Principles of Programming Languages

Ting Zhang

Iowa State University
Computer Science Department

Lecture Note 5
September 8, 10, 2009
LR(0) Parsing

Outline

1. LR Parsing Procedure
2. LR(0) Parsing Table Generation
3. Grammar and Language Classes

Outline

1. LR Parsing Procedure
2. LR(0) Parsing Table Generation
3. Grammar and Language Classes

Bottom-up Parsing

- A top-down parser begins with the start symbol and derives productions until it gets to the terminals.

IOWA STATE UNIVERSITY

Bottom-up Parsing

- A top-down parser begins with the start symbol and derives productions until it gets to the terminals.
- ☞ A bottom-up parse starts with the terminals and reduces them to the start symbol by applying the production rules backwards.

Bottom-up Parsing

- A top-down parser begins with the start symbol and derives productions until it gets to the terminals.
- A bottom-up parse starts with the terminals and reduces them to the start symbol by applying the production rules backwards.

☞ In general, bottom-up parsing methods are more powerful than top-down methods.

Bottom-up Parsing

- A top-down parser begins with the start symbol and derives productions until it gets to the terminals.
- A bottom-up parse starts with the terminals and reduces them to the start symbol by applying the production rules backwards.
- In general, bottom-up parsing methods are more powerful than top-down methods.

☞ LR(k) parsing is the class of the most commonly used bottom-up parsing algorithms.

LR Parsing

Repeatedly preform the following actions.

- Reduce: If we find a rule $S \rightarrow \alpha$, and if the contents of the stack are $\beta\alpha$ for some (possibly empty) β, then we reduce the stack to βS.

LR Parsing

Repeatedly preform the following actions.

- Reduce: If we find a rule $S \to \alpha$, and if the contents of the stack are $\beta\alpha$ for some (possibly empty) β, then we reduce the stack to βS.
- ☞ Shift: If it is impossible to perform a reduction and there are tokens remaining in the input, then we transfer a token from the input onto the stack.

LR Parsing

Repeatedly preform the following actions.

- Reduce: If we find a rule $S \rightarrow \alpha$, and if the contents of the stack are $\beta\alpha$ for some (possibly empty) β, then we reduce the stack to βS.
- Shift: If it is impossible to perform a reduction and there are tokens remaining in the input, then we transfer a token from the input onto the stack.
- ☞ Error: If neither of the two above cases apply, we have an error.

LR Parsing

Repeatedly preform the following actions.

- Reduce: If we find a rule $S \rightarrow \alpha$, and if the contents of the stack are $\beta\alpha$ for some (possibly empty) β, then we reduce the stack to βS.
- Shift: If it is impossible to perform a reduction and there are tokens remaining in the input, then we transfer a token from the input onto the stack.
- Error: If neither of the two above cases apply, we have an error.

Report success when reaching the end of input stream with the **accept** state.

LR Parsing Table

An LR parser uses two tables:

- ☞ Action table: Action[s,a] tells the parser what to do when the state on top of the stack is s and terminal a is the next input token.

LR Parsing Table

An LR parser uses two tables:

- Action table: Action[s,a] tells the parser what to do when the state on top of the stack is s and terminal a is the next input token.
- ☞ Goto table: Goto[s,X] indicates the new state to place on top of the stack after a reduction of the nonterminal X while state s is on top of the stack.

LR Parsing Procedure

- Action[top,a] = s: push *s* onto the stack and get the next token

LR Parsing Procedure

- Action[top,a] = s: push *s* onto the stack and get the next token
- Action[top,a] = $X \to Y_1 \cdot Y_k$: pop k states from the stack and push $Goto[top, X]$ onto the stack

LR Parsing Procedure

- Action[top,a] = s: push *s* onto the stack and get the next token
- Action[top,a] = $X \rightarrow Y_1 \cdot Y_k$: pop k states from the stack and push $Goto[top, X]$ onto the stack
- ☞ Action[top,a] = acc: parsing ends with success

