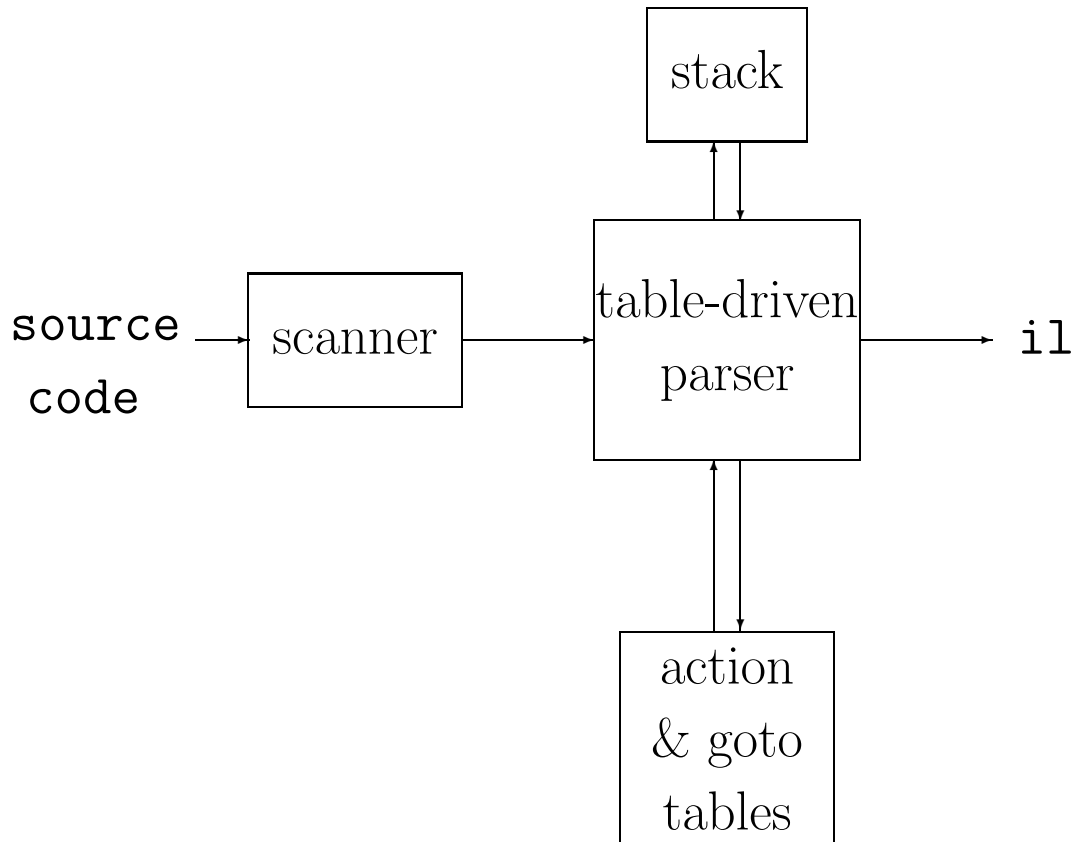


## Review: Table-driven LR parsing

---

A table-driven LR(k) parser looks like



Stack two items per state: *state* and *symbol*

## Review: $LR(1)$ parsing

---

The skeleton parser:

```
token = next_token()
repeat forever
  s = state on top of stack
  if action[s,token] = "shift  $s_i$ " then
    push token
    push state  $s_i$ 
    token = next_token()
  else if action[s,token] =
    "reduce  $A ::= \beta$ " then
    pop  $2 * |\beta|$  symbols
    s = state on top of stack
    push  $A$ 
    push goto[s, $A$ ]
  else if action[s, token] = "accept" then
    return
  else error()
```

This takes  $k$  shifts,  $l$  reduces, and 1 accept, where  $k$  is the length of the input string and  $l$  is the length of the reverse rightmost derivation.

**Note:** See Figure 4.36, p.251. Aho, Lam, Sethi, and Ullman

## LR(0) parsing: our example

Stack (without states)	Input	Handle	Action
\$	id - num * id	<i>none</i>	shift
\$id	- num * id	9,1	reduce 9
\$⟨factor⟩	- num * id	7,1	reduce 7
\$⟨term⟩	- num * id	4,1	reduce 4
\$⟨expr⟩	- num * id	<i>none</i>	shift
\$⟨expr⟩ -	num * id	<i>none</i>	shift
\$⟨expr⟩ - num	* id	8,3	reduce 8
\$⟨expr⟩ - ⟨factor⟩	* id	7,3	reduce 7
\$⟨ <b>expr</b> ⟩ - ⟨ <b>term</b> ⟩	* id	<i>none</i>	shift
\$⟨expr⟩ - ⟨term⟩ *	id	<i>none</i>	shift
\$⟨expr⟩ - ⟨term⟩ * id		9,5	reduce 9
\$⟨expr⟩ - ⟨term⟩ * ⟨factor⟩		5,5	reduce 5
\$⟨ <b>expr</b> ⟩ - ⟨ <b>term</b> ⟩		3,3	reduce 3
\$⟨expr⟩		1,1	reduce 1
\$⟨goal⟩		<i>none</i>	accept

The corresponding grammar and language is not LR(0).

