

Computer Science 160

Translation of Programming Languages

Instructor: Christopher Kruegel
(based on material by Tim Sherwood)

**Building a Handle Recognizing Machine:
[now, with a look-ahead token, which is LR(1)]**

LR(k) items

UC Santa Barbara

An LR(k) item is a pair $[A, B]$, where

A is a production $\alpha \rightarrow \beta \gamma \delta$ with a \bullet at some position in the *rhs*

B is a look-ahead string of length $\leq k$ (terminal symbols or \$)

Examples: $[\alpha \rightarrow \bullet \beta \gamma \delta, a]$, $[\alpha \rightarrow \beta \bullet \gamma \delta, a]$, $[\alpha \rightarrow \beta \gamma \bullet \delta, a]$, & $[\alpha \rightarrow \beta \gamma \delta \bullet, a]$

The \bullet in an item indicates the position of the top of the stack

LR(0) items $[\alpha \rightarrow \beta \bullet \gamma \delta]$ (no look-ahead symbol)

LR(1) items $[\alpha \rightarrow \beta \bullet \gamma \delta, a]$ (one token look-ahead)

LR(2) items $[\alpha \rightarrow \beta \bullet \gamma \delta, a b]$ (two token look-ahead) ...

LR(k) items

UC Santa Barbara

The \bullet in an item indicates the position of the top of the stack

$[\alpha \rightarrow \bullet \beta \gamma \delta, \mathbf{a}]$ means that the input seen so far is consistent with the use of $\alpha \rightarrow \beta \gamma \delta$ immediately after the symbol on top of the stack

$[\alpha \rightarrow \beta \gamma \bullet \delta, \mathbf{a}]$ means that the input seen so far is consistent with the use of $\alpha \rightarrow \beta \gamma \delta$ at this point in the parse, and that the parser has already recognized $\beta \gamma$.

$[\alpha \rightarrow \beta \gamma \delta \bullet, \mathbf{a}]$ means that the parser has seen $\beta \gamma \delta$, and the lookahead \mathbf{a} is consistent with reducing to α (for LR(k) parsers, \mathbf{a} is a string of terminal symbols of length k)

The table construction algorithm uses items to represent valid configurations of an LR(1) parser

LR(1) Items

UC Santa Barbara

The production $\alpha \rightarrow \bullet \beta \gamma \delta$, with lookahead **a**, generates 4 items

$[\alpha \rightarrow \bullet \beta \gamma \delta, \mathbf{a}]$, $[\alpha \rightarrow \beta \bullet \gamma \delta, \mathbf{a}]$, $[\alpha \rightarrow \beta \gamma \bullet \delta, \mathbf{a}]$, & $[\alpha \rightarrow \beta \gamma \delta \bullet, \mathbf{a}]$

The set of LR(1) items for a grammar is finite

What's the point of all these look-ahead symbols?

- Carry them along to choose correct reduction
- Look-ahead symbols are bookkeeping, *unless* item has \bullet at right end
 - Has no direct use in $[\alpha \rightarrow \beta \gamma \bullet \delta, \mathbf{a}]$
 - In $[\alpha \rightarrow \beta \gamma \delta \bullet, \mathbf{a}]$, a look-ahead of **a** implies a reduction by $\alpha \rightarrow \beta \gamma \delta$
 - For $\{ [\alpha \rightarrow \gamma \bullet, \mathbf{a}], [\beta \rightarrow \gamma \bullet \delta, \mathbf{b}] \}$

lookahead = **a** \Rightarrow *reduce* to α

lookahead $\in \text{FIRST}(\delta)$ \Rightarrow *shift*

\Rightarrow Limited right context is enough to pick the actions

Back to Finding Handles

UC Santa Barbara

Parser in a state where the stack (the fringe) was

Expr – *Term*

With look-ahead of *

How did it choose to expand *Term* rather than reduce to *Expr*?

