Compiler Construction
Lent Term 2015
Lectures 13 --- 16 (of 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**

Timothy G. Griffin

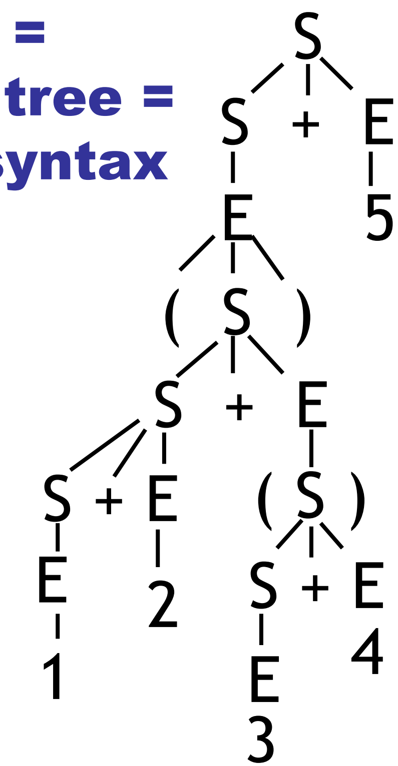
tgg22@cam.ac.uk

Computer Laboratory

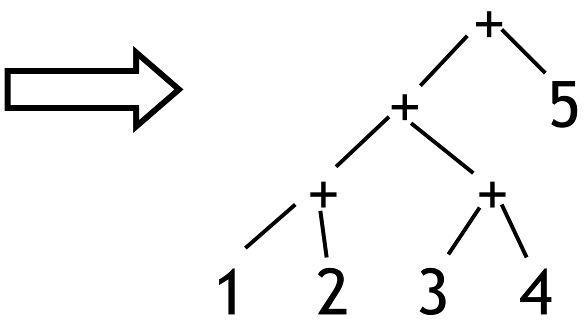
University of Cambridge

Concrete vs. Abstract Syntax Trees

parse tree = derivation tree = concrete syntax tree



Abstract Syntax Tree (AST)



An AST contains only the information needed to generate an intermediate representation

Normally a compiler constructs the concrete syntax tree only implicitly (in the parsing process) and explicitly constructs an AST.

On to Context Free Grammars (CFGs)

E ::= ID

E ::= NUM

E ::= E * E

E ::= E / E

E ::= E + E

E ::= E – E

E ::= (E)

E is a *non-terminal symbol*

ID and NUM are *lexical classes*

*, (,), +, and – are *terminal symbols*.

E ::= E + E is called a *production rule*.

Usually will write this way

E ::= ID | NUM | E * E | E / E | E + E | E – E | (E)

CFG Derivations

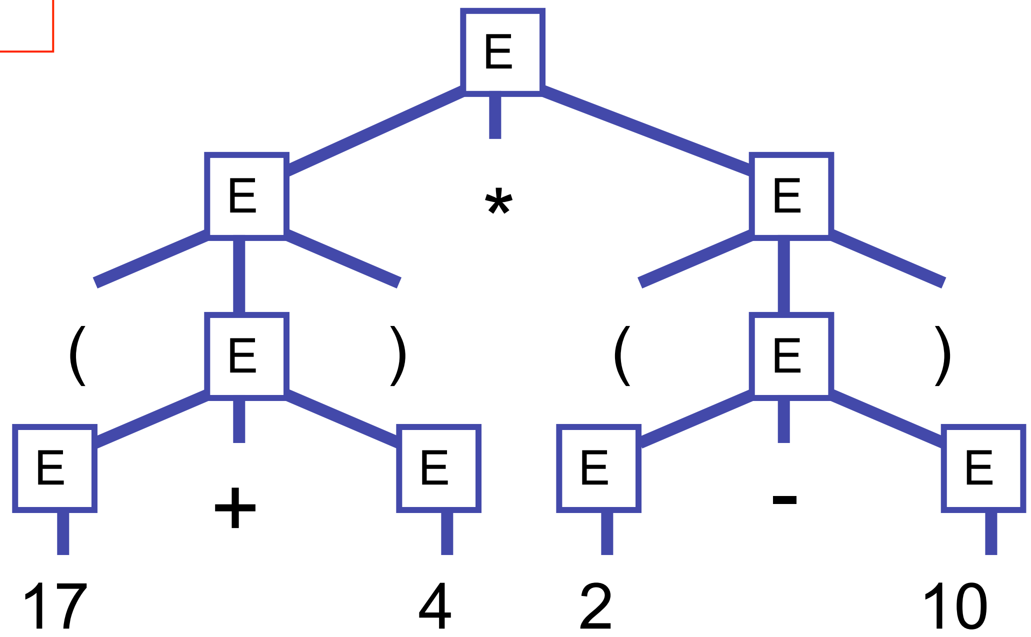
(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)

Rightmost derivation

E → E * E
→ E * (E)
→ E * (E – E)
→ E * (E – 10)
→ E * (2 – 10)
→ (E) * (2 – 10)
→ (E + E) * (2 – 10)
→ (E + 4) * (2 – E)
→ (17 + 4) * (2 – 10)

Leftmost derivation

E → E * E
→ (**E**) * E
→ (**E** + E) * E
→ (17 + E) * E
→ (17 + 4) * E
→ (17 + 4) * (E)
→ (17 + 4) * (**E** – E)
→ (17 + 4) * (2 – **E**)
→ (17 + 4) * (2 – 10)



The Derivation Tree for
(17 + 4) * (2 – 10)

More formally, ...

- **A CFG is a quadruple G = (N, T, R, S) where**
  - **N is the set of *non-terminal symbols***
  - **T is the set of *terminal symbols* (N and T disjoint)**
  - **$S \in N$ is the *start symbol***
  - **$R \subseteq N \times (N \cup T)^*$ is a set of rules**
- **Example: The grammar of nested parentheses G = (N, T, R, S) where**
  - **N = {S}**
  - **T ={ (,) }**
  - **R ={ (S, (S)) , (S, SS), (S,) }**

We will normally write R as

S ::= (S) | SS |

Derivations, more formally…

- Start from start symbol (*S*)
- Productions are used to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ comprised of both terminal and non-terminal symbols,
  and a production $A \rightarrow \beta$,
  a single step of derivation is
  $\alpha A\gamma \Rightarrow \alpha\beta\gamma$
  - *i.e.*, substitute β for an occurrence of A
- $\alpha \Rightarrow^* \beta$ means that b can be derived from a in 0 or more single steps
- $\alpha \Rightarrow^+ \beta$ means that b can be derived from a in 1 or more single steps

L(G) = The Language Generated by Grammar G

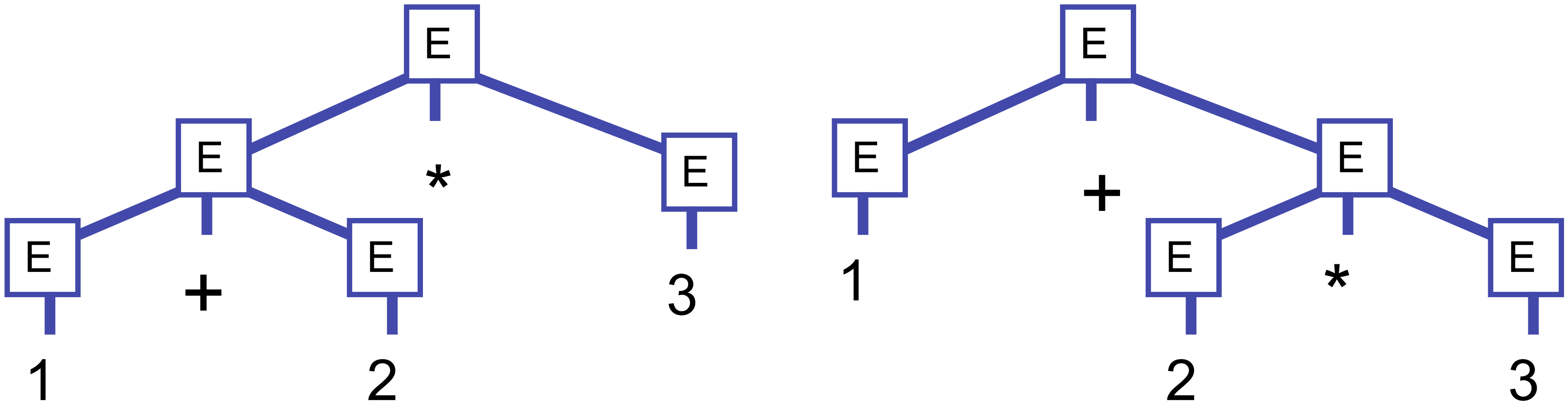
The language generated by G is the set of all terminal strings derivable from the start symbol S:

$$L(G) = \{w \in T^* \mid S \Rightarrow +w\}$$

For any subset W of T*, if there exists a CFG G such that L(G) = W, then W is called a Context-Free Language (CFL) over T.

