Module 3

Bottom Up Parsing: Shift Reduce Parsing-Operator Precedence Parsing

LR Parsing – Constructing SLR parsing tables, Constructing canonical LR parsing tables and Constructing LALR parsing tables.

Bottom Up Parsing

A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it *reduces* it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

Shift - Reduce Parsing

Shift reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top). This can be considered as the process of "reducing" a string w to the start symbol of a grammar. At each reduction step parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it *reduces* it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

In Shift-reduce parsing 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. 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 β of grammar symbols on top of the stack. It

then reduces β 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 as follows:

STACK	INPUT
$ S	$

Upon entering this configuration, the parser halts and announces successful completion of parsing.

There are actually four possible actions a shift-reduce parser can make:

(1) shift (2) reduce (3) accept and (4) error.

1. **Shift**: The next input symbol is shifted onto the top of the stack.
2. **Reduce**: The parser knows the right end of the string to be reduced must be at the top of the stack. It must then locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**: Announce successful completion of parsing.
4. **Error:** Discover a syntax error has occurred and calls an error recovery routine.

Example:

Following figure steps through the actions a shift-reduce parser might take in parsing the input string id1 *id2 according to the expression grammar.

E→ E + T | T

T→ T * F | F

F→ (E) | id

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

Configurations of a shift-reduce parser on input id1 * id2

Operator Precedence Parsing

Bottom-up parsers for a large class of context-free grammars can be easily developed using *operator grammars*.

In an operator grammar, no production rule can have:

- ε at the right side (no production).
- two adjacent non-terminals at the right side.

This property enables the implementation of efficient *operator-precedence parsers*.

Ex:

E→AB E→EOE E→E+E |

A→a E→id E*E |

B→b O→+|*|/ E/E | id

not operator grammar not operator grammar operator grammar

Precedence Relations

In operator-precedence parsing, we define three precedence relations between certain pairs of terminals as follows:

Relation	Meaning
a <· *b*	*a* yields precedence to *b (b has higher precedence than a)*
a =· *b*	*a* has the same precedence as *b*
a ·> *b*	*a* takes precedence over *b (b has lower precedence than a)*

- The intention of the precedence relations is to find the handle of a right sentential form,
  <· with marking the left end,

  =· appearing in the interior of the handle, and

  ·> marking the right end.

In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals.

Example: Consider the string **id + id * id** and the grammar is: E→ E+E | E-E | E*E | E/E | (E) | -E | id

The corresponding precedence relations is:

	id	+	*	$
Id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Then the string with the precedence relations inserted is:

$ <. id .> + <. id .> * <. id .> $

<. is inserted between the leftmost $ and id since <. is the entry in row $ and column id.

The handle can be found by the following rules:

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

In the above example:

$ **<· id ·>** + <· id ·> * <· id ·> $	E → id	$ **id** + id * id $
$ <· + **<· id ·>** * <· id ·> $	E → id	$ E + **id** * id $
$ <· + <· * **<· id ·>** $	E → id	$ E + E * **id** $
$ <· + **<· * ·>** $	E → E*E	$ E + **E * E** $
$ **<· + ·>** $	E → E+E	$ **E + E** $
$ $		$ E $

Operator Precedence Parsing Algorithm

Input: An input string w and a table of precedence relations.

Output: If w is well formed, a skeletal parse tree otherwise an error indication

Initialize: The stack contains $ and input buffer the string w$.

Algorithm:

set ip to point to the first symbol of w$;

repeat forever

```
        if ( $ is on top of the stack and ip points to $ ) then
                return
        else
       {
            let a be the topmost terminal symbol on the stack
           let b be the symbol pointed to by ip;

            if ( a <· b or a =· b ) then
          {     /* SHIFT */
               push b onto the stack;
               advance ip to the next input symbol;
           }
           else if ( a ·> b ) then              /* REDUCE */
           {
                    repeat
                    pop stack
                  until ( the top of stack terminal is related by <· to the terminal most
                          recently popped )
           }
           else error();
         }
```

Operator Precedence Parsing Algorithm – Example

stack	***input***	***action***		
$	id+id*id$	$ $\lessdot$ id	shift	
$id	+id*id$	id $\gtrdot$ +	reduce	E $\rightarrow$ id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id $\gtrdot$ *	reduce	E $\rightarrow$ id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id $\gtrdot$ $	reduce	E $\rightarrow$ id
$+*	$	* $\gtrdot$ $	reduce	E $\rightarrow$ E*E
$+	$	+ $\gtrdot$ $	reduce	E $\rightarrow$ E+E
$	$	accept		

Operator Precedence Grammar Advantages and Disadvantages:

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages
- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

Construction of precedence relation

1. For each non-terminal, i.e. left hand side, construct a Firstop list containing the first terminal in each production for that non-terminal. Where a non-terminal is the first symbol on the right hand side, include both it and the first terminal following, e.g. for

$$X \rightarrow \mathrm{a}\ldots|B\mathrm{c}$$

include a, c and B in X's *Firstop* list.

