BOTTOM-UP PARSING

This is a parsing strategy based on the reverse process to top-down parsing. Instead of expanding successful non-terminals according to production rules, a current string or right sentential form is collapsed each time until the start non-terminal is reached to predict the legal next symbol; i.e. it can be regarded as a series of reductions. This approach is also known as **shift-reduce parsing** and is the primary parsing method for many compilers, mainly due to its speed and the tools which automatically generate a parser based on the grammar.

Example

Consider the grammar below:

1 $S \rightarrow aABe$
2 $A \rightarrow Abc|b$
3 $B \rightarrow d$

To parse the sentence $abbcde$ using the bottom up approach gives the following reductions:

$a\mathbf{b}bcde$

$a\mathbf{Abc}de$ by 2

$aA\mathbf{d}e$ by 2

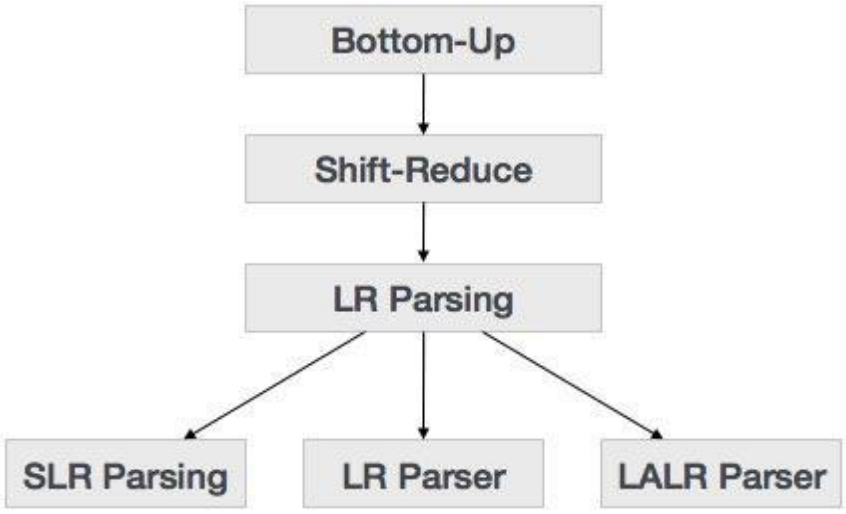
$\mathbf{aABe}$ by 3

S by 1

Reverse gives in right most derivation:

$$S \rightarrow aABe \rightarrow aAde \rightarrow aAbcde \rightarrow abbcde$$

Generally, bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The figure below depicts the bottom-up parsers available.



LR Parsing

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. It is one of the best methods of syntactic recognition of programming languages. The L stands for left to right scan, and R stands for right most derivation **in reverse**. In general, we can have $LR(k)$ parsing with k symbols of LOOKAHEAD. However, LR parsing refers to $LR(1)$ parsing.

Advantages of LR Parsing

i) LR parsers can recognize virtually all programming language constructs written with CFG grammars.
ii) It is the most general, non-backtracking technique known.
iii) It can be implemented in a very efficient manner.
iv) The language it can recognize is a proper super set of that of predictive parsers.
v) It can recognize syntax errors quickly.

Disadvantages

The primary drawback of LR parsers is that they require too much work to manually create LR parsing tables. However, tools exist to generate LR parsers from a given grammar i.e. parser generators such as YACC, BISON etc.

LR Parsing Methods

There are three widely used algorithms available for constructing an LR parser:

i) **SLR:** this stands for Simple LR. It is easy to implement but less powerful than other parsing methods. It generally works on smallest class of grammar and have few number of states, hence very small table

ii) **Canonical LR:** this is the most general and powerful. However, it is tedious and costly to implement, i.e. for the same grammar, it has got much number of states as compared to SLR parsers. Generally, it works on complete set of LR(1) Grammar and generates large table and large number of states

iii) **LALR:** this stands for LOOKAHEAD LR. It is a mixture of SLR and canonical LR, but it can be implemented efficiently i.e. it contains the same number of states as Simple LR parser for the same grammar.
Notice that most parser generators generate LALR parsers since they are a trade-off between power and efficiency.

