Programming Languages
Lecture06 – Bottom-Up Parsing

남 범 석

bnam@skku.edu

Bottom-up Parsing (definitions)

- Bottom-up parsing and reverse rightmost derivation
  - A derivation consists of a series of rewrite steps
  - A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S
  - $S \Rightarrow \gamma 0 \Rightarrow \gamma 1 \Rightarrow \gamma 2 \Rightarrow \ldots \Rightarrow \gamma n-1 \Rightarrow \gamma n \Rightarrow$ sentence

- We construct a parse tree from leaves to root
  - The reduction steps trace a rightmost derivation on reverse.

Leftmost Derivation vs Rightmost Derivation

- Leftmost Derivation
  - Apply production to the leftmost variable in each step
- Rightmost Derivation
  - Apply production to the rightmost variable in each step
- Ex.) S→aABCd
  - Leftmost derivation tree expands A first
  - Rightmost derivation tree expands C first

Finding Reductions (handles)

- Parser must find a substring β of the tree's frontier
  - Matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation
  - Informally, we call this substring β a *handle*
- Formally,
  - A *handle* of a right-sentential form γ is a pair $\langle A \rightarrow \beta, \boldsymbol{k} \rangle$
    - $A \rightarrow \beta \in P$
    - k is the position of β's rightmost symbol in γ (right-sentential form).
  - If $\langle A \rightarrow \beta, \boldsymbol{k} \rangle$ is a handle, then replace β at k with A
- Handle pruning
  - The process of discovering a handle & reducing it to the appropriate left-hand side (non-terminal) is called *handle pruning*
  - Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

Bottom-Up Parser Example

The expression grammar

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

Handles for rightmost derivation of x - 2 * y

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

Reverse rightmost derivation (RRD)

Handles are colored in blue

Bottom-Up Parsers

- Bottom-up parsers
  - Shift-Reduce Parser, also known as LR Parser
  - LR(k)
    - Most powerful deterministic bottom-up parsing using k lookaheads
  - SLR(k)
  - LALR(k)
- Components
  - Parse Stack
  - Shift-Reduce Driver
  - Action Table
  - Goto Table

