

Compiler Construction

Lectures 13 – 16

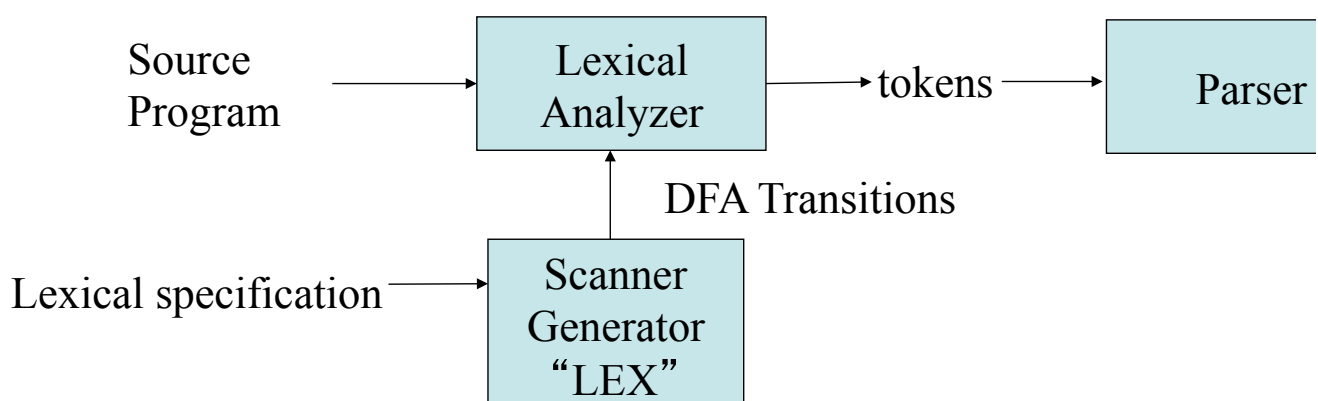
1. **Return to lexical analysis : application of Theory of Regular Languages and Finite Automata**
2. **Generating Recursive descent parsers**
3. **Beyond Recursive Descent Parsing I**
4. **Beyond Recursive Descent Parsing II**

Lent Term, 2013

Lecturer: Timothy G. Griffin

1

Generating Lexical Analyzers



The idea : use regular expressions as the basis of a lexical specification. The core of the lexical analyzer is then a deterministic finite automata (DFA)

2

Recall from *Regular Languages and Finite Automata* (Part 1A)

Regular expressions over an alphabet Σ

- each symbol $a \in \Sigma$ is a regular expression
- ε is a regular expression
- $\emptyset$ is a regular expression
- if r and s are regular expressions, then so is $(r|s)$
- if r and s are regular expressions, then so is rs
- if r is a regular expression, then so is $(r)^*$

Every regular expression is built up inductively, by *finitely many* applications of the above rules.

(N.B. we assume ε , $\emptyset$, $($, $)$, $|$, and $*$ are *not* symbols in Σ .)

Slide 5

Traditional Regular Language Problem

Given a regular expression,

e

and an input string w , determine if $w \in L(e)$

One method: Construct a DFA M from e and test if it accepts w .

Something closer to the “lexing problem”

Given an ordered list of regular expressions,

$$e_1 \quad e_2 \quad \dots \quad e_k$$

and an input string w , find a list of pairs

$$(i_1, w_1), (i_2, w_2), \dots (i_n, w_n)$$

such that

- 1) $w = w_1 w_2 \dots w_n$
- 2) $w_j \in L(e_{i_j})$
- 3) $w_j \in L(e_s) \rightarrow i_j \leq s$ (priority rule)
- 4) $\forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \dots w_n) : u \neq \varepsilon$
 $\rightarrow \forall s : w_j u \notin L(e_s)$ (longest match)

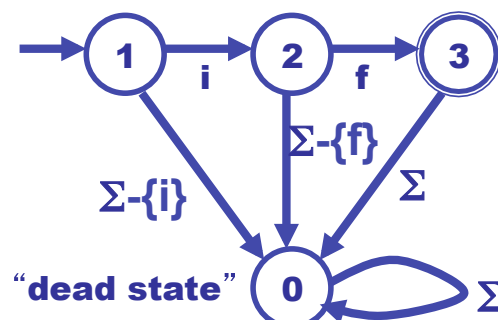
5

Define Tokens with Regular Expressions (Finite Automata)

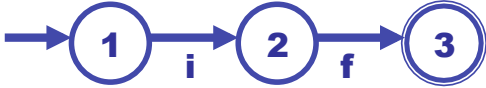
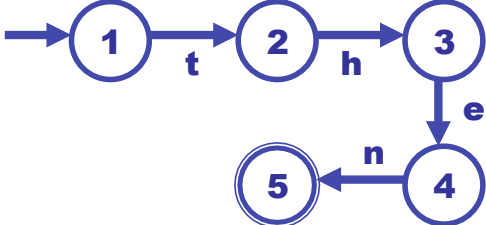
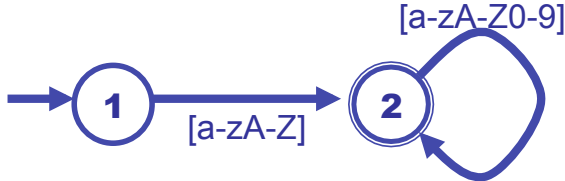
Keyword: if



This FA is really shorthand for:

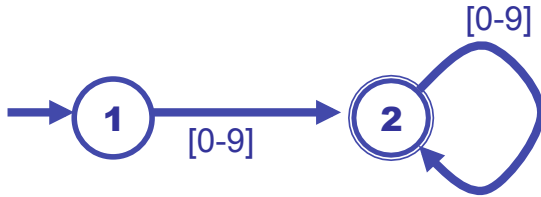
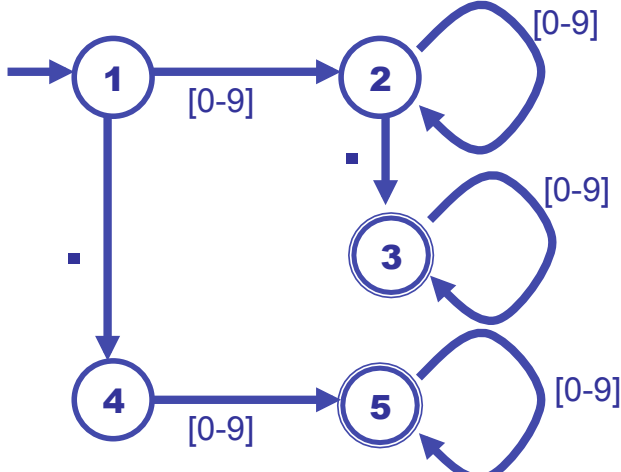


Define Tokens with Regular Expressions (Finite Automata)

Regular Expression	Finite Automata	Token
Keyword: if		KEY(IF)
Keyword: then		KEY(then)
Identifier: [a-zA-Z][a-zA-Z0-9]*		ID(s)