Implementing Shift Reduction

A shift-reduce parser is implemented using the following notation:

An input stream, containing
a phrase to be parsed and
a stack holding a symbols

The input stream holds terminals, the stack can hold a mixture of terminals and non-terminals, the latter generated by earlier reductions.
The operation **shift** moves a symbol from the input to the stack while the operation **reduce** combines the sequence ending with the last terminal shifted to form a non-terminal on the stack.
When the input is exhausted, the single start symbol should be presented assuming all reductions have been performed.

Example

Consider the following grammar:

$$exp \rightarrow exp + exp$$
$$exp \rightarrow exp * exp$$

$exp \rightarrow (exp)$
$exp \rightarrow id$

Show the sequence of shift/reduce for the string $\mathrm{id}_1 + \mathrm{id}_2 * \mathrm{id}_3$.

Stack	Input	Action
$	id1+id2*id3$	Shift
$id1	+id2*id3$	Reduce using exp->id
$exp	+id2*id3$	shift
$exp+	id2*id3$	shift
$exp+id2	*id3$	Reduce using exp->id
$exp+exp	*id3$	shift
$exp+exp*	id3$	shift
$exp+exp*id3	$	Reduce using exp->id
$exp+exp*exp	$	Reduce using exp->exp*exp
$exp+exp	$	Reduce using exp->exp+exp
$exp	$	Accept

Comparison between LL and LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Handles

A *handle* of a right-sentential form γ is a production $A \rightarrow \beta$ and a position in γ where β may be found and replaced by A to produce the previous right sentential form in a rightmost derivation of γ i.e.,
if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha \beta w$ then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$
Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols i.e. a handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot). Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we refer to the body β rather than $A \rightarrow \beta$ as a handle.

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning* that is start with a string of terminals w to be parsed. If w is a sentence of the grammar at hand, then let $w = \gamma_n$, where γ_n is the n^{th} right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n=w}$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \rightarrow \beta_n$ to obtain the previous right-sentential form γ_{n-1}.
Then repeat this process. That is; locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing.

Algorithm

Set *i* to *n* and apply the following simple algorithm

For *i* = *n* down to 1

1. Find the handle $Ai \rightarrow \beta i$ in yi
2. Replace β_i with A_i to generate y_{i-1}

This takes 2n steps, where n is the length of the derivation

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is to use a *shift-reduce* parser.
Shift-reduce parsers use a *stack* and an *input buffer* as shown below

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is $
   i). find the handle
   if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
   ii). prune the handle
   if we have a handle $A \rightarrow \beta$ on the stack, *reduce*
   i). pop $|\beta|$ symbols off the stack
   ii). push *A* onto the stack

Generally, a shift-reduce parser have four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate nonterminal LHS
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

LR Parsers

As with LL(1), the aim of LR parsers is to make the choice of action depend only on the next input symbol and the symbol on top of the stack. To achieve this, a DFA needs to be constructed. The DFA is used to determine the next action and it only needs to look at the current state (stored at the top of the stack) and the next input symbol (shift action) or nonterminal (reduce action).
We represent the DFA as a table, where we cross-index a DFA state with a symbol (terminal or nonterminal).
Generally, you need to encode the DFA in a table i.e. a shift-reduce parser's DFA can be encoded in two tables

- One row for each state
- action table encodes what to do given the current state and the next input symbol
- goto table encodes the transitions to take after a reduction

Once the the DFA has been encoded as a table, you need to perform one of the following actions:

Actions (1)

Given the current state and input symbol, the main possible actions are

- s_i – shift the input symbol and state i onto the stack (i.e., shift and move to state i)
- r_j – reduce using grammar production j

- The production number tells us how many <symbol, state> pairs to pop off the stack

Actions (2)

Other possible action table entries

- accept
- blank – no transition, which means syntax error
  - A LR parser will detect an error as soon as possible on a left-to-right scan
  - real compiler needs to produce an error message, recover, and continue parsing when this happens

