EXERCISE 1

AIM: Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and newlines

DESCRIPTION:

Lexical analysis reads the characters in the source program and groups them into stream of tokens in which each token represents a logically cohesive sequence of characters such as an identifier, keyword, and punctuation character. The character sequence forming a token is called lexeme of the token.

Example: A=B+C*24;

A→ identifier

= → assignment operator

B→ identifier

+→ Arithmetic operator

C→ identifier

*→ Arithmetic operator

ALGORITHM:

Step 1: Declare the necessary variables.

Step 2: Declare an array and store the keywords in that array

Step 3: Open the input file in read open

Step 4: read the string from the file till the end of file.

- If the first character in the string is # then print that string as header file
- If the string matches with any of the keywords print that string is a Keyword
- If the string matches with operator and special symbols print the Corresponding message
- If the string is not a keyword then print that as an identifier.

INPUT:

```
#include<stdio.h>
void main()
{
int a;
double b;
char c;
 printf("%d %b %c",a,b,c);
}
```

OUTPUT:

#include<stdio.h>

Void	Keyword
Main	Keyword
(	Left Parenthesis
)	Right Parenthesis
{	Open Brace
Int	Keyword
A	Identifier
;	Semicolon
B	Identifier
;	Semicolon
Char	Keyword
C	Identifier
;	Semicolon
(	Left Parenthesis
%c	Control string
,	Comma
A	Identifier
,	Comma
A	Identifier

, Comma

) Right Parenthesis

; Semicolon

} Close Brace

VIVA QUESTIONS:

1. What is a compiler?
2. What are the two parts of a compilation? Explain briefly.
3. List the subparts or phases of analysis part.
3. What is the output of lexical analyzer?
4. List the various phases of a compiler.
5. What are the classifications of a compiler?
6. Mention some of the cousins of a compiler.
7. List the phases that constitute the front end of a compiler.
8. Mention the back-end phases of a compiler.
9. Differentiate tokens
10. Differentiate patterns
11. Differentiate LEXEME.
12. List the various error recovery strategies for a lexical analysis.

EXERCISE 2

AIM: Simulate first and follow of a grammar

DESCRIPTION:

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.

FIRST(α):

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α→ϵ then ϵ is also in FIRST (α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set:

1. If x is terminal, and then FIRST(X) is {X}.
2. If X →ϵ is a production, then add ϵ to FIRST(X).
3. If X is non terminal and X →Y1 Y2 ... YK. is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ϵ is in all of FIRST(Y1), ... , FIRST(Yi-1); that is, Y1, ... ,Yi-1 → ϵ. If ϵ is in FIRST (Yj) for all j = 1, 2, ... , k, then add ϵ to FIRST(X). For example, everything in FIRST (Y1) is surely in FIRST(X). If Y1 does not derive ϵ, then we add nothing more to FIRST(X), but if Y1→ϵ, then we add FIRST (Y2) and so on.

FOLLOW (A):

For nonterminal A, if there are set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S→αAaβ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived ϵ and disappeared. If A can be the rightmost symbol in some sentential form, then $, representing the input right endmarker, is in FOLLOW (A).

To compute FOLLOW (A) for all no terminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production A→αBβ, then everything in FIRST(β), except for ϵ, is placed in

FOLLOW(B)

3. If there is a production A → αB, or a production A →αBβ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

ALGORITHM:

1. **Compute FIRST(X) as follows:**
   - if X is a terminal, then FIRST(X)={X}
   - if X→ε is a production, then add ε to FIRST(X)
   - if X is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs
2. **Compute FOLLOW as follows:**
   - FOLLOW(S) contains $, If S is the starting symbol of the grammar.
   - For productions A→αBβ, everything in FIRST(β) except ε goes into FOLLOW(B)
   - For productions A→αB or A→αBβ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

OUTPUT:

E →T E’

E’→ + T E’ | ϵ

T → F T’

T’→ * F T’ | ϵ

F → (E) | id

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E’) = {+, ϵ}

FIRST (T’) = {*, ϵ}

FOLLOW (E) = FOLLOW (E’) = {), $}

FOLLOW (T) = FOLLOW (T’) = {+,), $}

FOLLOW (F) = {+, *,), $}

VIVA QUESTIONS

1. Define parser.
2. Mention the basic issues in parsing.
3. Why lexical and syntax analyzers are separated out?
4. Define a context free grammar.
5. Briefly explain the concept of derivation.
6. Define ambiguous grammar.
7. What are the problems with top down parsing?
8. Write the algorithm for FIRST and FOLLOW.

