Compiler Construction
Lent Term 2020
Lecture 16
Parsing Part III : SLR(1)

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

This grammar will be our running example

$$G_2 = (N_2, T_1, P_2, E)$$

$N_2 = \{\mathrm{E}, \mathrm{T}, \mathrm{F}\}$ $\qquad$ $T_1 = \{+, *, (,), \mathrm{id}\}$

$P_2:$

$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T} \mid \mathrm{T}$ $\qquad$ (expressions)

$\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F} \mid \mathrm{F}$ $\qquad$ (terms)

$\mathrm{F} \rightarrow (\mathrm{E}) \mid \mathrm{id}$ $\qquad$ (factors)

A rightmost derivation of *(x+y)* forwards and backwards!

$$
\begin{aligned}
& E \\
& \Rightarrow_{rm} T \\
& \Rightarrow_{rm} F \\
& \Rightarrow_{rm} (E) \\
& \Rightarrow_{rm} (E+T) \\
& \Rightarrow_{rm} (E+F) \\
& \Rightarrow_{rm} (E+y) \\
& \Rightarrow_{rm} (T+y) \\
& \Rightarrow_{rm} (F+y) \\
& \Rightarrow_{rm} (x+y)
\end{aligned}
$$

FLIP!

$$
\begin{aligned}
& (x+y) \\
& \Leftarrow (F+y) \\
& \Leftarrow (T+y) \\
& \Leftarrow (E+y) \\
& \Leftarrow (E+F) \\
& \Leftarrow (E+T) \\
& \Leftarrow (E) \\
& \Leftarrow F \\
& \Leftarrow T \\
& \Leftarrow E
\end{aligned}
$$

View backwards derivation as a stack machine?

$$
\begin{aligned}
& (x+y) \\
\Leftarrow\ & (F+y) \\
\Leftarrow\ & (T+y) \\
\Leftarrow\ & (E+y) \\
\Leftarrow\ & (E+F) \\
\Leftarrow\ & (E+T) \\
\Leftarrow\ & (E) \\
\Leftarrow\ & F \\
\Leftarrow\ & T \\
\Leftarrow\ & E
\end{aligned}
$$

View the backwards derivation as a stack machine (use \$ as stack bottom and end-of-input).

stack	input
$\$$	$(x+y)\$$
$\$(F$	$+y)\$$
$\$(T$	$+y)\$$
$\$(E$	$+y)\$$
$\$(E+F$	$)\$$
$\$(E+T$	$)\$$
$\$(E)$	$\$$
$\$F$	$\$$
$\$T$	$\$$
$\$E$	$\$$

Can we make this work?

Let's invent "shift" and "reduce" actions and try to make it work. X=top-of-stack, a = input token

stack	input	action[X,a]
$\$$	$(x+y)\$$	shift (
$\$($	$x+y)\$$	shift id
$\$(id$	$+y)\$$	reduce $F \rightarrow id$
$\$(F$	$+y)\$$	reduce $T \rightarrow F$
$\$(T$	$+y)\$$	reduce $E \rightarrow T$
$\$(E$	$+y)\$$	shift +
$\$(E+$	$y)\$$	shift id

... BUT how do we decide when to shift and when to reduce?

stack	input	action[X,a]
$\$(E+id$	$)\$$	reduce $F \rightarrow id$
$\$(E+F$	$)\$$	reduce $T \rightarrow F$
$\$(E+T$	$)\$$	reduce $E \rightarrow E+T$
$\$(E$	$)\$$	shift)
$\$(E)$	$\$$	reduce $F \rightarrow (E)$
$\$F$	$\$$	reduce $T \rightarrow F$
$\$T$	$\$$	reduce $F \rightarrow E$
$\$E$	$\$$	accept!

Take a look at the stack contents.

$($	$(E+id$
$(id$	$(E+F$
$(F$	$(E+T$
$(T$	$(E$
$(E$	(E)
$(E+$	F
	T
	E

Amazing fact : the language of the stack is regular!

