UNIT-3

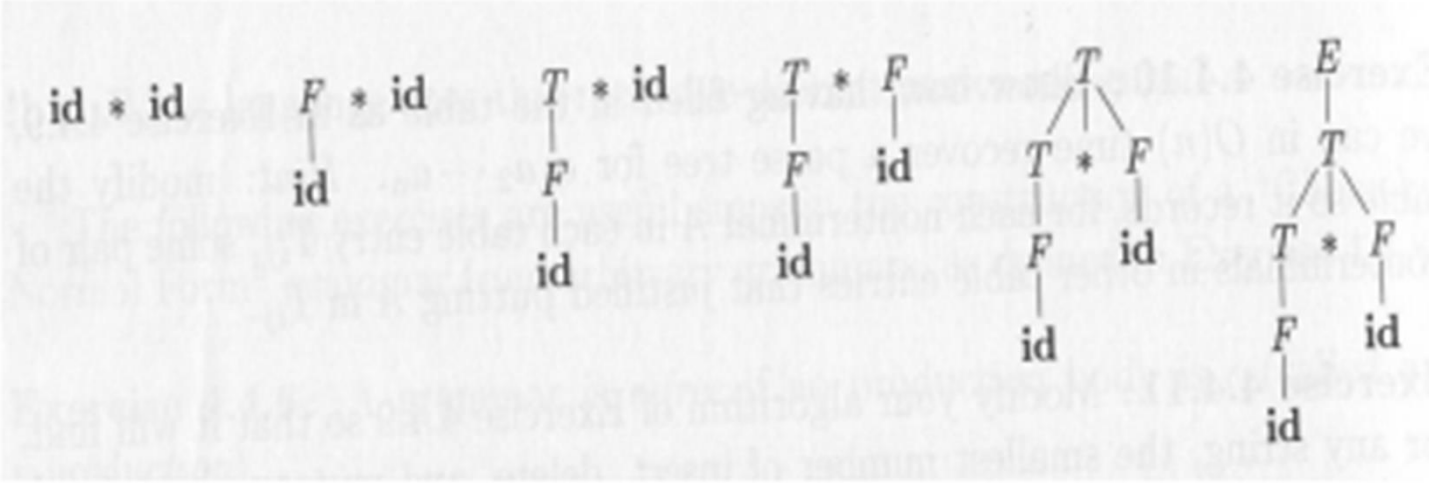
- **Bottom up Parsing:** Reductions – Handle Pruning - Shift Reduce Parsing – Conflicts During Shift–Reduce Parsing.
- **Introduction to simple LR Parsing:**
- Why LR Parsers – Items and LR(0) Automaton - The LR-Parsing Algorithm - Constructing SLR–Parsing Tables

Bottom up Parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root(the top).

Example

- Given the grammar:
  - $E \rightarrow T$
  - $T \rightarrow T * F$
  - $T \rightarrow F$
  - $F \rightarrow id$



Reductions

- **bottom-up parsing** as the process of "reducing" a string w to the **start symbol** of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the non terminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply.

- A reduction is the reverse of a step in a derivation
- The goal of bottom-up parsing is therefore to construct a derivation in reverse
- $E \Rightarrow T \Rightarrow T*F \Rightarrow T*id \Rightarrow F*id \Rightarrow id*id$.
- This derivation is in fact a **rightmost derivation**.

Handle Pruning

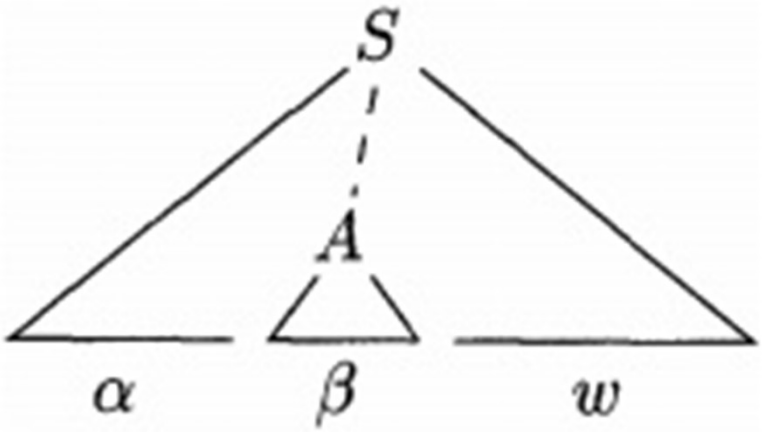
- Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse.
- Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

- The leftmost substring that matches the body of some production need not be a handle.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

- A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals *w* to be parsed.



Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input.
- Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.

- Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	w $

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string */3* of grammar symbols on top of the stack.

- It then reduces */3* to the head of the appropriate production.

- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty

STACK	INPUT
$ S	$

- Upon entering this configuration, the parser halts and announces successful completion of parsing.
- Steps through the actions a shift-reduce parser might take in parsing the input string id∗id according to the expression grammar

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.
- *Shift*. Shift the next input symbol onto the top of the stack.
- *Reduce*. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non terminal to replace the string.
- *Accept*. Announce successful completion of parsing.
- *Error*. Discover a syntax error and call an error recovery routine.

- Consider the grammar

$$S \rightarrow S + S$$
$$S \rightarrow S * S$$
$$S \rightarrow id$$

- Perform Shift Reduce parsing for input string "id + id + id"

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce S->id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce S->id
$S+S	+id$	Reduce S->S+S
$S	+id$	Shift
$S+	id$	Shift
$S+id	$	Reduce S->id
$S+S	$	Reduce S->S+S
$S	$	Accept

- Consider the grammar

  S –> (L) | a

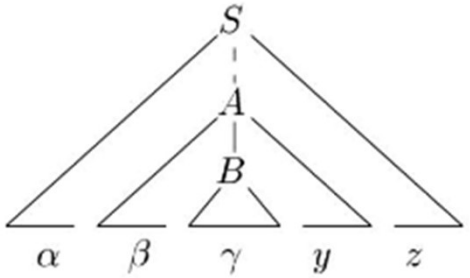
  L –> L , S | S

  Perform Shift Reduce parsing for input string “(a, (a, a)) “.

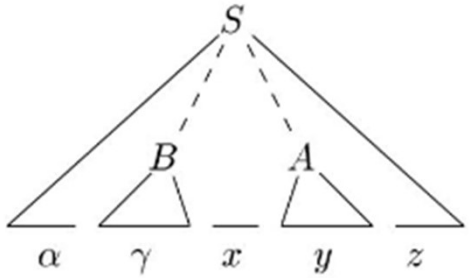
Stack	Input Buffer	Parsing Action
$	(a,(a,a))$	Shift
$(	a,(a,a))$	Shift
$(a	,(a,a))$	Reduce S → a
$(S	,(a,a))$	Reduce L → S
$(L	,(a,a))$	Shift
$(L,	(a,a))$	Shift
$(L,(	a,a))$	Shift
$(L,(a	,a))$	Reduce S → a
$(L,(S	,a))$	Reduce L → S
$(L,(L	,a))$	Shift

$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L, (L, S	)) $	Reduce L →L, S
$ (L, (L	)) $	Shift
$ (L, (L)	) $	Reduce S → (L)
$ (L, S	) $	Reduce L → L, S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

- The use of a **stack** in **shift-reduce parsing** is justified by an important fact: the **handle** will always eventually appear on top of the stack, never inside.
- This fact can be shown by considering the possible forms of two successive steps in any **rightmost derivation**.



Case (1)



Case (2)

Cases for two successive steps of a rightmost derivation

STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$
$\$\alpha\beta B$	$yz\$$
$\$\alpha\beta By$	$z\$$
$\$\alpha\gamma$	$xyz\$$
$\$\alpha Bxy$	$z\$$

In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle.

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used.
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a *shift/reduce conflict),* or cannot decide which of several reductions to make (a reduce/reduce conflict).
- We now give some examples of syntactic constructs that give rise to such grammars.
- Technically, these grammars are not in the LR(K) class of grammars defined we refer to them as non-LR grammars.
- The k in LR(k) refers to the number of symbols of look ahead on the input. Grammars used in compiling usually fall in the LR(1) class, with one symbol of look ahead at most.

