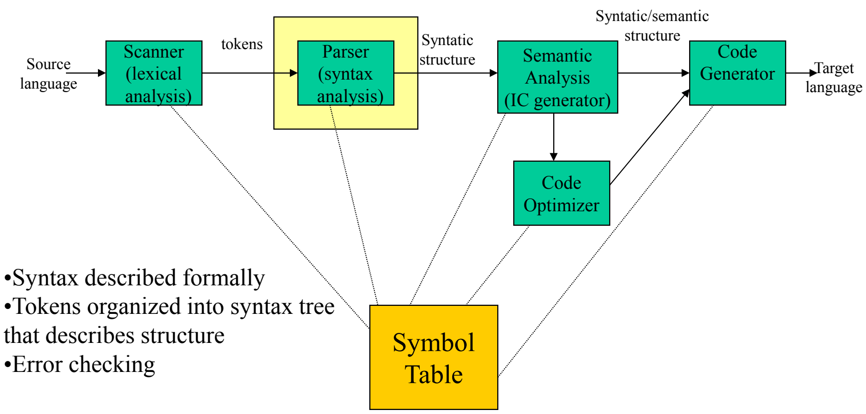
Lecture 4c: Bottom Up Parsing

COSC 4316

Static Analysis - Parsing



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking

Bottom-Up Parsing

- LR methods (Left-to-right, Rightmost derivation)
  - SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

3

Operator-Precedence Parsing

- Special case of shift-reduce parsing
- We may discuss this if time permits

4

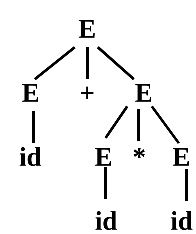
Shift-Reduce Parsing

- General Idea: Reducing a string *w* to the start symbol
- At each ***reduction*** step, a particular substring matching the RHS of a production is replaced by the symbol on the LHS
- A *rightmost derivation* is traced out *in reverse*

Example

- Consider the grammar E → E **+** E | E ***** E | **id** and the string **id + id * id**
- A rightmost derivation is

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \mathbf{id}$
$\Rightarrow E + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$



Example

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \mathbf{id}$
$\Rightarrow E + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

- Now, a bottom up parser is supposed to reconstruct this derivation by working backward from the token stream. Can we do this?
- We look at the string <u>**id**</u> **+ id * id**
- Assume the first id came from the production E → **id**, so we "*reduce*" the RHS to E + **id** * **id**
- The next token is "**+**" and we don't know what to do with it, so we "*shift*" ignoring it for now.
- When we get to the next **id**, we again reduce it to E. (E → **id**)

  E + E * **id**

Example

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \mathbf{id}$
$\Rightarrow E + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

E + E * **id**

- Now, we have a choice. We could reduce the E + E to E <u>*or*</u> we can shift, ignoring the *. Let's pretend we have an oracle that tells us to shift.
- So, now we are at the final **id**.
  We reduce **id** to E (E → **id**)
  E + E * E
- Now we see E * E and reduce that to E
  E + E
- And that reduces to the start symbol, E

Looking backwards:
id + **id** * **id**
E + **id** * **id**
E + E * **id**
E + E * E
E + E
E
a rightmost derivation in reverse

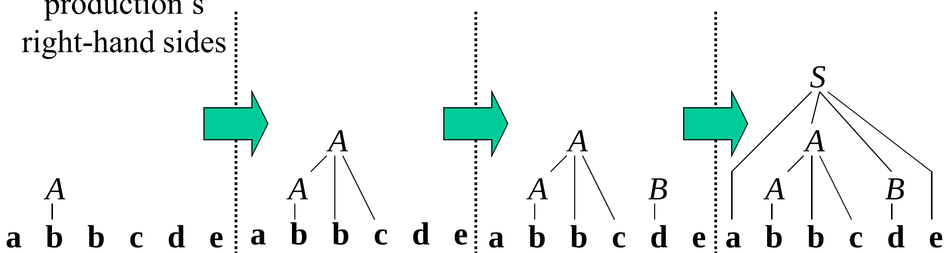
Shift-Reduce Parsing

Grammar:
$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$
$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$
$B \rightarrow \mathbf{d}$

These match production's right-hand sides

Reducing a sentence:
$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$
$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$
S

Shift-reduce corresponds to a rightmost derivation:
$S \Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$



9

Handles

- A ***handle*** is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:
$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$
$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$
$B \rightarrow \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$
$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$
S

Handle

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \underline{\mathbf{c}}\ \mathbf{d}\ \mathbf{e}$
$\mathbf{a}\ A\ A\ \mathbf{e}$
… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

10

Handles

- More formally, a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a derivation of γ.

