Bottom-up parsing

Recall

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

- If $\alpha \in V_t^*$, then α is 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 of 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

$$\begin{array}{c|lcl}
1 & S & \rightarrow & \texttt{a}AB\texttt{e} \\
2 & A & \rightarrow & A\texttt{bc} \\
3 & & | & \texttt{b} \\
4 & B & \rightarrow & \texttt{d}
\end{array}$$

and the input string `abbcde`

Prod'n.	Sentential Form
3	$\texttt{a}\boxed{\texttt{b}}\texttt{bcde}$
2	$\texttt{a}\boxed{A\texttt{bc}}\texttt{de}$
4	$\texttt{a}A\boxed{\texttt{d}}\texttt{e}$
1	$\boxed{\texttt{a}AB\texttt{e}}$
–	S

Scan the input and find *prefixes* of sentential forms!

Handles

What are we trying to find?

A substring α of the tree's upper frontier that

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

We call such a string a *handle*.

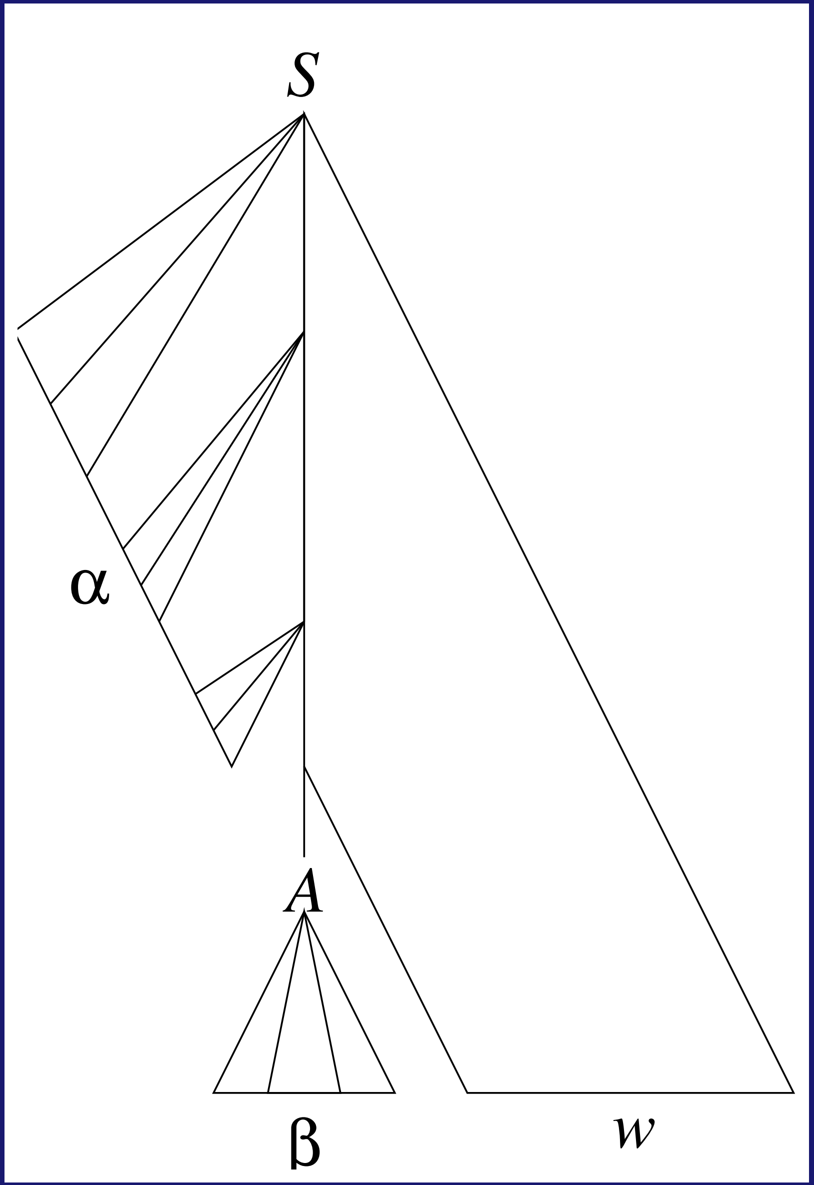
Formally:

> In a right-sentential form $\alpha\beta w$, the string β is a handle for production $A \to \beta$

i.e., if $S \Rightarrow^*_{\text{rm}} \alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$ then β is a handle for $A \to \beta$ in $\alpha\beta w$