7

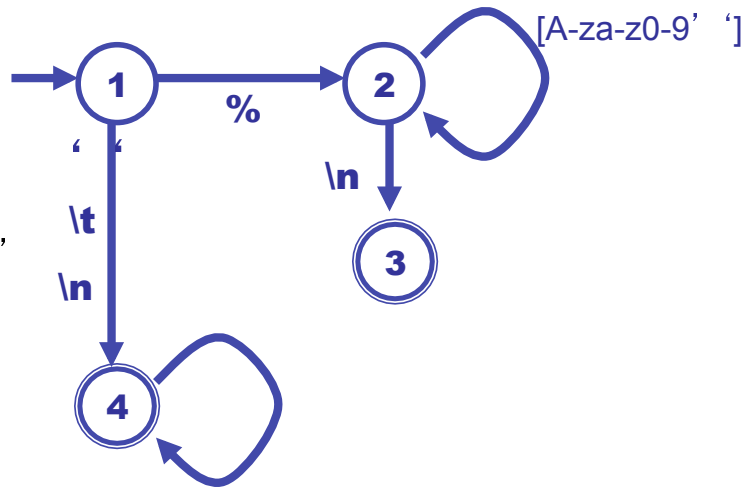
Define Tokens with Regular Expressions (Finite Automata)

Regular Expression	Finite Automata	Token
number: [0-9][0-9]*		NUM(n)
real: ([0-9]+ '.' [0-9]*) ([0-9]* '.' [0-9]+)		NUM(n)

8

No Tokens for “White-Space”

White-space:
 (' ' | '\n' | '\t')+
 | '%' [A-Za-z0-9' ']+ '\n'



9

Constructing a Lexer

INPUT:
 an **ordered**
 list of regular
 expressions

e_1
 e_2
 $\vdots$
 e_k

Construct all
 corresponding
 finite automata

NFA_1
 NFA_2
 $\vdots$
 NFA_k

Construct a single
 non-deterministic
 finite automata

NFA

Construct a single
 deterministic
 finite automata

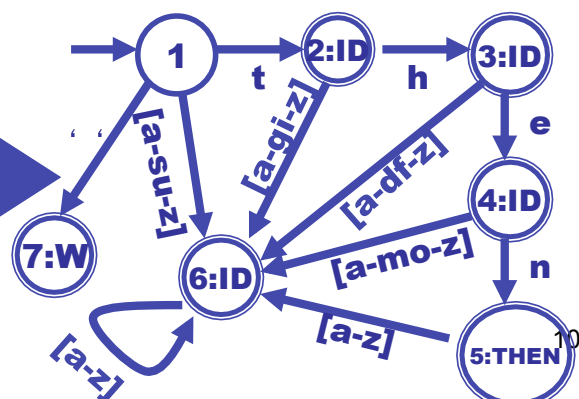
DFA

use priority

(1) Keyword : then

(2) Ident : $[a-z][a-z]^*$

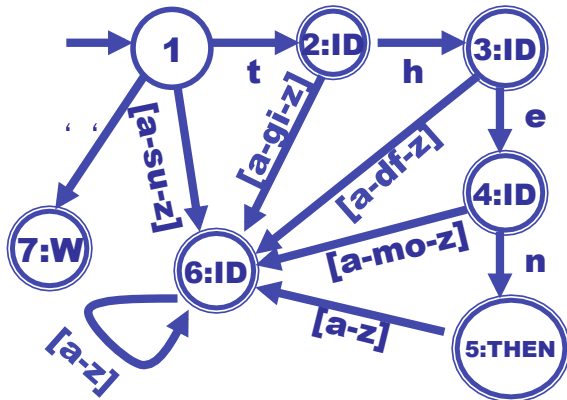
(2) White-space: ' '



What about longest match?

Start in initial state,
Repeat:

- (1) read input until dead state is reached. Emit token associated with last accepting state.
- (2) reset state to start state



| = current position, \$ = EOF

Input	current state	last accepting state
then thenx\$	1	0
t hen thenx\$	2	2
th en thenx\$	3	3
the n thenx\$	4	4
then thenx\$	5	5
then thenx\$	0	5
then thenx\$	1	0
then thenx\$	7	7
then t henx\$	0	7
then thenx\$	1	0
then t henx\$	2	2
then th enx\$	3	3
then the nx\$	4	4
then then x\$	5	5
then thenx \$	6	6
then thenx\$	0	6

11

Predictive (Recursive Descent) Parsing Can we automate this?

(G5)

S ::= if E then S else S
 | begin S L
 | print E

E ::= NUM = NUM

L ::= end
 | ; S L

```

int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else error();}

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

void L() {switch(tok) {
    case END:   eat(END); break;
    case SEMI:  eat(SEMI); S(); L(); break;
    default: error();
}}

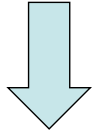
void E() {eat(NUM) ; eat(EQ); eat(NUM); }
  
```

Parse corresponds to a left-most derivation
constructed in a "top-down" manner

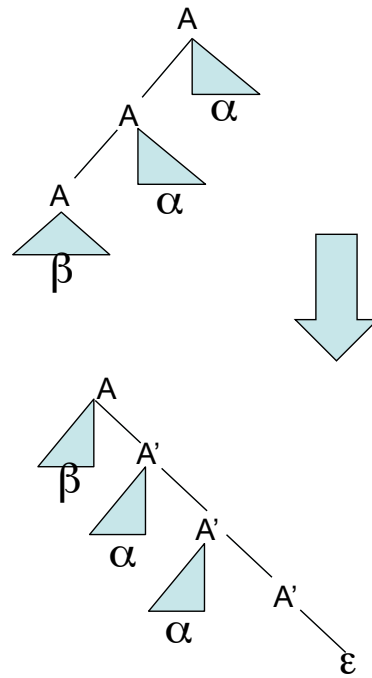
12

Eliminate Left-Recursion

Immediate left-recursion

$$A ::= A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_k \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$


$$A ::= \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' ::= \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_k A' \mid \epsilon$$


For eliminating left-recursion in general, see Aho and Ullman.¹³

Eliminating Left Recursion

(G2)
 $S ::= E\$$

$$E ::= E + T \mid E - T \mid T$$

$$T ::= T * F \mid T / F \mid F$$

$$F ::= \text{NUM} \mid \text{ID} \mid (E)$$

Note that

$$E ::= T \text{ and }$$

$$E ::= E + T$$

will cause problems

since $\text{FIRST}(T)$ will be included in $\text{FIRST}(E + T)$ ---- so how can we decide which production

To use based on next token?

Solution: eliminate "left recursion"!

$$E ::= T E'$$

$$E' ::= + T E' \mid - T E' \mid \epsilon$$

(G6)

$$S ::= E\$$$

$$E ::= T E'$$

$$E' ::= + T E' \mid - T E' \mid \epsilon$$

$$T ::= F T'$$

$$T' ::= * F T' \mid / F T' \mid \epsilon$$

$$F ::= \text{NUM} \mid \text{ID} \mid (E)$$

Eliminate left recursion

FIRST and FOLLOW

For each non-terminal X we need to compute

$FIRST[X]$ = the set of terminal symbols that
can begin strings derived from X

$FOLLOW[X]$ = the set of terminal symbols that
can immediately follow X in some
derivation

$nullable[X]$ = true if X can derive the empty string,
false otherwise

$nullable[Z] = \text{false}$, for Z in T

$nullable[Y_1 Y_2 \dots Y_k] = nullable[Y_1] \text{ and } \dots nullable[Y_k]$, for $Y(i)$ in $N \cup T$.

