

# Bottom-up parsing

---

## *Recall*

For a grammar  $G$ , with start symbol  $S$ , any string  $\alpha$  such that  $S \Rightarrow^* \alpha$  is called a *sentential form*

- If  $\alpha \in V_t^*$ , then  $\alpha$  is called a *sentence* in  $L(G)$
- Otherwise it is just a sentential form (not a sentence in  $L(G)$ )

A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

# Bottom-up parsing

---

Goal:

*Given an input string  $w$  and a grammar  $G$ , construct a parse tree by starting at the leaves and working to the root.*

The parser repeatedly matches a *right-sentential* form from the language against the tree's upper frontier.

At each match, it applies a *reduction* to build on the frontier:

- each reduction matches an upper frontier of the partially built tree to the RHS of some production
- each reduction adds a node on top of the frontier

The final result is a rightmost derivation, in reverse.

## Example

---

Consider the grammar

$$\begin{array}{l|l}
 1 & S \rightarrow aABe \\
 2 & A \rightarrow Abc \\
 3 & \quad | \quad b \\
 4 & B \rightarrow d
 \end{array}$$

and the input string abcde

Prod'n.	Sentential Form
3	a <span style="border: 1px solid black; padding: 2px;">b</span> bcde
2	a <span style="border: 1px solid black; padding: 2px;">Abc</span> de
4	aA <span style="border: 1px solid black; padding: 2px;">d</span> e
1	<span style="border: 1px solid black; padding: 2px;">aABe</span>
–	<i>S</i>

The trick appears to be scanning the input and finding valid sentential forms.

# Handles

---

*What are we trying to find?*

A substring  $\alpha$  of the tree's upper frontier that:

matches some production  $A \rightarrow \alpha$  where reducing  $\alpha$  to  $A$  is one step in the reverse of a rightmost derivation

We call such a string a *handle*.

Formally:

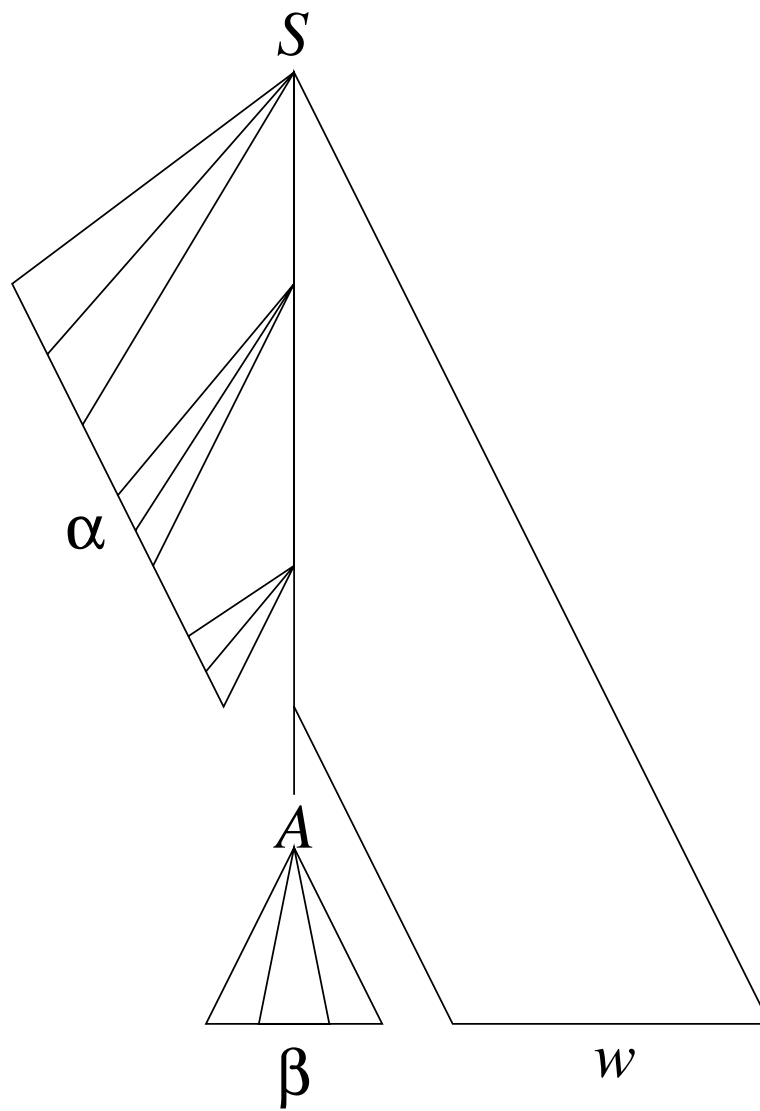
a *handle* of a right-sentential form  $\gamma$  is a production  $A \rightarrow \beta$  and a position in  $\gamma$  where  $\beta$  may be found and replaced by  $A$  to produce the previous right-sentential form in a rightmost derivation of  $\gamma$

i.e., if  $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha \beta w$  then  $A \rightarrow \beta$  in the position following  $\alpha$  is a handle of  $\alpha \beta w$

Because  $\gamma$  is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

# Handles

---



The handle  $A \rightarrow \beta$  in the parse tree for  $\alpha\beta w$

# Handles

---

*Theorem:*

If  $G$  is unambiguous then every right-sentential form has a unique handle.

