Bottom Up Parsing

PITT CS 1622

Bottom Up Parsing

❑ More powerful than top down
  ➢ Don't need left factored grammars
  ➢ Can handle left recursion
  ➢ Can parse a larger set of grammars (and languages)

❑ Begins at leaves and works to the top
  ➢ In reverse order of rightmost derivation
    (In effect, builds tree from left to right)

❑ Also known as Shift-Reduce parsing
  ➢ Involves two types of operations: shift and reduce
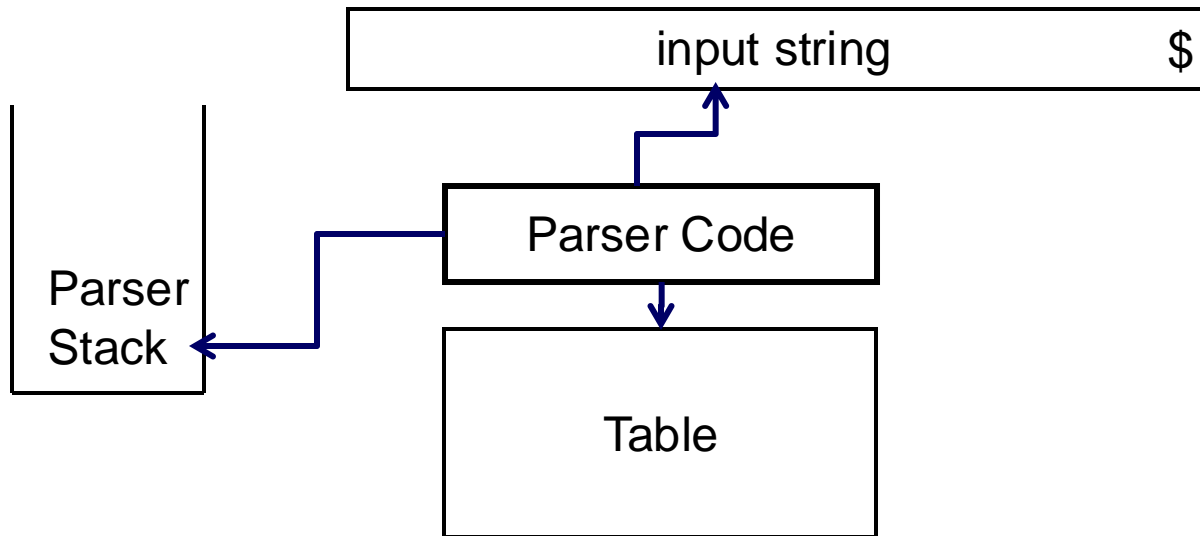
Parser Implementation
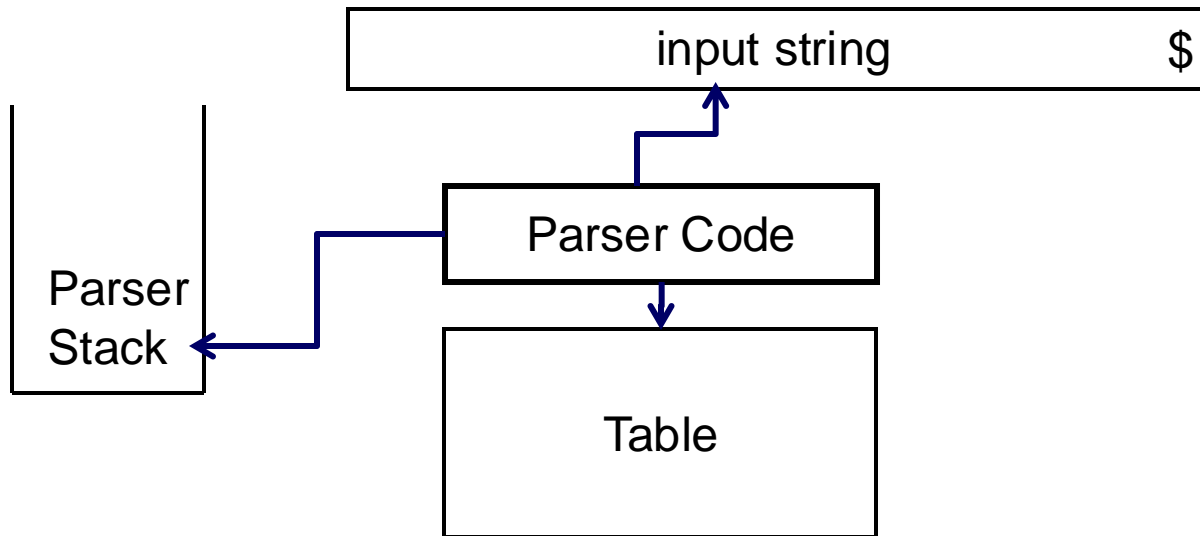
Parser Stack – holds consumed portion of derivation string

Table – "actions" performed based on rules of grammar, and current state of stack and input string

Parser Code – next action based on **(current token, stack top)**

Parser Implementation

```
┌─────────────────────────────────────────────┐
│  input string                            $   │
└─────────────────────────────────────────────┘
                          ▲
                          │
              ┌───────────────────────┐
  ┌──────┐    │     Parser Code       │
  │      │◄───│                       │
  │Parser│    └───────────────────────┘
  │Stack │                │
  │      │                ▼
  │      │    ┌───────────────────────┐
  └──────┘    │        Table          │
              │                       │
              └───────────────────────┘
```
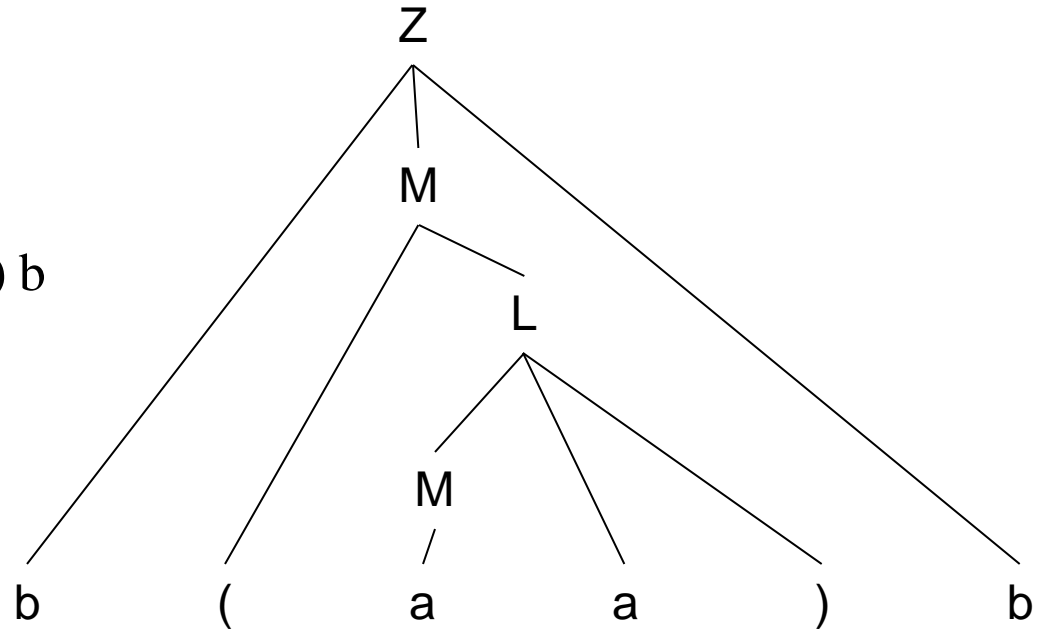
Actions
1. **Shift** – consume input symbol and push symbol onto the stack
2. **Reduce** – pop RHS at stack top and push LHS of a production rule, reducing stack contents
3. Accept – success (when reduced to start symbol and input at $)
4. Error

Z → b M b

M → (L | a

L → M a) |)

Considering string: w = b (a a) b

The rightmost derivation of this parse tree:

Z ⇒ b M b ⇒ b (L b ⇒ b (M a) b ⇒ b (a a) b

Bottom up parsing involves finding "handles" (RHSs) to reduce

b (a a) b ⇒ b (M a) b ⇒ b (L b ⇒ b M b ⇒ Z

$Z \rightarrow b\ M\ b$
$M \rightarrow (\ L\ |\ a$
$L \rightarrow M\ a\ )\ |\ )$

String
    b (a a) $

Stack	Input	Action
$	b (a a) b $	shift
$ b	(a a) b $	shift
$ b (	a a) b $	shift
$ b (a	a) b $	reduce
$ b (M	a) b $	shift
$ b (M a	) b $	shift
$ b (M a)	b $	reduce
$ b (L	b $	reduce
$ b M	b $	shift
$ b M b	$	reduce
$ Z	$	accept

Sentential Form and Handle

- ❑ **Sentential form**: Any string derivable from the start symbol
- ❑ **Handle**: RHS of a production rule that, when replaced with LHS in a sentential form, will lead to another sentential form

- ❑ Definition:
  - ➤ Let $\alpha\beta w$ be a sentential form where
    - α, β is a string of terminals and non-terminals
    - w is a string of terminals
    - $X \rightarrow \beta$ is a production rule
    - Then β is a handle of $\alpha\beta w$ if
    - $S \Rightarrow^* \alpha X w \Rightarrow \alpha\beta w$ by a rightmost derivation
  - ➤ Handles formalize the intuition "β should be reduced to X for a successful parse", but does not really say how to find them

Single Pass Left-to-Right Scan

❑ Note in the formulation of a handle $S \Rightarrow^* \alpha Xw \Rightarrow \alpha \beta w$

➢ α is a string of terminals and non-terminals

➢ w is a string of only terminals

❑ Why is this so?

➢ Let's assume w contained a non-terminal Y

➢ $S \Rightarrow^* \alpha Xw_1 Yw_2 \Rightarrow \alpha \beta w_1 Yw_2$, where $w = w_1 Yw_2$

➢ Above is not a rightmost derivation. Contradiction!

❑ What are the implications?