EXERCISE 3

AIM: Develop an operator precedence parser for a given language

DESCRIPTION

Operator precedence grammars rely on the following three precedence relations between the terminals:

Relation	Meaning
a <• b	a yields precedence to b
a =• b	a has the same precedence as b
a •> b	a takes precedence over b

These operator precedence relations allow delimiting the handles in the right sentential forms: <• marks the left end, =• appears in the interior of the handle, and •> marks the right end. Contrary to other shift-reduce parsers, all non-terminals are considered equal for the purpose of identifying handles.The relations do not have the same properties as their un-dotted counterparts; e. g. a =• b does not generally imply b =• a, and b •> a does not follow from a <• b. Furthermore, a =• a does not generally hold, and a •> a is possible.

Let us assume that between the terminals a_i and a_{i+1} there is always exactly one precedence relation. Suppose that $ is the end of the string. Then for all terminals b we define: $ <• b and b •> $. If we remove all non-terminals and place the correct precedence relation: <•, =•, •> between the remaining terminals, there remain strings that can be analyzed by an easily developed bottom-up parser.

Example:

If the grammar is

S→if C then S|a

C→b

Then

LEADING(S)={if,a}

LEADING(C)={b}

TRAILING(S)=then,a}

TRAILING(C)={b}

Precedence table:

	If	**Then**	**a**	**b**	**$**
If		=		<	
Then	<		<		>
A					>
B		>			
$	<		<		

ALGORITHM:

LEADING:

If a is in LEADING (A) if there is a production of the form A→ Ɣaδ where Ɣ is ϵ or a single non terminal.

TRAILING:

If a is in TRAILING (B) if there is a production of the form A→ Ɣaδ where δ is ϵ or a single non terminal.

TABLE DESIGNING:

Execute the following for each production $A \rightarrow x_1x_2x_3 \ldots x_n$ do

1. If x_i and x_{i+1} are terminals then set $x_i = x_{i+1}$
2. If i<=n-2 and x_i and x_{i+2} are terminals and x_{i+1} is a single non-terminal then set $x_i = x_{i+2}$
3. If x_i is a terminal and x_{i+1} is a non-terminal then for all a in LEADING(x_{i+1}) do set x_i <a
4. If x_i is a non-terminal and x_{i+1} is a terminal then for all a in TRAILING(x_i) do set a>x_{i+1}
5. Set $<a for all a in LEADING(S)

   And set b>$ for all b in TRAILING(S) where S is start symbol of G.

OUTPUT:

enter non terminals

+ - * /

Operator Precedence Parsing Table:

	+	-	*	/
+	=	=	<	<
-	=	=	<	<
*	>	>	=	=
/	>	>	=	=

VIVA QUESTIONS

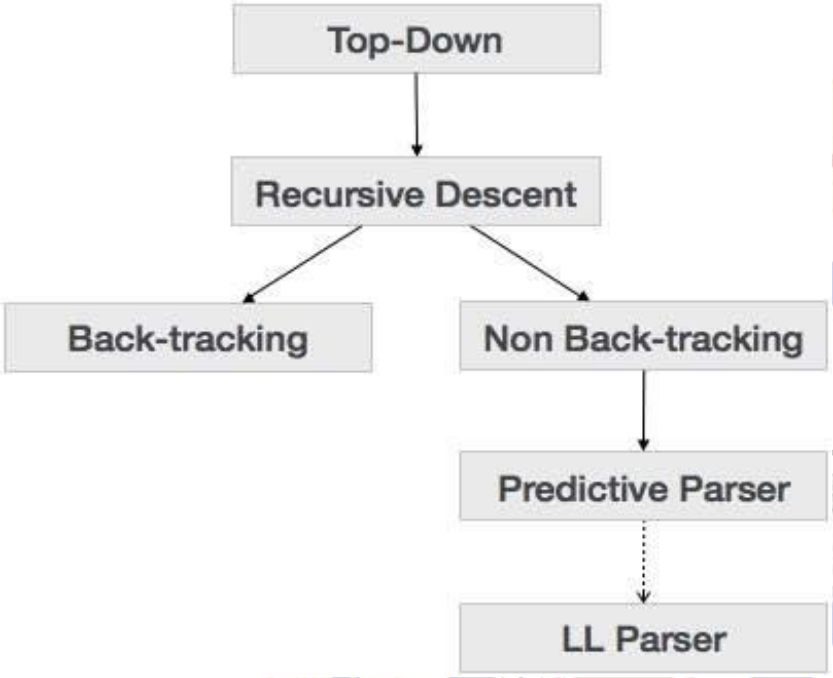
1. What is a operator precedence parser?
2. List the advantages and disadvantages of operator precedence parsing.
3. List the properties of LR parser.
4. What is phrase level error recovery?
5. Define a context free grammar.
6. Briefly explain the concept of derivation.

