

- 17)
- 1) Construct CLR parsing table from  
 $S \rightarrow AA$   
 $A \rightarrow Aa|b$
  - 2) Explain code motion, copy propagation, dead code elimination.

$$S \rightarrow AA$$
$$A \rightarrow aA/b$$
$$S' \rightarrow \emptyset$$
$$S \rightarrow AA$$
$$A \rightarrow aA/b$$


CLR

	a	b	\$	S	A
I <sub>0</sub>	S <sub>3</sub>	S <sub>4</sub>			2
I <sub>1</sub>			accept		
I <sub>2</sub>	S <sub>6</sub>	S <sub>7</sub>			5
I <sub>3</sub>	S <sub>3</sub>	S <sub>4</sub>			8
I <sub>4</sub>	R <sub>3</sub>	R <sub>3</sub>			
I <sub>5</sub>			R <sub>1</sub>		
I <sub>6</sub>					9
I <sub>7</sub>					
I <sub>8</sub>	R <sub>2</sub>	R <sub>2</sub>			
I <sub>9</sub>			R <sub>2</sub>		

LALR

	a	b	\$	S	A
I <sub>0</sub>	S <sub>36</sub>	S <sub>47</sub>			2
I <sub>1</sub>			accept		
I <sub>2</sub>	S <sub>36</sub>	S <sub>47</sub>			5
I <sub>36</sub>	S <sub>36</sub>	S <sub>47</sub>			89
I <sub>47</sub>	R <sub>3</sub>	R <sub>3</sub>			
I <sub>5</sub>		R <sub>1</sub>			
I <sub>6</sub>					
I <sub>89</sub>	R <sub>2</sub>	R <sub>2</sub>	R <sub>2</sub>		

## Code Motion:

- \* It reduces the evaluation frequency of expression
- \* It brings loop invariant statements out of the loop.

eg

```
a = 200
while (a > 0)
{
    b = x + y;
    if (a % b == 0)
        print(a)
}
```



```
a = 200
b = x + y
while (a > 0)
{
    if (a % b == 0)
        print(a)
}
```

Here, in this example the expression  $b = x + y$  is the loop invariant. 'b' is going to gain the same value throughout the loop. So, we place it out of loop.

Now, the expression  $b = x + y$  is evaluated only once before entering the loop. The frequency of evaluation of that expression is optimized.

- \* It is used to decrease the amount of code in loop.
- \* This transformation takes a statement of expression which can be moved outside the loop body without affecting the semantics of the program.



## variable propagation

### Before optimization

$c = a * b$

$x = a$

till

$d = x * b + 4$

### After optimization

$c = a * b$

$x = a$

till

$d = a * b + 4$

- \* Instead of taking reference from the variables, we directly use the variable 'a'.
- \* Instead of substitution, direct value is taken.
- \* Here 'x' and 'a' are identified as common sub expressions.

## Dead code Elimination

### Before elimination

$c = a * b$

$x = b$

till

$d = a * b + 4$

### After elimination

$c = a * b$

till

$d = a * b + 4$

- \* The useless expressions are removed to reduce the no. of lines of code and space.
- \* prevents compiler on evaluating the expressions which are no use to the process or to the final result.
- \* Before elimination, the expression  $x = b$  is called Dead state

9)

- 1) Explain the various issues in the design of code generation.
- 2) Explain code generation phase with simple code generation algorithm

9)

## 1) ISSUES IN CODE GENERATION:

- \* Input to code generators
- \* Target program
- \* Memory management
- \* Instruction selection
- \* Register Allocation
- \* Evaluation order
- \* Approaches to code generation.



## Input to code generators

It is the immediate code generated by the front end, along with information in the symbol table that determines the run time address of data object.

## Target program:

It is the output of the code generation. Absolute machine language as output that it can be fixed in memory location and can be immediately executed. Relocation to the machine language as an output allows sub program & subroutine to be compiled separately. Assembly language as output makes the code generation easier.

## Memory management:

mapping the names in the source program to the address of data objects is done by the front end and the code generator.

## Instruction selection:

selecting the best instruction will improve the efficiency of program. It includes the instruction that should be complete and uniform & it plays major role when efficiency is considered.

## Register Allocation:

Use of register make the computer faster in the comparison to that memory. During register allocation, we select only these set of variables that will reside in the registers. During a subsequent register assignment, the specific register is picked to variables.



### Evaluation order:

The code generator decide the order in which the instruction will be executed. The order of computation affects the efficiency.

### Approaches to code generation

It always generates the code of correct. It is essential because of the no. of special case that a generator might face.