- *Look-ahead* symbol is the key
- With look-ahead of + or –, parser should reduce to *Expr*
- With look-ahead of * or /, parser should shift
- Parser uses look-ahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism

Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	id - num * id \$	none	shift
\$ id	- num * id \$	9,1	red. 9
\$ Factor	- num * id \$	7,1	red. 7
\$ Term	- num * id \$	4,1	red. 4
\$ Expr	- num * id \$	none	shift
\$ Expr-	num * id \$	none	shift
\$ Expr- num	* id \$	8,3	red. 8
\$ Expr- Factor	* id \$	7,3	red. 7
\$ Expr- Term	* id \$	none	shift
\$ Expr- Term *	Id \$	none	shift
\$ Expr- Term * id	\$	9,5	red. 9
\$ Expr- Term * Factor	\$	5,5	red. 5
\$ Expr- Term	\$	3,3	red. 3
\$ Expr	\$	1,1	red. 1
\$ S	\$	none	accept

shift here

reduce here

1. Shift until TOS is the right end of a handle
2. Find the left end of the handle & reduce

LR(1) Table Construction

UC Santa Barbara

High-level overview

- 1 Build the handle recognizing DFA (aka *Canonical Collection* of sets of LR(1) items), $C = \{ I_0, I_1, \dots, I_n \}$
 - a) Introduce a new start symbol S' which has only one production
$$S' \rightarrow S$$
 - b) Initial state, I_0 should include
 - $[S' \rightarrow \bullet S, \$]$, along with any equivalent items
 - Derive equivalent items as $closure(I_0)$
 - c) Repeatedly compute, for each I_k , and each grammar symbol α , $goto(I_k, \alpha)$
 - If the set is not already in the collection, add it
 - Record all the transitions created by $goto()$This eventually reaches a fixed point
- 2 Fill in the ACTION and GOTO tables using the DFA

Computing Closures

UC Santa Barbara

$\text{closure}(I)$ adds all the items implied by items already in I

- Any item $[\alpha \rightarrow \beta \bullet A \delta, a]$ implies $[A \rightarrow \bullet \tau, x]$ for each production with A on the *lhs*, and $x \in \text{FIRST}(\delta a)$
- Since A is valid, any way to derive A is valid, too
- $\text{FIRST}(\delta a)$ tells us the set of things that could possibly come *after* this particular use of A (and would tell us the production to use)

The algorithm

```
Closure( I )
while ( I is still changing )
  for each item  $[\alpha \rightarrow \beta \bullet \gamma \delta, a] \in I$ 
    for each production  $\gamma \rightarrow \tau \in P$ 
      for each terminal  $b \in \text{FIRST}(\delta a)$ 
        if  $[\gamma \rightarrow \bullet \tau, b] \notin I$ 
          then add  $[\gamma \rightarrow \bullet \tau, b]$  to  $I$ 
```

Fixpoint computation

Example Grammar

UC Santa Barbara

Initial step builds the item $[S \rightarrow \cdot Z, \$]$
and takes its *closure*()

Closure($[S \rightarrow \cdot Z, \$]$)

1	S	→	Z
2	Z	→	Z z
3			z

<i>Item</i>	<i>From</i>
$[S \rightarrow \cdot Z, \$]$	Original item
$[Z \rightarrow \cdot Z z, \$]$	1, δ a is \$
$[Z \rightarrow \cdot z, \$]$	1, δ a is \$
$[Z \rightarrow \cdot Z z, z]$	2, δ a is z \$
$[Z \rightarrow \cdot z, z]$	2, δ a is z \$

So, initial state s_0 is

$\{ [S \rightarrow \cdot Z, \$], [Z \rightarrow \cdot Z z, \$], [Z \rightarrow \cdot z, \$], [Z \rightarrow \cdot Z z, z], [Z \rightarrow \cdot z, z] \}$