LR Parsing Procedure

- Action[top,a] = s: push *s* onto the stack and get the next token
- Action[top,a] = $X \rightarrow Y_1 \cdot Y_k$: pop *k* states from the stack and push $Goto[top, X]$ onto the stack
- Action[top,a] = acc: parsing ends with success
- ☞ Action[top,a] = -: syntax error

IOWA STATE UNIVERSITY

LR(0) Parsing: An Example

LR(0) grammar:

1. $S \rightarrow S(S)$
2. $S \rightarrow \epsilon$

LR(0) Parsing: An Example

LR(0) grammar:

1. $S \to S(S)$
2. $S \to \epsilon$

Parsing table:

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

LR(0) Parsing: An Example

LR(0) grammar:

1. $S \rightarrow S(S)$
2. $S \rightarrow \epsilon$

LR(0) Parsing: An Example

LR(0) grammar:

1. $S \rightarrow S(S)$
2. $S \rightarrow \epsilon$

Parsing table:

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

IOWA STATE UNIVERSITY

Meaning of LR(0) Parsing Table

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

- r2 means to reduce the handle on the top of the stack by (2) $S \rightarrow \epsilon$.

Meaning of LR(0) Parsing Table

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

- r2 means to reduce the handle on the top of the stack by (2) $S \rightarrow \epsilon$.
- ☞ s_2 means to shift the input symbol and push s_2 onto the stack.

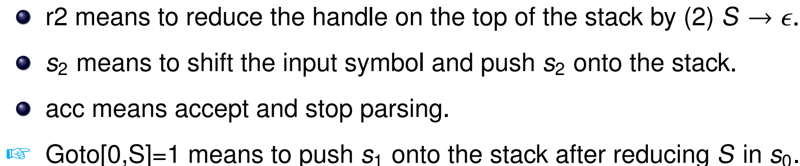
Meaning of LR(0) Parsing Table

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

- r2 means to reduce the handle on the top of the stack by (2) $S \rightarrow \epsilon$.
- s_2 means to shift the input symbol and push s_2 onto the stack.
- ☞ acc means accept and stop parsing.

Meaning of LR(0) Parsing Table

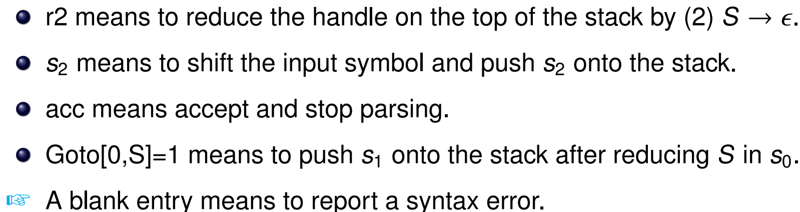
State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	



- r2 means to reduce the handle on the top of the stack by (2) $S \rightarrow \epsilon$.
- s_2 means to shift the input symbol and push s_2 onto the stack.
- acc means accept and stop parsing.
- ☞ Goto[0,S]=1 means to push s_1 onto the stack after reducing S in s_0.

Meaning of LR(0) Parsing Table

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	



- r2 means to reduce the handle on the top of the stack by (2) $S \to \epsilon$.
- s_2 means to shift the input symbol and push s_2 onto the stack.
- acc means accept and stop parsing.
- Goto[0,S]=1 means to push s_1 onto the stack after reducing S in s_0.
- ☞ A blank entry means to report a syntax error.

LR(0) Parsing of $()()$

Stack	Input	Action
s_0	()()$	reduce by (2) $S \to \epsilon$ and push s_1 onto the stack
$s_0 s_1$	()()$	shift s_2 onto the stack
$s_0 s_1 s_2$	)()$	reduce by (2) $S \to \epsilon$ and push s_3
$s_0 s_1 s_2 s_3$	)()$	shift s_4 onto the stack
$s_0 s_1 s_2 s_3 s_4$	()$	reduce by (1) $S \to S(S)$ and push s_1
$s_0 s_1$	()$	shift s_2 onto the stack
$s_0 s_1 s_2$	)$	reduce by (2) $S \to \epsilon$ and push s_3
$s_0 s_1 s_2 s_3$	)$	shift s_4 onto the stack
$s_0 s_1 s_2 s_3 s_4$	$	reduce by (1) $S \to S(S)$ and push s_1
$s_0 s_1$	$	accept