- e.g., if $S \Rightarrow^{*}_{rm} \alpha\, A\, w \Rightarrow_{rm} \alpha\, \beta\, w$
  then $A \rightarrow \beta$ in a position following is a handle of $\alpha\, \beta\, w$

  $w \equiv$ a string of terminals
  $\alpha, \beta, \gamma \ \equiv$ strings of grammar symbols

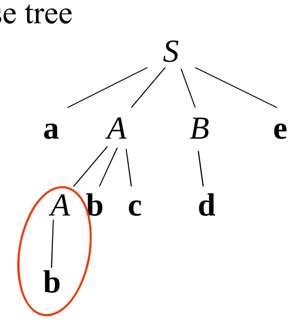
11

What a Handle represents

- A handle represents the leftmost complete subtree consisting of a node and all its children

Grammar:
$S \rightarrow$ **a** $A\ B$ **e**
$A \rightarrow A$ **b c** | **b**
$B \rightarrow$ **d**

Parse tree



Handle of **abbcde**

12

Handle Pruning

- ***Handle pruning*** – a means of obtaining a rightmost derivation in reverse
  - Start with a string of terminal *w*
  - $S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$
  - Locate handle β_n in γ_n and replace β_n by A where A → β_n is a production.
  - Repeat for n-1, n-2, …, 1 until S is reached

How do we locate a handle?

1. Shift zero or more input symbols on the stack until a handle β is on top of the stack
2. Reduce β to the left side of the appropriate production

Stack Implementation of Shift-Reduce Parsing

Grammar:
$E \to E + E$
$E \to E$ ***** E
$E \to$ **(** E **)**
$E \to$ **id**

Stack	Input	Action
$	**id+id*id$**	shift
$id	**+id*id$**	reduce $E \to$ **id**
**$**$E$	**+id*id$**	shift
$**$E$+**	**id*id$**	shift
$**$E$+id**	***id$**	reduce $E \to$ **id**
**$**$E$**+**$E$	***id$**	shift (or reduce?)
$**$E$**+**$E$***	**id$**	shift
$**$E$**+**$E$*id**	**$**	reduce $E \to$ **id**
**$**$E$**+**$E$*****$E$	**$**	reduce $E \to E$ ***** E
**$**$E$**+**$E$	**$**	reduce $E \to E$ **+** E
**$**$E$	**$**	accept

How to resolve conflicts?

Find handles to reduce

15

Conflicts

- *Shift-reduce* and *reduce-reduce* conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar
    - Shift-reduce usually fixable
    - Reduce-reduce is usually the result of an ambiguous grammar

16

LL vs. LR

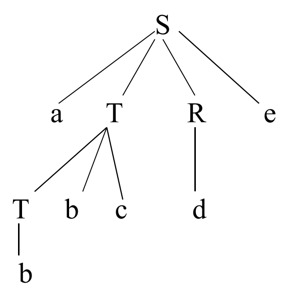
- LR (shift reduce) is more powerful than LL (predictive parsing)
- Can detect a syntactic error as soon as possible.
- LR is difficult to do by hand (unlike LL)

LR(*k*) Parsing – Bottom Up

- Construct parse tree from leaves, ‘reducing’ the string to the start symbol (and a single tree)
- During parse, we have a ‘forest’ of trees
- Shift-reduce parsing
  - **‘Shift’ a new input symbol**
  - **‘Reduce’ a group of symbols to a single non-terminal**
  - Choice is made using the *k* lookaheads
- LR(1)

Example

- S → a T R e
  T → T b c | b
  R → d



- Rightmost derivation:

  S → a T R e
  → a T **d** e
  → a **T** **b** **c** d e
  → a **b** b c d e

LR parsing corresponds to the rightmost derivation in reverse.

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: abbcde

Rightmost derivation:
S → a T R e
→ a T **d** e
→ a **T** **b** **c** d e
→ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

→ Shift a, Shift b

a b

Rightmost derivation:
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

→ Shift a, Shift b
→ Reduce T → b

T
|
a b

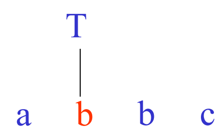
Rightmost derivation:
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: de

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c



Rightmost derivation:

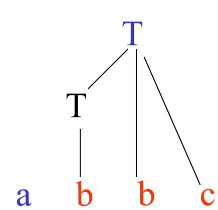
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: de

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c



Rightmost derivation:

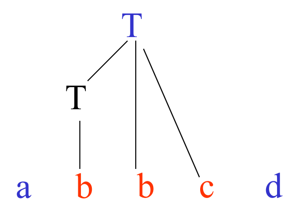
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d



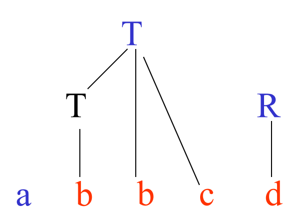
Rightmost derivation:
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d
- ➔ Reduce R → d