*Proof: (by definition)*

1.  $G$  is unambiguous  $\Rightarrow$  rightmost derivation is unique
2.  $\Rightarrow$  a unique production  $A \rightarrow \beta$  applied to take  $\gamma_{i-1}$  to  $\gamma_i$
3.  $\Rightarrow$  a unique position  $k$  at which  $A \rightarrow \beta$  is applied
4.  $\Rightarrow$  a unique handle  $A \rightarrow \beta$

## Example

The left-recursive expression grammar (*original form*)

1		<goal>	::=	<expr>
2		<expr>	::=	<expr> + <term>
3				<expr> - <term>
4				<term>
5		<term>	::=	<term> * <factor>
6				<term> / <factor>
7				<factor>
8		<factor>	::=	num
9				id

Prod'n.	Sentential Form
–	<goal>
1	<u>&lt;expr&gt;</u>
3	<u>&lt;expr&gt; - &lt;term&gt;</u>
5	<expr> - <u>&lt;term&gt; * &lt;factor&gt;</u>
9	<expr> - <term> * <u>id</u>
7	<expr> - <u>&lt;factor&gt;</u> * id
8	<expr> - <u>num</u> * id
4	<u>&lt;term&gt;</u> - num * id
7	<u>&lt;factor&gt;</u> - num * id
9	<u>id</u> - num * id

# Handle-pruning

---

The process to construct a bottom-up parse is called *handle-pruning*.

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n$$

we set  $i$  to  $n$  and apply the following simple algorithm

for  $i = n$  downto 1

1. find the handle  $A_i \rightarrow \beta_i$  in  $\gamma_i$
2. replace  $\beta_i$  with  $A_i$  to generate  $\gamma_{i-1}$

*This takes  $2n$  steps, where  $n$  is the length of the derivation*

# Stack implementation

---

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with \$
2. Repeat until the top of the stack is the goal symbol and the input token is \$
  - (a) *find the handle*  
if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
  - (b) *prune the handle*  
if we have a handle  $A \rightarrow \beta$  on the stack, *reduce*
    - i. pop  $|\beta|$  symbols off the stack
    - ii. push  $A$  onto the stack

## Example: back to $x - 2 * y$

1	<goal>	::=	<expr>
2	<expr>	::=	<expr> + <term>
3			<expr> - <term>
4			<term>
5	<term>	::=	<term> * <factor>
6			<term> / <factor>
7			<factor>
8	<factor>	::=	num
9			id

Stack	Input	Action
\$	id - num * id	shift
<u>\$id</u>	- num * id	reduce 9
<u>\$&lt;factor&gt;</u>	- num * id	reduce 7
<u>\$&lt;term&gt;</u>	- num * id	reduce 4
\$<expr>	- num * id	shift
\$<expr> -	num * id	shift
\$<expr> - <u>num</u>	* id	reduce 8
\$<expr> - <u>&lt;factor&gt;</u>	* id	reduce 7
\$<expr> - <term>	* id	shift
\$<expr> - <term> *	id	shift
\$<expr> - <term> * <u>id</u>		reduce 9
\$<expr> - <u>&lt;term&gt; * &lt;factor&gt;</u>		reduce 5
\$<expr> - <term>		reduce 3
<u>\$&lt;expr&gt;</u>		reduce 1
<u>\$&lt;goal&gt;</u>		accept

# Shift-reduce parsing

---

*Shift-reduce parsers are simple to understand*

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal LHS
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

Key insight: recognize handles with a DFA:

- DFA transitions shift states instead of symbols
- accepting states trigger reductions