Computing Gotos

UC Santa Barbara

$goto(I, x)$ computes the state that the parser would reach if it recognized an x while in state I

- $goto(\{ [\alpha \rightarrow \beta \cdot \gamma \delta, a] \}, \gamma)$ produces $[\alpha \rightarrow \beta \gamma \cdot \delta, a]$
- It also includes $closure([\alpha \rightarrow \beta \gamma \cdot \delta, a])$ to fill out the state

The algorithm

```
Goto( I, x )  
  new =  $\emptyset$   
  for each  $[\alpha \rightarrow \beta \cdot x \delta, a] \in I$   
    new = new  $\cup$   $[\alpha \rightarrow \beta x \cdot \delta, a]$   
  return closure(new)
```

- Not a fixpoint method
- Uses closure

Example Grammar

UC Santa Barbara

s_0 is $\{ [S \rightarrow \cdot Z, \$], [Z \rightarrow \cdot Z z, \$], [Z \rightarrow \cdot z, \$], [Z \rightarrow \cdot Z z, z], [Z \rightarrow \cdot z, z] \}$

$goto(S_0, z)$

- Loop produces

<i>Item</i>	<i>From</i>
$[Z \rightarrow z \cdot, \$]$	Item 3 in s_0
$[Z \rightarrow z \cdot, z]$	Item 5 in s_0

- Closure adds nothing since \cdot is at end of *rhs* in each item

In the construction, this produces s_2

$\{ [Z \rightarrow z \cdot, \{ \$, z \}] \}$

New, but obvious, notation
for two distinct items
 $[Z \rightarrow z \cdot, \$]$ and $[Z \rightarrow z \cdot, z]$

Canonical Collection of LR(1) Items

UC Santa Barbara

This is where we build the handle recognizing DFA!

Start from $I_0 = \text{closure}([S' \rightarrow \cdot S, \$])$

Repeatedly construct new states, until no new states are generated

The algorithm

$I_0 = \text{closure}([S' \rightarrow \cdot S, \$])$

$C = \{ I_0 \}$

while (C is still changing)

 for each $I_i \in C$ and for each $x \in (T \cup NT)$

$I_{new} = \text{goto}(I_i, x)$

 if $I_{new} \notin C$ then

$C = C \cup I_{new}$

 record transition $I_i \rightarrow I_{new}$ on x

- Fixed-point computation
- Loop adds to C
- $C \subseteq 2^{\text{ITEMS}}$, so C is finite

Algorithms - Overview

UC Santa Barbara

Computing closure of set of LR(1) items:

```
Closure( I )
while ( I is still changing )
  for each item  $[\alpha \rightarrow \beta \cdot \gamma \delta, a] \in I$ 
    for each production  $\gamma \rightarrow \tau \in P$ 
      for each terminal  $b \in \text{FIRST}(\delta a)$ 
        if  $[\gamma \rightarrow \cdot \tau, b] \notin I$ 
          then add  $[\gamma \rightarrow \cdot \tau, b]$  to I
```

Computing goto for set of LR(1) items:

```
Goto( I, x )
new =  $\emptyset$ 
for each  $[\alpha \rightarrow \beta \cdot x \delta, a] \in I$ 
  new = new  $\cup$   $[\alpha \rightarrow \beta x \cdot \delta, a]$ 
return closure(new)
```

Constructing canonical collection of LR(1) items:

```
 $I_0 = \text{closure}([S' \rightarrow \cdot S, \$])$ 
 $C = \{ I_0 \}$ 
while ( C is still changing )
  for each  $I_i \in C$  and for each  $x \in (T \cup NT)$ 
     $I_{new} = \text{goto}(I_i, x)$ 
    if  $I_{new} \notin C$  then
       $C = C \cup I_{new}$ 
    record transition  $I_i \rightarrow I_{new}$  on x
```