EXERCISE 4

AIM: Construct a recursive descent parser for an expression.

DESCRIPTION:

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes.



Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Back-tracking:

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).To understand this, take the following example of CFG.

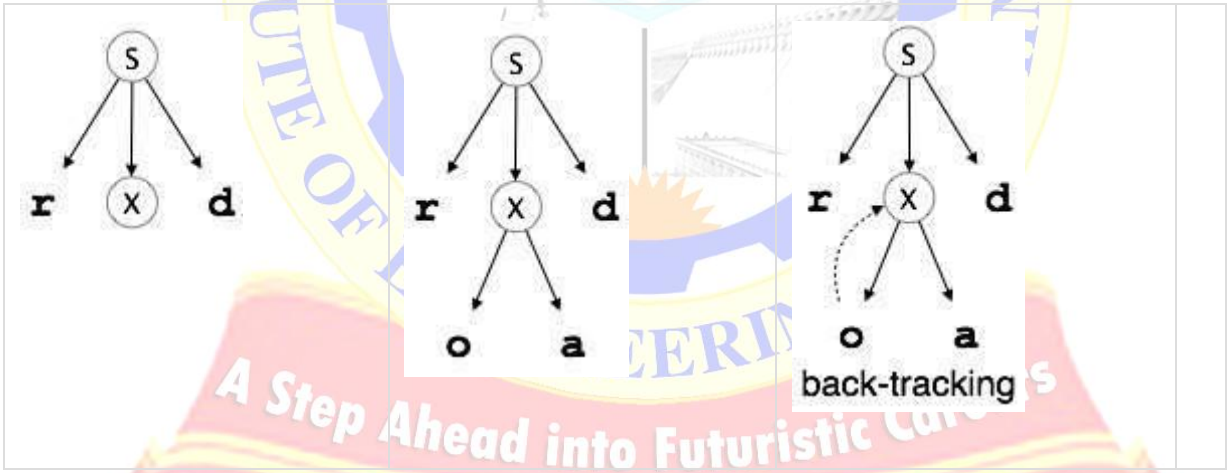
S → rXd | rZd

X → oa | ea

Z → ai

For an input string: read, a top-down parser will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).Now the parser matches all the input letters in an ordered manner. The string is accepted.



For factoring:

Give a set of productions of the form

A->αβ1|αβ2|…|αβn

Invent a new nonterminal Z and replace this set by the collection:

A->αZ

Z->β1|β2|…βn

For substitution:

If we are given a grammar containing A->Bα and if all of the productions with B on the left side are:

B->β1|β2|…βnα

ALGORITHM:

Step 1: start.

Step 2: Declare the prototype functions E() , EP(),T(), TP(),F()

Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals

Step 6: Read the parse string.

Step 7: stop the production

OUTPUT:

Grammer without left recursion

E ->TE'

E'->+TE'|ϵ

T->FT'

T'->*FT'|ϵ

F-> (E)|i

Enter the input expression:i+i*i

Expressions	**Sequence of production rules**
E=TE'	E->TE'
E=FT'E'	T->FT'
E=iT'E'	F->i
E=iϵE'	T'->ϵ
E=i+TE'	E'->+TE'
E=i+FT'E'	T->FT'
E=i+iT'E'	F->i
E=i+i*FT'E'	T'->*FT'
E=i+i*iT'E'	F->i

E=i+i*iϵE’ T’->ϵ

E=i+i*iϵ E’->ϵ

E=i+i*i accepted

VIVA QUESTIONS:

1. What type of parsing is Recursive Descent Parser?
2. What is Recursive descent parser?
3. What is Recursive descent parsing?
4. What is Backtracking?
5. What is top-down parsing?
6. What is left recursion?
7. What is Left factoring?

EXERCISE 5

AIM: Construct a LL (1) parser for an expression.

DESCRIPTION:

Algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL (1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL (1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL (1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions. However, if an LR parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

The prime requirements are: -

- Stack
- Parsing Table
- Input buffer
- Parsing program.