2. Similarly construct a *Lastop* list for each non-terminal, e.g. for

$$Y \rightarrow \ldots \mathrm{u} \mid \ldots \mathrm{v}\, W$$

include u, v and W in Y's *Lastop* list.

3. Compute the *Firstop*+ and *Lastop*+ lists using Warshall's Closure Algorithm, as follows:

   (a) Take each non-terminal in turn, in any order and look for it in all the Firstop lists. Add its own first symbol list to any other in which it occurs. Similarly process the Lastop lists.

   (b) The non-terminals may now be deleted from the lists.

4. Construct the precedence matrix by the following rules

   (a) Wherever terminal a immediately precedes non-terminal B in any production, put $\mathrm{a} \lessdot \alpha$, where α is any terminal in the *Firstop*+ list for B.

   (b) Wherever terminal b immediately follows non-terminal C in any production, put $\mathrm{b} \gtrdot \beta$, where β is any terminal in the *Lastop*+ list for C.

   (c) Wherever a sequence aBc or ac occurs in any production, put $\mathrm{a} \doteq \mathrm{c}$.

5. Add the relations \$ ⋖ a and a ⋗ \$ for all terminals in the *Firstop+* and *Lastop+* lists, respectively, for *S*.

Any entries left blank indicate that the two symbols should never occur consecutively in a handle. Such a sequence is a syntax error.

Example:

$$
\begin{array}{lcl}
S & \rightarrow & A \\
A & \rightarrow & T \mid A + T \mid A - T \\
T & \rightarrow & F \mid T * F \mid T / F \\
F & \rightarrow & P \mid P \uparrow F \\
P & \rightarrow & \mathrm{i} \mid \mathrm{n} \mid (A)
\end{array}
$$

Applying steps 1 and 2 gives us

Symbol	Firstop	Lastop
S	A	A
A	T + A -	T + -
T	F * T /	F * /
F	P ↑	P ↑ F
P	i n (	i n)

Applying step 3 gives us

Symbol	Firstop+	Lastop+
S	T + A - F * / P ↑ i n (	A T + - F * / P ↑ i n)
A	T + A - F * / P ↑ i n (	T + - F * / P ↑ i n)
T	F * T / P ↑ i n (	F * / P ↑ i n)
F	P ↑ i n (	P ↑ F i n)
P	i n (	i n)

Removing non-terminals gives

Symbol	Firstop+	Lastop+
S	+ - * / ↑ i n (	+ - * / ↑ i n)
A	+ - * / ↑ i n (	+ - * / ↑ i n)
T	* / ↑ i n (	* / ↑ i n)
F	↑ i n (	↑ i n)
P	i n (	i n)

Finally we use steps 4 and 5 to compute our precedence relation matrix.

	$	(	)	i	n	↑	*	/	+	-
$		⋖		⋖	⋖	⋖	⋖	⋖	⋖	⋖
(		⋖	≐	⋖	⋖	⋖	⋖	⋖	⋖	⋖
)	⋗		⋗			⋗	⋗	⋗	⋗	⋗
i	⋗		⋗			⋗	⋗	⋗	⋗	⋗
n	⋗		⋗			⋗	⋗	⋗	⋗	⋗
↑	⋗	⋖	⋗	⋖	⋖	⋖	⋗		⋗	⋗
*	⋗	⋖	⋗	⋖	⋖	⋖	⋗	⋗	⋗	⋗
/	⋗	⋖	⋗	⋖	⋖	⋖	⋗	⋗	⋗	⋗
+	⋗	⋖	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
-	⋗	⋖	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗

Error Recovery in Operator-Precedence Parsing

Error Cases:

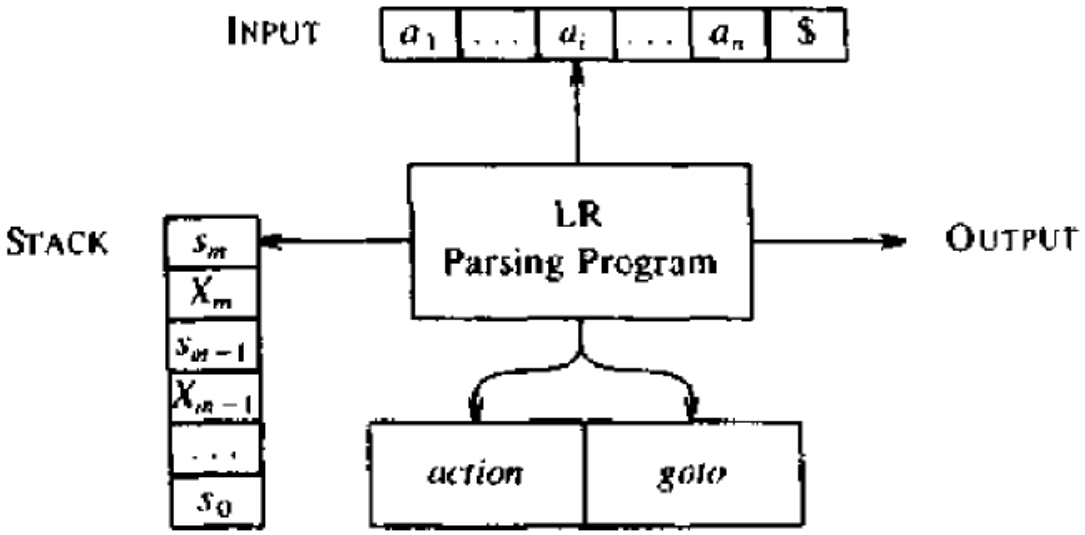
1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side.

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle "looks like" which right hand side. And tries to recover from that situation.
   Ex:

LR Parsers

- Efficient Bottom up parser
- LR(k):
  - L: Left to Right scanning of input
  - R: Right most derivation in reverse
  - k: number of input symbols of lookahead that are used in making parsing decisions
  - If k is not mentioned, LR(1) parser



Model of an LR parser.

- Requirements of LR parser:
  - Input Buffer
  - Stack
  - Parsing Table
  - LR Parsing Program
- **Stack**
  - Parsing program uses a stack to store string of the form **s0X1s1X2s2X3…Xmsm** where sm is on the top
  - Xi is a grammar symbol and Si is the state
  - Each state symbol summarizes the information contained in the stack below it
  - Combination of state symbol on stack top and current input symbol are used to index the parsing table and determine the shift reduce parsing decision
- **Parsing Table:**
  - Consist of 2 parts
    - 1. parsing action function ***action***
    - 2. goto function ***goto***
- **Parsing Program works as follows:**

- It determines sm, the state currently on top of stack and ai the current input symbol
- It consults parsing action table entry for ***action[sm,ai]*** which can have 4 values:
    1. Shift s, where s is a state
    2. Reduce by a grammar production A→b
    3. Accept
    4. Error
- The function ***goto*** takes a state and grammar symbol as arguments and produces a state.

- A **configuration** of LR parser is a pair whose first component is stack contents and second component is remaining input:

(s0 X1 s1 X2 . . .Xmsm, ai ai+1 . . . an $)

- The next move of the parser is determined by reading ***ai***, the current input symbol and ***sm***, state on stack top and then consulting parsing action table entry ***action[sm,ai].***
- The configuration resulting after each of four types of moves are as follows:

1. If $action[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, \quad a_{i+1} \cdots a_n\$)$$

Here the parser has shifted both the current input symbol a_i and the next state s, which is given in $action[s_m, a_i]$, onto the stack; a_{i+1} becomes the current input symbol.

2. If $action[s_m, a_i]$ = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, \quad a_i a_{i+1} \cdots a_n \$)$$

where $s = goto[s_{m-r}, A]$ and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for $goto[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move.

3. If $action[s_m, a_i]$ = accept, parsing is completed.

4. If $action[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

LR Parsing Algorithm

Input. An input string w and an LR parsing table with functions *action* and *goto* for a grammar G.

Output. If w is in $L(G)$, a bottom-up parse for w; otherwise, an error indication.

Method. Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the following program until an accept or error action is encountered.

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A → β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A → β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

Example:

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

STATE	action						goto		
	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			

Parsing table for expression grammar.