Ambiguity

(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)



Both derivation trees correspond to the string

1 + 2 * 3

This type of ambiguity will cause problems when we try to go from strings to derivation trees!

Problem: Generation vs. Parsing

- **Context-Free Grammars (CFGs) describe how to to *generate***
- ***Parsing* is the inverse of generation,**
  - **Given an input string, is it in the language generated by a CFG?**
  - **If so, construct a derivation tree (normally called a *parse tree*).**
  - **Ambiguity is a big problem**

Note : recent work on Parsing Expression Grammars (PEGs) represents an attempt to develop a formalism that describes parsing directly. This is beyond the scope of these lectures …

We can often modify the grammar in order to eliminate ambiguity

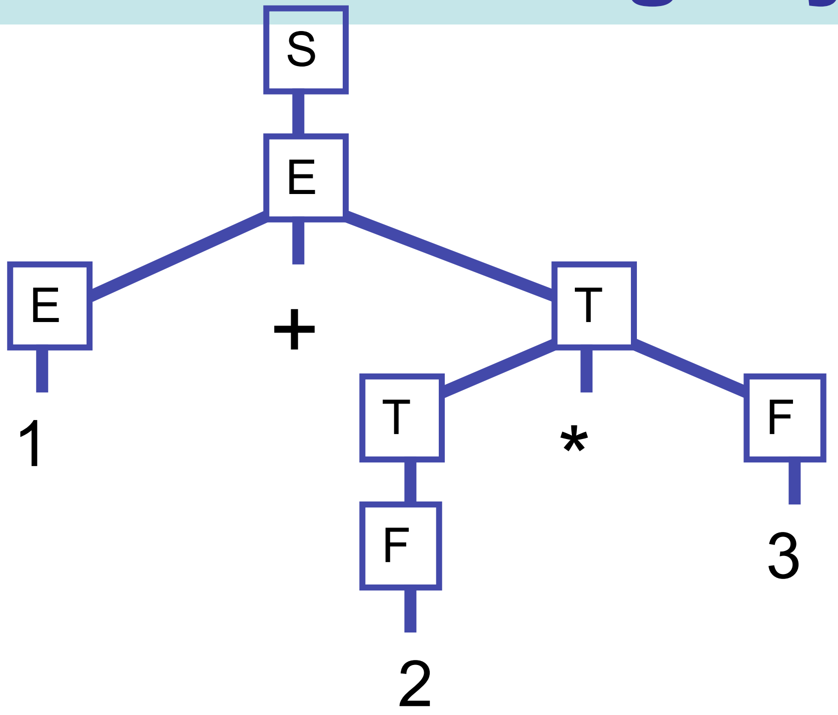
(G2)

S :: = E$ (start, $ = EOF)

E ::= E + T (expressions)
| E – T
| T

T ::= T * F (terms)
| T / F
| F

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



This is the *unique* derivation tree for the string

1 + 2 * 3$

Note: L(G1) = L(G2).
Can you prove it?

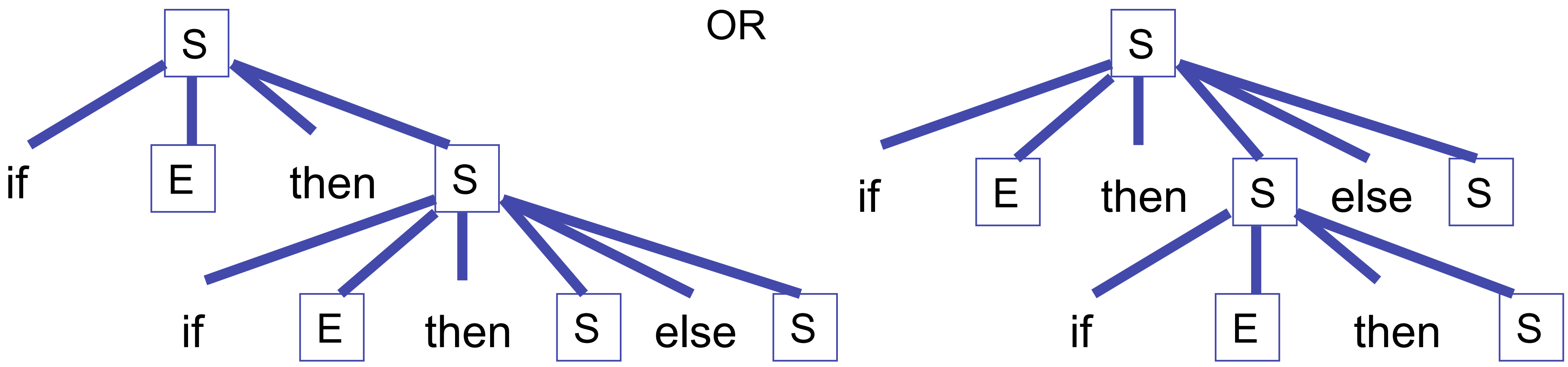
Famously Ambiguous

(G3) S ::= if E then S else S | if E then S | blah-blah

What does

if e1 then if e2 then s1 else s3

mean?



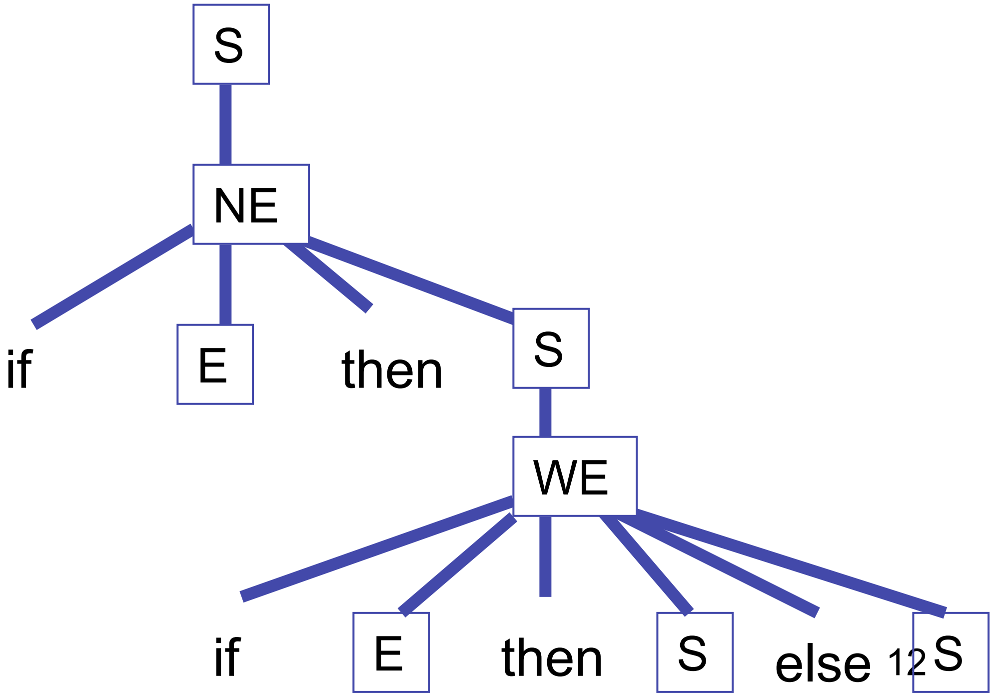
Rewrite?

(G4)
S ::= WE | NE
WE ::= **if** E **then** WE **else** WE | **blah-blah**
NE ::= **if** E **then** S
| **if** E **then** WE **else** NE

Now,

if e1 then if e2 then s1 else s3

has a unique derivation.



Note: L(G3) = L(G4).
Can you prove it?