All right-sentential forms have a suffix containing 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 γ_{i-1} to γ_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	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\vert$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\vert$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\vert$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\vert$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\texttt{num}$
9		$\vert$	$\texttt{id}$

Prod'n.	Sentential Form
–	$\langle\text{goal}\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle\text{expr}\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

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 for $A \rightarrow \beta$ on the stack, *reduce*:

      i) pop $|\beta|$ symbols off the stack

      ii) push A onto the stack

Example: back to $\mathtt{x} - 2 * \mathtt{y}$

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\vert$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\vert$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\vert$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\vert$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\mathtt{num}$
9		$\vert$	$\mathtt{id}$

Stack	Input	Action
$\$$	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	shift
$\$\underline{\mathtt{id}}$	$- \mathtt{num} * \mathtt{id}$	reduce 9
$\$\underline{\langle\text{factor}\rangle}$	$- \mathtt{num} * \mathtt{id}$	reduce 7
$\$\underline{\langle\text{term}\rangle}$	$- \mathtt{num} * \mathtt{id}$	reduce 4
$\$\langle\text{expr}\rangle$	$- \mathtt{num} * \mathtt{id}$	shift
$\$\langle\text{expr}\rangle\ -$	$\mathtt{num} * \mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \underline{\mathtt{num}}$	$* \mathtt{id}$	reduce 8
$\$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle}$	$* \mathtt{id}$	reduce 7
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle$	$* \mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle\ *$	$\mathtt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\mathtt{id}}$		reduce 9
$\$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$		reduce 5
$\$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$		reduce 3
$\$\underline{\langle\text{expr}\rangle}$		reduce 1
$\$\langle\text{goal}\rangle$		accept

1. *Shift until top of stack is the right end of a handle*
2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 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

But how do we know

- that there is a complete handle on the stack?
- which handle to use?

LR parsing: key insight

Recognize handles with a DFA [Knuth1965]

- 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 $A \to \beta$"
    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; *not* the same 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

LR(k) grammars

Informally, we say that a grammar G is $\mathrm{LR}(k)$ if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form*, and
2. *determine the production by which to reduce*

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

LR(k) grammars

Formally, a grammar G is LR(k) iff

1. $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{\mathrm{rm}} \alpha\beta w$, and
2. $S \Rightarrow^*_{\mathrm{rm}} \gamma B x \Rightarrow_{\mathrm{rm}} \alpha\beta y$, and
3. $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$

implies $\alpha A y = \gamma B x$

i.e., consider sentential forms $\alpha\beta w$ and $\alpha\beta y$, with common prefix $\alpha\beta$ and common k-symbol lookahead $\mathrm{FIRST}_k(y) = \mathrm{FIRST}_k(w)$, where there might be two choices, $\alpha A w$ and $\gamma B x$, that reduce to $\alpha\beta w$ and $\alpha\beta y$, respectively.

In an LR(k) grammar, there is no such choice: It must be that $\alpha A y = \gamma B x$.

Why study LR grammars?

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

We call these parsers LR(1) parsers.

- 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 derived from β

LR(k)**:** recognize the handle β after seeing everything derived from β plus k lookahead symbols

LR parsing

Three common algorithms 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 a realistic language 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 $[A \to \alpha \bullet \beta, \gamma]$, where

$A \to \alpha\beta$ is a production of G where the position of the $\bullet$ is arbitrary; it marks how much of the RHS of a production has already been seen

γ 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*)

$A \rightarrow XYZ$ has four associated 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:

- `closure0`(I) to build its states
- `goto0`(I, X) to determine its transitions

closure0

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

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

Let I be a set of LR(0) items. $\texttt{closure0}(I)$ is the smallest set such that

1. $I \subseteq \texttt{closure0}(s)$
2. $[A \to \alpha \bullet \beta] \in \texttt{closure0}(I)$ and $B \to \gamma$ a production implies $[B \to \bullet\gamma] \in \texttt{closure0}(I)$.

Implementation: Start with rule (1), then repeat rule (2) until no further items need to be added.

goto0

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

```
goto0(I,X) =
  closure0({[A → αX • β] | [A → α • Xβ] ∈ I})
```

If I is the set of valid items for some viable prefix γ, then GOTO(I,X) is the set of valid items for the viable prefix γX.

GOTO(I,X) represents state after recognizing X in state I.