ALGORITHM:

1. Compute FIRST(X) as follows:

- if X is a terminal, then FIRST(X)={X}
- if X→ε is a production, then add ε to FIRST(X)
- if X is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_n$ is a production, add $FIRST(Y_i)$ to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

2. Compute FOLLOW as follows:
   - FOLLOW(S) contains $, If S is the starting symbol of the grammar.
   - For productions A→αBβ, everything in FIRST(β) except ε goes into FOLLOW(B)
   - For productions A→αB or A→αBβ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)
3. For each production A→α do:
   - For each terminal a ∈ FIRST(α) add A→α to entry M[A,a]
   - If ε∈FIRST(α), add A→α to entry M[A,b] for each terminal b ∈ FOLLOW(A).
   - If ε∈FIRST(α) and $∈FOLLOW(A), add A→α to M[A,$]
4. If there is two or more number of productions in any cell then grammar is not LL (1).

OUTPUT:

Grammar is:

S→ iEtSS' | a

S'→eS|ϵ

E→b

FIRST(S) = {i,a}

FIRST(S') = {e,ϵ)

FIRST (E) = {b}

FOLLOW(S) = {e, $}

FOLLOW(S') = {e, $}

FOLLOW (E) = {t}

The predictive parsing table for above grammar is

PREDICTIVE PARSING TABLE:

	i	t	a	e	b	$
S	S→iEtSS'		S→ a			
S'				S'→eS S'→ϵ		S'→ϵ
E					E→b	

Given grammar is not LL(1)

VIVA QUESTIONS:

1. What is Predictive parser?
2. What is LL (1)?
3. Which type of parsing is LL (1) parser?
4. What is meaning of L and L in LL (1)
5. How we will say any grammar is LL (1) or not
6. What is the meaning of 1 in LL (1)?

EXERCISE 6

AIM: Design predictive parser for the given language.

DESCRIPTION:

The main idea of top-down parsing is :

- Start at the root, grow towards leaves
- Pick a production and try to match input
- May need to backtrack
- Our parser scans the input Left-to-right, generates a leftmost derivation and uses 1 symbol of lookahead.
- It is called an LL (1) parser.
- If you can build a parsing table with no multiply-defined entries, then the grammar is LL (1).
- Ambiguous grammars are never LL(1)
- Non-ambiguous grammars are not necessarily LL(1)

To design predictive parsing table, first we have to check, weather there is left recursion or not? If there is left recursion we have to eliminate that left recursion.

ELIMINATION OF LEFT RECURSION:

If grammar is $A \rightarrow A\alpha_1|A\alpha_2|A\alpha_3|\ldots\ldots\ldots|\beta_1|\beta_2|\beta_3|\ldots..|\beta_n$ Then after elimination of left recursion grammar is

$A \rightarrow \beta_1 A^{|}| \beta_2 A^{|}|\beta_3 A^{|}|\ldots..|\beta_n A^{|}$

$A^{|} \rightarrow \alpha_1 A^{|}|\alpha_2 A^{|}|\alpha_3 A^{|}|\ldots\ldots\ldots|\alpha_n A^{|}|$ €.

Then we have to calculate FIRST and FOLLOW set, then we have to design predictive parsing table.

ALGORITHM:

1. Compute FIRST(X) as follows:

- if X is a terminal, then FIRST(X)={X}
- if $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)
- if X is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_n$ is a production, add $FIRST(Y_i)$ to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

2. Compute FOLLOW(S) as follows:

- FOLLOW(S) contains $, If S is the starting symbol of the grammar.
- For productions A→αBβ, everything in FIRST(β) except ε goes into FOLLOW(B)
- For productions A→αB or A→αBβ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

3.For each production A→α do:

- For each terminal a ∈ FIRST(α) add A→α to entry M[A,a]
- If ε∈FIRST(α), add A→α to entry M[A,b] for each terminal b ∈ FOLLOW(A).
- If ε∈FIRST(α) and $∈FOLLOW(A), add A→α to M[A,$]

OUTPUT:

Enter the no. of co-ordinates

2

Enter the productions in a grammar

S->CC

C->eC | d

First set:

FIRST[S] = {e, d}

FIRST[C] = {e, d}

Follow set:

FOLLOW[S] = {$}

FOLLOW[C] = {e, d, $}

Parsing Entries:

M [S, e] =S->CC

M [S, d] =S->CC

M [C, e] =C->eC

M [C, d] =C->d

VIVA QUESTIONS:

1. What is Predictive parsing?
2. What is ambiguous grammar?
3. What is left most derivation?
4. What is right most derivation?
5. What is backtracking?
6. Explain rules to calculate first set.
7. Explain rules to calculate follow set.
8. What are the problems with top down parsing?

EXERCISE 7

AIM: Implementation of shift-reduce parsing algorithm.