Shift-Reduce Parser

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack can reduce using a handle <A→β.k> then
        // reduce β to A
            pop |β| (=k) symbols off the stack
            push A onto the stack
    else if (token != EOF) then
        // shift
            push token
            token ← next_token( )
    else // need to shift, but out of input
        report an error
```

How do errors show up?

- **failure to find a handle**
- **hitting EOF & needing to shift (final else clause)**

Either generates an error

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id		

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

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		

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

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		

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

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		

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

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			

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x – 2 * y

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
$ *Goal*		*none*	accept

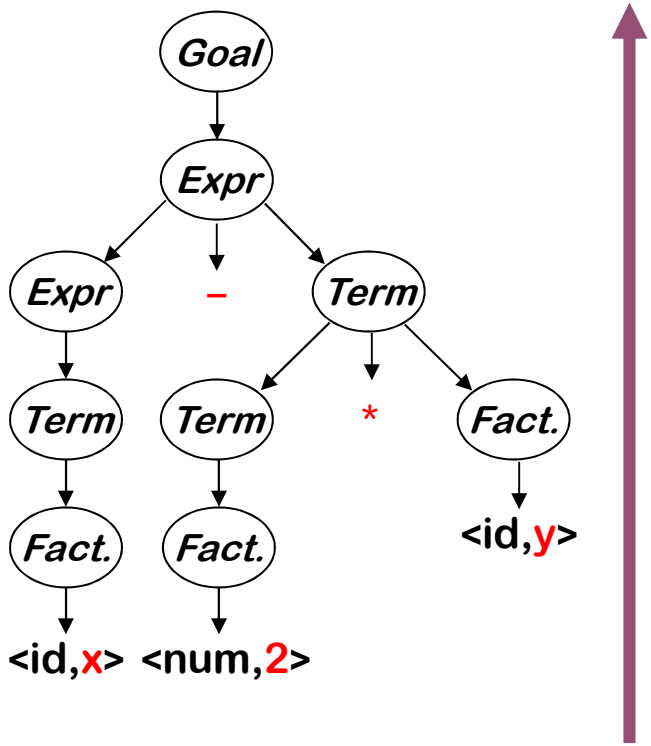
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
