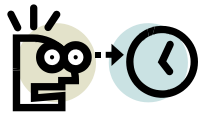
Shift/Reduce Parsing: Procrastination is Good

LR(k) Parsing Techniques

Wednesday, November 28, 2007
Reading: Appel 3.3



(Lyn before a CS235 lecture!)

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Thanks to: Randy, for many of the slides; Appel, for the examples.

Postponed decisions

- LL(k) parsing techniques must predict which production to use based on the next k tokens.
- LR(k)* parsers postpone the decision until it has seen input tokens of the entire right-hand side of the production in question.

*Stands for left-to-right parse, rightmost-derivation. We'll see why soon.

Rose Trees: A Motivational Example

Productions are numbered for easy reference

Appel uses $ instead of EOF

0	S'	→	S $
1	S	→	(L)
2	S	→	x
3	L	→	S
4	L	→	L , S

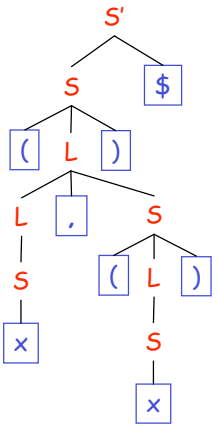
$G_{RoseTree}$
(Appel's Grammar 3.20)

Examples:

x$ (x)$ (x,x)$ (x,x,x)$

(x,(x))$ (((x,x),(x)),x,((x,x,x,x)))$

Parse tree for (x, (x))$



$G_{RoseTree}$ is Not LL(k) (i.e., not Predictive)

$G_{RoseTree}$

0	S'	→	S $
1	S	→	(L)
2	S	→	x
3	L	→	S
4	L	→	L , S

LL(1) Parsing Table

	x	(
S'	S' → S $	S' → S $
S	S → x	S → (L)
L	L → S L → L , S	L → S L → L , S

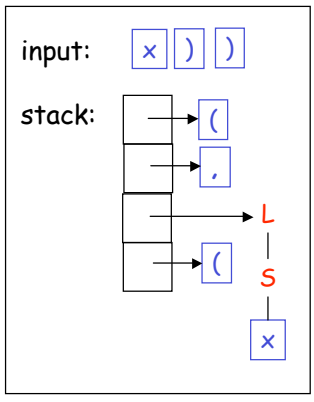
- $G_{RoseTree}$ is not LL(1)
- No amount of lookahead will help (consider (x,x,…,x)$)
- Left-recursion removal might help, but …
- Isn't there a better way?

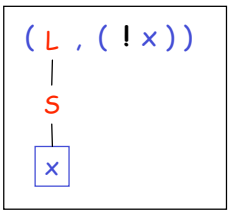
A Better Way: Shift/Reduce Parsing (LR(k))

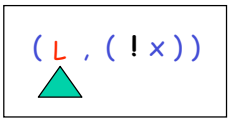
- The state of the parser (a **configuration**) has two components:
  1. The still-to-be-processed **input** tokens
  2. A **stack** of tokens and parse trees for the already processed tokens
- On each step of the parsing process, one of two actions occurs:
  1. The first input token is **shifted** to the top of the stack.
  2. The top k stack elements are **reduced** to a variable according to a production in the grammar.
- Parsing succeeds if there is a configuration where the stack contains only the "real" start symbol of the grammar (S, not S') after all input tokens except $ have been processed.
- Shift/reduce (LR(k)) parsing is more powerful than predictive (LL(k)) parsing because the decision of what to do next can take into account the stack elements as well as the next few input tokens

A Sample Configuration

Here's a sample intermediate configuration from parsing (x,(x))$ along with some abbreviations:







(To be more consistent with Appel's notatation, we should use a period rather than an exclamation point to separate stack and input, but the period is too hard to see.)