	STACK	INPUT	ACTION
(1)	0	id * id + id $	shift
(2)	0 id 5	* id + id $	reduce by $F \rightarrow$ id
(3)	0 F 3	* id + id $	reduce by $T \rightarrow F$
(4)	0 T 2	* id + id $	shift
(5)	0 T 2 * 7	id + id $	shift
(6)	0 T 2 * 7 id 5	+ id $	reduce by $F \rightarrow$ id
(7)	0 T 2 * 7 F 10	+ id $	reduce by $T \rightarrow T*F$
(8)	0 T 2	+ id $	reduce by $E \rightarrow T$
(9)	0 E 1	+ id $	shift
(10)	0 E 1 + 6	id $	shift
(11)	0 E 1 + 6 id 5	$	reduce by $F \rightarrow$ id
(12)	0 E 1 + 6 F 3	$	reduce by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E+T$
(14)	0 E 1	$	accept

Moves of LR parser on id * id + id.

Construction of LR Parsing Table

- 3 techniques for constructing LR parsing table for a grammar:
  - 1. Simple LR (SLR)
  - 2. Canonical LR
  - 3. LookAhead LR(LALR)
- Simple LR (SLR)
  - Easy to implement
  - Least powerful
- Canonical LR
  - Most powerful
  - Most expensive
- LALR
  - Intermediate in power and cost between other 2

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: $A \rightarrow .aBb$

  (four different possibility) $A \rightarrow a.Bb$

  $A \rightarrow aB.b$

  $A \rightarrow aBb.$
- A production rule of the form A → ε yields only one item A → **.**
- Intuitively, an item shows how much of a production we have seen till the current point in the parsing procedure.
- 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:
    - **Augmented grammar**
    - **Two functions: closure and goto**.

Augmented Grammar:

- G' is the augmented grammar of 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.

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 $\mathbf{A \rightarrow \alpha.B\beta}$ is in closure(I) and $\mathbf{B \rightarrow \gamma}$ is a production rule of G; then $\mathbf{B \rightarrow .\gamma}$ will be in the closure(I).
- We will apply this rule until no more new LR(0) items can be added to closure(I).

Computation of Closure

```
function closure ( I )
begin
        J := I;
        repeat
                for each item A → α.Bβ in J and each production
                        B→γ of G such that B→.γ is not in J do
                                add B→.γ to J
```

until no more items can be added to J

return J

end

The Closure Operation -- Example

Grammar:

E' → E

E → E+T

E → T

T → T*F

T → F

F → (E)

F → id

closure({E' → • E}) =

{ E' → • E

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 → α.Xβ in I then every item in **closure({A → αX.β})** 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' → E., E → E.+T}

goto(I,+) = { E → E+.T

T →.T*F

T →.F

F →.(E)