Goto

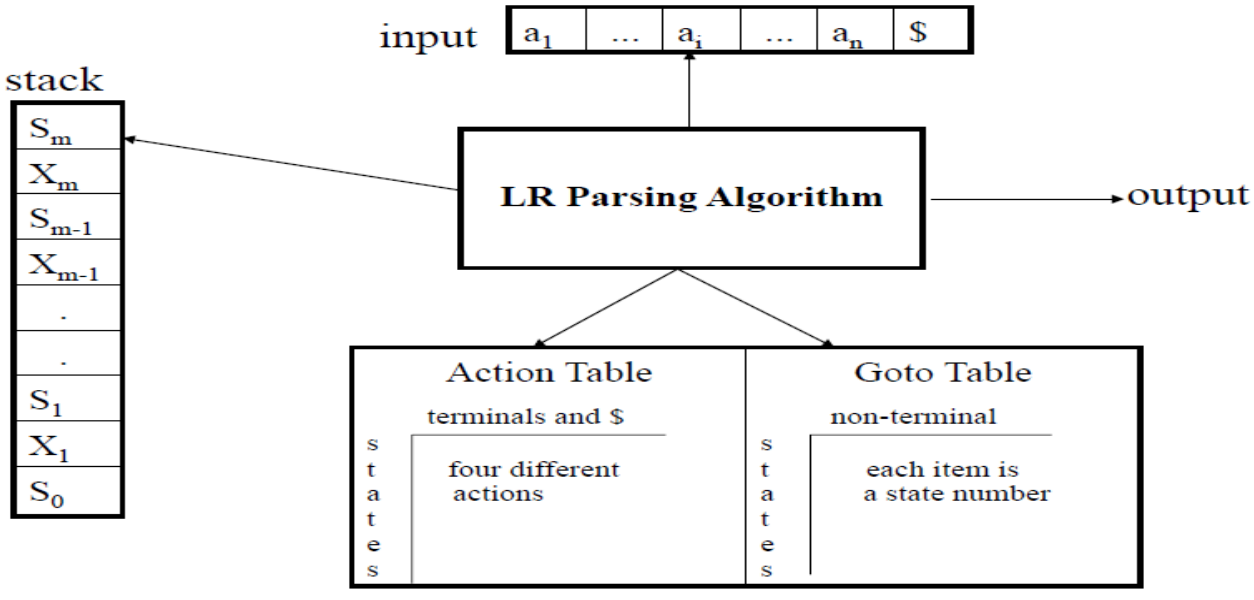
- When a reduction is performed, <symbol, state> pairs are popped from the stack revealing a state uncovered_s on the top of the stack
- goto[uncovered_s , A] is the new state to push on the stack when reducing production A ::= β (after popping β and revealing state uncovered_s on top)

Skeleton parser:

```
push s0
token = next_token()
repeat forever
   s = top of stack
   if action[s,token] == " shift si" then
      push si
      token = next_token()
   else if action[s,token] == " reduce A → β"
        then
      pop | β | states
      s' = top of stack
      push goto[s', A]
   else if action[s, token] == " accept" then
      return
   else error()
```

This takes *k* shifts, *l* reduces, and 1 accept, where *k* is the length of the input string and *l* is the length of the reverse rightmost derivation

Structure of a LR parser



Example
Consider the following grammar:

1. <goal> ::= <expr>
2. <expr> ::= <term> + <expr>
3. | <term>
4. <term> ::= <factor> * <term>
5. | <factor>
6. <factor> ::= id

The following is the corresponding parsing table

state	ACTION				GOTO		
	id	+	*	$	<expr>	<term>	<factor>
0	s4	–	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	s5	–	r3	–	–	–
3	–	r5	s6	r5	–	–	–
4	–	r6	r6	r6	–	–	–
5	s4	–	–	–	7	2	3
6	s4	–	–	–	–	8	3
7	–	–	–	r2	–	–	–
8	–	r4	–	r4	–	–	–