$FIRST[Z] = \{Z\}$, for Z in T

$FIRST[X Y_1 Y_2 \dots Y_k] = FIRST[X]$ if not $nullable[X]$

$FIRST[X Y_1 Y_2 \dots Y_k] = FIRST[X] \cup FIRST[Y_1 \dots Y_k]$ otherwise

15

Computing First, Follow, and nullable

For each terminal symbol Z

$FIRST[Z] := \{Z\}$;

$nullable[Z] := \text{false}$;

For each non-terminal symbol X

$FIRST[X] := FOLLOW[X] := \{\}$;

$nullable[X] := \text{false}$;

repeat

for each production $X \rightarrow Y_1 Y_2 \dots Y_k$

if $Y_1, \dots, Y_k$ are all nullable, or $k = 0$

then $nullable[X] := \text{true}$

for each i from 1 to k , each j from $i + 1$ to k

if $Y_1 \dots Y_{i-1}$ are all nullable or $i = 1$

then $FIRST[X] := FIRST[X] \cup FIRST[Y(i)]$

if $Y_{i+1} \dots Y_k$ are all nullable or if $i = k$

then $FOLLOW[Y(i)] := FOLLOW[Y(i)] \cup FOLLOW[X]$

if $Y_{i+1} \dots Y_{j-1}$ are all nullable or $i+1 = j$

then $FOLLOW[Y(i)] := FOLLOW[Y(i)] \cup FIRST[Y(j)]$

until there is no change

16

First, Follow, nullable table for G6

	Nullable	FIRST	FOLLOW
S	False	{ (, ID, NUM }	{ }
E	False	{ (, ID, NUM }	{), \$ }
E'	True	{ +, - }	{), \$ }
T	False	{ (, ID, NUM }	{), +, -, \$ }
T'	True	{ *, / }	{), +, -, \$ }
F	False	{ (, ID, NUM }	{), *, /, +, -, \$ }

(G6)
 $S ::= E\$$
 $E ::= T E'$
 $E' ::= + T E'$
 $\quad \quad | - T E'$
 $\quad \quad |$
 $T ::= F T'$
 $T' ::= * F T'$
 $\quad \quad | / F T'$
 $\quad \quad |$
 $F ::= \text{NUM}$
 $\quad \quad | \text{ID}$
 $\quad \quad | (E)$

17

Predictive Parsing Table for G6

Table[X, T] = Set of productions

$X ::= Y_1 \dots Y_k$ in Table[X, T]
 if T in FIRST[$Y_1 \dots Y_k$]
 or if (T in FOLLOW[X] and nullable[$Y_1 \dots Y_k$])

NOTE: this could lead to more than one entry! If so, out of luck --- can't do recursive descent parsing!

	+	*	(	)	ID	NUM	\$
S			$S ::= E\$$		$S ::= E\$$	$S ::= E\$$	
E			$E ::= T E'$		$E ::= T E'$	$E ::= T E'$	
E'	$E' ::= + T E'$				$E' ::=$		$E' ::=$
T			$T ::= F T'$		$T ::= F T'$	$T ::= F T'$	
T'	$T' ::=$	$T' ::= * F T'$			$T' ::=$		$T' ::=$
F			$F ::= (E)$		$F ::= \text{ID}$	$F ::= \text{NUM}$	

(entries for /, - are similar...)

18

Left-most derivation is constructed by recursive descent

Left-most derivation

(G6)

```

S ::= E$

E ::= TE'

E' ::= +TE'
      | -TE'
      |

T ::= FT'

T' ::= *FT'
      | /FT'
      |

F ::= NUM
      | ID
      | (E)
  
```

```

S → E$
  → TE' $
  → FT' E' $
  → (E) T' E' $
  → (TE' ) T' E' $
  → (FT' E' ) T' E' $
  → (17 T' E' ) T' E' $
  → (17 E' ) T' E' $
  → (17 + TE' ) T' E' $
  → (17 + FT' E' ) T' E' $
  → (17 + 4 T' E' ) T' E' $
  → (17 + 4 E' ) T' E' $
  → (17 + 4 ) T' E' $
  → (17 + 4 ) * FT' E' $
  → ...
  → ...
  → (17 + 4 ) * (2 - 10) T' E' $
  → (17 + 4 ) * (2 - 10) E' $
  → (17 + 4 ) * (2 - 10)
  
```

```

call S()
  on '(' call E()
    on '(' call T()
  ...
  ...
  ...
  
```

19

As a stack machine

```

S → E$
  → TE' $
  → FT' E' $
  → (E) T' E' $
  → (TE' ) T' E' $
  → (FT' E' ) T' E' $
  → (17 T' E' ) T' E' $
  → (17 E' ) T' E' $
  → (17 + TE' ) T' E' $
  → (17 + FT' E' ) T' E' $
  → (17 + 4 T' E' ) T' E' $
  → (17 + 4 E' ) T' E' $
  → (17 + 4 ) T' E' $
  → (17 + 4 ) * FT' E' $
  → ...
  → ...
  → (17 + 4 ) * (2 - 10) T' E' $
  → (17 + 4 ) * (2 - 10) E' $
  → (17 + 4 ) * (2 - 10)
  
```

```

      E$
      TE' $
      FT' E' $
  (    E) T' E' $
  (    TE' ) T' E' $
  (    FT' E' ) T' E' $
  (17   T' E' ) T' E' $
  (17   E' ) T' E' $
  (17 + TE' ) T' E' $
  (17 + FT' E' ) T' E' $
  (17 + 4 T' E' ) T' E' $
  (17 + 4 E' ) T' E' $
  (17 + 4 ) T' E' $
  (17 + 4 ) * FT' E' $
  ...
  ...
  (17 + 4 ) * (2 - 10) T' E' $
  (17 + 4 ) * (2 - 10) E' $
  (17 + 4 ) * (2 - 10)
  
```

20

But wait! What if there are conflicts in the predictive parsing table?

(G7)		Nullable	FIRST	FOLLOW
S ::= d X Y S	S	false	{ c,d ,a }	{ }
Y ::= c 	Y	true	{ c }	{ c,d,a }
X ::= Y a	X	true	{ c,a }	{ c, a,d }

The resulting “predictive” table is not so predictive....

	a	c	d
S	{ S ::= X Y S }	{ S ::= X Y S }	{ S ::= X Y S, S ::= d }
Y	{ Y ::= }	{ Y ::= , Y ::= c }	{ Y ::= }
X	{ X ::= a, X ::= Y }	{ X ::= Y }	{ X ::= Y }

21

LL(1), LL(k), LR(0), LR(1), ...

