UCS2702 COMPILER DESIGN

Unit 3

Intermediate Code Generation

Syntax-directed directions : synthesized attribute - inherited attribute - dependency graph - evaluation order of syntax directed definitions ; Intermediate languages: Syntax-tree - Three address code ; SDD for type checking - Declarations - Evaluation of expressions and flow of control statements - bottom-up evaluation of S-attribute definitions

* Introduction to Syntax-directed Translation

* Syntax - Directed Translation

→ Grammar symbols are associated with attributes to associate information with the programming language constructs that they represent.

→ Values of these attributes are evaluated by the semantic rules associated with the production rules

→ Evaluation of these semantic rules :

- may generate intermediate code
- may put info into the symbol table
- may perform type checking

- may issue error msgs

→ An attribute may hold : a string, number, memory location, a complex record

* Notations for associating Semantic Rules w/ Productions

① Syntax - Directed Definitions

② Translation Schemes

① Syntax - Directed Definitions

- give high - level specifications for translations

- hide many implementation details such as order of eval

- can associate a production rule w/ a set of semantic actions - and do not say when they will be evaluated

② Translation Schemes

→ indicate the order of evaluation of semantic actions

- TS give a little bit of information about implementation details.

- Semantic rules are represented in { }, { }

(contd.)

→ - Semantic rules set up dependencies between attributes which can be represented by a dependency graph

- Dependency graph determines evaluation order.

- Each grammar symbol is associated with a set of attributes which can be:

Learn this more

Synthesized Attributes - Derived from the attributes of the symbol's children in the parse tree

(Info found will from RHS)

eg. $E \rightarrow E_1 + T$ get $E_1.val$, $T.val$ to find $E.val$

Inherited Attributes - passed down from the parent/ sibling nodes in the parse tree.

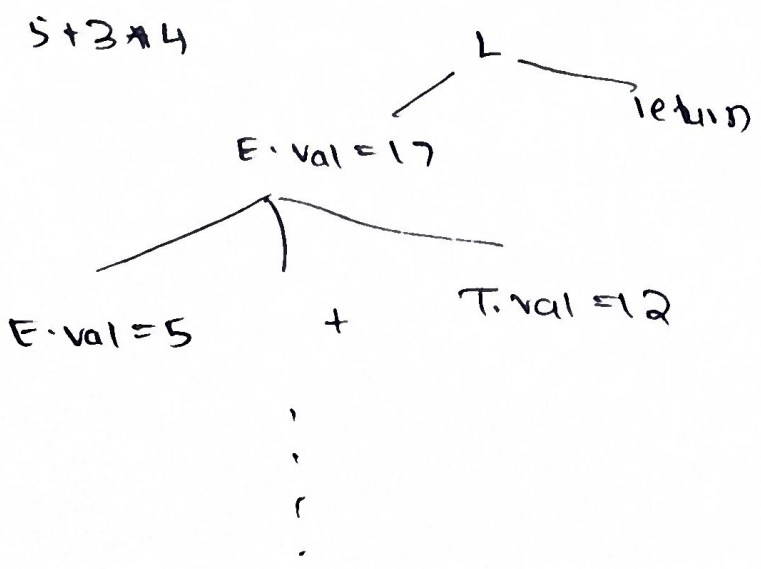
* Attribute Grammar

- a specific kind of SDD, where semantic rules have no side effects they only calculate attribute values.

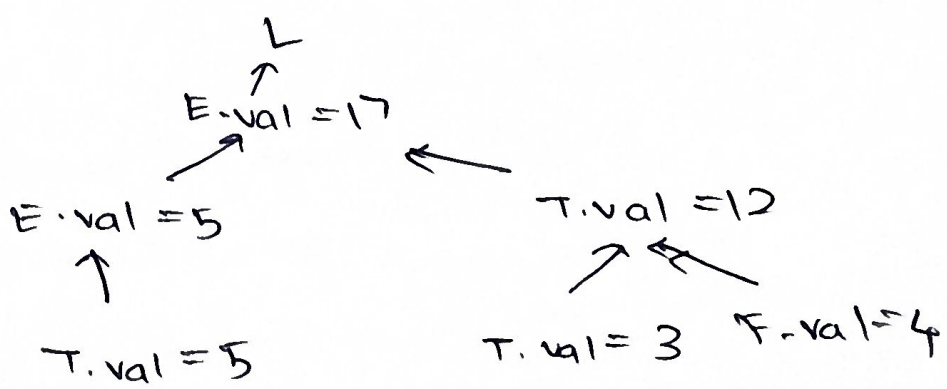
Rules are written as:

$$b = f(c_1, c_2, \ldots .. c_n)$$

Annotated Parse Tree



Dependency Graph



Example1 Write a translation scheme that converts infix expressions to the corresponding postfix expression