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

- An ambiguous grammar can never be a LR grammar.

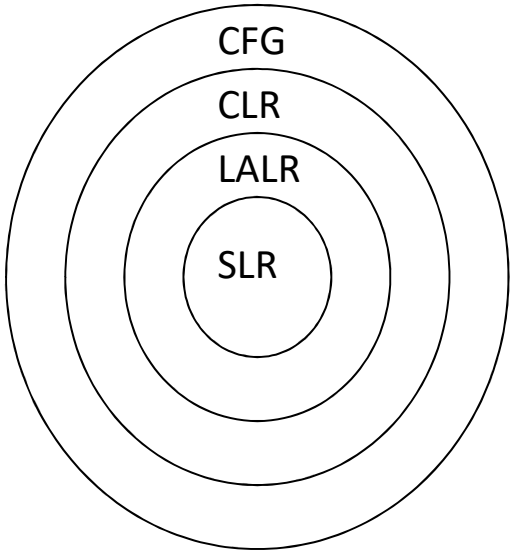
- **Introduction to simple LR Parsing:**
- Why LR Parsers – Items and LR(0) Automaton - The LR-Parsing Algorithm - Constructing SLR–Parsing Tables

Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

1. Operator-Precedence Parser

– simple, but only a small class of grammars.



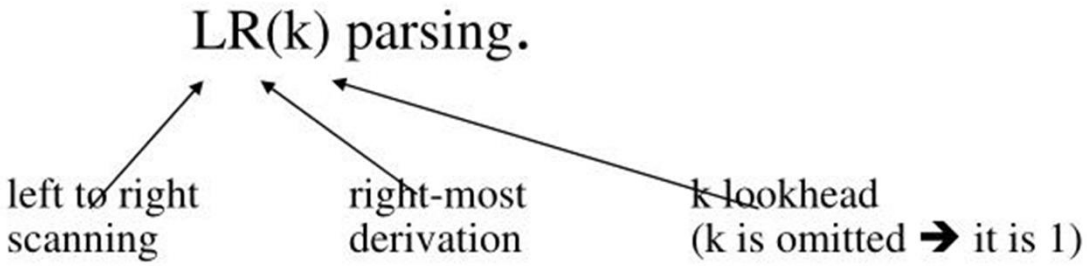
2. LR-Parsers

– covers wide range of grammars.
  - SLR – simple LR parser
  - Canonical LR – most general LR parser
  - LALR – intermediate LR parser (lookhead LR parser)

– SLR, CLR and LALR work same, only their parsing tables are different.

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:



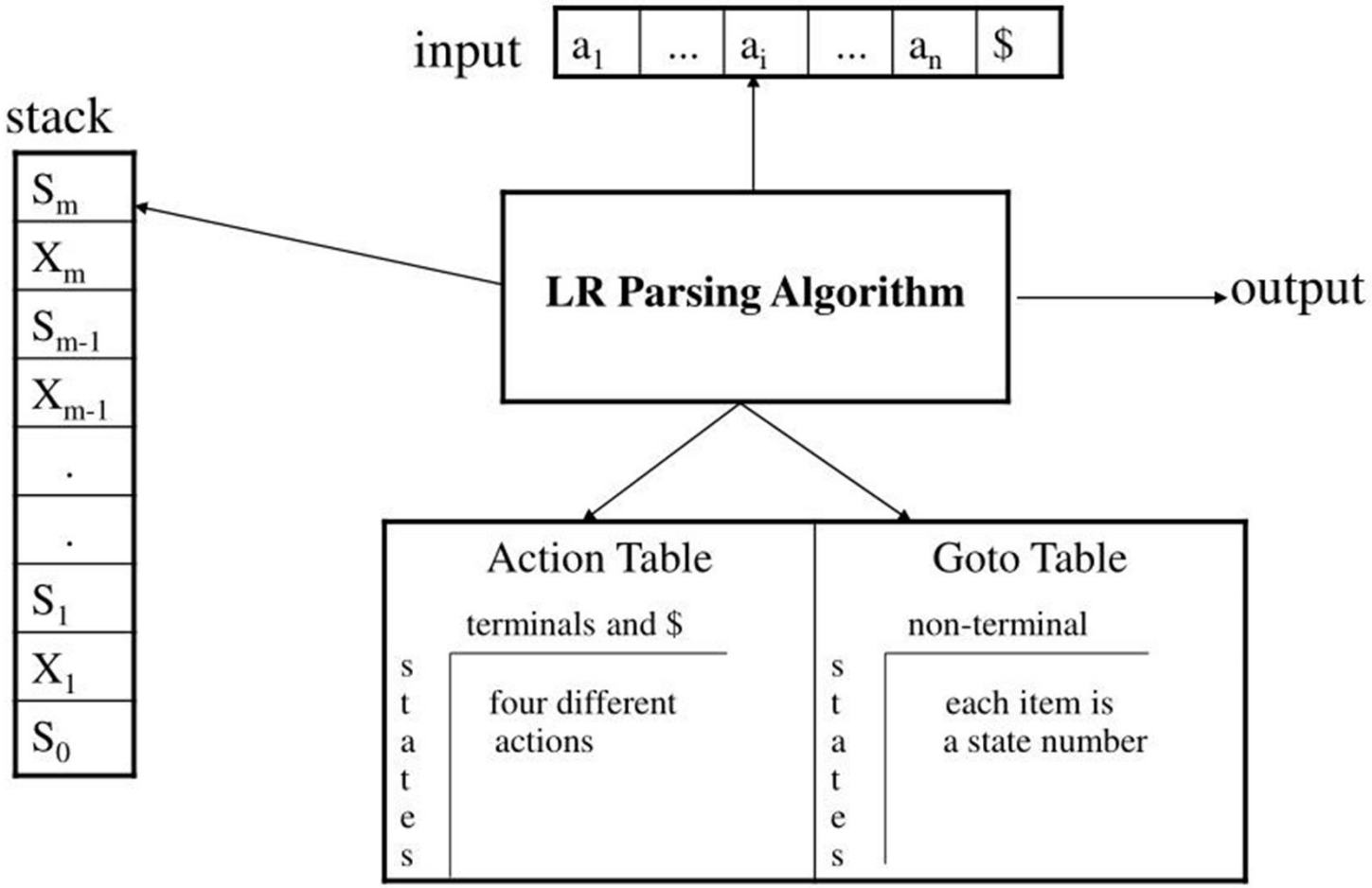
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.
  - Can recognize virtually all programming language constructs for which CFG can be written

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-ahead LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_m S_m a_i s, a_{i+1} \ldots a_n \$)$

2. **reduce $A \rightarrow \beta$** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **$s=goto[s_{m-r},A]$**

   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$

   - Output the reducing production reduce $A \rightarrow \beta$

3. **Accept** – If action$[S_m, a_i] =$ accept ,Parsing successfully completed
4. **Error** -- Parser detected an error (an empty entry in the action table) and calls an error recovery routine.

Constructing SLR Parsing Tables – LR(0) Item

- LR parser using SLR parsing table is called an SLR parser.
- A grammar for which an SLR parser can be constructed is an SLR grammar.
- An **LR(0) item** (**item**) of a grammar G is a production of G with a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: (four different possibility)
  - $A \rightarrow .aBb$
  - $A \rightarrow a.Bb$
  - $A \rightarrow aB.b$
  - $A \rightarrow aBb.$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A production rule of the form $A \rightarrow \varepsilon$ yields only one item $A \rightarrow .$
- Intuitively, an item shows how much of a production we have seen till the current point in the parsing procedure.

Constructing SLR Parsing Tables – LR(0) Item

- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- To construct the of canonical LR(0) collection for a grammar we define an augmented grammar and two functions- closure and goto.

- ***Augmented Grammar*:**

  G' is grammar G with a new production rule S'→S where S' is the new starting symbol. i.e $G \cup \{S' \rightarrow S\}$ where S is the start state of G.
- The start state of G' = S'.
- This is done to signal to the parser when the parsing should stop to announce acceptance of input.

