PROTEGE
CHEMICALS

Date : / /
Mo Tu We Th Fr Sa Su

PART 2.

1. LEXICAL ANALYSIS

Token Definitions

Token Type	Pattern/Regex	Token Name
KEYWORDS		
PLAYER	player	PLAYER
ENEMY	enemy	ENEMY
MOVE	move	MOVE
SET	set	SET
IF	if	IF
PRINT	print	PRINT
END	end	END
UP	up	UP
DOWN	down	DOWN
LEFT	left	LEFT
RIGHT	right	RIGHT
OPERATORS		
ASSIGN	=	ASSIGN
PLUS	+	PLUS
MINUS	-	MINUS
MULTIPLY	*	MULTIPLY
DIVIDE	/	DIVIDE
EQUAL	==	EQUAL
NOT_EQUAL	!=	NOT_EQUAL
GREATER	>	GREATER
LESS	<	LESS

Token Type	Pattern/Regex	Token Name
DELIMITERS		
LBRACE	{	LBRACE
RBRACE	}	RBRACE
LPAREN	(	LPAREN
RPAREN	)	RPAREN
SEMICOLON	;	SEMICOLON
DOT	.	DOT
COMMA	,	COMMA
LITERALS		
INTEGER	[0-9]+	INTEGER
STRING	"[^"]*"	STRING
IDENTIFIERS		
IDENTIFIER	[a-z][a-z0-9]*	IDENTIFIER
SPECIAL		
COMMENT	//.*\n	(ignored)
WHITESPACE	[\t\r\n]+	(ignored)

Date:

Handwritten design documents
lexical phase:

Example Code Used:-

Example A →

```
Player hero {
        x = 0;
        y = 0;
    score = 10;
}

enemy troll {
        x = 4;
        y = 4;
}

move hero up 3;
move hero right 2;

if hero.x + hero.y > troll.x {

    print "Advantage!";

}
```

Date:

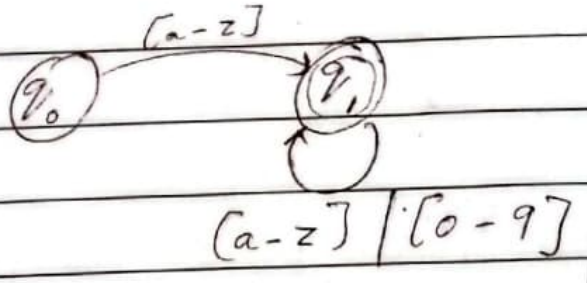
lexical phase

lexemme	token type
player	KEYWORD
hero	ID
{	LBRACE
x	ID
=	OPERATOR
0	INTEGER
;	SEMICOLON
y	ID
score	ID
10	INTEGER
}	RBRACE
enemy	KEYWORD
troll	ID
4	INTEGER
move	KEYWORD
up	DIRECTION
3	INTEGER
right	DIRECTION
2	INTEGER
If	KEYWORD
.	DOT
+	OPERATOR
>	REL_OR
Print	KEYWORD
"Advantage!"	STRING

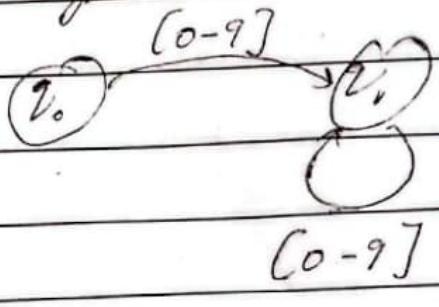
Date:

DFA Construction for lexical Analysis

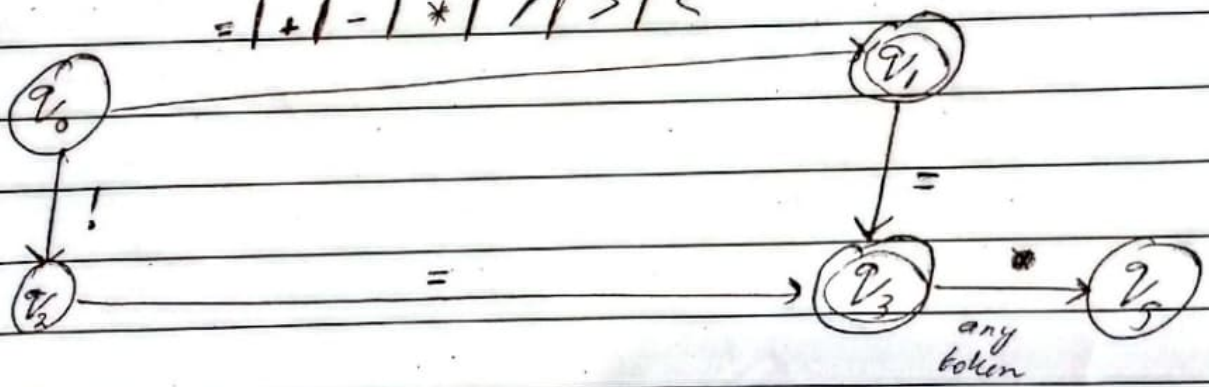
ID:



Integer:

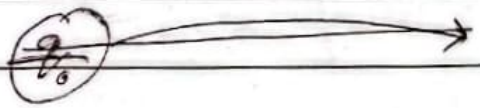


Operators:

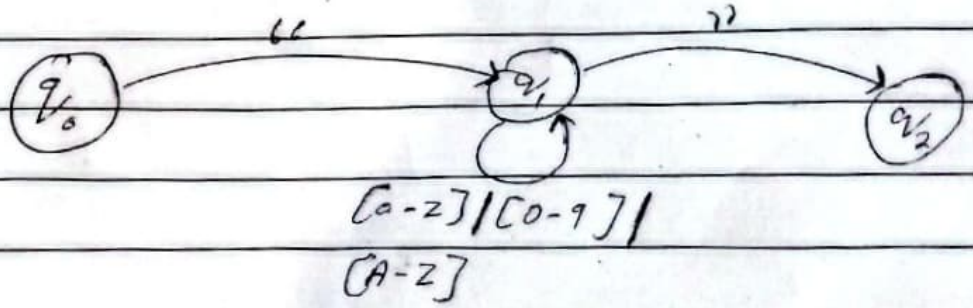


~~Print statement :~~

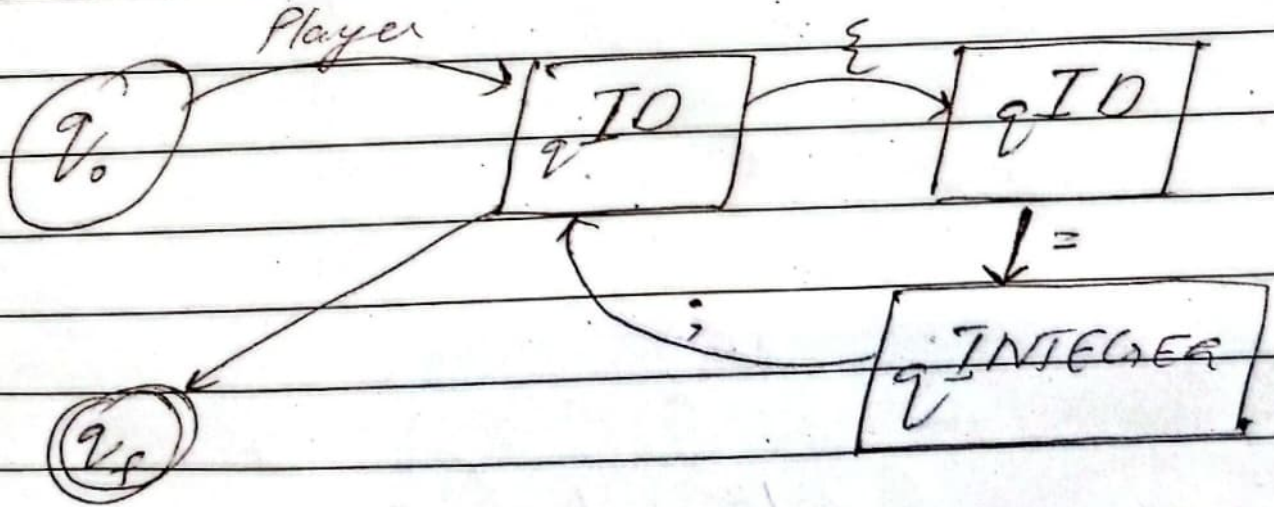
~~String:~~



String:

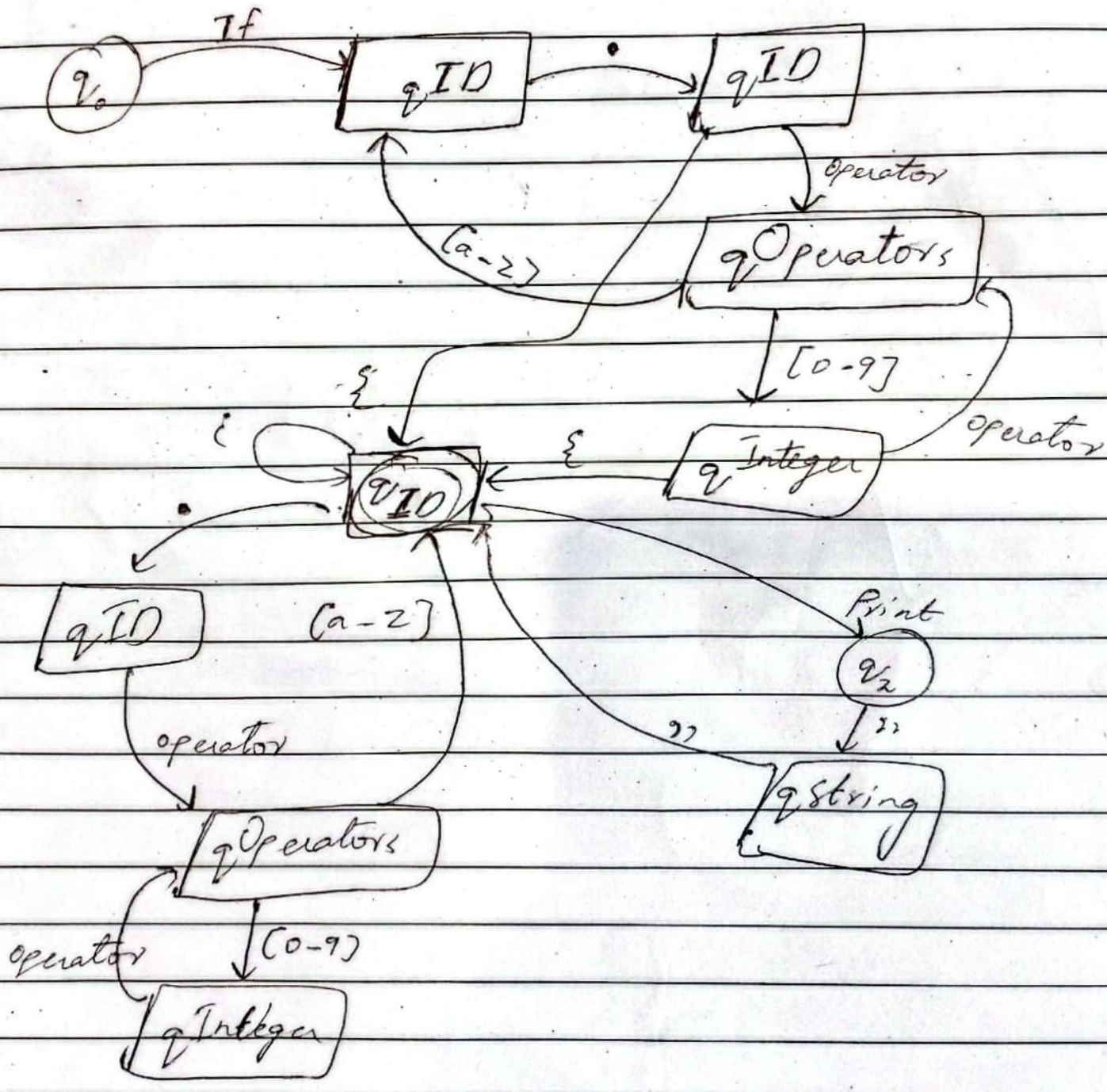


Class :



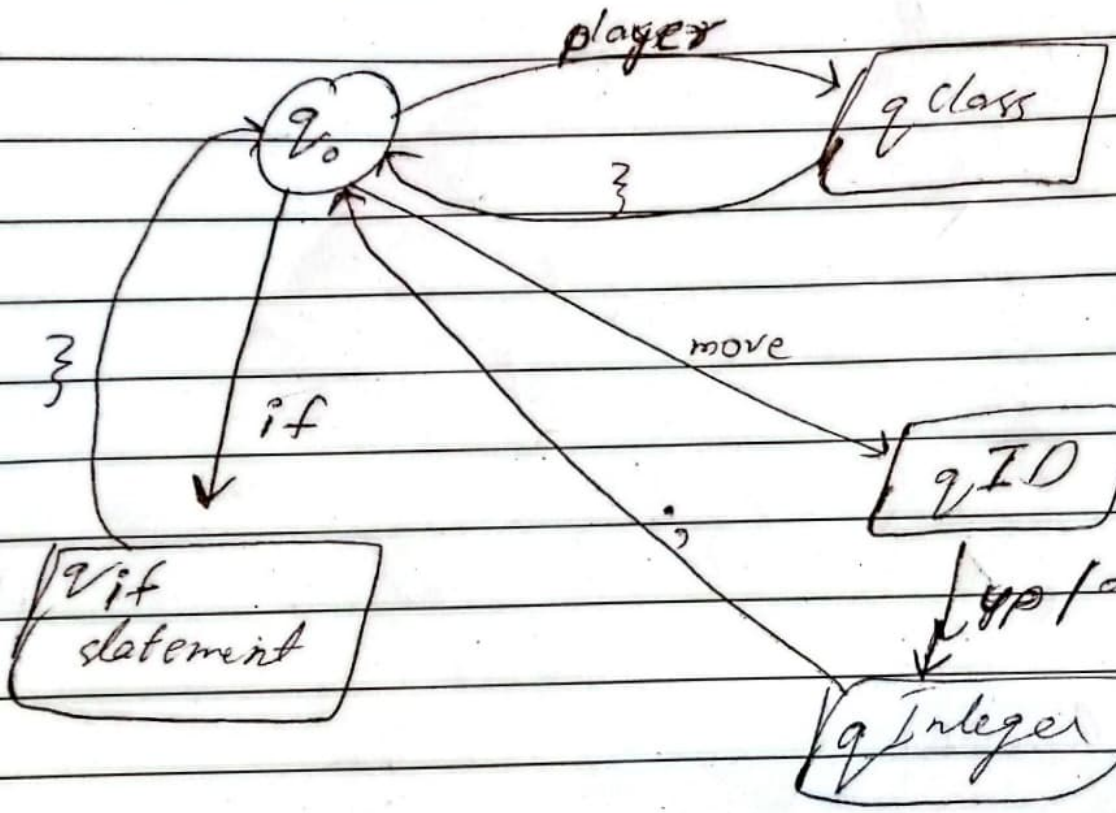
Date:

If statement:



Date:

final program:



Date:

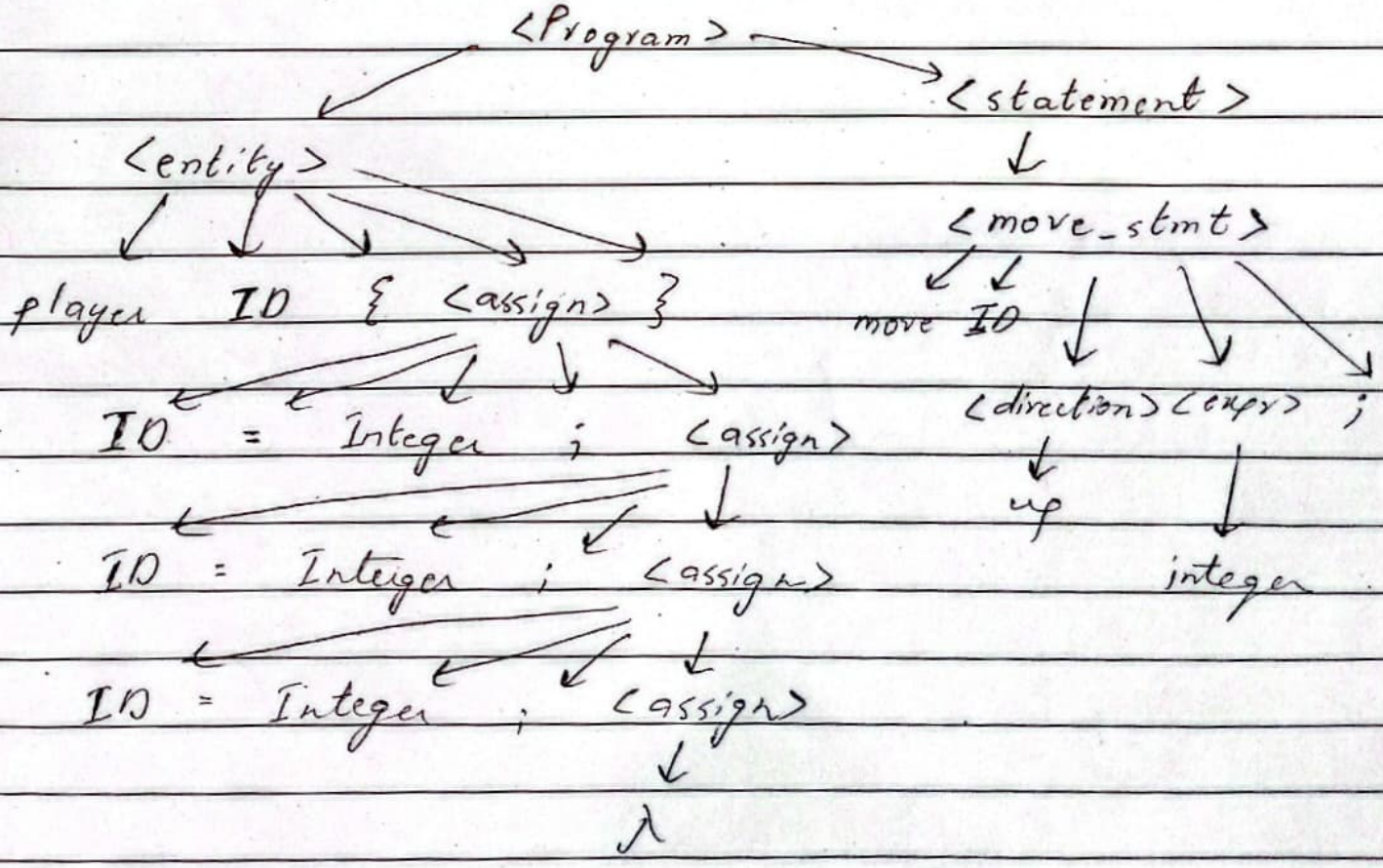
Syntax phase

Example B:

```
player hero {
    x = 0;
    y = 0;
    score = 10; }

move hero up };
```

Tree:



Date:

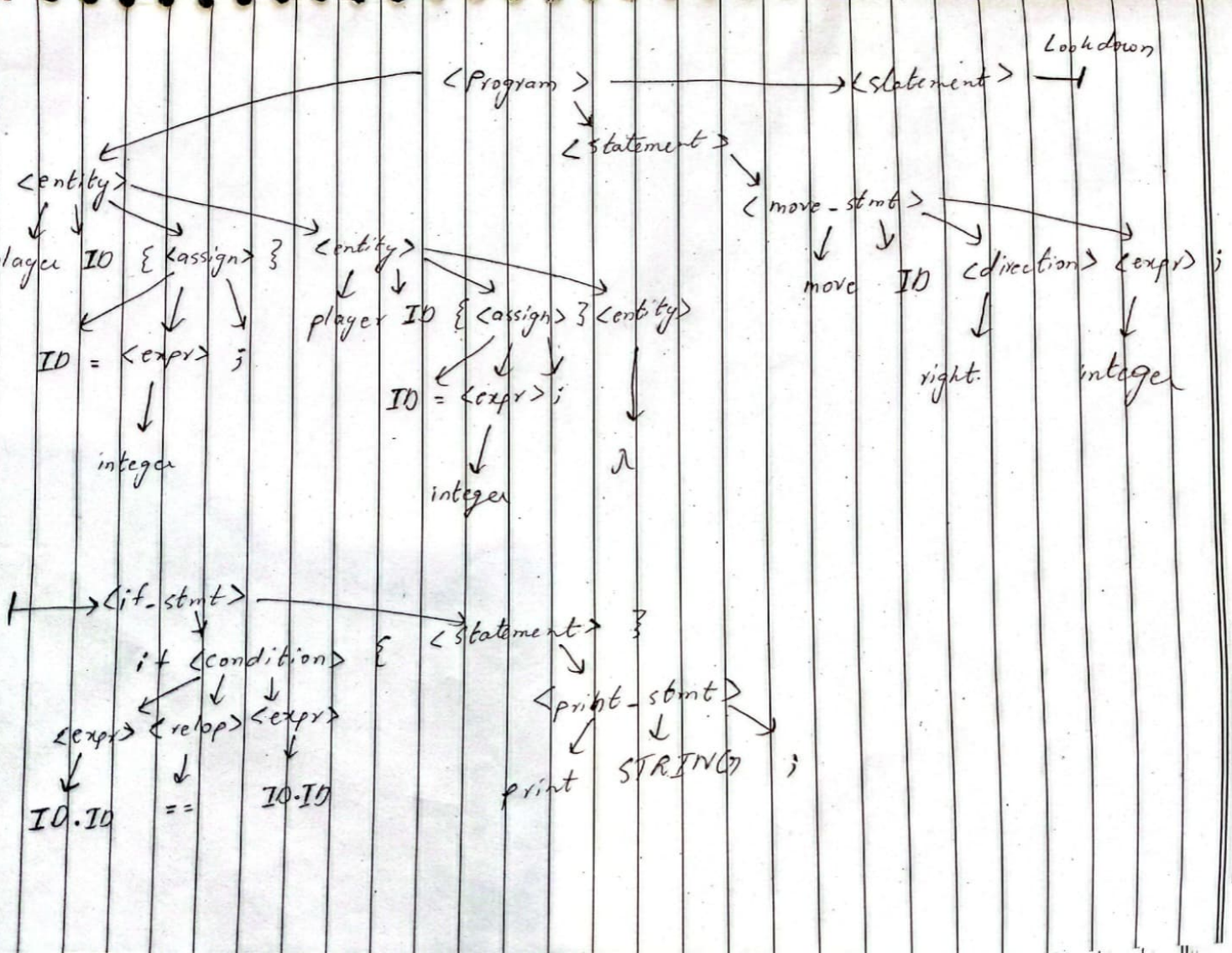
Program example C. (parse tree on next page)

```
Player hero {
    x = 0; }

Player enemy {
        x = 5;
}

move hero right 5;

if hero.x == enemy.x {
        print "Caught";
}
```



Page Victory

Date:

PROTEGE
CHEMICALS

Date : / /
Mo Tu We Th Fr Sa Su

PART 2.

3. SEMANTIC ANALYSIS

Semantic Rules in SDD Format

Production (CFG)	Semantic Rules (SDD)	Attribute Type
<expr> → <expr$_1$> + <term>	<expr>.type = int <expr>.node = Node('+', <expr$_1$>.node, <term>.node)	S-attr
<expr> → <expr$_1$> - <term>	<expr>.type = int <expr>.node = Node('-', <expr$_1$>.node, <term>.node)	S-attr
<expr> → <term>	<expr>.type = <term>.type <expr>.node = <term>.node	S-attr
<term> → <term$_1$> * <factor>	<term>.type = int <term>.node = Node('*', <term$_1$>.node, <factor>.node)	S-attr
<term> → <factor>	<term>.type = <factor>.type <term>.node = <factor>.node	S-attr
<factor> → INTEGER	<factor>.type = int <factor>.node = Leaf(INTEGER.value)	S-attr
<factor> → ID.ID	<factor>.type = lookup(ID_1, ID_2) <factor>.node = Leaf(ID_1.name, ID_2.name)	S-attr

PROTEGE
CHEMICALS

Date : / /
Mo Tu We Th Fr Sa Su

Semantic Rules in SDD Format continued ...

Production (CFG)	Semantic Rules (SDD)	Attribute Type
<condition> → <expr$_1$> == <expr$_2$>	<condition>.type = bool <condition>.node = Node('==', <expr$_1$>.node, <expr$_2$>.node)	S-attr
<assign> → ID = <expr>	addSymbol(ID.name, <expr>.type) <assign>.node = Node('=', ID, <expr>.node)	L-attr
<set_stmt> → set ID$_1$.ID$_2$ = <expr>	checkProperty(ID$_1$, ID$_2$) <set_stmt>.type = <expr>.type	L-attr
<entity> → player ID { <assigns> }	createEntity(ID.name, "player") <entity>.scope = ID.name	L-attr

Attribute Classifications

S-attributes

<expr>.type	type flows upward from children
<expr>.node	AST node synthesized from children
<term>.type	synthesized from factors
<factor>.type	synthesized from terminals
<condition>.type	synthesized from expressions

L-attributes

<entity>.scope	scope name passed down to properties
<assign>.scope	current entity scope from parent
<set_stmt>.entityName	entity context for property lookup

Date:

Semantic Phase

We will use example A, mentioned in the lexical phase.

Symbol table

Name	Type	Scope	~~Value~~ value
k			-
hero	player	Global	0
hero.x	integer	hero(player)	0
hero.y	int	hero(player)	10
hero.score	int	~~Global~~ hero(player)	-
~~enemy~~ troll	enemy	~~troll(enemy)~~ Global	4
troll.x	int	troll(enemy)	4
troll.y	int	troll(enemy	

Date:

Symbol table for error:

move ~~player~~ hero up 5;

~~class player~~ player hero {

}

~~Type~~

- hero is not declared yet
- when move hero up 5 statement looks hero ID in table it does not find it and throws error.

Name	Type	Scope	Value
hero	-	-	Not declared

PROTEGE CHEMICALS

Date : / /
Mo Tu We Th Fr Sa Su

PART 2.

4. INTERMEDIATE CODE GENERATION

Three-Address Code (TAC) Instruction Types

Type	Format	Example	Description
Assignment	x = y	t0 = 5	Simple Copy
Binary Op	x = y op z	t1 = a + b	Arithmetic/Logic
Unary Op	x = op y	t2 = -x	Negation
Copy	x = y	hero.x = t0	Property Assignment
Jump	goto L	goto L1	Unconditional Branch
Conditional	if x goto L	if t0 goto L2	Conditional Branch
Relational	x = y relop z	t4 = a == b	Comparison
Label	L:	L0:	Jump Target

TAC vs Source Code Comparison Table

Source Code	TAC Instructions	Temp Count	Instruction Count
x = 5;	x = 5	0	1
x = 5 + 3;	t0 = 5 + 3 x = t0	1	2
x = a + b * c;	t0 = b * c t1 = a + t0 x = t1	2	3
if x > 5 {...}	t0 = x > 5 if t0 goto L0 goto L1 L0: ... L1:	1	5+

Date:

Intermediate Code Generation using Example A from phase 1 (lexical Analysis)

Intermediate Code:

```
0   init_hero:
1       hero.x = 0
2       hero.y = 0
3       hero.score = 0
4   init_troll:
5       hero.x = 4
6       hero.y = 4
7
8   t0 = hero.y + 3
9   hero.y = t0
10
11  t0 = hero.x + 2
12  hero.x = t0
13
14  t0 = hero.x
15  t0 = t0 + hero.y
16  t1 = t0 > troll.x
17  if t1 goto L1
18      goto L2
19  L1:
20      print "Advantage"
21  L2:
```

Date:

Optimization Example
Using Previous immediate Code

```
0       init_hero:
1           hero.x = 2
2           hero.y = 3
3           hero.score = 10
4
5       init_troll:
6           troll.x = 4
7           troll.y = 4
8
9       t0 = hero.x
10      t0 = t0 + hero.y
11      t1 = t0 > troll.x
12      if t1 goto L1
13      goto L2
14      L1:
15      print "Advantage"
16      L2:
```

* In optimisation we initialised hero.x and hero.y to 2 and 3, respectively, as in immediate code after hero.x and hero.y is initialised to 0, there is no use. And then immediately these values are incremented to 3 and 2 before using them in if statement. This step removed the redundant code and initialised directly to 2 and 3.

PROTEGE
CHEMICALS

Date : / /
Mo Tu We Th Fr Sa Su

PART 2.

6. CODE GENERATION

Instruction Mapping : TAC Instruction → Interpreter Action

TAC Instruction	Interpreter Action
t = const	Store constant in temp
t = x op y	Compute and store
entity.prop = t	Update property
t = entity.prop	Load property
if t goto L	Conditional jump
goto L	Unconditional jump
print str	Output
label :	No-operation (mark position)

Date:

last stage : Executable Code

language Used : Python

```
hero = {'x': 2 , 'y': 3, 'score': 10}

troll = {'x': 4 , 'y': 4}

t0 = hero['x']
t0 = t0 + hero['y']
t1 = t0 > ~~hero['x']~~ troll['x']
if t1 :
    print("Advantage")
```