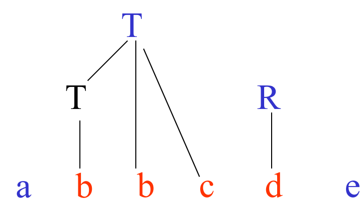
Rightmost derivation:
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input:

- Shift a, Shift b
- Reduce T → b
- Shift b, Shift c
- Reduce T → T b c
- Shift d
- Reduce R → d
- Shift e



Rightmost derivation:

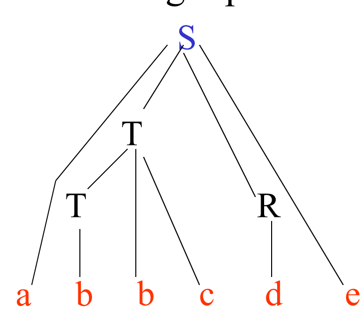
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input:

- Shift a, Shift b
- Reduce T → b
- Shift b, Shift c
- Reduce T → T b c
- Shift d
- Reduce R → d
- Shift e
- Reduce S → a T R e



Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

LR Parsing

- First described by Donald Knuth (1965)
- Major advantage:
  - Most grammars can be parsed by an LR parser without altering the grammar
- LR – reads the input left to right and maintains a stack
- The parser's stack contains ***states*** instead of symbols
  - The function of a state is to provide a coded indications of all the information contained in the stack below it.

LR Parsing

- The parser's parsing table contains ***actions***. The table is indexed by a state number and input token (i.e., M[s,**a**])
- Each entry M[s,**a**] can contain any of four types of values
  - ***shift***: the current input symbol **a** is discarded and the state number specified in M[s,**a**] is pushed on the stack
  - ***reduce*:**
    - M[s,**a**] identifies a production to be applied
    - # of states to be popped off the stack = # symbols on RHS of production
    - Put symbol changed to LHS of production.
  - ***accept***: Parsing completed successfully
  - ***error***: Syntax error detected

Example LR Table

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

Action table
Goto table

s means shift to to some state

r means reduce by some production

Algorithm: LR(1)

```
push($,0);   /* always pushing a symbol/state pair */
lookahead = yylex();
loop
   s = top(); /*always a state */
   if action[s,lookahead] = shift s'
      push(lookahead,s');  lookahead = yylex();
   else if action[s,lookahead] = reduce A → β
      pop size of β pairs
      s' = state on top of stack
      push(A,goto[s',A]);
  else if action[s,lookahead] = accept then return
  else error();
end loop;
```

In practice, the state-symbol pairs on the stack are usually just represented by the states themselves – the grammar symbol information is encoded into the state.

LR Parsing Example 1

Stack	Input	Action
\$0	a b b c d e \$	s1

State	a	b	c	d	e	\$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parsing Example 1

Stack	Input	Action
\$0	a b b c d e \$	s1
\$0,a1	b b c d e \$	s3

State	a	b	c	d	e	\$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)

goto(T,1)=2

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)
$0,a1,T2	d e $	s6
$0,a1,T2,d6	e $	r4 (R → d)

goto(T,1)=2

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parsing Example 1

Stack	Input	Action
$0	a b b c d e $	s1
$0,a1	b b c d e $	s3
$0,a1,b3	b c d e $	r3 (T → b)
$0,a1,T2	b c d e $	s5
$0,a1,T2,b5	c d e $	s8
$0,a1,T2,b5,c8	d e $	r2 (T → T b c)
$0,a1,T2	d e $	s6
$0,a1,T2,d6	e $	r4 (R → d)
$0,a1,T2,R4	e $	s7
$0,a1,T2,R4,e7	$	accept!

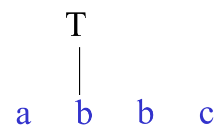
goto(R,2)=4

State	a	b	c	d	e	$	S	T	R
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

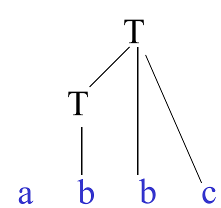
1: S → a T R e
2: T → T b c
3: T → b
4: R → d

LR Parse Stack

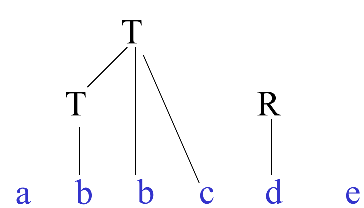
- During LR parsing, there is always a ‘forest’ of trees.
- Parse stack holds root of each of these trees:
  - For example, that stack $0,a1,T2,b5,c8 represents the corresponding forest



The next stack: $0,a1,T2



Later, we have $0,a1,T2,R6,e7



LR Parsing Example 2

Grammar:
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to \mathbf{(}\ E\ \mathbf{)}$
6. $F \to \mathbf{id}$

	action						*goto*		
state	**id**	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Shift & goto 5

Reduce by production #1

41

LR Parsing Example 2

	action						*goto*		
state	**id**	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 ***** 7	**id+id$**	shift 5
$ 0 *T* 2 ***** 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 ***** 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

Grammar:
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to \mathbf{(}\ E\ \mathbf{)}$
6. $F \to \mathbf{id}$

42

Where does the table come from?

