Bottom up Parsing - introduction

- Parsing technique where a string is recognized by constructing a *right-most* derivation in *reverse*
  Start from the leaves and work up towards root $\Longrightarrow$ reduce a string to a NT.
- Bottom up parsing is appealing:
  - Can be constructed for virtually all programming constructs
  - Most general non-backtracking shift-reduce method known
  - More powerful than predictive parsers
  - Detect a syntactic error as soon as possible to do on a left-to-right scan of input
  - More powerful than top-down parsers:
    - Left recursion or left factoring not a problem.
  - Construction though is much more complex.
- Three kinds of bottom up parsers (in the order of power) (all called LR parsers)
  1. SLR (Simple left to right)
  2. LALR (Lookahead LR)
  3. LR

How does bottom up parsing work? - an example

- Bottom up parsers: stack, table, input buffer + driver.
  Different LR(1) parsers differ in nature of table only. Rest same. Power comes from accuracy of table.
- Two possible actions:
  1. **shift** a terminal from input to stack
  2. **reduce** a string α

  Parsers called *shift-reduce* parsers.
- Grammar:
  $S' \rightarrow S$
  $S \rightarrow (S)S \mid \epsilon$
- Parse steps for input string: `()`

Parsing Stack	Input	Action
$	()$	shift
$(	)$	reduce $S \rightarrow \epsilon$
$(S	)$	shift
$(S)	$	reduce $S \rightarrow \epsilon$
$(S)S	$	reduce $S \rightarrow (S)S$
$ S	$	reduce $S' \rightarrow S$
$ S'	$	accept

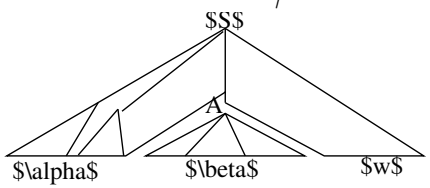
- At each point: i) should it shift or reduce? ii) if reduce, then which rule to use?

Bottom up parsers

- Note: recognition of string through right most derivation (in reverse order):

  S'
  $\overset{rm}{\Longrightarrow}$ $\underline{S}$
  $\overset{rm}{\Longrightarrow}$ $(S)\underline{S}$
  $\overset{rm}{\Longrightarrow}$ $(\underline{S})$
  $\overset{rm}{\Longrightarrow}$ $()$

- **Right-sentential form (rsf):** string that can be rhs of a rule.
  Example: (S)
  Right-sentential form usually split in stack and input.
- **Viable prefix:** Sequence of symbols on parsing stack.
  Example: (, (S, (S) are all viable prefixes of rsf (S).
- Bottom-up parser shifts symbols in stack until it knows it has the valid right hand, so it can reduce.
  $\Longrightarrow$ some mechanism for looking into stack (achieved through notion of states)
- **Handle:** A string that matches right side of a rule + usage of rule to reduce helps in final reduction.
  $A \overset{*}{\Rightarrow} \alpha A w \overset{*}{\Rightarrow} \alpha\beta w$
  Here $A \rightarrow \beta$ is a handle.



Bottom up parsing

- Example derivation:

  `Grammar:`

  $$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \mathbf{id} \end{aligned}$$

  `Input:` $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$

  `Two possible derivations:`

  E

  $$\begin{aligned} &\overset{rm}{\Longrightarrow} && \underline{E + E} \\ &\overset{rm}{\Longrightarrow} && E + \underline{E * E} \\ &\overset{rm}{\Longrightarrow} && E + E * \underline{\mathbf{id}_3} \\ &\overset{rm}{\Longrightarrow} && E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\overset{rm}{\Longrightarrow} && \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

  $$\begin{aligned} &\overset{rm}{\Longrightarrow} && \underline{E * E} \\ &\overset{rm}{\Longrightarrow} && E * \underline{\mathbf{id}_3} \\ &\overset{rm}{\Longrightarrow} && \underline{E + E} * \mathbf{id}_3 \\ &\overset{rm}{\Longrightarrow} && E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\ &\overset{rm}{\Longrightarrow} && \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3 \end{aligned}$$

- Primary goal of parser: how to select the right handle at each point...
- So primary issues:
  - What is starting state of a bottom up parser?
  - Given a stack state and an input symbol, should it shift? What is the basis for decision that it should shift?
  - If reduce, what should trigger reduction?