10		|	(*Expr*)

5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle within stack & reduce

Example

Stack	Input	Action
$	id – num * id	shift
$ id	– num * id	red. 9
$ *Factor*	– num * id	red. 7
$ *Term*	– num * id	red. 4
$ *Expr*	– num * id	shift
$ *Expr* –	num * id	shift
$ *Expr* – num	* id	red. 8
$ *Expr* – *Factor*	* id	red. 7
$ *Expr* – *Term*	* id	shift
$ *Expr* – *Term* *	id	shift
$ *Expr* – *Term* * id		red. 9
$ *Expr* – *Term* * *Factor*		red. 5
$ *Expr* – *Term*		red. 3
$ *Expr*		red. 1
$ *Goal*		accept



bottom-up building

Shift-reduce Parsing

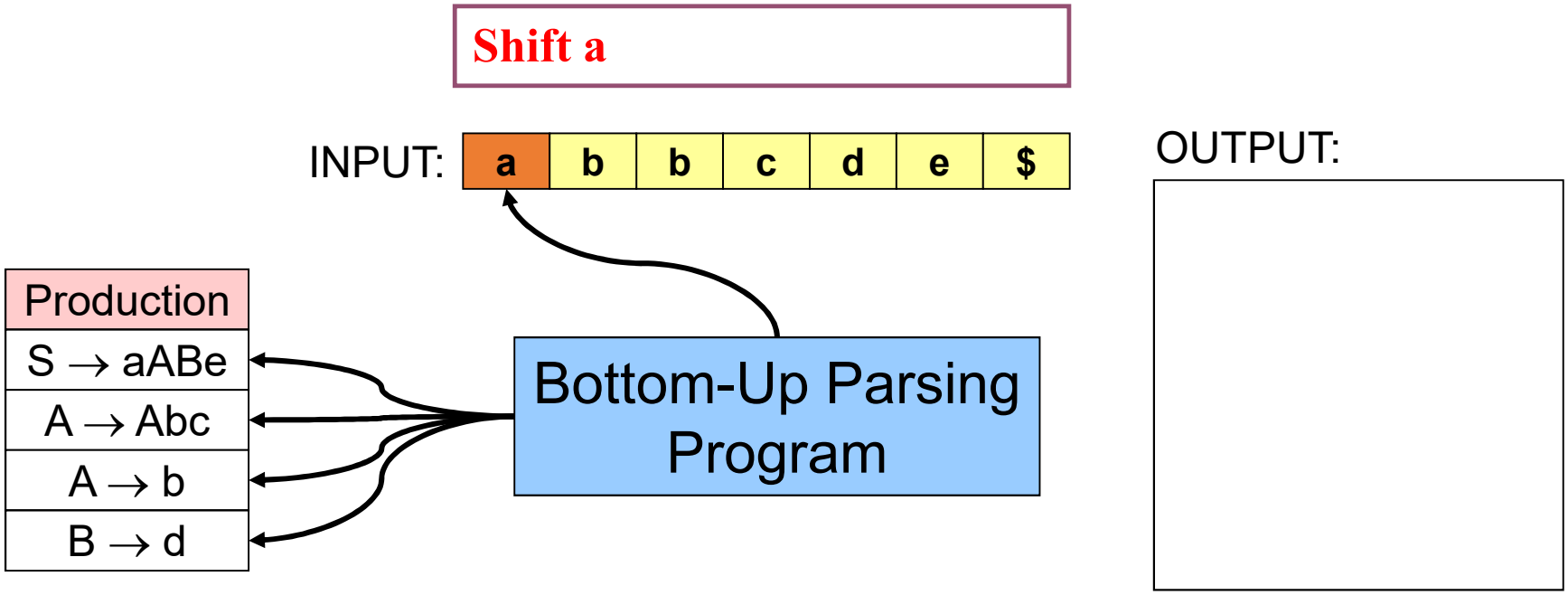
- Shift reduce parsers are easily built and easily understood
- A shift-reduce parser has just four actions
  - *Shift* — next word is shifted onto the stack
  - *Reduce* — right end of handle is at top of stack
    - Locate left end of handle within the stack
    - Pop handle off stack & push appropriate *lhs*
  - *Accept* — stop parsing & report success
  - *Error* — call an error reporting/recovery routine

Handle finding is key
- handle is on stack
- finite set of handles

⇒ use a DFA !

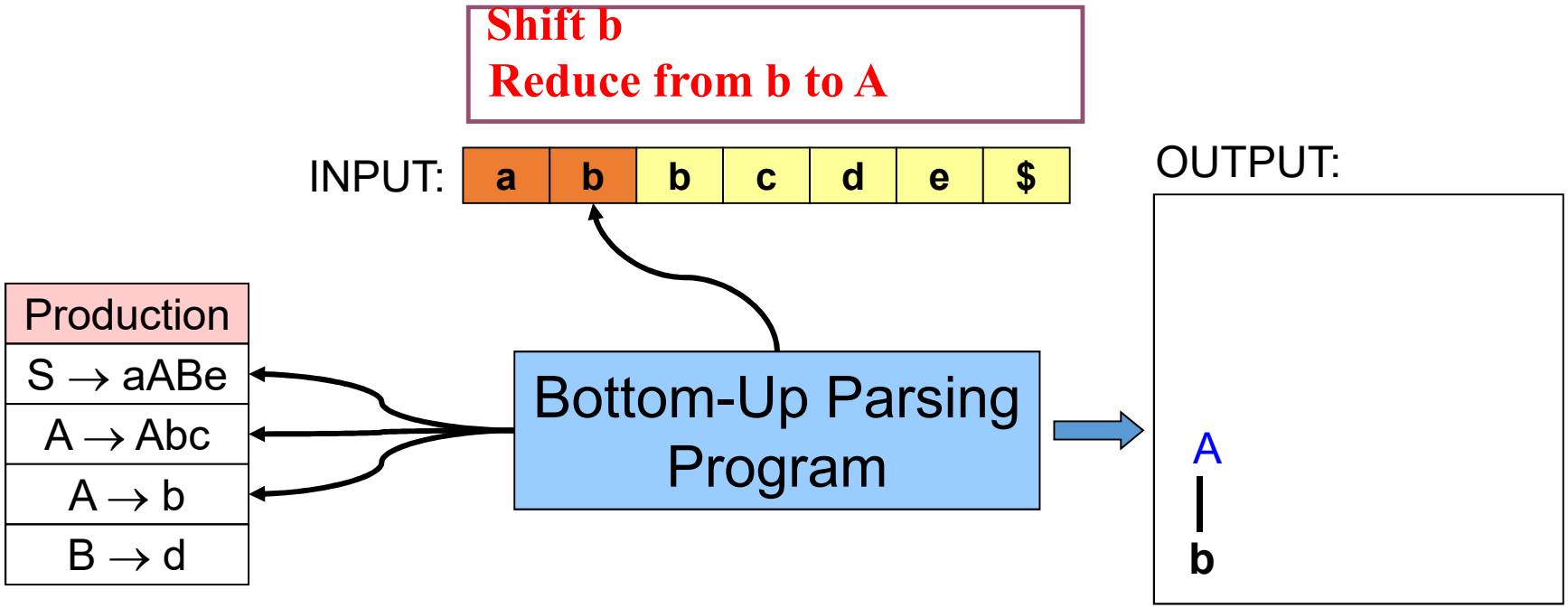
- Critical Question: How can we know when we have found a handle without generating lots of different derivations?