$LR(0)$ items

For every grammar production

$$\mathrm{A} \rightarrow \alpha\beta \qquad (\alpha, \beta \in (N \cup T)^*)$$

produce the LR(0) item

$$\mathrm{A} \rightarrow \alpha \bullet \beta$$

These will be the states of an NFA

accepting the "stack language".

Interpretation of state $A \rightarrow \alpha \bullet \beta$: we have read input w derivable from α ($\alpha \Rightarrow^*_{rm} w$) and we hope to see input derivable from β.

$LR(0)$ items for grammar G_2

$E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid id$

$E \to \bullet E + T$	$T \to \bullet T * T$	$F \to \bullet (E)$
$E \to E \bullet + T$	$T \to T \bullet * F$	$F \to (\bullet E)$
$E \to E + \bullet T$	$T \to T * \bullet F$	$F \to (E \bullet)$
$E \to E + T \bullet$	$T \to T * F \bullet$	$F \to (E) \bullet$
$E \to \bullet T$	$T \to \bullet F$	$F \to \bullet id$
$E \to T \bullet$	$T \to F \bullet$	$F \to id \bullet$

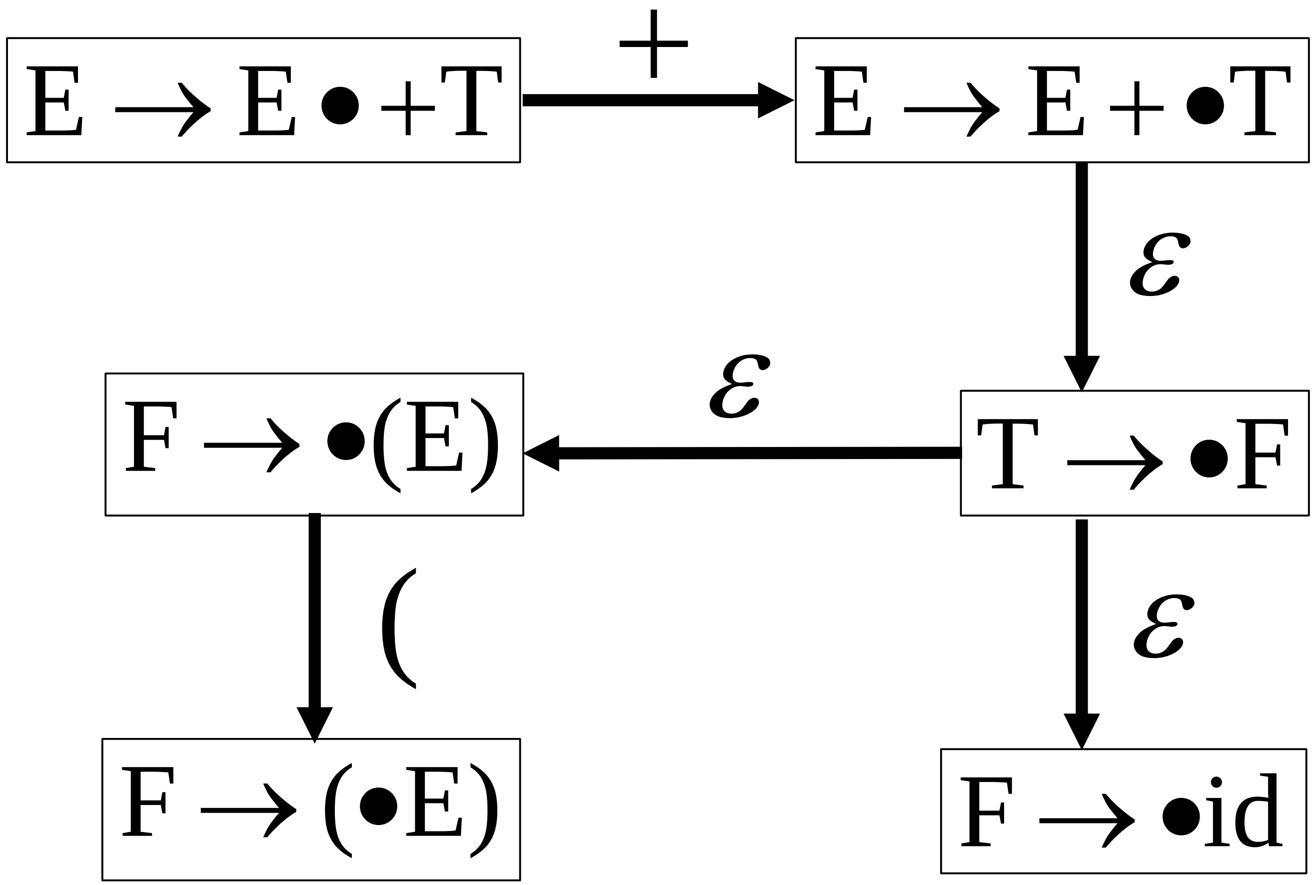
The NFA with $LR(0)$ items as states and every state is a final state

