160321733011

ASSIGNMENT - 1

Short Questions:

Solve left factoring with an example

In general if

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

is a production then it is not possible for us to take a decision whether to choose first rule or second.

In such a situation the above grammar can be left factored as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Example:-

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

The left factored grammar becomes

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

---

2. Create Recursive Descent Parsing with an example

A parser that uses collection of Recursive procedures for parsing parsing the given input string is called Recursive Descent Parser.

For each non terminal a seperate procedure is written and body of the procedure (code) is R.H.S of the corresponding non-terminal

Basic steps for Construction of RD Parser

- The R.H.S of the rule is directly converted into program code symbol by symbol.
- If i/p symbol is non-terminal then a call to procedure corresponding to the non-terminal is made.
- If the i/p symbol is terminal then it is matched with the lookahead from i/p.
  The lookahead pointer must be advanced on matching of the i/p symbol.
- If the production rule has many alternates then has to combined into single body of procedure.
- The parser should be activated by a procedure corresponding to the start symbol.

Example:- $E \rightarrow num\ T$

$T \rightarrow * num\ T \mid \epsilon$

For the given grammar RD Parser is;

```
procedure E
  { if lookahead = {num} then
      {
        match (num);
        T;
      }
    else
      error;
    if lookahead = '$'
      { declare success;
      }
    else error;
```

160321733011

```
procedure T
 {
   if lookahead = '*'
         {
           match ('*');
         }
    if lookahead = 'num'
         {
          match ('num');
           t;
         }
          else
              error;

         }
        else
              NULL;
   }
procedure match (token t)
   { if lookahead = t
     lookahead = next_token;
     else
        error
   }
procedure error
   { print (" Error !! ");
   }
```

3. Explain about SR and RR conflicts in parsing.

Ans SR (Shift-Reduce) Conflict:-

An SR conflict occurs when the parser's action table encounters a situation where it must choose between shifting the next i/p symbol on the stack (shift) or reducing the sym on the stack using a production rule (Red This conflict arises in Ambigious grammar.

If SR Conflict arises, then it can be avoided using SHIFT only.

R-R (Reduce-Reduce) Conflict:-

An RR conflict occurs when the parsers act table encounters a situation where it must choose between two different production rule to reduce the symbols on the stack.

If RR conflict arises, then it can be avoid using first Reduce.

These conflicts can be avoided by using the PRECEDENCE & ASSOCIATIVITY.

Long Questions

4. Perform shift reduce parsing with an exam

Ans Shift reduce parser attempts to construct parse tree from leaves to root.

It works on the principle of bottom-up parser

A shift reduce parser requires following data-structures

① The i/p buffer for storing the i/p string

② A stack for storing and accessing the L.H.S & RHS of ~~rules~~ rules

The parser performs following basic operations

1. SHIFT :- moving of the symbols from i/p buffer onto the stack.

2. REDUCE :- If the handles appears on top of the stack then reduction of it by appropriate rule is done. i.e RHS of rule is popped off and LHS is pushed in.

3. ACCEPT :- If the stack contains a START symbol only and i/p buffer is empty at the same time then that action is called ACCEPT.

4. ERROR :- A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called ERROR.

Example :- $E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow id$

Perform shift-reduce parsing on the i/p string

id - id * id.

Stack	I/p Buffer	Parsing Action
$	id-id*id	shift id.
$id	-id*id	reduce E→id
$E	-id*id	shift -
$E-	id*id	shift id
$E-id	*id	reduce E→id
$E-E	*id	shift *
$E-E*	id	shift id
$E-E*id	$	reduce E→id
$E-E*E	$	reduce E→E*
$E-E	$	reduce E→E-
$E	$	accept.

5 Construct CLR parsetable for the following gramm

$S \rightarrow CC$
$C \rightarrow aC$
$C \rightarrow d$

Ans STEP 1:- Construction of canonical set of items along with the lookahead.

for the given grammar, we initially add $S' \rightarrow .S$ in the set of item C.