  - Answer: we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
  - *LR(1)* parsers build a DFA that runs over the stack & finds them

Shift-reduce Parsing in Action (1)



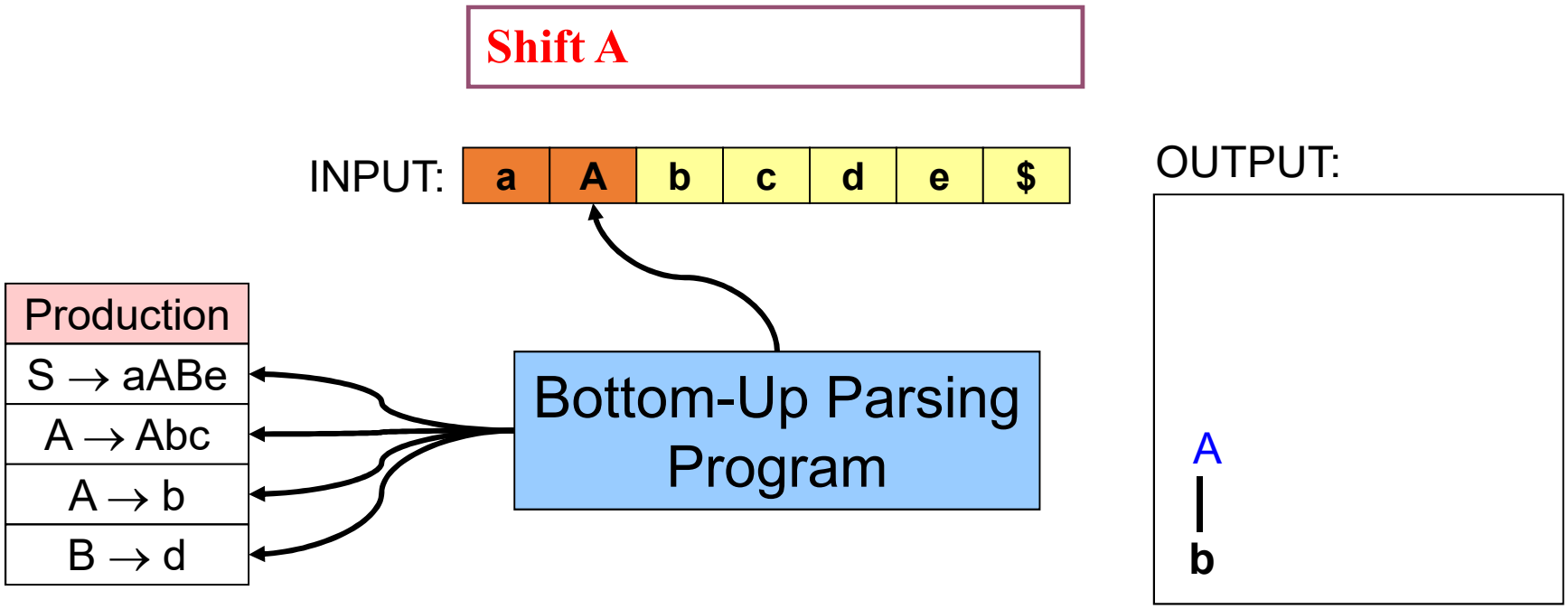
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (2)



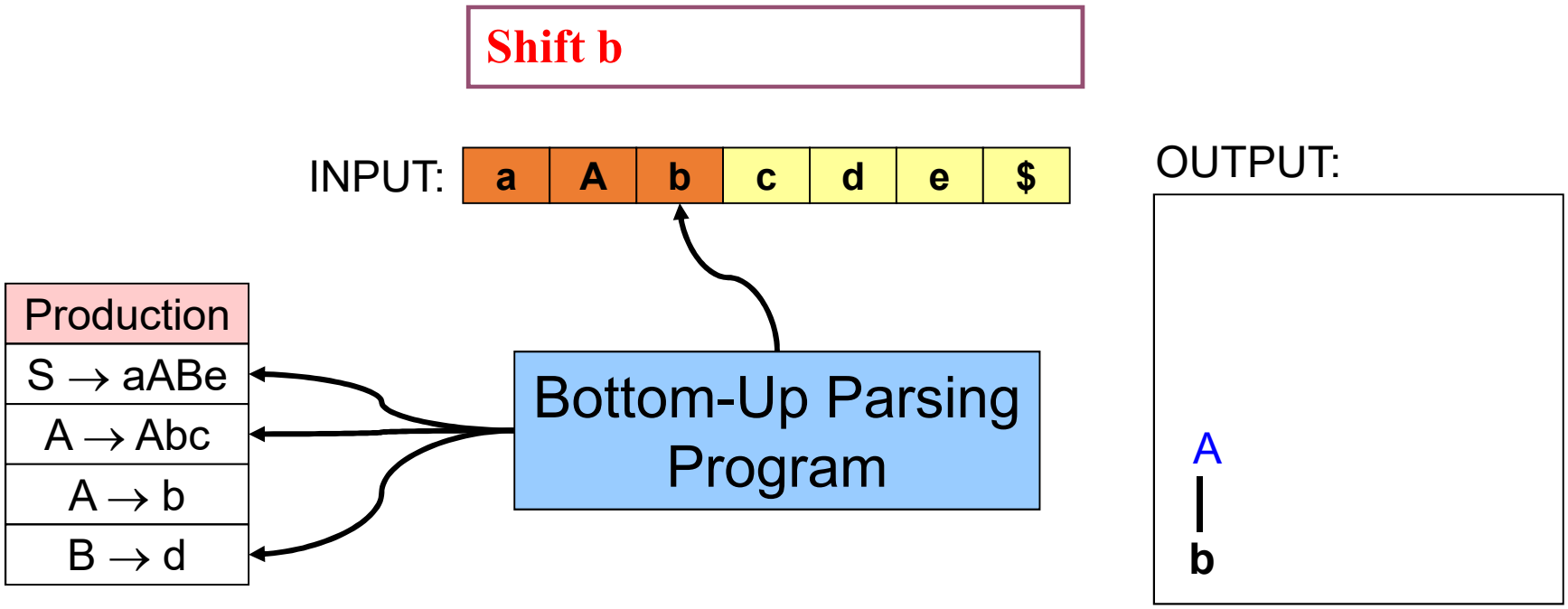
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (3)



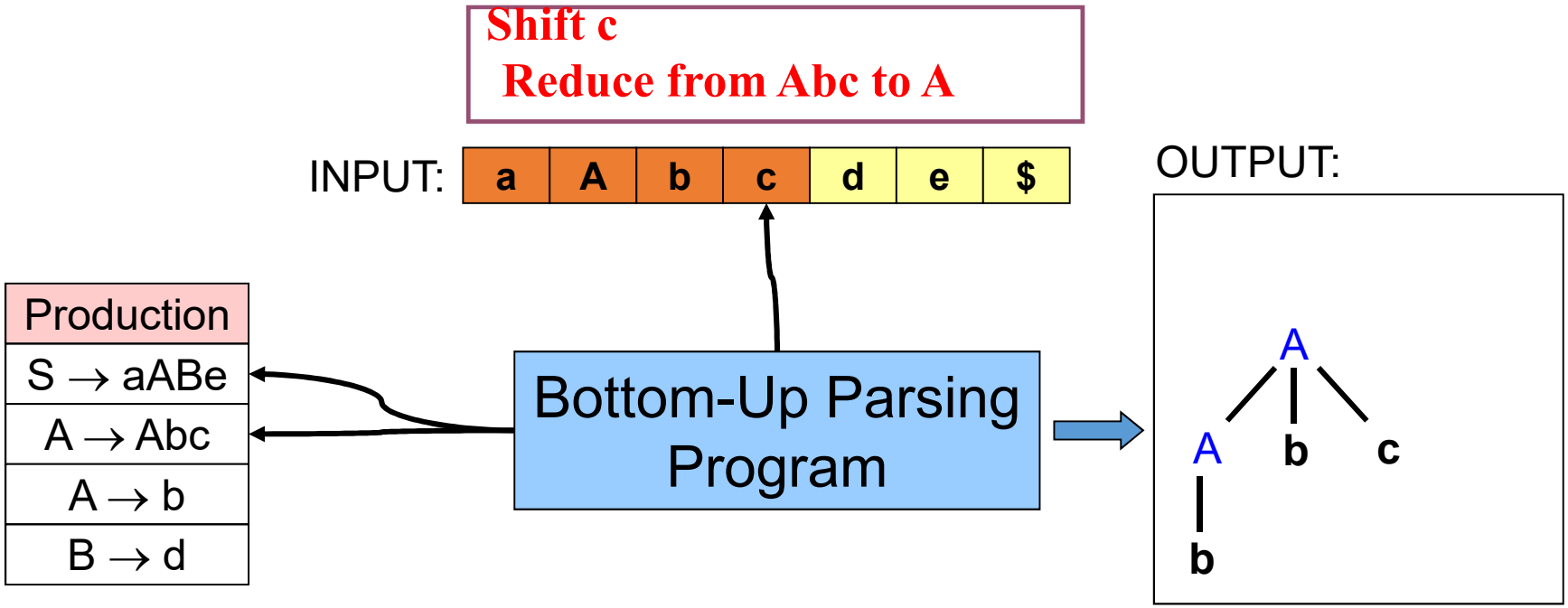
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (4)



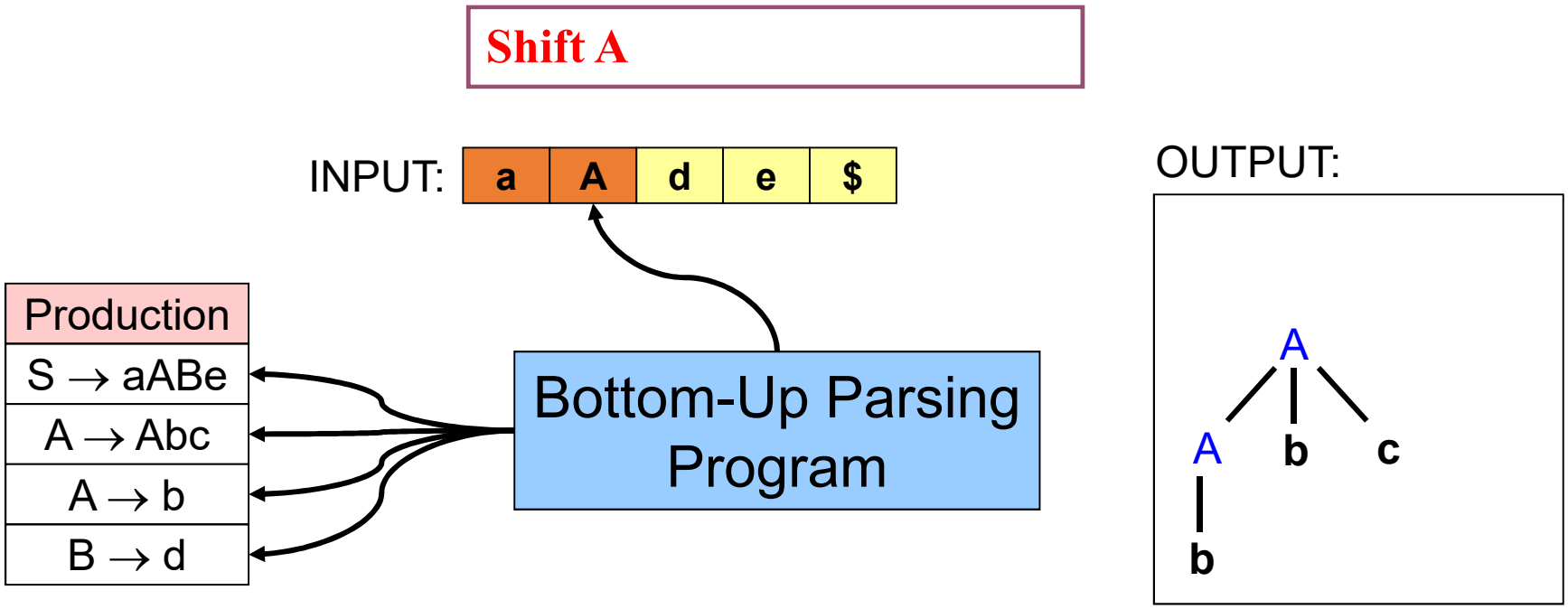
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (5)



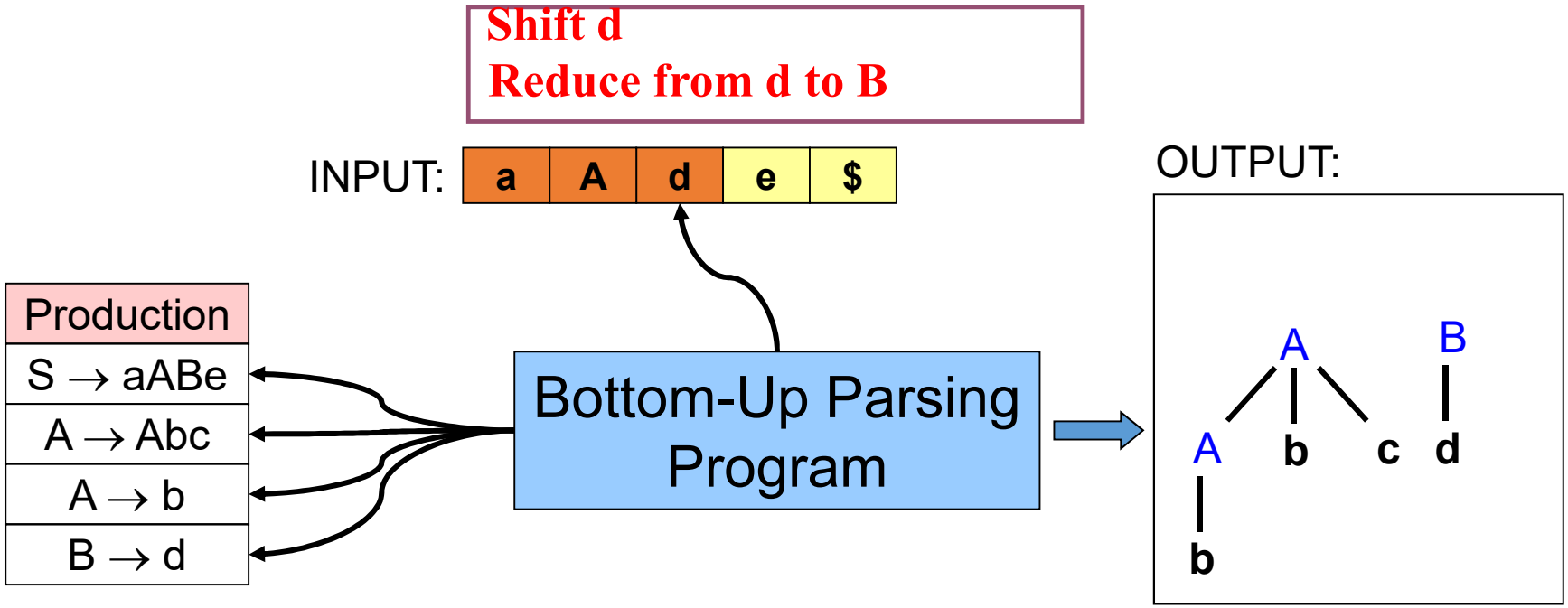
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (6)



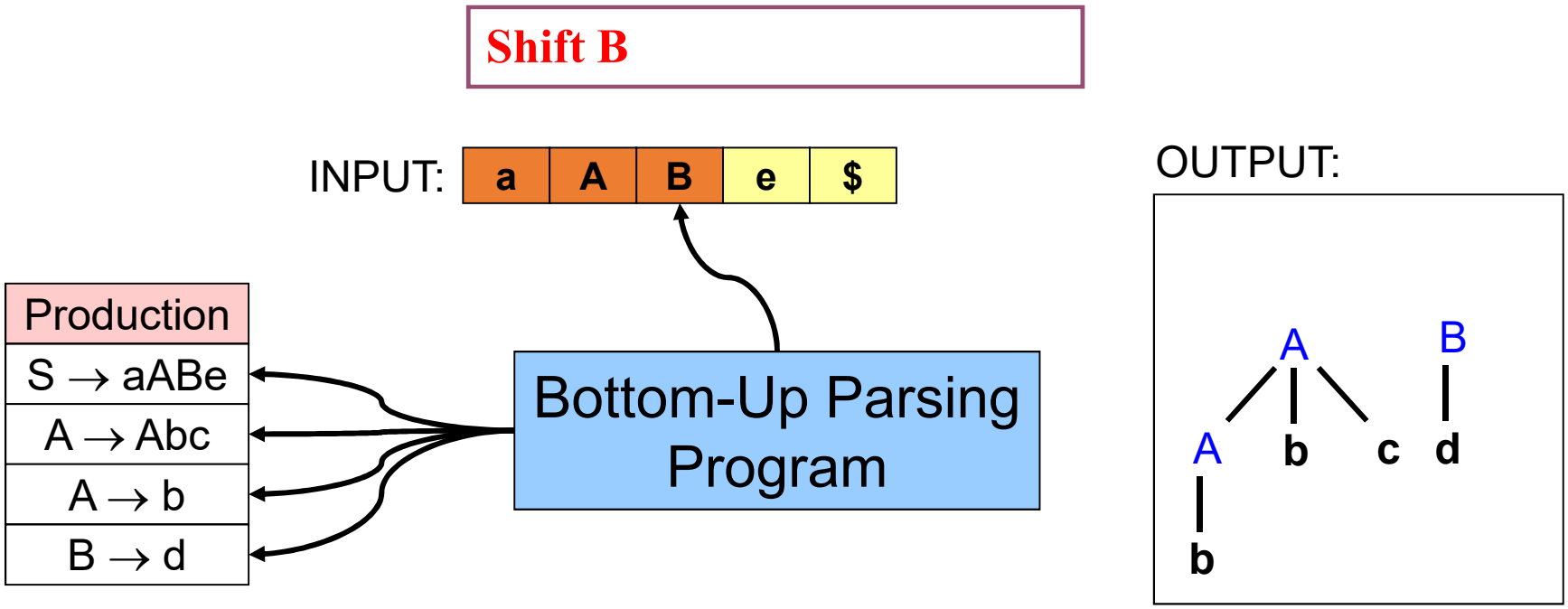
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (7)



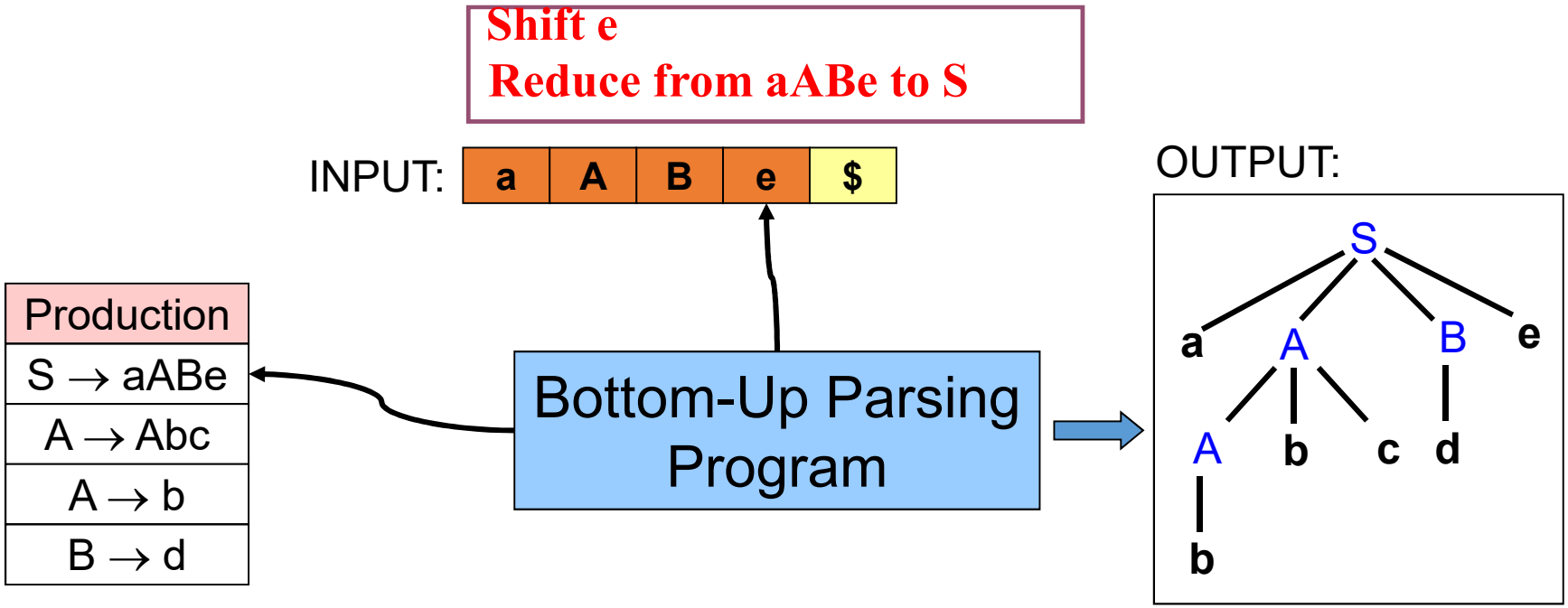
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (8)



The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (9)



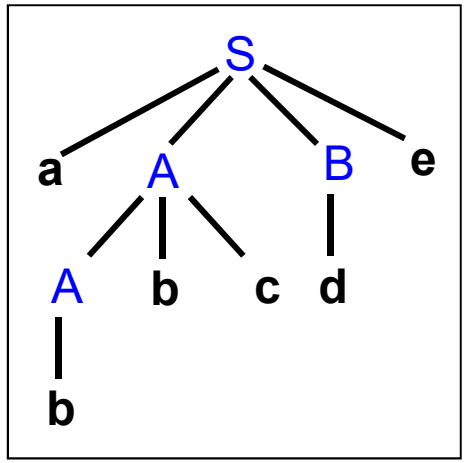
The parse stack concatenated with the remaining input represents a sentential form

Shift-reduce Parsing in Action (10)

Shift S
Hit the target $

INPUT: | S | $ |

OUTPUT:



Production
S → aABe
A → Abc
A → b
B → d

Bottom-Up Parsing Program

The parse stack concatenated with the remaining input represents a sentential form