- LL(k) : (L)eft-to-right parse, (L)eft-most derivation, k-symbol lookahead. Based on looking at the next k tokens, an LL(k) parser must *predict* the next production. We have been looking at LL(1).
- LR(k) : (L)eft-to-right parse, (R)ight-most derivation, k-symbol lookahead. Postpone production selection until *the entire* right-hand-side has been seen (and as many as k symbols beyond).
- LALR(1) : A special subclass of LR(1).

Example

(G8)

$S ::= S ; S \mid ID = E \mid \text{print } (L)$

$E ::= ID \mid NUM \mid E + E \mid (S, E)$

$L ::= E \mid L, E$

To be consistent, I should write the following, but I won't...

(G8)

$S ::= S \text{ SEMI } S \mid ID \text{ EQUAL } E \mid \text{PRINT LPAREN } L \text{ RPAREN}$

$E ::= ID \mid NUM \mid E \text{ PLUS } E \mid \text{LPAREN } S \text{ COMMA } E \text{ RPAREN}$

$L ::= E \mid L \text{ COMMA } E$

23

A right-most derivation ...

(G8)

$S ::= S ; S$
 $\mid ID = E$
 $\mid \text{print } (L)$

$E ::= ID$
 $\mid NUM$
 $\mid E + E$
 $\mid (S, E)$

$L ::= E$
 $\mid L, E$

S
 $\rightarrow S ; \underline{S}$
 $\rightarrow S ; ID = \underline{E}$
 $\rightarrow S ; ID = E + \underline{E}$
 $\rightarrow S ; ID = E + (S, \underline{E})$
 $\rightarrow S ; ID = E + (S, \underline{ID})$
 $\rightarrow S ; ID = E + (\underline{S}, d)$
 $\rightarrow S ; ID = E + (ID = \underline{E}, d)$
 $\rightarrow S ; ID = E + (ID = E + \underline{E}, d)$
 $\rightarrow S ; ID = E + (ID = E + \underline{NUM}, d)$
 $\rightarrow S ; ID = E + (ID = \underline{E} + 6, d)$
 $\rightarrow S ; ID = E + (ID = \underline{NUM} + 6, d)$
 $\rightarrow S ; ID = E + (\underline{ID} = 5 + 6, d)$
 $\rightarrow S ; ID = \underline{E} + (d = 5 + 6, d)$
 $\rightarrow S ; ID = \underline{ID} + (d = 5 + 6, d)$
 $\rightarrow S ; \underline{ID} = c + (d = 5 + 6, d)$
 $\rightarrow \underline{S} ; b = c + (d = 5 + 6, d)$
 $\rightarrow ID = \underline{E} ; b = c + (d = 5 + 6, d)$
 $\rightarrow ID = \underline{NUM} ; b = c + (d = 5 + 6, d)$
 $\rightarrow \underline{ID} = 7 ; b = c + (d = 5 + 6, d)$
 $\rightarrow a = 7 ; b = c + (d = 5 + 6, d)$

24

Now, turn it upside down ...

→ **a = 7 ; b = c + (d = 5 + 6, d)**
 → **ID = 7 ; b = c + (d = 5 + 6, d)**
 → **ID = NUM; b = c + (d = 5 + 6, d)**
 → **ID = E ; b = c + (d = 5 + 6, d)**
 → **S ; b = c + (d = 5 + 6, d)**
 → **S ; ID = c + (d = 5 + 6, d)**
 → **S ; ID = ID + (d = 5 + 6, d)**
 → **S ; ID = E + (d = 5 + 6, d)**
 → **S ; ID = E + (ID = 5 + 6, d)**
 → **S ; ID = E + (ID = NUM + 6, d)**
 → **S ; ID = E + (ID = E + 6, d)**
 → **S ; ID = E + (ID = E + NUM, d)**
 → **S ; ID = E + (ID = E + E, d)**
 → **S ; ID = E + (ID = E, d)**
 → **S ; ID = E + (S, d)**
 → **S ; ID = E + (S, ID)**
 → **S ; ID = E + (S, E)**
 → **S ; ID = E + E**
 → **S ; ID = E**
 → **S ; S**
S

25

Now, slice it down the middle...

	a = 7 ; b = c + (d = 5 + 6, d)	
ID	= 7 ; b = c + (d = 5 + 6, d)	
ID = NUM	; b = c + (d = 5 + 6, d)	
ID = E	; b = c + (d = 5 + 6, d)	
S	; b = c + (d = 5 + 6, d)	
S ; ID	= c + (d = 5 + 6, d)	
S ; ID = ID	+ (d = 5 + 6, d)	
S ; ID = E	+ (d = 5 + 6, d)	
S ; ID = E + (ID	= 5 + 6, d)	
S ; ID = E + (ID = NUM	+ 6, d)	
S ; ID = E + (ID = E	+ 6, d)	
S ; ID = E + (ID = E + NUM	, d)	
S ; ID = E + (ID = E + E	, d)	
S ; ID = E + (ID = E	, d)	
S ; ID = E + (S	, d)	
S ; ID = E + (S, ID	)	
S ; ID = E + (S, E)		
S ; ID = E + E		
S ; ID = E		
S ; S		
S		