LR(0) Parsing: A Bigger Example

LR(0) grammar:

1. $S \rightarrow E$
2. $E \rightarrow T$
3. $E \rightarrow E + T$
4. $T \rightarrow id$
5. $T \rightarrow (E)$

IOWA STATE UNIVERSITY

Parsing Table

State	Action					Goto	
	id	+	(	)	$	E	T
s_0	s4		s3			1	2
s_1		s5			acc		
s_2	r2	r2	r2	r2	r2		
s_3	s4		s3			6	2
s_4	r4	r4	r4	r4	r4		
s_5	s4		s3				8
s_6		s5		s7			
s_7	r3	r3	r3	r3	r3		
s_8	r1	r1	r1	r1	r1		

LR(0) Parsing of $id + (id)$

Stack	Input	Action
s_0	id+(id)$	shift s_4 onto the stack, move ahead in input
$s_0 s_4$	+(id)$	reduce by (4) $T \rightarrow id$, pop the stack, goto s_2
$s_0 s_2$	+(id)$	reduce by (2) $E \rightarrow T$ and goto s_1
$s_0 s_1$	+(id)$	shift s_5 onto the stack
$s_0 s_1 s_5$	(id)$	shift s_3 onto the stack
$s_0 s_1 s_5 s_3$	id)$	shift s_4 onto the stack
$s_0 s_1 s_5 s_3 s_4$	)$	reduce by (4) $T \rightarrow id$, goto s_2
$s_0 s_1 s_5 s_3 s_2$	)$	reduce by (2) $E \rightarrow T$ and goto s_6
$s_0 s_1 s_5 s_3 s_6$	)$	shift s_7 onto the stack
$s_0 s_1 s_5 s_3 s_6 s_7$	$	reduce by (3) $T \rightarrow (E)$ and goto s_8
$s_0 s_1 s_5 s_8$	$	reduce by (1) $E \rightarrow E + T$ and goto s_1
$s_0 s_1$	$	accept

The Essentials of LR Parsing

- It is a “recognizer”, not a “predictor”.

The Essentials of LR Parsing

- It is a “recognizer”, not a “predictor”.
- ☞ It builds a parse tree from the bottom up.

The Essentials of LR Parsing

- It is a "recognizer", not a "predictor".
- It builds a parse tree from the bottom up.
- ☞ The states keep track of which productions we might reduce.

The Problems of LR(0) Parsing

- Shift-Reduce Conflict: for example

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

The Problems of LR(0) Parsing

- Shift-Reduce Conflict: for example

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

☞ Reduce-Reduce Conflict: for example

$$
\begin{array}{lll}
1. & S & \rightarrow X \\
2. & X & \rightarrow Y \\
3. & & \rightarrow id \\
4. & Y & \rightarrow id
\end{array}
$$

Outline

1 LR Parsing Procedure

2 LR(0) Parsing Table Generation

3 Grammar and Language Classes

IOWA STATE UNIVERSITY

Configuration

☞ A production with a · somewhere in its righthand side is called a **configuration**.

Configuration

- A production with a $\cdot$ somewhere in its righthand side is called a **configuration**.

☞ $A \rightarrow \cdot\alpha$ indicates that we are about to match $A \rightarrow \alpha$.

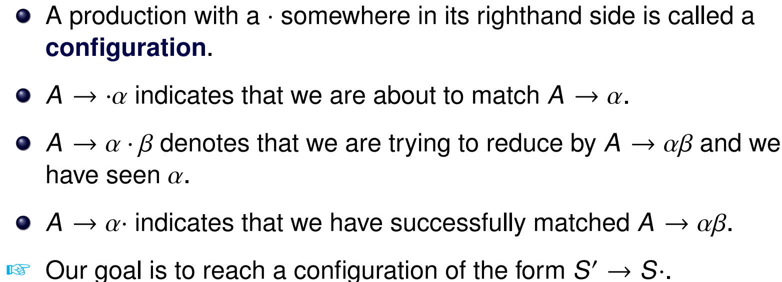
Configuration

- A production with a $\cdot$ somewhere in its righthand side is called a **configuration**.
- $A \rightarrow \cdot\alpha$ indicates that we are about to match $A \rightarrow \alpha$.
- ☞ $A \rightarrow \alpha \cdot \beta$ denotes that we are trying to reduce by $A \rightarrow \alpha\beta$ and we have seen α.