F →.id }

Construction of 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

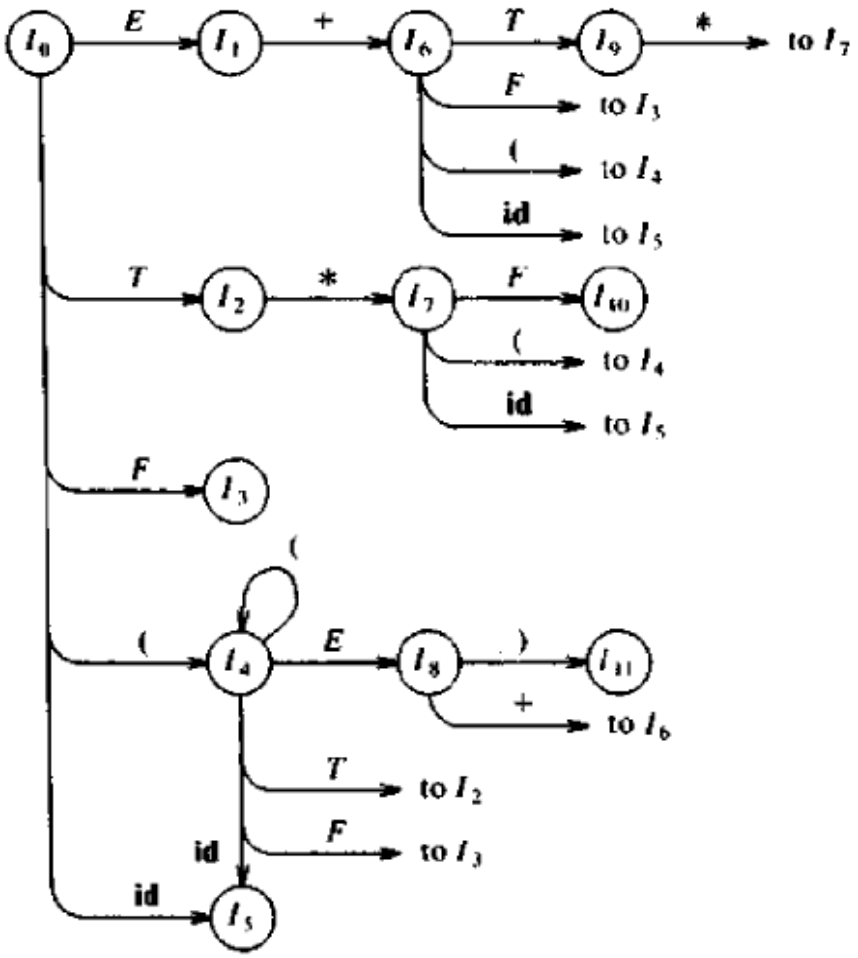
I_8: goto(I_4, E)
F → (E.)
E → E.+T

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

1. Construct the canonical collection of sets of LR(0) items for G'. **C←{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, **A→α.aβ** in I_i and goto(I_i,a)=I_j then action[i,a] is ***shift j*.**
    - If **A→α.** is in I_i , then **action[i,a]** is ***reduce A→α*** for all a in **FOLLOW(A)** where **A≠S'**.
    - If **S'→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]

Parsing Tables of Expression Grammar

	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				

Shift/Reduce and Reduce/Reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is **a**:
  - if the $A \rightarrow \alpha.$ in the I_i and **a** is FOLLOW(A)
  - In some situations, when state i appears on stack top, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in right sentential form.
- Thus the reduction by $A \rightarrow \alpha$ would be invalid on input a.

- Because of that we go for : Canonical LR Parser
  - In this, it is possible to carry more information in the state that will allow us to avoid some of these invalid reductions
- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

  $A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item

  (**a** is a terminal or end-marker.)
- Such an object is called LR(1) item.
  - 1 refers to the length of the second component
  - The lookahead has no effect in an item of the form $[A \rightarrow \alpha.\beta,a]$, where β is not $\in$.
  - But an item of the form $[A \rightarrow \alpha.,a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.
- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, **except that *closure* and *goto* operations work a little bit different.**

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha.B\beta,a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:

– If $A \rightarrow \alpha.X\beta,a$ in I then every item in **closure({A → αX.β,a})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

Algorithm: (Exactly same for LR(0))

C is { closure({S'→.S,$}) }

repeat the followings until no more set of LR(1) items can be added to ***C***.

for each 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***

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'. $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.,a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where A≠S'.
   - If $S' \rightarrow S.,\$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(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 contains S'→.S,$

Constructing LALR Parsing Table

- **LALR** stands for **Lookahead LR.**
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.

- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrik process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L.=R,\$$ ➔ $S \rightarrow L.=R$ Core

$R \rightarrow L.,\$$ $R \rightarrow L.$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1:$L \rightarrow id.,=$ A new state: I_{12}: $L \rightarrow id.,=$

➔ $L \rightarrow id.,\$$

I_2:$L \rightarrow id.,\$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

1. Construct C={I0, I1,..,In}, the collection of sets of LR(1) items
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union. $C=\{I_0,...,I_n\}$ ➔ $C'=\{J_1,...,J_m\}$ where $m \leq n$

3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.

    1. Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    ➔ cores of $goto(I_1,X),...,goto(I_k, X)$ must be same.

2. So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

    1. If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
    - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
    - The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing

Phrase-Level Error Recovery in LR Parsing

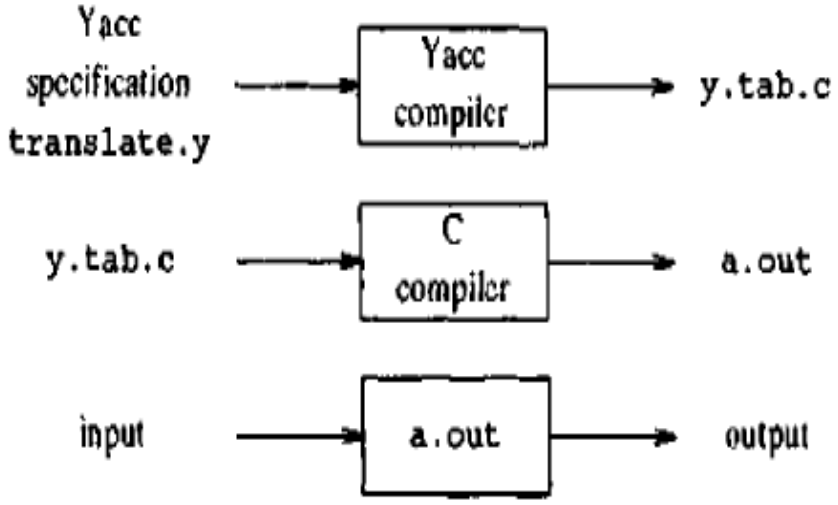
- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.

- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

YACC

- Parser Generator
- YACC – Yet Another Compiler Compiler
- Uses LALR parsing
- YACC generates C Code for syntax analyzer or parser
- YACC uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree

How YACC Works?



- Compiled as:

  yacc translate.y

  cc y.tab.c –ly

  where ly is the library that contains LR parsing program

A yacc source program has 3 parts:

declarations

%%

translation rules

%%

supporting c routines

Declaration Part:

- 2 optional sections
- In first section, we put ordinary C declarations delimited by **%{ and %}**. Here we place declarations of any temporaries used by the translation rules or procedures of 2nd and 3rd sections
- In second section, declarations of grammar tokens are present

Translation Rules part:

- Each rule consist of a grammar production and associated semantic action
- The production

<left side> → <alt 1> | <alt 2> | · · · | <alt *n*>

would be written in Yacc as

```
<left side>    :   <alt 1>    { semantic action 1 }
               |   <alt 2>    { semantic action 2 }
            . . .
               |   <alt n>    { semantic action n }
               ;
```

- Yacc semantic action is a sequence of C statements.
- Ex:

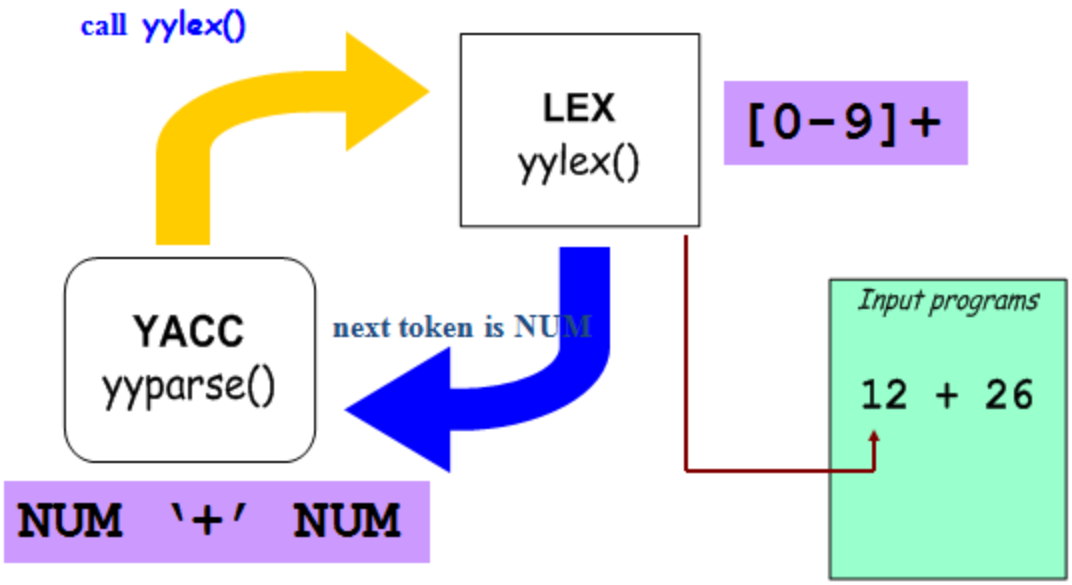
```
expr : expr '+' expr     {$$ = $1 + $3;}
     | expr '*' expr     {$$ = $1 * $3;}
     ;
```

- **$$** refers to the attribute value associated with the nonterminal on left
- **$i** refers to the value associated with the ith grammar symbol on the right

- In a YACC production,
  - a quoted single character ‘c’ is taken to be the **terminal** symbol c
  - unquoted strings of letters and digits not declared to be tokens are taken to be **non terminals**

Supporting C routines part:

- yylex() is must
- If necessary, error recovery routines can be added



YACC Specification of a simple desk calculator

Grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{digit}$$

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :   expr '\n'          { printf("%d\n", $1); }
        ;
expr    :   expr '+' term      { $$ = $1 + $3; }
        |   term
        ;
term    :   term '*' factor    { $$ = $1 * $3; }
        |   factor
        ;
factor  :   '(' expr ')'       { $$ = $2; }
        |   DIGIT
        ;

%%

yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```