# LR parsing

---

The skeleton parser:

```
push  $s_0$ 
token = next_token()
repeat forever
  s = top of stack
  if action[s,token] == " shift  $s_i$ " then
    push  $s_i$ 
    token = next_token()
  else if action[s,token] == " reduce  $A \rightarrow \beta$ "
    then
      pop |  $\beta$  | states
       $s'$  = top of stack
      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

## Example tables

state	ACTION				GOTO		
	id	+	*	\$	<expr>	<term>	<factor>
0	s4	-	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	s5	-	r3	-	-	-
3	-	r5	s6	r5	-	-	-
4	-	r6	r6	r6	-	-	-
5	s4	-	-	-	7	2	3
6	s4	-	-	-	-	8	3
7	-	-	-	r2	-	-	-
8	-	r4	-	r4	-	-	-

### The Grammar

1	<goal>	::=	<expr>
2	<expr>	::=	<term> + <expr>
3			<term>
4	<term>	::=	<factor> * <term>
5			<factor>
6	<factor>	::=	id

*Note:* This is a simple little right-recursive grammar. It is *not* the same grammar as in previous lectures.

## Example using the tables

---

Stack	Input	Action
\$ 0	id * id + id\$	s4
\$ 0 4	* id + id\$	r6
\$ 0 3	* id + id\$	s6
\$ 0 3 6	id + id\$	s4
\$ 0 3 6 4	+ id \$	r6
\$ 0 3 6 3	+ id \$	r5
\$ 0 3 6 8	+ id \$	r4
\$ 0 2	+ id \$	s5
\$ 0 2 5	id \$	s4
\$ 0 2 5 4	\$	r6
\$ 0 2 5 3	\$	r5
\$ 0 2 5 2	\$	r3
\$ 0 2 5 7	\$	r2
\$ 0 1	\$	acc

## Why study LR grammars?

---

LR(1) grammars are often used to construct parsers.

We call these parsers LR(1) parsers.

- used to be everyone's favourite parser (but top-down is making a comeback with JavaCC)
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

LL( $k$ ): recognize use of a production  $A \rightarrow \beta$  seeing first  $k$  symbols of  $\beta$

LR( $k$ ): recognize occurrence of  $\beta$  (the handle) having seen all of what is derived from  $\beta$  plus  $k$  symbols of lookahead

# LR parsing

---

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

1. SLR(1)
  - 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.

## LR( $k$ ) items

---

The table construction algorithms use sets of LR( $k$ ) *items* or *configurations* to represent the 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, marking how much of the RHS of a production has already been seen

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

Two cases of interest are  $k = 0$  and  $k = 1$ :

LR(0) items play a key role in the SLR(1) table construction algorithm.

LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms.

## Example

---

The  $\bullet$  indicates how much of an item we have seen at a given state in the parse:

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

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

LR(0) items: (*no lookahead*)

$A \rightarrow XYZ$  generates 4 LR(0) items:

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

# The characteristic finite state machine (CFSM)

---

The CFSM for a grammar is a DFA which recognizes *viable prefixes* of right-sentential forms:

A *viable prefix* is any prefix that does not extend beyond the handle.

It accepts when a handle has been discovered and needs to be reduced.

To construct the CFSM we need two functions:

- $\text{closure}_0(I)$  to build its states
- $\text{goto}_0(I, X)$  to determine its transitions

## closure0

---

Given an item  $[A \rightarrow \alpha \bullet B\beta]$ , 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  $[B \rightarrow \bullet\gamma]$  in the closure).

```
function closure0(I)
repeat
  if  $[A \rightarrow \alpha \bullet B\beta] \in I$ 
    add  $[B \rightarrow \bullet\gamma]$  to I
until no more items can be added to I
return I
```

## goto0

---

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

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

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  $I$ .

function goto0( $I, X$ )

  let  $J$  be the set of items  $[A \rightarrow \alpha X \bullet \beta]$   
  such that  $[A \rightarrow \alpha \bullet X \beta] \in I$   
  return closure0( $J$ )

## Building the LR(0) item sets

---

We start the construction with the item  $[S' \rightarrow \bullet S\$]$ , where

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

$S$  is the start symbol of  $G$

$\$$  represents EOF

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