Configuration

- A production with a $\cdot$ somewhere in its righthand side is called a **configuration**.
- $A \rightarrow \cdot\alpha$ indicates that we are about to match $A \rightarrow \alpha$.
- $A \rightarrow \alpha \cdot \beta$ denotes that we are trying to reduce by $A \rightarrow \alpha\beta$ and we have seen α.
- ☞ $A \rightarrow \alpha\cdot$ indicates that we have successfully matched $A \rightarrow \alpha\beta$.

Configuration



State, Successor, and State Set

☞ We may not know which production will eventually be fully matched.

State, Successor, and State Set

- We may not know which production will eventually be fully matched.

☞ A **state** is a configuration set consisting of all possible configurations that are matchable at a time.

State, Successor, and State Set

- We may not know which production will eventually be fully matched.
- A **state** is a configuration set consisting of all possible configurations that are matchable at a time.

☞ When we match a symbol, we need move to the **successor** state.

State, Successor, and State Set

- We may not know which production will eventually be fully matched.
- A **state** is a configuration set consisting of all possible configurations that are matchable at a time.
- When we match a symbol, we need move to the **successor** state.

☞ Repeatedly compute successor state, we eventually have a DFA.

Computing Configurations: Start

- When we predict a production, we place $\cdot$ at the beginning of a production: $A \rightarrow \cdot\alpha$.

Computing Configurations: Start

- When we predict a production, we place $\cdot$ at the beginning of a production: $A \to \cdot\alpha$.
- ☞ When we predict $A \to \cdot\epsilon$, the match is immediate and so we just use $A \to \epsilon\cdot$.

Computing Configurations: Start

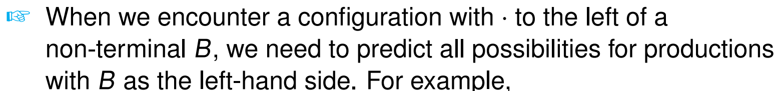
- When we predict a production, we place $\cdot$ at the beginning of a production: $A \to \cdot\alpha$.
- When we predict $A \to \cdot\epsilon$, the match is immediate and so we just use $A \to \epsilon\cdot$.
- ☞ At the start, we have to use the start symbol S. We do not yet know which one, so we predict them all. For example,

1. $S \to \cdot S(S)$
2. $S \to \cdot\epsilon$

Computing Configurations: Start

- When we predict a production, we place $\cdot$ at the beginning of a production: $A \rightarrow \cdot\alpha$.
- When we predict $A \rightarrow \cdot\epsilon$, the match is immediate and so we just use $A \rightarrow \epsilon\cdot$.
- At the start, we have to use the start symbol S. We do not yet know which one, so we predict them all. For example,

  1. $S \rightarrow \cdot S(S)$
  2. $S \rightarrow \cdot\epsilon$



☞ When we encounter a configuration with $\cdot$ to the left of a non-terminal B, we need to predict all possibilities for productions with B as the left-hand side. For example,

  1. $B \rightarrow \cdot\beta_1$
  2. $B \rightarrow \cdot\beta_2$
  3. ......

Computing Configurations: Closure

- The newly added configurations may predict other non-terminals, forcing additional productions to be included.

Computing Configurations: Closure

- The newly added configurations may predict other non-terminals, forcing additional productions to be included.
- ☞ We continue this process until no additional configurations can be added. The final configuration set is called **closure** of the original configuration set.