$S' \rightarrow .S$
$S \rightarrow .CC$
$C \rightarrow .aC$
$C \rightarrow .d$

160321733011

We initially add $S' \to .S$, $ as first role in I_0.

Now match $S' \to .S$, $

$A \to \alpha . X\beta$, a

$A = S'$, $\alpha = \varepsilon$, $X = S$, $\beta = \varepsilon$, $a = \$$

$X \to \gamma$, $b \in$ FIRST (β, a)

$S \to .CC$ FIRST $(\varepsilon, \$)$

$b = \$$

$\therefore$ $S \to .CC$, $ will be added to I_0

Now match $S \to .CC$, $

$A \to \alpha . X\beta$, a

$A = S$, $\alpha = \varepsilon$, $X = C$, $\beta = C$, $a = \$$

$X \to \gamma$, $b \in$ FIRST (β, a)

$C \to .aC$, $b \in$ FIRST $(C, \$)$

$C \to .d$, $b = a/d$

$\therefore$ $C \to .aC$, a/d

$C \to .d$, a/d will be added to I_0

Hence,

I_0 :

$S' \to .S$, $

$S \to .CC$, $

$C \to .aC$, $a/d

$C \to .d$, a/d.

Applying goto on I_0

goto (I_0, S)

I_1 : $S' \to S.$, $.

Repeat closure

goto (I_0, C) Apply closure

Now match

$S \to C.C, \$$

$A \to \alpha.X\beta, a$

$A \Rightarrow S$, $\alpha = C$, $X = C$, $\beta = \varepsilon$, $a = \$$

$X \to \gamma$, $b \in$ FIRST (β, a)

$C \to .aC, \$$

$C \to .d, \$$

goto (I_0, C)

I_2 :

$S \to C.C, \$$

$C \to .aC, \$$

$C \to .d, \$$

goto (I_0, a)

~~$C \to a.C$~~

$C \to a.C, a/d$

Applying closure

$A \to \alpha.X\beta, a$

$A = C$, $\alpha = a$, $X = C$, $\beta = \varepsilon$, $a = a/d$

$C \to .aC, a/d$

$C \to .d, a/d$

goto (I_0, a)

I_3 :

$C \to a.C, a/d$

$C \to .aC, a/d$

$C \to .d, a/d$

goto (I0, d)

I_4 : C → d. , a/d

Applying goto on I_0 is completed.

Now applying goto on I_1

It is a kernel item. goto cannot be applied

Now applying goto on I_2

goto (I_2, C)

I_5 : S → CC. , $

goto (I_2, a)

I_6 : C → a.C, $
C → .aC, $
C → .d , $

goto (I_2, d)

I_7 : C → d. , $

Applying goto on I_2 is completed

Now applying goto on I_3

goto (I_3, C)

I_8 : C → aC. , a/d

goto (I_3, a)

C → a.C, a/d - I_3

goto (I_3, d)

C → d. , a/d - I_4

repeated states.

Applying goto on I_3 is completed.

Now a
Cannot apply goto on I_4, I_5 since they Kernel items.

Now applying goto on I_6.

goto (I_6, C)

I_9 : $C \to aC.$, \$

goto (I_6, a)

$C \to a.C$, \$ - I_6

goto (I_6, d)

$C \to d.$, \$ - I_7

repeated states

STEP 2:- Construction of CLR Parsing Table

CLR Parsing Table

States	Action			Goto	
	a	d	\$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		9
6	S_6	S_7			
7			r_3		
8	r_2	r_2			
9			r_2		

160321733011

STEP 3 :- Parsing the Input string

Let i/p - 'aadd'

Stack	I/P Buffer	Parsing Action
$0	aadd$	Shift a
$0a3	add$	shift a
$0a3a3	dd$	shift d
$0a3a3d4	d$	reduce C → d
$0a3a3C8	d$	reduce C → aC
$0a3C8	d$	reduce C → aC
$0C2	d$	shift d
$0C2d7	$	reduce C → d
$0C2C5	$	reduce S → CC
$0S1	$	Accept.