Parsing using the table for the input id*id+id

Stack	Input	Action
$ 0	id * id + id$	s4
$ 0 4	* id + id$	r6
$ 0 3	* id + id$	s6
$ 0 3 6	id + id$	s4
$ 0 3 6 4	+ id $	r6
$ 0 3 6 3	+ id $	r5
$ 0 3 6 8	+ id $	r4
$ 0 2	+ id $	s5
$ 0 2 5	id $	s4
$ 0 2 5 4	$	r6
$ 0 2 5 3	$	r5
$ 0 2 5 2	$	r3
$ 0 2 5 7	$	r2
$ 0 1	$	acc

LR(*k*) items
The table construction algorithms use sets of LR(*k*) *items* or *configurations* to represent the possible states in a parse. An LR(*k*) item is a pair [α, β], where

- α is a production from *G* with a • at some position in the RHS, marking how much of the RHS of a production has already been seen

– β is a lookahead string containing k symbols (terminals or \$)

Two cases of interest are $k = 0$ and $k = 1$:

- LR(0) items play a key role in the SLR(1) table construction algorithm. LR(0) items means *no lookahead* symbols
- LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms.

Note: LR(k) recognize occurrence of β (the handle) having seen all of what is derived from β plus k symbols of lookahead

Example

The • indicates how much of an item we have seen at a given state in the parse:
[$A \rightarrow \bullet XY\ Z$] indicates that the parser is looking for a string that can be derived from $XY\ Z$
[$A \rightarrow XY \bullet Z$] indicates that the parser has seen a string derived from XY and is looking for one derivable from Z
Generally, LR(0) item is a grammar rule with a dot on the right-hand side. For example, the rule $A \rightarrow XY\ Z$ generates four LR(0) items:

1. [$A \rightarrow \bullet XY\ Z$]
2. [$A \rightarrow X \bullet Y\ Z$]
3. [$A \rightarrow XY \bullet Z$]
4. [$A \rightarrow XY\ Z\bullet$]

The rule A → ε generates only A → • .

An LR(0) state is a set of (LR(0)) items. The items of a state indicate how much of the input has been recognized.

The characteristic finite state machine (CFSM)

The CFSM for a grammar is a DFA which recognizes *viable prefixes* of right-sentential forms. A *viable prefix* is any prefix that does not extend beyond the handle. It accepts when a handle has been discovered and needs to be reduced.

Key insight: recognize handles with a DFA:

- DFA transitions shift states instead of symbols
- Accepting states trigger reductions

To construct the CFSM you need two functions:

i). closure0 (I) to build its states

ii). goto0 (I,X) to determine its transitions

i). closure0

Given an item [$A \rightarrow \alpha \bullet B\beta$], its closure contains the item and any other items that can generate legal substrings to follow α. Thus, if the parser has viable prefix α on its stack, the input should reduce to $B\beta$ (or γ for some other item [$B \rightarrow \bullet\gamma$] in the closure).

```
function closure0(I)
repeat
     if [A → α • B β] ∈ I
          add [B → • γ] to I
until no more items can be added to I
return I
```

Generally, if I contains A → μ • B χ, then closure(I) adds to I all items B → • ν for each rule for B, continuing recursively, i.e. adds all rules that may be needed in recognizing B.

Example, for the expression grammar

1) E' → E
2) E → E + T
3) E → T
4) T → T * F
5) T → F
6) F → (E)
7) F → id

The closure of {E' → • E} is:

E' → • E E → • T T → • F F → • id
E → • E + T T → • T * F F → • (E)

ii). goto0

Let *I* be a set of LR(0) items and *X* be a grammar symbol, then, GOTO(*I,X*) is the closure of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta] \in I$

If *I* is the set of valid items for some viable prefix γ, then GOTO(*I,X*) is the set of valid items for the viable prefix γX.