Constructing SLR Parsing Tables – LR(0) Item

- **Complete and Incomplete Items:**

  An LR(0) item is complete if '.' Is the last symbol in RHS else it is incomplete.

  For every rule A $\rightarrow\alpha$, $\alpha \neq \varepsilon$, there is only one complete item A $\rightarrow\alpha.$, but as many incomplete items as there are grammar symbols.

Kernel and Non-Kernel items:

Kernel items include the set of items that do not have the dot at leftmost end.

S'→ .S is an exception and is considered to be a kernel item.

Non-kernel items are the items which have the dot at leftmost end.

Sets of items are formed by taking the closure of a set of kernel items.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

  1. Initially, every LR(0) item in I is added to closure(I).
  2. If $\mathbf{A \rightarrow \alpha \bullet B \beta}$ is in closure(I) and $\mathbf{B \rightarrow \gamma}$ is a production rule of G; then $\mathbf{B \rightarrow \bullet \gamma}$ will be in the closure(I).

  We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

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

closure({E' → •E}) =

{ E' → •E ⟵ kernel items
  E → •E+T
  E → •T
  T → •T*F
  T → •F
  F → •(E)
  F → •id }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta$ in I then every item in **closure({$A \rightarrow \alpha X.\beta$})** will be in goto(I,X).
  - If I is the set of items that are valid for some viable prefix γ, then goto(I,X) is the set of items that are valid for the viable prefix γX.

Example:

I ={ $E' \rightarrow E.$, $E \rightarrow E.+T$}

goto(I,+) = { $E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$ }

Example

I ={ E' → .E,
E → .E+T,
E → .T,
T → .T*F,
T → .F,
F → .(E),
F → .id }

goto (I, E) = { E' → E., E → E.+T }

goto (I, T) = { E → T., T → T.*F }

goto (I, F) = {T → F. }

goto(I,id) = { F → id. }

goto (I, ( ) = { F → (.E), E → .E+T,
E → .T, T → .T*F, T → .F,
F → .(E), F → .id }

STEP 1

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- ***Algorithm***:

Procedure items(G')

begin

C := { closure({S'→.S}) }

repeat for each set of items I in ***C*** and each grammar symbol X

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

add goto(I,X) to ***C***

until no more set of LR(0) items can be added to ***C***.

end

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

I_0: E' → .E.
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: goto(I_0, E)
E' → E.
E → E.+T

I_2: goto(I_0, T)
E → T.
T → T.*F

I_3: goto(I_0, F)
T → F.

I_4 : goto(I_0, ( )
F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: goto(I_0, id)
F → id.

I_6: goto(I_1, +)
E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: goto(I_2, *)
T → T*.F
F → .(E)
F → .id