```
function items( $G'$ )
   $s_0 = \text{closure}_0(\{[S' \rightarrow \bullet S\$]\})$ 
   $S = \{s_0\}$ 
  repeat
    for each set of items  $s \in S$ 
      for each grammar symbol  $X$ 
        if  $\text{goto}_0(s, X) \neq \emptyset$  and  $\text{goto}_0(s, X) \notin S$ 
          add  $\text{goto}_0(s, X)$  to  $S$ 
  until no more item sets can be added to  $S$ 
  return  $S$ 
```

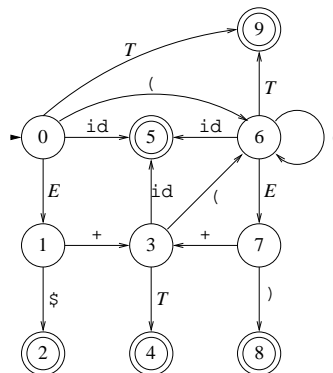
# LR(0) example

---

1	$S \rightarrow E\$$
2	$E \rightarrow E + T$
3	$\quad \quad \quad   \quad T$
4	$T \rightarrow \text{id}$
5	$\quad \quad \quad   \quad (E)$

$I_0 : S \rightarrow \bullet E\$$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet \text{id}$ $T \rightarrow \bullet (E)$	$I_4 : E \rightarrow E + T \bullet$ $I_5 : T \rightarrow \text{id} \bullet$ $I_6 : T \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$
$I_1 : S \rightarrow E \bullet \$$ $E \rightarrow E \bullet + T$	$T \rightarrow \bullet \text{id}$ $T \rightarrow \bullet (E)$
$I_2 : S \rightarrow E \$ \bullet$ $I_3 : E \rightarrow E + \bullet T$ $T \rightarrow \bullet \text{id}$ $T \rightarrow \bullet (E)$	$I_7 : T \rightarrow (E \bullet)$ $E \rightarrow E \bullet + T$ $I_8 : T \rightarrow (E) \bullet$ $I_9 : E \rightarrow T \bullet$

The corresponding CFSM:

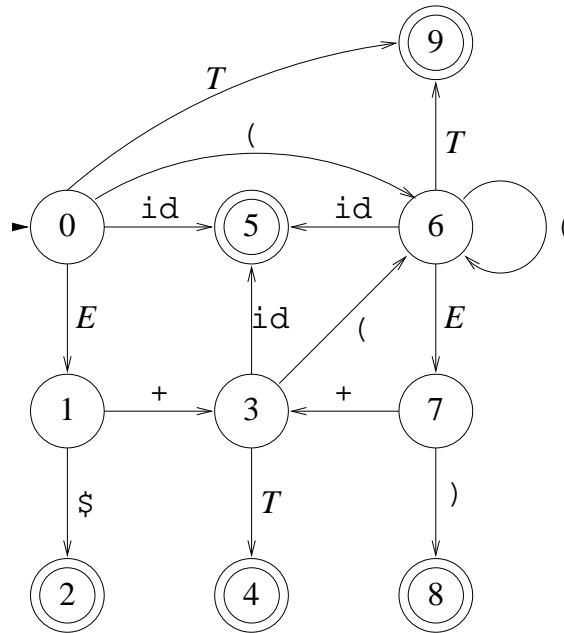


# Constructing the LR(0) parsing table

---

1. construct the collection of sets of LR(0) items for  $S'$
2. state  $i$  of the CFSM is constructed from  $I_i$ 
  - (a)  $[A \rightarrow \alpha \bullet a\beta] \in I_i$  and  $\text{goto0}(I_i, a) = I_j$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"shift } j\text{"}$
  - (b)  $[A \rightarrow \alpha \bullet] \in I_i, A \neq S'$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"reduce } A \rightarrow \alpha\text{"}, \forall a$
  - (c)  $[S' \rightarrow S\$ \bullet] \in I_i$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"accept"}, \forall a$
3.  $\text{goto0}(I_i, A) = I_j$   
 $\Rightarrow \text{GOTO}[i, A] = j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser  $s_0$  is  $\text{closure0}([S' \rightarrow \bullet S\$])$

# LR(0) example



state	ACTION					GOTO		
	id	(	)	+	\$	S	E	T
0	s5	s6	-	-	-	-	1	9
1	-	-	-	s3	s2	-	-	-
2	acc	acc	acc	acc	acc	-	-	-
3	s5	s6	-	-	-	-	-	4
4	r2	r2	r2	r2	r2	-	-	-
5	r4	r4	r4	r4	r4	-	-	-
6	s5	s6	-	-	-	-	7	9
7	-	-	s8	s3	-	-	-	-
8	r5	r5	r5	r5	r5	-	-	-
9	r3	r3	r3	r3	r3	-	-	-

## Conflicts in the ACTION table

---

If the LR(0) parsing table contains any multiply-defined ACTION entries then  $G$  is not LR(0)

Two conflicts arise:

**shift-reduce** : both shift and reduce possible in same item set

**reduce-reduce** : more than one distinct reduce action possible in same item set

Conflicts can be resolved through *lookahead* in ACTION. Consider:

- $A \rightarrow \epsilon|a\alpha$   
 $\Rightarrow$  shift-reduce conflict
- $a:=b+c*d$   
requires lookahead to avoid shift-reduce conflict after shifting  $c$  (need to see  $*$  to give precedence over  $+$ )

## A simple approach to adding lookahead: SLR(1)

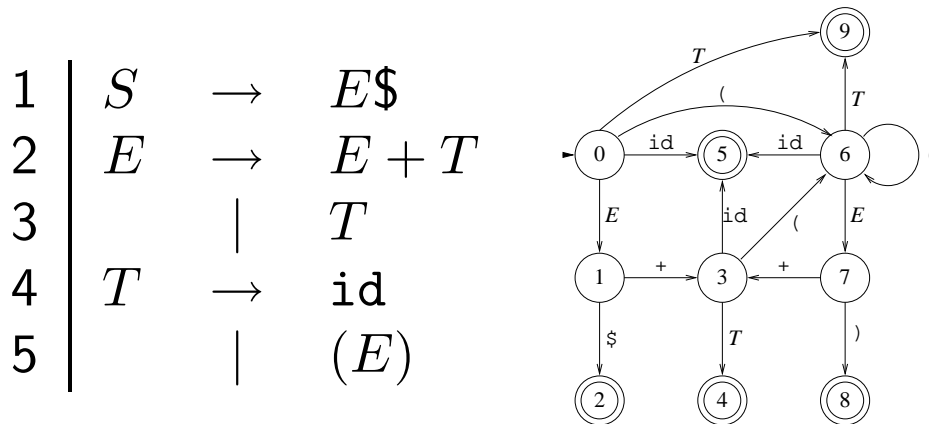
---

Add lookaheads after building LR(0) item sets

Constructing the SLR(1) parsing table:

1. construct the collection of sets of LR(0) items for  $G'$