- Canonical collection construction algorithm is the algorithm for constructing handle recognizing DFA
- Uses Closure to compute the states of the DFA
- Uses Goto to compute the transitions of the DFA

Practical Approach to LR(1) Parsing

UC Santa Barbara

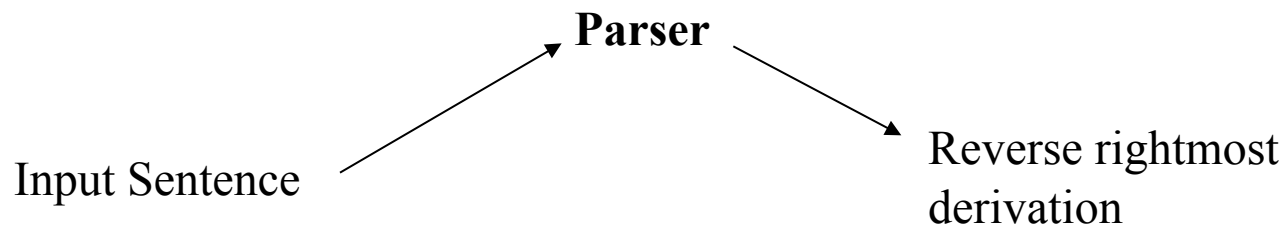
Start with
Grammar

→ Construct a DFA representing
all possible legal transition on
terminal and non-terminals.
Technically, this is a bunch of
NFAs grouped together via
e-closure

Canonical Collection

This is the DFA which
represents all valid transitions
through the grammar. We
need this for efficient **handle**
finding

↓
Use the CC to fill in the
LR tables (ACTION and
GOTO tables), which is
the way to program an
automated LR(1) parser