### 2) CODE GENERATION

- \* The following actions are  $x = y \text{ OP } z$ 
  - \* Invoke a function get reg to determine the location  $L$ , where the result of the computation  $y \text{ OP } z$  should be stored.
  - \* Construct the address destructor for  $y$  to determine  $y'$ , the current location of  $y$  prefer the register for  $y'$  if the value of  $y$  is currently both in memory & register.
  - \* Generate the instruction  $\text{OP } z'$ , where  $z'$  is a currently location of  $z$ . After execution of  $x = y \text{ OP } z$ , it will no longer contain  $y$  or  $z$ .

35)

1) Consider the grammar

 $S \rightarrow (L) | a$  $L \rightarrow L, S | S$ 

- f) What are the terminal, non-terminal and start symbol?
- g) Find parse tree for the following sentences
- (iv)  $(a, a)$
- (v)  $(a, (a, a))$
- (vi)  $(a, ((a, a), (a, a)))$

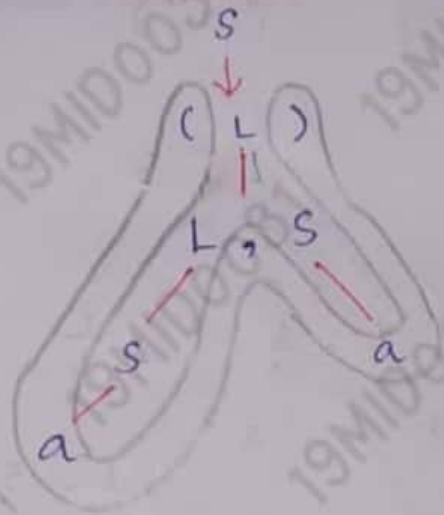
ss)  $S \rightarrow (L) | a$   
 $L \rightarrow L, S | S$