$E \rightarrow TR$

$R \rightarrow +T \{ print("+") \} R_1$

$R \rightarrow \epsilon$

$T \rightarrow id \{ print(id.name) \}$

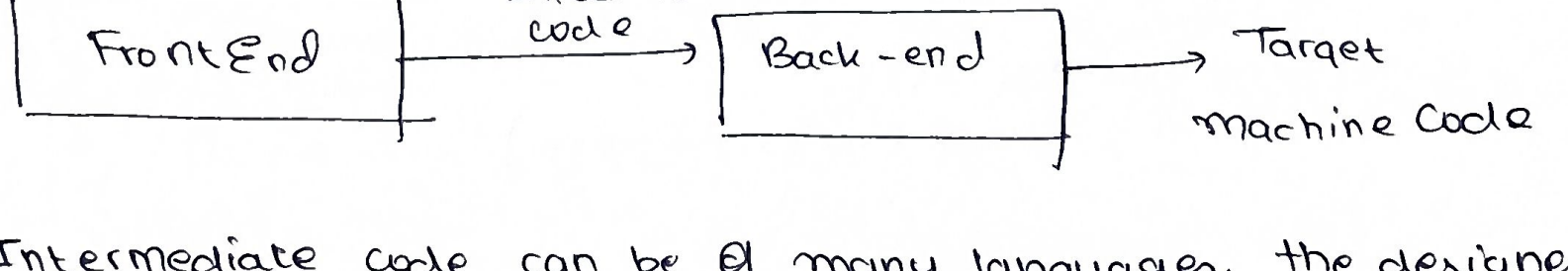
a+b+c $\Rightarrow$ ab+c+

* Introduction to Intermediate Code Generation

→ code which is machine independent code

→ close to machine instructs

→ allows a variety of optimizations to be implemented in a machine-independent way



→ Intermediate code can be of many languages, the designer of the compiler decides this intermediate language.

Notations

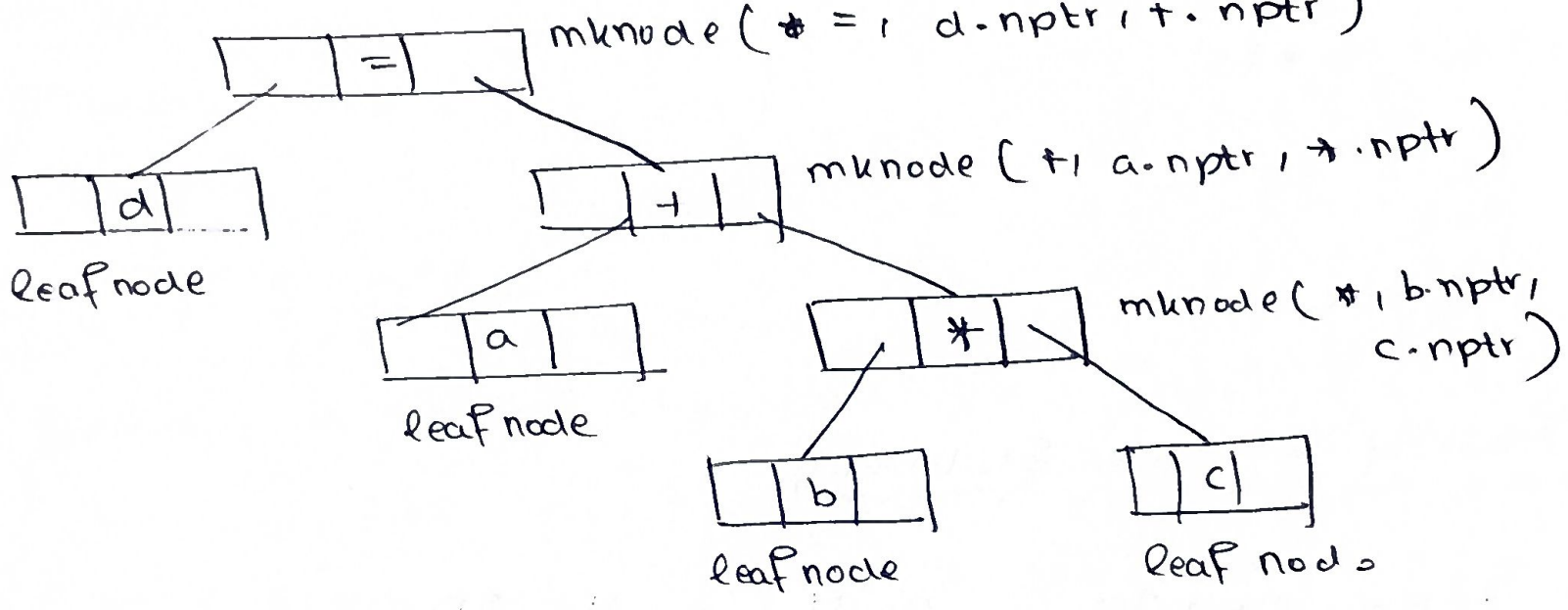
- syntax trees
- postfix notation
- 3-address code