2. state  $i$  of the CFSM is constructed from  $I_i$ 
  - (a)  $[A \rightarrow \alpha \bullet a\beta] \in I_i$  and  $\text{goto0}(I_i, a) = I_j$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"shift } j\text{"}, \forall a \neq \$$
  - (b)  $[A \rightarrow \alpha \bullet] \in I_i, A \neq S'$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"reduce } A \rightarrow \alpha\text{"},$   
 $\forall a \in \text{FOLLOW}(A)$
  - (c)  $[S' \rightarrow S \bullet \$] \in I_i$   
 $\Rightarrow \text{ACTION}[i, \$] = \text{"accept"}$
3.  $\text{goto0}(I_i, A) = I_j$   
 $\Rightarrow \text{GOTO}[i, A] = j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser  $s_0$  is  $\text{closure0}([S' \rightarrow \bullet S\$])$

## From previous example



$$\text{FOLLOW}(E) = \text{FOLLOW}(T) = \{\$, +, )\}$$

state	ACTION					GOTO		
	id	( )	+	\$		S	E	T
0	s5	s6	-	-	-	-	1	9
1	-	-	-	s3	acc	-	-	-
2	-	-	-	-	-	-	-	-
3	s5	s6	-	-	-	-	-	4
4	-	-	r2	r2	r2	-	-	-
5	-	-	r4	r4	r4	-	-	-
6	s5	s6	-	-	-	-	7	9
7	-	-	s8	s3	-	-	-	-
8	-	-	r5	r5	r5	-	-	-
9	-	-	r3	r3	r3	-	-	-

# Example: A grammar that is not LR(0)

1	$S \rightarrow E\$$
2	$E \rightarrow E + T$
3	$\quad \quad \quad   \quad T$
4	$T \rightarrow T * F$
5	$\quad \quad \quad   \quad F$
6	$F \rightarrow \text{id}$
7	$\quad \quad \quad   \quad (E)$

	FOLLOW
$E$	$\{+, ), \$\}$
$T$	$\{+, *, ), \$\}$
$F$	$\{+, *, ), \$\}$

LR(0) item sets:

$I_0 :$ <ul style="list-style-type: none"> <li><math>S \rightarrow \bullet E \\$</math></li> <li><math>E \rightarrow \bullet E + T</math></li> <li><math>E \rightarrow \bullet T</math></li> <li><math>T \rightarrow \bullet T * F</math></li> <li><math>T \rightarrow \bullet F</math></li> <li><math>F \rightarrow \bullet \text{id}</math></li> <li><math>F \rightarrow \bullet (E)</math></li> </ul>	$I_6 :$ <ul style="list-style-type: none"> <li><math>F \rightarrow (\bullet E)</math></li> <li><math>E \rightarrow \bullet E + T</math></li> <li><math>E \rightarrow \bullet T</math></li> <li><math>T \rightarrow \bullet T * F</math></li> <li><math>T \rightarrow \bullet F</math></li> <li><math>F \rightarrow \bullet \text{id}</math></li> <li><math>F \rightarrow \bullet (E)</math></li> </ul>
$I_1 :$ <ul style="list-style-type: none"> <li><math>S \rightarrow E \bullet \\$</math></li> <li><math>E \rightarrow E \bullet + T</math></li> </ul>	$I_7 :$ <ul style="list-style-type: none"> <li><math>E \rightarrow T \bullet</math></li> <li><math>T \rightarrow T \bullet * F</math></li> </ul>
$I_2 :$ <ul style="list-style-type: none"> <li><math>S \rightarrow E \\$ \bullet</math></li> </ul>	$I_8 :$ <ul style="list-style-type: none"> <li><math>T \rightarrow T * \bullet F</math></li> <li><math>F \rightarrow \bullet \text{id}</math></li> <li><math>F \rightarrow \bullet (E)</math></li> </ul>
$I_3 :$ <ul style="list-style-type: none"> <li><math>E \rightarrow E + \bullet T</math></li> <li><math>T \rightarrow \bullet T * F</math></li> <li><math>T \rightarrow \bullet F</math></li> <li><math>F \rightarrow \bullet \text{id}</math></li> <li><math>F \rightarrow \bullet (E)</math></li> </ul>	$I_9 :$ <ul style="list-style-type: none"> <li><math>T \rightarrow T * F \bullet</math></li> </ul>