DESCRIPTION:

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the non-terminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

Action Table: The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state *s* and the current lookahead terminal is *t*, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift *s'*

Shift state s' onto the parse stack.

Reduce *r*

Reduce by rule *r*. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error.

ALGORITHM:

STEP1: Initial State: the stack consists of the single state, s0; ip points to the first character in w.

STEP 2: For top-of-stack symbol, s, and next input symbol, a case action of T[s,a]

STEP 3: Shift x: (x is a STATE number) push a, then x on the top of the stack and Advance ip to point to the next input symbol.

STEP 4: Reduce y: (y is a PRODUCTION number) Assume that the production is of the form A ==> beta pop 2 * |beta| symbols of the stack.

STEP 5: At this point the top of the stack should be a state number, say s'. push A, then goto of T[s',A] (a state number) on the top of the stack.

OUTPUT:

GRAMMAR IS:

E→E+E

E→E/E

E→E*E

E→ a/b

Enter the input symbol: a+a

Stack implementation table:

STACK	INPUT SYMBOL	ACTION
$	a+a$	--
$a	+a$	shift a
$E	+a$	E□ a
$E+	a$	shift +
$E+a	$	shift a
$E+E	$	E□ a
$E	$	E→E*E
$E	$	ACCEPT

VIVA QUESTIONS

1. What type parser is SR parser?
2. What is bottom-up parser?
3. What is SHIFT?
4. What is REDUCE?
5. What is ACCEPT?
6. List the properties of LR parser.
7. Mention the types of LR parser.
8. Define LR(0) items.
9. Define handle.

EXERCISE 8

AIM: Implementation of LALR parsing table

DESCRIPTION:

The LALR (Look Ahead-LR) parsing technique is between SLR and Canonical LR, both in terms of power of parsing grammars and ease of implementation. This method is often used in practice because the tables obtained by it are considerably smaller than the Canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar. The same is almost true for SLR grammars, but there are a few constructs that cannot be handled by SLR techniques.

A core is a set of LR (0) (SLR) items for the grammar, and an LR (1) (Canonical LR) grammar may produce more than two sets of items with the same core. The core does not contain any look ahead information.

Example:

Let s1 and s2 are two states in a Canonical LR grammar.

S1 – {C ->c.C, c/d; C -> .cC, c/d; C -> .d, c/d}

S2 – {C ->c.C, $; C -> .cC, $; C -> .d, $}

These two states have the same core consisting of only the production rules without any look ahead information.

CONSTRUCTION IDEA:

1. Construct the set of LR (1) items.
2. Merge the sets with common core together as one set, if no conflict (shift-shift or shift-reduce) arises.
3. If a conflict arises it implies that the grammar is not LALR.
4. The parsing table is constructed from the collection of merged sets of items using the same algorithm for LR (1) parsing.

ALGORITHM:

Creating new item sets;

1. For each terminal and non-terminal symbol A appearing after a '•' in each already existing item set k, create a new item set m by adding to m all the rules of k where '•' is followed by A, but only if m will not be the same as an already existing item set after step 3.

2. Shift all the '•'s for each rule in the new item set one symbol to the right
3. Create the closure of the new item set
4. Repeat from step 1 for all newly created item sets, until no more new sets appear

Algorithm to design CLR parsing table:

1. Construct C= {I_0,I_1……..,I_n}, the collection of sets of LR(1) items for G'.
2. State I of the parser is constructed from I_i. The parsing actions for state I are determined as follows :

   a) If [A → α. a β, b] is in I_i, and goto (Ii, a) = Ij, then set action [i,a] to "shift j." Here, a is required to be a terminal.

   b) If [A → α., a] is in I_i, A ≠ S', then set action[i,a] to "reduce A → α."

   c) If [S'→S.,$] is in I_i, then set action[i ,$] to "accept."
3. The goto transition for state i are determined as follows: If goto(I_i , A)= I_j ,then goto[i,A]=j.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set containing item [S'→.S, $].

Algorithm to design LALR parsing table:

1. Construct C= {I0, I1, I2… In}, the collection of sets of LR(1) items.
2. For each core present in among the set of LR (1) items , find all sets having the core, and replace these sets by their union.
3. Parsing action table is constructed as for Canonical LR.
4. The goto table is constructed by taking the union of all sets of items having the same core. If J is the union of one or more sets of LR (1) items, that is, J=I1 U I2 U … U Ik, then the cores of goto(I1,X), goto(I2,X),…, goto(Ik, X) are the same as all of them have same core. Let K be the union of all sets of items having same core as goto(I1, X). Then goto(J,X)=K.