A. Syntax Trees

$d = a + b * c$

Parse Tree

The abstract syntax tree would be as follows:



B. Postfix Notation

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

in postfix would be:

a b c uminus * b c uminus * + assign

Pro: easy to generate

con: stack operations are more difficult to optimize

c. **Three Address Code**

→ any expression is represented with a maximum of 3 address code

→ longer expressions are broken down.

eg. a := b * -c + b * -c

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

Kinds of 3 Address Statements

1. Binary Operator : result := y op z
2. Unary Operator : result := op y
3. Move Operator : result := y
4. Unconditional Jumps : goto L
5. Conditional Jumps : If y relop z goto L
6. Procedure Parameters : param x
7. Procedure Calls : call p, n

8. Indexed Assignments : $x := y[i]$

$y[i] := x$

9. Address and Pointer Assignments : $x := \&y$

$x := *y$

* Implementation of Three-Address Statements

1. Quadruples
2. Triples
3. Indirect Triples

1. Quadruples

→ A quadruple is a record structure with four fields :

op, arg1, arg2, result

op field contains an internal code for the operator

eg. $t_1 := -c$

$t_2 := b * t_1$

#	op	Arg1	Arg2	Res
(0)	uminus	c	-	t_1
(1)	*	b	t_1	t_2

Advantage : easy to rearrange

Disadvantage : lots of temporary variables

2. **Triples** Op Arg1 , Arg2

Result is implicitly taken as # [(0), (1), ...]

	#	Op	Arg1	Arg2
t1 := -c	(0)	uminus	c	
t2 := b * t1	(1)	*	b	(0)

Advantages : Temporaries are implicit

Disadvantage : Difficult to rearrange

3. **Indirect Triples**

t1 := -c

t2 := b * t1

#	Stmt
(0)	(14)
(1)	(15)

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	b	(14)

Program

Triple Contains

Advantage : Temporaries are implicit and easier to rearrange code.

Example : Translate the expression -(a+b) * (c+d) + (a+b+c) into quadruples, triples & indirect triple

Example : TAC example - PPT-1

* Bottom-up evaluation of S-attributed definitions

→ S-attributed definitions use only synthesized attributes, i.e attributes are computed using information from the children in the parse tree.

→ These attributes are evaluated in a bottom-up manner.

This is easier to implement because:

(i) There is no need to handle dependencies on parents or sibling nodes

(ii) Values are computed only during reductions in a single bottom-up pass.

Bottom-up Evaluation

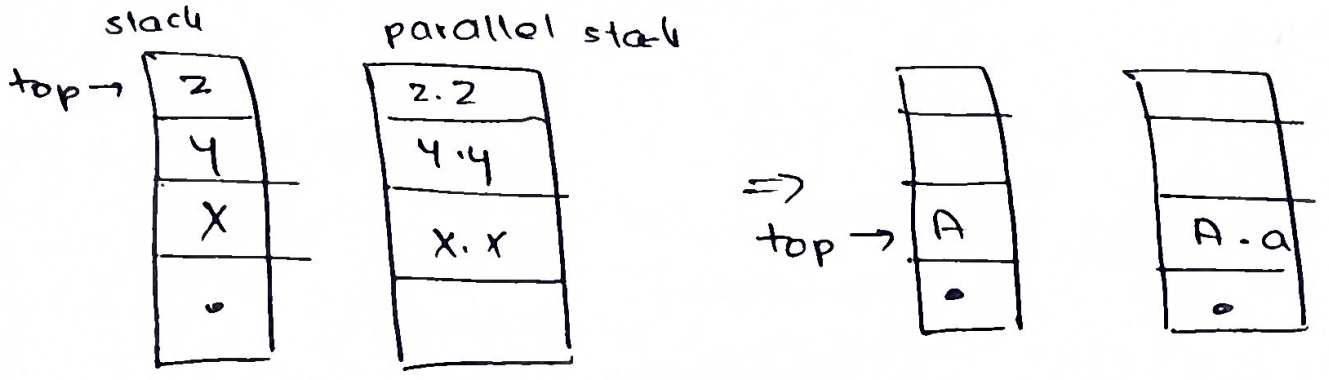
→ A parallel stack is maintained.

→ When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.