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`

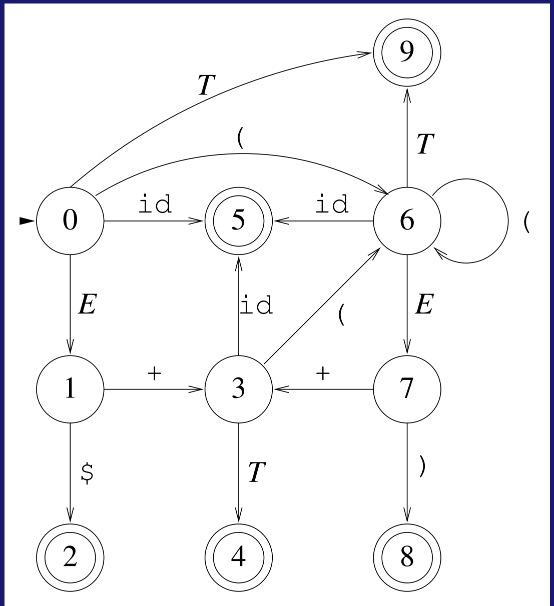
To compute the collection of sets of LR(0) items

```
function items($G'$)
  $I_0 \leftarrow$ closure0($\{[S' \rightarrow \bullet S\$]\}$)
  $\mathcal{S} \leftarrow \mathcal{W} \leftarrow \{I_0\}$
  while $W \neq \emptyset$
    remove $I$ from $\mathcal{W}$
    for each grammar symbol $X$
      if goto0$(I,X) \neq \emptyset$ and goto0$(I,X) \notin \mathcal{S} \cup \mathcal{W}$
        add goto0$(I,X)$ to $\mathcal{S}$ and $\mathcal{W}$
  return $\mathcal{S}$
```

LR(0) example

$$\begin{array}{r|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & | & T \\
4 & T & \rightarrow & \texttt{id} \\
5 & & | & (E)
\end{array}$$

The corresponding CFSM:



$$\begin{array}{rl}
I_0: & S \rightarrow \bullet E\$ \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet\texttt{id} \\
& T \rightarrow \bullet(E) \\
I_1: & S \rightarrow E\bullet\$ \\
& E \rightarrow E\bullet+T \\
I_2: & S \rightarrow E\$\bullet \\
I_3: & E \rightarrow E+\bullet T \\
& T \rightarrow \bullet\texttt{id} \\
& T \rightarrow \bullet(E)
\end{array}$$

$$\begin{array}{rl}
I_4: & E \rightarrow E+T\bullet \\
I_5: & T \rightarrow \texttt{id}\bullet \\
I_6: & T \rightarrow (\bullet E) \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet\texttt{id} \\
& T \rightarrow \bullet(E) \\
I_7: & T \rightarrow (E\bullet) \\
& E \rightarrow E\bullet+T \\
I_8: & T \rightarrow (E)\bullet \\
I_9: & E \rightarrow T\bullet
\end{array}$$

Constructing the LR(0) parsing table

1. construct the collection of sets of LR(0) items for G': $\{I_0, I_1, \ldots\}$
2. state i of the CFSM is constructed from I_i

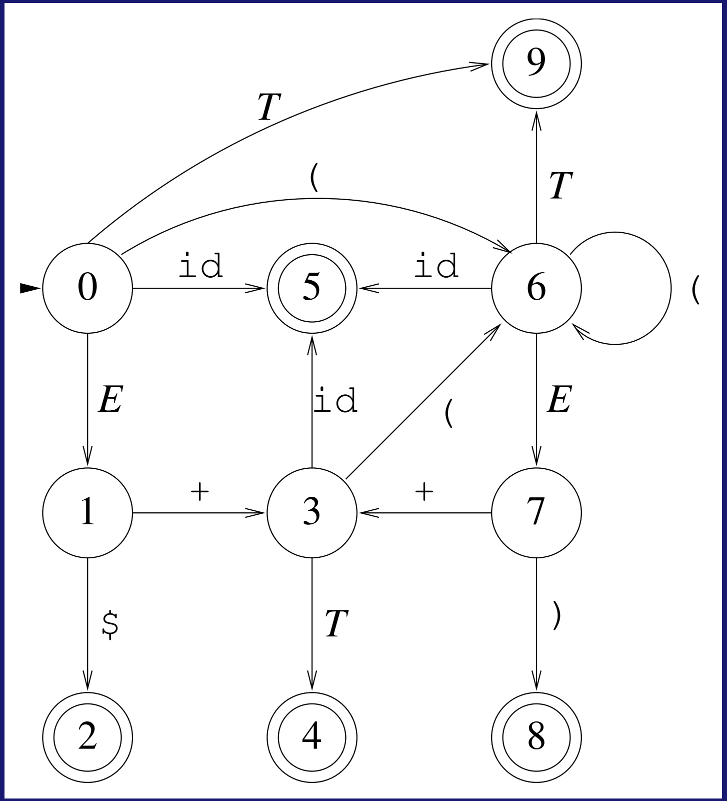
   (a) $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $\texttt{goto0}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift* j"