A stack of terminals and non-terminals

The rest of the input string

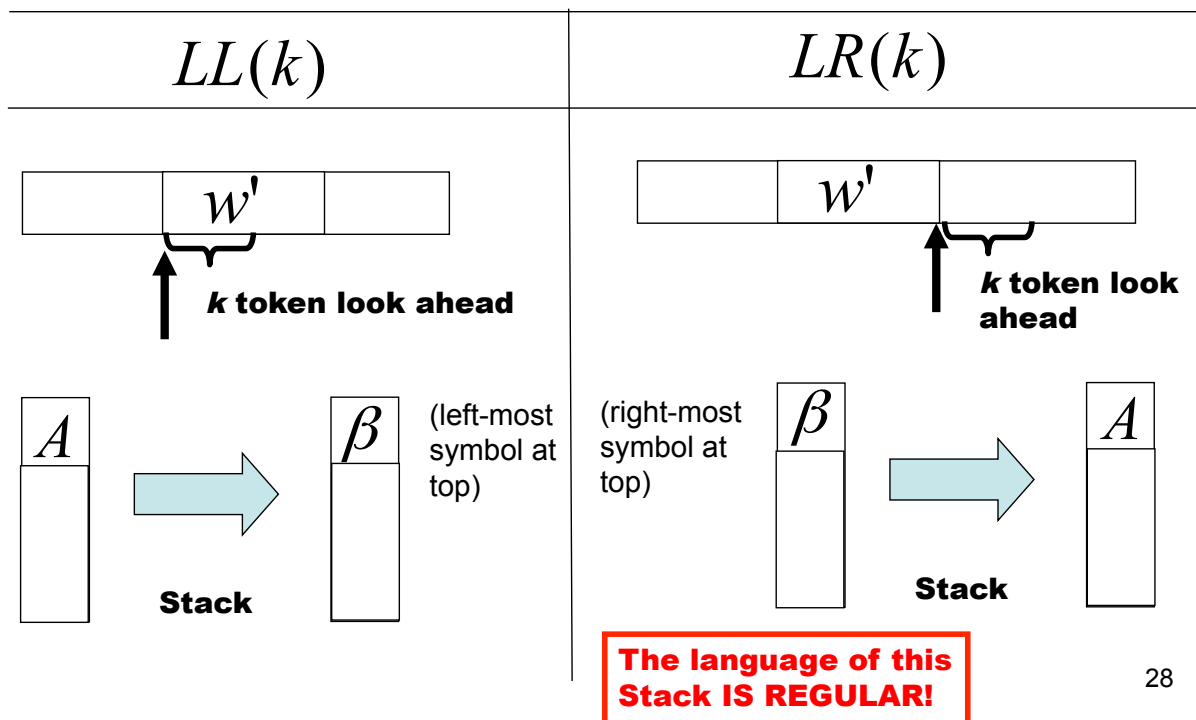
26

Now, add some actions. s = SHIFT, r = REDUCE

<pre> ID ID = NUM ID = E S S ; ID S ; ID = ID S ; ID = E S ; ID = E + (ID S ; ID = E + (ID = NUM S ; ID = E + (ID = E S ; ID = E + (ID = E + NUM S ; ID = E + (ID = E + E S ; ID = E + (ID = E S ; ID = E + (S S ; ID = E + (S, ID S ; ID = E + (S, E) S ; ID = E + E S ; ID = E S ; S S </pre>	<pre> a = 7 ; b = c + (d = 5 + 6, d) = 7 ; b = c + (d = 5 + 6, d) ; b = c + (d = 5 + 6, d) ; b = c + (d = 5 + 6, d) ; b = c + (d = 5 + 6, d) = c + (d = 5 + 6, d) + (d = 5 + 6, d) + (d = 5 + 6, d) = 5 + 6, d) + 6, d) + 6, d) , d) , d) , d)))) s s, s r E ::= NUM r S ::= ID = E s, s s, s r E ::= ID s, s, s s, s r E ::= NUM s, s r E ::= NUM r E ::= E+E, s, s r S ::= ID = E R E ::= ID s, r E ::= (S, E) r E ::= E + E r S ::= ID = E r S ::= S ; S </pre>
SHIFT = LEX + move token to stack	
ACTIONS	

LL(k) vs. LR(k) reductions

$$A \rightarrow \beta \Rightarrow^* w' \quad (\beta \in (T \cup N)^*, w' \in T^*)$$



Q: How do we know when to shift and when to reduce? A: Build a FSA from LR(0) Items!

(G10)

$S ::= A \$$

$A ::= (A)$
 $| ()$

If

$X ::= \alpha\beta$

is a production, then

$X ::= \alpha ' \beta$

is an LR(0) item.

$S ::= ' A \$$

$S ::= A ' \$$

$A ::= ' (A)$

$A ::= (' A)$

$A ::= (A ')$

$A ::= (A) '$

$A ::= ' ()$

$A ::= (')$

$A ::= () '$

LR(0) items indicate what is on the stack (to the left of the ') and what is still in the input stream (to the right of the ')

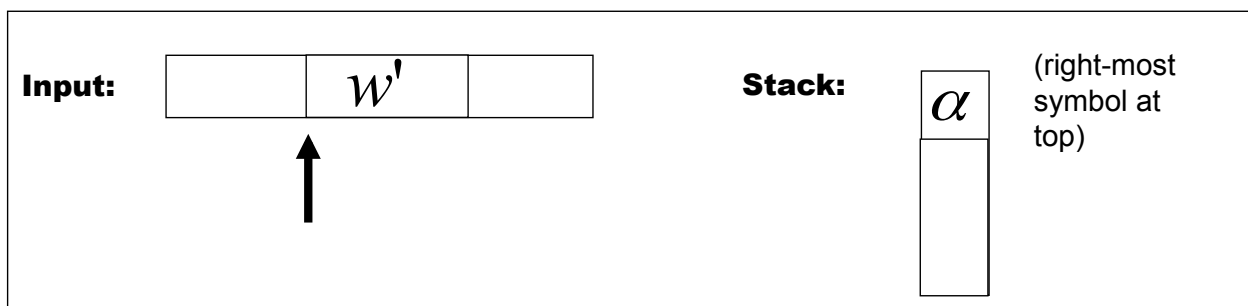
29

LR(k) states (non-deterministic)

The state

$(A \rightarrow \alpha \bullet \beta, a_1 a_2 \cdots a_k)$

should represent this situation:



with $\beta a_1 a_2 \cdots a_k \xRightarrow{*} w'$

30

Key idea behind LR(0) items

- If the “current state” contains the item $A ::= \alpha \,, c \beta$ and the current symbol in the input buffer is c
 - the state prompts parser to perform a shift action
 - next state will contain $A ::= \alpha c \,, \beta$
- If the “state” contains the item $A ::= \alpha \,,$
 - the state prompts parser to perform a reduce action
- If the “state” contains the item $S ::= \alpha \,, \$$ and the input buffer is empty
 - the state prompts parser to accept
- But How about $A ::= \alpha \,, X \beta$ where X is a nonterminal?

31

The NFA for LR(0) items

- The transition of LR(0) items can be represented by an NFA, in which
 - 1. each LR(0) item is a state,
 - 2. there is a transition from item $A ::= \alpha \,, c \beta$ to item $A ::= \alpha c \,, \beta$ with label c , where c is a terminal symbol
 - 3. there is an ϵ -transition from item $A ::= \alpha \,, X \beta$ to $X ::= \,, \gamma$, where X is a non-terminal
 - 4. $S ::= \,, A \$$ is the start state
 - 5. $A ::= \alpha \,,$ is a final state.

32

Example NFA for Items

S ::= ' A \$

A ::= (' A)

A ::= ' ()

S ::= A ' \$

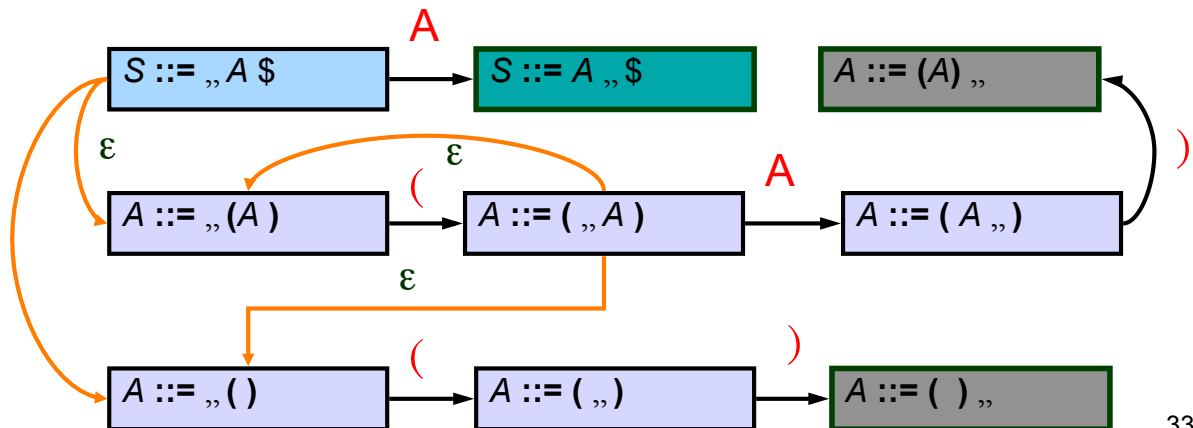
A ::= (A ')