→ Evaluate the values of the attributes during reductions

$A \rightarrow XYZ$

$A.a = f(X.x, Y.y, Z.z)$



* Type Checking

→ Type checking ensures that the types of program constructs (eg: variables, expressions and function arguments) match the expected type in the context

→ The information gathered during type checking is essential for code generation, ensuring that operations are valid.

* Type System

→ A type system is a set of rules to assign type expressions to program components.

Components

(i) Type Checker - Implements the type system, often syntax-directed (works alongside the parser)

(ii) Sound Type System - Guarantees that no type errors will occur at runtime (eliminates runtime checks)

* Type Expression

→ The type of language construct is expressed by a type expression can be

(i) basic types

(ii) type name

(iii) type constructors
- array
- pointers
- functions

* Type Checking Expressions

A. **Constants**

E → int_const {E.type = const}

E → float_const {E.type = float}

B. **Identifiers**

E → id {E.type = symbol_table_lookup(id)}

C. **Operations**

E → E1 mod E2 {E.type = If (E1.type == int && E2.type == int)

and int else type-error}

D. **Binary Operations**

E → E1 + E2 { if (E1.type == int and E2.type == int)

then E.type = int

else if (E1.type == int and E2.type == real)

then E.type = real

else if (E1.type = real and E2.type == real)

then E2.type = real

else if (E1.type = real and E2.type = real)

then E.type = real

else

E.type = type-error

E. **Arrays**

$E \rightarrow E_1 [E_2]$ { E.type = if (E2.type == integer and E1.type == array (S,T)) {
then T
}
else
type-error
}

F. **Dereferencing**

$E \rightarrow * E_1$ { E.type = if (E_1.type == pointer(T)) {
then T
}
else
type-error
}

G. **Assignment**

$S \rightarrow id := E$ { S.type = if (id.type == E.type)
then void
else type-error
}

H. **Conditionals**

$S \rightarrow$ if E then S_1 { S.type = if (E.type == boolean)
then S_1.type
else
type-error
}

I. Loops

$S \rightarrow$ while E do S_1 { S.type = if (E.type == boolean)

then S_1.type

else type - error

}

II. Compound Statements

$S \rightarrow S_1 ; S_2$

S.type = if (S_1.type == void and S_2.type == void)

then void

else

type - error

X. Functions

$E \rightarrow E_1 (E_2)$ { if (E_2.type = s and E_1.type = s → t)

then E.type = t

else

E.type = type_error }

Syntax-Directed Translation (SDT) Rules & Examples

A. **Declarative Statements**

1. $P \rightarrow$ {offset = 0} D

2. $D \rightarrow D ; D$

3. $D \rightarrow id : T$ { enter (id.name, T.type, offset);
offset = offset + T.width }

4. $T \rightarrow int$ { T.type = int , T.width = 4 }

5. $T \rightarrow real$ { T.type = real , T.width = 8 }

6. $T \rightarrow array [num]$ of T_1 { T.type = array (num.val, T_1.type);
T.width = num.val * T.width }

7. $T \rightarrow *T_1$ { T.type = pointer (T_1.type);
T.width = 4 }

enter = create a symbol table entry with the given values

B. Nested Procedure Declarations

1. $P \rightarrow MD$ { addwidth (top (tblptr) , top (offset)); pop(tblptr); pop(offset); }

2. $M \rightarrow \epsilon$ { t = mktable(nil); push (t, tblptr); push (0, offset) }

3. $D \rightarrow D ; D$

4. $D \rightarrow$ proc id N D ; S { t = top (tblptr); addswith (t, top (offset)); pop(tblptr); pop(offset); enter proc (top(tblptr), id.name, t }

5. $D \rightarrow$ id : T { enter (top(tblptr), id.name, T.type, top(offset)) top(offset) = top (offset) + T. width }

6. $N \rightarrow \epsilon$ { t = mktable (top(tblptr)); push (t, tblptr); push (0, offset) }

C. **Assignment Statements**

P → MD

M → ε

D → D ; D | id : T | proc id N D ; S

N → ε

S → s ; S

S → id := E { p = lookup(id.name);
if p = nil then
error()
else
emit(id.place ':=' E.place)
}

E → E1 + E2 {E.place = newtemp()
emit(E.place ':=' E1.place '+' E2.place)}

E → E1 * E2 {E.place := newtemp(),
emit(E.place ':=' E1.place '*' E2.place)}

E → -E1 {E.place := newotemp()
emit(E.place ':=' 'uminus' E1.place)}

E → (E1) {E.place := E1.place}

E → id {p := lookup(id.name);
if p = nil then error()
else
E.place = p
}

D. Boolean Expressions

$E \rightarrow E_1$ or E_2 { E.place = newtemp();
emit (E.place = E.place or E_2.place)