*Theorem:* A language L has an LR(0) grammar iff

- L is deterministic
- no proper prefix of a word in L is in L (*prefix property*)

## LR parsing

There are three commonly used algorithms to build tables for an “LR” parser:

1.  $SLR(1)$  =  $LR(0)$  + FOLLOW

- smallest class of grammars
- smallest tables (number of states)
- simple, fast construction

2.  $LR(1)$

- full set of  $LR(1)$  grammars
- largest tables (number of states)
- slow, large construction

3.  $LALR(1)$

- intermediate sized set of grammars
- same number of states as  $SLR(1)$
- canonical construction is slow and large
- better construction techniques exist

An  $LR(1)$  parser for either ALGOL or PASCAL has several thousand states, while an  $SLR(1)$  or  $LALR(1)$  parser for the same language may have several hundred states

## SLR(1) parsing

---

*Viable prefix* of a right-sentential form:

- contains both terminals and nonterminals
- can be recognized with DFA

Building a *SLR* parser

- construct DFA for recognizing viable prefixes
- augment with FOLLOW to disambiguate actions

States in the DFA are sets of *LR(0)* items (subset construction)

**Note:** An “augmented/extended grammar” (ECFG) is one where the start symbol appears only on the *lhs* of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production  $S' ::= S$

## LR(0) items

An  $LR(0)$  item is a string  $[\alpha]$ , where

$\alpha$  is a production from  $G$  with a  $\bullet$  at some position in the *rhs*

The  $\bullet$  indicates how much of an item we have seen at a given state in the parsing process.

$[A ::= \bullet XYZ]$  indicates that the parser is looking for a string that can be derived from  $XYZ$

$[A ::= XY \bullet Z]$  indicates that the parser has seen a string derived from  $XY$  and is looking for one derivable from  $Z$

$LR(0)$  Items (*no lookahead*)

$A ::= XYZ$  generates 4  $LR(0)$  items.

1.  $[A ::= \bullet XYZ]$
2.  $[A ::= X \bullet YZ]$
3.  $[A ::= XY \bullet Z]$
4.  $[A ::= XYZ \bullet]$

## Canonical LR(0) items

The  $SLR(1)$  table construction algorithm uses a specific set of sets of  $LR(0)$  items.

These sets are called the *canonical collection of sets of  $LR(0)$  items* for a grammar  $G$ .

The canonical collection corresponds to the set of states of the DFA that recognizes viable prefixes. Each state is the set of valid  $LR(0)$  items at a particular point in the parse.

The  $LR(0)$  item  $[A ::= \beta_1 \bullet \beta_2]$  is *valid* for a viable prefix  $\alpha\beta_1$  if there is a derivation  $S' \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2w$ . In general, an item will be valid for many viable prefixes.

## Canonical Collection of LR(0) items

To construct the canonical collection we need two functions:

- **closure**( $I$ )

if  $[A ::= \alpha \bullet B\beta] \in I_j$ , then, in state  $j$ , the parser might next see a string derivable from  $B\beta$

$\Rightarrow$  to form its closure, add all items of the form  $[B ::= \bullet\gamma] \in G$

- **GOTO**( $I, X$ )

If  $I$  is the set of items that are valid for some viable prefix  $\gamma$ , then **GOTO**( $I, X$ ) is the set of items that are valid for the viable prefix  $\gamma X$ .

## SLR(1) parser example

---

### The Grammar

1		E	::=	T + E
2				T
3		T	::=	id

### The Augmented Grammar

0		S'	::=	E
1		E	::=	T + E
2				T
3		T	::=	id

Symbol	FIRST	FOLLOW
S'	{ id }	{ eof }
E	{ id }	{ eof }
T	{ id }	{ +, eof }

## Closure( $I$ )

Given an item  $[A ::= \alpha \bullet B\beta]$ , its closure contains the item and any other items that can generate legal substrings to follow  $\alpha$ .