INPUT:

GRAMMAR IS:

1. S' -> S
2. S -> CC
3. C -> cC
4. C -> d

OUTPUT:

Set of items in CLR parsing

I0 : S' -> .S, $
S -> .CC, $
C -> .c C, c /d
C -> .d, c /d

I1: S' -> S., $

I2: S -> C.C, $
C -> .Cc, $
C -> .d, $

I3: C -> c. C, c /d
C -> .Cc, c /d
C -> .d, c /d

I4: C -> d., c /d

I5: S -> CC., $

I6: C -> c.C, $
C -> .cC, $
C -> .d, $

I7: C -> d., $

I8: C -> cC., c /d

I9: C -> cC., $

CLR parsing table:

STATE	Actions			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	R4	R4			
5			R2		
6	S6	S7			9
7			R4		
8	R3	R3			
9			R3		

LALR parsing table:

START	ACTION			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
36	S36	S47			89
47	R47	R47	R47		
5			R2		
89	R36	R36	R36		

VIVA QUESTIONS:

1. What is the full name of LALR?
2. What is LALR parser?
3. What is parsing?
4. What is the meaning of L and L in LL (1).
6. What is the meaning of 1 in LL (1)?
7. How we can merge the items in LALR parsing table?
8. What is the difference between top down and bottom up parsing?
9. What is the more powerful parser

EXERCISE 9

AIM: Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools

DESCRIPTION:

The flex input file consists of three sections, separated by a line with just %% in it:

Definition section
%%
Transition Rule section
%%
Auxilary procedure section

First section is definition Section.In definition section we have to declare all the header files and variables, what we are using in our program.definition section end with %%. Second section is translation rules section.here we have to represent patterns and associated actions.rules section also end with %%. Third section is user code. Here we have to write the 'C' code to print the results as output.

DIGIT [0-9]
ID [a-z][a-z0-9]*
letter = a|…|z|A|…|Z
digit = 0|…|9
'\t' tab
\b' backspace
'\f' form feed

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example, Defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to DIGIT is

{DIGIT}+"."{DIGIT}* is identical to **([0-9])+"."([0-9])***

COMMANDS TO RUN THE CODE:

```
[root@localhost ~]# lex prob2.l
[root@localhost ~]# cc lex.yy.c -o prob2 -ll
[root@localhost ~]#. /prob2
```

OUT PUT:

44

Number

Hai

Word

*

Operator

@

Special character

VIVA QUESTIONS:

1. LEX programming consisting of how many sections?

2. What is the role of definition section?

3. What is the role of translation rules section?

4. What is regular expression?

5. What is the symbol to separate each section in lex program?

6. Write regular expression for identifier.

7. What is lex?

8. Define compiler-compiler.

EXERCISE 10

AIM: Write a program to perform loop unrolling.

DESCRIPTION:

Loop unrolling transforms a loop into a sequence of statements. It is a parallelizing and optimizing compiler technique where loop unrolling us used to eliminate loop overhead to test loop control flow such as loop index values and termination conditions technique was also used to expose instruction-level parallelism.

Example:

In the code fragment below, the body of the loop can be replicated once and the number of iterations can be reduced from 100 to 50.

```
for (i = 0; i < 100; i++)
  g ();
```

Below is the code fragment after loop unrolling?

```
for (i = 0; i < 100; i += 2)
{
  g ();
  g ();
}
```

ALGORITHM:

Step1: start

Step2: declare n

Step3: enter n value

Step4: when i<n do

Step5: write for loop block

Step6: increment i value

Step7: goto step 4

INPUT:

```
 for (i=0; i<10; i++)
{
a=b+c;
```

}

OUTPUT:

Enter unrolling factor: 2

for (i=0; i<10; i=i+2)

{

a=b+c;

a=b+c;

}

VIVA QUESTIONS:

1. Define loop unrolling?
2. What are the advantages of loop unrolling?
3. What is unrolling factor?
4. What is the importance of loop?
5. What are the phases of code improvement phase?
6. What are the basic goals of code movement?
7. What are the different data flow properties?