➢ $\alpha \beta$ is consumed input and w is unconsumed input

➢ Reduction only happens at the frontier of consumed input

$\rightarrow$ Amenable to single pass left-to-right scan

Handle Always Occurs at Top of Stack

❑ Grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Sentential form	Handle	Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E*E$	$E \rightarrow E*E$
$E + E$	$E+E$	$E \rightarrow E+E$
E		

❑ # indicates top of stack (at the frontier of reduction where our β is)
Left of # : stack contents, Right of # : unconsumed input string

$id_1 \# + id_2 * id_3 \Rightarrow E \# + id_2 * id_3 \Rightarrow E + \# id_2 * id_3 \Rightarrow E + id_2 \# * id_3$

$\Rightarrow E + E \# * id_3 \Rightarrow E + E * id_3 \# \Rightarrow E + E * E\# \Rightarrow E + E \# \Rightarrow E$

❑ Stack works because reducing $X \rightarrow \beta$ involves popping recently pushed symbols

Handle Always Occurs at Top of Stack

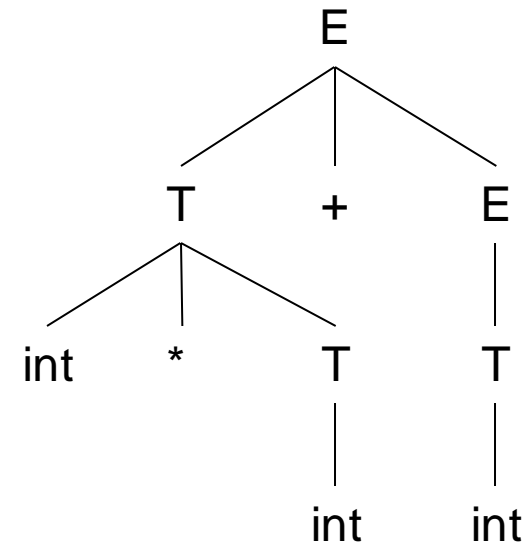❑ Consider our usual grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow int * T \mid int \mid (E)$$

Consider the string: int * int + int

sentential form	production
int * int # + int	T → int
int * T # + int	T → int * T
T + int #	T → int
T + T #	E → T
T + E #	E → T + E
E #	

❑ Reduction of a handle always happens at the top of the stack

❑ Makes life easier for parser (no need to access middle of stack)

Ambiguous Grammars

❑ Conflicts arise with ambiguous grammars

➢ Just like LL parsing, bottom up parsing tries to predict the correct action, but if there are multiple correct actions, conflicts arise

❑ Example:

➢ Consider the ambiguous grammar

$$E \rightarrow E * E \mid E + E \mid (E) \mid int$$

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**reduce E → E * E**
E # + int	shift
E + # int	shift
E + int #	reduce E → int
E + E #	reduce E → E + E
E #	

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**shift**
E * E + # int	shift
E * E + int #	reduce E → int
E * E + E #	reduce E → E + E
E * E #	reduce E → E * E
E #	

Ambiguity

❑ Previous shift-reduce conflict occurred because of ambiguity
  ➢ Due to lack of <u>precedence</u> between + and * in the grammar
  ➢ Ambiguity shows up as "conflicts" in the parsing table
    (More than one action in parse table, just like for LL parsers)

❑ Shift-reduce conflict also occurs with input "int + int + int"
  ➢ Due to ambiguous <u>associativity</u> of * and +

❑ Can always rewrite to encode precedence and associativity
  ➢ But can sometimes result in convoluted grammars
  ➢ Tools have other means to encode precedence and association
    %left '+' ' '
    %left ' ' '/ '

Properties of Bottom Up Parsing

❑ Handles always appear at the top of the stack

  ➢ Never in middle of stack

  ➢ Justifies use of stack in shift – reduce parsing

❑ Results in an easily generalized shift – reduce strategy

  ➢ If there is no handle at the top of the stack, shift

  ➢ If there is a handle, reduce to the non-terminal

  ➢ Easy to automate the synthesis of the parser using a table

❑ Can have conflicts

  ➢ If it is legal to either shift or reduce then there is a shift-reduce conflict.

  ➢ If there are two legal reductions, then there is a reduce-reduce conflict.

  ➢ Most often occur because of ambiguous grammars

    • In rare cases, because of non-ambiguous grammars not amenable to parser

Types of Bottom Up Parsers

❑ Types of bottom up parsers
- ➢ Simple precedence parsers
- ➢ Operator precedence parsers
- ➢ LR family parsers
- ➢ …

❑ In this course, we will only discuss LR family parsers
- ➢ Most automated tools generate either LL or LR parsers

LR Parsers are more powerful than LL

❑ **LR family of parsers**

➢ LR(k)   L – left to right scan

   R – rightmost derivation in reverse

   k elements of look ahead

❑ **Pros in comparison to LL(k)**

1. More powerful than LL(k)

   • Handles more grammars: no left recursion removal, no left factoring needed

   • Handles more languages: LL(k) $\subset$ LR(k)

2. As efficient as LL(k)

   • Linear in time and space to length of input (same as LL(k))

3. As convenient as LL(k)

   • Can generate automatically from grammar – YACC, Bison

A Hierarchy of Grammar Classes
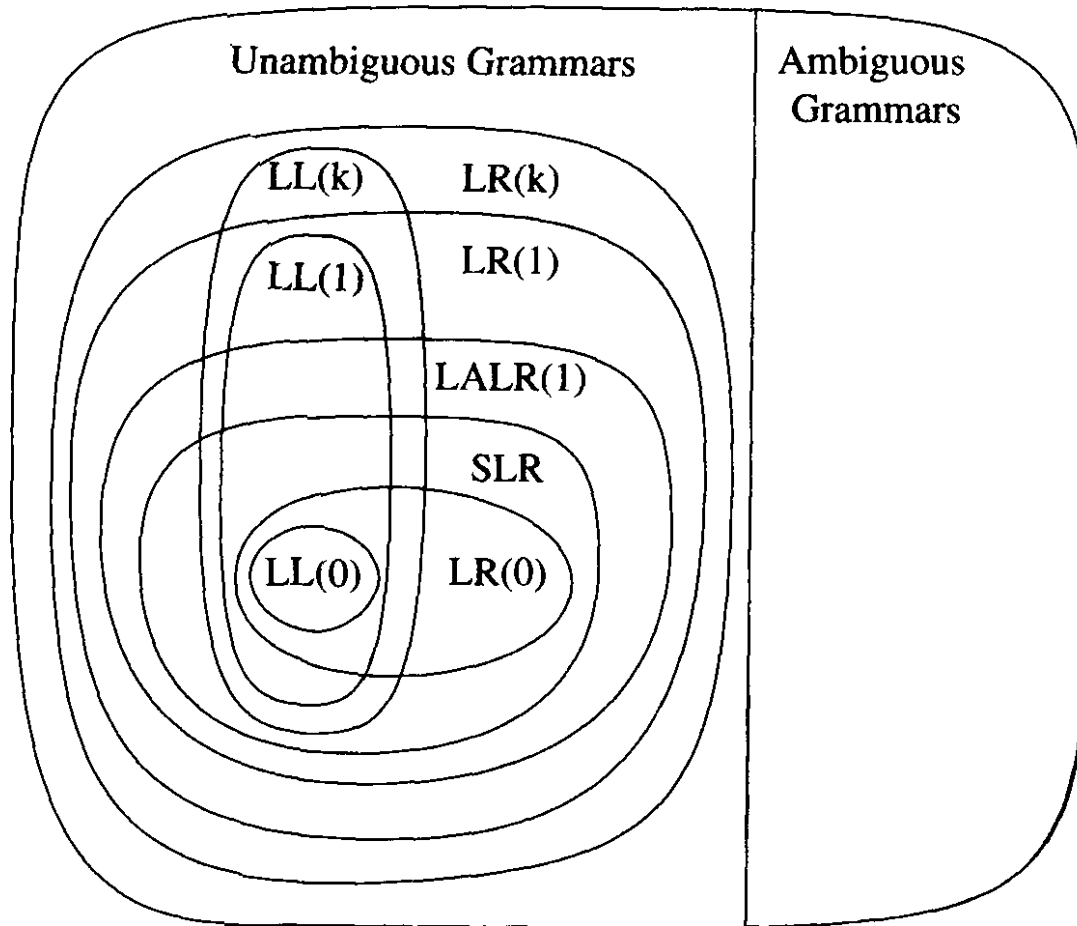
LR Parsers are harder to deal with

❑ Cons in comparison to LL(k)

1. More complex in structure compared to LL(k)
   - Structure of parser looks nothing like grammar
   - Parse conflicts are hard to understand and debug
2. Harder to emit informative error messages and recover from errors
   - LR is a bottom-up while LL is a top-down parser
   - When parse error occurs,
     LR: Knows only of currently reduced non-terminal
     LL: Knows how upper levels of tree look like and context of error
   - → LL can emit smart error messages referring to context of error
   - → LL can perform better error recovery according to context

Implementation
-- LR Parsing

Pitt, CS 2210

Viable Prefix

❑ Definition: α is a viable prefix if

- ➢ There is a w where αw is a rightmost sentential form (prefix)
- ➢ α does not extend beyond the rightmost handle (viable)
- ➢ In other words, if there is a w where α#w is a configuration of a shift-reduce parser

b (a #a) b ⇒ b (M# a) b ⇒ b (L# b ⇒ b M # b ⇒ Z#

❑ If contents of parse stack is a viable prefix, that means the parser is on the right track

❑ Shift-reduce parsing is the process of massaging the contents of the parse stack from viable prefix to viable prefix

- ➢ Error if neither shifting or reducing results in a viable prefix

Massaging into a Viable Prefix

❑ How do you know what results in a viable prefix?

➢ Example grammar
S → a B S | b
B → b

➢ Example shift and reduce on: a # b b

Shift: a # b b ⇒ a b # b   How do you know shifting is the answer?

Reduce: a b # b ⇒ a B # b   Should I apply B → b (and not S → b)?

❑ You need to keep track of where you are on the RHS of rules

➢ In example
Shift: a # b b ⇒ a b # b
S → a # B S, B → # b ⇒ S → a # B S, B → b #
Reduce: a b # b ⇒ a B # b
S → a # B S, B → b # ⇒ S → a B # S, S → # a B S, S → # b

LR(0) Item Notation

❑ LR(0) Item: a production + a dot on the RHS

➢ Dot indicates extent of production already seen

➢ In example grammar
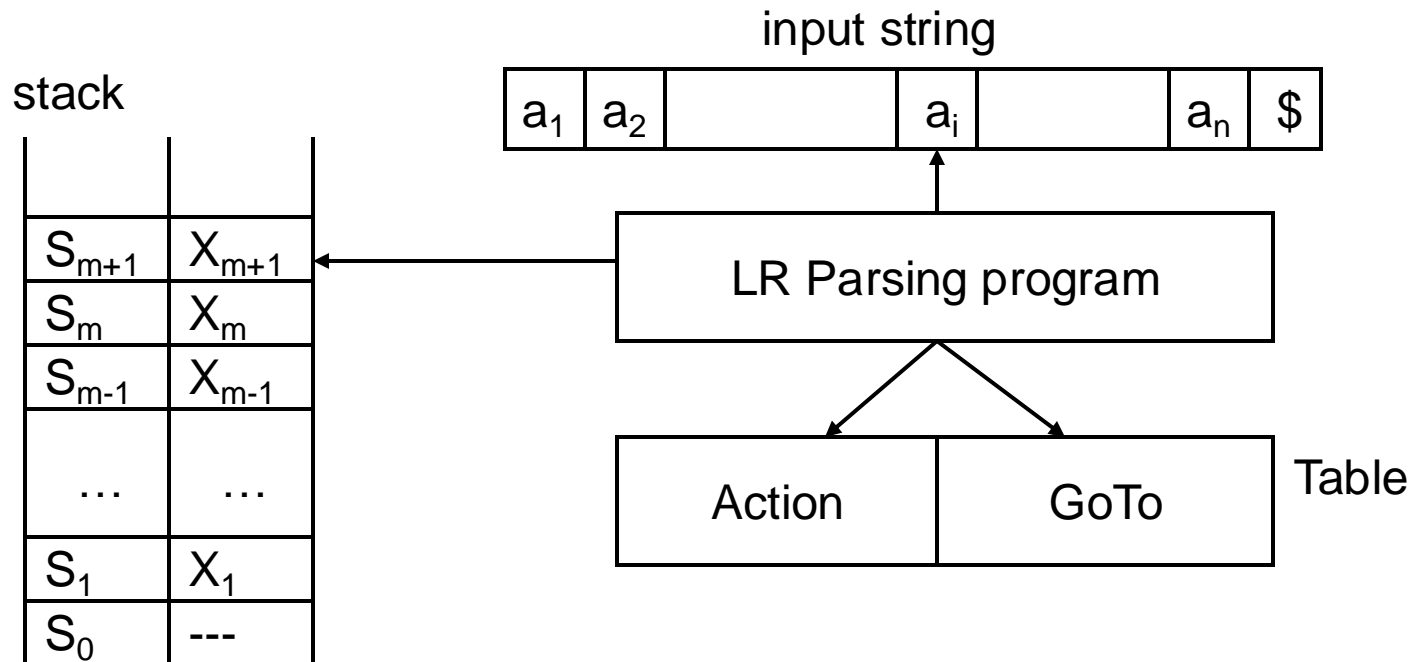
Items for production S → a B S

S→ . a B S
S→ a . B S
S→ a B . S
S→ a B S .