Computing Closures

```
1: ConfigSet Closure(ConfigSet C) {
2: repeat
3:     if A → α · Bβ is in C then
4:         add all configurations of the form B → ·γ to C
5:     end if
6: until no more configurations can be added
7: return C
8: }
```

Example: Computing Closures

Grammar:

$$
\begin{array}{lll}
1. & S & \rightarrow A\ b \\
2. & A & \rightarrow C\ D \\
3. & C & \rightarrow D \\
4. & C & \rightarrow c \\
5. & D & \rightarrow d
\end{array}
$$

IOWA STATE UNIVERSITY

Example: Computing Closures

Grammar:

$$
\begin{array}{llll}
1. & S & \rightarrow & A\ b \\
2. & A & \rightarrow & C\ D \\
3. & C & \rightarrow & D \\
4. & C & \rightarrow & c \\
5. & D & \rightarrow & d
\end{array}
$$

Compute Closure($S \rightarrow \cdot Ab$):

☞ First we include $A \rightarrow \cdot CD$.

Example: Computing Closures

Grammar:

$$
\begin{array}{lll}
1. & S & \to A\ b \\
2. & A & \to C\ D \\
3. & C & \to D \\
4. & C & \to c \\
5. & D & \to d
\end{array}
$$

Compute Closure($S \to \cdot Ab$):

- First we include $A \to \cdot CD$.
- ☞ Then we include $C \to \cdot D$ and $C \to \cdot c$.

IOWA STATE UNIVERSITY

Example: Computing Closures

Grammar:

$$\begin{array}{lll} 1. & S & \rightarrow A\ b \\ 2. & A & \rightarrow C\ D \\ 3. & C & \rightarrow D \\ 4. & C & \rightarrow c \\ 5. & D & \rightarrow d \end{array}$$

Compute Closure($S \rightarrow \cdot Ab$):

- First we include $A \rightarrow \cdot CD$.
- Then we include $C \rightarrow \cdot D$ and $C \rightarrow \cdot c$.
- ☞ At last we include $D \rightarrow \cdot d$.

Example: Computing Closures

Grammar:

$$
\begin{array}{lll}
1. & S & \rightarrow A\ b \\
2. & A & \rightarrow C\ D \\
3. & C & \rightarrow D \\
4. & C & \rightarrow c \\
5. & D & \rightarrow d
\end{array}
$$

Example: Computing Closures

Grammar:

$$\begin{array}{lll} 1. & S & \rightarrow A\ b \\ 2. & A & \rightarrow C\ D \\ 3. & C & \rightarrow D \\ 4. & C & \rightarrow c \\ 5. & D & \rightarrow d \end{array}$$

Closure($S \rightarrow \cdot Ab$) is:

$$\begin{array}{lll} 1. & S & \rightarrow \cdot\ A\ b \\ 2. & A & \rightarrow \cdot\ C\ D \\ 3. & C & \rightarrow \cdot\ D \\ 4. & C & \rightarrow \cdot c \\ 5. & D & \rightarrow \cdot d \end{array}$$

Successor State

☞ When we match a symbol, we shift · past the symbol just matched symbol.

```
1: ConfigSet Successor(ConfigSet C, Symbol X) {
2: B = ∅
3: for each configuration c in C do
4:     if c is of the form A → α · Xβ then
5:         add A → αX · β to B
6:     end if
7: end for
8: return Closure(B)
9: }
```

Successor State

- When we match a symbol, we shift $\cdot$ past the symbol just matched symbol.
- ☞ Configurations that do not have a dot to the left of the matched symbol are deleted.

```
1: ConfigSet Successor(ConfigSet C, Symbol X) {
2: B = ∅
3: for each configuration c in C do
4:     if c is of the form A → α · Xβ then
5:         add A → αX · β to B
6:     end if
7: end for
8: return Closure(B)
9: }
```

Example: Computing Successors

The successor of

$$
\begin{array}{lll}
1. & S & \rightarrow \cdot\, A\ b \\
2. & A & \rightarrow \cdot\, C\ D \\
3. & C & \rightarrow \cdot\, D \\
4. & C & \rightarrow \cdot c \\
5. & D & \rightarrow \cdot d
\end{array}
$$

with respect to C is

$$
\begin{array}{lll}
6. & A & \rightarrow C\ \cdot\ D \\
7. & D & \rightarrow \cdot\, d
\end{array}
$$