Bottom up parsing

- Stack: contains configuration of the form $(s_0 X_1 s_1 X_2 \cdots X_m s_m)$
  X_i: grammar symbol, s_i: state
- LR parser configuration: (stack, input)
  $(s_0 X_1 s_1 X_2 \cdots X_m s_m, a_i a_{i+1} \cdots a_n)$
- Parsing tables:
  - action: actions to take for a state/input pair.
  - goto: defines state transitions on reduces

  Focus: how to construct this parsing table?
- How does parser behave?
  1. $action[s_m, a_i] = shift\ \ s \implies$ new configuration:
     $(s_0 X_1 s_1 X_2 \cdots X_m s_m a_i s, a_{i+1} \cdots a_n)$
  2. $action[s_m, a_i] = reduce\ A \rightarrow \beta \implies$ new configuration:
     $(s_0 X_1 s_1 X_2 \cdots X_{m-r} s_{m-r} A s, a_{i+1} \cdots a_n)$ where $r = |\beta|$, and $goto[s_{m-r}] = s$.
     - Pop off $2r$ symbols from stack ($r = |\beta|$)
     - Assume top of stack = s; Push A now
     - Push goto(s, A)
  3. $action[s_m, a_i] = accept \implies$ done.
  4. $action[s_m, a_i] = error \implies$ error.

Example

- Grammar:
  1. $S' \rightarrow S$
  2. $S \rightarrow (S)S$
  3. $S \rightarrow \epsilon$
- Action table:

State	(	)	$	Goto(S)
0	s2	r3	r3	1
1		s3	accept	
2	s2	r3	r3	3
3		s4		
4	s2	r3	r3	5
5		r2	r2	

- Example: input string ()()

	Stack	Input	Action
1	$ 0	()()$	shift 2
2	$ 0(2	)()$	reduce 3
3	$ 0(2 S 3	)()$	shift 4
4	$ 0(2 S 3) 4	) $	shift 2
5	$ 0(2 S 3) 4 (2	)$	reduce 3
6	$ 0(2 S 3) 4 (2 S 3	)$	shift 4
7	$ 0(2 S 3) 4 (2 S 3) 4	$	reduce 3
8	$ 0(2 S 3) 4 (2 S 3) 4 S 5	$	reduce 2
9	$ 0(2 S 3) 4 S 5	$	reduce 2
10	$ 0 S 1	$	accept

Bottom up parsing

- So main concern: how to construct table?
  1. Construct set of states
  2. Construct action table
  3. Construct goto table
- What is a state?
  A state is of the form $[A \rightarrow \alpha.B\beta, a]$.
  1. Expect to reduce by $A \rightarrow \alpha B\beta$.
  2. Have already see α and is on stack.
  3. Expects to see symbols that will be generated by B.
  4. If β is ϵ,
     - (In LALR and LR) If next input symbol is a, reduce by $A \rightarrow \alpha B\beta$.
     - (In SLR), if next input symbol in Follow(A), then reduce.
- Construction of set of states:
  1. Start with an initial state/configuration/item and take closure to construct first state.
  2. Determine what next state/configuration will be.
     Take its closure to determine the next state.
  3. Iterate previous step until no more transitions are possible.
- Construct Goto table: Define transition from one state to another state over a nonterminal.
- Construct Action table: When to shift and reduce on a terminal symbol.