$E \rightarrow E_1$ and E_2 { E.place = newtemp();
emit (E.place = E_1.place and E_2.place ; }

$E \rightarrow$ not E { E.place = newtemp();
emit (E.place = not E.place); }

$E \rightarrow (E_1)$ { E.place = E_1.place }

$E \rightarrow id_1$ relop id_2 { E.place = newtemp();
emit (if id_1.place relop.op id_2.place
goto nextstat+3);
emit (E.place = 0)
emit (goto nextstat+2);
emit (E.place = 1); }

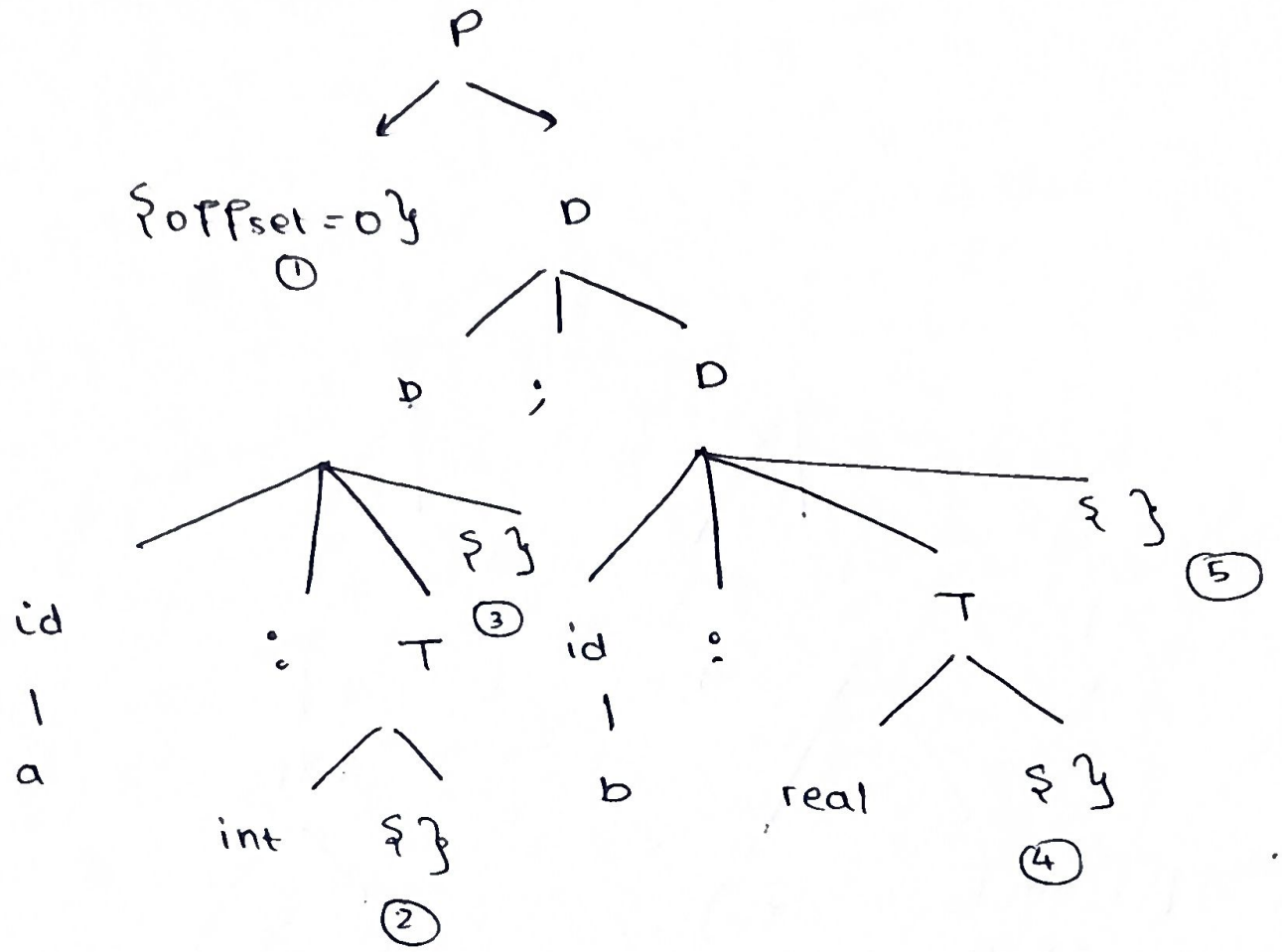
$E \rightarrow$ true { E.place = newtemp()
emit (E.place = 1) }