$I_4 :$ <ul style="list-style-type: none"> <li><math>T \rightarrow F \bullet</math></li> </ul>	$I_{10} :$ <ul style="list-style-type: none"> <li><math>F \rightarrow (E) \bullet</math></li> </ul>
$I_5 :$ <ul style="list-style-type: none"> <li><math>F \rightarrow \text{id} \bullet</math></li> </ul>	$I_{11} :$ <ul style="list-style-type: none"> <li><math>E \rightarrow E + T \bullet</math></li> <li><math>T \rightarrow T \bullet * F</math></li> </ul>
	$I_{12} :$ <ul style="list-style-type: none"> <li><math>F \rightarrow (E \bullet)</math></li> <li><math>E \rightarrow E \bullet + T</math></li> </ul>

## Example: But it is SLR(1)

---

state	ACTION						GOTO			
	+	*	id	(	)	\$	S	E	T	F
0	–	–	s5	s6	–	–	–	1	7	4
1	s3	–	–	–	–	acc	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–
3	–	–	s5	s6	–	–	–	–	11	4
4	r5	r5	–	–	r5	r5	–	–	–	–
5	r6	r6	–	–	r6	r6	–	–	–	–
6	–	–	s5	s6	–	–	–	12	7	4
7	r3	s8	–	–	r3	r3	–	–	–	–
8	–	–	s5	s6	–	–	–	–	–	9
9	r4	r4	–	–	r4	r4	–	–	–	–
10	r7	r7	–	–	r7	r7	–	–	–	–
11	r2	s8	–	–	r2	r2	–	–	–	–
12	s3	–	–	–	s10	–	–	–	–	–

# Example: A grammar that is not SLR(1)

---

Consider:

$$\begin{array}{rcl}
 S & \rightarrow & L = R \\
 & | & R \\
 L & \rightarrow & *R \\
 & | & \text{id} \\
 R & \rightarrow & L
 \end{array}$$

Its LR(0) item sets:

$$\begin{array}{ll}
 I_0 : & S' \rightarrow \bullet S \$ \\
 & S \rightarrow \bullet L = R \\
 & S \rightarrow \bullet R \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \text{id} \\
 & R \rightarrow \bullet L \\
 I_1 : & S' \rightarrow S \bullet \$ \\
 I_2 : & S \rightarrow L \bullet = R \\
 & R \rightarrow L \bullet \\
 I_3 : & S \rightarrow R \bullet \\
 I_4 : & L \rightarrow * \bullet R \\
 & R \rightarrow \bullet L \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \text{id} \\
 I_5 : & L \rightarrow \text{id} \bullet \\
 I_6 : & S \rightarrow L = \bullet R \\
 & R \rightarrow \bullet L \\
 & L \rightarrow \bullet * R \\
 & L \rightarrow \bullet \text{id} \\
 I_7 : & L \rightarrow * R \bullet \\
 I_8 : & R \rightarrow L \bullet \\
 I_9 : & S \rightarrow L = R \bullet
 \end{array}$$

Consider  $I_2$ :

$$= \in \text{FOLLOW}(R) \quad (S \Rightarrow L = R \Rightarrow *R = R)$$

## LR(1) items

---

An LR(1) item is one in which

- All the lookahead strings are constrained to have length 1
- Look something like  $[A \rightarrow X \bullet YZ, a]$

What's the point of the lookahead symbols?

- carry along to choose correct reduction when there is a choice
- lookaheads are bookkeeping, unless item has  $\bullet$  at right end:
  - in  $[A \rightarrow X \bullet YZ, a]$ ,  $a$  has no direct use
  - in  $[A \rightarrow XYZ \bullet, a]$ ,  $a$  is useful
- allows use of grammars that are not *uniquely invertible*<sup>1</sup>

**The point:** For  $[A \rightarrow \alpha \bullet, a]$  and  $[B \rightarrow \alpha \bullet, b]$ , we can decide between reducing to  $A$  or  $B$  by looking at limited right context

---

<sup>1</sup>a grammar is uniquely invertible if no two productions have the same RHS

## **closure1( $I$ )**

---

Given an item  $[A \rightarrow \alpha \bullet B\beta, a]$ , 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, the input should reduce to  $B\beta$  (or  $\gamma$  for some other item  $[B \rightarrow \bullet\gamma, b]$  in the closure).