Fun Fun Facts

See Hopcroft and Ullman, “Introduction to Automata Theory, Languages, and Computation”

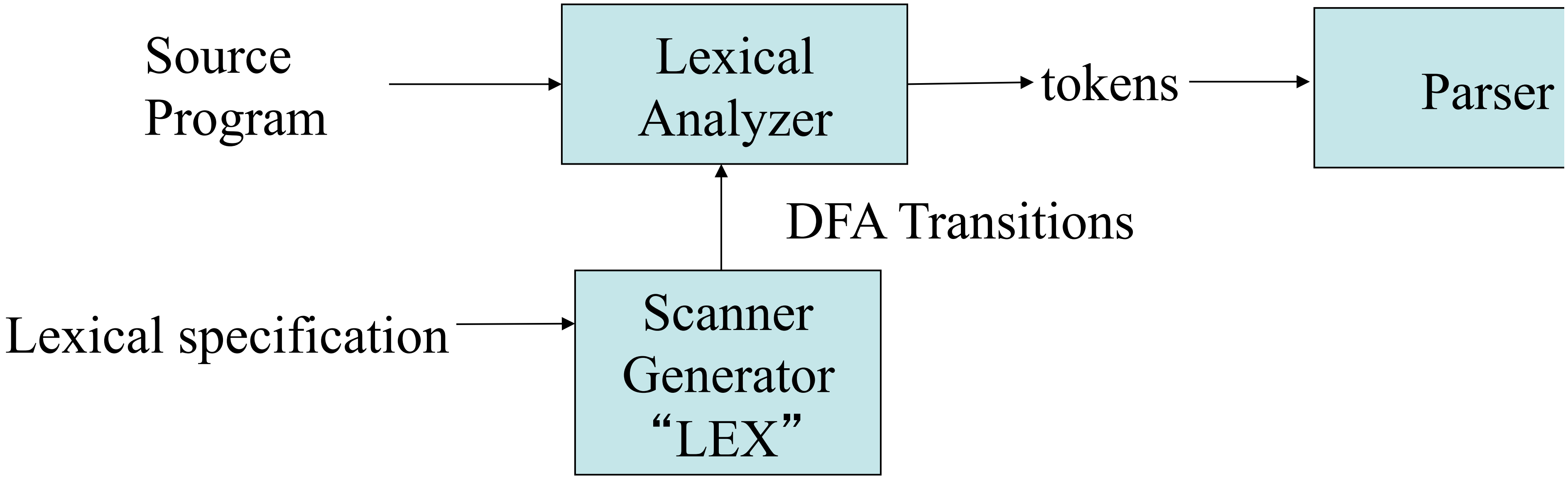
(1) Some context free languages are *inherently ambiguous* --- every context-free grammar will be ambiguous. For example:

$$L = \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\}$$

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!

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)

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.)

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,

$$e_1 \quad e_2 \quad \cdots \quad e_k$$

and an input string w, find a list of pairs

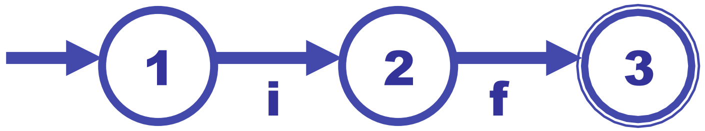
$$(i_1, w_1), (i_2, w_2), \ldots (i_n, w_n)$$

such that

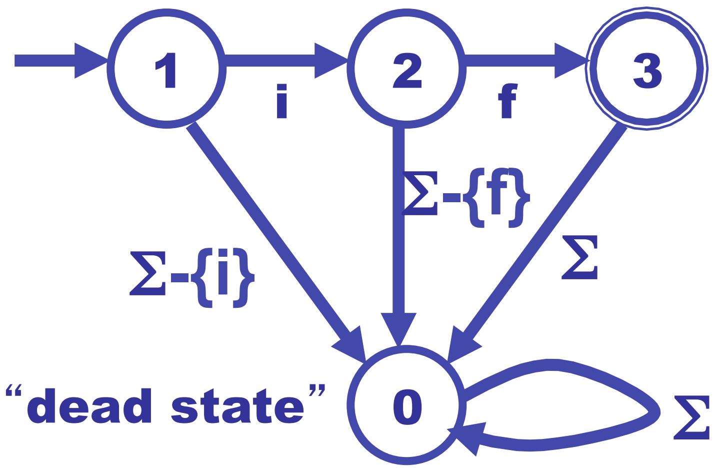
1) $w = w_1 w_2 \ldots w_n$
2) $w_j \in L(e_{i_j})$
3) $w_j \in L(e_s) \rightarrow i_j \leq s$ (priority rule)
4) $\forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \cdots w_n) : u \neq \varepsilon$
$\rightarrow \forall s : w_j u \notin L(e_s)$ (longest match)

Define Tokens with Regular Expressions (Finite Automata)

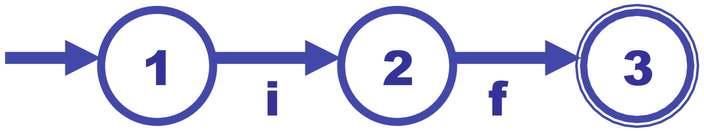
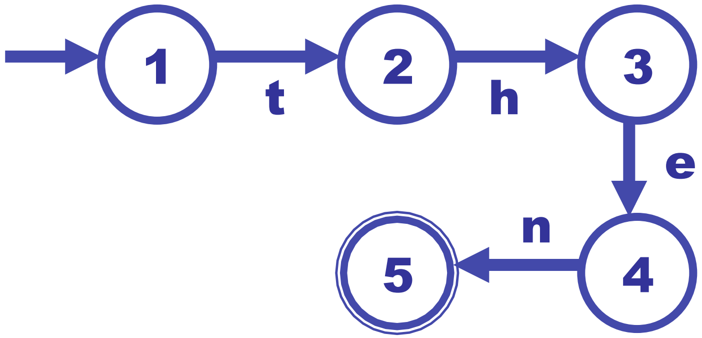
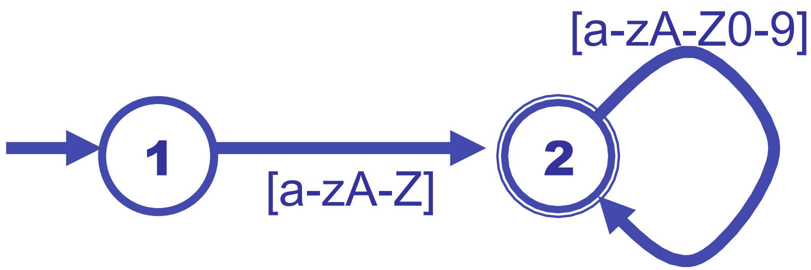
Keyword: if



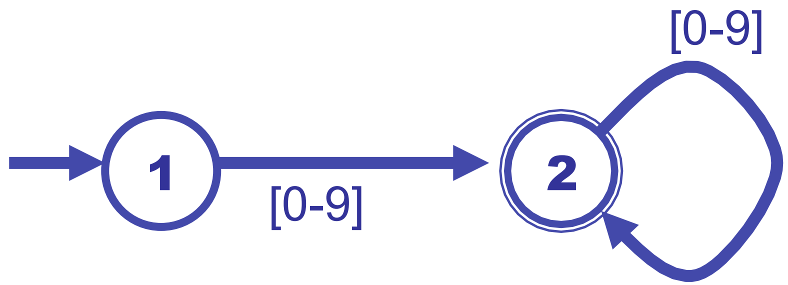
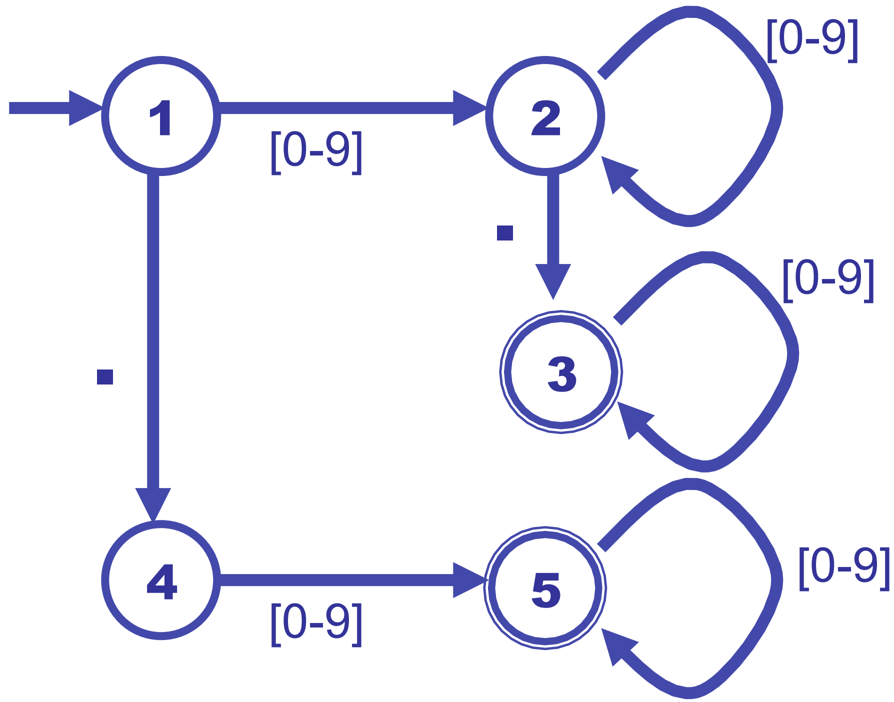
This FA is really shorthand for:



Define Tokens with Regular Expressions (Finite Automata)