Shift-Reduce Parser

- Critical Question: How can we know when we have found a handle without generating lots of different derivations?

  →Have we reached the end of handles and how long is the handle?
  → Which non-terminal does the handle reduce to?

  - Answer: we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
    - ACTION table
    - GOTO table
  - LR(1) parsers build a DFA that runs over the stack & finds them

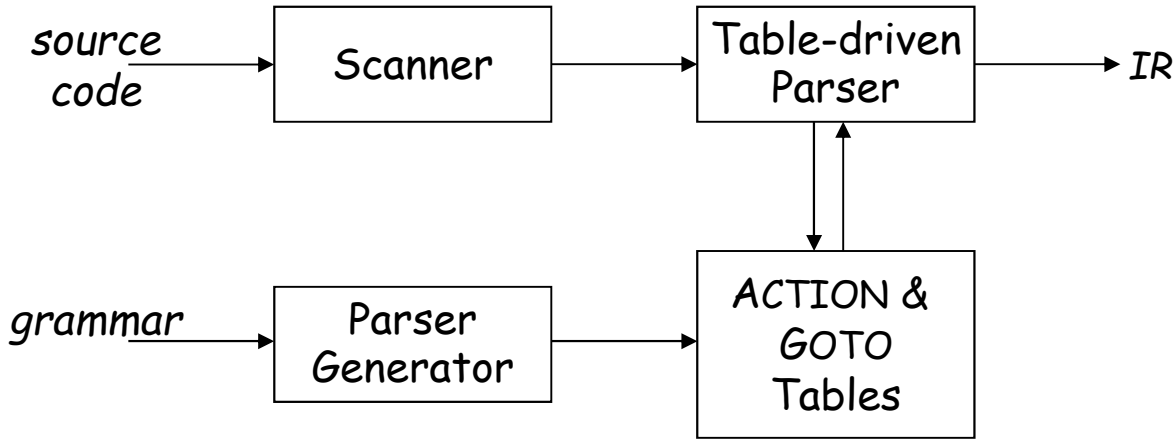
LR(1) Parser

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar
  - Informal definition or LR(1) grammar:
    - A grammar is LR(1) if, given a rightmost derivation

      $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow$ *sentence*
    - We can
      1. *isolate the handle of each right-sentential form* γ_i, and
      2. *determine the production with which to reduce, by scanning* γ_i *from left-to-right, going at most 1 symbol beyond the right end of the handle of* γ_i

LR(1) Parsers

- A table-driven LR(1) parser looks like



- Tables *can* be built by hand
  - Action table, which specifies what actions to take
    - Shift, reduce, accept or error
  - Goto table, which specifies state transition
    - To indicate transition of finite state machine