An Example: Parsing (x,(x))$

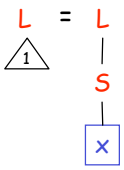
!(x,(x))$

⇒(!x,(x))$ *shift* (

⇒(x!,(x))$ *shift* x

⇒(S!,(x))$ *reduce* S → x

⇒(L!,(x))$ *reduce* L → S, where



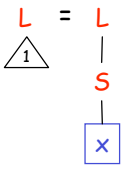
⇒(L,!(x))$ *shift* ,

⇒(L,(!x))$ *shift* (

⇒(L,(x!))$ *shift* x

Parsing (x,(x))$ (Continued)

(L,(x!))$

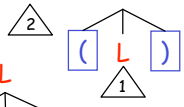


⇒(L,(S!))$ *reduce* S → x

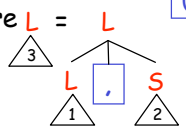
⇒(L,(L!))$ *reduce* L → S

⇒(L,(L)!)$ *shift*)

⇒(L, S!)$ *reduce* S → (L), where



⇒(L!)$ *reduce* L → L,S, where

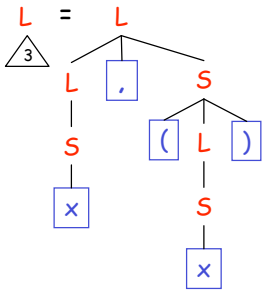


Parsing (x,(x))$ (Continued Again)

(L !) $
3

⇒ (L) ! $ *shift*)
3

⇒ S ! $ *reduce* S → (L) , where S = S
4 4



By convention, this is an accepting state
(we never reduce S' → S $)

A Rightmost Derivation of (x, (x))$

S' ⇒ S $	*by* S' → S $
⇒ (L) $	*by* S → (L)
⇒ (L , S) $	*by* L → L , S
⇒ (L , (L)) $	*by* S → (L)
⇒ (L , (S)) $	*by* L → S
⇒ (L , (x)) $	*by* S → x
⇒ (S , (x)) $	*by* L → S
⇒ (x , (x)) $	*by* S → x

Observe that our shift/reduce parsing example for (x, (x))$ performs the productions of the rightmost derivation precisely in reverse order.

If we had constructed a derivation rather than a parse tree, we would have constructed precisely the rightmost derivation.

LR(k) Parsing

A context-free grammar is LR(k) iff it can be parsed by a shift/reduce parser using *k* tokens of lookahead from the input.

In **LR**(k)

- The **L** means that the tokens are processed **L**eft-to-right
- The **R** means that the result of parsing is a parse tree constructed via a **R**ightmost derivation.

LR(k) parsing is guided by a parsing table. Studying such tables is the next item on our agenda.

We'll see that G_{RoseTree} is LR(0) : the parser doesn't actually need to look at any input tokens in order to determine whether to shift or reduce. It makes this decision based on the stack alone.

We'll also see a grammar that is LR(1) but not LR(0).

Sandwiched between LR(0) and LR(1) is something called SLR.

A Parsing Table for G_{RoseTree}

0	S' → S $
1	S → (L)
2	S → x
3	L → S
4	L → L , S

tokens | variables

top of stack states	(	)	x	,	$	S	L
1: ⊥	s3		s2			g4	
2: x	r2	r2	r2	r2	r2		
3: (	s3		s2			g7	g5
4: ⊥ S					a		
5: (L		s6		s8			
6: (L)	r1	r1	r1	r1	r1		
7: (S	r3	r3	r3	r3	r3		
8: (L ,	s3		s2			g9	
9: (L , S	r4	r4	r4	r4	r4		

sk **shifts** configuration … (*i*: … γ) ! †T to … (*k*: γ †) ! T

rn **reduces** configuration … (*i*: … γ) (*j*: δ) ! T to … (*k*: … γ V) ! T, where the *n*th production is V → δ and in state *i*, V **goes to** *k* via *gk*

a **accepts** the configuration ⊥S ! $, where S is the "real" start symbol

More About Parsing Tables

0	S' → S $
1	S → (L)
2	S → x
3	L → S
4	L → L , S

tokens — variables

top of stack states

	(	)	x	,	$	S	L
1: ⊥	s3		s2			g4	
2: x	r2	r2	r2	r2	r2		
3: (	s3		s2			g7	g5
4: ⊥ S					a		
5: (L		s6		s8			
6: (L)	r1	r1	r1	r1	r1		
7: (S	r3	r3	r3	r3	r3		
8: (L ,	s3		s2			g9	
9: (L , S	r4	r4	r4	r4	r4		

Stack states can be determined by an FA reading stack elements bottom up.

This table is LR(0) because the decision about whether to shift or reduce is based only on stack state.