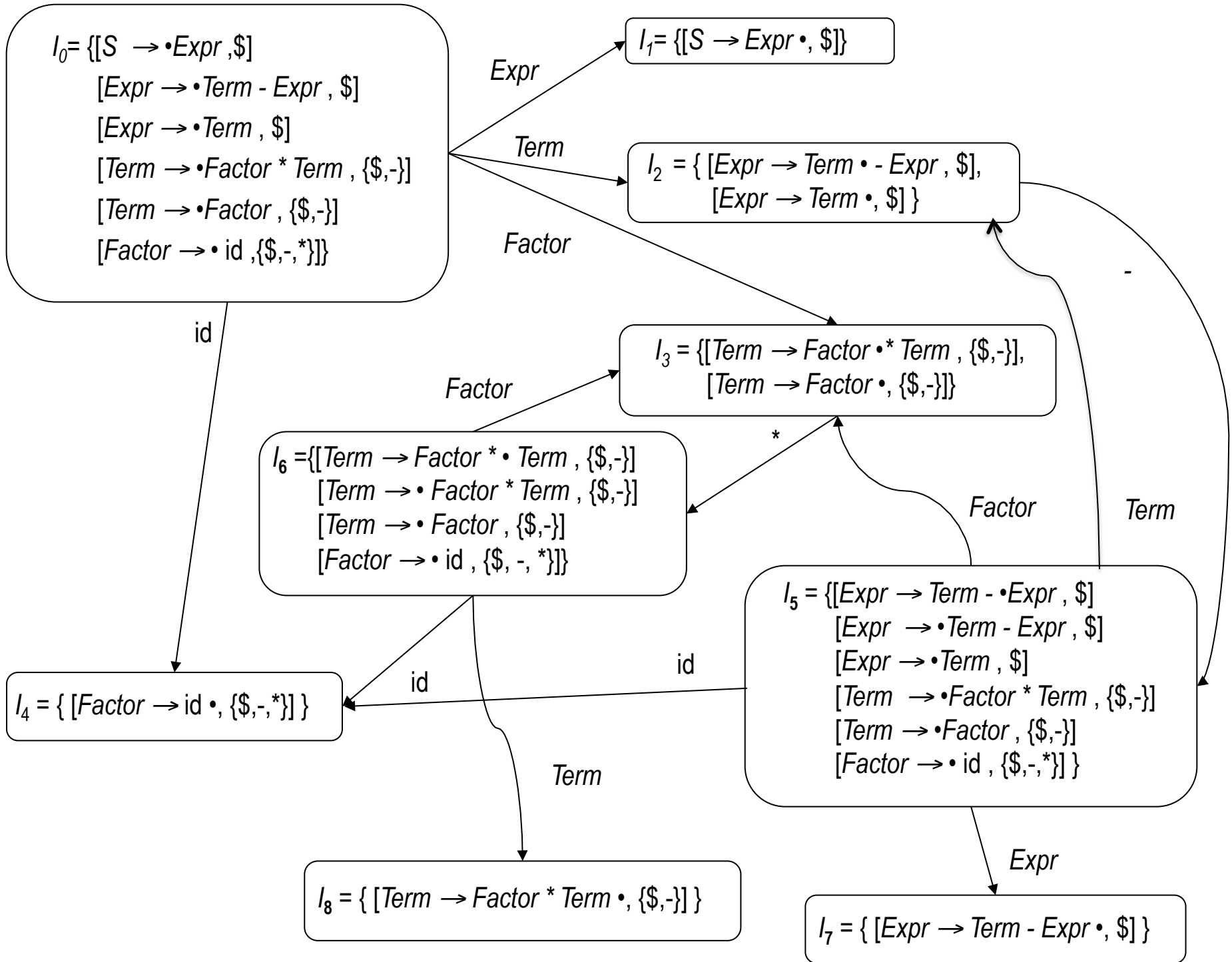


Example

Simplified, right recursive expression grammar

$S \rightarrow Expr$
 $Expr \rightarrow Term - Expr$
 $Expr \rightarrow Term$
 $Term \rightarrow Factor * Term$
 $Term \rightarrow Factor$
 $Factor \rightarrow id$

<i>Symbol</i>	FIRST
S	{ id }
Expr	{ id }
Term	{ id }
Factor	{ id }
-	{ - }
*	{ * }
id	{ id }



Constructing the ACTION and GOTO Tables

UC Santa Barbara

The algorithm

**x is the state number
Each state
corresponds to a set
of LR(1) items**

```
for each set of items  $I_x \in C$ 
  for each  $item \in I_x$ 
    if  $item$  is  $[\alpha \rightarrow \beta \cdot a \gamma, b]$  and  $a \in T$  and  $goto(I_x, a) = I_k$ ,
      then  $ACTION[x, a] \leftarrow$  "shift  $k$ "
    else if  $item$  is  $[S' \rightarrow S \cdot, \$]$ 
      then  $ACTION[x, \$] \leftarrow$  "accept"
    else if  $item$  is  $[\alpha \rightarrow \beta \cdot, a]$ 
      then  $ACTION[x, a] \leftarrow$  "reduce  $\alpha \rightarrow \beta$ "
  for each  $n \in NT$ 
    if  $goto(I_x, n) = I_k$ 
      then  $GOTO[x, n] \leftarrow k$ 
```

Example (Constructing the LR(1) tables)

The algorithm produces the following table

	ACTION				GOTO		
	id	-	*	\$	<i>Expr</i>	<i>Term</i>	<i>Factor</i>
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

Parsing Example (x-z*y)

