Compiler Design

Lecture-10

State transition and Shift-Reduce Conflicts

Topics Covered

State Transitions

Building Table of States & Transitions

Shift/Reduce Conflicts

State Transitions

Given set of items, compute new state(s) for each symbol (terminal and non-terminal) after dot

- state transitions correspond to shift actions

New item derived from old item by shifting dot over symbol

- do closure to compute new state Initial state (1):
  `S′ ::= . S $ S ::= . beep S ::= .{ L }`
- State (2) reached on transition that shifts `S`**:**
  `S′ ::= S . $`
- State (3) reached on transition that shifts **beep**:
  `S ::= beep .`
- State (4) reached on transition that shifts **{**:

```
S ::= { . L }
L ::= . S
L ::= . L ; S
S ::= . beep
S ::= . { L }
```

Accepting Transitions

If state has `S' ::= ... . $` item,
then add transition labeled$ to the accept action

Example:
`S' ::= S . $`
has transition labeled `$` to `accept` action

Reducing States

If state has `lhs ::= rhs .` item, then it has a `reduce` `lhs ::= rhs` action

Example:

`S ::= beep .`

has `reduce` `S ::= beep` action

No label; this state always reduces this production

- what if other items in this state shift, or accept?
- what if other items in this state reduce differently?

Rest of the States, Part 1

State (4): if shift **beep**, goto State (3)
State (4): if shift **{**, goto State (4)
State (4): if shift `S`, goto State (5)
State (4): if shift `L`, goto State (6)

State (5):

```
    L ::= S .
```

State (6):

```
    S ::= { L . }
    L ::= L . ; S
```

State (6): if shift **}**, goto State (7)
State (6): if shift **;**, goto State (8)

Rest of the States (Part 2)

State (7):

```
S ::= { L } .
```

State (8):

```
L ::= L ; . S
S ::= . beep
S ::= . { L }
```

State (8): if shift **beep**,	goto State (3)
State (8): if shift **{**,	goto State (4)
State (8): if shift `S`,	goto State (9)

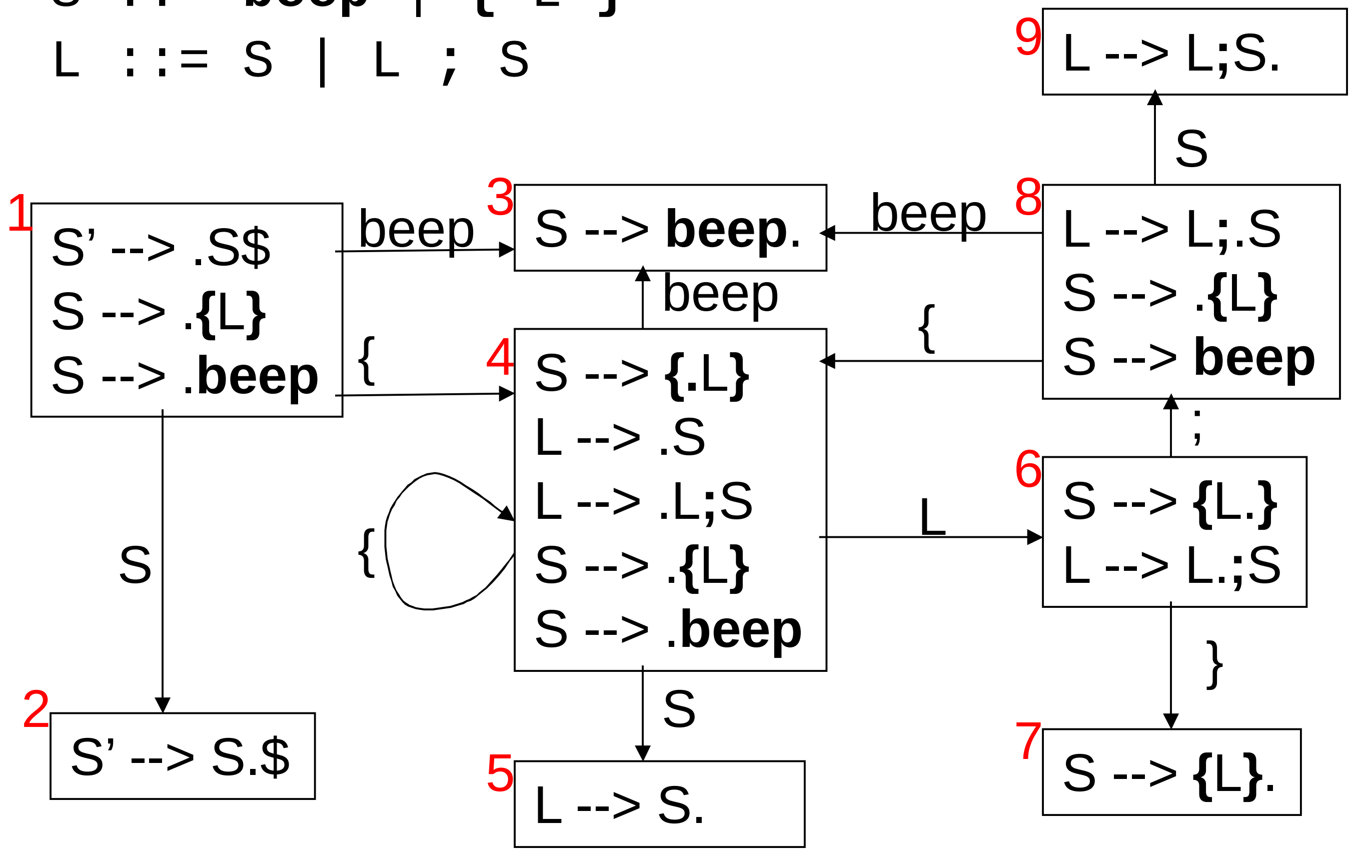
State (9):

```
L ::= L ; S .
```

(whew)