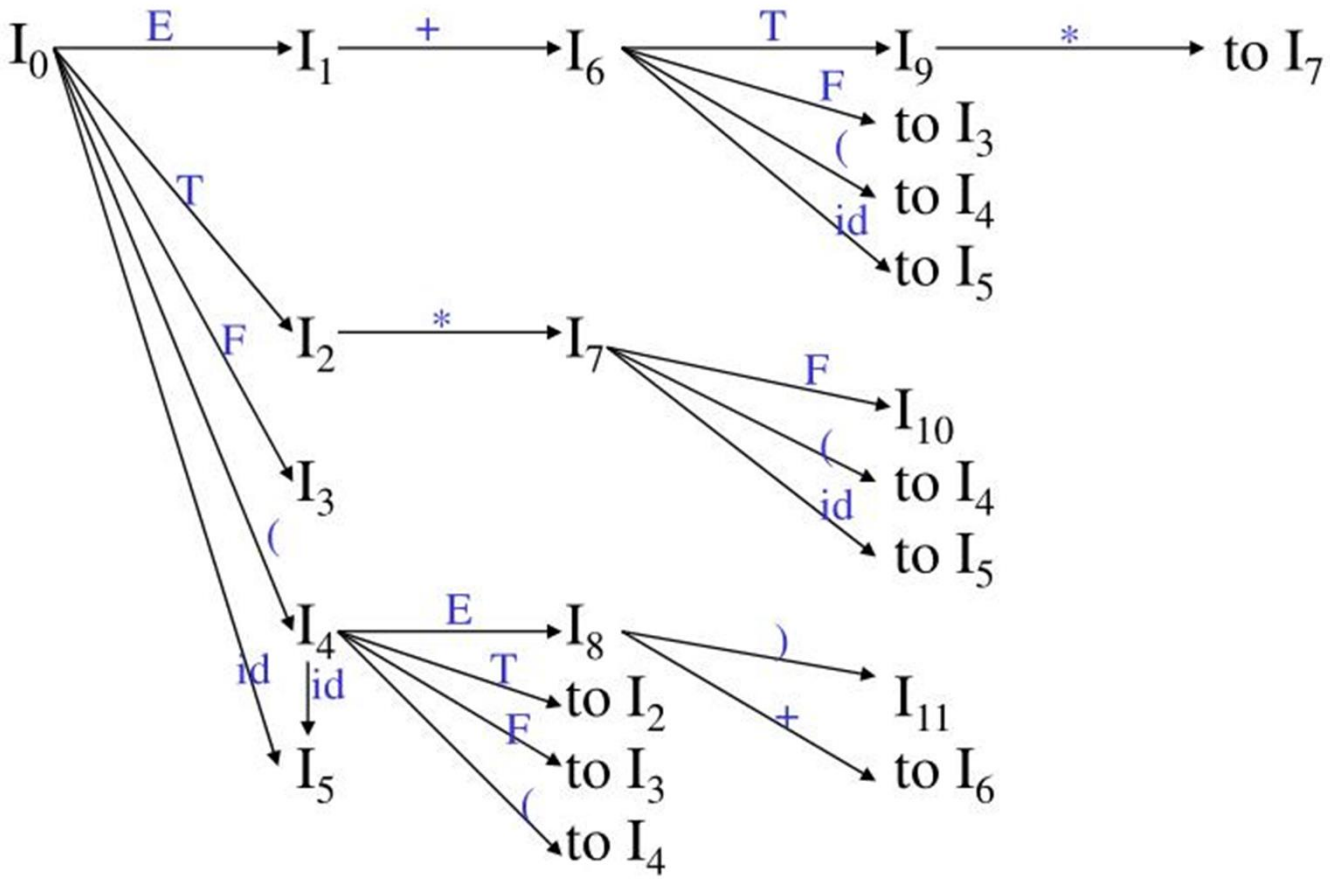
I_8: goto(I_4, E)
F → (E.)
E → E.+T

I_9:goto(I_6, T)
E → E+T.
T → T.*F

I_{10}: goto(I_7, F)
T → T*F.

I_{11} : goto(I_8,))
F → (E).

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'. $\mathbf{C \leftarrow \{I_0,...,I_n\}}$
2. State i is constructed from I_i. The parsing actions for state I are determined as follows:
   - If a is a terminal, $\mathbf{A \rightarrow \alpha.a\beta}$ in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j.***
   - If $\mathbf{A \rightarrow \alpha.}$ is in I_i, then **action[i,a]** is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ for all a in **FOLLOW(A)** where **A≠S'**.
   - If $\mathbf{S' \rightarrow S.}$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser is the one construcetd from the sets of items containing [S'→.S]

SLR Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

	Action Table							Goto Table		
state	**id**	**+**	*	(	)	**$**		**E**	**T**	**F**
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

LR Parsing Algorithm

- set ip to point to the first symbol in ω$
- initialize stack to S0
- repeat forever
- let ‘s’ be topmost state on stack & ‘a’ be symbol pointed to by ip
- if action[s,a] = shift s’
  - push a then s’ onto stack
  - advance ip to next input symbol
- else if action[s,a] = reduce $A \rightarrow \beta$
  - pop 2*| β | symbols of stack
  - let s’ be state now on top of stack
  - push A then goto[s’,A] onto stack
  - output production $A \rightarrow \beta$
- else if action[s,a] == accept
  - return success
- else
  - error()

Actions of SLR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	