UC Santa Barbara

Stack	Input	Action
S_0	$\$id * id - id$	S4
$S_0 id S_4$	$\$id * id -$	R6, G3
$S_0 F S_3$	$\$id * id -$	R5, G2
$S_0 T S_2$	$\$id * id -$	S5
$S_0 T S_2 - S_5$	$\$id * id$	S4
$S_0 T S_2 - S_5 id S_4$	$\$id *$	R6, G3
$S_0 T S_3 - S_5 F S_3$	$\$id *$	S6
$S_0 T S_3 - S_5 F S_3 * S_6$	$\$id$	S4
$S_0 T S_3 - S_5 F S_3 * S_6 id S_4$	$\$$	R6, G3
$S_0 T S_3 - S_5 F S_3 * S_6 F S_3$	$\$$	R5, G8
$S_0 T S_3 - S_5 F S_3 * S_6 T S_8$	$\$$	R4, G2
$S_0 T S_3 - S_5 T S_2$	$\$$	R3, G7
$S_0 T S_3 - S_5 E S_7$	$\$$	R2, G1
$S_0 E S_1$	$\$$	ACC

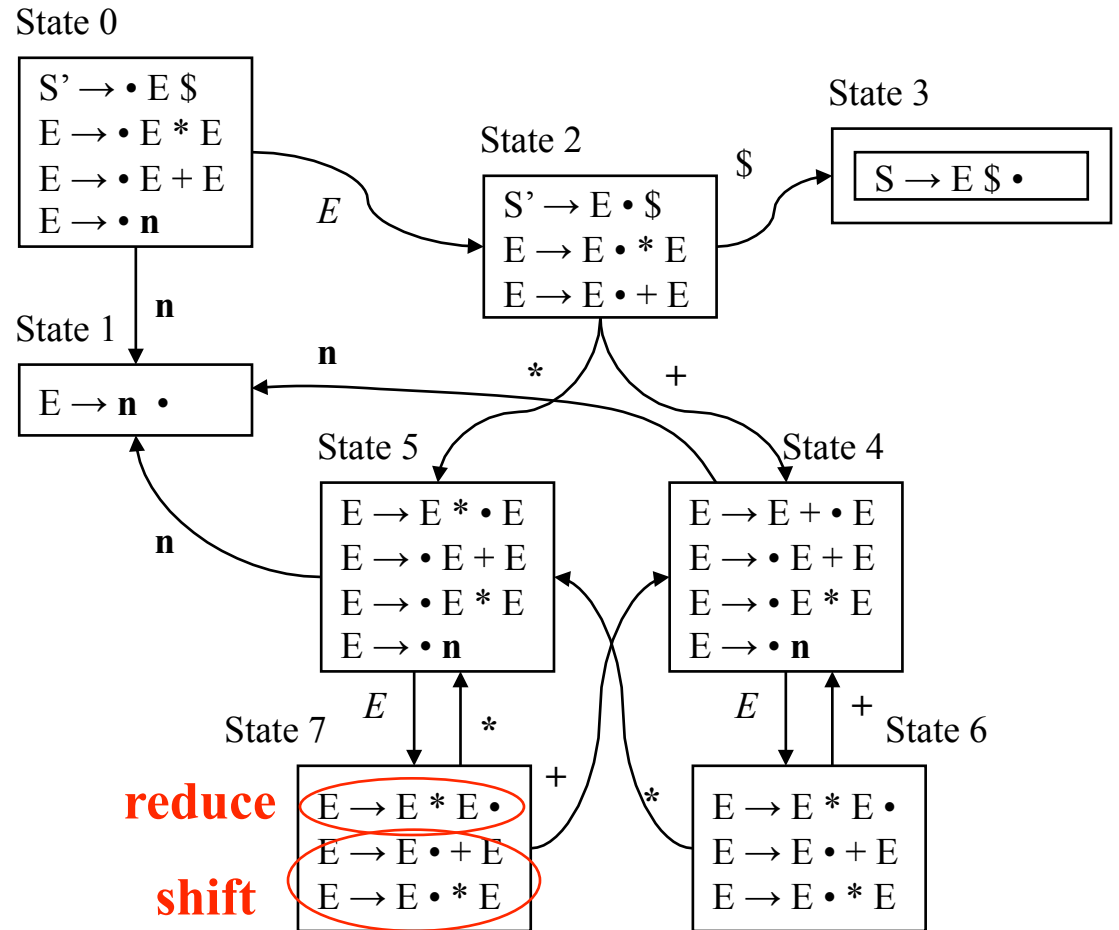
	ACTION				GOTO		
	id	-	*	\$	Expr	Term	Factor
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