$E \rightarrow$ false { E.place = newtemp();
emit (E.place = 0); }

① Example 1 - Declarative Statements

a : int ;

b : real ;



Perform DFS

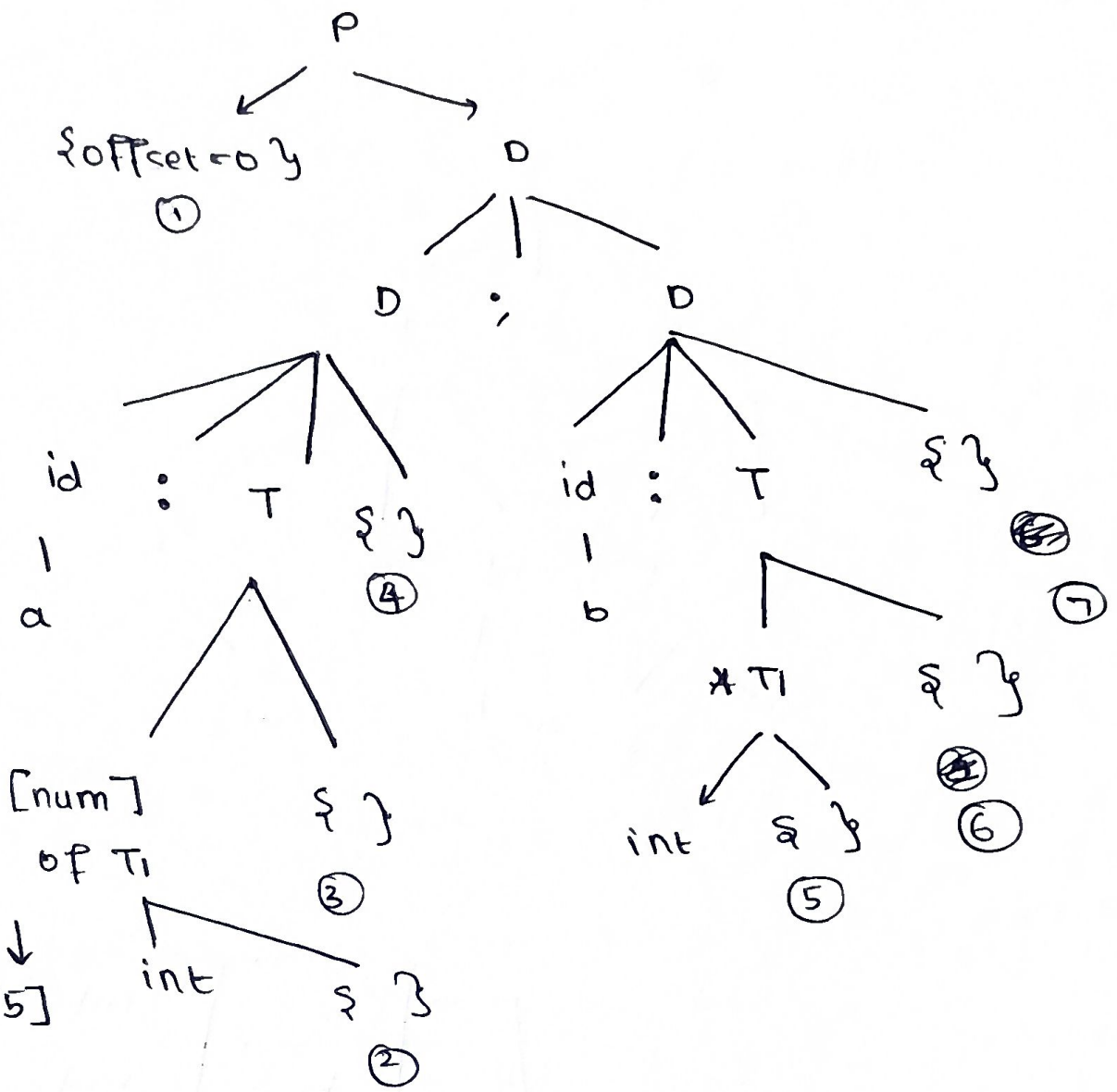
1. offset = 0
2. T.type = int
3. T.width = 4
4. enter (id.name, T.type, offset)
5. offset = 0 + 4 = 4
6. T.type = real
7. T.width = 8
8. enter (id.name, T.type, offset);
9. offset = 4 + 8 = 12

a	int	0
b	real	4

② Example 2 : Arrays and Pointers Declaration

a : array [5] of int

b : * int



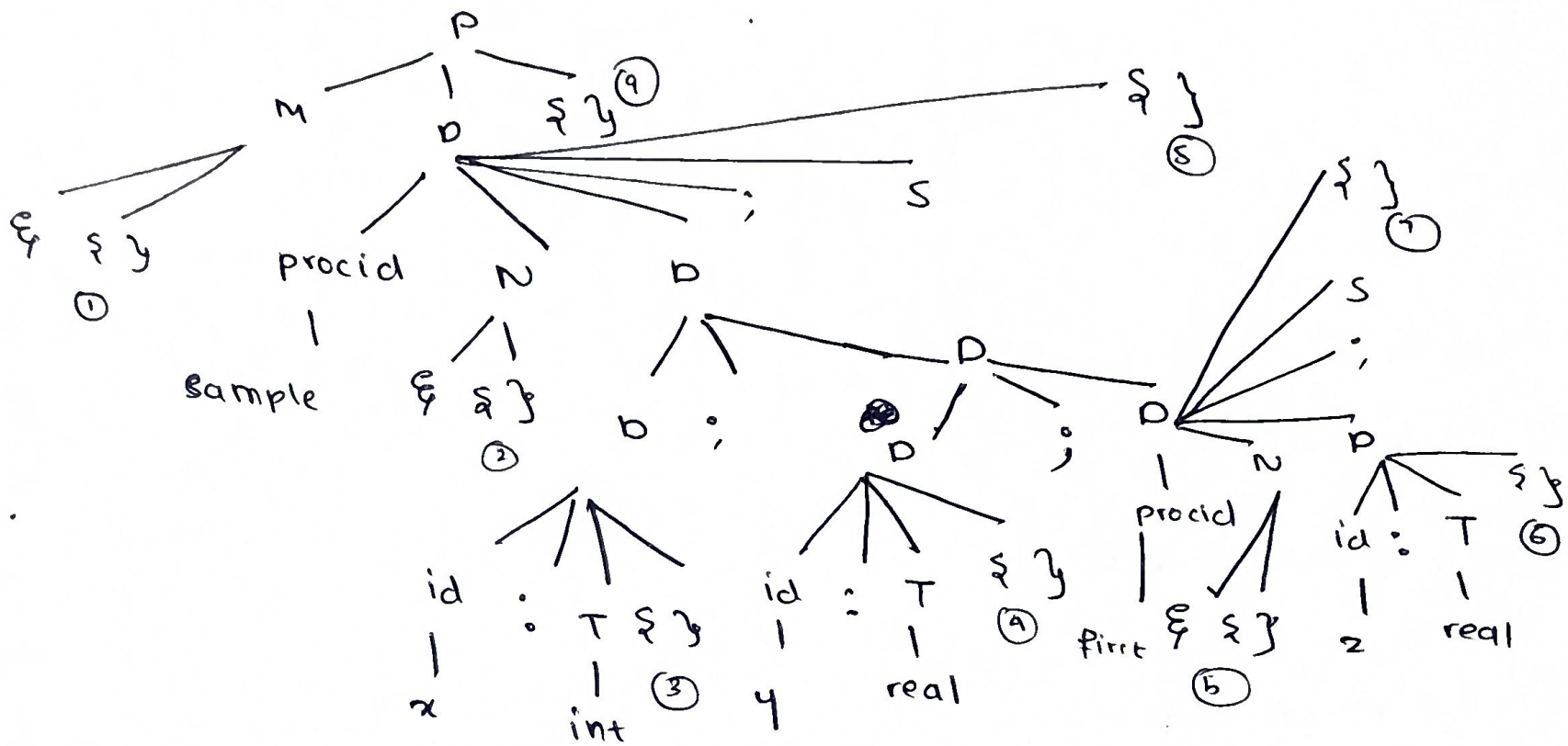
Perform DFS

1. offset = 0
2. T.type = int 3. T.width = 4
3. T.type = array (num.val, T1.type)
   = array (5, int)
4. T.width = num.val * T.width
   = 5 * 4 = 20
5. enter (id.name, T.type, offset)
6. offset = offset + T.width = 0 + 20 = 20
7. T.type = int
   T.width = 4
8. T.type = pointer (int)
   T.width = 4
9. enter (id.name, T.type, offset)
   offset = 20 + 4 = 24