  - We push states, rather than symbols onto the stack
  - Each state represents the possible sub-trees of the parse tree

LR Table Construction

- A production has the form
  - $A \rightarrow X_1 X_2 \ldots X_j$
- By adding a dot, we get a configuration (or an item)
  - $A \rightarrow \bullet X_1 X_2 \ldots X_j$
  - $A \rightarrow X_1 X_2 \ldots X_i \bullet X_{i+1} \ldots X_j$
  - $A \rightarrow X_1 X_2 \ldots X_j \bullet$
- The • indicates how much of a RHS has been shifted into the stack.
- An item with the • at the end of the RHS
  - $A \rightarrow X_1 X_2 \ldots X_j \bullet$
  - indicates (or recognized) that RHS should be reduced to LHS
- An item with the • at the beginning of RHS
  - $A \rightarrow \bullet X_1 X_2 \ldots X_j$
  - predicts that RHS will be shifted into the stack

LR Table Construction

- A state is a set of configurations
  - This means that the actual state of LR parsers is denoted by one of the items.
- The closure0 operation:
  - if there is a configuration $B \rightarrow \delta \bullet A\ \rho$ in the set then add all configurations of the form $A \rightarrow \bullet\ \gamma$ to the set.
- The initial configuration
  - s0 = closure0({$S \rightarrow \bullet\ \alpha$ })

EX: for grammar G_1 :

1. $S' \rightarrow S\$$