Many of the reduce entries in this table are bogus (no valid configuration)

Table-Guided LR Parsing

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

$_1$! (x , (x)) $

$\Rightarrow_1$ ($_3$! x , (x)) $

$\Rightarrow_1$ ($_3$ x $_2$! , (x)) $

$\Rightarrow_1$ ($_3$ S $_7$! , (x)) $ reduce by rule 2: S → x

$\Rightarrow_1$ ($_3$ L $_5$! , (x)) $ reduce by rule 3: L → S

$\Rightarrow_1$ ($_3$ L $_5$, $_8$! (x)) $ shift ,

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ ($_3$! x)) $ shift (

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ ($_3$ x $_2$!)) $ shift x

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ ($_3$ S $_7$!)) $ reduce by rule 2: S → x

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ ($_3$ L $_5$!)) $ reduce by rule 3: L → S

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ ($_3$ L $_5$) $_6$!) $ shift)

$\Rightarrow_1$ ($_3$ L $_5$, $_8$ S $_9$!) $ reduce by rule 1: S → (L)

$\Rightarrow_1$ ($_3$ L $_5$!) $ reduce by rule 4: L → L , S

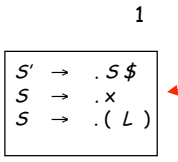
$\Rightarrow_1$ ($_3$ L $_5$) ! $ shift)

$\Rightarrow_1$ S $_4$! $ reduce by rule 1: S → (L)

⇒ **accept!**

How To Construct an LR(0) Parsing Table?

Idea: Starting with the top production S' → S $,
create collections of abstract configurations that denote stack states, and simulate shift/reduce parsing on these states.



State is a set of **items**.
An item is a single abstract configuration

(In our states, we'll use periods rather than exclamation points to be consistent with Appel.)

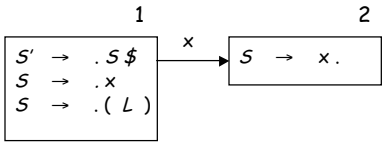
An item is a grammar rule combined with a dot that indicates a position in its right0hand side.

0 S' → S $

1 S → (L)
3 S → x

3 L → S
4 L → L , S

Shift actions

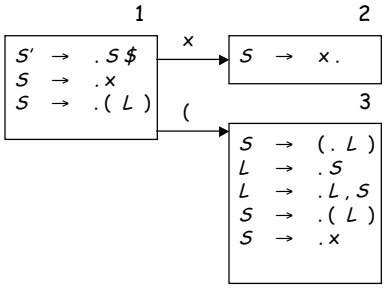


0 S' → S $

1 S → (L)
3 S → x

3 L → S
4 L → L , S

Or we might shift a (

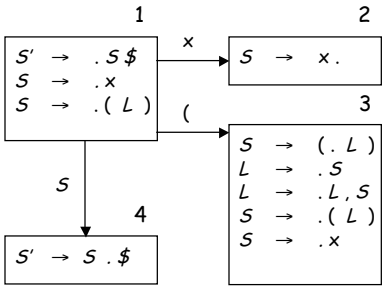


0 S' → S $

1 S → (L)
3 S → x

3 L → S
4 L → L , S

Goto actions

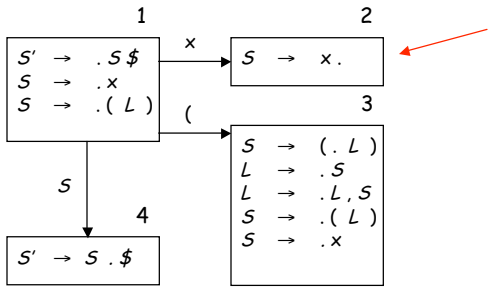


0 S' → S $

1 S → (L)
3 S → x

3 L → S
4 L → L , S

Reduce actions



Dot at the end of an item indicates the recognition of the right-hand side of a production