A ::= (')

A ::= ' (A)

A ::= (A) '

A ::= () '



33

The DFA from LR(0) items

- After the NFA for LR(0) is constructed, the resulting DFA for LR(0) parsing can be obtained by the usual NFA2DFA construction.
- we thus require
 - ϵ -closure (I)
 - move(S, a)

Fixed Point Algorithm for Closure(I)

- Every item in **I** is also an item in **Closure(I)**
- If $A ::= \alpha _, B \beta$ is in **Closure(I)** and $B ::= _, \gamma$ is an item, then add $B ::= _, \gamma$ to **Closure(I)**
- Repeat until no more new items can be added to **Closure(I)**

Examples of Closure

Closure($\{A ::= (\text{ „ } A)\}$) =

$$\left\{ \begin{array}{l} A ::= (\text{ „ } A) \\ A ::= \text{ „ } (A) \\ A ::= \text{ „ } () \end{array} \right\}$$

• closure($\{S ::= \text{ „ } A \$\}$)

$$\left\{ \begin{array}{l} S ::= \text{ „ } A \$ \\ A ::= \text{ „ } (A) \\ A ::= \text{ „ } () \end{array} \right\}$$

S ::= ' A \$
S ::= A ' \$
A ::= ' (A)
A ::= (' A)
A ::= (A ')
A ::= (A) '
A ::= ' ()
A ::= (')
A ::= () '

35

Goto() of a set of items

- Goto finds the new state after consuming a grammar symbol while in the current state
- Algorithm for $Goto(I, X)$
 where I is a set of items
 and X is a non-terminal

$$Goto(I, X) = \text{Closure}(\{ A ::= \alpha X \text{ „ } \beta \mid A ::= \alpha \text{ „ } X \beta \text{ in } I \})$$

- goto is the new set obtained by
 “moving the dot” over X

36

Examples of Goto

- Goto ($\{A ::= \text{„}(A)\}$, $()$)

$$\left\{ \begin{array}{l} A ::= (\text{„} A) \\ A ::= \text{„} (A) \\ A ::= \text{„} () \end{array} \right\}$$

- Goto ($\{A ::= (\text{„} A)\}$, A)

$$\left\{ A ::= (A \text{„}) \right\}$$

$S ::= ' A \$$
 $S ::= A ' \$$
 $A ::= ' (A)$
 $A ::= (' A)$
 $A ::= (A ')$
 $A ::= (A) '$
 $A ::= ' ()$
 $A ::= (')$
 $A ::= () '$

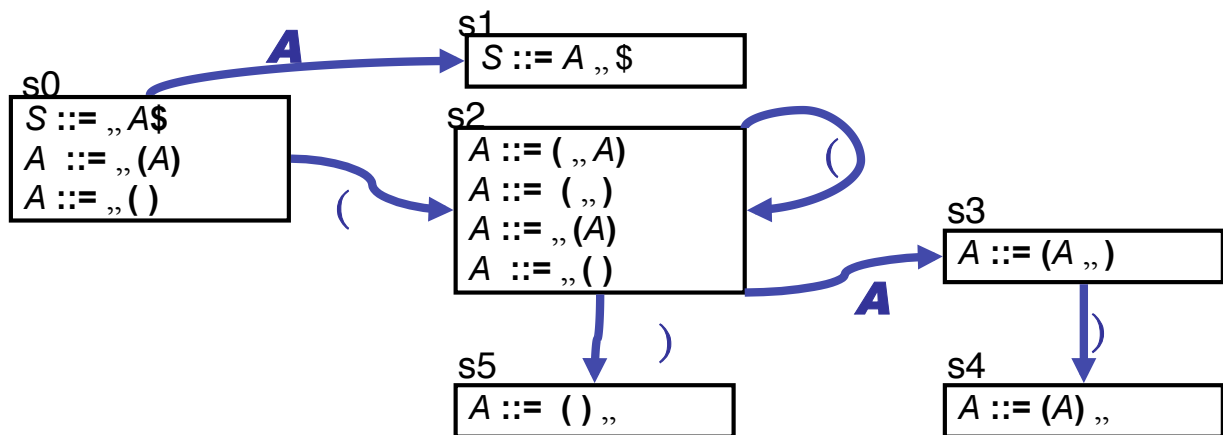
37

Building the DFA states

- Essentially the usual NFA2DFA construction!!
- Let A be the start symbol and S a new start symbol.
- Create a new rule $S ::= A \$$
- Create the first state to be $\text{Closure}(\{ S ::= \text{„} A \$\})$
- Pick a state I
 - for each item $A ::= \alpha \text{„} X \beta$ in I
 - find $\text{Goto}(I, X)$
 - if $\text{Goto}(I, X)$ is not already a state, make one
 - Add an edge X from state I to $\text{Goto}(I, X)$ state
- Repeat until no more additions possible

38

DFA Example



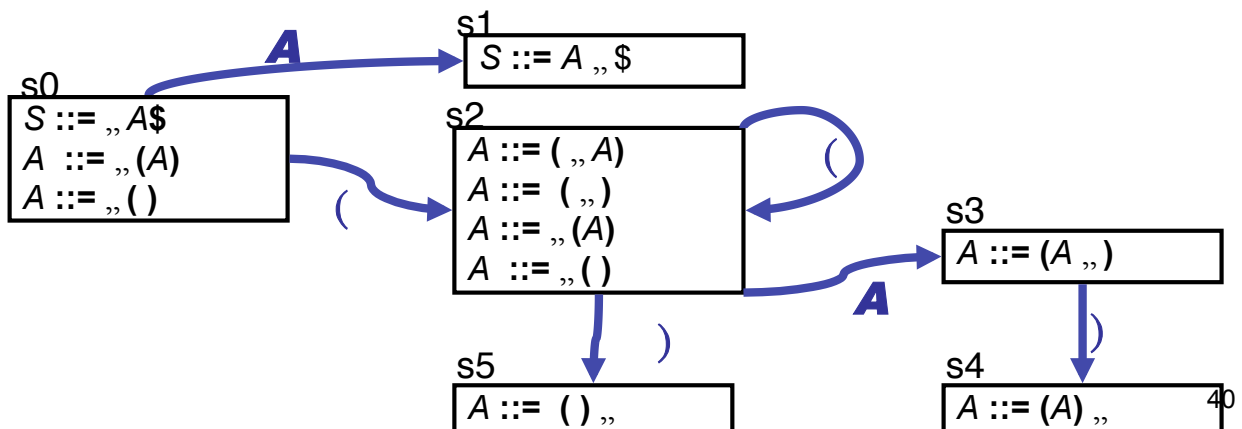
39

Creating the Parse Table(s)

(G10)

- (1) $S ::= A \$$**
- (2) $A ::= (A)$**
- (3) $A ::= ()$**

State	(	)	\$	A
s0	shift to s2			goto s1
s1			accept	
s2	shift to s2	shift to s5		goto s3
s3		shift to s4		
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	



40

Parsing with an LR Table

Use table and top-of-stack and input symbol to get action:

If action is

shift s_n : advance input one token,
push s_n on stack

reduce $X ::= \alpha$: pop stack $2 * |\alpha|$ times (grammar symbols
are paired with states). In the state
now on top of stack,
use goto table to get next
state s_n ,
push it on top of stack

accept : stop and accept

error : weep (actually, produce a good error
message)

41

Parsing, again...