2. $S \rightarrow ID \mid \varepsilon$

closure0({ $S' \rightarrow \bullet\ S\$$ }) =

{ $S' \rightarrow \bullet\ S$,

$S \rightarrow \bullet\ ID$,

$S \rightarrow \varepsilon \bullet$ }

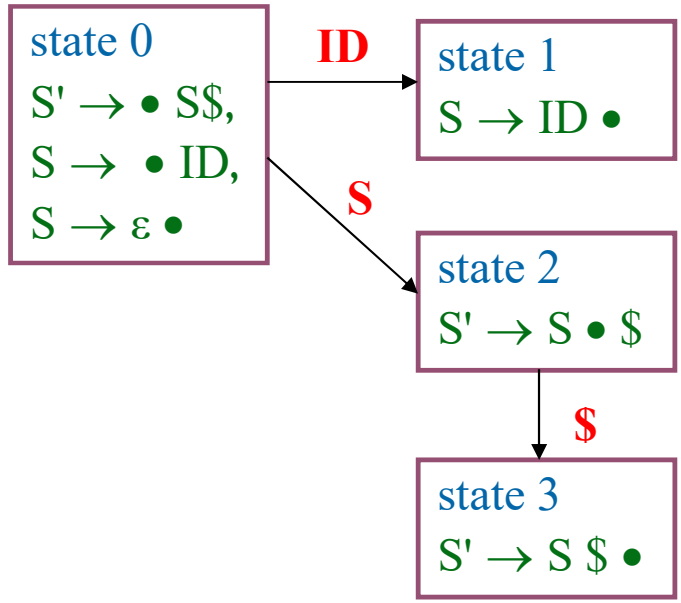
Q: Why the grammar use S'→S$?
A: Easy to check the end of parsing

LR Table Construction

- Characteristic finite state machine (CFSM)
  - It is a finite automaton and the goto table of LR parsers
  - Identifying configuration sets and successor operation with CFSM states and transitions

EX: for grammar G_1 :

1. S'→S$
2. S→ID| ε



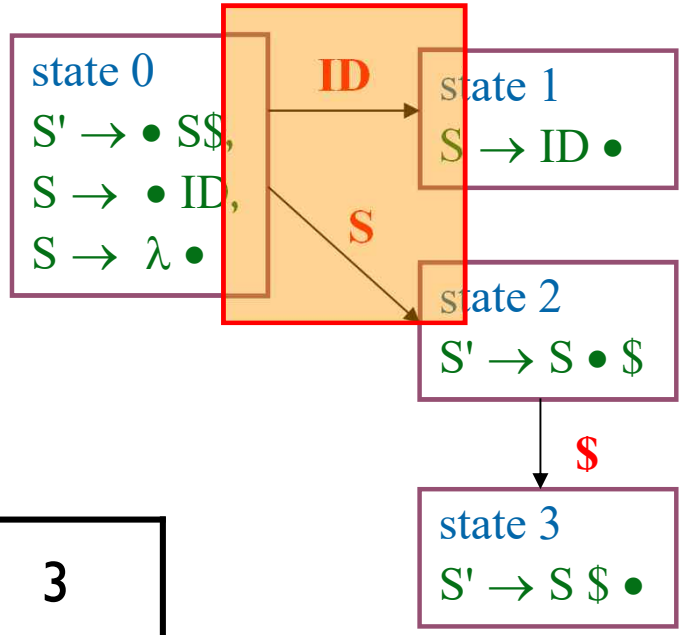
State	Symbol		
	ID	$	S
0	1	4	2
1	4	4	4
2	4	3	4
3	4	4	4
4			

goto table

LR Table Construction

EX: for grammar G_1 :

1. S'→S$
2. S→ID|λ



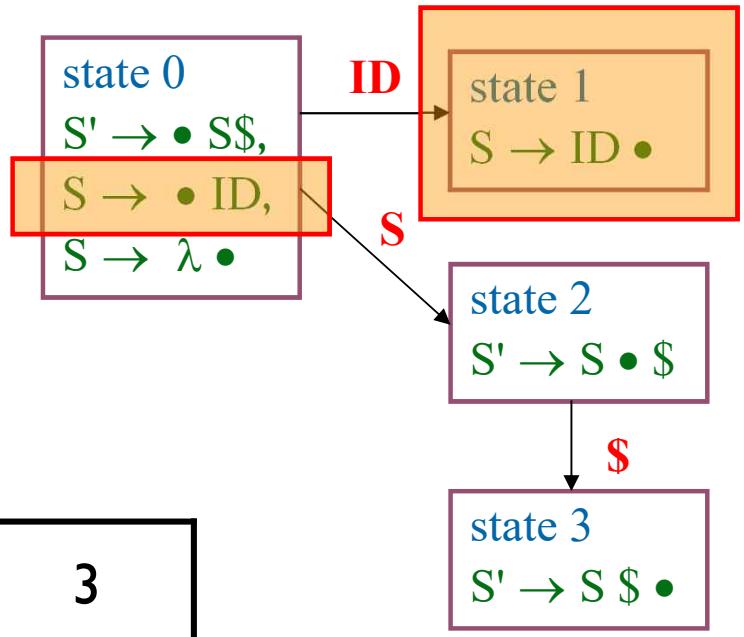
state	0	1	2	3
action	S	R2	S	Accept

action table

LR Table Construction

EX: for grammar G_1 :

1. S'→S$
2. S→ID|λ



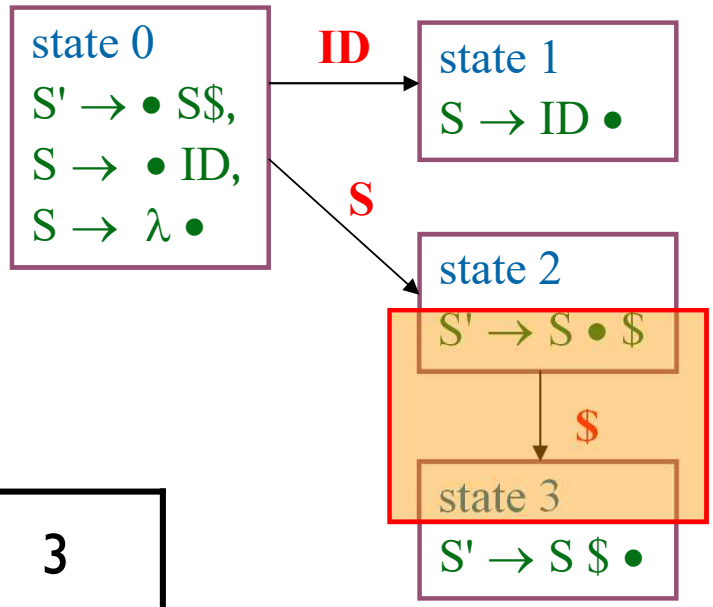
state	0	1	2	3
action	S	R2	S	Accept

action table

LR Table Construction

EX: for grammar G_1 :

1. S'→S$
2. S→ID|λ



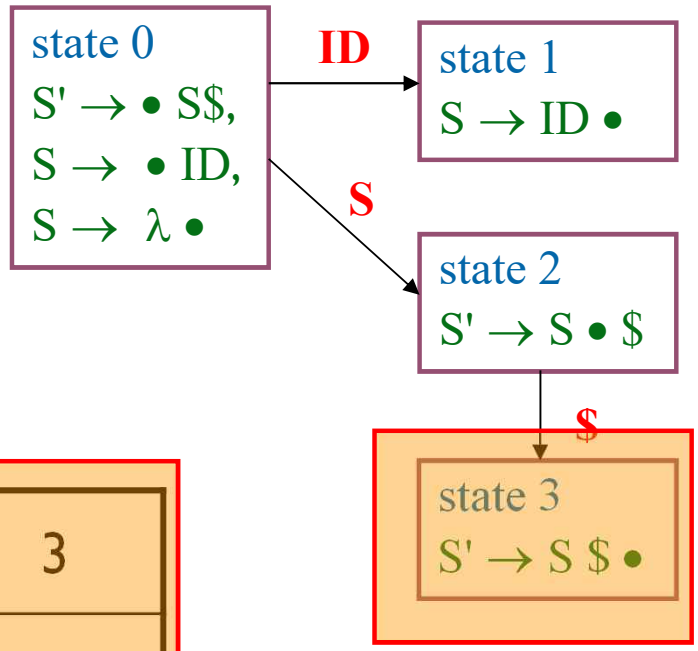
state	0	1	2	3
action	S	R2	S	Accept

action table

LR Table Construction

EX: for grammar G_1 :

1. S'→S$
2. S→ID|λ



state	0	1	2	3
action	S	R2	S	Accept

action table

LR(1) Skeleton Parser