EXERCISE 11

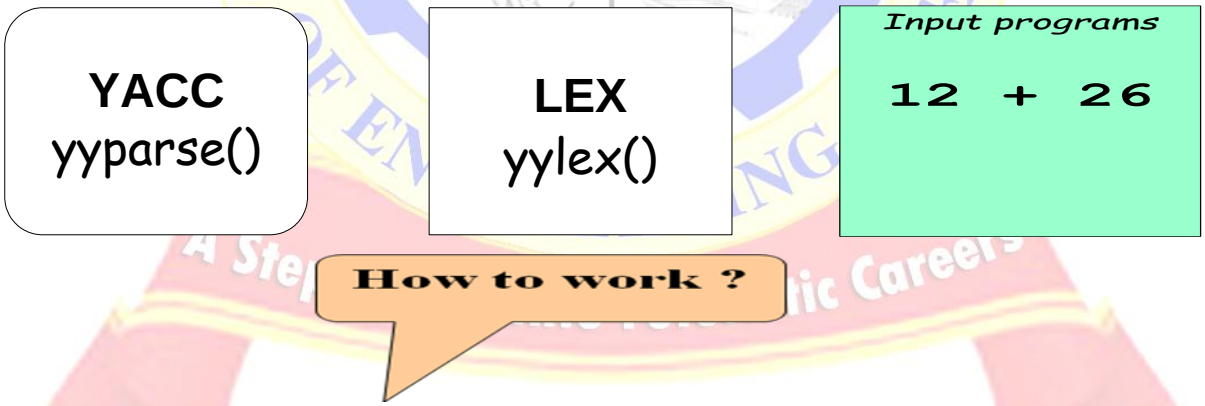
AIM: Program to implement BCNF to YACC conversion.

DESCRIPTION:

Yacc is a computer program for the UNIX operating system. The name is an acronym for "Yet another Compiler Compiler. It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. It was originally developed in the early 1970s by Stephen C. Johnson at AT&T Corporation.

Yacc and similar programs (largely reimplementations) have been very popular. Yacc itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs such as Berkeley Yacc, GNU bison, MKS Yacc and Abraxas PCYACC. Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.Lexical analyzer generators

LEX and YACC: a team



ALGORITHM:

Step 1: start

Step 2: %%

Step 3: declare identifier [a-zA-Z][a-zA-Z0=9]*

Step 4: numbers [0-9]+|([0-9]*\.[0-9]+)

Step 5: %%

Step 6: {initialize structure and structure name and integer array variables Item[100],top }stk

Step 7: initialize integer variables 0, StNo,Ind,tInd⇓0, tindex ⇓Index Extern int LineNo

Step 8: %tokenNUM VAR RELOP %%token MAIN IF ELSE WHILE TYPE

Step 9: %type EXPR ASSIGNMENT CONDITION IFST ELSEST WHILE LOOP %left _-_ _+‘

%left _*‘‘/‘

Step 10: %%

Step 11: identifier { strcpy(yylval.var, yytext) return VAR }

Step 12: number { strcpy(yylval.var, yytext) return NUM }

Step 13: %% \< | \> | \>= | \<= | == { strcpy(yylval.var, yytext) return RELOP }

Step14: %%

Step15: tab space [\t]

Step16: new line \n LineNo++

Step17: return yytext[0]

Step18: %%

Step 19: End

OUTPUT:

$lex int.l

$yacc –d int.y

$gcc lex.yy.c y.tab.c –ll –lm

$./a.out test.c

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	to
1	==	to	FALSE	5
2	+	a	b	t1
3	=	t1		a
4	GOTO			5
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t3
16	=	t6		c

VIVA QUESTIONS:

1. Explain LEX.
2. Explain YACC.
3. What is the use of YACC .
4. What is the full form of YACC
5. Why we have to include ‘y.tab.h’ in LEX?
6. Explain features of UNIX?
7. What is BCNF?

EXERCISE 12

AIM: Write a program for constant propagation.

DESCRIPTION:

The algorithm we shall present basically tries to find for every statement in the program a mapping between variables, and values of N T ∪⊥ {, } . If a variable is mapped to a constant number, that number is the variables value in that statement on every execution. If a variable is mapped to T (top), its value in the statement is not known to be constant, and in the variable is mapped to ⊥ (bottom), its value is not initialized on every execution, or the statement is unreachable. The algorithm for assigning the mappings to the statements is an iterative algorithm that traverses the control flow graph of the algorithm, and updates each mapping according to the mapping of the previous statement, and the functionality of the statement. The traversal is iterative, because non-trivial programs have circles in their control flow graphs, and it ends when a "fixed-point" is reached – i.e., further iterations don't change the mappings.