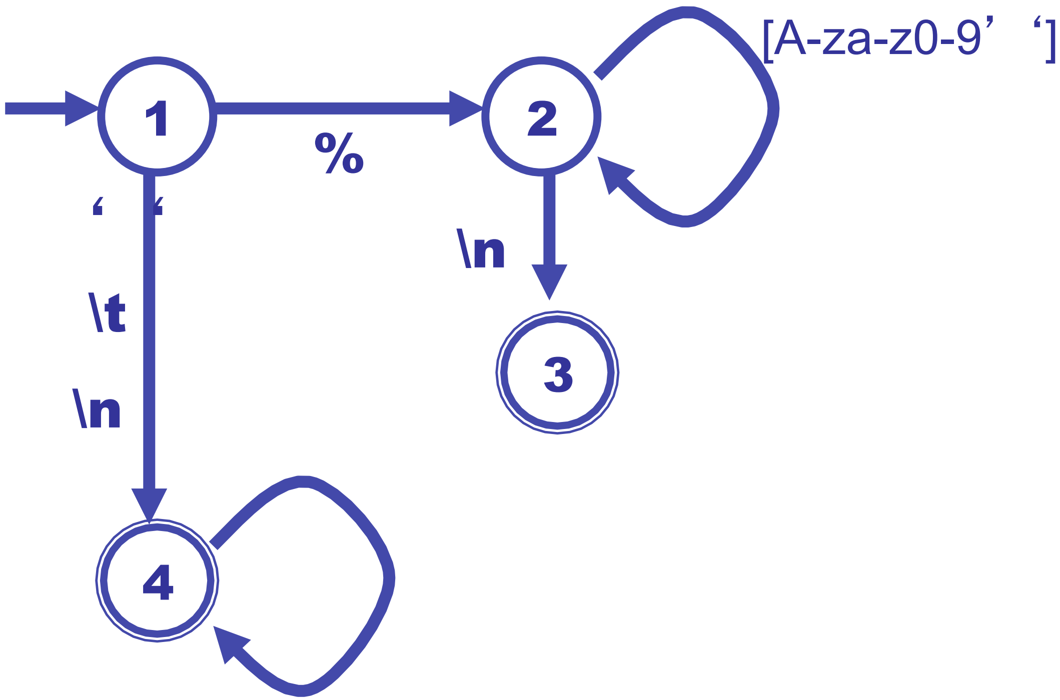
Regular Expression	Finite Automata	Token
Keyword: if		**KEY(IF)**
Keyword: then		**KEY(then)**
Identifier: [a-zA-Z][a-zA-Z0-9]*		**ID(s)**

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)**

No Tokens for “White-Space”

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



Constructing a Lexer

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

Construct all corresponding finite automata

Construct a single non-deterministic finite automata

Construct a single deterministic finite automata

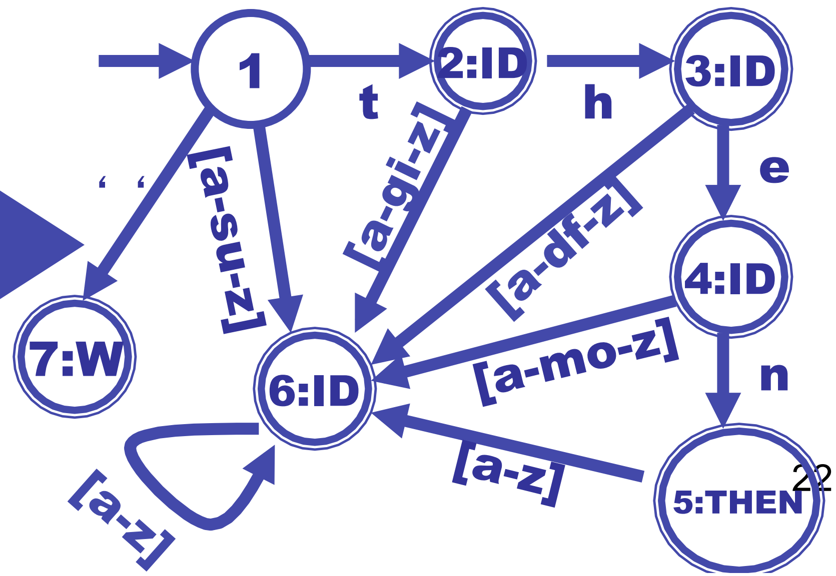
e_1, e_2, ⋮, e_k → NFA_1, NFA_2, ⋮, NFA_k → (use priority) → NFA → DFA

---

(1) Keyword : then

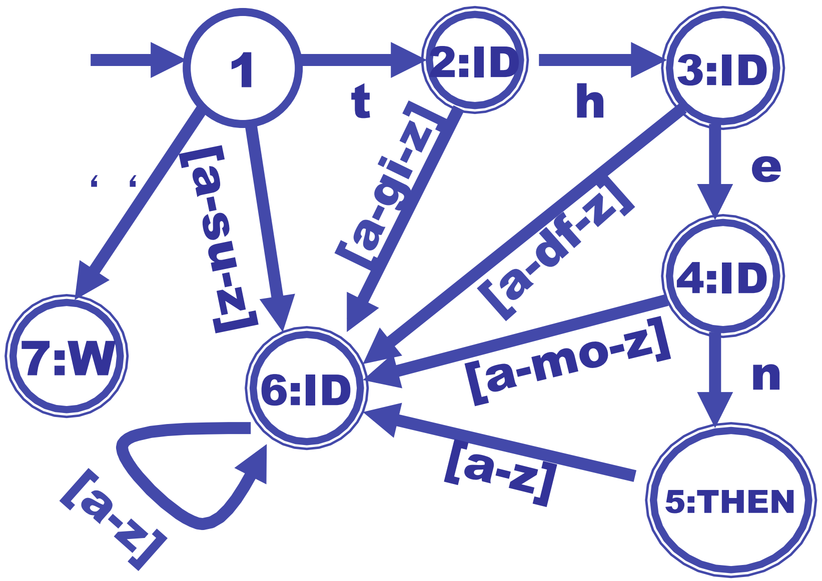
(2) Ident : [a-z][a-z]*

(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 EMIT KEY(THEN)
then| thenx$	1	0 RESET
then |thenx$	7	7
then t|henx$	0	7 EMIT WHITE(' ')
then |thenx$	1	0 RESET
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 EMIT ID(thenx)

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

From Andrew Appel, "Modern Compiler Implementation in Java" page 46

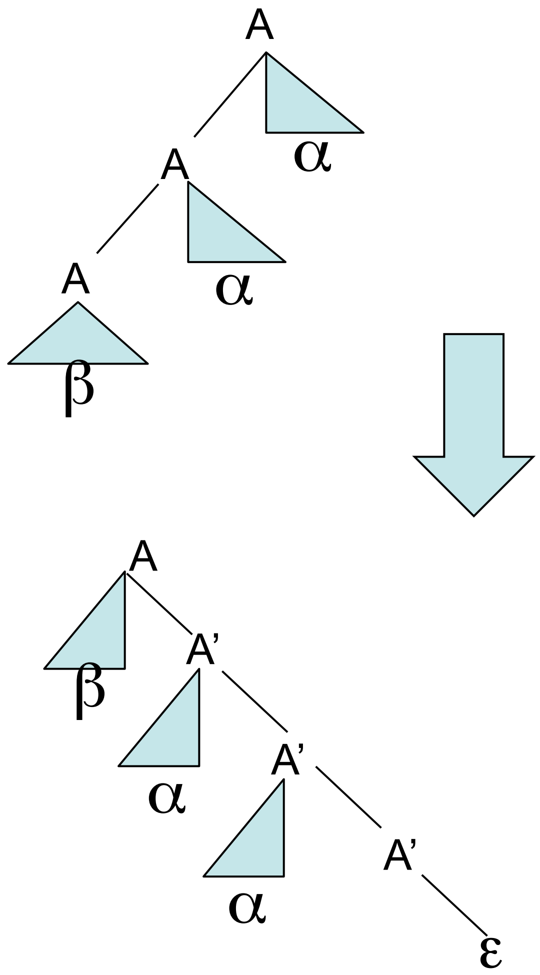
Eliminate Left-Recursion

Immediate left-recursion

$$A ::= A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha k \mid \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$$

$$A ::= \beta 1\ A' \mid \beta 2\ A' \mid \ldots \mid \beta n\ A'$$

$$A' ::= \alpha 1\ A' \mid \alpha 2\ A' \mid \ldots \mid \alpha k\ A' \mid \varepsilon$$



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

Eliminating Left Recursion

(G2)
S :: = E$

E ::= E + T
  | E – T
  | T

T ::= T * F
  | T / F
  | F

F ::= NUM
  | ID
  | (E)

Note that
  E ::= T and
  E ::= E + T
will cause problems
since FIRST(T) will be included
in FIRST(E + T) ---- so how can
we decide which poduction
To use based on next token?