State Set

```
1:  StateSet BuildStates(ConfigSet C_0) {
2:  F = ∅
3:  for each configuration C in F do
4:      for each symbol X do
5:          if Successor(C, X) is not empty then
6:              add Successor(C, X) to F
7:          end if
8:      end for
9:  end for
10: return F
11: }
```

Example: State Set

$$I_0: \quad E \rightarrow \cdot E + T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot (E)$$
$$T \rightarrow \cdot id$$

IOWA STATE UNIVERSITY

Example: State Set

$$
\begin{aligned}
I_0 : \quad & E \rightarrow \cdot E + T \\
& E \rightarrow \cdot T \\
& T \rightarrow \cdot (E) \\
& T \rightarrow \cdot id
\end{aligned}
$$

4 successors:

E	T	id	(
$I_1 : E \rightarrow E \cdot + T$	$I_2 : E \rightarrow T \cdot$	$I_4 : T \rightarrow id \cdot$	$I_3 : T \rightarrow (\cdot E)$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$

Example: State Set

$I_1 : E \rightarrow E \cdot + T$

Example: State Set

$I_1 : E \rightarrow E \cdot + T$

1 successor:

$+$
$I_5 : E \rightarrow E + \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$

Example: State Set

$$\begin{aligned} I_5 : E &\rightarrow E\ +\ \cdot T \\ T &\rightarrow \cdot\ (\ E\ ) \\ T &\rightarrow \cdot\ id \end{aligned}$$

IOWA STATE UNIVERSITY

Example: State Set

$$
\begin{aligned}
I_5 : E &\rightarrow E + \cdot T \\
T &\rightarrow \cdot (E) \\
T &\rightarrow \cdot id
\end{aligned}
$$

3 successors:

T	id	(
$I_8 : E \rightarrow E + T \cdot$	$I_4 : T \rightarrow id \cdot$	$I_3 : T \rightarrow (\cdot E)$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot (E)$ $T \rightarrow \cdot id$

Example: State Set

$$
\begin{aligned}
I_3 : T &\rightarrow (\cdot E) \\
E &\rightarrow \cdot E + T \\
E &\rightarrow \cdot T \\
T &\rightarrow \cdot (E) \\
T &\rightarrow \cdot id
\end{aligned}
$$

IOWA STATE UNIVERSITY

Example: State Set

$$
\begin{aligned}
I_3 : T &\rightarrow (\ \cdot E\ ) \\
E &\rightarrow \cdot E + T \\
E &\rightarrow \cdot T \\
T &\rightarrow \cdot (\ E\ ) \\
T &\rightarrow \cdot\, id
\end{aligned}
$$

4 successors:

T	id	E	(
$I_2 : E \rightarrow T\ \cdot$	$I_4 : T \rightarrow id\ \cdot$	$I_6 : T \rightarrow (\ E \cdot\ )$ $E \rightarrow E\ \cdot + T$	$I_3 : T \rightarrow (\ \cdot E\ )$ $E \rightarrow \cdot E + T$ $E \rightarrow \cdot T$ $T \rightarrow \cdot (\ E\ )$ $T \rightarrow \cdot\, id$

IOWA STATE UNIVERSITY

Example: State Set

$$I_6 : T \rightarrow (\ E \cdot\ )$$
$$E \rightarrow E\ \cdot + T$$

Example: State Set

$$I_6 : T \rightarrow (\; E \cdot)$$
$$E \rightarrow E \; \cdot + T$$

2 successors:

$+$	$)$
$I_5 : E \rightarrow E \; + \; \cdot T$ $T \rightarrow \cdot (\; E \;)$ $T \rightarrow \cdot \, id$	$I_7 : T \rightarrow (\; E \;) \cdot$

Example: State Set

$I_2 : E \rightarrow T \cdot$
$I_4 : T \rightarrow id \cdot$
$I_7 : T \rightarrow (E) \cdot$
$I_8 : E \rightarrow E + T \cdot$

Example: State Set

$I_2 : E \rightarrow T \cdot$
$I_4 : T \rightarrow id \cdot$
$I_7 : T \rightarrow (E) \cdot$
$I_8 : E \rightarrow E + T \cdot$

None has a successor!