\* terminals =  $\{ (, ), ,, a \}$

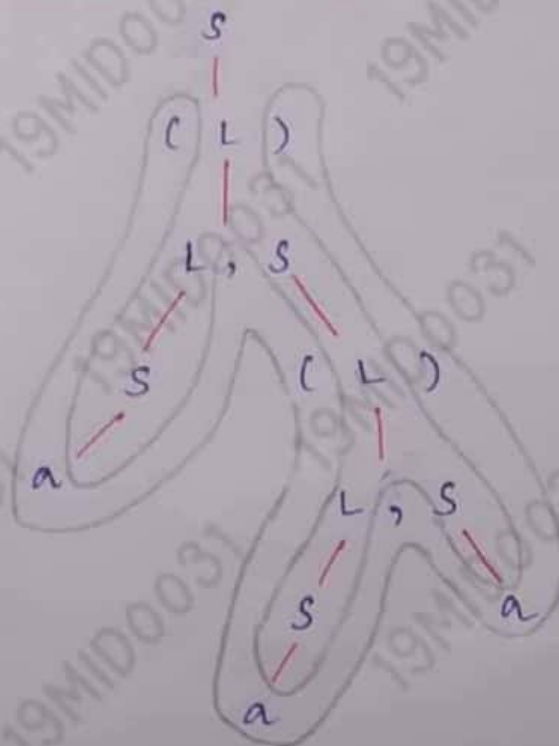
\* Non-terminals =  $\{ S, L \}$

\* Start symbol =  $\{ S \}$

\* parse tree for  $(a, a)$

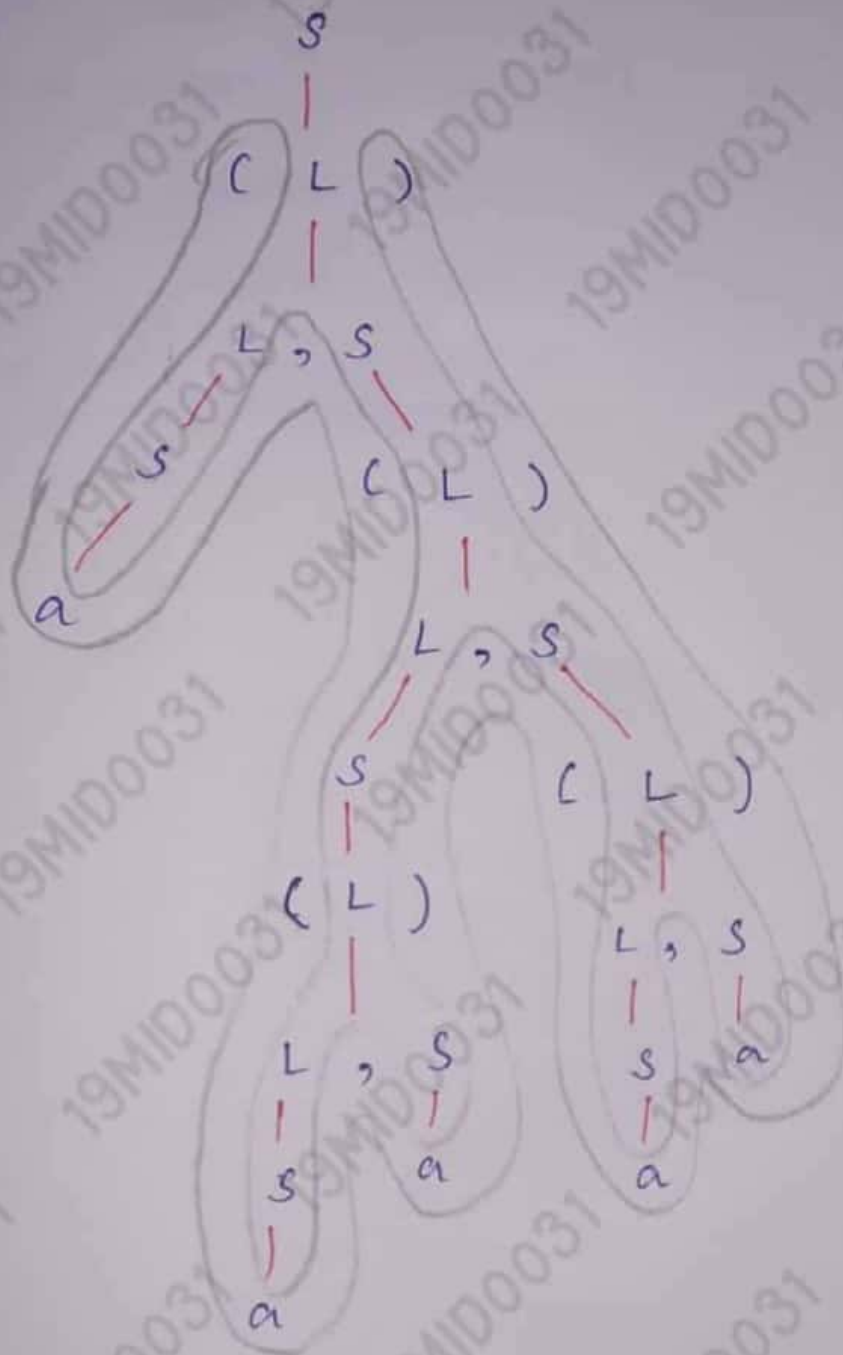


\* parse tree for  $(a, (a, a))$





\* parse tree for  $(a, ((a, a), (a, a)))$



4)

1. Explain & example following code optimization

- a) Common sub expression elimination
- b) Copy propagation
- c) Dead code elimination
- d) Code motion

2. Eliminate left recursion, perform left factoring and find:  
FIRST & FOLLOW

$E \rightarrow E+T \mid T$

$T \rightarrow \text{id} \mid \text{id} [ \ ] \mid \text{id} [ X ]$

$X \rightarrow E, E \mid E$

## variable propagation

Before optimization

$c = a * b$

$x = a$

till

$d = x * b + 4$

After optimization

$c = a * b$

$x = a$

till

$d = a * b + 4$

- \* Instead of taking reference from the variables, we directly use the variable 'a'.
- \* Instead of substitution, direct value is taken.
- \* Here 'x' and 'a' are identified as common sub expressions.

## Dead code Elimination

Before elimination

$c = a * b$

$x = b \rightarrow$  dead state

till

$d = a * b + 4$

After elimination

$c = a * b$

till

$d = a * b + 4$

- \* The useless expressions are removed to reduce the no. of lines of code and space.
- \* prevents compiler on evaluating the expressions which are no use to the process or to the final result.
- \* Before elimination, the expression  $x = b$  is called Dead state



Code Motion:

- \* It reduces the evaluation frequency of expression
- \* It brings loop invariant statements out of the loop.

eg

```

a = 200
while (a > 0)
{
    b = x + y;
    if (a % b == 0)
        print(a)
}

```



```

a = 200
b = x + y
while (a > 0)
{
    if (a % b == 0)
        print(a)
}

```

Here, in this example the expression  $b = x + y$  is the loop invariant. 'b' is going to gain the same value throughout the loop. So, we place it out of loop.

Now, the expression  $b = x + y$  is evaluated only once before entering the loop. The frequency of evaluation of that expression is optimized.

- \* It is used to decrease the amount of code in loop.
- \* This transformation takes a statement of expression which can be moved outside the loop body without affecting the semantics of the program.

2)

1) Find the following grammar is LL(1), LR(1)

 $S \rightarrow \underline{AaAb} \mid \underline{BbBa}$ 

2) Check whether the following grammar is LR(0), SLR(1), LALR and LR(1)

 $S \rightarrow \underline{AaAb} \mid \underline{BbBa}$  $A \rightarrow \epsilon$  $B \rightarrow \epsilon$

$$\beta \rightarrow \varepsilon$$
$$s' \rightarrow s$$
$$S \rightarrow AaAb / BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow E$$

Handwritten LR(0) item sets and transitions for the grammar:

- Item Set 0 (Start):**
  - $S' \rightarrow \cdot S$
  - $S \rightarrow \cdot A A A b \mid \cdot B b B a$
  - $A \rightarrow \cdot \epsilon$
  - $B \rightarrow \cdot \epsilon$
- Transitions from Item Set 0:**
  - On **S**: Item Set 1 ( $S' \rightarrow S \cdot$ )
  - On **A**: Item Set 2 ( $S \rightarrow A \cdot A A b$ )
  - On **B**: Item Set 3 ( $S \rightarrow B \cdot b B a$ )
- Item Set 1:**
  - $S' \rightarrow S \cdot$
- Item Set 2:**
  - $S \rightarrow A \cdot A A b$
- Item Set 3:**
  - $S \rightarrow B \cdot b B a$
- Transitions from Item Set 2:**
  - On **a**: Item Set 4 ( $S \rightarrow A A \cdot A b$ )
- Transitions from Item Set 3:**
  - On **b**: Item Set 5 ( $S \rightarrow B b \cdot B a$ )
- Item Set 4:**
  - $S \rightarrow A A \cdot A b$
- Item Set 5:**
  - $S \rightarrow B b \cdot B a$
- Transitions from Item Set 4:**
  - On **a**: Item Set 6 ( $S \rightarrow A A A \cdot b$ )
- Transitions from Item Set 5:**
  - On **a**: Item Set 7 ( $S \rightarrow B b B \cdot a$ )
- Item Set 6:**
  - $S \rightarrow A A A \cdot b$
- Item Set 7:**
  - $S \rightarrow B b B \cdot a$

	ACTION			GOV O		
	a	b	\$	A	B	S
0				2	3	1
1			accept			
2	S4					
3		S5				
4				6		
5					7	
6		S8				
7	S9					
8	r1	r1	r1			
9	r2	r2	r2			

Therefore it is  $LR(0)$

\* If it is LRCO,  
definitely it is SLRC()



2)  $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$\text{FIRST}(S) = \{a, b\}$

$\text{FIRST}(A) = \{\epsilon\}$

$\text{FIRST}(B) = \{\epsilon\}$

$\text{FOLLOW}(S) = \{\$ \}$

$\text{FOLLOW}(A) = \{a, b\}$

$\text{FOLLOW}(B) = \{b, a\}$

Parsing table

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

No Repetitions  
 $\therefore \text{LL}(1)$

- 11) |
- 1) Discuss the role of finite automata in compiler.
  - 2) State what strategy LEX should adopt if keywords are not reserved words.

## 1) ROLE OF FINITE AUTOMATA

Finite automata is a state machine that takes a string as a symbol as input & changes its state accordingly & it is a recognizer for regular expression when a regular expression string is fed into finite automata.

The machine consists

$Q \rightarrow$  set of states

$\Sigma \rightarrow$  input symbols

$q \rightarrow$  starting state

$F \rightarrow$  final state

$\delta \rightarrow$  transition function.

2) RE	Matches	Example
$c$	single character not operator	$x$
$\backslash c$	Any character following	$\backslash *$
$"s"$	string $s$ literally	$"***"$
$\wedge$	Beginning of line	$\wedge abc$
$\$$	End of line	$abc\$$
$[s]$	any character in $s$	$[abs]$
$[^s]$	Any except character not $s$	$[^abc]$
$a^0$	zero	$a^*$
$a^1$	one	$a^+$
$a^?$	zero	$a^?$
$a\{m,n\}$	$m$ to $n$ occurrence of $a$	$a\{1,5\}$
$a_1 a_2$	$a_1$ then $a_2$	$ab$
$(a)$	$a$	$(a \cdot b)$
$a_1 / a_2$	$a_1$ when followed by $a_2$	$abc/123$
$a_1   a_2$	$a_1$ or $a_2$	$a \cdot b$