GOTO(*I,X*) represents state after recognizing *X* in state *I*.

```
function goto0(I,X)
     let J be the set of items [A → α X • β]
          such that [A → α • X β] ∈ I
     return closure0(J)
```

The result of goto(I, X) contains for every item $A \rightarrow \mu \bullet X \chi$ in I the closure of $A \rightarrow \mu X \bullet \chi$, i.e. the state after X is recognized, for $X \in V$. For example, if I = {E' → • E, E → E • + T}, then goto(I, +) is:

E → E + • T T → • F F → • id
T → • T * F F → • (E)

Building the LR(0) item sets

You need to start the construction with the item [*S'* → •*S*$],
Where

- *S'* is the start symbol of the augmented grammar *G'*
- *S* is the start symbol of *G*
- $ represents EOF

We assume without loss of generality that each grammar contains a rule S' → S, where S' is the start symbol and S' does not occur anywhere else.
The following is the algorithm to compute for the collection of sets of LR(0) items

```
function items(G')
     s0 = closure0({[S' → • S$]})
     S = {s0}
     repeat
          for each set of items s ∈ S
               for each grammar symbol X
                    if goto0(s,X) ≠ ∅ and goto0(s,X) ∉ S
                         add goto0(s,X) to S
```

```
            until no more item sets can be added to S
    return S
```

Example of LR(0)

Consider the following grammar

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

The following are the LR(0) itemsets

I_0 :
$S \rightarrow \bullet E\$$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet$ id
$T \rightarrow \bullet(E)$

I_1 :
$S \rightarrow E\bullet\$$
$E \rightarrow E \bullet + T$

I_2 : $S \rightarrow E\$\bullet$

I_3 :
$E \rightarrow E + \bullet T$
$T \rightarrow \bullet id$
$T \rightarrow \bullet(E)$

I_4 : $E \rightarrow E + T\bullet$

I_5 : $T \rightarrow id \bullet$

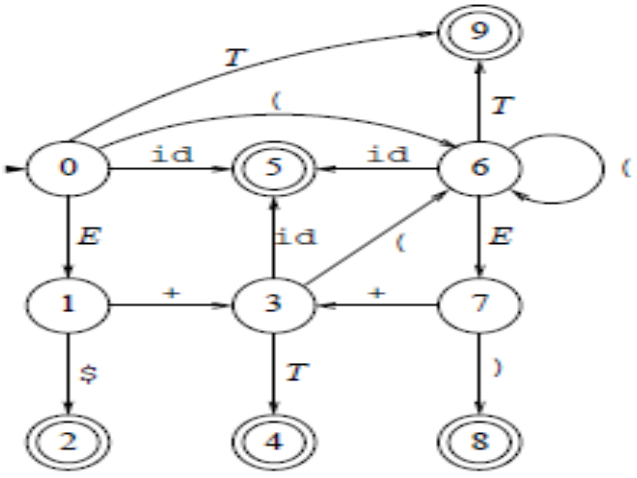
I_6 :
$T \rightarrow (\bullet E)$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet id$
$T \rightarrow \bullet(E)$

I_7 :
$T \rightarrow (E\bullet)$
$E \rightarrow E \bullet + T$

I_8 : $T \rightarrow (E)\bullet$

I_9 : $E \rightarrow T\bullet$

The following is the corresponding (characteristic finite state machine) CFSM:



Constructing the LR(0) parsing table

1. construct the collection of sets of LR(0) items for S'
2. state i of the CFSM is constructed from I_i

a) $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and goto0(Ii, a) = I_j
$\Rightarrow$ ACTION[i, a] = "*shift* j"
b) $[A \rightarrow \alpha\bullet] \in I_i, A \neq S'$
$\Rightarrow$ ACTION[i, a] = "*reduce* $A \rightarrow \alpha$", $\forall a$
c) $[S' \rightarrow S\$\bullet] \in I_i$
$\Rightarrow$ ACTION[i, a] = "*accept*", $\forall$

3. goto0(I_i,A) = I_j
$\Rightarrow$ GOTO[i,A] = j
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is closure0([$S' \rightarrow \bullet S\$$])