1. $S \rightarrow Expr$
2. $Expr \rightarrow Term - Expr$
3. $Expr \rightarrow Term$
4. $Term \rightarrow Factor * Term$
5. $Term \rightarrow Factor$
6. $Factor \rightarrow id$

Conflicts and Associativity/Precedence

example	
0	$S' \rightarrow E \$$
1	$E \rightarrow E * E$
2	$E \rightarrow E + E$
3	$E \rightarrow n$

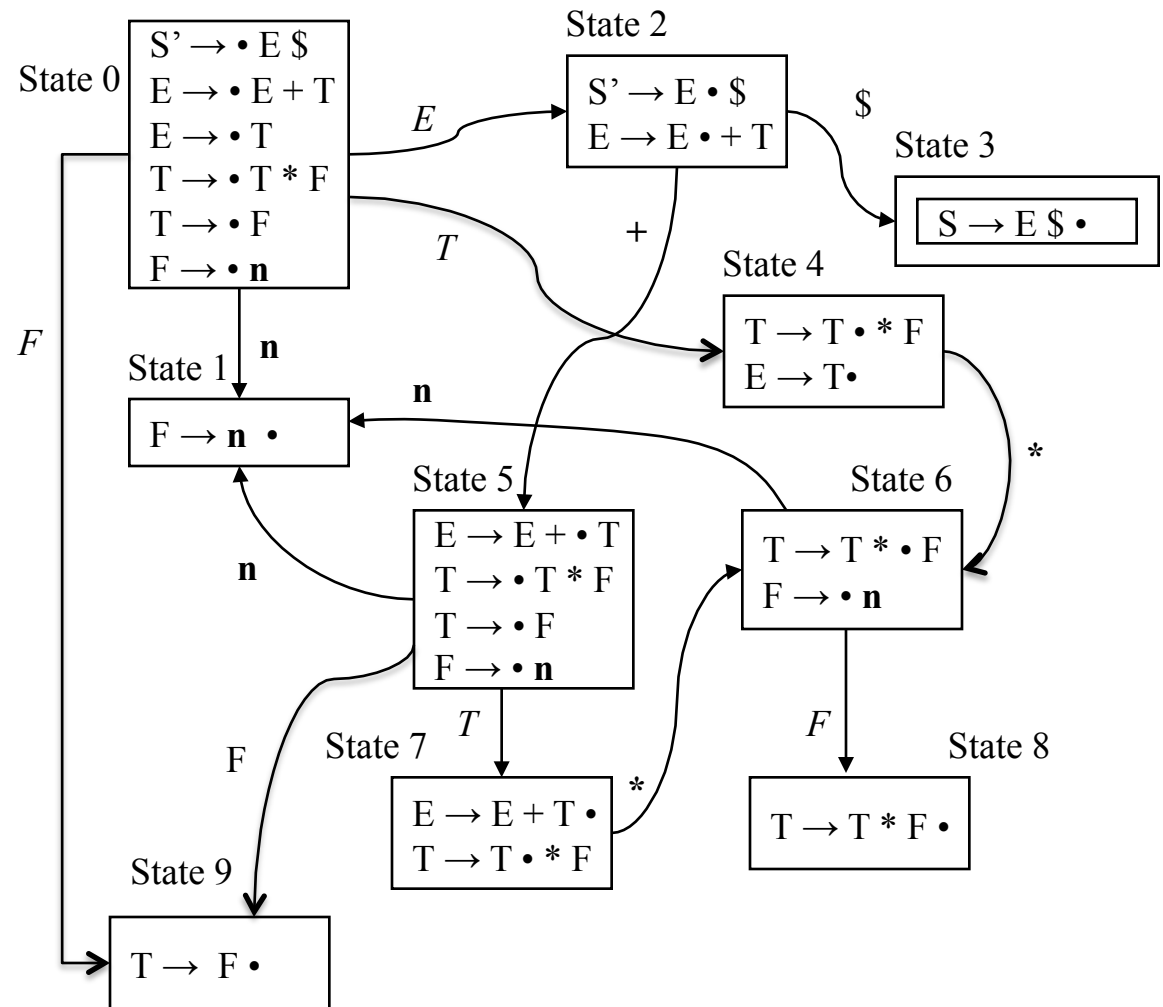
	*	+	\$	n	E
0				S1	2
1	R3	R3	R3		
2	S5	S4	S3		
3			A		
4				S1	6
5				S1	7
6	S5/R1	S4/R2	R2	R2	
7	S5/R1	S4/R1	R1	R1	



