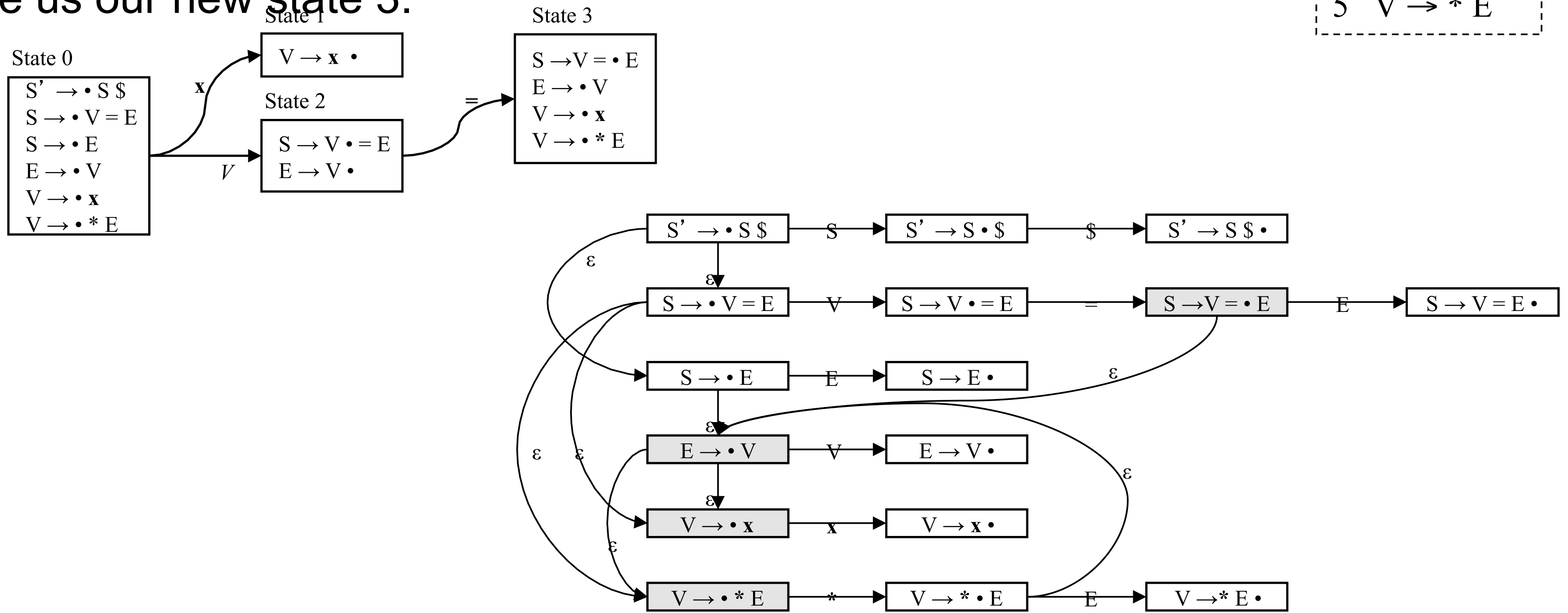
1 S → V = E

2 S → E

3 E → V

4 V → **x**

5 V → * E



Understanding a Handle Recognizing Machine

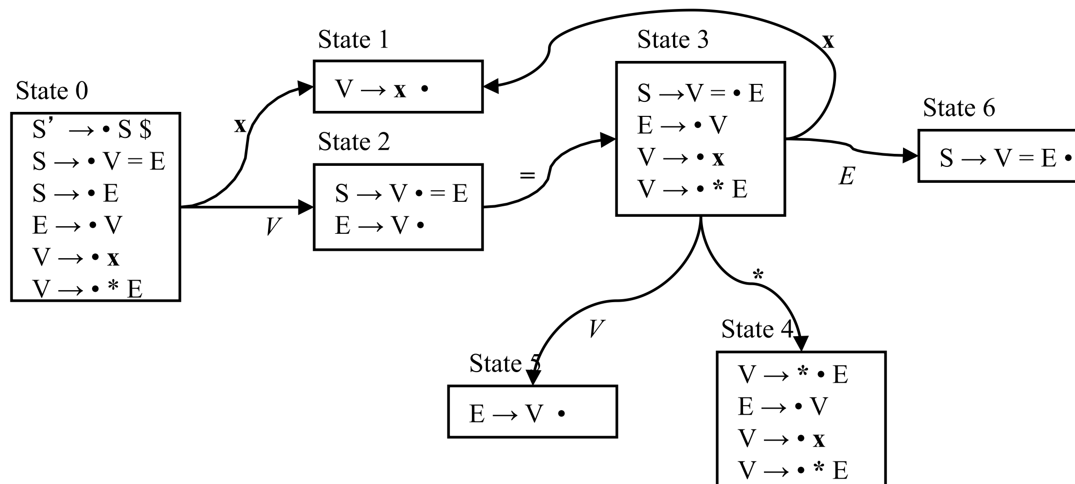
example

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

At each point in time, we are going to build out our state machine by considering what the possible out edges are for each new state.

To figure out what each possible out edge is, we need to look at all the symbols that are to the right of the dot. For example, from state 1 there is no symbol the right of the dot. For State 2, there is only one thing to the right of the dot, the "=" sign. The list of out edges from State 3 are "E", "V", "x", and "*". Let's start with "x". The "x" will lead to a state with the item "V → • **x**". But we *already* have such
an state, it is called state 1.

We can grow the new states for "E", "V", and "*" in a similar way.



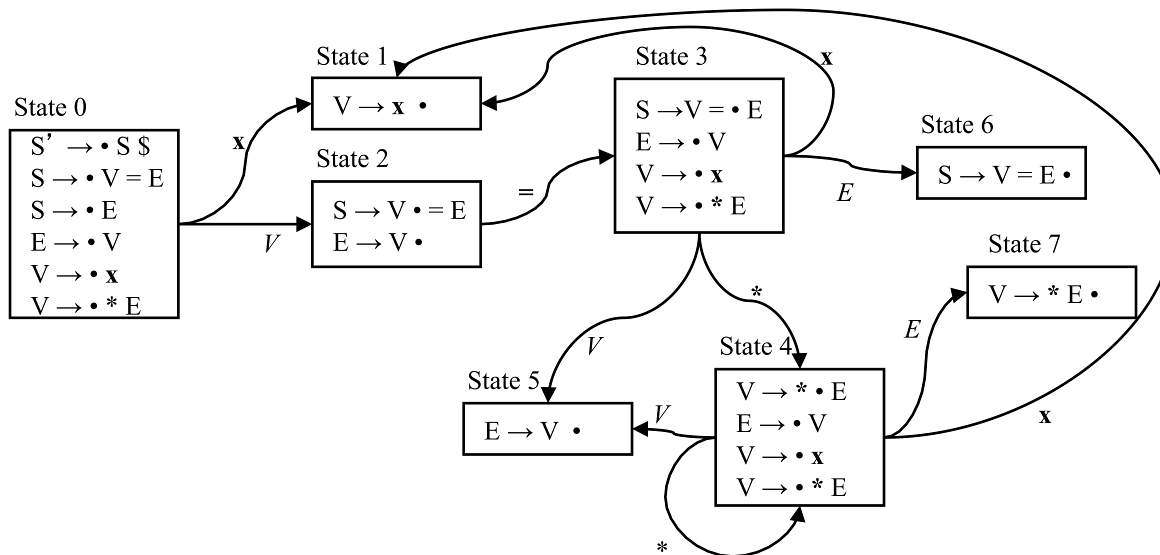
Understanding a Handle Recognizing Machine

In the same way we can build the states that stem from state 4. State 4 should have out edges for “E”, “V”, “x”, and “*” because all of those symbol appear to the right of the “•”.

In this case, there are many “new” states that are the same as existing states, so we just link them together as we did from State 3 to State 1 for “x”.

example

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



Understanding a Handle Recognizing Machine

Let's go back and re-evaluate State 0. What are the out edges do we need to consider for State 0? We should have out edges for "S", "V", "x", and "*". The
edge that goes from State 0 to State 8 with an S get's us close to done.

Finally, if we end up in state 9 then we are done! That means we have seen successfully parsed S, and all that we are left with is the end of file!

example

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

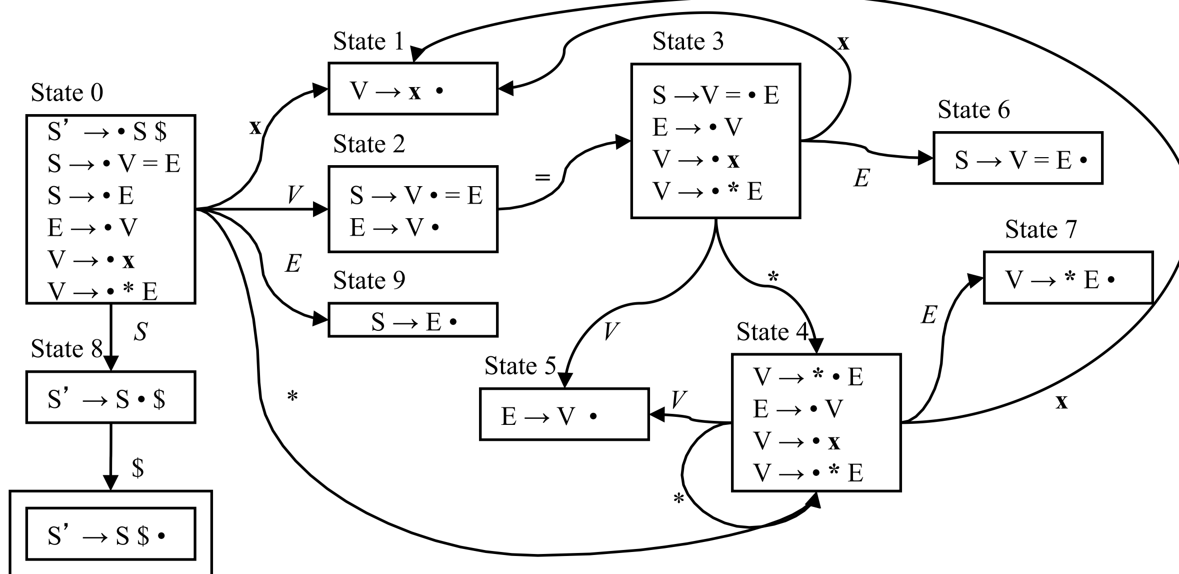


Table Example

	ACTION				GOTO		
	x	*	=	$	S	E	V
0	S1	S4			8	9	2
1			R4	R4			
2			S3/R3	R3			
3	S1	S4				6	5
4	S1	S4				7	5
5			R3	R3			
6				R1			
7			R5	R5			
8				A			
9				R2			

example

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

Table Example

Shift/Reduce conflict will choose to shift

	ACTION				GOTO		
	x	*	=	$	S	E	V
0	S1	S4			8	9	2
1			R4	R4			
2			S3/R3	R3			
3	S1	S4				6	5
4	S1	S4				7	5
5			R3	R3			
6				R1			
7			R5	R5			
8				A			
9				R2			

example

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

Building LR Parsers

How do we generate the ACTION and GOTO tables?

- Use the grammar to build a model of the handle recognizing DFA
- Use the DFA model to build ACTION & GOTO tables
- If construction succeeds, the grammar is LR

How do we build the handle recognizing DFA ?

- Encode the set of productions that can be used as handles in the DFA state – Also use LR(k) items
- Use two functions *goto(s,* α *)* and *closure(s)*
  - *closure()* finds the sets of "equivalent" parsing states
  - *goto()* figures out what state will result after we match something
- Build up the states and transition functions of the DFA
- Use this information to fill in the ACTION and GOTO tables