Bottom Up Parsing - SLR(1)

- Parser table:
  - i) Augment grammar by adding a new starting symbol, S' and production $S' \rightarrow S$ where S is original starting symbol.
  - ii) Construct LR(0) items (states of parser)
  - iii) Find transitions from one item to another on both terminals and non-terminals
  - iv) Find follow for NTs.
  - v) Use LR(0) items and follow to construct table.
- What are LR(0) items and how to construct them?
  - An item $A \rightarrow \alpha.B\beta$ means:
    α has already been seen, and expect to see $B\beta$. The "." determines how much has been seen.
    Note: stack will hold α.
  - Construct LR(0) items through following steps:
    - Set initial item $= [S' \rightarrow .S]$
    - Use *closure* to find all items: closure(I) is set of closure of items in set I in grammar G:
      if $[A \rightarrow \alpha.B\beta]$ is in closure(I), add all items $[B \rightarrow .\gamma]$ if there is a rule $B \rightarrow \gamma$ and items do not exist already..
      Apply until no more items can be added.

Closure - example

- Example grammar:
  $E' \rightarrow E \quad E \rightarrow E + T \quad E \rightarrow T$
  $T \rightarrow T * F \quad T \rightarrow F \quad F \rightarrow (E) | \mathbf{id}$
- Initial item $I = \{[E' \rightarrow .E]\}$
- Closure(I) =
  1. From productions of E: $[E \rightarrow .E + T]$, $[E \rightarrow .T]$
  2. Now, need to include items from T: $[T \rightarrow .T * F]$, $[T \rightarrow .F]$
  3. Similarly add items from F: $[F \rightarrow .(E)]$, $[F \rightarrow .\mathbf{id}\ ]$
  4. Closure(I) = $\{ [E' \rightarrow .E], [E \rightarrow .E + T], [E \rightarrow .T], [F \rightarrow .(E)], [F \rightarrow .\mathbf{id}\ ] \}$
- All of these form one single state. Why?
  Starting with $[E' \rightarrow .E]$, can move to any of the above items without taking any input, that is ϵ transition.
- Closure of an item constructs one state. However, we want to find all states $\Longrightarrow$ find possible transitions

How to evaluate possible transitions?

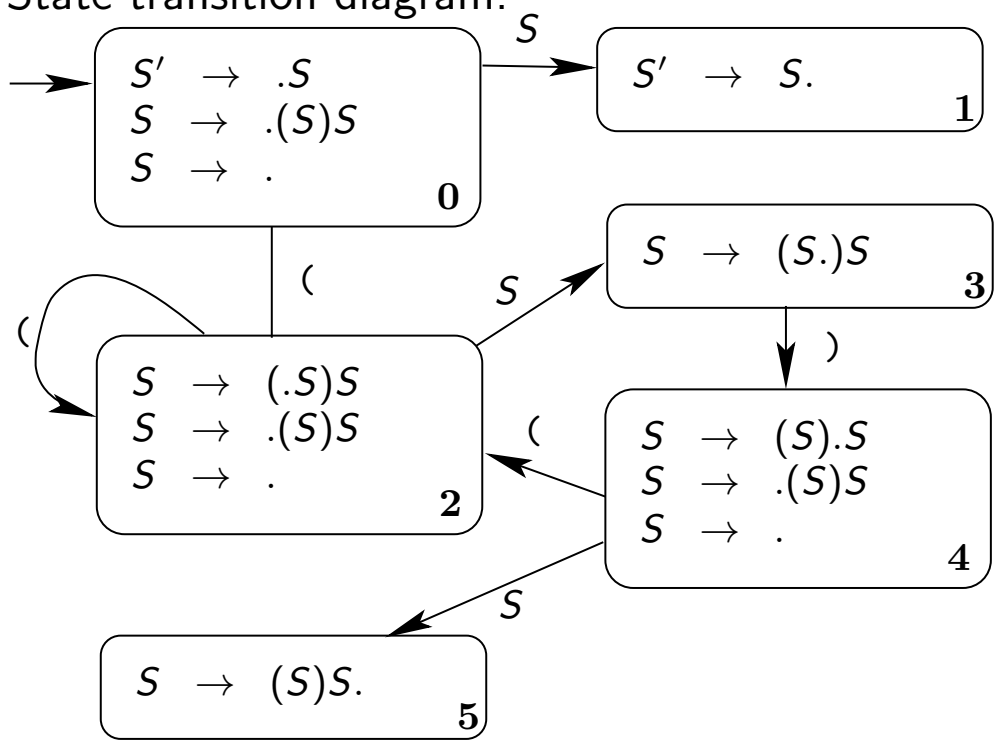
- Goto(I, X) specifies a state to which I moves on input X.
- How to evaluate it?
  1. For each item of form $[A \rightarrow \alpha.X\beta]$, add item $[A \rightarrow \alpha X.\beta]$.
  2. Add closure of $[A \rightarrow \alpha X.\beta]$ as well.
- Transition on a nonterminal:
  - Example: Goto(I_0, E): I_1:
    $[E' \rightarrow E.]$ $[E \rightarrow E. + T]$
    Above defines a state after a set of input symbols and nonterminals (that is a right side handle) has been recognized by reducing them to E. Hence, a transition from I_0 to this new state.
  - Why both $[E' \rightarrow E.]$ and $[E \rightarrow E. + T]$?
    The reason is that one can go from $[E' \rightarrow .E]$ to $[E' \rightarrow E.]$ on E as well from from $[E \rightarrow .E + T]$ to $[E \rightarrow E. + T]$.
- Transition on input symbol: Move from one state to another after seeing an input symbol.
  Transition from I_1 on '+' to state I_2:
  $[E \rightarrow E + .T]$ $[T \rightarrow .T * F]$ $[T \rightarrow .F]$
  $[F \rightarrow .(E)]$ $[F \rightarrow .\mathbf{id}\ ]$
  State signifies that '+' has been seen (and is on the stack), and expect to see symbols derivable from T or F.

Example

- Grammar:
  1. $S' \rightarrow S$
  2. $S \rightarrow (S)S$
  3. $S \rightarrow \epsilon$
- Items:

$S' \rightarrow .S$	$S' \rightarrow S.$
$S \rightarrow .(S)S$	$S \rightarrow (.S)S$
$S \rightarrow (S.)S$	$S \rightarrow (S).S$
$S \rightarrow (S)S.$	$S \rightarrow .$

- State transition diagram:



Example

```
0. E' -> E
1. E -> E | T
2. E -> T
3. T -> TF
4. T -> F
5. F -> (E)
6. F -> id
7. F -> F*
s0:
        [E' -> .E]
        [E -> .E | T]
        [E -> .T]
        [T -> .TF]
        [T -> .F]
        [F -> .(E)]
        [F -> .id]
        [F -> .F*]
     ** transitions from s0
s1:
        [E' -> E.]                      goto(s0, E) = s1
        [E- -> E.|T]
s2:
        [E->T.]                         goto(s0, T) = s2
        [T -> T.F]
        [F -> .(E)]
        [F -> .id]
        [F -> .F*]
s3:
        [T -> F.]                       goto(s0, F) = s3
        [F -> F.*]
s4:
        [F -> (.E)]                     goto(s0, () = s4
        [E -> .E | T]
        [E -> .T]
        [T -> .TF]
        [T -> .F]
        [F -> .(E)]
        [F -> .id]
        [F -> .F*]
s5:
        [F -> id.]                      goto(s0, id) = s5
        ** transitions from s1:
s6:
        [E -> E | . T]                  goto(s1, |) = s6
        [T -> .TF]
        [T -> .F]
        [F -> .(E)]
        [F -> .id]
        [F -> .F*]
        ** transitions from s2:
s7:
        [T -> TF.]
        [F -> F.*]                      goto(s2, F) = s7
                                        goto(s2, () = s4
                                        goto(s2, id) = s5
        **transitions from s3:
s8:
        [F -> F*.]                      goto(s3, *) = s8
        ** transitions from s4:
s9:
        [F -> (E.)]                     goto(s4, E) = s9
        [E -> E.| T ]                    goto(s4, T) = s2
                                        goto(s4, F) = s3
                                        goto(s4, () = s4
                                        goto(s4, id) = s5
        ** transitions from s5:
        ** transitions from s6
s10:
        [E -> E | T.]                   goto(s6, T) = s10
        [T -> T.F]
        [F -> .(E)]
        [F -> .id]
        [F -> .F*]
                                        goto(s6, F) = s3
                                        goto(s6, () = s4
                                        goto(s6, id) = s5

        ** transitions from s7
                                        goto(s7, *) = s8
        ** transitions from s8
        ** transitions from s9
s11:
        [F -> (E).]                     goto(s9, )) = s11
                                        goto(s9, |) = s6
        ** transitions from s10
                                        goto(s10, F) = s7
                                        goto(S10, () = s4
                                        goto(s10, id) = s5
        ** transitions from s11
```

How to construct table

- Goto Table: specifies transition between states over Nonterminals.
  Example: Transition from I_0 on I_1 over E. What does this transition say?
  It says that we have seen all that can be derived from E.
- Action table: two components:
  1. Shifts: Shifts over terminals specify transition from one state to another.
     Example: On seeing '+', we can move from state I_1 to I_2. Also, we shift '+' on stack.
  2. Reduces: For each state, see if it possible to reduce: that is, dot is at the end of some right hand side. It means that right hand side for that production is on the stack.
     Need to decide if we can reduce using that production.
     How to decide? SLR(1) decides on basis of FOLLOW Set.
     Example: If in state I_1, we have an item $[E' \rightarrow E.]$. What does it mean to be in state I_1? It means that we have already reduced by E. So it is possible to reduce right hand side by $E' \rightarrow E$. If input symbol is in Follow(E'), the above reduction takes place.
- The parser driver basically works from this table shifting symbols and moving from state to state, and reducing.

Table construction

- Table for example grammar of slide 2
- Find follow:

  Follow(S') = {$}

  Follow(S) = {$,)}
- SLR(1) table:

State	Input			GoTo
	(	)	$	S
0	s2	r3	r3	1
1		s3	accept	
2	s2	r3	r3	3
3		s4		
4	s2	r3	r3	5
5		r2	r2	

- Steps by algorithm for input () ():

	Parsing Stack	Input	Action
1	$0	()()$	shift 2
2	$0(2	)()$	reduce 3
3	$0(2S3	)()$	shift 4
4	$0(2S3)4	()$	shift 2
5	$0(2S3)4(2	)$	reduce 3
6	$0(2S3)4(2S3	)$	shift 4
7	$0(2S3)4(2S3)4	$	reduce 3
8	$0(2S3)4(2S3)4S5	$	reduce 2
9	$0(2S3)4S5	$	reduce 2
10	$0 S 1	$	accept

Table construction - cont'd.

- Table for example grammar of slide 4, 8
- Find follow:

```
Follow(E') = {$}
Follow(E) = {|, )} U {$} = {|, ), $}
Follow(T) = Follow(E) U First(F)
        First(F) = {(, id}
     => Follow(T) = {|, ), (, id, $}
Follow(F) = {*} U Follow(T) = {*, |, ), (, id, $}
```

- SLR(1) table:

	|	id	*	(	)	$	E	T	F
0		s5		s4			1	2	3
1	s6					r1			
2	r2	s5		s4	r2	r2			s7
3	r4	r4	s8	r4	r4	r4			
4		s5		s4			s9	s2	s3
5	r6	r6	r6	r6	r6	r6			
6		s5		s4				s10	s3
7	r3	r3	s8	r3	r3	r3			
8	r7	r7	r7	r7	r7	r7			
9					s11				
10	r1	s5		s4	r1	r1			s7
11	r5	r5	r5	r5	r5	r5			

SLR parsing

- A grammar is SLR(1) iff, for any state s,
  - for any item $[a \rightarrow \alpha.X\beta]$, there is no complete item $[B \rightarrow \gamma.]$ in s with X in Follow(B) (no shift-reduce conflict), and
  - for any two complete items $[A \rightarrow \alpha.]$ and $[B \rightarrow \beta.]$ in s, sets Follow(A) and Follow(B) should not have any common elements.
- Example of a language where SLR(1) parsers are not enough:
- Grammar:
  $S \rightarrow \textbf{id} \mid V := E$
  $V \rightarrow \textbf{id}$
  $E \rightarrow V \mid n$
- Consider initial parser state:
  $[S' \rightarrow .S]$
  $[S \rightarrow .\textbf{id}\ ]$
  $[S \rightarrow .V := E]$
  $[V \rightarrow .\textbf{id}\ ]$

  Transition on **id** to state.
  $[S \rightarrow \textbf{id}\ .]$
  $[V \rightarrow \textbf{id}\ .]$

- Follow(S) = {\$}, Follow(V) = {:=, \$}
  We can therefore reduce by $S \rightarrow \textbf{id}$ and $V \rightarrow \textbf{id}$ under input symbol \$.

Disambiguating Rules for Parsing Conflicts

- Consider a simple version of if-then-else:

  0. $S' \rightarrow S$
  1. $S \rightarrow I$
  2. $S \rightarrow other$
  3. $I \rightarrow if\ S$
  4. $I \rightarrow if\ S\ else\ S$

- Parsing table:

State	Input				GoTo	
	if	else	other	$	S	I
0	s4		s3		1	2
1				accept		
2		r1		r1		
3		r2		r2		
4	s4		s3		5	2
5		s6/r3		r3		
6	s4		s3		7	2
7		r4		r4		

- State 5:
  $[I \rightarrow if\ S.]$
  $[I \rightarrow if\ S.elseS]$
- Choose s6 to denote that else matches with the closest if.

LR(1) and LALR(1) Parsers

- Most general form of bottom up: LR parsing, also called *canonical LR(1)* parsing
- Problem with SLR(1): uses follow as a lookahead. LR(1) solves problem by constructing DFA that keeps track of i) what has been seen (states), and ii) and what we expect to see (lookahead).
- LR(1) item: $[A \rightarrow \alpha.\beta, a]$. a is a look ahead token of size 1.
- What does additional information give us?
  For items $[A \rightarrow \alpha., a]$, reduce by $A \rightarrow \alpha$ only if next input symbol is a.
  Set of all such a's is subset of Follow(A).
- Method for constructing collections of LR(1) items: extend SLR(1) method so that additional information is computed.
  - Closure of $[A \rightarrow \alpha.B\beta, a]$: for every production $B \rightarrow \gamma$ and each terminal b in First(βa), add $[B \rightarrow .\gamma, b]$ if not there.
  - goto(I, X), where I is a set of items: For $[A \rightarrow \alpha.X\beta, a]$, add all items J of form $[A \rightarrow \alpha X.\beta, a]$ and their closures.
  - Find all items:
    - C := {closure of $[S' \rightarrow S, \$]$}
    - For each I in C, and grammar symbol X, add goto(I, X) to C.

Example

- Grammar:

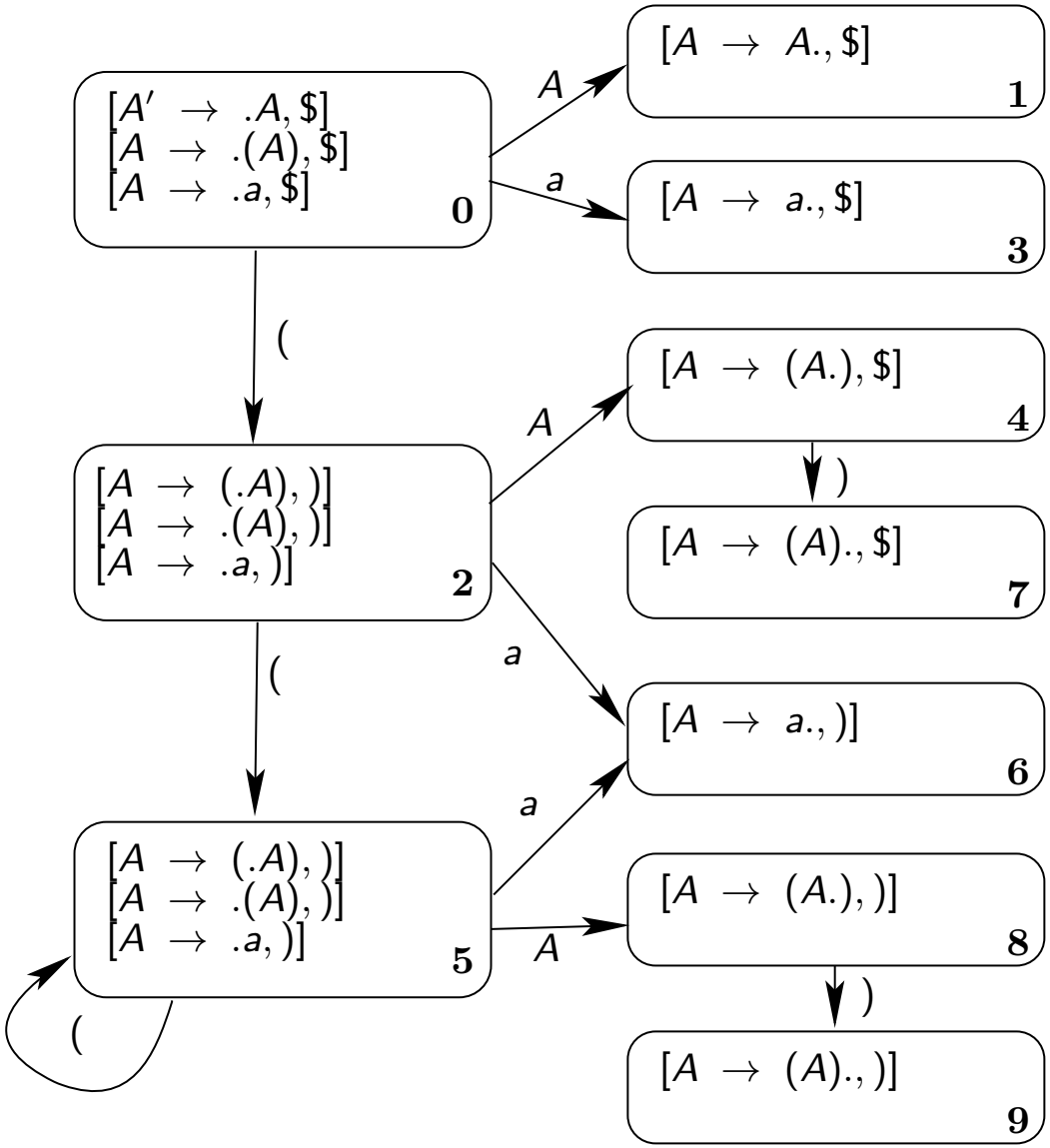
  $A \rightarrow (A) \mid a$

- Construct set of items:

```
State 0:    [A' → .A, $]
            [A → .(A), $]
            [A → .a, $]
                Transition (state 0, A)
State 1:    [A' → A., $]
                Transition (state 0, (
State 2:    [A → (.A), $]
            [A → .(A), )]
            [A → .a, )]
                Transition (state 0, a)
State 3:    [A → a., $]
                Transition (state 2, A)
State 4:    [A → (A.), $]
                Transition (state 2, (
State 5:    [A → (.A), )]
            [A → .(A), )]
            [A → .a, )]
                Transition (state 2, a)
State 6:    [A → a., )]
                Transition (state 4, ))
State 7:    [A → (A)., $]
                Transition (state 5, (
State 8:    [A → (A.), )]
                Transition (state 8, ))
State 9:    [A → (A)., )]
```

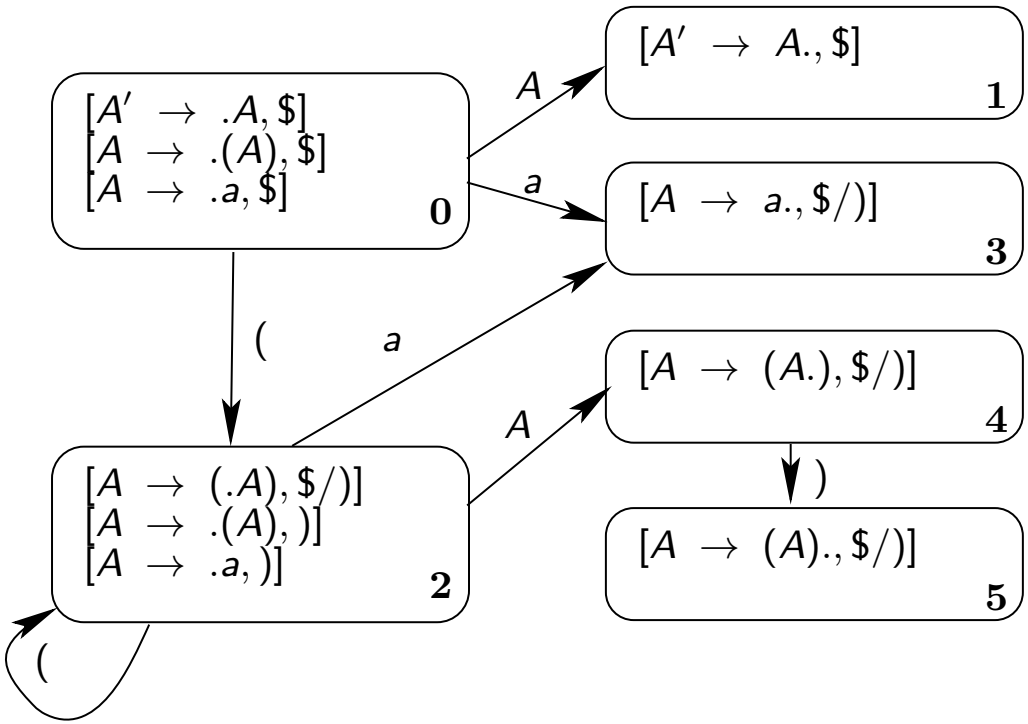
State diagram

- LR(1) state diagram:



LALR(1) parsing tables

- Table obtained is considerably smaller than LR(1).
- Note: many states in LR(1) table have identical first component, and different lookahead component.
  LALR: combine these states.
- Example:



- Note: Such a composition of states may lead to reduce/reduce conflicts.

Summary of Results

- Every SLR(1) grammar is unambiguous. However, not all unambiguous grammar is SLR(1).

$$\begin{aligned} S &\rightarrow L = R | R \\ L &\rightarrow *R | \mathbf{id} \\ R &\rightarrow L \end{aligned}$$

- Every grammar that has an SLR(1) parser is an LR(1) grammar but not vice-versa:

$$\begin{aligned} Z &\rightarrow S \\ S &\rightarrow E = E | \mathbf{id} \\ E &\rightarrow E + \mathbf{id} \ \ | \mathbf{id} \end{aligned}$$

  SLR(1) table has reduce-reduce conflicts.
- Every grammar that has an LALR(1) parser is an LR(1) grammar but not vice versa.

$$\begin{aligned} S &\rightarrow aAd | bBd | aBe | bAe \\ A &\rightarrow c \\ B &\rightarrow c \end{aligned}$$

  LALR(1) table has reduce-reduce conflicts. LR(1) does not.
- There are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action conflicts.

$$\begin{aligned} S &\rightarrow aAc \\ A &\rightarrow bAb | b \end{aligned}$$

- LR(0) $\subset$ LR(1) $\subset$ LR(2) $\cdots$ $\subset$ LR(k) $\cdots$