0 S' → S $

1 S → (L)
3 S → x

3 L → S
4 L → L , S

Closure and goto

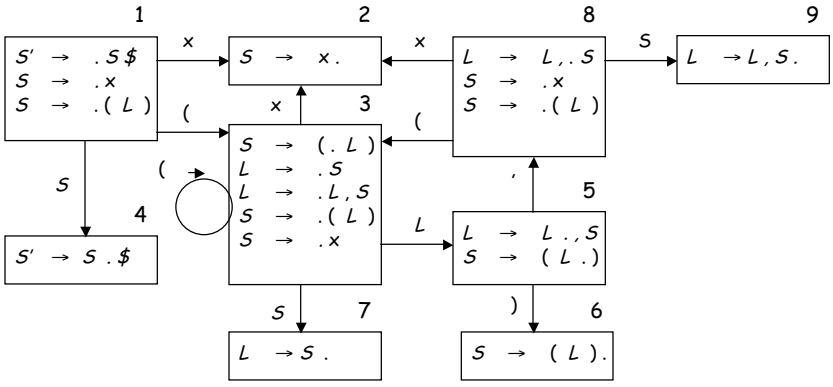
Closure(I) =
  repeat
    for any item $A \rightarrow \alpha.X\beta$ in I
      for any production $X \rightarrow \gamma$
        $I \leftarrow I \cup \{X \rightarrow .\gamma\}$
  until I does not change
  return I

Goto(I, X) =
  set J to the empty set
    for any item $A \rightarrow \alpha.X\beta$ in I
      add $A \rightarrow \alpha X.\beta$ to J
    return Closure(J)

LR(0) parser construction algorithm

Initialize T to {**Closure**($S' \rightarrow . S \$$)}
Initialize E to empty
repeat
  for each state I in T
    for any item $A \rightarrow \alpha.X\beta$ in I
      let J be **Goto**(I, X)
        $T \leftarrow T \cup \{J\}$
        $E \leftarrow T \cup \{I \rightarrow J\}$
until E and T do not change in this iteration

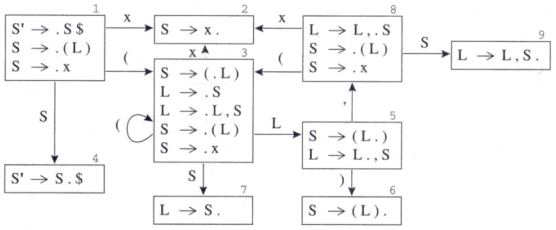
LR(0) states for $G_{RoseTree}$

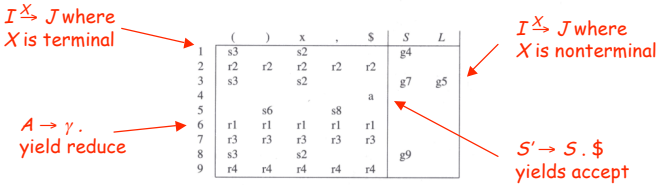


0 $S' \rightarrow S \$$
1 $S \rightarrow (L)$
3 $S \rightarrow x$
3 $L \rightarrow S$
4 $L \rightarrow L , S$

LR(0) reduce actions & parse table

$R \leftarrow \{\}$
for each state I in T
  for each item $A \rightarrow \alpha\ .$ in I
    $R \leftarrow R \cup \{ (I, A \rightarrow \alpha) \}$



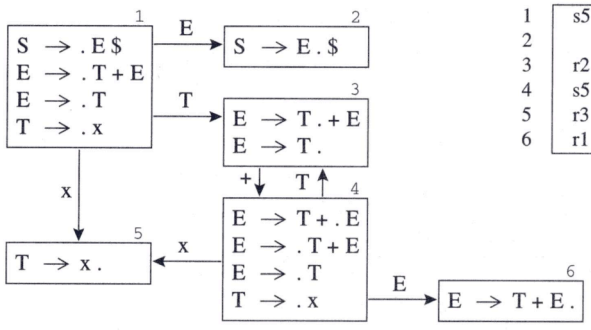


0 $S' \rightarrow S\ \$$
1 $S \rightarrow (\ L\ )$
3 $S \rightarrow x$
3 $L \rightarrow S$
4 $L \rightarrow L\ ,\ S$

Let's try that again

shift-reduce parsing conflict

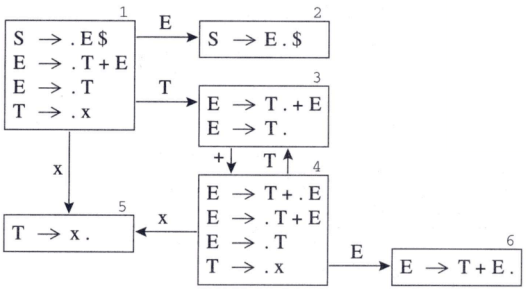
	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		