```
stack.push(INVALID); stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
        s = stack.top();
        if ( ACTION[s,token] == "shift snext" ) then {
                stack.push(token); stack.push(snext);
                token ← scanner.next_token();
        }
        else if ( ACTION[s,token] == "reduce A→β" ) then {
                stack.popnum(2*|β|); // pop 2*|β| symbols
                s = stack.top();
                stack.push(A); stack.push(GOTO[s,A]);
        }
        else if ( ACTION[s,token] == "accept"
                                & token == EOF ) then {
                not_found = false;
        }
        else report a syntax error and recover;
}
report success;
```

The skeleton parser

- push tokens & NTs along with DFA states
- uses ACTION & GOTO tables (DFA)
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

LR(1) Parsers

- The Big Picture
  - Model the state of the parser
  - Use two functions goto(s, X) and closure(s)
    - goto() is analogous to state transition in finite automata
    - closure() adds information to form a state
  - Build up the states and transition functions of the DFA
  - Use this information to fill in the ACTION and GOTO tables

LR(0) example

- Z → E
- E → E + T | T
- T → i | (E)

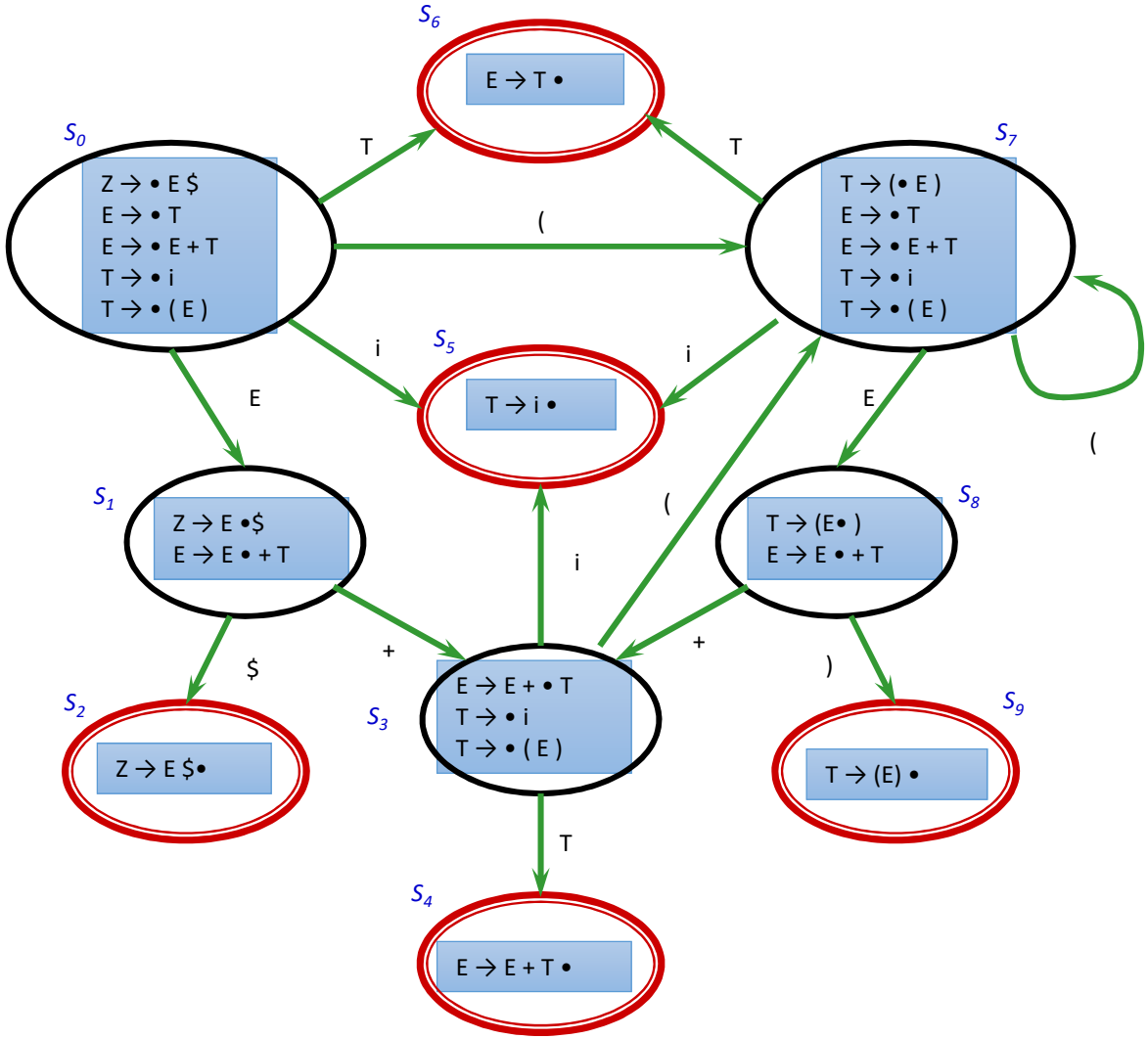


Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(1)	0	begin SimpleStmt ; SimpleStmt ; end $	Shift 1

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(2)	0,1	SimpleStmt ; SimpleStmt ; end $	Shift 5

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(3)	0,1,5	; SimpleStmt ; end $	Shift 6

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(4)	0,1,5,6	SimpleStmt ; end $	Shift 5

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(5)	0,1,5,6,5	; end $	Shift 6

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input		Action
(6)	0,1,5,6,5,6,λ	end $	/* goto(6,<stmts>) = 10 */	Reduce 4

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

<program>												
<stmts>		2			7		10			11		

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input		Action
(7)	0,1,5,6,*5,6,10*	end $	/* goto(6,<stmts>) = 10 */	Reduce 2

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

<program>												
<stmts>		2			7		10			11		

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input		Action
(8)	0,1,*5,6,10*	end $	/* goto(1,<stmts>) = 2 */	Reduce 2

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

goto table

<program>												
<stmts>		2			7		10			11		

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(9)	0,1,2	end $	Shift 3

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Table-driven Parsing

- grammar G_0
  1. <program>→begin<stmts>end$
  2. <stmts> →SimpleStmt;<stmts>
  3. <stmts> →begin<stmts>end;<str
  4. <stmts> →λ

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

Figure 6.3 A Shift-Reduce `go_to` Table for G_0

- tracing steps

Step	Parse Stack	Remaining Input	Action
(10)	0,1,2,3	$	Accept

action table

Symbol	State											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								

Summary

- Bottom-up parser
  - Reverse rightmost derivation
  - Handle pruning, reduction
- Shift-reduce parser
  - Reduce if found a handle in stack
  - Otherwise, shift a token (push on to stack)
- LR(1) parser
  - Discover handles from DFA
  - ACTION, GOTO tables from DFA