```
function closure1( $I$ )
repeat
  if  $[A \rightarrow \alpha \bullet B\beta, a] \in I$ 
    add  $[B \rightarrow \bullet\gamma, b]$  to  $I$ , where  $b \in \text{FIRST}(\beta a)$ 
until no more items can be added to  $I$ 
return  $I$ 
```

## **goto1( $I$ )**

---

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

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

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  $I$ .

function  $\text{goto1}(I, X)$

let  $J$  be the set of items

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

return  $\text{closure1}(J)$

## Building the LR(1) item sets for grammar $G$

---

We start the construction with the item  $[S' \rightarrow \bullet S, \$]$ ,  
where

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

$S$  is the start symbol of  $G$

$\$$  represents EOF

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

```
function items( $G'$ )
   $s_0 = \text{closure}_1(\{[S' \rightarrow \bullet S, \$]\})$ 
   $S = \{s_0\}$ 
  repeat
    for each set of items  $s \in S$ 
      for each grammar symbol  $X$ 
        if  $\text{goto}_1(s, X) \neq \emptyset$  and  $\text{goto}_1(s, X) \notin S$ 
          add  $\text{goto}_1(s, X)$  to  $S$ 
  until no more item sets can be added to  $S$ 
  return  $S$ 
```

# Constructing the LR(1) parsing table

---

Build lookahead into the DFA to begin with

1. construct the collection of sets of LR(1) items for  $G'$
2. state  $i$  of the LR(1) machine is constructed from  $I_i$ 
  - (a)  $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$  and  $\text{goto1}(I_i, a) = I_j$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"shift } j\text{"}$
  - (b)  $[A \rightarrow \alpha \bullet, \underline{a}] \in I_i, A \neq S'$   
 $\Rightarrow \text{ACTION}[i, \underline{a}] = \text{"reduce } A \rightarrow \alpha\text{"}$
  - (c)  $[S' \rightarrow S \bullet, \$] \in I_i$   
 $\Rightarrow \text{ACTION}[i, \$] = \text{"accept"}$
3.  $\text{goto1}(I_i, A) = I_j$   
 $\Rightarrow \text{GOTO}[i, A] = j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser  $s_0$  is  $\text{closure1}([S' \rightarrow \bullet S, \$])$

## Back to previous example ( $\notin$ SLR(1))

$$\begin{array}{rcl}
 S & \rightarrow & L = R \\
 & | & R \\
 L & \rightarrow & *R \\
 & | & \text{id} \\
 R & \rightarrow & L
 \end{array}$$

Its LR(1) item sets:

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

$I_2$  no longer has shift-reduce conflict:  
reduce on \$, shift on =

# Example: back to the SLR(1) expression grammar

---

In general, LR(1) has many more states than LR(0)/SLR(1):

1	$S \rightarrow E$
2	$E \rightarrow E + T$
3	$\quad \quad \quad   \quad T$
4	$T \rightarrow T * F$
5	$\quad \quad \quad   \quad F$
6	$F \rightarrow \text{id}$
7	$\quad \quad \quad   \quad (E)$

LR(1) item sets:

$I_0 :$ $S \rightarrow \bullet E, \quad \$$ $E \rightarrow \bullet E + T, \quad +\$$ $E \rightarrow \bullet T, \quad +\$$ $T \rightarrow \bullet T * F, \quad *+\$$ $T \rightarrow \bullet F, \quad *+\$$ $F \rightarrow \bullet \text{id}, \quad *+\$$ $F \rightarrow \bullet (E), \quad *+\$$	$I'_0 : \text{shifting } ($ $S \rightarrow (\bullet E), \quad *+\$$ $E \rightarrow \bullet E + T, \quad +)$ $E \rightarrow \bullet T, \quad +)$ $T \rightarrow \bullet T * F, \quad *+)$ $T \rightarrow \bullet F, \quad *+)$ $F \rightarrow \bullet \text{id}, \quad *+)$ $F \rightarrow \bullet (E), \quad *+)$	$I''_0 : \text{shifting } ($ $S \rightarrow (\bullet E), \quad *+)$ $E \rightarrow \bullet E + T, \quad +)$ $E \rightarrow \bullet T, \quad +)$ $T \rightarrow \bullet T * F, \quad *+)$ $T \rightarrow \bullet F, \quad *+)$ $F \rightarrow \bullet \text{id}, \quad *+)$ $F \rightarrow \bullet (E), \quad *+)$
--	--	--

## Another example

Consider:

0	$S'$	$\rightarrow$	$S$
1	$S$	$\rightarrow$	$CC$
2	$C$	$\rightarrow$	$cC$
3		$ $	$d$

LR(1) item sets:

$I_0 :$	$S' \rightarrow \bullet S,$	$\$$
	$S \rightarrow \bullet CC,$	$\$$
	$C \rightarrow \bullet cC,$	$cd$
	$C \rightarrow \bullet d,$	$cd$
$I_1 :$	$S' \rightarrow S \bullet,$	$\$$
$I_2 :$	$S \rightarrow C \bullet C,$	$\$$
	$C \rightarrow \bullet cC,$	$\$$
	$C \rightarrow \bullet d,$	$\$$
$I_3 :$	$C \rightarrow c \bullet C,$	$cd$
	$C \rightarrow \bullet cC,$	$cd$
	$C \rightarrow \bullet d,$	$cd$
$I_4 :$	$C \rightarrow d \bullet,$	$cd$
$I_5 :$	$S \rightarrow CC \bullet,$	$\$$
$I_6 :$	$C \rightarrow c \bullet C,$	$\$$
	$C \rightarrow \bullet cC,$	$\$$
	$C \rightarrow \bullet d,$	$\$$
$I_7 :$	$C \rightarrow d \bullet,$	$\$$
$I_8 :$	$C \rightarrow cC \bullet,$	$cd$
$I_9 :$	$C \rightarrow cC \bullet,$	$\$$