Conflicts and Associativity/Precedence

- example
- 0 $S' \rightarrow E \$$
 - 1 $E \rightarrow E + T$
 - 2 $E \rightarrow T$
 - 3 $T \rightarrow T * F$
 - 4 $T \rightarrow F$
 - 5 $F \rightarrow n$

	+	*	\$	n	E	T	F
0				S1	2	4	9
1	R5	R5	R5				
2	S5		S3				
3			A				
4	R2	S6	R2				
5				S1		7	9
6				S1			8
7	R1	S6	R1				
8	R3	R3	R3				
9	R4	R4	R4				



What can go wrong in LR(1) parsing?

UC Santa Barbara

What if state s contains $[\alpha \rightarrow \beta \cdot a\gamma, b]$ and $[\alpha \rightarrow \beta \cdot, a]$?

- First item generates “shift”, second generates “reduce”
- Both define $\text{ACTION}[s,a]$ — cannot do both actions
- This is called a *shift/reduce conflict*
- Modify the grammar to eliminate it
- Shifting will often resolve it correctly (dangling else problem?)

What if set s contains $[\alpha \rightarrow \beta \cdot, a]$ and $[\gamma \rightarrow \beta \cdot, a]$?

- Each generates “reduce”, but with a different production
- Both define $\text{ACTION}[s,a]$ — cannot do both reductions
- This is called a *reduce/reduce conflict*
- Modify the grammar to eliminate it

In either case, the grammar is not LR(1)

Error recovery

UC Santa Barbara

- **Panic-mode recovery:** On discovering an error, discard input symbols one at a time until one synchronizing token is found
 - For example delimiters such as “;” or “}” can be used as synchronizing tokens
- **Phrase-level recovery:** On discovering an error make local corrections to the input
 - For example replace “,” with “;”
- **Error-productions:** If we have a good idea about what type of errors occur, we can augment the grammar with error productions and generate appropriate error messages when an error production is used
- **Global correction:** Given an incorrect input string try to find a correct string which will require minimum changes to the input string
 - In general too costly

Direct Encoding of Parse Tables

UC Santa Barbara

Rather than using a table-driven interpreter ...

- Generate spaghetti code that implements the logic
- Each state becomes a small case statement or if-then-else
- Analogous to direct coding a scanner

Advantages

- No table lookups and address calculations
- No representation for don't care states
- No outer loop — it is implicit in the code for the states

This produces a faster parser with more code but no table

LR Parsers

UC Santa Barbara

- LR(k) parsers are table-driven, bottom-up, shift-reduce parsers that use a limited right context (k-token look-ahead) for handle recognition
- LR(k): Left-to-right scan of the input, rightmost derivation in reverse with k token look-ahead

A grammar is LR(k) if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

We can

1. *isolate the handle of each right-sentential form γ_i , 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

Summary

UC Santa Barbara

	<i>Advantages</i>	<i>Disadvantages</i>
Top-down recursive descent	Fast Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes