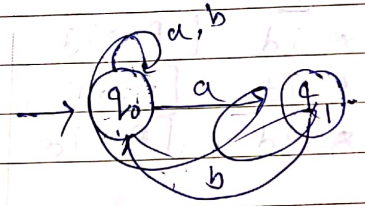
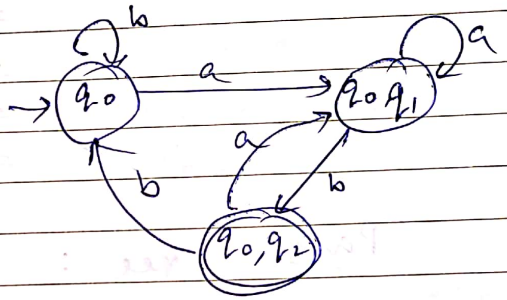
COMPILER DESIGN

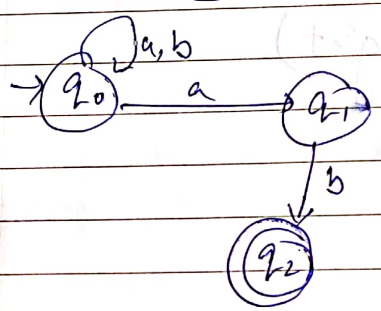
(18) NFA to DFA

State	a	b
→ q_0	q_0, q_1	q_0
q_1	ϕ	q_2
(*) q_2	ϕ	ϕ

State	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
(*) $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$







LL(1) parsing - top down.

(*) G → P = { E → E+T | T
T → T*F | F
F → (E) | id }

{ V, T, P, S }

string : id * id

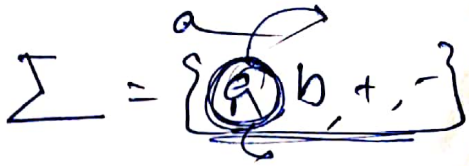
E → T
⇒ T * F [F → id]
⇒ T * id [T → F]
⇒ F * id [F → id]
⇒ id * id

Parse Tree : (Right)

Handle

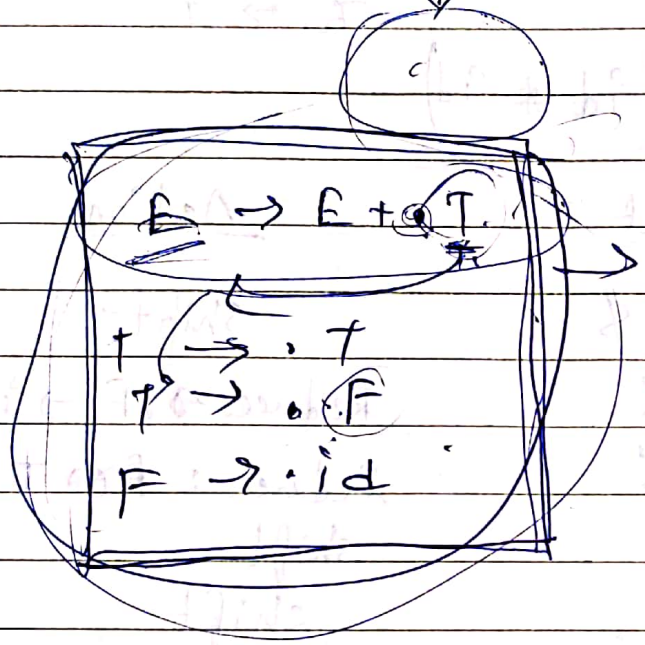
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E	(id * id)	

Stack	Input	Action
$	id * id $	shift
$ id	* id $	Reduce : F → id
$ F	* id $	Reduce : T → F
$ T	* id $	shift
$ T *	id $	shift
$ T * id	$	Reduce : F → id
$ T * F	$	Reduce : T → T * F
[illegible]	[illegible]	[illegible]
$ T	$	Reduce : E → T
$ E	$	stop + accept



LR Parsing

$E \to \cdot E + T$ —(E)→ $E \to E \cdot + T$



GOTO (I, x)

↓ ↓

state string

$E \to \cdot E + T$ GOTO (I_0, E)

I_0

$E \to E \cdot + T$ GOTO (I_2, +)

I_1

$E \to E + \cdot T$

$T \to \cdot T$

$T \to \cdot F$

$F \to \cdot id$

I_2

P → { E → E + T | T
T → T * F | F
F → id }

items → E → · E + T

E' → · E

E → · T

T → · T * F

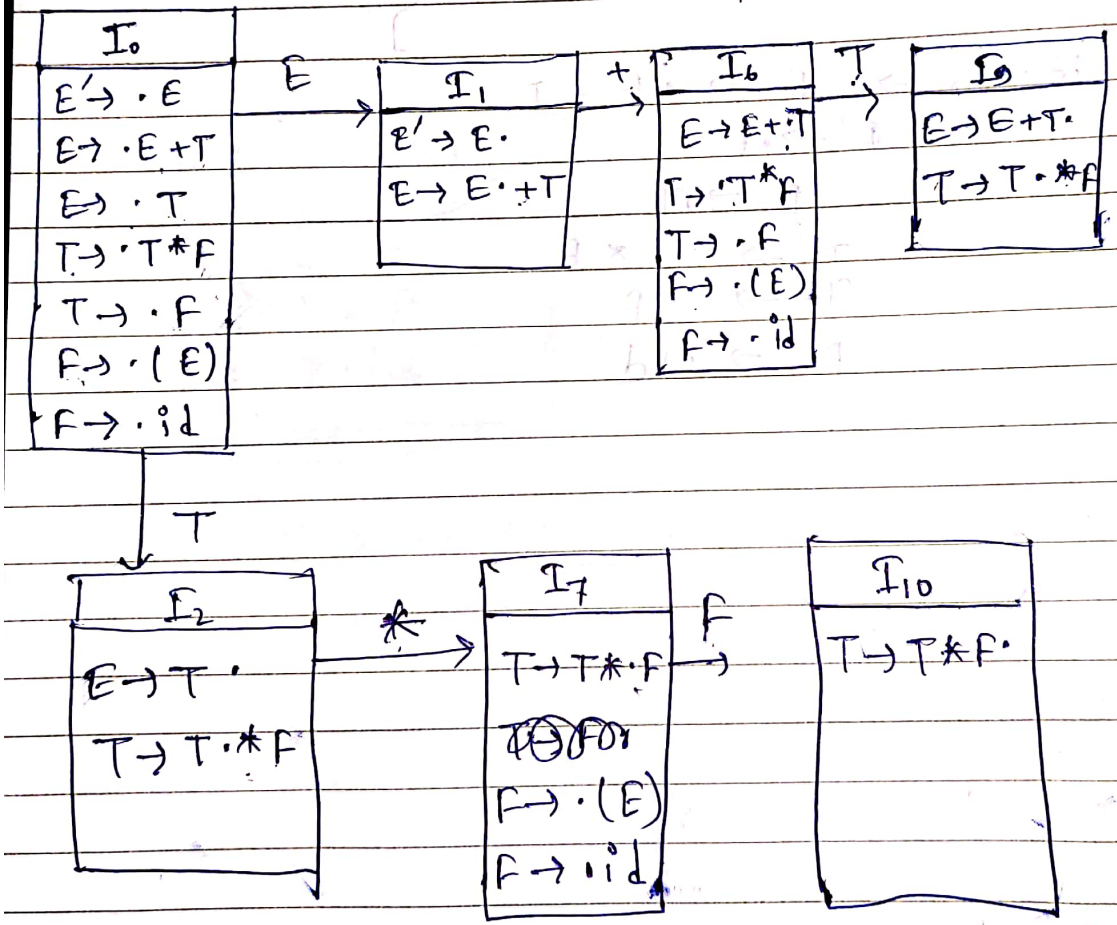
T → · F

F → · id

(12) $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

closure of items :-

$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$



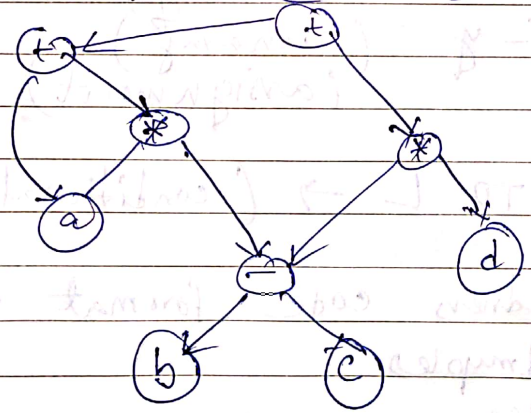
(*) Parsing in SLR automata

String (id * id)

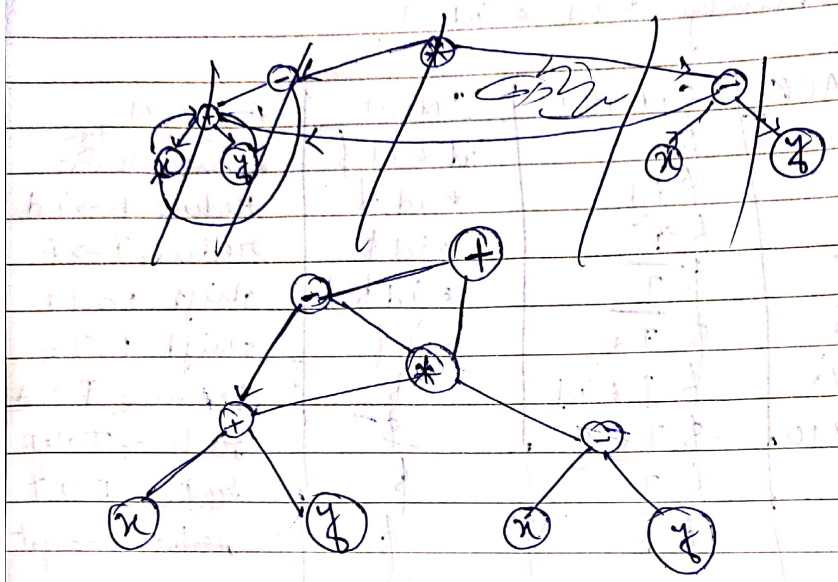
STACK	SYMBOL	INPUT	ACTION
0	$	id * id $	shift to 5
05	$ id	* id $	reduce F → id
03	$ F	* id $	reduce T → F
02	$ T	* id $	shift to 7
027	$ T *	id $	shift to 5
0275	$ T * id	$	reduce F → id
02710	$ T * F	$	reduce T → T * F
02	$ T	$	reduce E → T
01	$ E	$	accept

(*) DAG:

a + a * (b − c) + (b − c) * d



(ii) $((x+y) - ((x+y) * (x-y))) + (x+y) * (x-y)$



Types of Instruction :-

1. $x = y + z$ (binary)
2. $x = -y$ (unary)
3. $x = y$ (assignment)
4. if x GOTO L → (conditional)

* 3 - address code format -
  - Quadruples
  - Triples
  - Indirect Triples.

See C

8 (A) (a) $\Sigma = \{a, b\}$.

~~RE = (b* a b* a b*)*~~

(a) $RE = ((a+b)^* \cdot (a+b)^*)^*$

2 Length

{ ab, ba, bb, aa }

$RE = ab + ba + bb + aa$

+ → OR
. → AND
* → Kleen *

$b(a+b) + a(a+b)$

$= (a+b) \cdot (a+b)$

$RE \rightarrow ((a+b) \cdot (a+b))^*$

(b) $\Sigma = \{a, b\}$

$RE \rightarrow a(a+b)^* b + b(a+b)^* a$

(B) $S \rightarrow SS+ \mid SSk \mid a$

(a) String $(aa+ak)$

$S \rightarrow SSk$

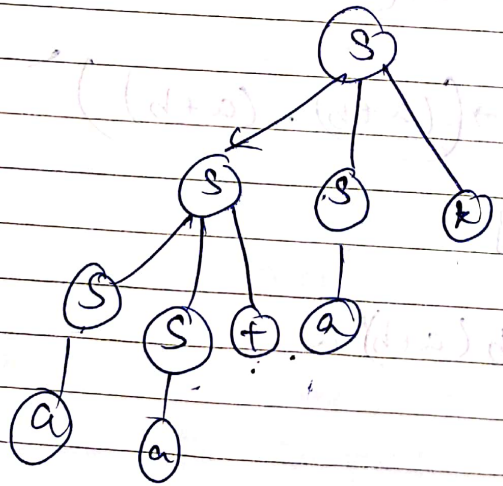
$\rightarrow SSSak$

$\rightarrow SS+ak$

$\rightarrow Sa+ak$

$\rightarrow aa+ak$

(b) Parse Tree



(9) position = initial + rate * 60;

(1) Lexical Analyzer

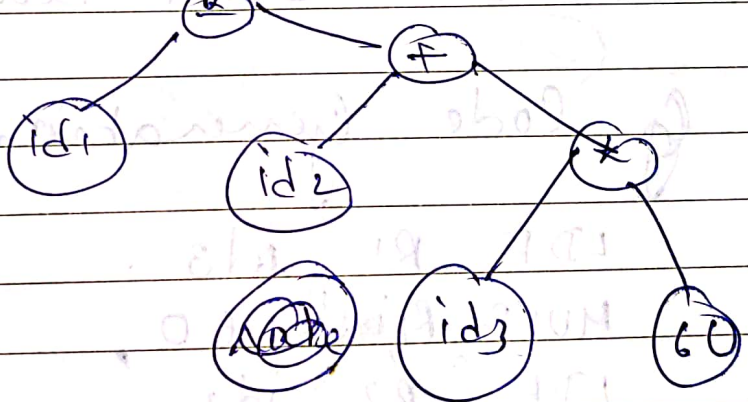
<id1> <=> <id2> <+> <id3 X *> <60>

<1>

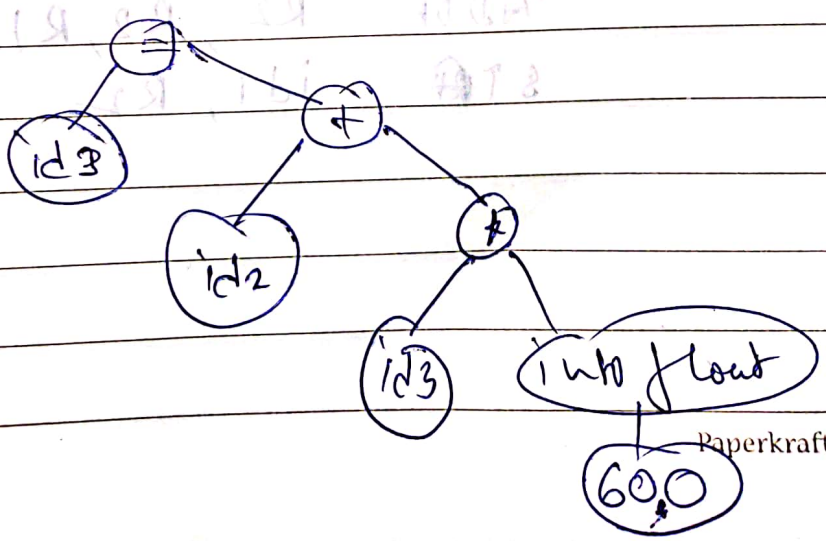
Symbol table

		type
1	position	float
2	init	float
3	rate	float

(2) Syntax Analyzer



(3) Symantic



C) Design a DFA that accepts all strings over Σ = {a}?

(4) Intermediate Cod Gen

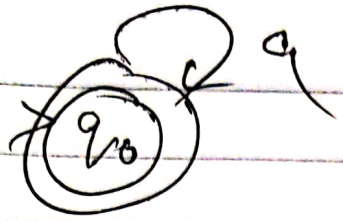
temp1 = (inttofloat) 60
temp2 = rate * temp1
temp3 = initial + temp2
position = temp3

(5) Code Optimization

temp1 = id3 * (inttofloat) 60
id1 = id2 + temp1

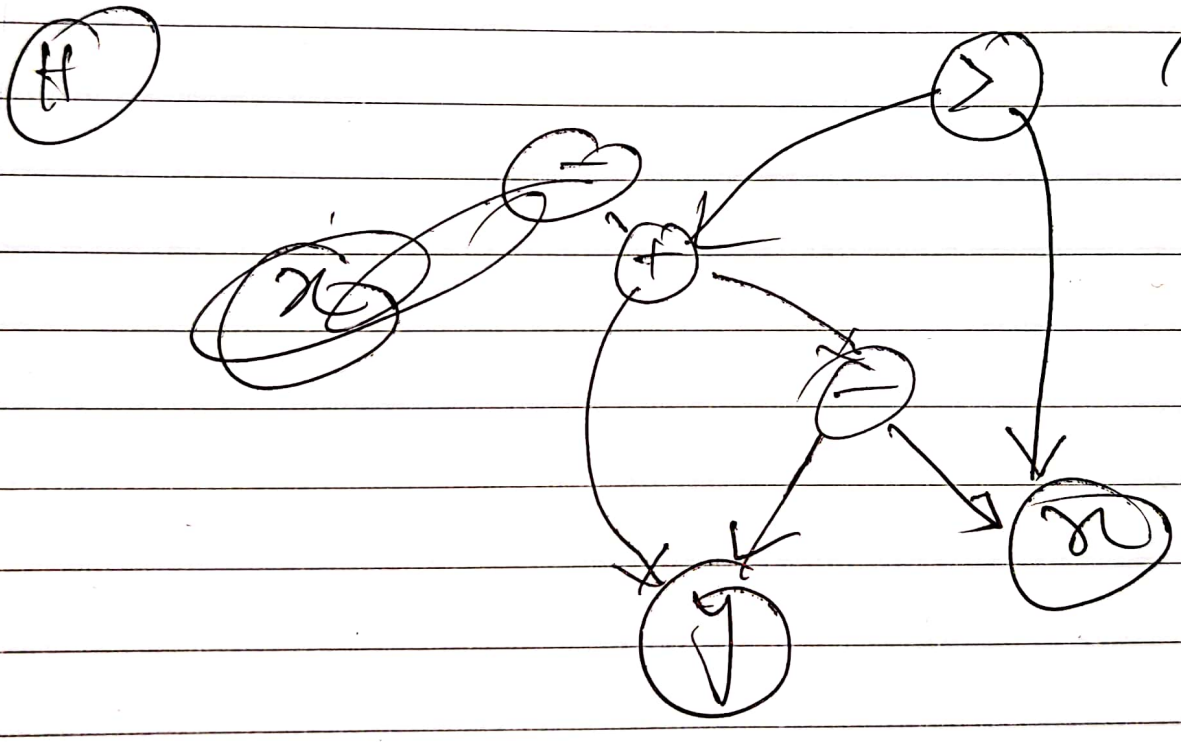
(6) Code Generation:

LDF R1, id3
MULF R1, R1, 60
LDF R2, id2
ADDF R2, R2, R1
STF id1, R2



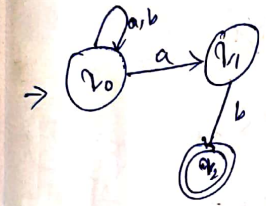
(1) $(0+1)^*01(1+0)$ q

string → (1 0 1) 0 1 1 0

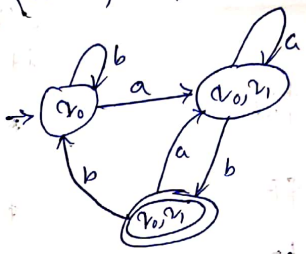


⊛ NFA to DFA:

State	a	b
q_0	q_0, q_1	q_0
q_1	ϕ	q_2
q_2	ϕ	ϕ



State	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$



⊛ LL(1) - Top down

$G \rightarrow P = \{ E \rightarrow E + T \mid T$

$\therefore T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id \}$

String = id * id

$E \Rightarrow T$

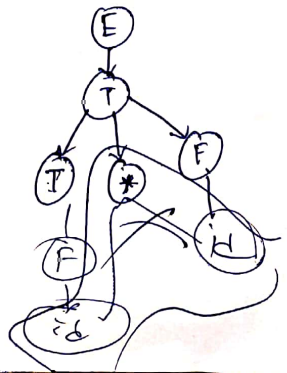
$\Rightarrow T * F$

$\Rightarrow F * F$

$\Rightarrow id * F$

$\Rightarrow id * id$

} left parsing



(*) Bottom up approach:

P: { E → E+T | T
T → T*F | F
F → (E) | id }

Stack	Input	Action
$	id * id $	
$id	* id $	shift.
$F	* id $	F → id
$T	* id $	T → F
$T*	id $	shift.
$T*id	$	shift
$T*F	$	F → id
$T	$	T → T*F
$E	$	Stack & Accept.

⊛ LR →

Item → production rule with a dot.

$E \to \cdot E + T$

$\to E \cdot + T$

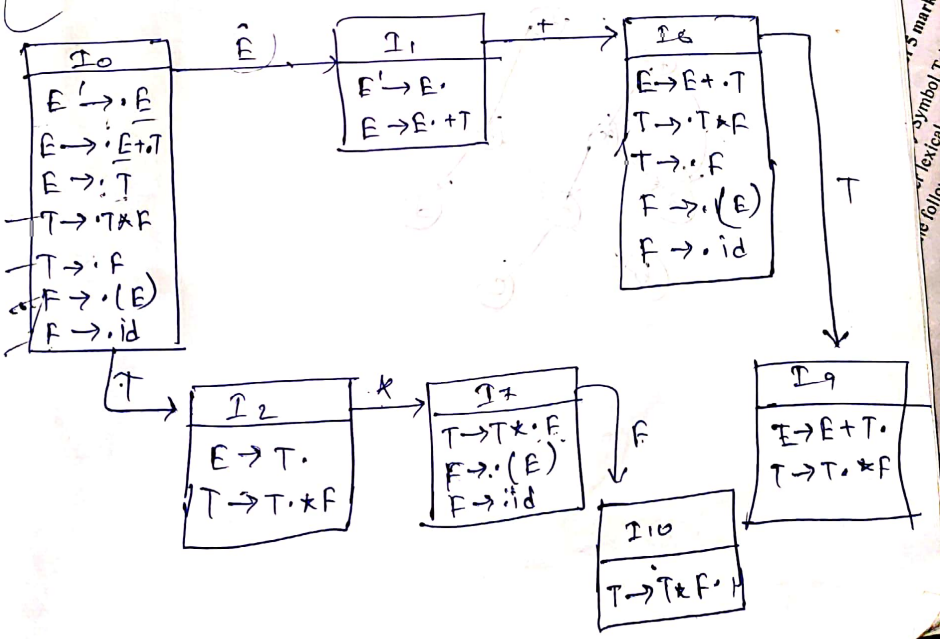
$T \to \cdot T$
$T \to \cdot F$
$E \to \cdot id$ → Exhaustive closure.

Mid Term

(12)

$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

} 7 Items



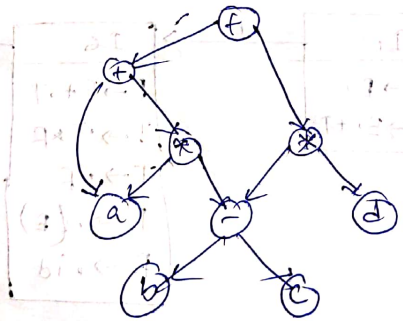
(*) Parsing in SLR automata

string (id * id)

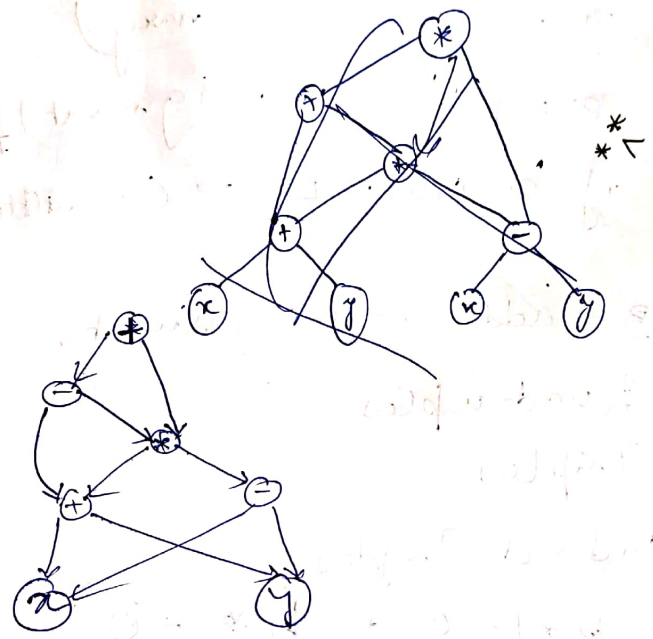
Stack	Symbol	Input	Action
0	$	id * id$	shift I5
05	$id	* id$	reduce F→id
03	$F	* id$	reduce T→F
02	$T	* id$	shift to I7
027	$T*	id$	shift to I5
0275	$T*id	$	reduce F→id
02710	$T*F	$	reduce T→T*F
02	$T	$	reduce E→T
01	$E	$	accept.

DAG

a + a * (b - c) + (b - c) * d



n) $((x+y) + ((x+y) * (x-y)) + ((x+y) * (x-y))$



$t_1 = x + y$

$t_2 = x - y$

$t_3 = t_1 * t_2$

$t_4 = t_1 - t_3$

$t_5 = t_4 + t_3$

(*) Types of Instruction:

1. $x = y + z$ (binary)
2. $x = -y$ (unary)
3. $x = y$ (assignment)
4. if x GOTO L → (conditional)

(2) 3 - addr code format:

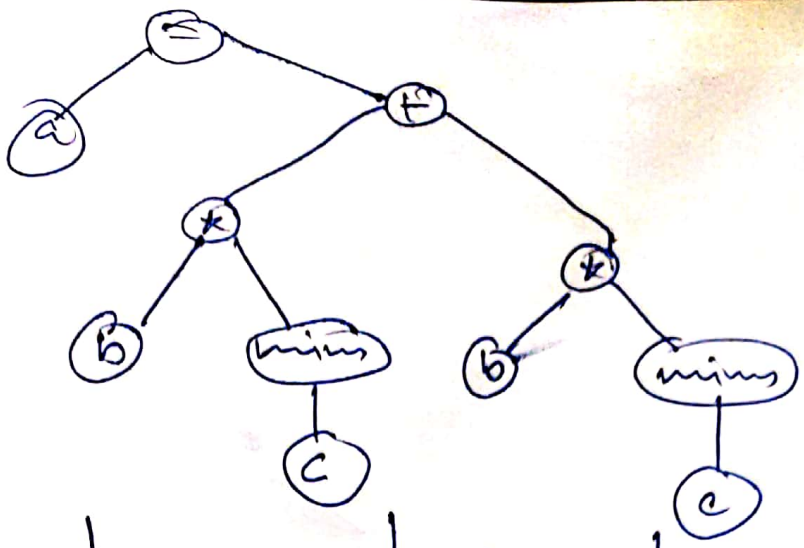
- Quadruples
- Triples
- Indirect Triples

(1) $a = b * -c + b * -c$

$t_1 = -c$

$t_2 = b * t_1$

operator	arg 1	arg 2	result
minus	c		t_1
*	b	t_1	t_2
+	t_2	t_2	a



operator	arg 1	arg 2	result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	a		a

(*) Set of all even length string over $\Sigma = \{a, b\}$

aa

18/10/19

COMPILER DESIGN.

(*) We focus on 2 main ~~steps~~ phases:
- Lexical Analysis
- Syntax Analysis.

(*) Tools used in lexical Analysis:
- formal languages
- automata theory
  - finite automata (both DFA & NFA)
  - Regular languages
  - Regular expression
  - grammers (regular & context free)

imp [conversion from NFA to DFA]

⊛ Formal language:

• Finite set of symbols – { a, b, ... z, A, B ... Z }
{ 0, ... 9 }

This finite set of symbols is referred as 'Alphabet'
↳ From this set of symbol we get string

• String is a sequence of symbols. → Infinite set.
↳ (Set of all strings) → subset

• The subset of set of all strings is known as language.

• Σ → denotes Alphabets.

⊛ Σ = { 0, 1 } 01101 → string

• The length of the string is the total number of positions present in the string.

⊛ Strings with length k (Σ^k)

eg: ~~$\Sigma^* = \{ 'A', 'B' \}$ → AA, AB~~

$\Sigma^0 = \{ \epsilon \}$

$\Sigma^1 = \{ 0, 1 \}$

$\Sigma^2 = \{ 00, 01, 10, 11 \}$

⊛ Union of all sets (Σ^*)

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots \infty \; \Sigma^\infty$

This is the set of all strings.

∴ Every language is a subset of Σ^*.

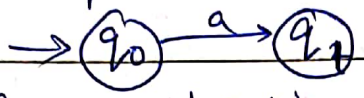
⊛ Finite Automata - There will be finite number of states. Every state will have a i/p & o/p. It's a mathematical representation of machines. It has no memory.

[Grammers is the generator of the language.

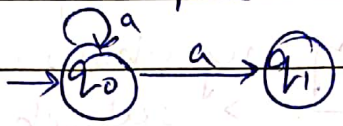
Regex

ip

* DFA - For a particular ~~wait~~ symbol, one state will go to a particular state.



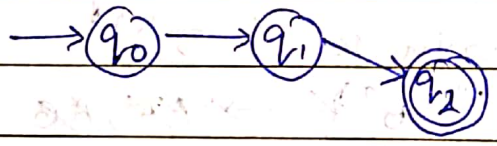
* NFA - For a particular ip symbol, one state can go to multiple states.



* Regular language = { ... } set of strings.
  $\Sigma = \{ \ldots \}$

Design a finite automata that will accept all the strings in a regular language.

* In order to accept a string, there is a final state.



- If the automata reaches the finite state, for every symols, then it is said that the string is accepted.
- when all the strings are accepted by a finite automata, then a language is accepted.

Regular Grammer → generates → Regular Language → Representation → Regular Expression

Regular Language → Accepted by → Finite Automata

⊛ Conversion from NFA to DFA:

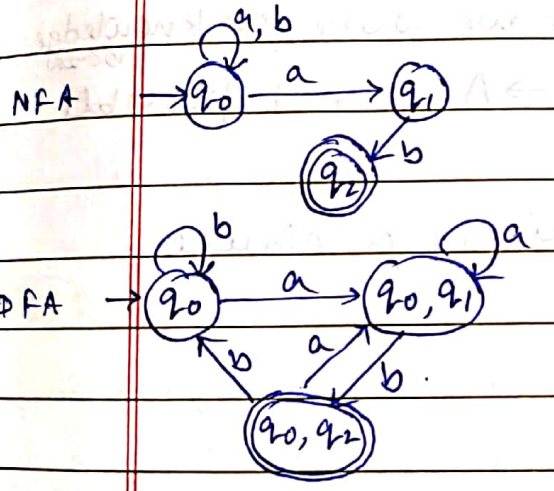
For every NFA, there is a equivalent DFA. Equivalent in the way that it accepts the same language. This is represented by transition table.

NFA

State	a	b
$\rightarrow q_0$	q_0, q_1	q_0
q_1	ϕ	q_2
$*q_2$	ϕ	ϕ

DFA

State	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
~~q_1~~	~~ϕ~~	~~q_2~~
~~$*q_2$~~	~~ϕ~~	~~ϕ~~



$\delta\{q_0, a\} = \{q_0, q_1\}$ transition function.

$\delta\{q_0, b\} = \{q_0\}$

$\delta\{q_1, a\} = \{\phi\}$

$\delta\{q_1, b\} = \{q_2\}$

$\delta\{\{q_0, q_1\}, a\} = \delta\{q_0, a\} \cup \delta\{q_1, a\}$

[The state which has q_2 is the final state].

⊛ Regular Expression:

- Union → + (a+b).
- Concatination → .
- Kleen star → * (closure).

language	Regular Expr.
{a}	each a
{a, b}	a + b (either or)
{ab}	a.b
{ϵ, a, aa, ---}	a^*

⊛ Find re of ~~over~~ string of length 2:

$L = \{aa, ab, ba, bb\}$.

Reg Ex = ~~a+b~~ $aa + ab + ba + bb$.

$= a.(a+b) + b.(a+b)$

$= (a+b).(a+b)$

<u> <big> This is big text </big></u>
<strike> This is wrong </strike>
</body>

23/10/19

COMPILER DESIGN (JC Martin Books)

* Set of all strings $\Sigma = \{a, b\}$ with length 2.
  $L = (a+b).(a+b)$ [language RE]

* Set of all even length strings:
  $L = \{\epsilon, aa, ab, ba, bb, aaaa, abab, \ldots \infty\}$
  $RE = ((a+b).(a+b))^*$

* Set of all strings with exactly two a's
  $L = \{aa, baa, aabb, aaba, \ldots\}$
  $RE = b^* a b^* a b^*$

* Set of all strings with even nos of a's
  $L = \{b, bb, bbb, aab\}$.
  $RE = (b^* a b^* a b^*)^* + b^*$

* Set of all string that begins and ends with 2 diff symbols
  $L = \{ab, abb, ba, baaba \ldots\}$
  ~~$RE = a^* b^* + b^* a^*$~~ $RE = a(a+b)^* b + b(a+b)^* a$

[+ → OR
  . → AND]

⊛ Grammers: Set of rules.

$<V, T, P, S>$

V → Set of non terminals {A, B} (variables)

T → Set of terminals {a, b, } = Σ

P → Set of Production Rules {Head → Body} (combination of symbols (V, T))

S → Starting Symbol.

eg: $V = \{S\}$

$T = \{+, -, a\}$

$P = \{ S \to SS+ \mid SS- \mid a \}$

$S = S$.

$S \Rightarrow \underline{S}S-$

$\Rightarrow \underline{S}S+S-$

$\Rightarrow aS+S-$

$\Rightarrow aa+S-$

$\Rightarrow aa+a-$

When there is no V. then string is generated

⊛ Grammer & Language: If we can generate all strings with a particular grammer, then that grammer is said to generate all ~~language~~ strings of that particular language.

⊛ $L = (a+b).(a+b)$

Grammer: $V = \{S, A\}$. $T = \{a, b\}$

$P = \{ S \to AA, \quad A \to a \mid b \}$

$S = S$.

(*) $L = (a+b)^*$

grammer = $P = \{ S \to aS \mid bS \mid \epsilon \}$

(*) $L = a(a+b)^* b$

$P = \{ S \to aAbB \mid \epsilon,$
$A \to a,$
$B \to b \}$

$P = \{ S \to aAb, A \to aA \mid bA \mid \epsilon \}$

(*) $L = a^n b^n \mid n > 1$

$P = \{ S \to AB \mid$
$A \to a$
$B \to b \}$

$P = \{ S \to aSb \mid ab \}$

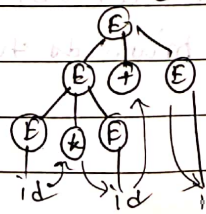
(*) Set of all Palindromes over $\Sigma = \{a, b\}$.

$P = \{ S \to aSa \mid bSb \mid a \mid b \mid \epsilon \}$.

(*) $E \to E + E \mid E * E \mid id$. → grammer

id * id + id (string)

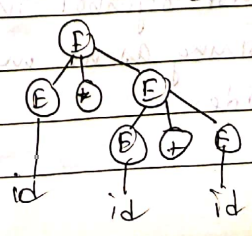
$E \Rightarrow E * E$
$\Rightarrow E * E + E$
$\Rightarrow id * id + id$.



(L → R read all deep nodes)

Syntax trees

$E \Rightarrow E * E$
$\Rightarrow E * E + E$
$\Rightarrow id * id + id$.



2. What do you mean by ... ?
... entries after lexical analysis?
... NFA to an equivalent DFA;

* Ambigious Grammer → For same string if we can come up with different tree it is known as ambigious grammer.

30/10/19

COMPILER DESIGN

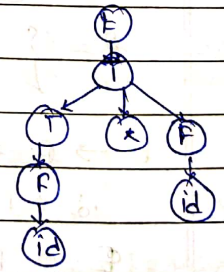
* Top Down Parsing - We try generating some string from the grammer.
* Bottom up parsing - We move from the string to starting symbol.

* Top down parsing - eg:

eg: Gr = $P = \{ E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id \}$

String = id * id

$E \to T$
$\to T * F$
$\to F * F$
$\to id * F$
$\to id * id$.



It is called top down parsing as it starts from the very first symbol in grammer and reaches the target string as we move downwards towards the leaf nodes.

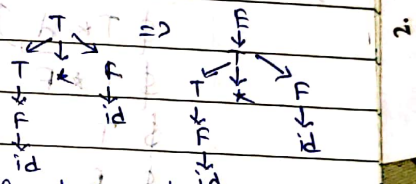
Handelpruning?
If not possible to reduce, the substring is discarded, this is called handelpruning.

2 operators
- Shift (going to other symbol)
- Reduce (replace some part of the string using production rule)

Bottom up parsing - (Reducing an input string to its starting symbol)

id * id	F → id	(LL(1) parser)
F * id	F → id	(Left most derivation)
F * F	T → F	(LR parser)
T * F	T → T * F	↳ Right most derivation in Reverse.)
T	E → T	
E		

id * id ⇒ F * id ⇒ F * F ⇒ T * F ⇒ ⇒



Parsing process - reads the string from left to right.

	Handle	
$id_1 * id_2$	id_1	F → id
F * id_2	F * id_2	T → F
T * id_2	id_2	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

LR

Handle - part of the string that the parser is focusing on for reduction.

$id_1 + id_2$	id_1	F → id
F + id_2	F	T → F
T + id_2	T	E → T
E + id_2	id_2	F → id_2
E + F	F	T → F
E + T	E + T	E → E + T

⊛ Stack implementation :

string : id * id $ marks the end of the string

marks the end of the stack →

STACK	INPUT	ACTION
$	id * id $	Shift
$id	* id $	Reduce, F→id
$ F	* id $	Reduce, T→F
$ T	* id $	Shift
$ T *	id $	Shift
$ T * id	$	Reduce, F→id
$ T * F	$	Reduce, T→T*F
$ T	$	Reduce, E→T
$ E	$	Stop / ACCEPT.

handle part of the string from top of the stack.

challenge to identify the handle

challenge to make the decision to shift when it can't be reduced

↳ Shift

31/10/19 COMPILER DESIGN [4th PDF - 242-248]

* LR parsing:

If some part of the string is matching with some part of production then it is represented by.

$A \rightarrow X Y \cdot Z$ (match found in input string)

$\alpha B \beta$ (XY...)

- Any production rule with · notation is called an item. (LR(0) item).

* Closure of set of items: (I) → Closure(I)

Grammer: $E \rightarrow \cdot E + T \mid \cdot T$

$T \rightarrow \cdot T * F \mid \cdot F$

$F \rightarrow \cdot id$

1. $I = \{ E \rightarrow \cdot E + T,\ E \rightarrow \cdot T,\ T \rightarrow \cdot T * F,\ T \rightarrow \cdot F,\ F \rightarrow \cdot id \}$.

2. CLOSURE (I) ← I

9 } → SLR automata / LR(0) automata construction.
→ complete the following SLR automata.

3. $A \rightarrow \alpha . B\beta$ (item) & production rules

$B \rightarrow .\gamma$.

add, $B \rightarrow .\gamma$ to CLOSURE(I).

* Canonical LR(0): collection / exhaustive set of all the closures.

* Each of these sets are considered as one state. All these states are known as finite automata or SLR automata or LR(0) automata.

* GOTO → function that is used to build LR(0) / SLR automata. It takes 2 inputs. Transition from one item set to another. GOTO(I, x)

* Augment Grammer:

$E' \rightarrow E$

↑ special starting symbol.

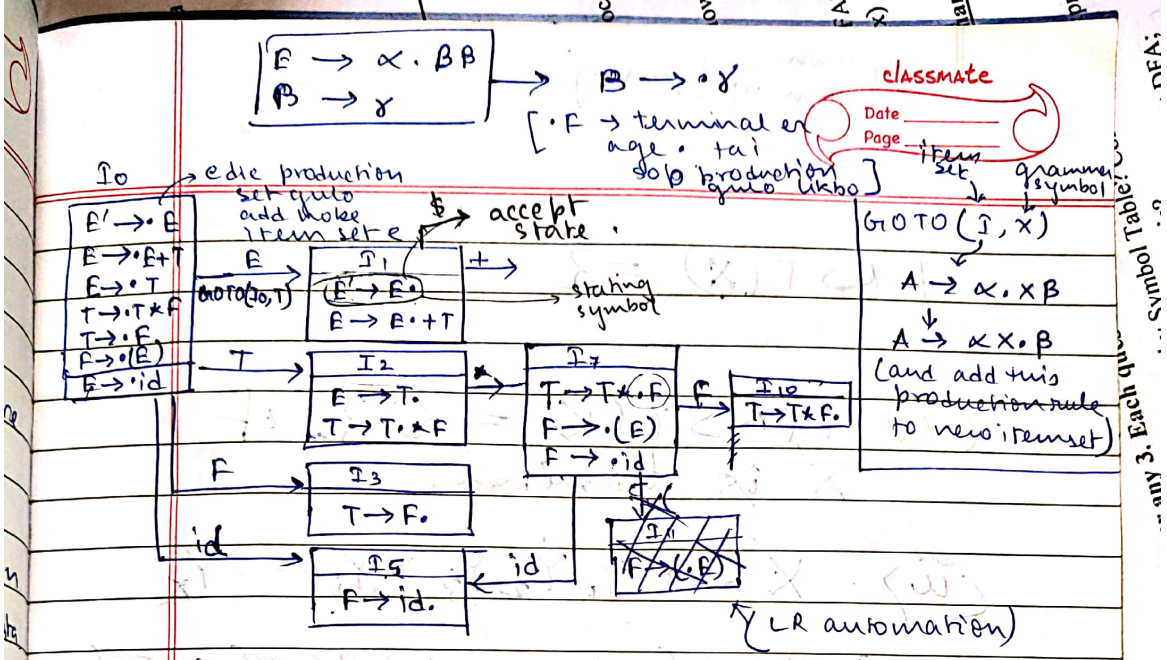
If this is added to an existing grammer then it becomes an augment grammer. This is used to stop the parsing.

* closure of Item sets:

Augment Grammer:

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

$T \rightarrow .T * F,\ T \rightarrow .F,\ F \rightarrow .(E),\ F \rightarrow .id$



(*) Parsing string id * id in SLR automata

Item set 0	STACK	SYMBOL	INPUT	ACTION	Priority is given to reduction (shift).
(1)	0	$	id * id $	shift to I5	
(2)	0 5	$ id	* id $	reduce by F → id	if we cannot shift we reduce
(3) corresponding state of F	0 3	$ F	* id $	reduce T → F	
(4)	0 2	$ T	* id $	shift to I7	
(5)	0 2 7	$ T *	id $	shift to I5	
(6)	0 2 7 5	$ T * id	$	reduce F → id	
(7)	0 2 7 10	$ T * F	$	reduce T → T*F	
(8)	0 2	$ T	$	reduce E → T	
(9)	0 1	$ E	$	accept	

- For parser generation we have automated tools such as YACC. (Yet Another compiler compiler)

(Verification of a grammar)

Syntax Analysis (Parsing)
- Top down (LL(1))
- Bottom up (LR(0))

Semantic Analysis - Done using ~~software~~ (Scope)

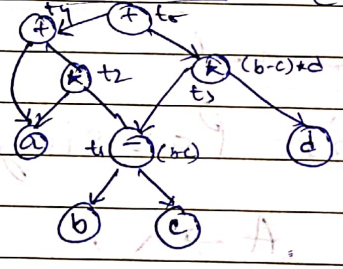
COMPILER DESIGN

1/17/19

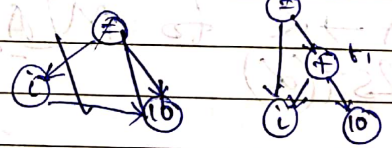
* Intermediate code generation:
  - Similar to assembly level language representation
  - 3-address code
  - DAG (Directed Acyclic Graph)

* DAG (Directed Acyclic Graph):
  A set of vertices and edges. Graphs are algebraic structures without any algebraic operators.

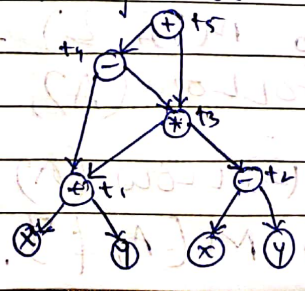
eg: 1. a + a * (b-c) + (b-c) * d



eg: 2. i = i + 10



3. ((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))



LEX, YACC, Activation Record → short note (Symbol table) → syntax analysis phase.

classmate Date Page

* 3 address code representation:
- For every operator node, take 1 variable.

1. $t_1 = b + c$.
   $t_2 = a * t_1$
   $t_3 = t_1 * d$.
   $t_4 = a + t_2$
   $t_5 = t_4 + t_3$.

2. $t_1 = i + 10$
   $i = t_1$

3. $t_1 = x + y$
   $t_2 = x - y$
   $t_3 = t_1 * t_2$
   $t_4 = t_1 - t_3$.
   $t_5 = t_4 + t_3$.

* Types of Instructions:
1. $x = y + z$ (operators: +; operands: y, z) → binary operator instruction
2. $x = -y$ → unary operator "
3. $x = y$ → assignment
4. if x GOTO L → conditional

* Store 3-address code:
stored in memory in some format
→ Quadruples, → Triples → Indirect Triples.
(data structures → list)

(unary)

* Quadruples: eg: $a = b * -c + b * -c$

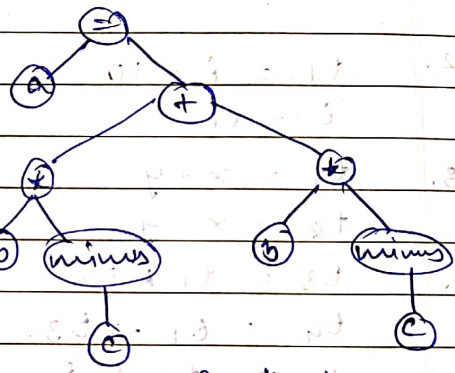
	operator	arg1/ operand1	arg2/ operand2	result
(1)	minus	c		t_1
(2)	*	b	t_1	t_2
(3)	+	t_2	t_2	a

3 addr code:

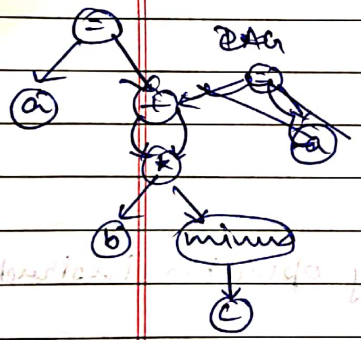
$t_1 = \text{minus } c$

$t_2 = b * t_1$

$a = t_2 + t_2$



syntax tree



DAG (identify common subexpressions so that it can be optimised in the next stage).

Intermediate code is generate from ~~start~~ syntax tree, so the table will not be optimised.

3 addr code of syntax tree:

$t_1 = \text{minus } c$

$t_2 = b * t_1$

$t_3 = \text{minus } c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

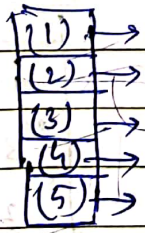
$a = t_5$

operator	arg 1	arg 2	result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

* Triple: Try to reduce the variables to reduce code size.

	operation	arg 1	arg 2	
(0)	minus	c		
(1)	*	b	(0)	reference of instruction
(2)	minus	c		
(3)	*	b	(2)	
(4)	+	(1)	(3)	
(5)	=	a	(4)	

* Indirect Triple: The structure of triple remain same. Only one array is maintained which is the pointer to the instruction



reference to the actual code is uploaded here, so it can be relocatable.

65-90 → A-Z
97-122 → a-z

→ sizeof( )
↓
unary operator

1/11/19

COMPILER DESIGN

PRACTICAL:

1. C program to check identifier is valid or not.
   INPUT: max — valid
   123 max — not valid
   max $ — valid.

2. Take input of string for any length.
   - scanf("%[^\n]")
   - while((ch = getchar()) != '\n')
   - realloc

3. Find the no of tokens in a strings
   INPUT: int a = 10;
   OUTPUT: 5.
   statement → Declaration & Defination.

4. Replace malloc() with calloc().
   I/P → c = (int *) malloc (n * sizeof(int));
   O/P → c = (int *) calloc (n, sizeof(int));

5. check necessary prerequisits of c program
   → main(), { }, int main() { ... return(); }.
   FILE *fp;

5. Find integer type variables:

I/P → int a, b, c;

O/P → a, b, c.

6. Total ~~no~~ amt. of memory to be allocated.

I/P → int a, b, c; } use sizeof();

O/P → 6.

7. Check validity of function prototype.

I/P → <returntype> <funcname> <(parameterlist)>;

I/P → void fun();

void fun(); invalid.

void fun(int,); invalid.

(0+1) Semantic Analyzer?

7. What is 'Three address code'?