$$A \to \alpha \bullet c\beta \xrightarrow{\;c\;} A \to \alpha c \bullet \beta$$

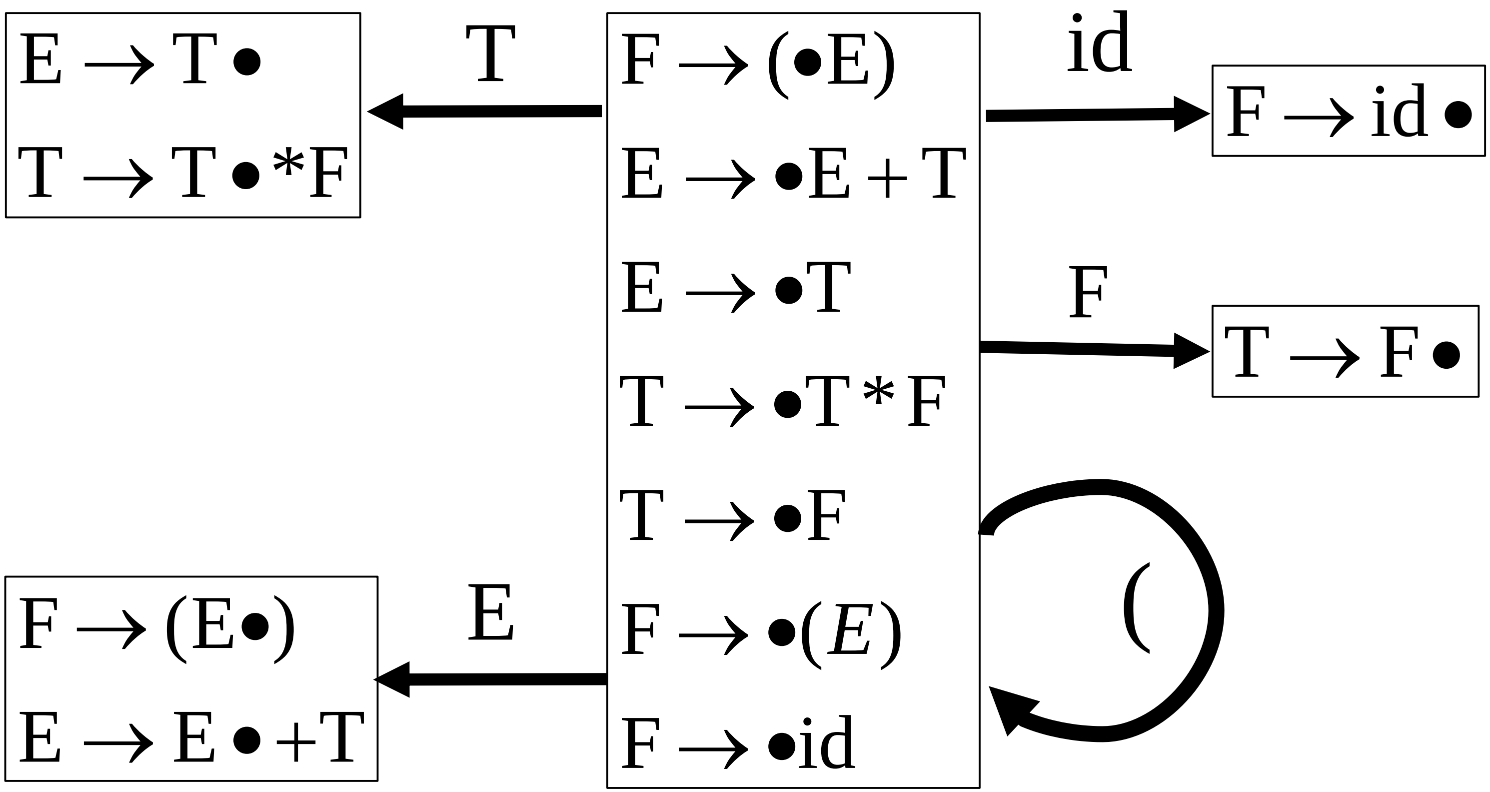
$$A \to \alpha \bullet B\beta \xrightarrow{\;\varepsilon\;} B \to \bullet\gamma$$

Now use the NFA to DFA algorithm to produce a DFA.

A few NFA transitions for grammar G_2



A few DFA transitions for grammar G_2



Start state?

In general, add new production $S' \to S$, where S is the original start symbol. For the simple term grammar, add production

$$E' \to E$$

which produces two items

$$E' \to \bullet E$$

$$E' \to E \bullet$$

DFA start state is $\varepsilon-\text{closure}(\{E' \to \bullet E\}) =$

$$\begin{array}{|l|}\hline E' \to \bullet E \\ E \to \bullet E + T \\ E \to \bullet T \\ T \to \bullet T * F \\ T \to \bullet F \\ F \to \bullet (E) \\ F \to \bullet \mathrm{id} \\ \hline \end{array}$$

The DFA transition function δ?

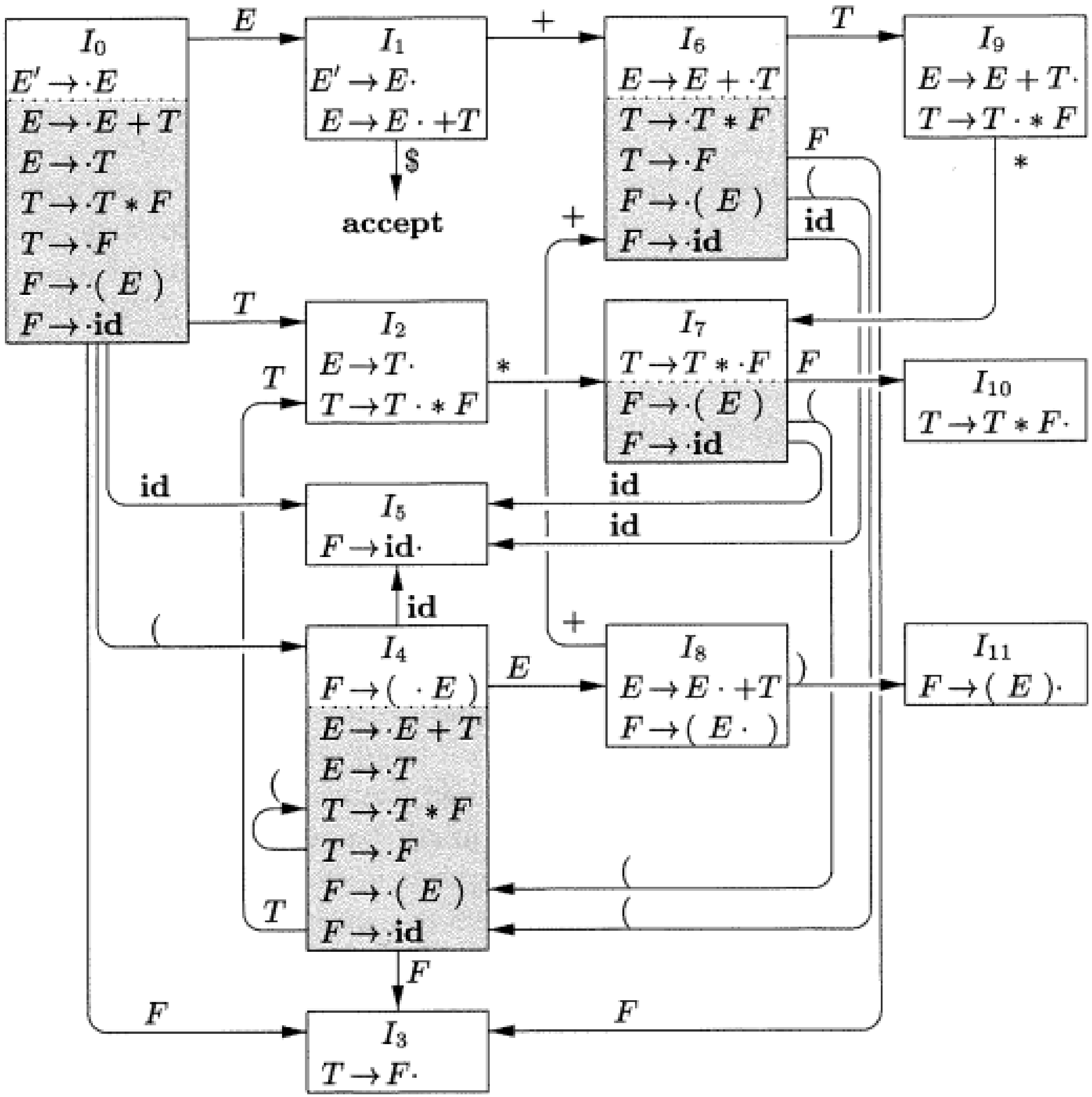
For this DFA

$$\delta(\mathrm{I}, \mathrm{X}) = \varepsilon\text{-closure}(\{\mathrm{A} \to \alpha \mathrm{X} \bullet \beta \mid \mathrm{A} \to \alpha \bullet \mathrm{X} \beta \in I\})$$

The book calls this GOTO(I, X).

The book also repeats the construction of DFA but this time specialised to LR(0) items (using function called CLOSURE). I see no reason to do this since we already know how to build a DFA from an NFA (see Lexing lecture).

Full DFA for the stack language of G_2



From Compilers by Aho, Lam, Sethi, Ullman

Replace the stack contents with state numbers

	0
$($	04
$(id$	045
$(F$	043
$(T$	042
$(E$	048
$(E+$	0486

$(E+id$	04865
$(E+F$	04863
$(E+T$	04869
$(E$	048
(E)	04 11
F	03
T	02
E	01

The generic LR algorithm

$a := \text{first symbol of input w\$}$

while(true)

$\quad s := \text{state at top of stack}$

$\quad \text{if ACTION[s, a] = shift t}$

$\quad \text{then push t on stack}$

$\qquad a := \text{next input token}$

$\quad \text{else if ACTION[s, a] = reduce } A \to \beta$

$\qquad \text{then pop } |\beta| \text{ states off the stack}$

$\qquad\quad t := \text{state at top of stack}$

$\qquad\quad \text{push GOTO[t, A] onto the stack}$

$\quad \text{else if ACTION[s, a] = accept}$

$\qquad \text{then accept and exit}$

$\quad \text{else ERROR}$

ACTION and GOTO for SLR(1)

If $[A \rightarrow \alpha \bullet a\beta] \in \mathrm{I_i}$ and $\delta(\mathrm{I_i}, a) = \mathrm{I_j}$ then ACTION[i,a] = shift j

If $[A \rightarrow \alpha \bullet] \in \mathrm{I_i}$ and $A \neq S'$

then for all $a \in$ FOLLOW(A),

$$\text{ACTION}[i,a] = \text{reduce } A \rightarrow \alpha$$

Note that there may be conflicts here!

If $[S' \rightarrow S \bullet] \in \mathrm{I_i}$ then ACTION[i,$] = accept

If $\delta(\mathrm{I_i}, A) = \mathrm{I_j}$ then GOTO[i, A] = j

(Now do you see why I prefer to use $\delta()$ rather than GOTO()?)

ACTION and GOTO for SLR(1)

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

From Compilers by Aho, Lam, Sethi, Ullman

Example parse

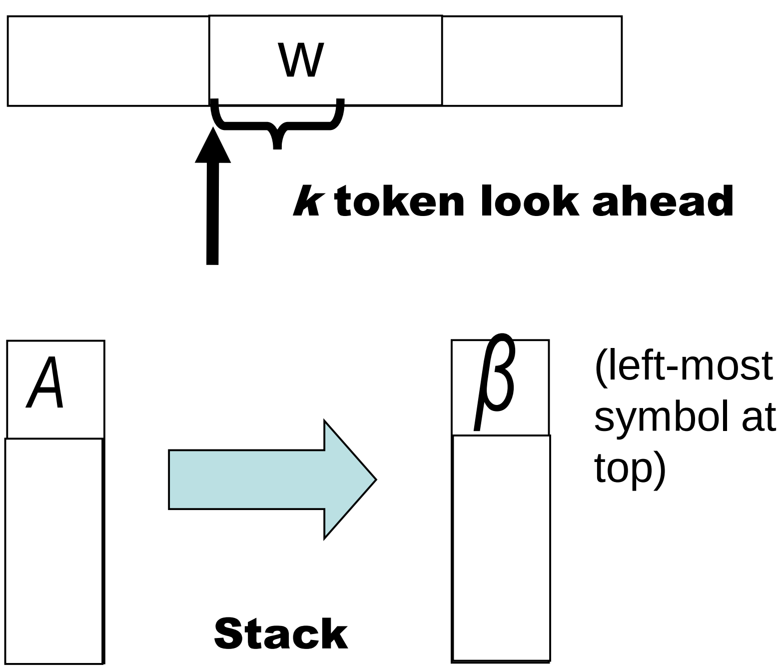
	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\ \$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\ \$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\ \$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\ \$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\ \$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\ \$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\ \$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\ \$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\ \$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

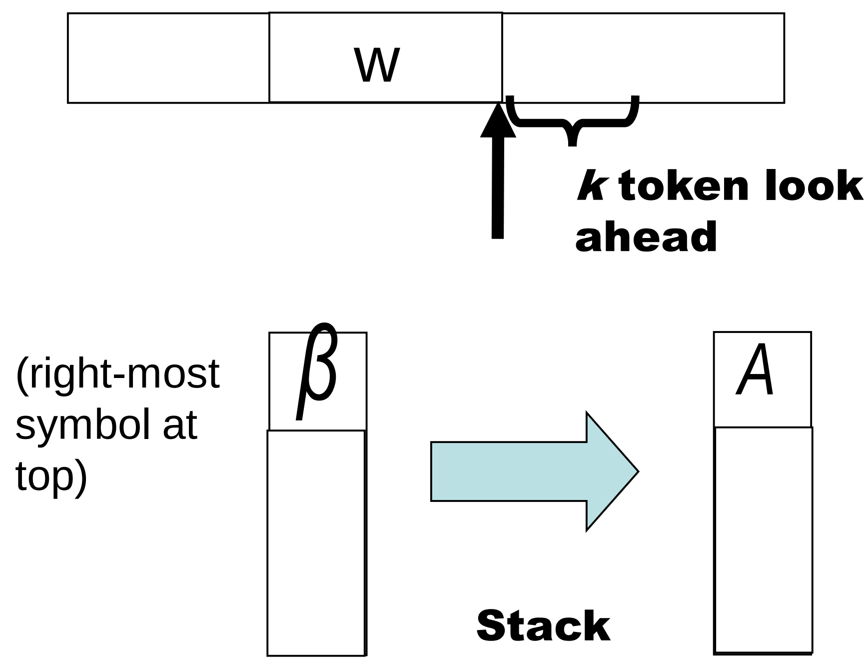
From Compilers by Aho, Lam, Sethi, Ullman

LL(k) vs. LR(k) reductions (SLR(1) as well)

$$A \rightarrow \beta \Rightarrow^{+} w \quad \beta \in (T \cup N)^{*} \quad w \in T^{*}$$

$LL(k)$	$LR(k)$





Beyond SLR(1)

Problems: there may be shift-reduce or reduce-reduce conflicts when ACTION and GOTO are not uniquely defined.

Either fix grammar or use a more powerful technique.

For example, LR(1) parsing starts with items of the form

$$[A \rightarrow \alpha \bullet \beta, a]$$

where a is an explicit look-ahead token.