Solution: eliminate "left recursion"!

  E ::= T E'

  E' ::= + T E'
    |

Eliminate left recursion

(G6)
S :: = E$

E ::= T E'

E' ::= + T E'
  | – T E'
  |

T ::= F T'

T' ::= * F T'
  | / F T'
  |

F ::= NUM
  | ID
  | (E)

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 of X can derive the empty string, false otherwise

nullable[Z] = false, for Z in T

nullable[Y1 Y2 … Yk] = nullable[Y1] and … nullable[Yk], for Y(i) in N union T.

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

FIRST[X Y1 Y2 … Yk] = FIRST[X] if not nullable[X]

FIRST[X Y1 Y2 … Yk] =FIRST[X] union FIRST[Y1 … Yk] otherwise

Computing First, Follow, and nullable

```
For each terminal symbol Z
   FIRST[Z] := {Z};
   nullable[Z] := false;

For each non-terminal symbol X
  FIRST[X] := FOLLOW[X] := {};
  nullable[X] := false;

repeat
   for each production X → Y1 Y2 … Yk
      if Y1, … Yk are all nullable, or k = 0
         then nullable[X] := true
      for each i from 1 to k, each j from i + l to k
         if Y1 … Y(i-1) are all nullable or i = 1
            then  FIRST[X] := FIRST[X] union FIRST[Y(i)]
         if Y(i+1) … Yk are all nullable or if i = k
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FOLLOW[X]
         if Y(i+1) … Y(j-1) are all nullable or i+1 = j
            then FOLLOW[Y(i)] := FOLLOW[Y(i)] union FIRST[Y(j)]
until there is no change
```

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'
| – T E'
|

T ::= F T'

T' ::= * F T'
| / F T'
|

F ::= NUM
| ID
| (E)

Predictive Parsing Table for G6

Table[X, T] = Set of productions

X ::= Y1…Yk in Table[X, T]
if T in FIRST[Y1 … Yk]
or if (T in FOLLOW[X] and nullable[Y1 … Yk])

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 ::= ID	F ::= NUM	

(entries for /, - are similar…)

Left-most derivation is constructed by recursive descent

Left-most derivation

```
(G6)
 S :: = E$

 E ::= T E'

 E'  ::= + T E'
        | – T E'
        |

T ::= F T'

T'  ::= * F T'
       | / F T'
       |

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