   (b) $[A \rightarrow \alpha\bullet] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*reduce* $A \rightarrow \alpha$", $\forall a$

   (c) $[S' \rightarrow S\$\bullet] \in I_i$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*accept*", $\forall a$

3. $\texttt{goto0}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser corresponds to $I_0 = \texttt{closure0}([S' \rightarrow \bullet S\$])$

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 kinds of conflict 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:

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

SLR(1): simple lookahead LR

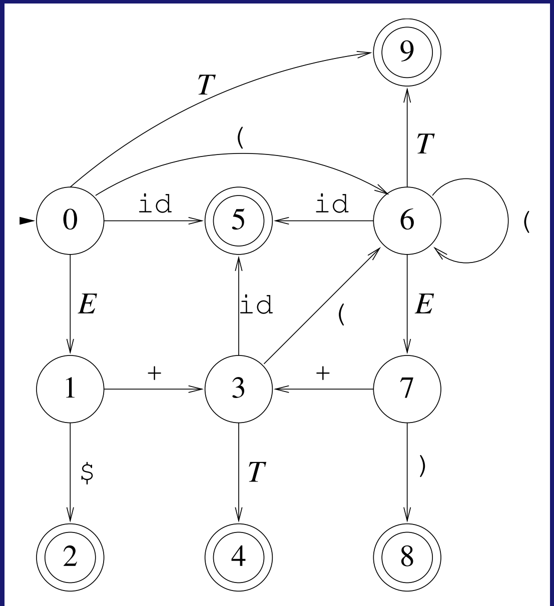
Add lookaheads after building LR(0) item sets

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) $[A \to \alpha \bullet a\beta] \in I_i$ and $\texttt{goto0}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift* j", $\underline{\forall a \neq \$}$
   (b) $[A \to \alpha\bullet] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*reduce* $A \to \alpha$", $\underline{\forall a \in \text{FOLLOW}(A)}$
   (c) $\underline{[S' \to S \bullet \$]} \in I_i$
   $\Rightarrow$ ACTION$[i, \underline{\$}] \leftarrow$ "*accept*"
3. $\texttt{goto0}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{closure0}([S' \to \bullet S\$])$

From previous example

$$
\begin{array}{r|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & | & T \\
4 & T & \rightarrow & \texttt{id} \\
5 & & | & (E)
\end{array}
$$



FOLLOW(E) = 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)

$$
\begin{array}{l|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & | & T \\
4 & T & \rightarrow & T*F \\
5 & & | & F \\
6 & F & \rightarrow & \texttt{id} \\
7 & & | & (E)
\end{array}
$$

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

$$
\begin{array}{ll}
I_0 : & S \rightarrow \bullet E\$ \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet\texttt{id} \\
& F \rightarrow \bullet(E) \\
I_1 : & S \rightarrow E\bullet\$ \\
& E \rightarrow E\bullet+T \\
I_2 : & S \rightarrow E\$\bullet \\
I_3 : & E \rightarrow E+\bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet\texttt{id} \\
& F \rightarrow \bullet(E) \\
I_4 : & T \rightarrow F\bullet \\
I_5 : & F \rightarrow \texttt{id}\bullet
\end{array}
$$

$$
\begin{array}{ll}
I_6 : & F \rightarrow (\bullet E) \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet\texttt{id} \\
& F \rightarrow \bullet(E) \\
I_7 : & E \rightarrow T\bullet \\
& T \rightarrow T\bullet *F \\
I_8 : & T \rightarrow T*\bullet F \\
& F \rightarrow \bullet\texttt{id} \\
& F \rightarrow \bullet(E) \\
I_9 : & T \rightarrow T*F\bullet \\
I_{10} : & F \rightarrow (E)\bullet \\
I_{11} : & E \rightarrow E+T\bullet \\
& T \rightarrow T\bullet *F \\
I_{12} : & F \rightarrow (E\bullet) \\
& E \rightarrow E\bullet +T
\end{array}
$$

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:

$$\begin{array}{lcl} S & \rightarrow & L = R \\ & \mid & R \\ L & \rightarrow & *R \\ & \mid & \texttt{id} \\ R & \rightarrow & L \end{array}$$