❑ Items denote the idea of the viable prefix. E.g.

➢ S→ . a B S : to be a viable prefix, terminal 'a' needs to be shifted

➢ S→ a . B S : to be a viable prefix, a set of terminals need to be shifted and reduced to non-terminal 'B'

States in the LR Parser

❑ State of LL parser is simply tuple (stack top symbol, input)

❑ State of LR parser is more complex

➢ Must keep track of where we are in the RHS production rules

➢ State is denoted by a set of LR(0) items

❑ Why a *set* of LR(0) items?

➢ There may be multiple candidate RHSs for the prefix. E.g.

For grammar $S \rightarrow a\ b\ |\ a\ c$,

$S \rightarrow a\ .\ b$ and $S \rightarrow a\ .\ c$ would be items in the same set

➢ Even for one RHS, you may need multiple items expressing the same position, if the following symbol is a non-terminal. E.g.

For grammar $S \rightarrow a\ B,\ B \rightarrow b$

$S \rightarrow a\ .\ B$ and $B \rightarrow .\ b$ would be items in the same set

❑ LR parsers keep track of states alongside symbols in stack

Parser Implementation in More Detail

input string

| a_1 | a_2 | | a_i | | a_n | $ |

stack

S_{m+1}	X_{m+1}
S_m	X_m
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

LR Parsing program

| Action | GoTo | Table |

- Each grammar symbol X_i is associated with a state S_i
- Contents of stack $(X_1 X_2 … X_m)$ is a viable prefix
- Contents of stack + input $(X_1 X_2 … X_m a_i … a_n)$ is a right sentential form
  - If the input string is a member of the language
- Uses **state** at the top of stack and current input to index into parsing table to determine whether to shift or reduce

Parser Actions

S_m	X_m
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

…	X_{m+1}	…	$

$S_m \quad : A \rightarrow X_m . X_{m+1}$
$S_{m-1} : A \rightarrow . X_m X_{m+1}$
$\qquad\quad B \rightarrow X_{m-1} . A$

Shift

S_{m+1}	X_{m+1}
S_m	X_m
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

…	X_{m+1}	…	$

$S_{m+1}: A \rightarrow X_m X_{m+1} .$
$S_m \quad : A \rightarrow X_m . X_{m+1}$
$S_{m-1} : A \rightarrow . X_m X_{m+1}$
$\qquad\quad B \rightarrow X_{m-1} . A$

…	X_{m+2}	…	$

Reduce(1)

$S_{m+1}: A \rightarrow X_m X_{m+1} .$
$S_m \quad : A \rightarrow X_m . X_{m+1}$
$S_{m-1} : A \rightarrow . X_m X_{m+1}$
$\qquad\quad B \rightarrow X_{m-1} . A$

S_{m+1}	X_{m+1}
S_m	X_m
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

(2)

	A
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

GOTO

$S_A \quad : B \rightarrow X_{m-1} A .$
$S_{m-1} : A \rightarrow . X_m X_{m+1}$
$\qquad\quad B \rightarrow X_{m-1} . A$

S_A	A
S_{m-1}	X_{m-1}
…	…
S_1	X_1
S_0	---

Parser Actions

❑ Assume configuration = $S_0X_1S_1X_2S_2…X_mS_m\#a_ia_{i+1}…a_n\$$

❑ Actions can be one of:

1. Shift input a_i and push new state **S**
   - New configuration = $S_0X_1S_1X_2S_2…X_mS_m\ a_i\ S\ \#\ a_{i+1}…\$)$
   - Where Action $[S_m, a_i]$ = s[S]

2. Reduce using Rule **R** (A → β) and push new state **S**
   - Let k = | β |, pop 2*k symbols and push A
   - New configuration = $S_0X_1S_1…X_{m-k}S_{m-k}A\ S\ \#\ a_i\ a_{i+1}…\$$
   - Where Action $[S_m, a_i]$ = r[R] and GoTo $[S_{m-k}, A]$ = [S]

3. Accept – parsing is complete (Action $[S_m, a_i]$ = accept)

4. Error – report and stop (Action $[S_m, a_i]$ = error)

Parse Table: Action and Goto

❑ Action $[S_m, a_i]$ can be one of:

- s[S]: shift input symbol a_i and push state S
  (One item in S_m must be of the form $A \rightarrow \alpha \cdot a_i \beta$)

- r[R]: reduce using rule R on seeing input symbol a_i
  (One item in S_m must be R: $A \rightarrow \alpha \cdot$, where $a_i \in$ Follow(A))

  - Use GoTo $[S_{m-|\alpha|}, A]$ to figure out state to push with A

- Accept (One item in S_m must be S' $\rightarrow$ S $\cdot$ where S is the original start symbol, and a_i must be $\$$)

- Error (Cannot shift, reduce, accept on symbol a_i in state S_m)

❑ GoTo $[S_m, X_i]$ is [S]:

- Next state to push when pushing nonterminal X_i from a reduction
  (At least one item in S_m must be of the form $A \rightarrow \alpha \cdot X_i \beta$)

- Similar to shifting input except now we are "shifting" a nonterminal

❑ **Grammar**

1. S→E
2. E→E+T
3. E→T
4. T→id
5. T→(E)

Non-terminal	Follow
S	$
E	+) $
T	+) $

ACTION

	+	id	(	)	$
S0		s3	s4		
S1	s7				accept
S2	r3			r3	r3
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	r2			r2	r2

GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

❑ Grammar
1. S→E
2. E→E+T
3. E→T
4. T→id
5. T→(E)

Non-terminal	Follow
S	$
E	+) $
T	+) $

ACTION

	+	id	(	)	$
S0		s3	s4		
S1	s7				accept
S2	**r3**			**r3**	**r3**
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	**r2**			**r2**	**r2**

GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

Parse Table in Action

❑ Example input string

            id   +   id   +   id

❑ Parser actions

Stack	Input	Actions
S0	id + id + id $	Action[S0, id] (s3): Shift "id", Push S3
S0 id S3	+ id + id $	Action[S3, +] (r4): Reduce rule 4 (T→id) GoTo[S0, T] (2): Push S2
S0 T S2	+ id + id $	Action[S2, +] (r3): Reduce rule 3 (E→T) GoTo[S0, E] (1): Push S1
S0 E S1	+ id + id $	Action[S1, +] (s7): Shift "+", Push S7
S0 E S1 + S7	id + id $	Action[S7, id] (s7): Shift "id", Push S3
…	…	…
S0 E S1 + S7 T S8	+ id $	Action[S8, +] (r2): Reduce rule 2 (E→E+T) GoTo[S0, E] (1): Push S1
…	…	…

Power Added to DFA by Stack

❑ LR parser is basically DFA+Stack (Pushdown Automaton)

❑ DFA: can only remember one state ("dot" in current rule)

❑ DFA + Stack: remembers current state and all past states ("dots" in rules higher up in the tree waiting for non-terminals)

Stack	Input	Action
S0	id + id $	s3
S0 id S3	+ id $	r4, goto[S0, T]
S0 T S2	+ id $	r3, goto[S0, E]
S0 E S1	+ id $	s7
S0 E S1 + S7	id $	s3
S0 E S1 + S7 id S3	$	r4, goto[S7, T]
S0 E S1 + S7 T S8	$	r2, goto[S0, E]
S0 E S1	$	Accept

Power Added to DFA by Stack

❑ Remember the following CFG for the language { $[^i]^i | i >= 1$}?
$S \rightarrow [\ S\ ]\ |\ [\ ]$

❑ Regular grammars (or DFAs) could not recognize language because the state machine had to "count"

❑ LR parsers can use stack to count by pushing as many states as there are [symbols in the input string

❑ Q: Is this language LL(1)?

➢ Yes. After left-factoring.
$S \rightarrow [\ S', S' \rightarrow S\ ]\ |\ ]$

➢ Now stack counts] symbols.

➢ Same pushdown automaton but different usage

S0 S
    S0 [**S1** S   ]
      S0 [**S1** S   ]
        S0 [**S1** S   ]
            ⋮

LR Parse Table Construction

❑ Must be able to decide on action from:
  ➢ State at the top of stack
  ➢ Next k input symbols (In practice, k = 1 is often sufficient)
❑ To construct LR parse table from grammar
  ➢ Two phases
    • Build deterministic finite state automaton to go from state to state
    • Express DFA using Action and GoTo tables
❑ State: Where we are currently in the structure of the grammar
  ➢ Expressed as a set of LR(0) items
  ➢ Each item expresses position in terms of the RHS of a rule

Construction of LR States

1. Create augmented grammar G' for G
   - Given G: $S \rightarrow \alpha \mid \beta$, create G': $S' \rightarrow S$ $S \rightarrow \alpha \mid \beta$
   - Creates a single rule $S' \rightarrow S$ that when reduced, signals acceptance
2. Create first state by performing a *closure* on initial item $S' \rightarrow \cdot S$
   - Closure(I): computes set of items expressing the same position as I
   - Closure($\{S' \rightarrow \cdot S\}$) = $\{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}$
3. Create additional states by performing a *goto* on each symbol
   - Goto(I, X): creates state that can be reached by advancing X
   - If α was single symbol, the following new state would be created:
     Goto($\{S' \rightarrow \cdot S, S \rightarrow \cdot \alpha, S \rightarrow \cdot \beta\}, \alpha$) =
     Closure($\{S \rightarrow \alpha \cdot\}$) = $\{S \rightarrow \alpha \cdot\}$
4. Repeatedly perform gotos until there are no more states to add

Closure Function

❑ Closure(I) where I is a set of items

➢ Returns the state (set of items) that express the same position as I

➢ Items in I are called kernel items

➢ Rest of items in closure(I) are called non-kernel items

❑ Let N be a non-terminal

➢ If dot is in front of N, then add each production for that N and put dot at the beginning of the RHS

- $A \rightarrow \alpha \,.\, B \,\beta$ is in I   ; we expect to see a string derived from B

- $B \rightarrow \,.\, \gamma$ is added to the closure, where $B \rightarrow \gamma$ is a production