state	ACTION			GOTO	
	<i>c</i>	<i>d</i>	$\$$	<i>S</i>	<i>C</i>
0	s3	s4	—	1	2
1	—	—	acc	—	—
2	s6	s7	—	—	5
3	s3	s4	—	—	8
4	r3	r3	—	—	—
5	—	—	r1	—	—
6	s6	s7	—	—	9
7	—	—	r3	—	—
8	r2	r2	—	—	—
9	—	—	r2	—	—

## LALR(1) parsing

---

Define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- $\{[A \rightarrow \alpha \bullet \beta, a], [A' \rightarrow \alpha' \bullet \beta', b]\}$ , and
- $\{[A \rightarrow \alpha \bullet \beta, c], [A' \rightarrow \alpha' \bullet \beta', d]\}$

have the same core.

*Key idea:*

If two sets of LR(1) items,  $I_i$  and  $I_j$ , have the same core, we can merge the states that represent them in the ACTION and GOTO tables.

# LALR(1) table construction

---

To construct LALR(1) parsing tables, we can insert a single step into the LR(1) algorithm

**(1.5)** For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.

The goto function must be updated to reflect the replacement sets.

The resulting algorithm has large space requirements.

# LALR(1) table construction

---

The revised (*and renumbered*) algorithm

1. construct the collection of sets of LR(1) items for  $G'$
2. for each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union. (Update the goto function incrementally)
3. state  $i$  of the LALR(1) machine is constructed from  $I_i$ 
  - (a)  $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$  and  $\text{goto1}(I_i, a) = I_j$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"shift } j\text{"}$
  - (b)  $[A \rightarrow \alpha \bullet, a] \in I_i, A \neq S'$   
 $\Rightarrow \text{ACTION}[i, a] = \text{"reduce } A \rightarrow \alpha\text{"}$
  - (c)  $[S' \rightarrow S \bullet, \$] \in I_i$   
 $\Rightarrow \text{ACTION}[i, \$] = \text{"accept"}$
4.  $\text{goto1}(I_i, A) = I_j$   
 $\Rightarrow \text{GOTO}[i, A] = j$
5. set undefined entries in ACTION and GOTO to "error"
6. initial state of parser  $s_0$  is  $\text{closure1}([S' \rightarrow \bullet S, \$])$

# Example

Reconsider:

0	$S' \rightarrow S$
1	$S \rightarrow CC$
2	$C \rightarrow cC$
3	$\quad \quad \quad   \quad d$

LR(1) item sets:

$I_0 : S' \rightarrow \bullet S, \$$	$S \rightarrow \bullet CC, \$$	$C \rightarrow \bullet cC, cd$	$C \rightarrow \bullet d, cd$
$I_1 : S' \rightarrow S \bullet, \$$	$I_2 : S \rightarrow C \bullet C, \$$	$C \rightarrow \bullet cC, \$$	$C \rightarrow \bullet d, \$$
$I_3 : C \rightarrow c \bullet C, cd$	$C \rightarrow \bullet cC, cd$	$C \rightarrow \bullet d, cd$	
$I_4 : C \rightarrow d \bullet, cd$	$I_5 : S \rightarrow CC \bullet, \$$	$I_6 : C \rightarrow c \bullet C, \$$	$C \rightarrow \bullet cC, \$$
$I_7 : C \rightarrow d \bullet, \$$	$I_8 : C \rightarrow cC \bullet, cd$	$I_9 : C \rightarrow cC \bullet, \$$	

Merged states:

$I_{36} : C \rightarrow c \bullet C, cd\$$	$C \rightarrow \bullet cC, cd\$$	$C \rightarrow \bullet d, cd\$$
$I_{47} : C \rightarrow d \bullet, cd\$$	$I_{89} : C \rightarrow cC \bullet, cd\$$	

state	ACTION			GOTO	
	<i>c</i>	<i>d</i>	$\$$	<i>S</i>	<i>C</i>
0	s36	s47	–	1	2
1	–	–	acc	–	–
2	s36	s47	–	–	5
36	s36	s47	–	–	89
47	r3	r3	r3	–	–
5	–	–	r1	–	–
89	r2	r2	r2	–	–

# The role of precedence

---

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- lookahead with higher precedence  $\Rightarrow$  *shift*
- same precedence, left associative  $\Rightarrow$  *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees  $\Rightarrow$  fewer reductions

Classic application: expression grammars

With precedence and associativity, we can use:

$$\begin{array}{l}
 E \quad \rightarrow \quad E * E \\
 \quad \quad | \quad E / E \\
 \quad \quad | \quad E + E \\
 \quad \quad | \quad E - E \\
 \quad \quad | \quad (E) \\
 \quad \quad | \quad -E \\
 \quad \quad | \quad \text{id} \\
 \quad \quad | \quad \text{num}
 \end{array}$$

# Error recovery in shift-reduce parsers

---

## The problem

- encounter an invalid token
- bad pieces of tree hanging from stack
- incorrect entries in symbol table

We want to *parse* the rest of the file

## Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message (*including line number*)

# Left versus right recursion

---

Right Recursion:

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:

- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

# Parsing review

---

## *Recursive descent*

A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

## LL( $k$ )

An LL( $k$ ) parser must be able to recognize the use of a production after seeing only the first  $k$  symbols of its right hand side.

## LR( $k$ )

An LR( $k$ ) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with  $k$  symbols of lookahead.