0 $S \rightarrow E\ \$$
1 $E \rightarrow T + E$
2 $E \rightarrow T$
3 $T \rightarrow x$

SLR parsing

```
R ← {}
for each state I in T
  for each item A → α . in I
    for each token X in FOLLOW(A)
      R ← R ∪ { (I, X, A → α) }
```

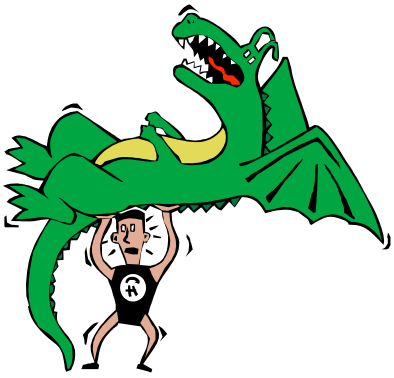


	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

0 S → E $
1 E → T + E
2 E → T
3 T → x

LR(1) parsing

- Even more powerful than SLR is the LR(1) parsing algorithm.
- Most programming languages whose syntax is describable by a context-free grammar have an LR(1) grammar.

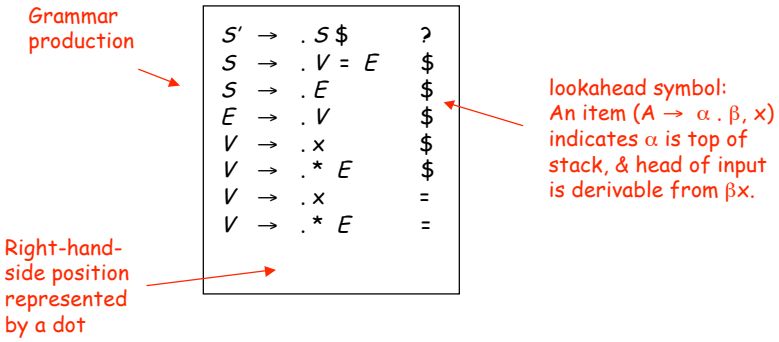


C pointer-dereference operator

$$
\begin{aligned}
S' &\Rightarrow \underline{S}\ \$ \\
&\Rightarrow V = \underline{E}\ \$ \\
&\Rightarrow V = \underline{V}\ \$ \\
&\Rightarrow V = * \underline{E}\ \$ \\
&\Rightarrow V = * \underline{V}\ \$ \\
&\Rightarrow \underline{V} = * x\ \$ \\
&\Rightarrow x = * x\ \$
\end{aligned}
$$

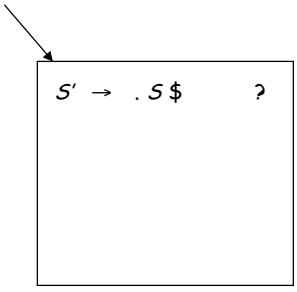
0	$S' \rightarrow S\ \$$	3	$E \rightarrow V$
1	$S \rightarrow V = E$	4	$V \rightarrow x$
2	$S \rightarrow E$	5	$V \rightarrow * E$

An LR(1) item consists of …



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

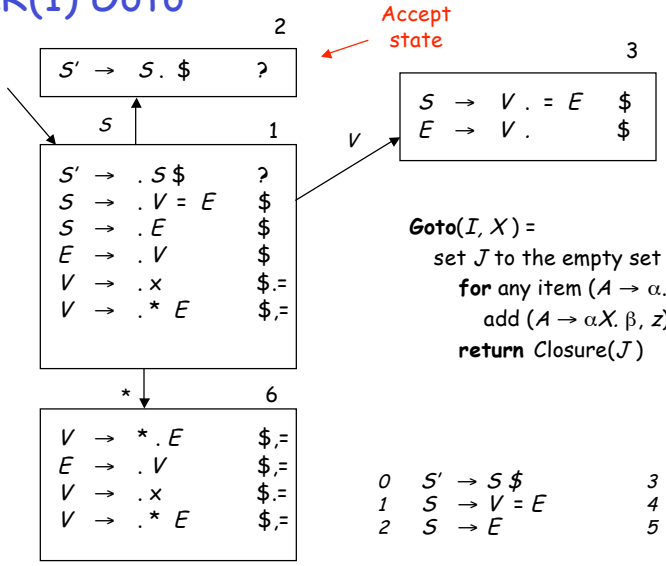
LR(1) Closure



Closure(I) =
  repeat
    for any item ($A \rightarrow \alpha . X \beta$, z) in I
      for any production $X \rightarrow \gamma$
        for any $w \in$ FIRST(β z)
          $I \leftarrow I \cup \{ (X \rightarrow . \gamma, w) \}$
  until I does not change
  return I