- Apply rule until nothing is added

➢ Given $S \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow id \mid (E)$

Closure({ $S \rightarrow .\, E$ }) = { $S \rightarrow .E, E \rightarrow .\, E + T, E \rightarrow .\, T, T \rightarrow .\, id, \; T \rightarrow .\, (E)$ }

Kernel and Non-kernel Items

❑ Two kinds of items

➢ Kernel items

- Items that act as "seed" items when creating a state
- What items act as seed items when states are created?
  - Initial state: S' → . S
  - Additional state: from goto(I, X) so has X at left of dot
- Besides S' → . S, all kernel items have dot in middle of RHS

➢ Non-kernel items

- Items added during the closure of kernel items
- All non-kernel items have dot at the beginning of RHS
- Trivially derivable from kernel items so often not explicitly expressed as part of state to save memory footprint

Goto Function

❑ Goto (I, X) where I is a set of items and X is a symbol

  ➢ Returns state (set of items) that can be reached by advancing X

  ➢ For each $\underline{A \rightarrow \alpha . X \beta}$ in I,

   $\underline{Closure(A \rightarrow \alpha X . \beta)}$ is added to $\underline{goto(I, X)}$

  ➢ X can be a terminal or non-terminal

    • Terminal if obtained from input string by shifting

    • Non-terminal if obtained from reduction

  ➢ Example

    • Goto({T $\rightarrow$. (E)}, ( ) = closure({T $\rightarrow$ (. E)})

❑ Ensures every symbol consumption results in a viable prefix

  ➢ If parser configuration is $\alpha\#\beta$, then goto(I, X) results in $\alpha X\#\beta$'

  ➢ If α was a viable prefix, αX is trivially a viable prefix

Construction of DFA

❑ Algorithm to compute set C (set of all states in DFA)

   void items (G') {

    C = {closure({S→ . S})} // Add initial state to C

    repeat

     for (each state I in C)

      for (each grammar symbol X)

       if (goto(I, X) is not empty and not in C)

        add goto(I, X) to C

    until no new states are added to C

❑ All new states are added through goto(I, X)

   ➢ States transitions are done on symbol X

❑ Example: $S \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow id \mid (E)$

- $S_0 = closure (\{S \rightarrow . E\})$
  $= \{S \rightarrow . E, E \rightarrow . E + T, E \rightarrow . T, T \rightarrow . id, T \rightarrow . (E)\}$
- $goto(S_0, E) = closure (\{S \rightarrow E ., S \rightarrow E . + T\})$
  $S_1 \qquad = \{S \rightarrow E . , S \rightarrow E . + T\}$
- $goto(S_0, T) = closure (\{E \rightarrow T .\})$
  $S_2 \qquad = \{E \rightarrow T . \}$
- $goto(S_0, id) = closure (\{T \rightarrow id .\})$
  $S_3 \qquad = \{T \rightarrow id . \}$
- ……
- $S_8 = …$

❏ DFA for the previous grammar (* are closures applied to kernel items)

S0
```
* S→ . E
  E→ . E+T
  E→ . T
  T→ . Id
  T→ . (E)
```

—E→ **S1**
```
* S→E .
* E→E .+T
```

—+→ **S7**
```
* E→E+ .T
  T→ . id
  T→ . (E)
```

—T→ **S8**
```
* E→E+T.
```

From S7: —id→ **S3**, —(→ **S4**

From S0: —T→ **S2**
```
* E→ T .
```

From S0: —id→ **S3**
```
* T→ id .
```

From S0: —(→ **S4**
```
* T→ ( . E)
  E→ . E+T
  E→ . T
  T→ . Id
  T→ . (E)
```

—E→ **S5**
```
* T→ (E . )
* E→ E . +T
```

—)→ **S6**
```
* T→ (E) .
```

From S4: —T→ **S2**, —id→ **S3**, —(→ **S4**

From S5: —+→ **S7**

Building Parse Table from DFA

- ACTION [state, input symbol/terminal symbol]
- GOTO [state, non-terminal symbol]

❑ ACTION:

1. If $[A \rightarrow \alpha \bullet a\beta]$ is in S_i and $goto(S_i, a) = S_j$, where "a" is a terminal
     then $ACTION[S_i, a] = shift\ j$

2. If $[A \rightarrow \alpha \bullet]$ is in S_i
     then $ACTION[S_i, a] = reduce\ A \rightarrow \alpha$ for all $a \in Follow(A)$

3. If $[S' \rightarrow S_0 \bullet]$ is in S_i
     then $ACTION[S_i, \$] = accept$

   ➢ If no conflicts among 1 and 2
        then it is said that this parser is able to parse the given grammar

❑ GOTO

1. if $goto(S_i, A) = S_j$ then $GOTO[S_i, A] = S_j$

❑ All entries not filled are errors

❑ Grammar

1. S→E
2. E→E+T
3. E→T
4. T→id
5. T→(E)

Non-terminal	Follow
S	$
E	+) $
T	+) $

ACTION

	+	id	(	)	$
S0		s3	s4		
S1	s7				accept
S2	r3			r3	r3
S3	r4			r4	r4
S4		s3	s4		
S5	s7			s6	
S6	r5			r5	r5
S7		s3	s4		
S8	r2			r2	r2

GOTO

	E	T	S
S0	1	2	
S1			
S2			
S3			
S4	5	2	
S5			
S6			
S7		8	
S8			

Types of LR Parsers

❑ SLR – simple LR (what we saw so far was SLR(1))

  ➢ Easiest to implement

  ➢ Not as powerful

❑ Canonical LR

  ➢ Most powerful

  ➢ Expensive to implement

❑ LALR

  ➢ Look ahead LR

  ➢ In between the 2 previous ones in power and overhead

Overall parsing algorithm is the same – table is different

Conflict

❑ Consider the grammar G

$S \rightarrow A\ b\ c\ |\ B\ b\ d$

$A \rightarrow a$ $b \in Follow(A)$ and also $b \in Follow(B)$

$B \rightarrow a$

S_0: $S \rightarrow .Abc$, $S \rightarrow .Bbd$, $A \rightarrow .a$, $B \rightarrow .a$ — a → S_1: $A \rightarrow a.$, $B \rightarrow a.$

❑ What is reduced when "a b" is seen? reduce to A or B?

➢ Reduce-reduce conflict

❑ G is not SLR(1) but SLR(2)

➢ We need 2 symbols of look ahead to look past b:

    b c – reduce to A

    b d – reduce to B

➢ Possible to extend SLR(1) to k symbols of look ahead – allows larger class of CFGs to be parsed

SLR(k)

❑ Extend SLR(1) definition to SLR(k) as follows

let $\alpha, \beta \in V^*$

➢ $First_k(\alpha) = \{ x \in V_T^* | (\alpha \rightarrow^* x\beta$ and $|x| <= k)\}$ gives
- all terminal strings of size $<= k$ derivable from α
- all k-symbol terminal prefixes of strings derivable from α

➢ $Follow_k(B) = \{w \in V_T^* | S \rightarrow^* \alpha B\gamma$ and $w \in First_k(\gamma)\}$ gives
- all k symbol terminal strings that can follow B in some derivation
- all shorter terminal strings that can follow B in a sentential form

Parse Table

Let S be a state and lookahead $b \in V_T^*$ such that $|b| \leq k$

1. If $A \rightarrow \alpha. \in S$ and $b \in Follow_k(A)$ then
   - Action(S,b) – reduce using production $A \rightarrow \alpha$,
2. If $D \rightarrow \alpha.a\ \gamma \in S$ and $a \in V_T$ and $b \in First_k(a\ \gamma\ Follow_k(D))$
   - Action(S,b) = shift "a" and push state goto(S,a)

   For k =1, this definition reduces to SLR(1)
       Reduce: Trivially true
       Shift: $First_1(a\ \gamma\ Follow_1(D)) = \{a\}$

SLR(k)

❑ Consider

$$S \rightarrow A\ b^{k-1}\ c \mid B\ b^{k-1}\ d$$
$$A \rightarrow a$$
$$B \rightarrow a$$

SLR(k) not SLR(k-1)

➤ cannot decide what to reduce,

➤ reduce a to A or B depends the next k symbols

$$b^{k-1}\ c\ \ or\ b^{k-1}\ d$$

Non SLR(k)

❑ Consider another Grammar G

- $S \rightarrow j\ A\ j\ |\ A\ m\ |\ a\ j$

  $A \rightarrow a$

Conflict

S1

| $S \rightarrow a.j$ |
| $A \rightarrow a.$ |

j → Note only m can follow A

S0

| $S \rightarrow .jAj$ |
| $S \rightarrow .Am$ |
| $S \rightarrow .aj$ |
| $A \rightarrow .a$ |

a →

j →

S2

| $S \rightarrow j.Aj$ |
| $A \rightarrow .a$ |

a → Note only j can follow A

A →

Follow(A) = {j, m}

State S1: [A→a.] – reduce using this production (on j or m)

[S→a.j] – shift j ➔ shift-reduce conflict ➔ not SLR(1)

? SLR(k)?

For reducing A→a.: $Follow_k(A) = First_k(j) + First_k(m) = \{j, m\}$,

For shifting S →a.j: $First_k(jFollow_k(S)) = \{j\}$ so not SLR(k) for any k !!!

Why?

❑ Look ahead is too crude
  ➢ In S1, if A → a is reduced then {m} is the only possible symbol that can be seen – the only valid look ahead
  ➢ Fact that {j} can follow A in another context is irrelevant

❑ Want to compare look ahead in a state to those symbols that might actually occur in the context represented by the state.

❑ Done in Canonical LR !!!
  ➢ Determine look ahead appropriate to the context of the state
  ➢ States contains a look ahead – will be used only for reductions

SLR(1)

X
Y → (A → B.)

Z
→ (…, A → B.)

Follow(A)={a,b,c}

LALR(1)

X
Y → (A → B. ,a/b)

Z
→ (…, A → B. ,c)

state merging

LR(1)

X → (A → B. ,a)

Y → (A → B. ,b)

Z → (…, A → B. ,c)

state splitting

Constructing Canonical LR

❑ Problem: Follow set in SLR is not precise enough

➢ Follow set ignores context where reduction for item occurs

❑ Solution: Define a more precise follow set

➢ For each item, encode a more precise follow set according to context

➢ Use more precise follow set when deciding whether to reduce item

❑ LR(1) item: LR item with one lookahead

➢ [A→α.β, a] where A→αβ is a production and a is a terminal or $

- Meaning: Only terminal a can follow A in this context

- Interpretation: After β is shifted and eventually we reach [A→αβ., a], only reduce A if lookahead matches terminal a

➢ Second lookahead component will always be a subset of Follow(A)

Constructing Canonical LR

❑ Essentially the same as LR(0) items only adding lookahead
  ➢ Modify closure and goto function

❑ Changes for **closure**
  ➢ $[A \rightarrow \alpha.B\beta, a]$ and $B \rightarrow \delta$ then

  $[B \rightarrow . \delta, c]$ where $c \in First(\beta a)$

❑ Changes for **goto** function
  ➢ Carry over look ahead

  $[A \rightarrow \alpha.X\beta, a] \in I$ then goto $(I, X) = [A \rightarrow \alpha X. \beta, a]$

Example

❑ Grammar
$$S' \rightarrow S$$
$$S \rightarrow CC$$
$$C \rightarrow eC \mid d$$

❑ S0: closure(S' →.S, $)

    [S' →.S,$]

    [S→.CC, $]   first($\varepsilon$$)={$}
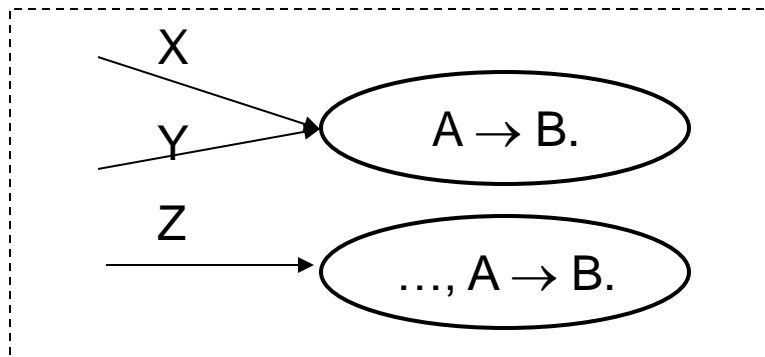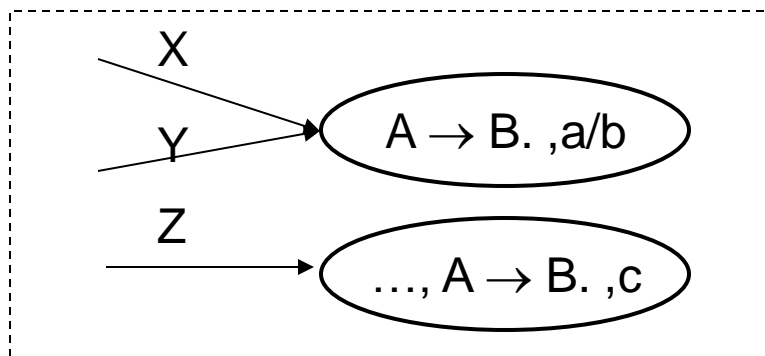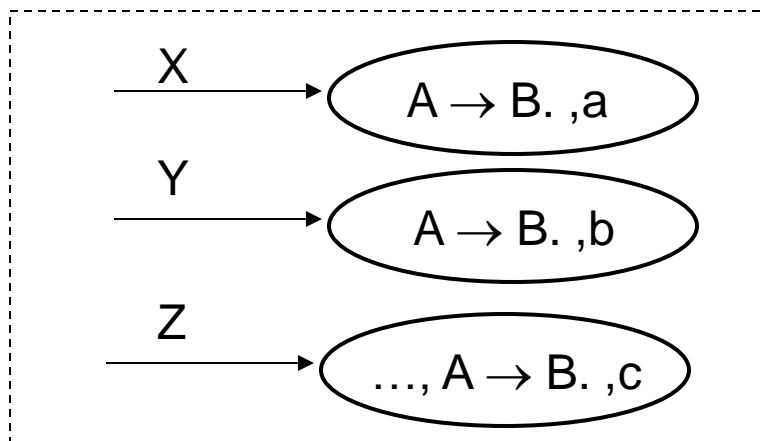
    [C→.eC, e/d]   first(C$)={e,d}

    [C→.d, e/d]   first(C$)={e,d}

❑ S1: goto(S0, S) = closure(S' →S., $)

    [S' →S. , $]

❑ S2: goto(S0, C) = closure(S →C.C, $)

    [S →C.C, $]

    [C →.eC, $]   first($\varepsilon$$)={$}

    [C →.d, $]   first($\varepsilon$$)={$}

❑ S3: goto(S0,e) = closure(C→e.C, e/d)

       [C →e.C, e/d]

       [C →.eC, e/d]   first(εe/d) = {e,d}

       [C →.d, e/d]   first(εe/d) = {e,d}

❑ S4: goto(S0, d) = closure(C→d., e/d)

       [C→d., e/d]

❑ S5: goto(S2, C) = closure(S→CC., $)

       [S→CC., $]

❑ S6: goto(S2,e) = closure(C→e.C, $)

       [C →e.C, $]

       [C →.eC, $]   first($\varepsilon$$) = {$}

       [C →.d, $]   first($\varepsilon$$) = {$}

❑ S7: goto(S2, d) = closure(C→d., $)

       [C→d., $]

❑ S8: goto(S3, C) = closure(C→eC., e/d)

       [C→eC., e/d]

❑ S9: goto(S6,C) = closure(C→eC., $)

       [C→eC., $]
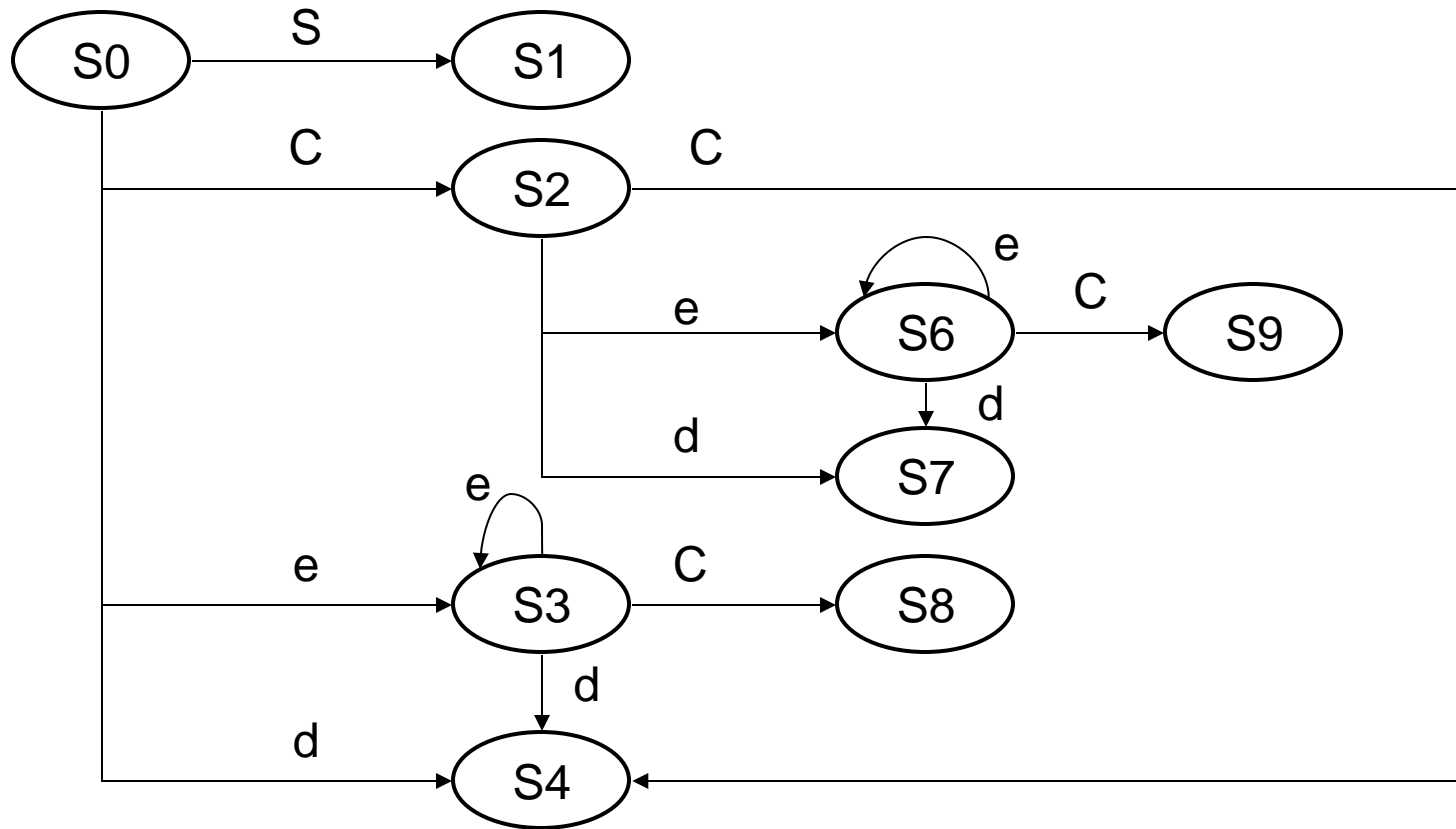
Note S3, S6 are same except for lookahead (also true for S4, S7 and S8, S9)
In SLR(1) – one state represents both

Constructing Canonical LR Parse Table

❑ Shifting: same as before

❑ Reducing:

➢ Don't use follow set (too coarse grain)

➢ Reduce only if input matches lookahead for item

❑ Action and GOTO

1. if [A→α•aβ,b] ∈Si and goto(Si, a) = Sj,
   Action[I,a] = s[Sj] – shift and goto state j if input matches a
   Note: same as SLR

1. if [A→α•,a] ∈Si
   Action[I,a] = r[R] – reduce R: A→α if input matches a
   Note: for SLR, reduced if input matches Follow(A)

2. if [S' →S., $] ∈Si,
   Action[i,$] = accept

❑ Revisit SLR and LR

➢ S→aEa | bEb | aFb | bFa

E→e
F→e
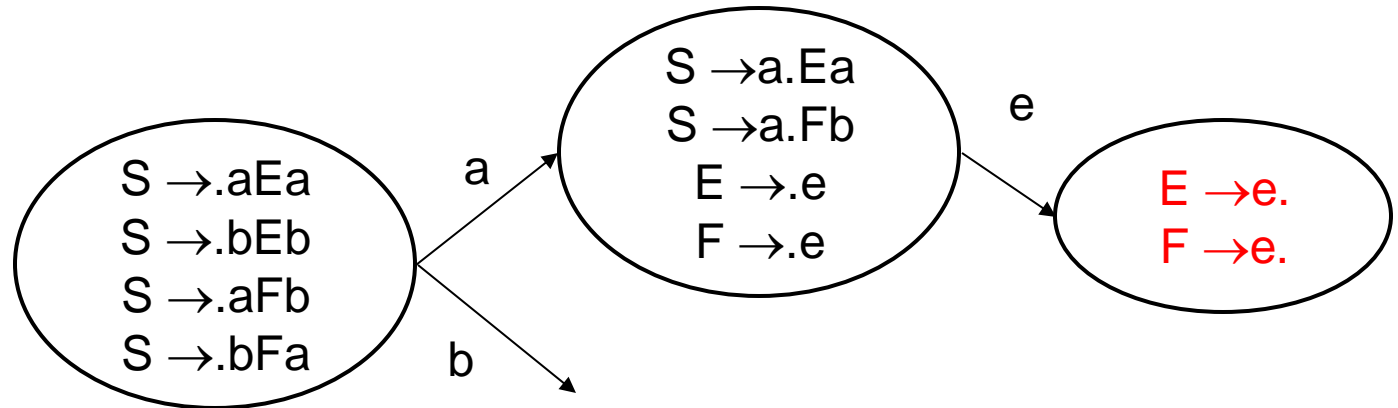
❑ SLR: reduce/reduce conflict   Follow(E,F) = {a,b}

aea ⇒ E→e   and   bea ⇒ F→e

❑ Is this LR(1)? Will not have a conflict because states will be split to take into account this context