```
S  → E$
   → T E' $
   → F T'  E' $
   → ( E ) T'  E' $
   → ( T E'  ) T'  E' $
   → ( F T'  E'  ) T'  E' $
   → ( 17 T'  E'  ) T'  E' $
   → ( 17 E'  ) T'  E' $
   → ( 17 + T E'  ) T'  E' $
   → ( 17 + F T'  E'  ) T'  E' $
   → ( 17 + 4 T'  E'  ) T'  E' $
   → ( 17 + 4 E'  ) T'  E' $
   → ( 17 + 4 ) T'  E' $
   → ( 17 + 4 ) * F T'  E' $
   → …
   → …
   → ( 17 + 4 ) * ( 2 – 10 ) T'  E' $
   → ( 17 + 4 ) * ( 2 – 10 ) E' $
   → ( 17 + 4 ) * ( 2 – 10 )
```

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

As a stack machine

S → E$
→ T E' $
→ F T' E' $
→ (E) T' E' $
→ (T E') T' E' $
→ (F T' E') T' E' $
→ (17 T' E') T' E' $
→ (17 E') T' E' $
→ (17 + T E') T' E' $
→ (17 + F T' E') T' E' $
→ (17 + 4 T' E') T' E' $
→ (17 + 4 E') T' E' $
→ (17 + 4) T' E' $
→ (17 + 4) * F T' E' $
→ …
→ …
→ (17 + 4) * (2 – 10) T' E' $
→ (17 + 4) * (2 – 10) E' $
→ (17 + 4) * (2 – 10)

	E$
	T E' $
	F T' E' $
(	E) T' E' $
(	T E') T' E' $
(	F T' E') T' E' $
(17	T' E') T' E' $
(17	E') T' E' $
(17 +	T E') T' E' $
(17 +	F T' E') T' E' $
(17 + 4	T' E') T' E' $
(17 + 4	E') T' E' $
(17 + 4)	T' E' $
(17 + 4) *	F T' E' $
…	
…	
(17 + 4) * (2 – 10)	T' E' $
(17 + 4) * (2 – 10)	E' $
(17 + 4) * (2 – 10)	

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

(G7)

S :: = d | X Y S

Y ::= c |

X ::= Y | a

	Nullable	FIRST	FOLLOW
S	false	{ c,d ,a}	{ }
Y	true	{ c }	{ c,d,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 }

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 | ID = E | print (L)

E ::= ID | NUM | E + E | (S, E)

L ::= E | L, E

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

(G8)

S :: = S SEMI S | ID EQUAL E | PRINT LPAREN L RPAREN

E ::= ID | NUM | E PLUS E | LPAREN S COMMA E RPAREN

L ::= E | L COMMA E

A right-most derivation …

(G8)

S ::= S ; S
| ID = E
| print (L)

E ::= ID
| NUM
| E + E
| (S, E)

L ::= E
| L, E

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

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

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

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

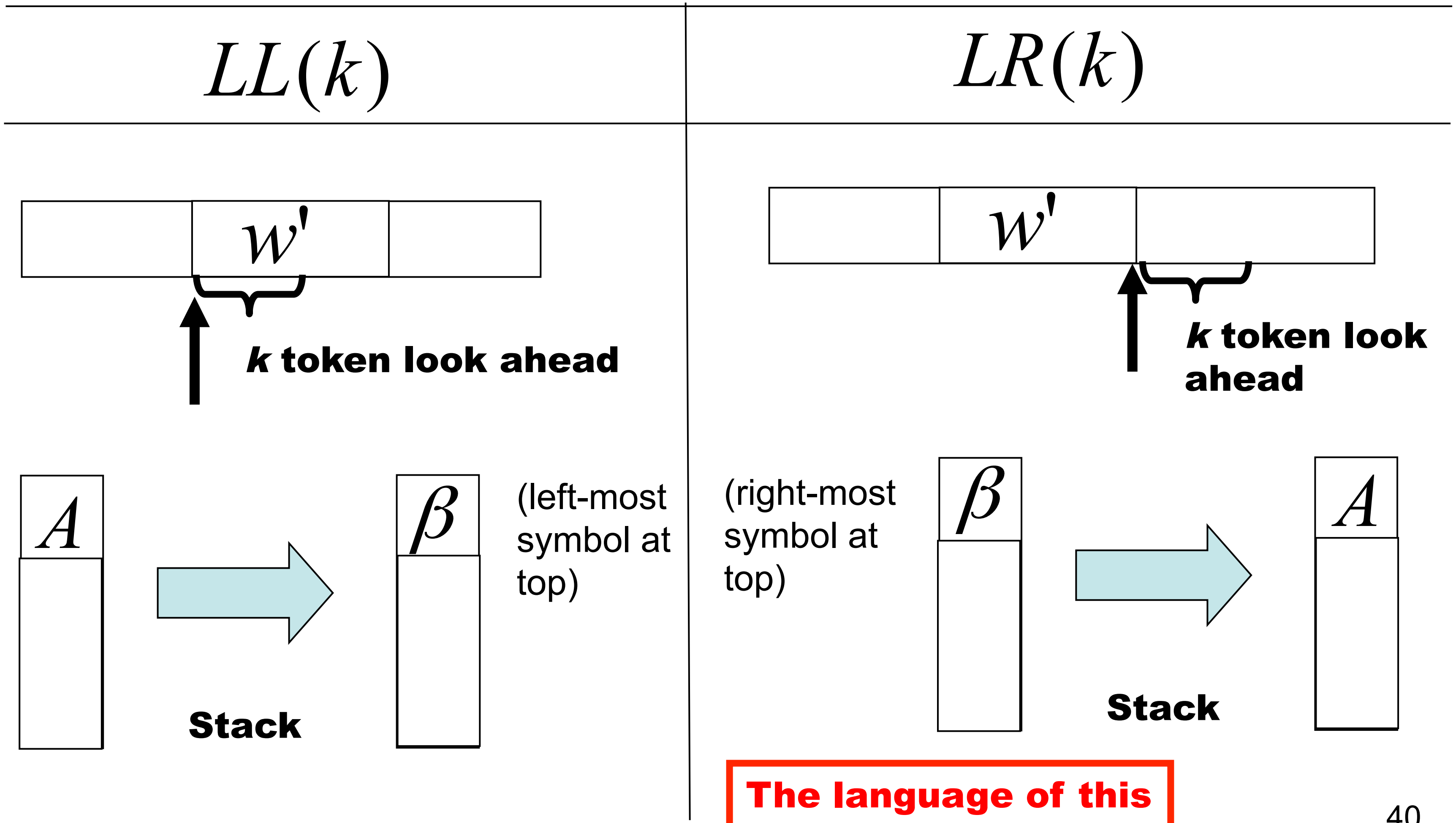
	a = 7 ; b = c + (d = 5 + 6, d)	s
ID	= 7 ; b = c + (d = 5 + 6, d)	s, s
ID = NUM	; b = c + (d = 5 + 6, d)	r E ::= NUM
ID = E	; b = c + (d = 5 + 6, d)	r S ::= ID = E
S	; b = c + (d = 5 + 6, d)	s, s
S ; ID	= c + (d = 5 + 6, d)	s, s
S ; ID = ID	+ (d = 5 + 6, d)	r E ::= ID
S ; ID = E	+ (d = 5 + 6, d)	s, s,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

SHIFT = LEX + move token to stack

ACTIONS

LL(k) vs. LR(k) reductions

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



The language of this Stack IS REGULAR!

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 ::= $\alpha \bullet \beta$

is an LR(0) item.

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

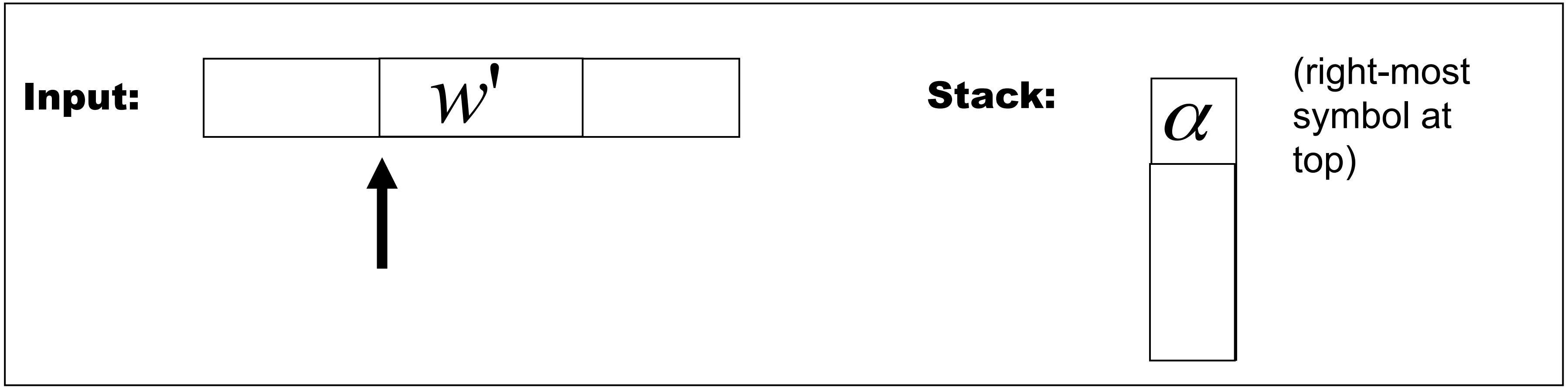
LR(0) items indicate what is on the stack (to the left of the •) and what is still in the input stream (to the right of the •)

LR(k) states (non-deterministic)

The state

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

should represent this situation:



with

$$\beta a_1 a_2 \cdots a_k \Rightarrow^{*} w'$$

Key idea behind LR(0) items

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

The NFA for LR(0) items

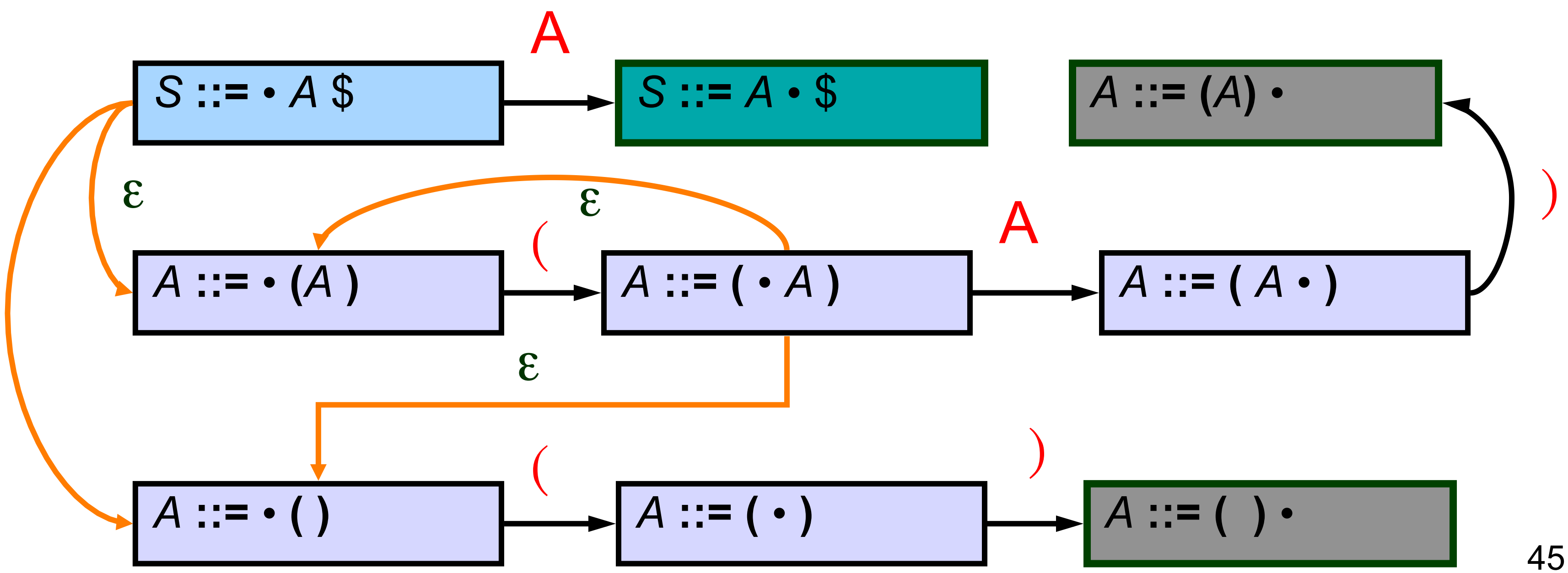