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

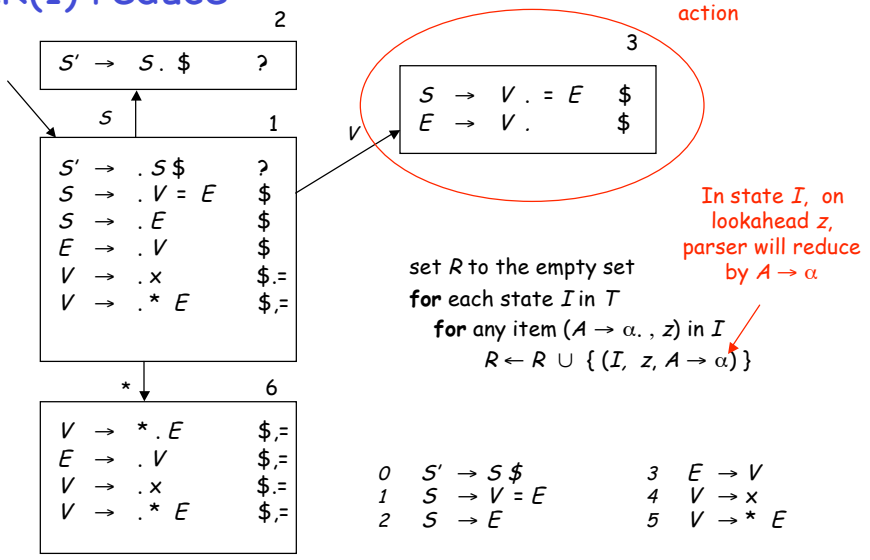
LR(1) Goto



Goto(I, X) =
  set J to the empty set
    for any item ($A \rightarrow \alpha . X \beta$, z) in I
      add ($A \rightarrow \alpha X . \beta$, z) to J
    return Closure(J)

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

LR(1) reduce



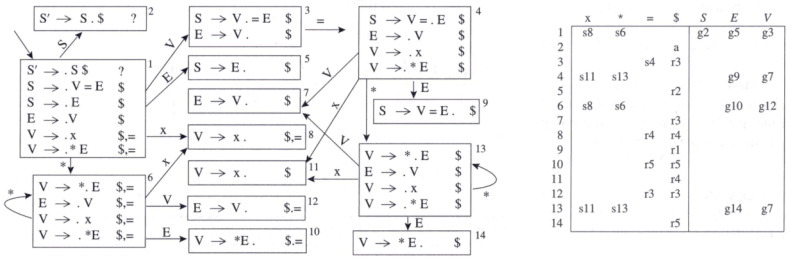
Yields reduce action

In state *I*, on lookahead *z*, parser will reduce by $A \rightarrow \alpha$

set *R* to the empty set
for each state *I* in *T*
  for any item $(A \rightarrow \alpha\,.\,, z)$ in *I*
    $R \leftarrow R \cup \{ (I, z, A \rightarrow \alpha) \}$

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

LR(1) states and parsing table



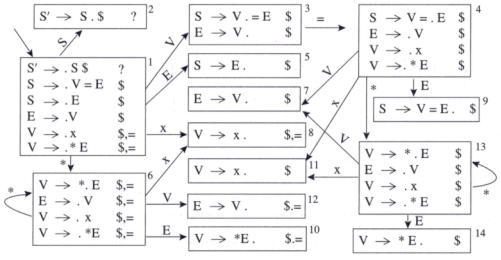
	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

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

LR(1) parsing tables can be very large

- A smaller table can be made by merging any two sates whose items are identical except for the lookahead sets.
- The resulting parser is called an LALR(1) parser for lookahead LR(1).

LALR(1) parsing table



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

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

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

A hierarchy of grammar classes