E if followed by a/b preceded by a/b, respectively

F if followed by a/b preceded by b/a, respectively

❑ SLR: Follow(E) = Follow(F) = {a,b}

S →.aEa
S →.bEb
S →.aFb
S →.bFa

a →

S →a.Ea
S →a.Fb
E →.e
F →.e

e →

E →e.
F →e.

b →

S →b.Eb
S →b.Fa
E →.e
F →.e

e →

E →e.
F →e.
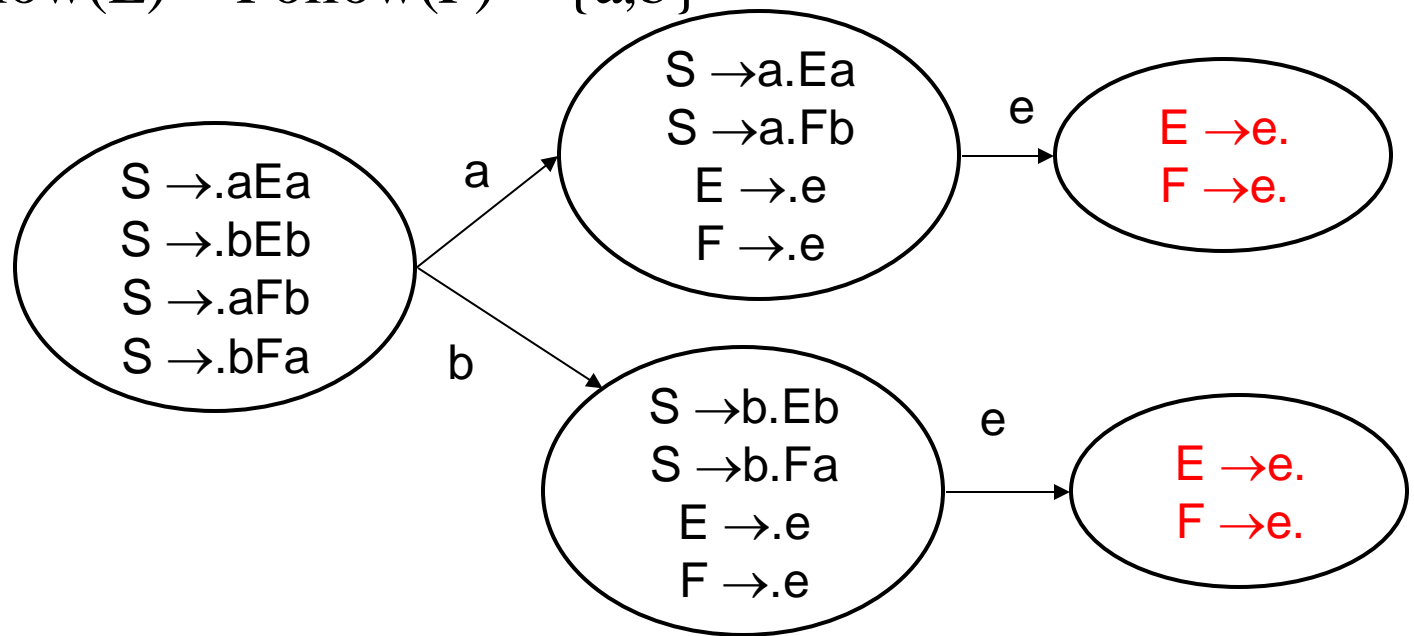
❑ LR: Follow sets more precise

S0

a →

S1

e →

S3

[E→e., a]
[F→e., b]

b →

S2

e →

S4

[E→e., b]
[F→e., a]
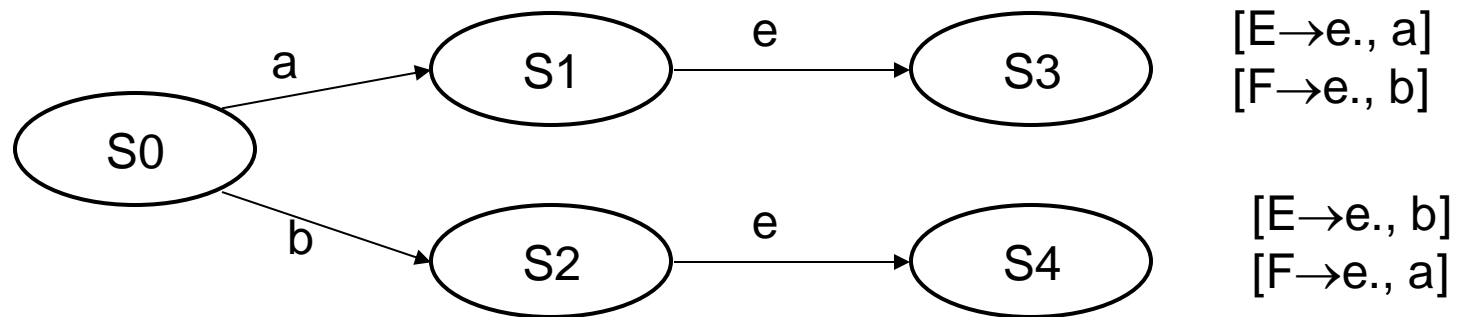
SLR(1) and LR(1)

❑ LR(1) more powerful than SLR(1) – can parse more grammars

❑ But LR(1) may end up with many more states than SLR(1)

➢ One LR(0) item may split up to many LR(1) items
(As many as all combinations of lookahead possible – potentially powerset of entire alphabet)

❑ LALR(1) – compromise between LR(1) and SLR(1)

➢ Constructed by merging LR(1) states with the same core

• Ends up with same number of states as SLR(1)

• But items still retain some lookahead info – still better than SLR(1)

➢ Used in practice because most programming language syntactic structures can be represented by LALR (not true for SLR)
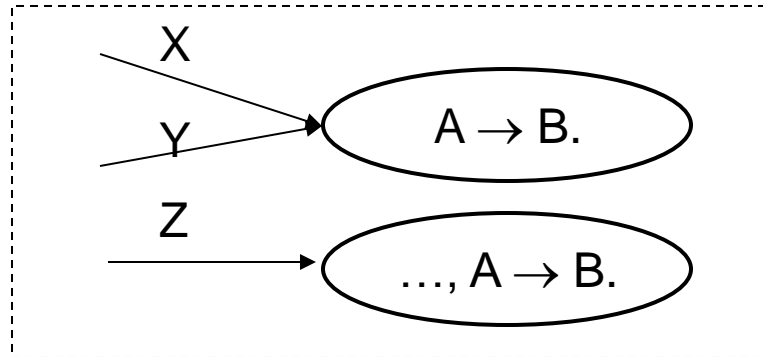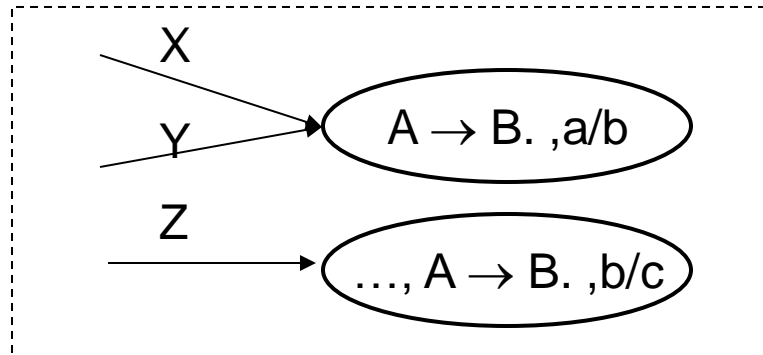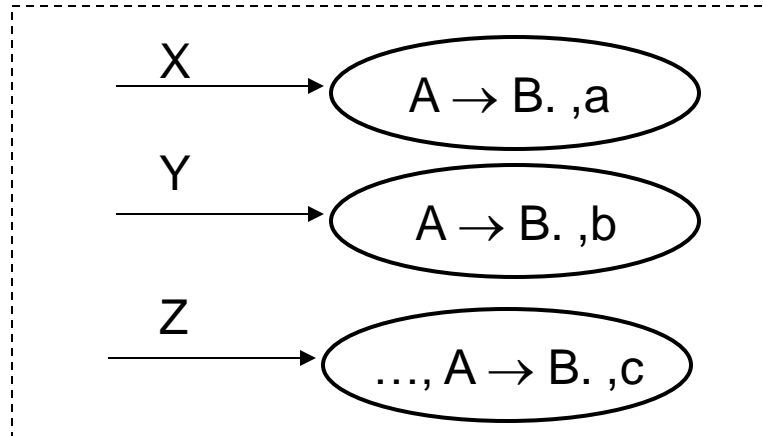
Example

❑ Grammar

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow eC \mid d$

S3: goto(S0,e)=closure(C→e.C, e/d)

               [C →e.C, e/d]

               [C →.eC, e/d]

               [C →.d, e/d]

S4: goto(S0, d)=closure(C→d., e/d)

               [C→d., e/d]

S8: goto(S3, C)=closure(C→eC., e/d)

               [C→eC., e/d]

S6: goto(S2,e)=closure(C→e.C, $)

               [C →e.C, $]

               [C →.eC, $]

               [C →.d, $]

S7: goto(S2, d)=closure(C→d., $)

               [C→d., $]

S9: goto(S6,C)=closure(C→eC., $)