ALGORITHM:

Step1: start

Step2: declare a=30, b=3,c

Step3: identify constant values assigned to variables

Step4: if variable is not changed then

Step5: replace the variable with constant value in source code

Step 6: End

INPUT:

```
int a,b=5,c;
for (i=0;i<10;i++)
{
    a=b+c;
}
```

OUTPUT:

```
int a,b=5,c;
for (i=0;i<10;i++)
{
        a=5+c;
}
```

VIVA QUESTIONS:

1. What is constant propagation?
2. What is code motion?
3. What are the properties of optimizing compiler?
4. What is meant by Dead code? Give an example.
5. Mention the issues to be considered while applying the techniques for code optimization.
6. Define code optimization.
7. List out the criteria for code improving transformations.
8. What are the basic goals of code movement?

EXERCISE 1

AIM: Write a program to find out whether given string is identifier or not

DESCRIPTION: Identifier refers to name given to entities such as variables, functions, structures etc. Identifier must be unique. They are created to give unique name to an entity to identify it during the execution of the program. For example:

int money;

double accountBalance;

Here, **money** and **accountBalance** are identifiers.

Also remember, identifier names must be different from keywords. You cannot use **int** as an identifier because **int** is a keyword.

Rules for writing an identifier

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore. However, it is discouraged to start an identifier name with an underscore.
3. There is no rule on length of an identifier. However, the first 31 characters of identifiers are discriminated by the compiler.

ALGORITHM:

Step1: Start

Step2: Check if the first char is a letter goto step 3 with rest of the string else goto 5.

Step3: Check if the first character of the given string is a letter or a digit repeat step 3 with rest of the string else goto step 5.

Step4: print "The given string is an identifier" and goto step 6.

Step5: Print "The given string is not an identifier".

Step6: Exit

INPUT:

Enter the desired String: a123

OUTPUT:

The given string is an identifier

VIVA QUESTIONS:

1) What is an identifier?
2) What are the rules followed for constructing an identifier?
3) How do you declare a variable that will hold string values?
4) Can the "if" function be used in comparing strings?
5) What are the different identifiers in C language?
6) What is the length of the int variable?
7) What is local variable?
8) What is global variable?
9) What is keyword?
10) What is assignment?

EXERCISE 2

AIM: Write a program to check whether a string belongs to a grammar or not

DESCRIPTION: In parsing we have to check whether the string belongs to the grammar or not. For this we have try to derive the string from the given grammar. By taking first production of the grammar we have to start the derivation process. By replacing non terminals with right side part of the arrow from the relevant production of the grammar we have to continue. Finally if we derive the string then string belongs to the grammar. If string is not derived in any possible way then we have to announce the string is not belongs to the grammar.

ALGORITHM:

Step1: Start.

Step2: Declare two character arrays str[],token[] and initialize integer variables a=0, b=0, c, d. Input the string from the user.

Step3: Repeat steps 5 to 12 till str[a] =’\0’.

→ If str[a] =='(' or str[a] =='{' then token[b] =’4’, b++.

→If str[a] ==')' or str[a] ==’}’ then token[b] =’5’, b++.

Step4: Check if isdigit (str[a]) then repeat steps 8 till is digit (str[a]) a++.

Step5: a--, token[b] =’6’, b++.

Step6: If(str[a]=='+’) then token[b]='2',b++.

Step7: If(str[a]=='*') then token[b]=’3’,b++.

→ a++

token[b]='\0';

then print the token generated for the string .

Step8: b=0.

Repeat step 22 to 31 till token[b]!='\0'

Step9: c=0.

Repeat step 24 to 30 till (token[b]=='6' and token[b+1]=='2' and token[b+2]=='6') or (token[b]=='6' and token[b+1]=='3'and token[b+2]=='6') or (token[b]=='4' and token[b+1]=='6' and token[b+2]=='5') or (token[c]!='\0').

token[c]='6';

c++;

step10: Repeat step 27 to 28 till token[c]!='\0'.

token[c]=token[c+2].c++.

Step11:token[c-2]='\0'. print token. b++.

Step12: Compare token with 6 and store the result in d.

Step13: If d=0 then print that the string is in the grammar.

→ Else print that the string is not in the grammar.

Step14: Stop.

INPUT:

Enter the grammar

A->BC

B->dd

C->Be

Enter the string

dddde

OUTPUT:

String belongs to the grammar.

VIVA QUESTIONS:

1. What is grammar?
2. What is production?
3. What is terminal?
4. What is nonterminal?
5. What is string?
6. What is top down parsing?
7. What is bottom up parsing?
8. What is parsing?
9. What is derivation?
10. What is the difference between left most and right most derivation?


LENDI
INSTITUTE OF ENGINEERING & TECHNOLOGY
LIET
A Step Ahead into Futuristic Careers