Parsing Table

```
1:  ParsingTable BuildTable(StateSet 𝓕 = {C_0, ..., C_n}) {
2:  for each configuration set C_i in 𝓕 do
3:      if S' → S· is in C_i then
4:          set Action[i, $] to accept
5:      else if A → α· in C_i then
6:          set Action[i, a] to A → α for all terminal a
7:      else if A → α · aβ is in C_i and Successor(C_i, a) = C_j then
8:          set Action[i, a] to shift j
9:      end if
10:     for each nonterminal A do
11:         if Successor(C_i, A) = C_j then
12:             set Goto[i, A] = j
13:         end if
14:     end for
15: end for
16: }
```

Outline

1 LR Parsing Procedure

2 LR(0) Parsing Table Generation

3 Grammar and Language Classes

Beyond LR(0)

- SLR: peeks at upcoming input and use FOLLOW sets to resolve conflicts.

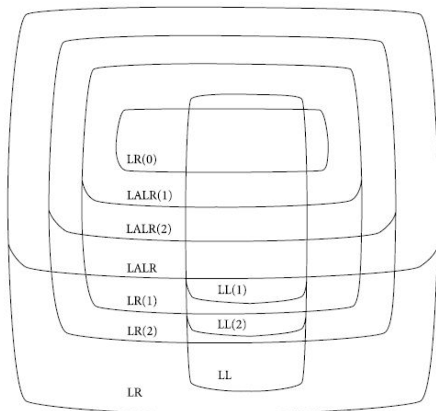
Beyond LR(0)

- SLR: peeks at upcoming input and use FOLLOW sets to resolve conflicts.
- ☞ LALR: improves on SLR by using **state specific** FOLLOW sets instead.

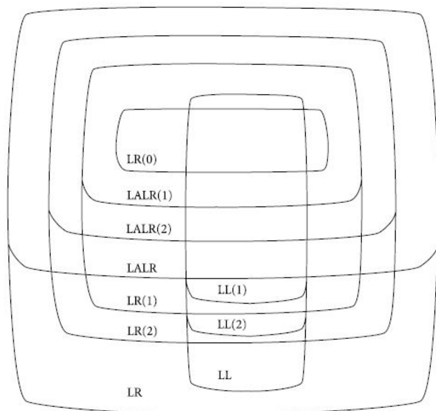
Beyond LR(0)

- SLR: peeks at upcoming input and use FOLLOW sets to resolve conflicts.
- LALR: improves on SLR by using **state specific** FOLLOW sets instead.
- ☞ LR: duplicates states in order to resolve conflicts.

Grammar Containment Relation

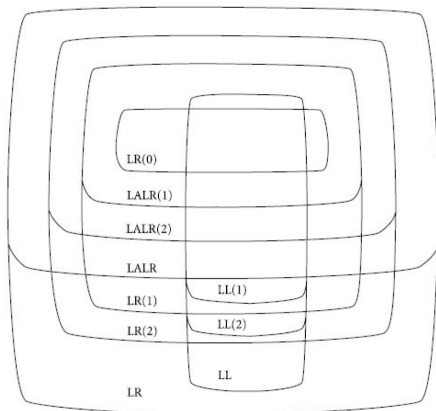


Grammar Containment Relation



(a) SLL(k) is just inside LL(k), for $k \geq 2$, but has the same relationship to everything else.

Grammar Containment Relation



(a) SLL(k) is just inside LL(k), for $k \geq 2$, but has the same relationship to everything else.

(b) SLR(k) is just inside LALR(k), for $k \geq 1$, but has the same relationship to everything else.

IOWA STATE UNIVERSITY

Language Containment Relation