               [C→eC., $]
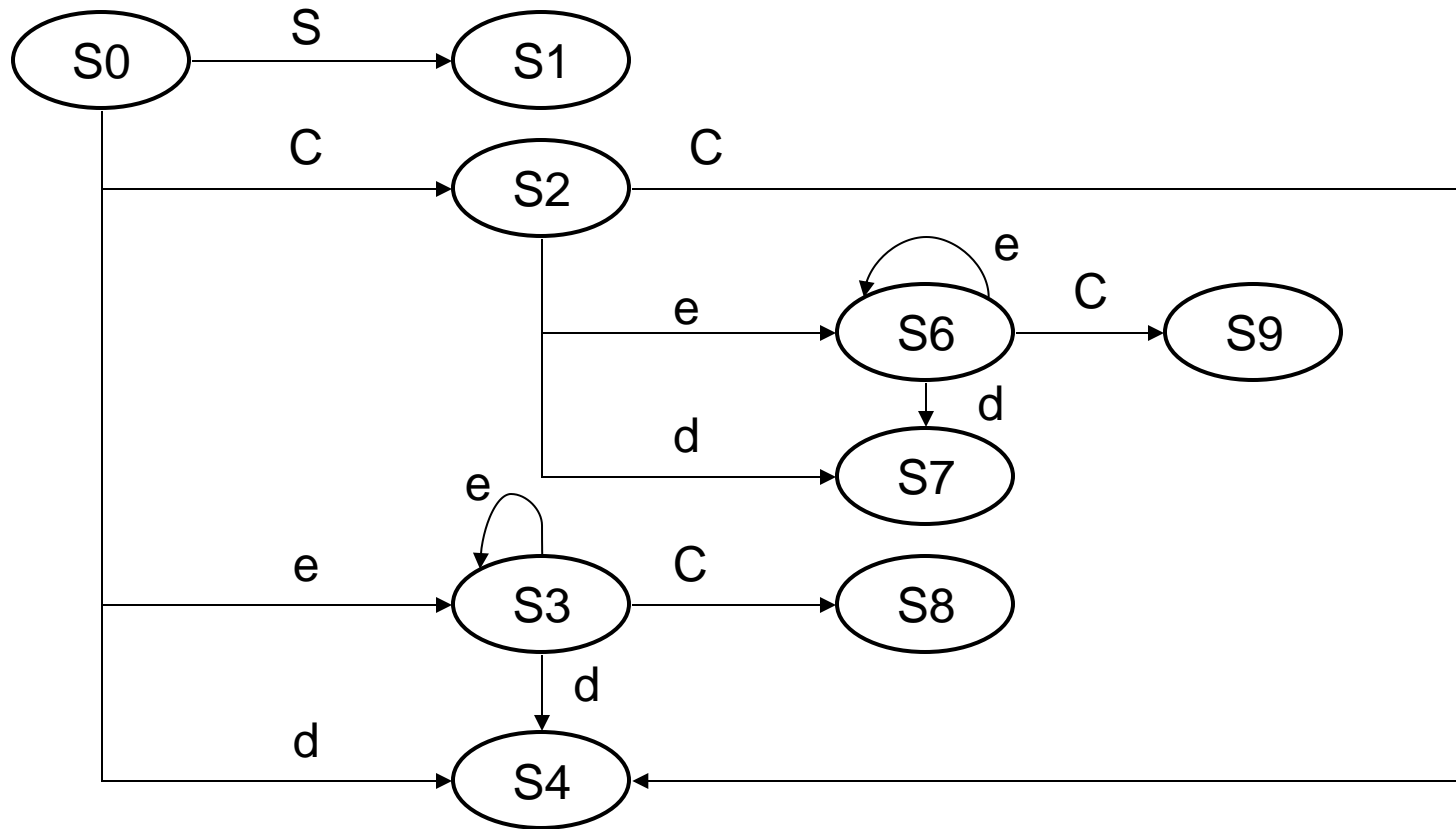
Note S3 and S6 are the same except for look ahead (same for S4 and S7)
In SLR(1) – one state represents both

Merging states

❑ Can merge S3 and S6

S3: goto(S0,e)=closure(C→e.C, e/d)	S6: goto(S2,e)=closure(C→e.C, $)
[C →e.C, e/d]	[C →e.C, $]
[C →.eC, e/d]	[C →.eC, $]
[C →.d, e/d]	[C →.d, $]

S36:   [C →e.C, e/d/$]
            [C →.eC, e/d/$]
            [C →.d, e/d/$]

❑ Similarly

- • S47: [C→d., e/d/$]
- • S89: [C→eC., e/d/$]

Effects of Merging

1. Detection of errors may be delayed

   ➤ On error, LALR parsers will not perform shifts beyond an LR parser but may perform more reductions before finding error

   ➤ Example: $S' \rightarrow S$ $S \rightarrow CC$

                              $C \rightarrow eC \mid d$

            and   input string eed$

   • Canonical LR: Parse Stack S0 e S3 e S3 d S4

                            State S4 on $ input = error S4:{$C \rightarrow d.$, e/d}

   • LALR:

            stack: S0 e S<u>36</u> e S<u>36</u> d S<u>47</u>   ➔ state S<u>47</u> input $, reduce $C \rightarrow d$

            stack: S0e S<u>36</u> e S<u>36</u> C S<u>89</u>   ➔ reduce $C \rightarrow eC$

            stack: S0 e S<u>36</u> C S<u>89</u>   ➔ reduce $C \rightarrow eC$

            stack: S0 C S2   ➔ state S2 on input $, error

Effects of Merging

2. Merging of states can introduce conflicts
   - cannot introduce shift-reduce conflicts
   - can introduce reduce-reduce conflicts

❑ Shift-reduce conflicts

Suppose Sij: $[A \rightarrow \alpha., a]$ reduce on input a
$[B \rightarrow \beta.a\delta, b]$ shift on input a

formed by merging Si and Sj

Cores are the same for Si, Sj and one of them must contain

$[A \rightarrow \alpha., a]$ and $[B \rightarrow \beta.a\delta, b]$

➔ shift-reduce conflicts were already present in either Si and Sj (or both) and not newly introduced by merging

Reduce-reduce Conflicts

$S \rightarrow aEa \mid bEb \mid aFb \mid bFa$

$E \rightarrow e$

$F \rightarrow e$

S3: $[E \rightarrow e., a]$ viable prefix ae

        $[F \rightarrow e., b]$

S4: $[E \rightarrow e., b]$ viable prefix be

        $[F \rightarrow e., a]$

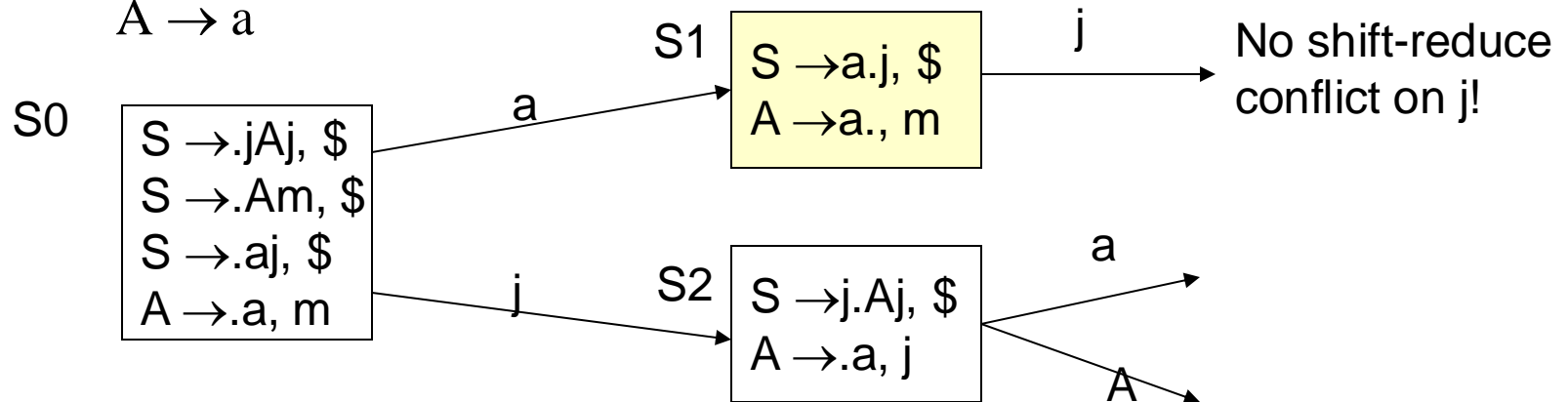Merging   S34: $[E \rightarrow e., a/b]$

                      $[F \rightarrow e., a/b]$

- both reductions are called on inputs a and b,
  i.e. <u>reduce-reduce conflict</u>

Non SLR(k) but LALR(1)

❏ Let's consider the Non SLR(k) Grammar G again

- $S \rightarrow j A j \mid A m \mid a j$
  $A \rightarrow a$

S0

```
S →.jAj, $
S →.Am, $
S →.aj, $
A →.a, m
```

S1

```
S →a.j, $
A →a., m
```

No shift-reduce conflict on j!

S2

```
S →j.Aj, $
A →.a, j
```

Follow(A) = {j, m}

State S1: $[A \rightarrow a.]$ – reduce using this production (on j or m)

$[S \rightarrow a.j]$ – shift j ➜ not SLR(1)

But LALR(1)

For reducing $A \rightarrow a.$: lookahead must be {m}

For shifting $S \rightarrow a.j$: $First_k(jFollow_k(S)) = \{j\}$ ➜ so LALR(1)

Construction on LALR Parser

❑ One solution:

➢ Construct LR(1) states

➢ Merge states with same core

➢ if no conflicts, you have a LALR parser

❑ Inefficient because of building LR(1) items are expensive in time and space (then what is the point of using LALR?)

❑ Efficient construction of LALR parsers

➢ Avoids initial construction of LR(1) states

➢ Merges states on-the-fly (step-by-step merging)

• States are created as in LR(1)

• On state creation, immediately merge if there is an opportunity

Compaction of LALR Parse Table

❑ A typical language grammar with 50-100 terminals and 100 productions may have an LALR parser with several hundred states and thousands of action entries

➤ Often multiple rows of table are identical so share the rows
- Make states point to an array of unique rows

➤ Often most entries in a row are empty
- Instead of an actual row array, use row lists of (input, action) pairs

➤ Slows access to the table but can reduce memory footprint

Error Recovery

❑ Error recovery: How does the parser uncover multiple errors?

❑ Error detected when parser consults parsing table and hits an empty entry

➢ Compared to LR, SLR and LALR parser may go through several reductions before detecting an error

- Due to more coarse-grained use of lookahead
- But never shifts beyond an erroneous symbol

❑ Simple error recovery (by discarding offending code sequence)

1. Decide on non-terminal A: candidate for discarding

- Typically an expression, statement, or block

2. Continue to scan down the stack until a state S with a goto on a particular non-terminal A is found

3. Discard input symbols until a symbol 'a' is found that can follow A

- E.g. if A is a statement then 'a' would be ';'

4. Push state Goto[a,A] on stack and continue parsing

A Hierarchy of Grammar Classes
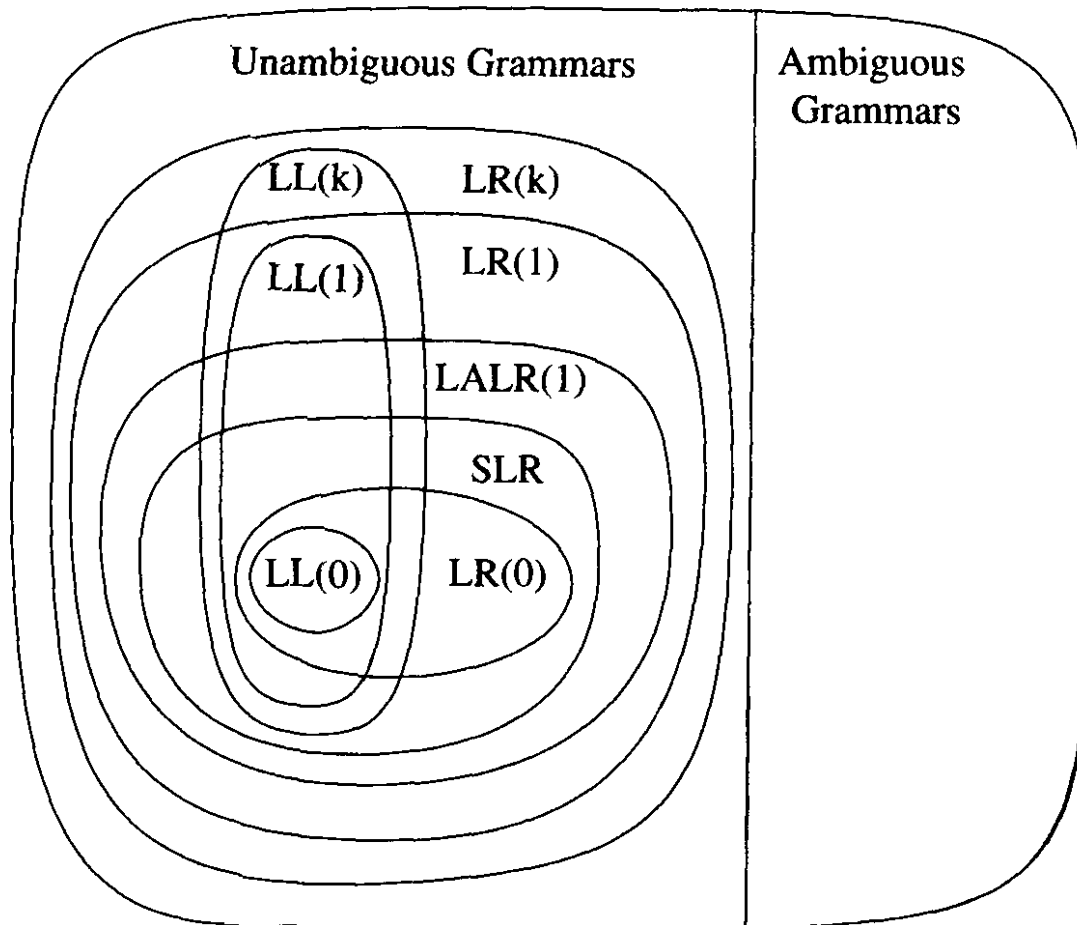
LALR vs. LR Parsing (LALR < LR)

❑ LR(k) is strictly more powerful compared to LALR(k)

➢ LALR merges and reduces number of states.