(G10)

(1) $S ::= A\$$

(2) $A ::= (A)$

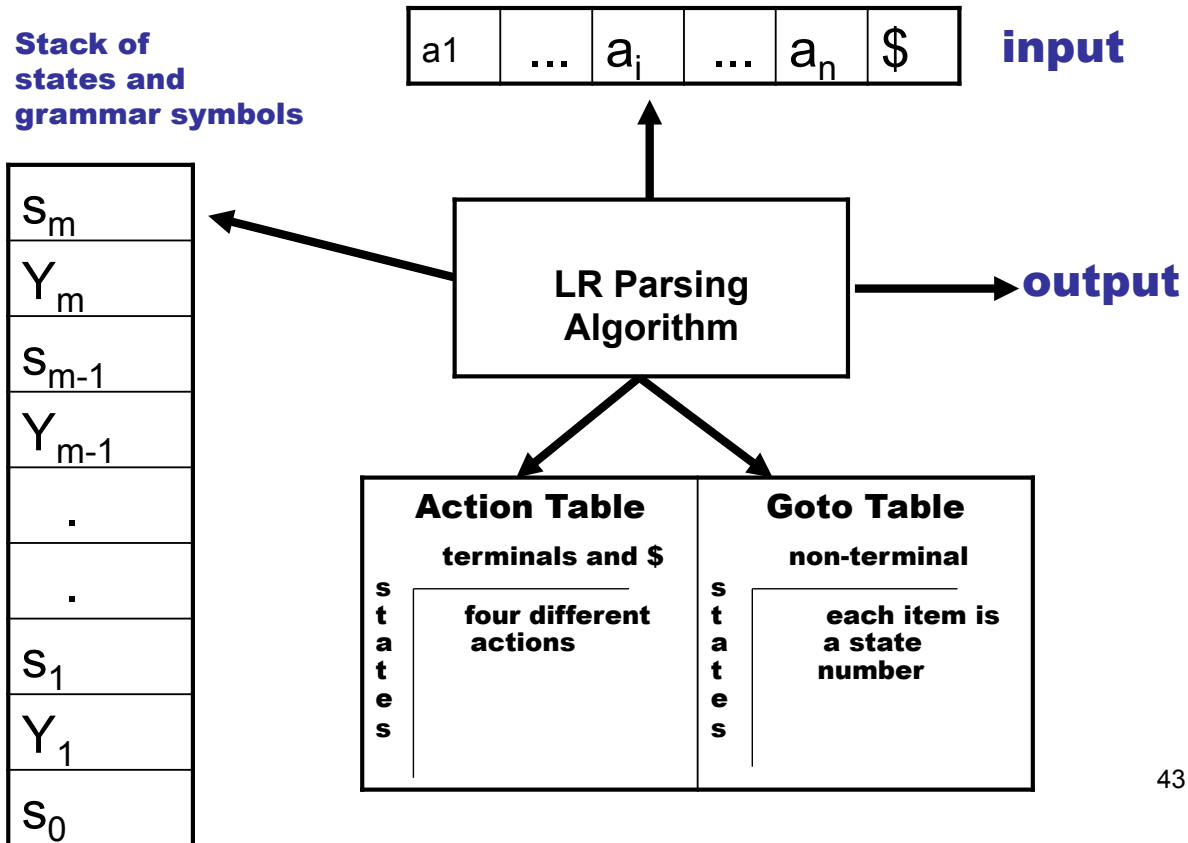
(3) $A ::= ()$

State	ACTION				Goto
	(	)	\$	A	
s0	shift to s2				goto s1
s1			accept		
s2	shift to s2	shift to s5			goto s3
s3		shift to s4			
s4	reduce (2)	reduce (2)	reduce (2)		
s5	reduce (3)	reduce (3)	reduce (3)		

s0	(())\$	shift s2
s0 (s2	())\$	shift s2
s0 (s2 (s2	))\$	shift s5
s0 (s2 (s2) s5	)\$	reduce $A ::= ()$
s0 (s2 A	)\$	goto s3
s0 (s2 A s3	)\$	shift s4
s0 (s2 A s3) s4	\$	reduce $A ::= (A)$
s0 A	\$	goto s1
s0 A s1	\$	ACCEPT!

42

LR Parsing Algorithm



43

Problem With LR(0) Parsing

- ‘ **No lookahead**
- ‘ **Vulnerable to unnecessary conflicts**
 - **Shift/Reduce Conflicts** (may reduce too soon in some cases)
 - **Reduce/Reduce Conflicts**
- ‘ **Solutions:**
 - **LR(1) parsing - systematic lookahead**

44

LR(1) Items

- An LR(1) item is a pair:
 $(X ::= \alpha . \beta, a)$
 - $X ::= \alpha\beta$ is a production
 - a is a terminal (the lookahead terminal)
 - LR(1) means 1 lookahead terminal
- $[X ::= \alpha . \beta, a]$ describes a context of the parser
 - We are trying to find an X followed by an a , and
 - We have (at least) α already on top of the stack
 - Thus we need to see next a prefix derived from βa

45

The Closure Operation

- Need to modify closure operation:.

Closure(Items) =

repeat

for each $[X ::= \alpha . Y\beta, a]$ in Items

for each production $Y ::= \gamma$

for each b in $\text{First}(\beta a)$

add $[Y ::= .\gamma, b]$ to Items

until Items is unchanged

46

Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items
- The start state contains $(S' ::= .S\$, \text{dummy})$
- A state that contains $[X ::= \alpha., b]$ is labeled with “reduce with $X ::= \alpha$ on lookahead b ”
- And now the transitions ...