Thus, if the parser has viable prefix  $\alpha$  on its stack, the input should reduce to  $B\beta$  (or  $\gamma$  for some other item  $[B ::= \bullet\gamma]$  in the closure).

To compute  $\text{closure}(I)$

```
function closure(I)
  repeat
    new_item ← false
    for each item  $[A ::= \alpha \bullet B\beta] \in I$ ,
      each production  $B ::= \gamma \in G'$ 
        if  $[B ::= \bullet\gamma] \notin I$  then
          add  $[B ::= \bullet\gamma]$  to I
          new_item ← true
        endif
  until (new_item = false)
  return I
```

## Goto( $I, X$ )

Let  $I$  be a set of  $LR(0)$  items and  $X$  be a grammar symbol.

Then,  $\text{GOTO}(I, X)$  is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta] \text{ such that } [A ::= \alpha \bullet X \beta] \in I$$

If  $I$  is the set of valid items for some viable prefix  $\gamma$ , then  $\text{goto}(I, X)$  is the set of valid items for the viable prefix  $\gamma X$ .

$\text{goto}(I, X)$  represents state after recognizing  $X$  in state  $I$ .

To compute  $\text{goto}(I, X)$

```
function goto(I, X)
  J ← set of items [A ::= αX • β]
    such that [A ::= α • Xβ] ∈ I
  J' ← closure(J)
  return J'
```

## Collection of sets of LR(0) items

We start the construction of the collection of sets of LR(0) items with the item  $[S' ::= \bullet S]$ , where

$S'$  is the start symbol of the augmented grammar  $G'$   
 $S$  is the start symbol of  $G$

To compute the collection of sets of LR(0) items

```
procedure items( $G'$ )
   $S_0 \leftarrow \text{closure}(\{[S' ::= \bullet S]\})$ 
  Items  $\leftarrow \{ S_0 \}$ 
  ToDo  $\leftarrow \{ S_0 \}$ 
  while ToDo not empty do
    remove  $S_i$  from ToDo
    for each grammar symbol  $X$  do
       $S_{new} \leftarrow \text{goto}(S_i, X)$ 
      if  $S_{new}$  is a new state then
        Items  $\leftarrow \text{Items} \cup \{ S_{new} \}$ 
        ToDo  $\leftarrow \text{ToDo} \cup \{ S_{new} \}$ 
      endif
    endfor
  endwhile
  return Items
```

## LR(0) machines

### LR(0) DFA

- states – canonical sets of LR(0) items
- edges – goto transitions
- recognizes all viable prefixes
- no lookahead

Reducing a handle (rhs of production) to a nonterminal can be viewed as:

- returning to state at beginning of handle
- making transition on nonterminal for this state

To return to state at beginning of the handle, we must use the stack to store the state!

## SLR(1) tables

---

### SLR(1) parser

- augment  $LR(0)$  machine
- add FOLLOW information using one token of lookahead
- encoded as **ACTION**, **GOTO** tables

### ACTION table

- for each [state, lookahead] pair
- have we reached end of handle?
- if not, shift
- if at end of handle, reduce
- may also accept or error
- use lookahead to guide decision