Its LR(0) item sets:

$$\begin{array}{l} I_0 : S' \rightarrow \bullet S\$ \\ \quad S \rightarrow \bullet L = R \\ \quad S \rightarrow \bullet R \\ \quad L \rightarrow \bullet * R \\ \quad L \rightarrow \bullet \texttt{id} \\ \quad R \rightarrow \bullet L \\ I_1 : S' \rightarrow S \bullet \$ \\ I_2 : S \rightarrow L \bullet = R \\ \quad R \rightarrow L \bullet \\ I_3 : S \rightarrow R \bullet \\ I_4 : L \rightarrow \texttt{id} \bullet \end{array}$$

$$\begin{array}{l} I_5 : L \rightarrow * \bullet R \\ \quad R \rightarrow \bullet L \\ \quad L \rightarrow \bullet * R \\ \quad L \rightarrow \bullet \texttt{id} \\ I_6 : S \rightarrow L = \bullet R \\ \quad R \rightarrow \bullet L \\ \quad L \rightarrow \bullet * R \\ \quad L \rightarrow \bullet \texttt{id} \\ I_7 : L \rightarrow *R \bullet \\ I_8 : R \rightarrow L \bullet \\ I_9 : S \rightarrow L = R \bullet \end{array}$$

Now consider I_2: $=$ $\in$ FOLLOW(R) $(S \Rightarrow L = R \Rightarrow *R = R)$

LR(1) items

Definition: An LR(k) item is a pair $[A \rightarrow \alpha \bullet \beta, \gamma]$, where

$A \rightarrow \alpha\beta$ is a production of G with a $\bullet$ at some position in the RHS, marking how much of the RHS of a production has been seen

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

What about LR(1) items?

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

LR(1) items

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 • 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*[†]

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

[†] 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 α.

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

Given an LR(1) item set I, `closure1`(I) is the smallest set such that

1. $I \subseteq \texttt{closure1}(I)$
2. if $[A \rightarrow \alpha \bullet B\beta, a] \in \texttt{closure1}(I)$, $B \rightarrow \gamma$ is a productions, and $b \in \text{FIRST}(\beta a)$, then $[B \rightarrow \bullet\gamma, b] \in \texttt{closure1}(I)$

goto1(I)

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

```
goto1(I,X) =
  closure1({[A → αX • β, a] | [A → α • Xβ, a] ∈ I})
```

If I is the set of valid items for some viable prefix γ, then GOTO(I,X) is the set of valid items for the viable prefix γX.

goto(I,X) represents the state after recognizing X in state I.

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($G'$)
  $I_0 \leftarrow$ closure1($\{[S' \rightarrow \bullet S, \$]\}$)
  $\mathcal{W} \leftarrow \mathcal{S} \leftarrow \{s_0\}$
  while $\mathcal{W} \neq \emptyset$
    remove $I$ from $\mathcal{W}$
    for each grammar symbol $X$
      if goto1($I,X$) $\neq \emptyset$ and goto1($I,X$) $\notin \mathcal{S} \cup \mathcal{W}$
        add goto1($I,X$) to $\mathcal{S}$ and $\mathcal{W}$
  return $\mathcal{S}$
```

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 $\texttt{goto1}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift j*"
   (b) $[A \rightarrow \alpha\bullet, \underline{a}] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \alpha$"
   (c) $[S' \rightarrow S\bullet, \$] \in I_i$
   $\Rightarrow$ ACTION$[i, \$] \leftarrow$ "*accept*"
3. $\texttt{goto1}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser corresponds to $I_0 = \texttt{closure1}([S' \rightarrow \bullet S, \$])$

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

$$\begin{array}{lcl}
S & \to & L = R \\
 & \mid & R \\
L & \to & *R \\
 & \mid & \texttt{id} \\
R & \to & L
\end{array}$$

$$\begin{array}{lll}
I_0 : & S' \to \bullet S, & \$ \\
 & S \to \bullet L = R, & \$ \\
 & S \to \bullet R, & \$ \\
 & L \to \bullet * R, & = \\
 & L \to \bullet \texttt{id}, & = \\
 & R \to \bullet L, & \$ \\
 & L \to \bullet * R, & \$ \\
 & L \to \bullet \texttt{id}, & \$ \\
I_1 : & S' \to S \bullet, & \$ \\
I_2 : & S \to L \bullet = R, & \$ \\
 & R \to L \bullet, & \$ \\
I_3 : & S \to R \bullet, & \$ \\
I_4 : & L \to * \bullet R, & = \$ \\
 & R \to \bullet L, & = \$ \\
 & L \to \bullet * R, & = \$ \\
 & L \to \bullet \texttt{id}, & = \$
\end{array}$$

$$\begin{array}{lll}
I_5 : & L \to \texttt{id} \bullet, & = \$ \\
I_6 : & S \to L = \bullet R, & \$ \\
 & R \to \bullet L, & \$ \\
 & L \to \bullet * R, & \$ \\
 & L \to \bullet \texttt{id}, & \$ \\
I_7 : & L \to *R \bullet, & = \$ \\
I_8 : & R \to L \bullet, & = \$ \\
I_9 : & S \to L = R \bullet, & \$ \\
I_{10} : & R \to L \bullet, & \$ \\
I_{11} : & L \to * \bullet R, & \$ \\
 & R \to \bullet L, & \$ \\
 & L \to \bullet * R, & \$ \\
 & L \to \bullet \texttt{id}, & \$ \\
I_{12} : & L \to \texttt{id} \bullet, & \$ \\
I_{13} : & L \to *R \bullet, & \$
\end{array}$$

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

Example: back to SLR(1) expression grammar

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

$$\begin{array}{r|lcl}
1 & S & \rightarrow & E \\
2 & E & \rightarrow & E+T \\
3 & & \mid & T
\end{array}
\qquad
\begin{array}{r|lcl}
4 & T & \rightarrow & T*F \\
5 & & \mid & F \\
6 & F & \rightarrow & \mathtt{id} \\
7 & & \mid & (E)
\end{array}$$

LR(1) item sets:

I_0 :

$$\begin{array}{ll}
S \rightarrow \bullet E, & \$ \\
E \rightarrow \bullet E+T, & +\$ \\
E \rightarrow \bullet T, & +\$ \\
T \rightarrow \bullet T*F, & *+\$ \\
T \rightarrow \bullet F, & *+\$ \\
F \rightarrow \bullet \mathtt{id}, & *+\$ \\
F \rightarrow \bullet (E), & *+\$
\end{array}$$

I'_0 :shifting (

$$\begin{array}{ll}
F \rightarrow (\bullet E), & *+\$ \\
E \rightarrow \bullet E+T, & +) \\
E \rightarrow \bullet T, & +) \\
T \rightarrow \bullet T*F, & *+) \\
T \rightarrow \bullet F, & *+) \\
F \rightarrow \bullet \mathtt{id}, & *+) \\
F \rightarrow \bullet (E), & *+)
\end{array}$$

I''_0 :shifting (

$$\begin{array}{ll}
F \rightarrow (\bullet E), & *+) \\
E \rightarrow \bullet E+T, & +) \\
E \rightarrow \bullet T, & +) \\
T \rightarrow \bullet T*F, & *+) \\
T \rightarrow \bullet F, & *+) \\
F \rightarrow \bullet \mathtt{id}, & *+) \\
F \rightarrow \bullet (E), & *+)
\end{array}$$

Another example

Consider:

$$\begin{array}{c|lcl} 0 & S' & \rightarrow & S \\ 1 & S & \rightarrow & CC \\ 2 & C & \rightarrow & cC \\ 3 & & \mid & d \end{array}$$

state	ACTION			GOTO	
	c	d	$\$$	S	C
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	–	–

LR(1) item sets:

$$\begin{array}{lll} I_0 : & S' \rightarrow \bullet S, & \$ \\ & S \rightarrow \bullet CC, & \$ \\ & C \rightarrow \bullet cC, & cd \\ & C \rightarrow \bullet d, & cd \\ I_1 : & S' \rightarrow S \bullet, & \$ \\ I_2 : & S \rightarrow C \bullet C, & \$ \\ & C \rightarrow \bullet cC, & \$ \\ & C \rightarrow \bullet d, & \$ \\ I_3 : & C \rightarrow c \bullet C, & cd \\ & C \rightarrow \bullet cC, & cd \\ & C \rightarrow \bullet d, & cd \end{array}$$