- The transition of LR(0) items can be represented by an NFA, in which
  - 1. each LR(0) item is a state,
  - 2. there is a transition from item $A ::= \alpha \bullet c \beta$
    to item $A ::= \alpha c \bullet \beta$ with label c, where c is a terminal symbol
  - 3. there is an ε-transition from item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

Example NFA for Items

S ::= • A $ S ::= A • $ A ::= • (A)

A ::= (• A) A ::= (A •) A ::= (A) •

A ::= • ( ) A ::= (•) A ::= ( ) •



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 ::= α • B β is in Closure**(I)** and B ::= • γ is an item, then add B ::= • γ to Closure**(I)**
- Repeat until no more new items can be added to Closure**(I)**

Examples of Closure

Closure({A ::= (• A)}) =

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

- closure({S ::= • A \$})

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

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

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

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

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

Examples of Goto

- Goto ({*A* ::= •(*A*)}, ()

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

- Goto ({*A* ::= (• *A*)}, *A*)

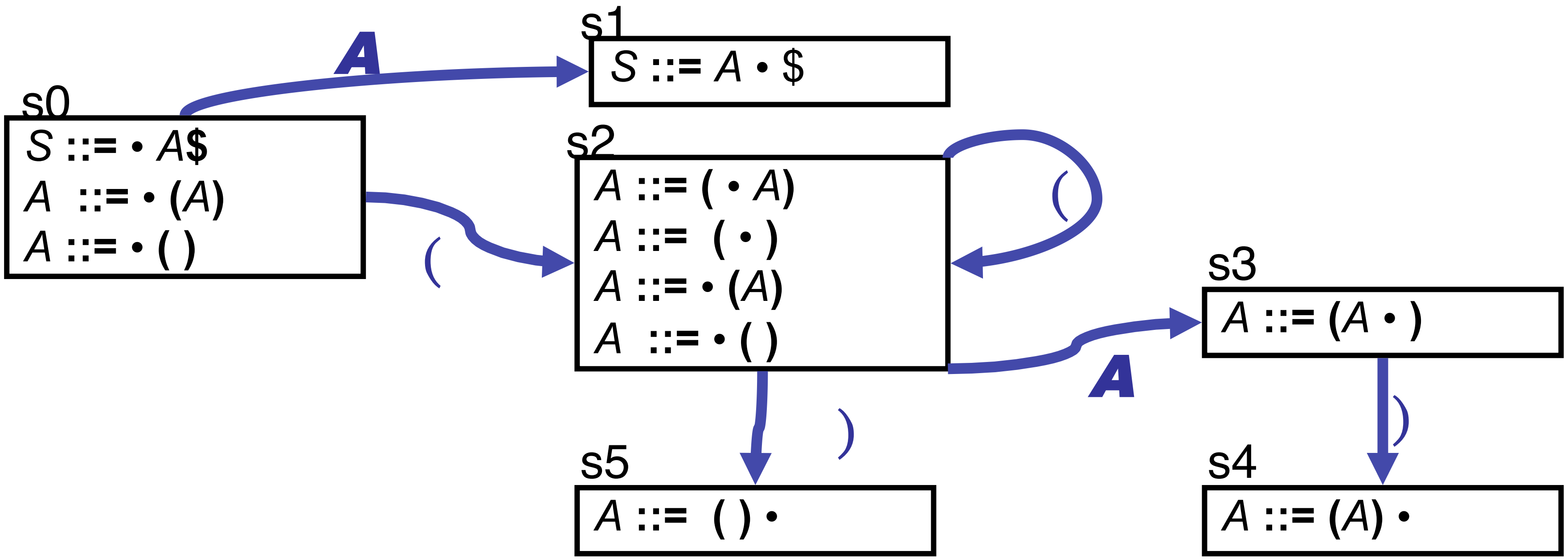
$$\left\{ A ::= (A \bullet\ ) \right\}$$

S ::= • A $
S ::= A • $
A ::= • (A)
A ::= (• A)
A ::= (A •)
A ::= (A) •
A ::= • ( )
A ::= (•)
A ::= ( ) •

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 Closure({ *S* **::= •** ***A*** $})
- Pick a state **I**
  - for each item A **::=** α • X β in **I**
    - find Goto**(I, X)**
    - if Goto**(I, X)** is not already a state, make one
    - Add an edge X from state **I** to Goto**(I, X)** state
- Repeat until no more additions possible

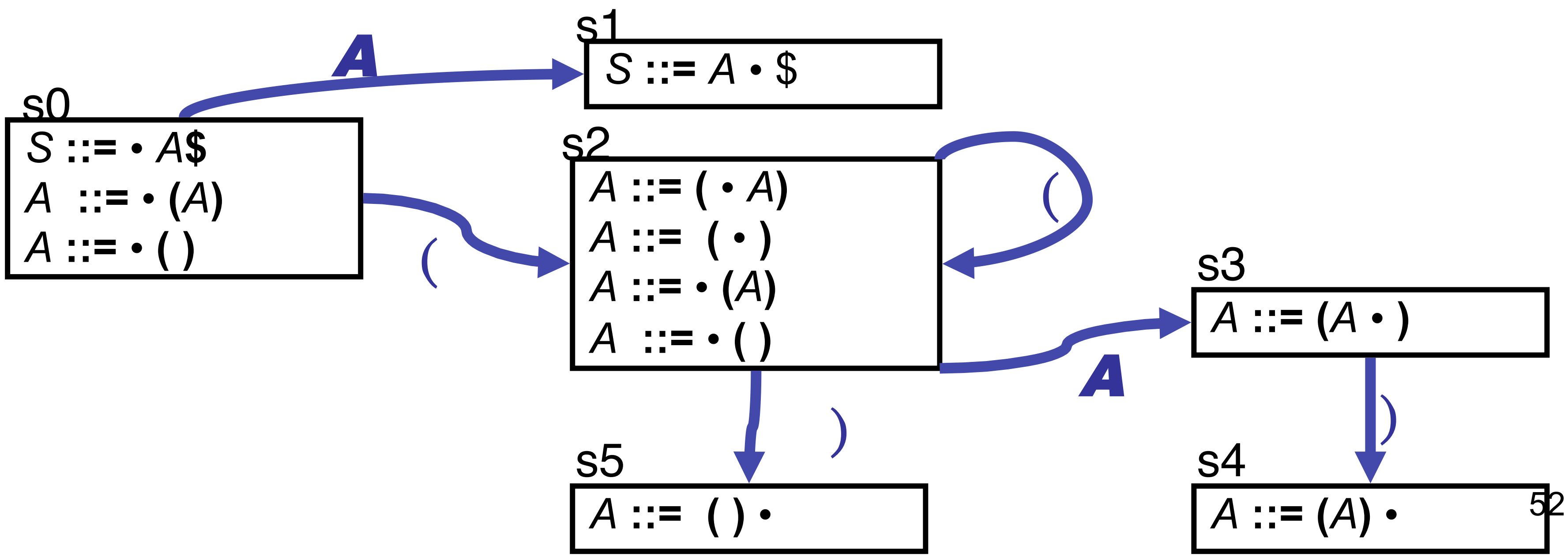
DFA Example



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)	



Parsing with an LR Table

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

If action is

- **shift sn : advance input one token, push sn on stack**
- **reduce X ::= α : 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 sn, push it on top of stack**
- **accept : stop and accept**
- **error : weep (actually, produce a good error message)**

Parsing, again…

(G10)

(1) S ::= A$

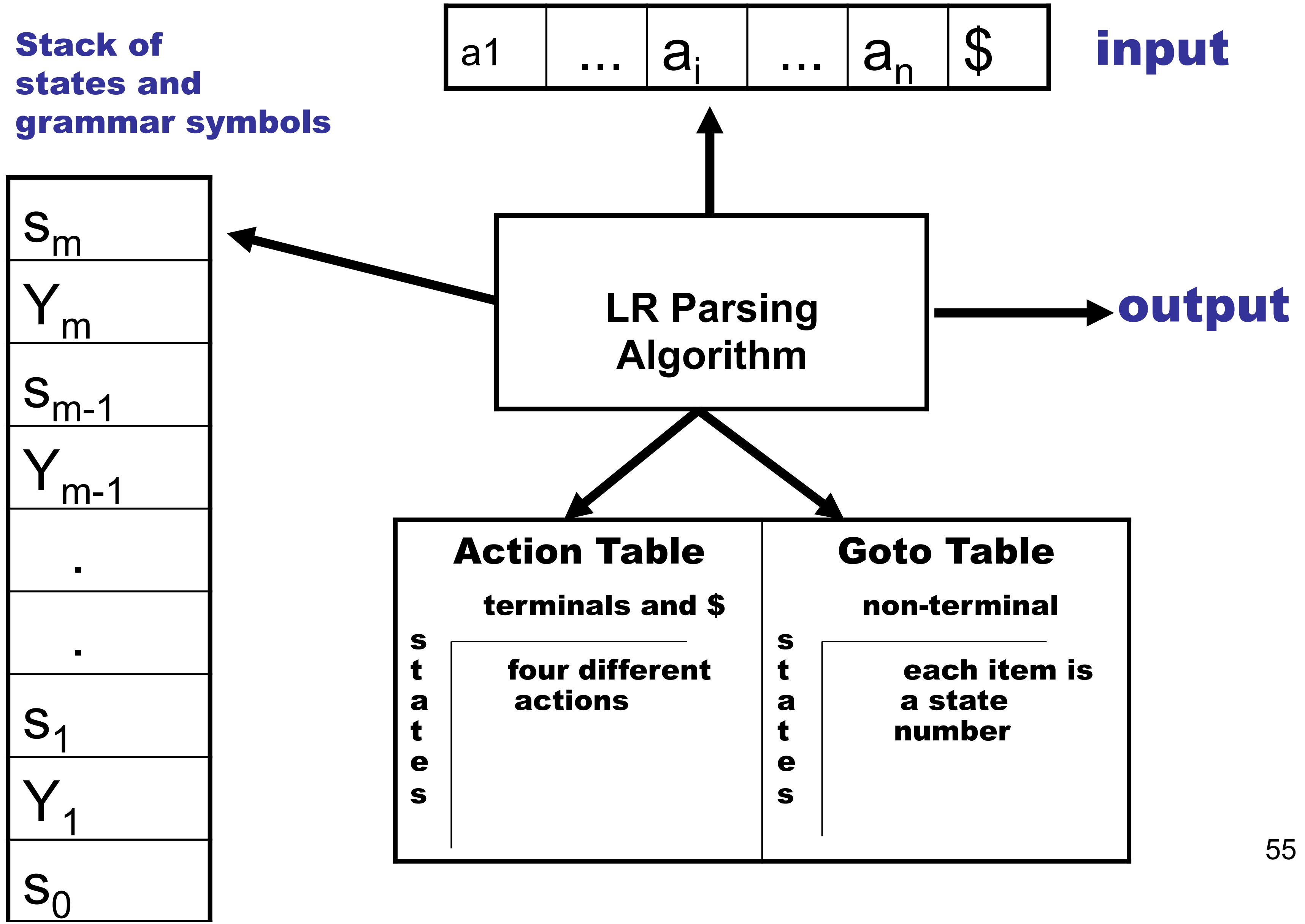
(2) A ::= (A)

(3) A ::= ( )

		ACTION		Goto
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)	

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!**

LR Parsing Algorithm



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

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

The Closure Operation

- Need to modify closure operation:.

Closure(Items) =

  repeat

    for each [X ::= α . Yβ, a] in Items

      for each production Y ::= γ

        for each b in First(βa)

          add [Y ::= .γ, b] to Items

  until Items is unchanged

Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items
- The start state contains (S' ::= .S$, dummy)
- A state that contains [X ::= α., b] is labeled with "reduce with X ::= α on lookahead b"
- And now the transitions …

The DFA Transitions

- A state s that contains [X ::= α.Yβ, b] has a transition labeled y to the state obtained from Transition(s, Y)
  - Y can be a terminal or a non-terminal