- Handle – "a substring that matches the right side of a production and whose reduction to the non-terminal represents one step along the reverse of a rightmost derivation"
- Using the grammar, want to create a DFA to find handles.

SLR parsing

- Simplest LR algorithm
- Provide an understanding of
  - the basic mechanics of shift/reduce parsing
  - source of shift/reduce and reduce/reduce conflicts
- There are better (more powerful) algorithms (LALR, LR) but we won't study them in as much detail here.

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
  - These can show how far a parse has progressed
- Central problem of shift/reduce parsers is when to shift and when to reduce
- In a ***reduce*** step, the symbols on top of the stack correspond to the RHS of a production.
  - We refer to the symbols below as a ***prefix*** of the LHS symbol

45

SLR Parsing

- A ***viable prefix*** of a sentential form is a prfix which does not extend beyond the handle.
- Viable prefixes form a regular language(!), so we can construct a DFA to recognize them.
- Overview:
  - Build the LR(0) DFA by
    - *Closure operation* to construct LR(0) items
    - *Goto operation* to determine transitions
  - Construct the SLR parsing table from the DFA
  - LR parser program uses the SLR parsing table to determine shift/reduce operations

46

Generating SLR parse tables

- The first step is the construction of an ***augmented grammar***
- Augmented grammar: grammar with new start symbol and production S’ → S where S is old start symbol.
  - Augmentation only required if there is no single production to signal the end.
- When the parser attempts to reduce S’ → S it knows that the parse has succeeded. (S’ never occurs on the RHS)

Original Grammar:
$E \rightarrow E + T \mid T$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid$ **id**

Augmented Grammar:
$S' \rightarrow E$
$E \rightarrow E + T \mid T$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid$ **id**

LR(0) items

- Canonical LR(0) collections are the basis for constructing SLR (simple LR) parsers
- **Defn**: LR(0) item of a grammar G is a production of G with a dot at some point on the right side.
- A → X Y Z yields <u>*four different*</u> LR(0) items:
  - [A → . X Y Z]
  - [A → X . Y Z]
  - [A → X Y . Z]
  - [A → X Y Z .]
- A → ε yields one item
  - [A → .]

Define 2 operations

1. ***closure***(I)
2. ***goto***(I,*X*)

Closure(I) function

Closure(I) where I is a set of LR(0) items =

– Every item in I (*kernel*) and
– If [A → α . B β] is in closure(I) and [B → γ] is a production, add [B → . γ] to closure(I) (if not already there).
– Keep applying this rule until no more items can be added.

Closure Example

E' → E

E → E + T | T

T → T * F | F

F → (E) | id

Closure({[E' → · E]}) = {[E' → · E] , [E → · E + T], [E → · T,]
[T → · T * F], [T → · F],
[F → · (E)], [E → · id]}

The Closure Operation (Example)

closure({[*E*' → •*E*]}) =

{ [*E*' → • *E*] }

{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*] }

Add [*E*→•γ]

{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*] }

Add [*T*→•γ]

{ [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*]
[*F* → • **(** *E* **)**]
[*F* → • **id**] }

Add [*F*→•γ]

Grammar:
E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → **(** *E* **)**
F → **id**

Closure Example

E’ → E

E → E + T | T

T → T * F | F

F → (E) | **id**

Closure({[T→ T * . F]}) = {[T → T * . F], [F → . (E)], [F → . **id**]}

Closure({[E → E . + T], [F → . **id**]}) = {[E → E . + T], [F → . **id**]}

Closure Example

E’ → E

E → E + T | T

T → T * F | F

F → (E) | id

Closure({[F → (. E)]}

= {[F → (. E)], [E → . E + T], [E → . T]}

= {[F → (. E)], [E → . E + T], [E → . T], [T → . T * F], [T → . F]}

= {[F → (. E)], [E → . E + T], [E → . T], [T → . T * F], [T → . F], [F → . id], [F → . (E)]}

Goto function

goto(I,*X*), where I is a set of items and *X* is a grammar symbol, is the *closure*([A → α *X*•B]) where [A → α•*X* β] ∈ I.

The set goto(I,X) is called the ***successor*** of I under the symbol X

1. For each item $[A \to \alpha \bullet X \beta] \in I$, add the set of items *closure*($\{[A \to \alpha X \bullet \beta]\}$) to *goto*($I$,$X$) if not already there
2. Repeat step 1 until no more items can be added to *goto*(I,X)
3. Intuitively, *goto*(I,X) is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

The Goto Operation (Example 1)

Suppose I = { [$E' \to \bullet\, E$]
[$E \to \bullet\, E + T$]
[$E \to \bullet\, T$]
[$T \to \bullet\, T$ * F]
[$T \to \bullet\, F$]
[$F \to \bullet$ **(** E **)**]
[$F \to \bullet$ **id**] }

Then *goto*(I,E)
= *closure*({[$E' \to E\, \bullet$, $E \to E \bullet + T$]})
= { [$E' \to E\, \bullet$]
[$E \to E \bullet + T$] }

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to$ **(** E **)**
$F \to$ **id**

The Goto Operation (Example 2)

Suppose $I = \{ [E' \to E \bullet], [E \to E \bullet + T] \}$

Then $goto(I,+) = closure(\{[E \to E + \bullet T]\}) = \{ [E \to E + \bullet T]$
$[T \to \bullet T * F]$
$[T \to \bullet F]$
$[F \to \bullet \mathbf{(}\ E\ \mathbf{)}]$
$[F \to \bullet \mathbf{id}] \}$

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to \mathbf{(}\ E\ \mathbf{)}$
$F \to \mathbf{id}$

57

Algorithm to construct collection $C = \{I_0, I_1, \ldots, I_n\}$ for augmented grammar G'

<u>algorithm</u> items(G')
<u>begin</u>
    $I_0 \leftarrow$ closure([S' → • S]}
    $C \leftarrow \{I_0\}$
    <u>repeat</u>
        <u>for</u> each set of items $I_i \in C$ <u>and</u> each grammar symbol X <u>do</u>
            <u>if</u> goto(I_i,X) $\neq \phi$ <u>and</u> goto(I_i,X) $\notin C$ <u>then</u>
                add goto(I_i,X) to C
            <u>endif</u>
        <u>endfor</u>
    <u>until</u> no more sets of items can be added to C
<u>end</u>

Exercise

- Construct canonical set of items I_0 .. I_8 for grammar

$S' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow (\ E\ )$
$T \rightarrow \textbf{id}$

```
algorithm items(G')
begin
  I0 ← closure([S' → • S])
  C ← {I0}
  repeat
    for each set of items Ii ∈ C and each grammar symbol X do
      if goto(Ii,X) ≠ ϕ and goto(Ii,X) ∉ C then
        add goto(Ii,X) to C
      endif
    endfor
  until no more sets of items can be added to C
end
```

(hey Doc, look at your old faded notes – p. 90)

LR(0) items

$S' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow (\ E\ ) \mid \textbf{id}$

I_0: $S' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot (\ E\ )$
$T \rightarrow \cdot \textbf{id}$

goto(I_0,E) = I_1 :
$S' \rightarrow E \cdot$
$E \rightarrow E \cdot + T$

goto(I_0,T) = I_2 :
$E \rightarrow T \cdot$

goto(I_0, **(**) = I_3 :
$T \rightarrow (\ \cdot E\ )$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot (\ E\ )$
$T \rightarrow \cdot \textbf{id}$

goto(I_0,**id**) = I_4 :
$T \rightarrow \textbf{id} \cdot$

goto(I_1, **+**) = I_5 :
$E \rightarrow E + \cdot T$
$T \rightarrow \cdot (\ E\ )$
$T \rightarrow \cdot \textbf{id}$

goto(I_3,E) = I_6 :
$T \rightarrow (\ E \cdot)$
$E \rightarrow E \cdot + T$

goto(I_3, T) = I_2

goto(I_3, **(**) = I_3

goto(I_3, **id**) = I_4

goto(I_5, T) = I_7 :
$E \rightarrow E + T \cdot$

goto(I_5, **(**) = I_3

goto(I_5, **id**) = I_4

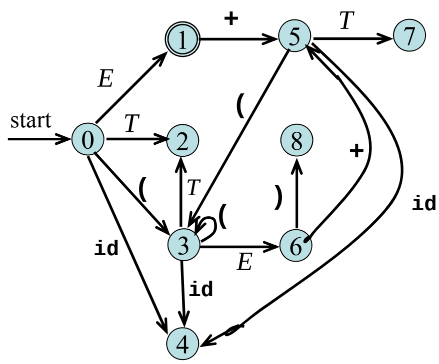
goto(I_6, **)**) = I_8 :
$T \rightarrow (\ E\ ) \cdot$

goto(I_6, **+**) = I_5

```
algorithm items(G')
begin
  I0 ← closure([S' → • S])
  C ← {I0}
  repeat
    for each set of items Ii ∈ C and each grammar symbol X do
      if goto(Ii,X) ≠ ϕ and goto(Ii,X) ∉ C then
        add goto(Ii,X) to C
      endif
    endfor
  until no more sets of items can be added to C
end
```

DFA for the ***goto***'s

- Recall that the viable prefixes form a regular language, so we can construct a DFA to recognize them.



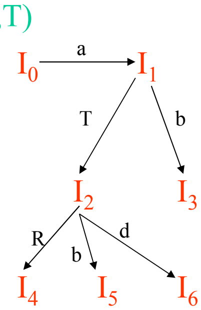
Example 1

Grammar: S → a T R e, T → T b c | b, R → d

I_0: S → . a T R e Goto({S → . a T R e },a) = I_1

I_1: S → a . T R e Goto({S → a . T R e , T → . T b c},T)
 T → . T b c = I_2
 T → . b Goto({T → . b },b) = I_3

I_2: S → a T . R e goto 4
 T → T . b c goto 5
 R → . d goto 6



kernel of each item set is in blue

Example 1

Grammar: S → a T R e, T → T b c | b, R → d

I_3: T → b . reduce

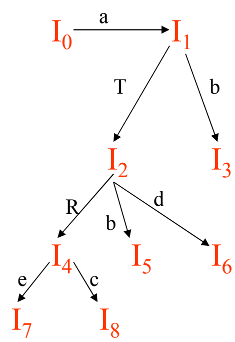
I_4: S → a T R . e goto state 7

I_5: T → T b . c goto state 8

I_6: R → d . reduce

I_7: S → a T R e . reduce

I_8: T → T b c . reduce



Algorithm for canonical sets restated for implementation

state = 0; max_state = 1;
kernel[0] = [S' → . S]
loop
  c = closure(kernel[state]);
  for t in c, where all productions are form A → α . B β
    if exists k <= state where t = kernel[k] then goto(state,B) = k;
    else
      kernel[max_state] = goto(state,B) = t;
      max_state++;
  state++;
until state+1 = max_state;

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_0: S' → . S
S → . A S
S → . b
A → . S A
A → . c

I_1: S' → S .
A → S . A
A → . S A
A → . c
S → . A S
S → . b

Example 2

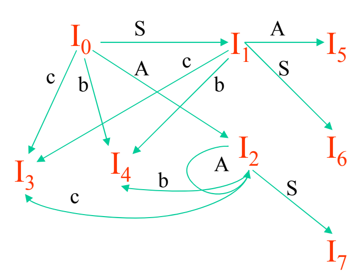
Grammar: S' → S, S → A S | b, A → S A | c

I_2: S → A . S
S → . A S
S → . b
A → . S A
A → . c

I_3: A → c .

I_4: S → b .

So far:



Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_5: S → A . S
A → S A .
S → . A S
S → . b
A → . S A
A → . c

I_6: A → S . A
A → . S A
A → . c
S → . A S
S → . b

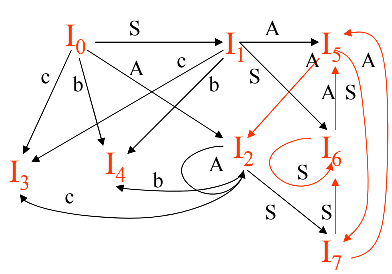
I_7: S → A S .
A → S . A
A → . S A
A → . c
S → . A S
S → . b

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_0: S' → . S
I_1: S' → S .
A → S . A
I_2: S → A . S
I_3: A → c .
I_4: S → b .
I_5: S → A . S
A → S A .
I_6: A → S . A
I_7: S → A S .
A → S . A

So far:



I5—I7 also have connections to I3 and I4

Generating SLR parse tables

- Construct C = {…} the LR(0) items as in previous slides
- Action table for state i of parser:
  - If $[A \rightarrow \alpha \cdot \mathbf{a}\ \beta] \in I_i$ and goto(I_i,**a**) = I_j then
    action[i,a] = sj
  - If $[A \rightarrow \alpha \cdot] \in I_i$, where $A \neq S'$, then
    action[i,**a**] = rn (where $A \rightarrow \alpha$ is production n) for all $\mathbf{a} \in$ FOLLOW(A)
  - If $[S' \rightarrow S, \$] \in I_i$, action[i,$] = *accept*

  All undefined entries are *error*
- Goto Table for state i of parser:
  - If $[A \rightarrow \alpha\ .\ B] \in I_i$ and goto(I_i,B) = I_j then
    goto[i,B] = j

Exercise

- Construct SLR parse table for

0. $S' \rightarrow E$
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow \mathbf{(}\ E\ \mathbf{)}$
4. $T \rightarrow \mathbf{id}$