### GOTO table

- for each [state, nonterminal] pair
- pick state to go to after reduction

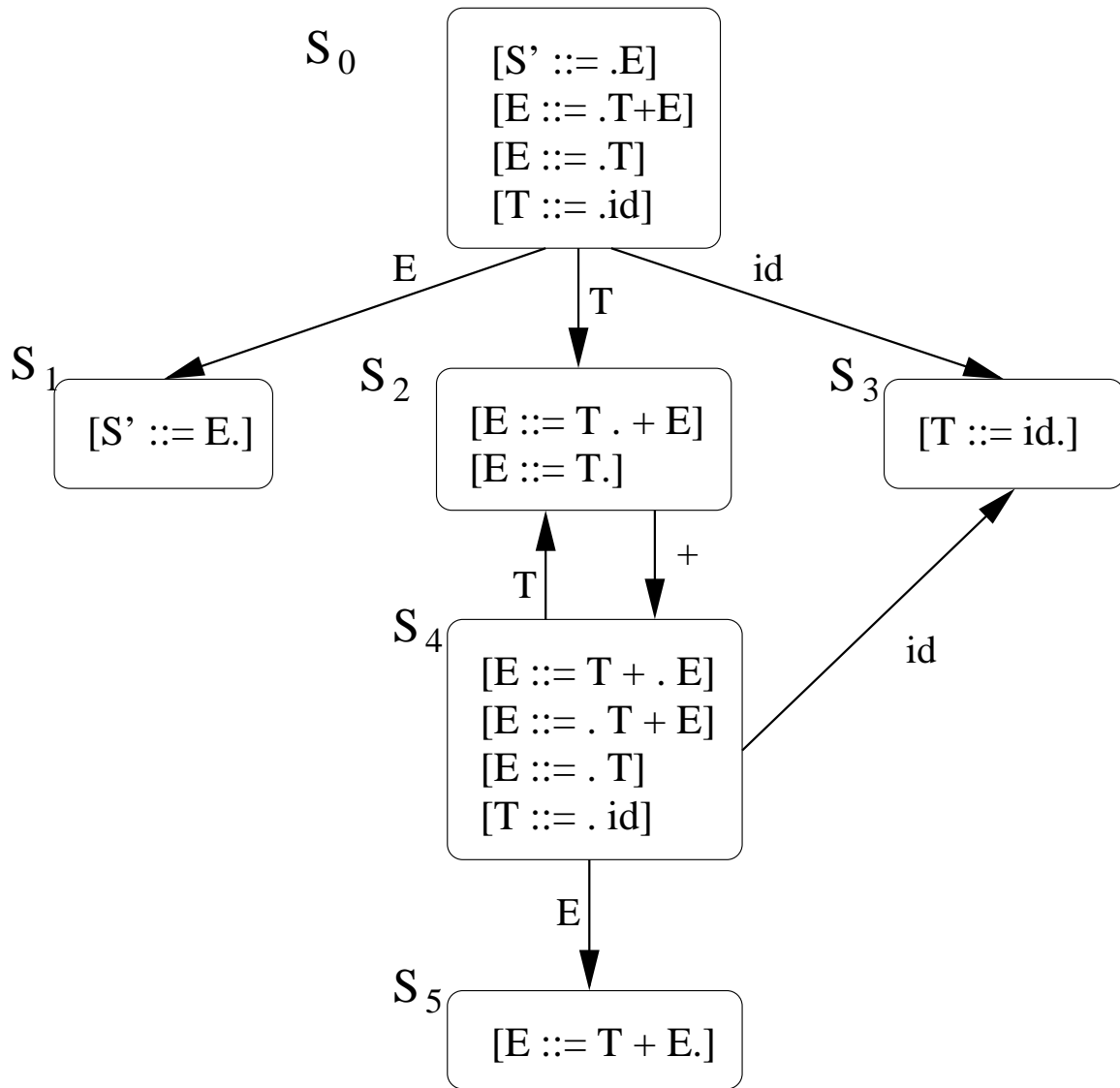
## SLR(1) table construction

### The Algorithm

1. construct the collection of sets of  $LR(0)$  items for  $G'$ .
2. State  $i$  of the parser is constructed from  $I_i$ .
  - (a) if  $[A ::= \alpha \bullet a\beta] \in I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set **ACTION** $[i, a]$  to “*shift j*”. ( $a$  must be a terminal)
  - (b) if  $[A ::= \alpha \bullet] \in I_i$ , then set **ACTION** $[i, a]$  to “*reduce A ::=  $\alpha$* ” for all  $a$  in  $\text{FOLLOW}(A)$ .
  - (c) if  $[S' ::= S \bullet] \in I_i$ , then set **ACTION** $[i, \text{eof}]$  to “*accept*”.
3. If  $\text{goto}(I_i, A) = I_j$ , then set **GOTO** $[i, A]$  to  $j$ .
4. All other entries in **ACTION** and **GOTO** are set to “*error*”
5. The initial state of the parser is the state constructed from the set containing the item  $[S' ::= \bullet S]$ .

# SLR(1) parser example

---



---

DFA for viable prefixes based on LR(0) canonical collection

## Example LR(0) states

---

$S_0$ : [  $S' ::= \bullet E$  ],  
[  $E ::= \bullet T + E$  ],  
[  $E ::= \bullet T$  ],  
[  $T ::= \bullet \text{id}$  ]

$S_1$ : [  $S' ::= E \bullet$  ]

$S_2$ : [  $E ::= T \bullet + E$  ],  
[  $E ::= T \bullet$  ]

$S_3$ : [  $T ::= \text{id} \bullet$  ]

$S_4$ : [  $E ::= T + \bullet E$  ],  
[  $E ::= \bullet T + E$  ],  
[  $E ::= \bullet T$  ],  
[  $T ::= \bullet \text{id}$  ]

$S_5$ : [  $E ::= T + E \bullet$  ]

## Example GOTO function

---

*Start*

$$S_0 \leftarrow \text{closure} ( \{ [ S ::= \bullet E ] \} )$$

*Iteration 1*

$$\text{goto}(S_0, E) = S_1$$

$$\text{goto}(S_0, T) = S_2$$

$$\text{goto}(S_0, \text{id}) = S_3$$

*Iteration 2*

$$\text{goto}(S_2, +) = S_4$$

*Iteration 3*

$$\text{goto}(S_4, \text{id}) = S_3$$

$$\text{goto}(S_4, E) = S_5$$

$$\text{goto}(S_4, T) = S_2$$

## Example ACTION and GOTO tables

---

	ACTION			GOTO	
	id	+	eof	E	T
$S_0$	shift 3	—	—	1	2
$S_1$	—	—	accept	—	—
$S_2$	—	shift 4	reduce 2	—	—
$S_3$	—	reduce 3	reduce 3	—	—
$S_4$	shift 3	—	—	5	2
$S_5$	—	—	reduce 1	—	—

Stack	Input	Action
\$ 0	id + id eof	shift 3
\$ 0 id 3	+ id eof	reduce 3 (T ::= id)
\$ 0 T 2	+ id eof	shift 4
\$ 0 T 2 + 4	id eof	shift 3
\$ 0 T 2 + 4 id 3	eof	reduce 3 (T ::= id)
\$ 0 T 2 + 4 T 2	eof	reduce 2 (E ::= T)
\$ 0 T 2 + 4 E 5	eof	reduce 1 (E ::= T + E)
\$ 0 E 1	eof	accept

## What can go wrong?

---

Example: A simple grammar

- |                  |                      |
|------------------|----------------------|
| 1. $S' ::= S$    | 4. $L ::= *R$        |
| 2. $S ::= L = R$ | 5. $L ::= \text{id}$ |
| 3. $S ::= R$     | 6. $R ::= L$         |

*Canonical LR(0) collection*

---

$$I_0 : \{ [S' ::= \bullet S], [S ::= \bullet L = R], [S ::= \bullet R], \\ [L ::= \bullet * R], [L ::= \bullet \text{id}], [R ::= \bullet L] \}$$

$$I_1 : \{ [S' ::= S \bullet] \}$$

$$I_2 : \{ [S ::= L \bullet = R], [R ::= L \bullet] \}$$

$$I_3 : \{ [S ::= R \bullet] \}$$

$$I_4 : \{ [L ::= * \bullet R], [R ::= \bullet L], [L ::= \bullet * R], \\ [L ::= \bullet \text{id}] \}$$

$$I_5 : \{ [L ::= \text{id} \bullet] \}$$

$$I_6 : \{ [S ::= L = \bullet R], [R ::= \bullet L], [L ::= \bullet * R], \\ [L ::= \bullet \text{id}] \}$$

$$I_7 : \{ [L ::= *R \bullet] \}$$

$$I_8 : \{ [R ::= L \bullet] \}$$

$$I_9 : \{ [S ::= L = R \bullet] \}$$

## SLR(1) table construction

Symbol	FIRST	FOLLOW
$S'$	{ id, * }	{ eof }
$S$	{ id, * }	{ eof }
$L$	{ id, * }	{ =, eof }
$R$	{ id, * }	{ =, eof }

Consider the set of items  $I_2$ . The action table is defined as follows:

$[S ::= L\bullet = R]$  implies ACTION[2, =] = "shift 6"

$[R ::= L\bullet]$  implies ACTION[2, =] = "reduce 6"

Due to multiple definitions of the position in the action table, the grammar is not  $SLR(1)$ .

## What can go wrong?

---

Two cases arise

*shift/reduce*

This is called a *shift/reduce* conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

**classic example:** dangling else

*reduce/reduce*

This is called a *reduce/reduce* conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

**classic example:** PL/I call and subscript

## LR(1)

---

$SLR(1)$  parsers may not be able to parse some  $LR$  grammars.

Problem is that lookahead information is added to  $LR(0)$  parser at the end of construction.

We can get more powerful parser by keeping track of lookahead information in the states of the parser.

*If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is  $LR(1)$*

## $LR(k)$ items

The table construction algorithms use  $LR(k)$  items to represent the set of possible states in a parse

An  $LR(k)$  item is a pair  $[\alpha, \beta]$ , where

$\alpha$  is a production from  $G$  with a  $\bullet$  at some position in the *rhs*

$\beta$  is a lookahead string containing  $k$  symbols (terminals or **eof**)

What about  $LR(1)$  items?

- example  $LR(1)$  item:  $[A ::= X \bullet YZ, a]$
- $LR(1)$  items have lookahead strings of length 1
- several  $LR(1)$  items may have the same **core**

$$[A ::= X \bullet YZ, a]$$

$$[A ::= X \bullet YZ, b]$$

we represent this as

$$[A ::= X \bullet YZ, \{a, b\}]$$

## LR(1) lookahead

*What's the point of all these lookahead symbols?*

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping unless item has • at right end.
  - in  $[A ::= X \bullet YZ, \mathbf{a}]$ ,  $\mathbf{a}$  has no direct use
  - in  $[A ::= XYZ\bullet, \mathbf{a}]$ ,  $\mathbf{a}$  is useful

Recall, the *SLR(1)* construction uses *LR(0)* items!

*The point*

For  $[A ::= \alpha\bullet, \mathbf{a}]$  and  $[B ::= \alpha\bullet, \mathbf{b}]$ , we can decide between reducing to  $A$  and to  $B$  by looking at limited right context!

## Canonical $LR(1)$ items

---

The canonical collection of sets of  $LR(1)$  items:

- sets of valid items for viable prefixes of the grammar
- sets of items derivable from  $[S' ::= \bullet S, \text{eof}]$  using **goto** and **closure** functions — both functions preserve validity.

A  $LR(1)$  item  $[A ::= \alpha \bullet \beta, a]$  is *valid* for a viable prefix  $\gamma$  if there is a derivation  $S \Rightarrow_{rm}^* \delta A w \Rightarrow_{rm} \delta \alpha \beta w$ , where

- $\gamma = \delta \alpha$ , and
- either  $a$  is the first symbol of  $w$ , or  $w$  is  $\epsilon$  and  $a$  is **eof**.

Essentially,

- Each  $LR(1)$  item in a set in the canonical collection represents a state in an NFA that recognizes viable prefixes.
- Grouping these items together is really the DFA subset construction.

## LR(1) closure

Given an item  $[A ::= \alpha \bullet B\beta, \mathbf{a}]$ , its closure contains the item and any other items that can generate legal substrings to follow  $\alpha$ .

Thus, if the parser has viable prefix  $\alpha$  on its stack, a substring of the input should reduce to  $B\beta$  (or  $\gamma$  for some other item  $[B ::= \bullet\gamma, \mathbf{b}]$  in the closure).

To compute  $\text{closure}(I)$

```
function closure(I)
  repeat
    new_item ← false
    for each item  $[A ::= \alpha \bullet B\beta, \mathbf{a}] \in I$ ,
      each production  $B ::= \gamma \in G'$ ,
      and each terminal  $\mathbf{b} \in \text{FIRST}(\beta\mathbf{a})$ ,
      if  $[B ::= \bullet\gamma, \mathbf{b}] \notin I$  then
        add  $[B ::= \bullet\gamma, \mathbf{b}]$  to I
        new_item ← true
      endif
  until (new_item = false)
  return I
```

Aho, Sethi, and Ullman, Figure 4.38

## LR(1) goto

---

Let  $I$  be a set of  $LR(1)$  items and  $X$  be a grammar symbol.

Then,  $\text{goto}(I, X)$  is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta, \mathbf{a}] \text{ such that } [A ::= \alpha \bullet X \beta, \mathbf{a}] \in I$$

If  $I$  is the set of valid items for some viable prefix  $\gamma$ , then  $\text{goto}(I, X)$  is the set of valid items for the viable prefix  $\gamma X$ .

$\text{goto}(I, X)$  represents state after recognizing  $X$  in state  $I$ .

To compute  $\text{goto}(I, X)$

```
function goto(I, X)
  J ← set of items [A ::= αX • β, a]
    such that [A ::= α • Xβ, a] ∈ I
  J' ← closure(J)
  return J'
```

Aho, Sethi, and Ullman, Figure 4.38

## Collection of sets of LR(1) items

We start the construction of the canonical collection of LR(1) items with the item  $[S' ::= \bullet S, \text{eof}]$ , where

- $S'$  is the start symbol of the augmented grammar  $G'$
- $S$  is the start symbol of  $G$ , and
- $\text{eof}$  is the right end of string marker

To compute the collection of sets of LR(1) items

```
procedure items( $G'$ )
   $C \leftarrow \{\text{closure}(\{[S' ::= \bullet S, \text{eof}]\})\}$ 
  repeat
    new_item  $\leftarrow$  false
    for each set of items  $I$  in  $C$  and
      each grammar symbol  $X$  such that
        goto( $I, X$ )  $\neq \emptyset$  and
        goto( $I, X$ )  $\notin C$ 
          add goto( $I, X$ ) to  $C$ 
          new_item  $\leftarrow$  true
    endfor
  until (new_item = false)
```

Aho, Sethi, and Ullman, Figure 4.38

## LR(1) table construction

### The Algorithm

1. construct the collection of sets of  $LR(1)$  items for  $G'$ .
2. State  $i$  of the parser is constructed from  $I_i$ .
  - (a) if  $[A ::= \alpha \bullet a\beta, \mathbf{b}] \in I_i$  and  $\text{goto}(I_i, \mathbf{a}) = I_j$ , then set  $\mathbf{ACTION}[i, \mathbf{a}]$  to “*shift j*”. ( $\mathbf{a}$  must be a terminal)
  - (b) if  $[A ::= \alpha \bullet, \mathbf{a}] \in I_i$ , then set  $\mathbf{ACTION}[i, \mathbf{a}]$  to “*reduce A ::=  $\alpha$* ”.
  - (c) if  $[S' ::= S \bullet, \mathbf{eof}] \in I_i$ , then set  $\mathbf{ACTION}[i, \mathbf{eof}]$  to “*accept*”.
3. If  $\text{goto}(I_i, A) = I_j$ , then set  $\mathbf{GOTO}[i, A]$  to  $j$ .
4. All other entries in  $\mathbf{ACTION}$  and  $\mathbf{GOTO}$  are set to “*error*”
5. The initial state of the parser is the state constructed from the set containing the item  $[S' ::= \bullet S, \mathbf{eof}]$ .

Aho, Sethi, and Ullman, Algorithm 4.10

## *Example*

---

Our simple grammar

1.  $S' ::= S$
2.  $S ::= L = R$
3.  $S ::= R$
4.  $L ::= *R$
5.  $L ::= \text{id}$
6.  $R ::= L$

## Canonical LR(1) collection

---

$$\begin{aligned} I_0 &: \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}], \\ &\quad [S ::= \bullet R, \text{eof}], [L ::= \bullet * R, \{=, \text{eof}\}], \\ &\quad [L ::= \bullet \text{id}, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \} \\ I_1 &: \{ [S' ::= S\bullet, \text{eof}] \} \\ I_2 &: \{ [S ::= L\bullet = R, \text{eof}], [R ::= L\bullet, \text{eof}] \} \\ I_3 &: \{ [S ::= R\bullet, \text{eof}] \} \\ I_4 &: \{ [L ::= * \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}], \\ &\quad [L ::= \bullet * R, \{=, \text{eof}\}], [L ::= \bullet \text{id}, \{=, \text{eof}\}] \} \\ I_5 &: \{ [L ::= \text{id}\bullet, \{=, \text{eof}\}] \} \\ I_6 &: \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}], \\ &\quad [L ::= \bullet * R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \} \\ I_7 &: \{ [L ::= *R\bullet, \{=, \text{eof}\}] \} \\ I_8 &: \{ [R ::= L\bullet, \{=, \text{eof}\}] \} \\ I_9 &: \{ [S ::= L = R\bullet, \text{eof}] \} \\ I_{10} &: \{ [R ::= L\bullet, \text{eof}] \} \\ I_{11} &: \{ [L ::= * \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}], \\ &\quad [L ::= \bullet * R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \} \\ I_{12} &: \{ [L ::= \text{id}\bullet, \text{eof}] \} \\ I_{13} &: \{ [L ::= *R\bullet, \text{eof}] \} \end{aligned}$$

## LR(1) table construction

*So, does it work now?*

Example: Consider the set of items  $I_2$ . The action table is defined as follows:

$$[S ::= L\bullet = R, \text{eof}] \Rightarrow \text{ACTION}[2, =] = \text{"shift 6"}$$
$$[R ::= L\bullet, \text{eof}] \Rightarrow \text{ACTION}[2, \text{eof}] = \text{"reduce 6"}$$

**Yes**, the table construction defines only single entries for each position in the **ACTION** table.

The price we pay: more states

- construction of table takes longer
- table is larger

Here is an idea: *LALR*(1) parsing: Union states that are sets of *LR*(1) items with the same *core*.

## Summary: Resolving parse conflicts

---

Parse conflicts possible when certain *LR* items are found in the same state.

Depending on parser, may choose between *LR* items using lookahead.

Legal lookahead for *LR* items must be disjoint, else conflict exists.

	Shift-Reduce $[ A ::= \alpha \bullet , \Delta ]$ $[ B ::= \beta \bullet \gamma , \Omega ]$	Reduce-Reduce $[ A ::= \alpha \bullet , \Delta ]$ $[ B ::= \beta \bullet , \Omega ]$
<i>LR</i> (0)	conflict	conflict
<i>SLR</i> (1)	$FOLLOW(A)$ $\cap$ $FIRST(\gamma)$	$FOLLOW(A)$ $\cap$ $FOLLOW(B)$
<i>LR</i> (1)	$\Delta \cap FIRST(\gamma)$	$\Delta \cap \Omega$

# Lex and Yacc

---

Scanner is defined in `scan.l`

How to specify and use attributes in YACC?

- Define attributes as types in file `attr.h`:

```
typedef union {int num; char *str;} tokentype;
```

```
typedef struct {int a; int b} infonode;
```

- Include type attribute name in `%union` in `parse.y`

```
%union {tokentype token; infonode myinfo; ... }
```

- Assign attributes in `parse.y` to

- Terminals:

```
%token <token> ID CONST
```

- Non-Terminals:

```
%type <myinfo> block variables procdecls cmpdstmt
```

- Accessing attribute values in `parse.y`

- use `$$`, `$1`, `$2`, etc. notation, counting from left to right; **embedded actions** need to be counted (implicit token)!

```
block : variables {$1.a=NewValue(); } procdecls
```

```
    cmpdstmt  {$$.a =$1.a+$3.a + $4.b;}
```

# Lex and Yacc ( see lex&yacc, Levine et al., ALSU 4.9, and man pages)

---

Example: A simple language and its (partial) interpreter. See [~uli/cs515/examples/interpreter](http://uli/cs515/examples/interpreter) on **ilab** for a similar, complete example.

```
%token PROG PERIOD WRITELN BEG END ASG DIV EXP
%token <token> ID NUM

%type <value> exp constant
%start program
%left '+' '-'
%left '*' DIV

%%
program : PROG ID ';' block PERIOD {addname($2.str);}
        ;
block   : BEG stmtlist END
        ;
stmtlist : stmtlist ';' stmt
         | stmt
         ;
stmt    :  astmt
         | writestmt
         ;
writestmt: WRITELN '(' exp ')'      {printf(" ANSWER >> %d\n", $3);}
        ;
astmt   : ID ASG exp                {addname($1.str); setvalue($1.str, $3);}
        ;
exp     : exp '+' exp               {addop("+"); $$ = $1 + $3;}
         | exp '-' exp              {addop("-"); $$ = $1 - $3;}
         | exp '*' exp              {addop("*"); $$ = $1 * $3;}
         | exp DIV exp              {addop("div"); $$ = $1 / $3;}
         | '(' exp ')'              {$$ = $2;}
         | ID                       {$$ = getvalue($1.str);}
         | constant                 {$$ = $1;}
        ;
constant: NUM                       {$$ = $1.num;}
        ;
%%
```

# Error Recovery in Lex and Yacc

## The problem

encounter an invalid token

We want to *parse* the rest of the file

## Basic idea (panic mode):

- Assume something went wrong while trying to find handle for nonterminal  $A$
- Pretend handle for  $A$  has been found; pop “handle”, skip over input to find terminal that can follow  $A$

Restarting the parser (panic mode) — see ALSU  
pp.295-297

- find a restartable state on the stack (has transition for nonterminal  $A$ )
- move to a consistent place in the input (token that can follow  $A$ )
- perform (error) reduction (for nonterminal  $A$ )
- print an informative message *(line number)*

## Error recovery in *yacc*

*Yacc*'s error mechanism (note: version dependent!)

- designated token **error**
- used in *error production* of the form  $A ::= \text{error } \alpha$
- $\alpha$  specifies synchronization points

When error is discovered

- pops stack until it finds state where it can *shift* the **error** token
- resumes parsing to match  $\alpha$   
special cases:
  - $\alpha = w$ : skip input until  $w$  has been read
  - $\alpha = \epsilon$ : skip input until state transition on input token is defined
- error productions can have actions

*This mechanism is fairly general.*

See *lex & yacc*, Levine, Mason, and Brown, pp. 188, 247–252

## Error recovery in *yacc*

---

```
stmt_list : stmt
          | stmt_list ; stmt
```

*can be augmented with error*

```
stmt_list : stmt
          | error
          | stmt_list ; stmt
```

*this should*

- throw out the erroneous statement
- synchronize at “;” or “end” (implicit)
- invoke `yyerror("syntax error")` (implicit); user may want to insert explicit call, e.g.,  
`error {yyerror("illegal statement")}`

Other “natural” places for error productions

- all the “lists” (high level structure rules)
- missing parentheses or brackets
- extra operator or missing operator