47

The DFA Transitions

- A state s that contains $[X ::= \alpha.Y\beta, b]$ has a transition labeled y to the state obtained from $\text{Transition}(s, Y)$
 - Y can be a terminal or a non-terminal

$\text{Transition}(s, Y)$

Items = { }

for each $[X ::= \alpha.Y\beta, b]$ in s

add $[X ! \alpha Y.\beta, b]$ to Items

return $\text{Closure}(\text{Items})$

48

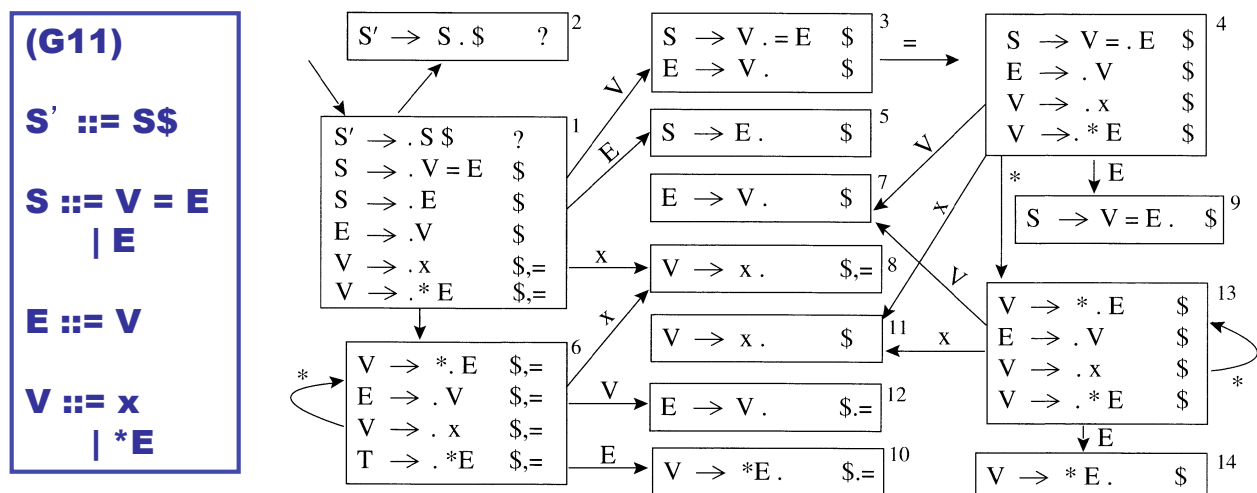
LR(1)-the parse table

- Shift and goto as before
- Reduce
 - state I with item $(A \rightarrow \alpha., z)$ gives a reduce $A \rightarrow \alpha$ if z is the next character in the input.

- LR(1)-parse tables are very big

49

LR(1)-DFA



LR(1)-parse table

	x	*	=	\$	S	E	V		x	*	=	\$	S	E	V
1	s8	s6			g2	g5	g3	8			r4	r4			
2				acc				9				r1			
3			s4	r3				10			r5	r5			
4	s11	s13				g9	g7	11				r4			
5				r2				12			r3	r3			
6	s8	s6				g10	g12	13	s11	s13				g14	g7
7				r3				14				r5			

51

LALR States

- Consider for example the LR(1) states

$$\{[X ::= \alpha. , a], [Y ::= \beta. , c]\}$$

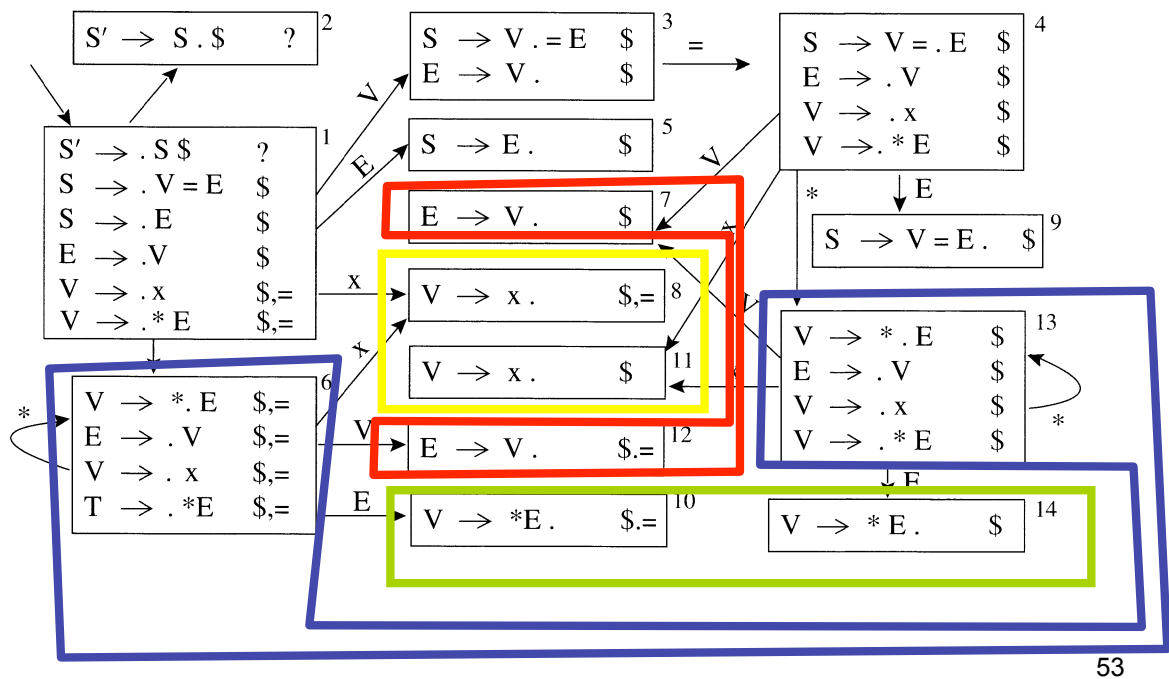
$$\{[X ::= \alpha. , b], [Y ::= \beta. , d]\}$$
- They have the same core and can be merged to the state

$$\{[X ::= \alpha. , a/b], [Y ::= \beta. , c/d]\}$$
- These are called LALR(1) states
 - Stands for LookAhead LR
 - Typically 10 times fewer LALR(1) states than LR(1)

52

For LALR(1), Collapse States ...

Combine states 6 and 13, 7 and 12, 8 and 11, 10 and 14.



53

LALR(1)-parse-table

	x	*	=	\$	S	E	V
1	s8	s6			g2	g5	g3
2				acc			
3			s4	r3			
4	s8	s6				g9	g7
5							
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

54

LALR vs. LR Parsing

- LALR languages are not “natural”
 - They are an efficiency hack on LR languages
- You may see claims that any reasonable programming language has a LALR(1) grammar, {Arguably this is done by defining languages without an LALR(1) grammar as unreasonable 😊 }.
- In any case, LALR(1) has become a standard for programming languages and for parser generators, in spite of its apparent complexity.

55

Lexer and Parser Generators

<http://catalog.compilertools.net/lexparse.html>

ACCENT	HAPPY
AFLEX AYACC	HOLUB
ALE	<u>LEX</u>
ANAGRAM	LLGEN
BISON	PCYACC
BISON/IEFFEL	PRECC
BTYACC	PROGRAMMAR
BYACC	RDP
COGENCEE	VISUALPARSE++
COCO	<u>YACC</u>
DEPOT4	YACC++
FLEX	...

56