The following is the LR(0) parsing table

state	ACTION					GOTO		
	id	(	)	+	$	S	E	T
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	s2	–	–	–
2	acc	acc	acc	acc	acc	–	–	–
3	s5	s6	–	–	–	–	–	4
4	r2	r2	r2	r2	r2	–	–	–
5	r4	r4	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	r5	r5	r5	r5	r5	–	–	–
9	r3	r3	r3	r3	r3	–	–	–

Conflicts in the ACTION table
If the LR(0) parsing table contains any multiply defined ACTION entries then *G* is not LR(0)
Two conflicts arise:

shift-reduce : both shift and reduce possible in same item set
reduce-reduce : more than one distinct reduce action possible in same item set

LR(0) has a reduce/reduce conflict if:

- Any state has two reduce items:
- X → β. and Y → ω.

LR(0) has a shift/reduce conflict if:

- Any state has a reduce item and a shift item:
- X → β. and Y → ω.tδ

Conflicts can be resolved through *lookahead* in ACTION. Consider:

- $A \rightarrow \varepsilon | a\alpha$
$\Rightarrow$ shift-reduce conflict
- a:=b+c*d
requires lookahead to avoid shift-reduce conflict after shifting *c* (need to see * to give precedence over +)

Generally, if the grammar is not LR(0), you need to resolve conflicts in the states using one look-ahead symbol

A simple approach to adding lookahead: SLR(1)
Add lookaheads after building LR(0) item sets

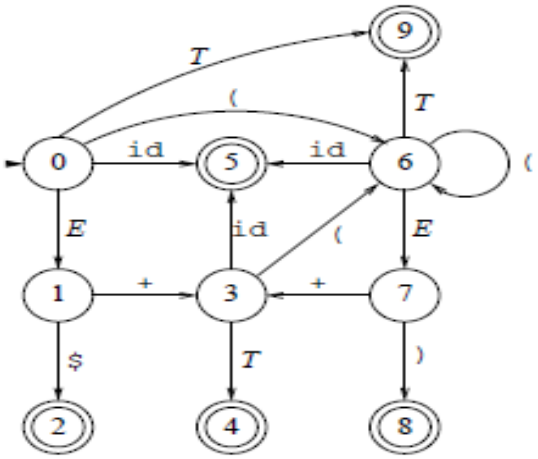
Constructing the SLR(1) parsing table:

1. construct the collection of sets of LR(0) items for *G'*
2. state *i* of the CFSM is constructed from I_i
   a) $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and goto0(I_i, *a*) = I_j
      $\Rightarrow$ ACTION[*i*, *a*] = "*shift j*", $\underline{\forall a \neq \$}$
   b) $[A \rightarrow \alpha\bullet] \in Ii, A \neq S'$
      $\Rightarrow$ ACTION[*i*, *a*] = "*reduce* $A \rightarrow \alpha$",
      $\underline{\forall a \in \text{FOLLOW}(A)}$
   c) $\underline{[S' \rightarrow S \bullet \$]} \in I_i$
      $\Rightarrow$ ACTION[*i*, $] = "*accept*"
3. goto0(I_i,*A*) = I_j
   $\Rightarrow$ GOTO[*i*,*A*] = *j*
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is closure0([$S' \rightarrow \bullet S\$$])

From the previous example
Consider the following grammar

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

The following is the corresponding CSFM



FOLLOW(*E*) = FOLLOW(*T*) = {$,+,)}

state	ACTION					GOTO		
	id	(	)	+	$	S	E	T
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	acc	–	–	–
2	–	–	–	–	–	–	–	–
3	s5	s6	–	–	–	–	–	4
4	–	–	r2	r2	r2	–	–	–
5	–	–	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	–	–	r5	r5	r5	–	–	–
9	–	–	r3	r3	r3	–	–	–

SLR(1) grammars

- A grammar for which there is no (shift/reduce or reduce/reduce) conflict during the construction of the SLR table is called SLR(1) (or SLR in short).
- All SLR grammars are unambiguous but many unambiguous grammars are not SLR
- There are more SLR grammars than LL(1) grammars but there are LL(1) grammars that are not SLR.