LR(0) State Diagram

S' ::= S $
S ::= **beep** | **{** L **}**
L ::= S | L **;** S


1
S' --> .S$
S --> .**{**L**}**
S --> .**beep**

2
S' --> S.$

3
S --> **beep**.

4
S --> **{**.L**}**
L --> .S
L --> .L;S
S --> .**{**L**}**
S --> .**beep**

5
L --> S.

6
S --> **{**L.**}**
L --> L.;S

7
S --> **{**L**}**.

8
L --> L**;**.S
S --> .**{**L**}**
S --> **beep**

9
L --> L;S.

Building Table of States & Transitions

Create a row for each state

Create a column for each terminal, non-terminal, and $

For every "state (*i*): if shift *X* goto state (*j*)" transition:

- if *X* is a terminal, put "shift, goto *j*" action in row *i*, column *X*
- if *X* is a non-terminal, put "goto *j*" action in row *i*, column *X*

For every "state (*i*): if $ accept" transition:

- put "accept" action in row *i*, column $

For every "state (*i*): `lhs ::= rhs.`" action:

- put "`reduce lhs ::= rhs`" action in all columns of row *i*

Table of This Grammar

State	{	}	beep	;	S	L	$
1	s,g4		s,g3		g2		
2							a!
3	`reduce S ::= beep`						
4	s,g4		s,g3		g5	g6	
5	`reduce L ::= S`						
6		s,g7		s,g8			
7	`reduce S ::= { L }`						
8	s,g4		s,g3		g9		
9	`reduce L ::= L ; S`						

Example

S’::= S $
S ::= **beep** | **{** L **}**
L ::= S | L ; S

St	{	}	beep	;	S	L	$
1	s,g4		s,g3		g2		
2							a!
3	reduce S ::= beep						
4	s,g4		s,g3		g5	g6	
5	reduce L ::= S						
6		s,g7		s,g8			
7	reduce S ::= { L }						
8	s,g4		s,g3		g9		
9	reduce L ::= L ; S						

1	{ beep ; { beep } } $
1 { 4	beep ; { beep } } $
1 { 4 beep 3	; { beep } } $
1 { 4 S 5	; { beep } } $
1 { 4 L 6	; { beep } } $
1 { 4 L 6 ; 8	{ beep } } $
1 { 4 L 6 ; 8 { 4	beep } } $
1 { 4 L 6 ; 8 { 4 beep 3	} } $
1 { 4 L 6 ; 8 { 4 S 5	} } $
1 { 4 L 6 ; 8 { 4 L 6	} } $
1 { 4 L 6 ; 8 { 4 L 6 } 7	} $
1 { 4 L 6 ; 8 S 9	} $
1 { 4 L 6	} $
1 { 4 L 6 } 7	$
1 S 2	$
accept	

Problems In Shift-Reduce Parsing

Can write grammars that cannot be handled with shift-reduce parsing

Shift/reduce conflict:

- state has both shift action(s) and reduce actions

Reduce/reduce conflict:

- state has more than one reduce action

Shift/Reduce Conflicts

LR(0) example:

```
E ::= E + T | T
```

State: `E ::= E . + T`

```
E ::= T .
```

- Can shift +
- Can reduce `E ::= T`

LR(k) example:

```
S ::= if E then S |
      if E then S else S | ...
```

State: `S ::= if E then S .`

```
S ::= if E then S . else S
```

- Can shift `else`
- Can reduce `S ::= if E then S`

Avoiding Shift-Reduce Conflicts

Can rewrite grammar to remove conflict

- E.g. `Matched Stmt vs. Unmatched Stmt`

Can resolve in favor of shift action

- try to find longest r.h.s. before reducing

  works well in practice

  `yacc`, `jflex`, et al. do this

Reduce/Reduce Conflicts

Example:

```
Stmt ::= Type id ; | LHS = Expr ; | ...
  ...
LHS ::= id | LHS [ Expr ] | ...
  ...
Type ::= id | Type [] | ...
```

State: `Type ::= id .`

`LHS  ::= id .`

Can reduce `Type ::= id`

Can reduce `LHS  ::= id`

Avoid Reduce/Reduce Conflicts

Can rewrite grammar to remove conflict

- can be hard
  - e.g. C/C++ declaration vs. expression problem
  - e.g. MiniJava array declaration vs. array store problem

Can resolve in favor of one of the reduce actions

- but which?
- `yacc`, `jflex`, et al. Pick reduce action for production listed textually first in specification