```
Transition(s, Y)
  Items = {}
  for each [X ::= α.Yβ, b] in s
      add [X ! αY.β, b] to Items
  return Closure(Items)
```

LR(1)-the parse table

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

- LR(1)-parse tables are very big

LR(1)-DFA

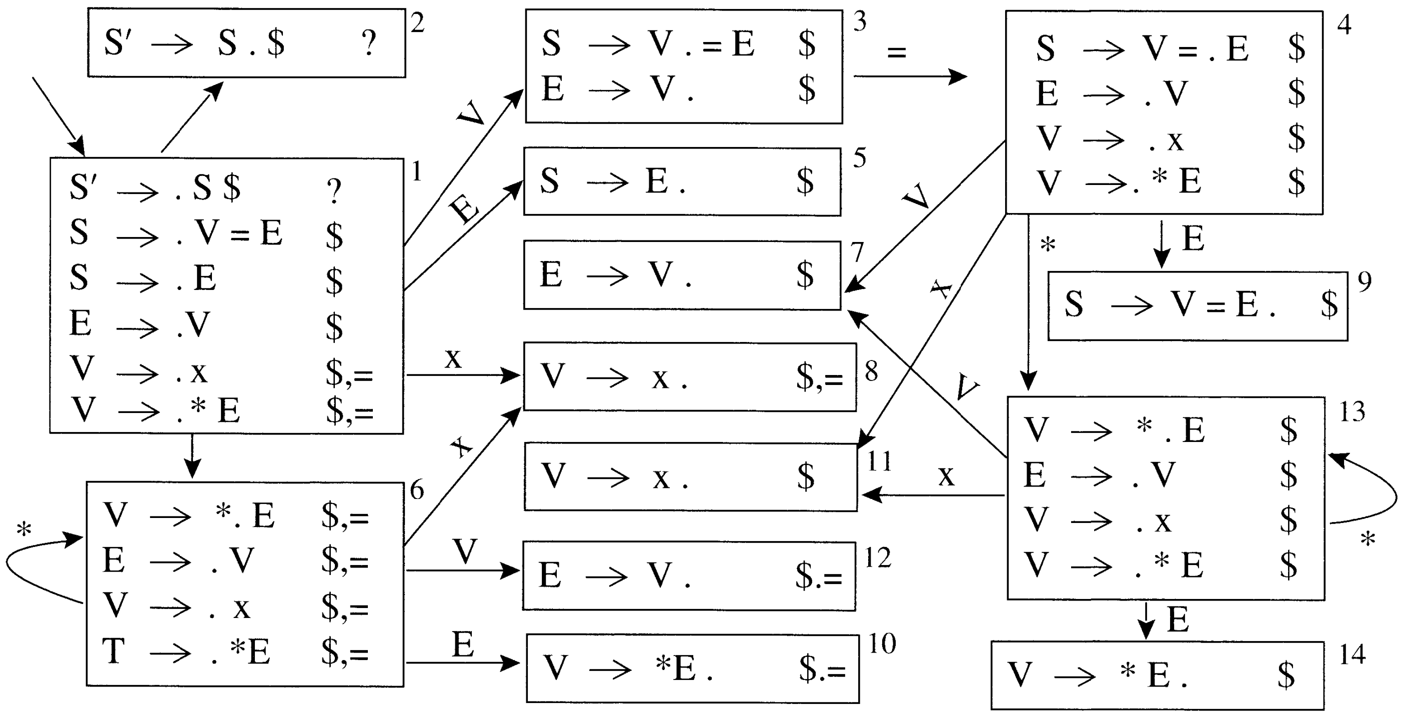
(G11)

S' ::= S$

S ::= V = E
| E

E ::= V

V ::= x
| *E



From Andrew Appel, “Modern Compiler Implementation in Java” page 65

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			

LALR States

- Consider for example the LR(1) states

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

$$\{[X ::= \alpha.\ ,\ b], [Y ::= \beta.\ ,\ d]\}$$

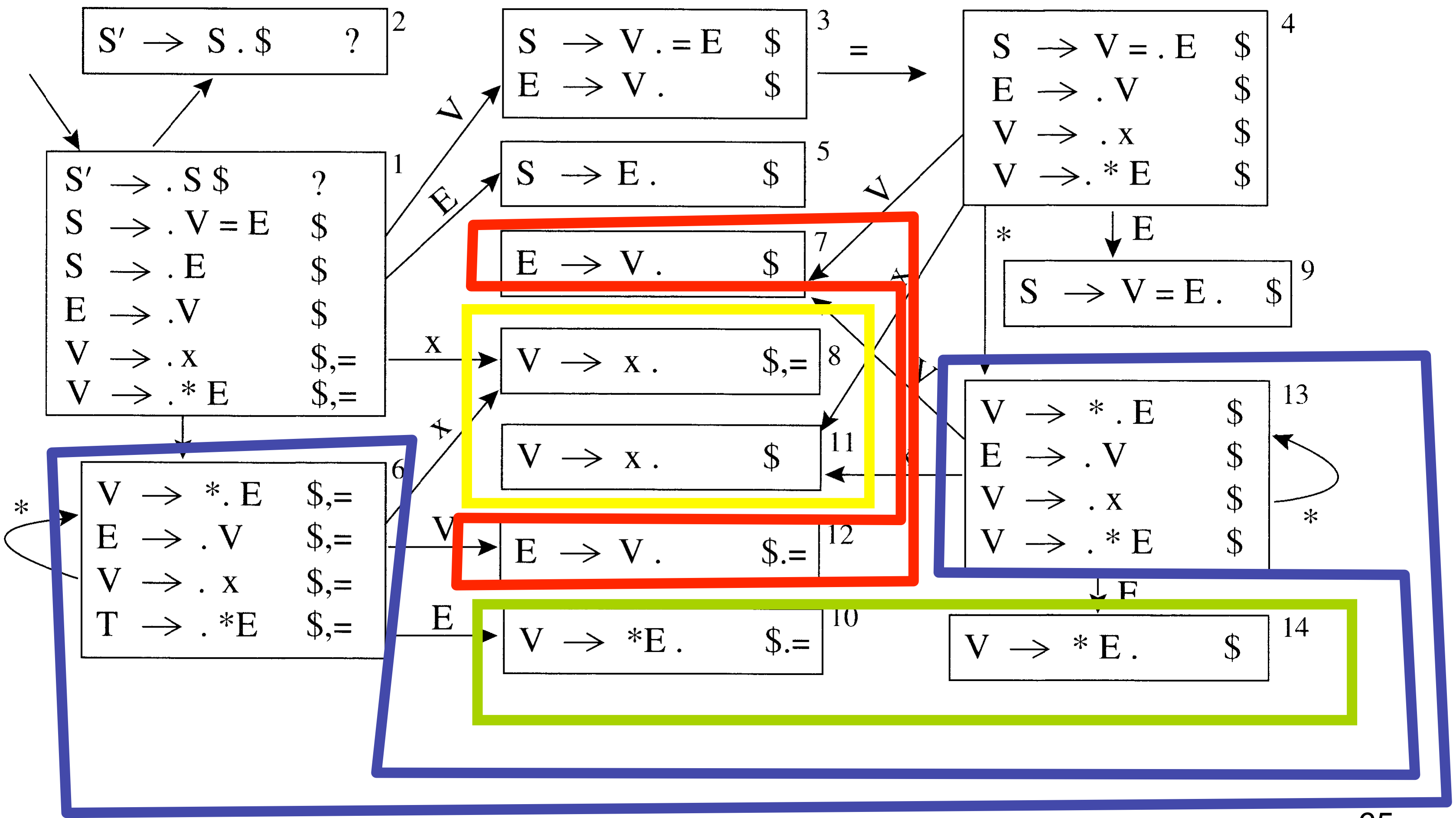
- They have the same *core* and can be merged to the state

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

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

For LALR(1), Collapse States ...

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



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			

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.