$$\begin{array}{lll} I_4 : & C \rightarrow d \bullet, & cd \\ I_5 : & S \rightarrow CC \bullet, & \$ \\ I_6 : & C \rightarrow c \bullet C, & \$ \\ & C \rightarrow \bullet cC, & \$ \\ & C \rightarrow \bullet d, & \$ \\ I_7 : & C \rightarrow d \bullet, & \$ \\ I_8 : & C \rightarrow cC \bullet, & cd \\ I_9 : & C \rightarrow cC \bullet, & \$ \end{array}$$

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

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

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 $\texttt{goto1}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift* j"
   (b) $[A \rightarrow \alpha\bullet, a] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*reduce* $A \rightarrow \alpha$"
   (c) $[S' \rightarrow S\bullet, \$] \in I_i \Rightarrow$ ACTION$[i, \$] \leftarrow$ "*accept*"
4. $\texttt{goto1}(I_i, A) = I_j \Rightarrow$ GOTO$[i, A] \leftarrow j$
5. set undefined entries in ACTION and GOTO to "*error*"
6. initial state of parser corresponds to $I_0 = \texttt{closure1}([S' \rightarrow \bullet S, \$])$

Example

Reconsider:

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

$$\begin{array}{ll} I_0 : S' \rightarrow \bullet S, & \$ \\ \quad S \rightarrow \bullet CC, & \$ \\ \quad C \rightarrow \bullet cC, & cd \\ \quad C \rightarrow \bullet d, & cd \\ I_1 : S' \rightarrow S\bullet, & \$ \\ I_2 : S \rightarrow C \bullet C, & \$ \\ \quad C \rightarrow \bullet cC, & \$ \\ \quad C \rightarrow \bullet d, & \$ \end{array}$$

$$\begin{array}{ll} I_3 : C \rightarrow c \bullet C, & cd \\ \quad C \rightarrow \bullet cC, & cd \\ \quad C \rightarrow \bullet d, & cd \\ I_4 : C \rightarrow d\bullet, & cd \\ I_5 : S \rightarrow CC\bullet, & \$ \end{array}$$

$$\begin{array}{ll} I_6 : C \rightarrow c \bullet C, & \$ \\ \quad C \rightarrow \bullet cC, & \$ \\ \quad C \rightarrow \bullet d, & \$ \\ I_7 : C \rightarrow d\bullet, & \$ \\ I_8 : C \rightarrow cC\bullet, & cd \\ I_9 : C \rightarrow cC\bullet, & \$ \end{array}$$

Merged states:

$$\begin{array}{ll} I_{36} : C \rightarrow c \bullet C, & cd\$ \\ \quad C \rightarrow \bullet cC, & cd\$ \\ \quad C \rightarrow \bullet d, & cd\$ \\ I_{47} : C \rightarrow d\bullet, & cd\$ \\ I_{89} : C \rightarrow cC\bullet, & cd\$ \end{array}$$

state	ACTION			GOTO	
	c	d	$\$$	S	C
0	s36	s47	–	1	2
1	–	–	acc	–	–
2	s36	s47	–	–	5
36	s36	s47	–	–	8
47	r3	r3	r3	–	–
5	–	–	r1	–	–
89	r2	r2	r2	–	–

More efficient LALR(1) construction

Observe that we can:

- represent I_i by its *basis* or *kernel*:
  items that are either $[S' \to \bullet S, \$]$
  or do not have $\bullet$ at the left of the RHS

- compute *shift*, *reduce* and *goto* actions for state derived from I_i directly from its kernel

This leads to a method that avoids building the complete canonical collection of sets of LR(1) items

The role of precedence

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

- lookahead with higher precedence ⇒ *shift*
- same precedence, left associative ⇒ *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees ⇒ fewer reductions

Classic application: expression grammars

The role of precedence

With precedence and associativity, we can use:

$$
\begin{array}{lcl}
E & \rightarrow & E * E \\
  & \mid & E / E \\
  & \mid & E + E \\
  & \mid & E - E \\
  & \mid & (E) \\
  & \mid & \texttt{-}E \\
  & \mid & \texttt{id} \\
  & \mid & \texttt{num}
\end{array}
$$

This eliminates useless reductions (*single productions*)

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 to `stderr` (*line number*)

Error recovery in yacc/bison/Java CUP

The error mechanism

- designated token `error`
- valid in any production
- `error` shows syncronization points

When an error is discovered

- pops the stack until `error` is legal
- skips input tokens until it successfully shifts 3
- `error` productions can have actions

This mechanism is fairly general

See §Error Recovery of the on-line CUP manual

Example

Using `error`