FIRST(S') = { **id**, **(**}
FIRST(E) = { **id**, **(**}
FIRST(T') = { **id**, **(**}
FOLLOW(S') = { **$**}
FOLLOW(E) = { **)**, **+**, **$**}
FOLLOW(T) = { **)**, **+**, **$**}

I_0 : $S' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot \mathbf{(}\ E\ \mathbf{)}$
$T \rightarrow \cdot \mathbf{id}$

goto(I_0,E) = I_1 :
$S' \rightarrow E \cdot$
$E \rightarrow E \cdot + T$

goto(I_0,T) = I_2 :
$E \rightarrow T \cdot$

goto(I_0, **(**) = I_3 :
$T \rightarrow \mathbf{(} \cdot E\ \mathbf{)}$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot \mathbf{(}\ E\ \mathbf{)}$
$T \rightarrow \cdot \mathbf{id}$

goto(I_0,**id**) = I_4 :
$T \rightarrow \mathbf{id} \cdot$

goto(I_1,**+**) = I_5 :
$E \rightarrow E + \cdot T$
$T \rightarrow \cdot \mathbf{(}\ E\ \mathbf{)}$
$T \rightarrow \cdot \mathbf{id}$

goto(I_3,E) = I_6 :
$T \rightarrow \mathbf{(}\ E \cdot \mathbf{)}$
$E \rightarrow E \cdot + T$

goto(I_3, T) = I_2

goto(I_3, **(**) = I_3

goto(I_3, **id**) = I_4

goto(I_5, T) = I_7 :
$E \rightarrow E + T \cdot$

goto(I_5, **(**) = I_3

goto(I_5, **id**) = I_4

goto(I_6, **)**) = I_8 :
$T \rightarrow \mathbf{(}\ E\ \mathbf{)} \cdot$

goto(I_6,**+**) = I_5

	action					goto	
State	**+**	**(**	**)**	**id**	**$**	**E**	**T**
0		s3		s4		1	2
1	s5				*accept*		
2	r2		r2		r2		
3		s3		s4		6	2
4	r4		r4		r4		
5		s3		s4			7
6	s5		s8				
7	r1		r1		r1		
8	r3		r3		r3		

Parsing (id+id)+id

0. $S' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to (E)$
4. $T \to$ id

Stack	Input	Action
0	(id + id) + id $	s3
0 3	id + id) + id $	s4
0 3 4	+ id) + id $	r4 (T → id)
0 3	T + id) + id $	*goto* 2
0 3 2	+ id) + id $	r2 (E → T)
0 3	E + id) + id $	*goto* 6
0 3 6	+ id) + id $	s5
0 3 6 5	id) + id $	s4
0 3 6 5 4	) + id $	r4 (T → id)
0 3 6 5	T) + id $	*goto* 7
0 3 6 5 7	) + id $	r1 (E → E + T)
0 3	E) + id $	*goto* 6
0 3 6	) + id $	s8

	action					goto	
State	+	(	)	id	$	E	T
0		s3		s4		1	2
1	s5				*accept*		
2	r2		r2		r2		
3		s3		s4		6	2
4	r4		r4		r4		
5		s3		s4			7
6	s5		s8				
7	r1		r1		r1		
8	r3		r3		r3		

Parsing (id+id)+id

0. $S' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to (E)$
4. $T \to$ id

Stack	Input	Action
0 3 6	) + id $	s8
0 3 6 8	+ id $	r1 (T → (E))
0	T + id $	*goto* 2
0 2	+ id $	r2 (E → T)
0	E + id $	*goto* 1
0 1	+ id $	s5
0 1 5	id $	s4
0 1 5 4	$	r4 (T → id)
0 1 5	T $	*goto* 7
0 1 5 7	$	r1 (E → E + T)
0	E $	*goto* 1
0 1	$	*accept*

	action					goto	
State	+	(	)	id	$	E	T
0		s3		s4		1	2
1	s5				*accept*		
2	r2		r2		r2		
3		s3		s4		6	2
4	r4		r4		r4		
5		s3		s4			7
6	s5		s8				
7	r1		r1		r1		
8	r3		r3		r3		

Example 2

Some grammars cannot be parsed using SLR techniques:

Grammar: S' → S, S → A S | b, A → S A | c

	First	Follow
S'	c b	$
S	c b	$ c b
A	c b	c b

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_0: S' → . S goto 1
S → . A S goto 2
S → . b goto 3
A → . S A goto 1
A → . c goto 4
I_1: S' → S . reduce
A → S . A goto 5
A → . S A goto 6
A → . c goto 4
S → . A S goto 5
S → . b goto 3

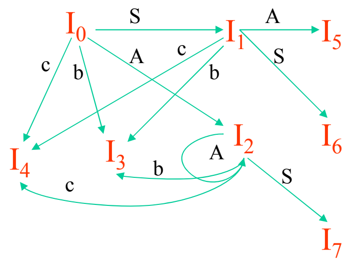
State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2					
3					
4					
5					
6					
7					
8					

Example 2

Grammar: S’ → S, S → A S | b, A → S A | c

I_2: S → A . S
S → . A S
S → . b
A → . S A
A → . A
I_3: S → b .
I_4: A → c .

So far:



LR Table for Example 2

State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2	s4	s3		7	2
3	r3	r3	r3		
4	r5	r5			
5					
6					
7					
8					

1: S’ → S
2: S → A S
3: S → b
4: A → S A
5: A→ c

Example 2

Grammar: S' → S, S → A S | b, A → S A | c

I_5: S → A . S
A → S A .
S → . A S
S → . b
A → . S A
A → . c

I_6: A → S . A
A → . S A
A → . c
S → . A S
S → . b

I_7: S → A S .
A → S . A
A → . S A
A → . c
S → . A S
S → . b

LR Table for Example 2

I "cheated", and let **yacc** generate the LR(0) items and the parse table

State	c	b	$	S	A
0	s4	s3		1	2
1	s4	s3	acc	6	5
2	s4	s3		7	2
3	r3	r3	r3		
4	r5	r5			
5	s4/r4	s3/r4		7	2
6	s4	s3		6	5
7	s4/r2	s3/r2	r2	6	5

Shift/reduce conflicts

1: S' → S
2: S → A S
3: S → b
4: A → S A
5: A → c

LR Conflicts

- Shift/reduce
  - When it cannot be determined whether to shift the next symbol or reduce by a production
  - Typically, the default is to shift.
  - Examples: previous grammar, dangling else

if_stmt → if expr then stmt | if expr then stmt else stmt

if ex1 then
if ex2 then
stmt;
else ← which '*if*' owns this *else*??

LR Conflicts

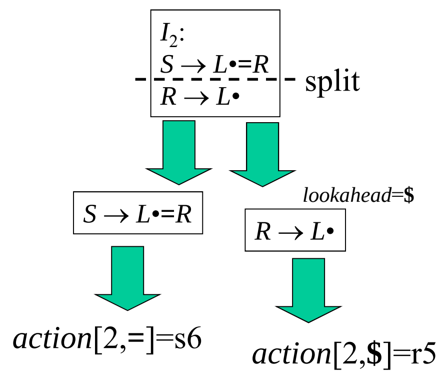
- Reduce/reduce
  - When it cannot be determined which production to reduce by
  - Example:

stmt → id (expr_list) ← function call
expr → id (expr_list) ← array (as in Ada)

  - Convention: use first production in grammar or use more powerful technique

SLR Versus LR(1)

- Split the SLR states by adding LR(1) lookahead
- Unambiguous grammar
  1. $S \to L = R$
  2. $\quad\mid R$
  3. $L \to$ ***** R
  4. $\quad\mid$ **id**
  5. $R \to L$



Should not reduce on =, because no right-sentential form begins with ***R*** =

81

LR(1) Items

- An *LR*(1) *item*
  $[A\to\alpha\bullet\beta, a]$
  contains a *lookahead* terminal a, meaning α already on top of the stack, expect to see βa
- For items of the form
  $[A\to\alpha\bullet, a]$
  the lookahead a is used to reduce $A\to\alpha$ only if the next input is a
- For items of the form
  $[A\to\alpha\bullet\beta, a]$
  with $\beta\neq\varepsilon$ the lookahead has no effect

82

LALR(1) Grammars

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

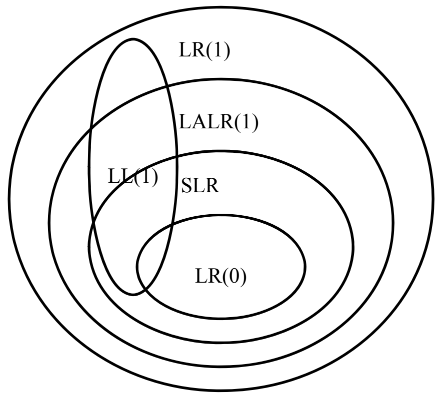
83

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals $\times$ terminals $\rightarrow$ productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states $\times$ terminals $\rightarrow$ shift/reduce actions
  - LR states $\times$ nonterminals $\rightarrow$ goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

84

LL, SLR, LR, LALR Grammars



85

Error Recovery in LR Parsing

- Panic mode
  - Pop until state with a goto on a nonterminal *A* is found, (where *A* represents a major programming construct), push *A*
  - Discard input symbols until one is found in the FOLLOW set of *A*
- Phrase-level recovery
  - Implement error routines for every error entry in table
- Error productions
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue

86