❑ Slightly unintuitive since unlike LL, LR since no formal definition exists on what is LALR

➢ Definition by construction: if LALR parser has no conflicts

➢ If there is a reduce-reduce conflict, is it because of merging?

❑ However, LALR(1) has become a standard for programming languages and for parser generators

➢ YACC, Bison, etc.

➢ Most PLs have an LALR(1) grammar

➢ Reduce-reduce conflicts due to merging are rare (mostly due to ambiguity)

LL vs. LR Parsing (LL < LR)

❑ LL(k) parser, each expansion A→α is decided on the basis of
  ➢ Current non-terminal at the top of the stack
    • Which LHS to produce
  ➢ k terminals of lookahead at *beginning* of RHS
    • Must guess which RHS by peeking at first few terminals of RHS

❑ LR(k) parser, each reduction A→α• is decided on the basis of
  ➢ RHS at the top of the stack
    • Can postpone choice of RHS until entire RHS is seen
    • Common left factor is okay – waits until entire RHS is seen anyway
    • Left recursion is okay – does not impede forming RHS for reduction
  ➢ k terminals of lookahead *beyond* RHS
    • Can decide on RHS after looking at entire RHS plus lookahead

LL vs. SLR Parsing (LL != SLR)

❑ Neither is strictly more powerful than the other

❑ Advantage of SLR: can delay decision until entire RHS seen

➢ LL must decide RHS with a few symbols of lookahead

❑ Disadvantage of SLR: lookahead applied out of context

➢ Consider grammar: $S \rightarrow Bb \mid Cc \mid aBc$, $B \rightarrow \varepsilon$, $C \rightarrow \varepsilon$

➢ Initial state $S_0 = \{S \rightarrow . Bb \mid . Cc \mid . aBc, B \rightarrow ., C \rightarrow .\}$

➢ For SLR(1), reduce-reduce conflict on $B \rightarrow .$ and $C \rightarrow .$

• Follow(B) = {b, **c**} and Follow(C) = {**c**}

➢ For LL(1), no conflict

• First(Bb) = {b}, First(Cc) = {c}, First(aBC) = {a}

❑ For the same reason, LL != LALR

LR(0) == LALR(0) == SLR(0) > LL(0)

❑ LR(0) == LALR(0) == SLR(0)

➢ No lookahead for reducing

- Lookahead components are meaningless (hence LR==LALR==SLR)
- Must reduce regardless of Follow sets
- If a state contains a reduce item, there can be no other reduce items or shift items for that state, or there will be a conflict

➢ Makes grammars very restrictive. Not used very much.

❑ LL(0) < LR(0)

➢ LL(0) can only have one RHS per non-terminal to avoid conflict

➢ LR(0) can still have multiple RHSs per non-terminal

➢ E.g. S → a | b is not LL(0) but is LR(0)

L(Rec. Descent) == L(GLR) == L(CFG)

❑ L(Recursive Descent) == L(CFG)

➢ Can parse all CFGs by trial-and-error until input string match

➢ Including ambiguous CFGs (accepts first encountered parse tree)

➢ A general top-down parser for all CFGs can be constructed by using LL(k) parsing table, and falling back on recursive descent

❑ Does that make top-down parsers superior to bottom-up?

➢ No. Same trial-and-error strategy can be employed for bottom-up

➢ GLR (Generalized LR) parser: a parser for all CFGs that relies on an LR parsing table, but falls back on trial-and-error on conflict

➢ L(GLR(k)) == L(CFG)

➢ Any LR table (e.g. SLR, LALR, Canonical LR) can be used

➢ GLR implementations: GNU Bison etc. (but not Yacc)

Using Automatic Tools
-- YACC

Pitt, CS 2210

Using a Parser Generator

❑ YACC is an LALR(1) parser generator

  ➢ YACC: Yet Another Compiler-Compiler

❑ YACC constructs an LALR(1) table and reports an error when a table entry is multiply defined

  ➢ A shift and a reduce – reports shift/reduce conflict

  ➢ Multiple reduces – reports reduce/reduce conflict

  ➢ Most conflicts are due to ambiguous grammars

  ➢ Must resolve conflicts

    • By specifying associativity or precedence rules

    • By modifying the grammar

    • YACC outputs detail about where the conflict occurred (by default, in the file "y.output")

Shift/Reduce Conflicts

❑ Typically due to ambiguities in the grammar

❑ Classic example: the dangling else

  $S \rightarrow$ if E then S | if E then S else S | OTHER

will have DFA state containing

  $[S \rightarrow$ if E then S. , else]

  $[S \rightarrow$ if E then S. else S , else]

so on 'else' we can shift or reduce

❑ Default (YACC, bison, etc.) behavior is to shift

  ➢ Default behavior is the correct one in this case

  ➢ Better not to rely on this and remove ambiguity

More Shift/Reduce Conflicts

❑ Consider the ambiguous grammar

$$E \rightarrow E+E \mid E*E \mid int$$

we will have the states containing

$[E \rightarrow E* . E, +/*]$ $[E \rightarrow E*E . , +/*]$

$[E \rightarrow . E+E, +/*]$ $\overset{E}{\Rightarrow}$ $[E \rightarrow E . +E, +/*]$

...   ...

Again we have a shift/reduce conflict on input +

- In this case, we need to reduce (* is higher than +)
- Easy (better) solution: declare precedence rules for * and +
- Hard solution: rewrite grammar to be unambiguous

More Shift/Reduce Conflicts

❑ Declaring precedence and associativity in YACC

%left '+' '-'

%left '*' '/'

➢ Interpretation:

- +, -, *, / are left associative
- +, - have lower precedence compared to *, / (associativity declarations are in the order of increasing precedence)
- Precedence of a candidate rule for reduction is the precedence of the last terminal in that rule (e.g. For 'E→ E+E **.**' , level is same as '+')

➢ Resolve shift/reduce conflict with a shift if:

- No precedence declared for either rule or terminal
- Input terminal has higher precedence than the rule
- The precedence levels are the same and right associative

Use Precedence to Solve S/R Conflict

$$[E \rightarrow E^* . E, +/*] \qquad [E \rightarrow E^*E ., +/*]$$

$$[E \rightarrow . E+E, +/*] \overset{E}{\Rightarrow} [E \rightarrow E. +E, +/*]$$

$$... \qquad\qquad\qquad ...$$

❑ we will choose reduce because precedence of rule
$E \rightarrow E^*E$ is higher than that of terminal +

$$[E \rightarrow E+ . E, +/*] \qquad [E \rightarrow E+E ., +/*]$$

$$[E \rightarrow . E+E, +/*] \overset{E}{\Rightarrow} [E \rightarrow E. +E, +/*]$$

$$... \qquad\qquad\qquad ...$$

❑ we will choose reduce because $E \rightarrow E+E$ and + have the same
precedence and + is left-associative

❑ Back to our dangling else example

   [S → if E then S. , else]

   [S → if E then S. else S, else]

- Can eliminate conflict by declaring 'else' with higher precedence than 'then'

- But this looks much less intuitive compared to arithmetic operator precedence

- Best to avoid overuse of precedence declarations that do not enhance the readability of your code

Reduce/Reduce Conflicts

❑ Usually due to ambiguity in the grammar

❑ Example: a sequence of identifiers

$S \rightarrow \varepsilon \mid id \mid id\ S$

There are two parse trees for the string 'id'

$S \rightarrow id$

$S \rightarrow id\ S \rightarrow id$

How does this confuse the parser?

Reduce/Reduce Conflicts

❑ Consider the states

[S'→.S , $]   [S →id. , $]

[S → . , $]   [S →id.S, $]

[S → . id, $] $\overset{id}{\Rightarrow}$ [S →. , $]

[S → . id S, $]   [S →. id, $]

                                               [S →. id S, $]

Reduce/reduce conflict on input "id$"

    S' → S → id

    S' → S → id S → id

Better rewrite the grammar: S → ε | id S

Semantic Actions

❑ Semantic actions are implemented for LR parsing

➢ keep attributes on the semantic stack – parallel to the parse stack

- on shift a, push attribute for a on semantic stack

- on reduce $X \rightarrow \alpha$

- pop attributes for α

- compute attribute for X

- push it on the semantic stack

❑ Creating an AST

➢ Bottom up

➢ Create leaf node from attribute values of token(s) in RHS

➢ Create internal node from subtree(s) passed on from RHS

Performing Semantic Actions

❑ Compute the value

$$E \to T + E1 \qquad\qquad \{E.val = T.val + E1.val\}$$

$$| \; T \qquad\qquad \{E.val = T.val\}$$

$$T \to int * T1 \qquad\qquad \{T.val = int.val * T1.val\}$$

$$| \; int \qquad\qquad \{T.val = int.val\}$$

consider the parsing of the string 3 * 5 + 8

❑ Recall: creating the AST

$$E \to int \qquad E.ast = mkleaf(int.lexval)$$

$$| \; E1+E2 \qquad E.ast = mktree(plus, E1.ast, E2.ast)$$

$$| \; (E1) \qquad E.ast = E1.ast$$

➢ a bottom-up evaluation of the ast attribute:

E.ast = mktree(plus, mkleaf(5),

mktree(plus, mkleaf(2), mkleaf(3)))
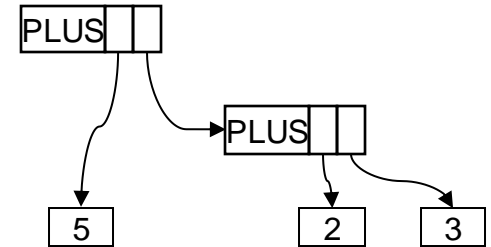
Notes on Parsing

❑ Parsing

  ➢ A solid foundation: CFG

  ➢ A simple parser: LL(1)

  ➢ A more powerful parser: LR(1)

  ➢ An efficient compromise: LALR(1)

  ➢ LALR(1) parser generators