```
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`"
- invoke `yyerror("syntax error")`

Other "natural" places for errors

- all the "lists": `FieldList`, `CaseList`
- missing parentheses or brackets (`yychar`)
- extra operator or missing operator

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).

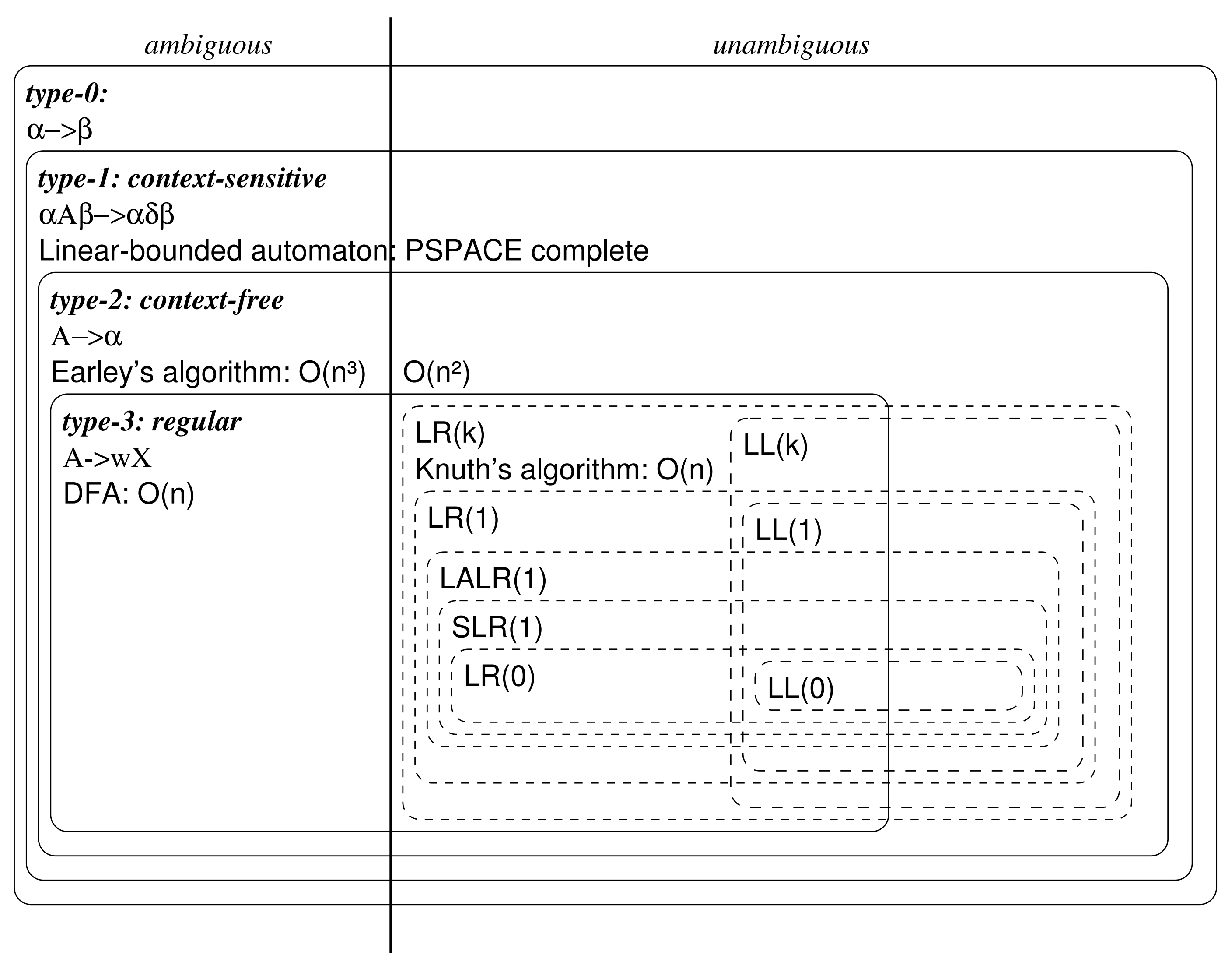
LL(k)

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.

Complexity of parsing: grammar hierarchy



ambiguous

unambiguous

type-0:

α–>β

type-1: context-sensitive

αAβ–>αδβ

Linear-bounded automaton: PSPACE complete

type-2: context-free

A–>α

Earley's algorithm: $O(n^3)$

$O(n^2)$

type-3: regular

A->wX

DFA: O(n)

LR(k)

Knuth's algorithm: O(n)

LL(k)

LR(1)

LL(1)

LALR(1)

SLR(1)

LR(0)

LL(0)

Note: this is a hierarchy of grammars *not* languages

Language vs. grammar

For example, every regular *language* has a grammar that is LL(1), but not all regular grammars are LL(1). Consider:

$$\begin{array}{rcl} S & \rightarrow & ab \\ S & \rightarrow & ac \end{array}$$

Without left-factoring, this grammar is not LL(1).