a	array	0
b	pointer (int)	20

③ Example Nested Procedure Declaration

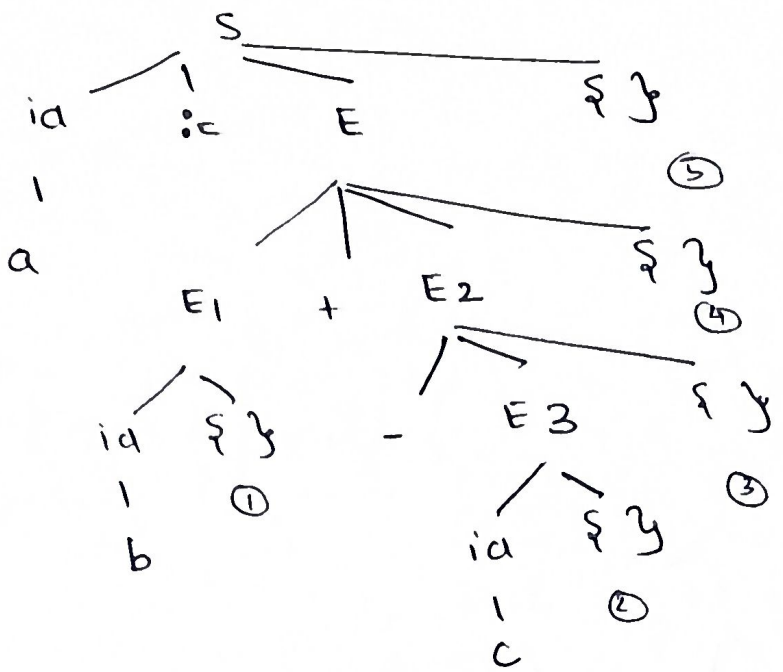
```
Procedure sample
    x : int
    y : real

    procedure sample first
        z : real
```



(do DFS and fill table)

④ Example : Assignment Statement

a = b + -c



Traverse DF

① E1.place = a

② E3.place = c

③ E3.place = t1

t1 := uminus c

④ E.place = t2

t2 := b + t1

⑤ ~~a.place~~ id.place = E.place

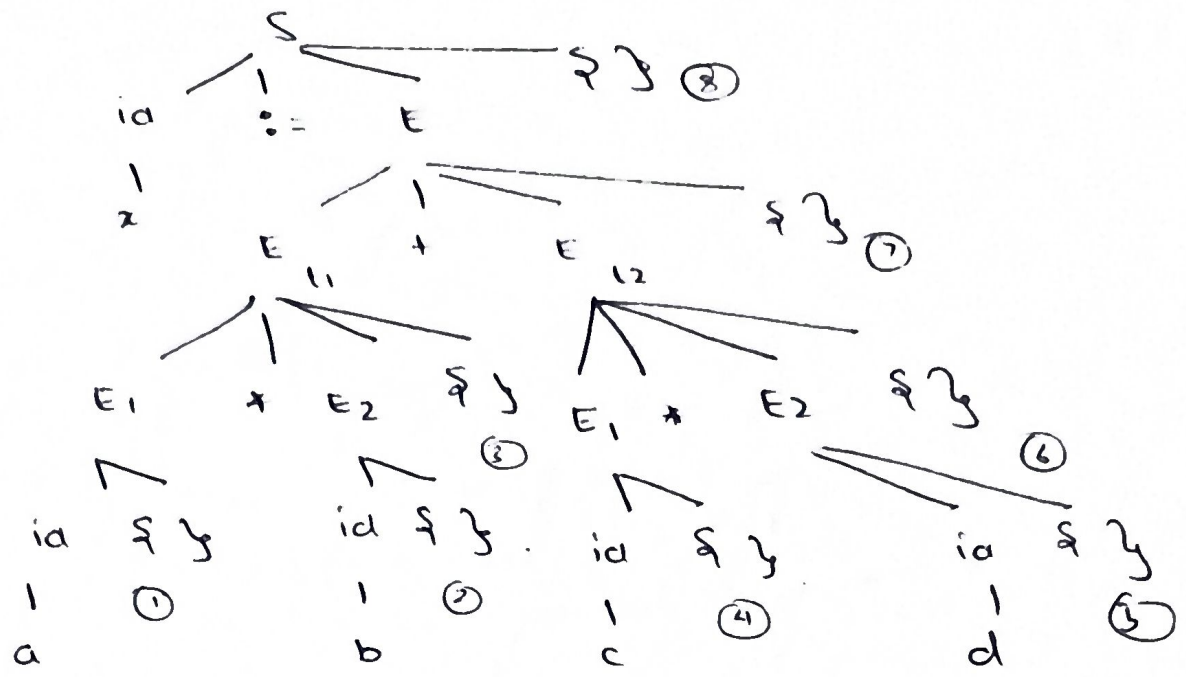
a = t2

TAC

t1 = uminus c t2 = b + t1 a = t2

(5.) Example : Arithmetic Expression

x := a * b + c * d



E1.place = a

E2.place = b

E.place = t1

$t_1 := a * b$

E1.place = c

E2.place = d

$t_2 := c * d$

E.place = t3

$t_3 = t_1 + t_2$

$a = t_3$

TAC

$t_1 = a * b$

$t_2 = c * d$

$t_3 = t_1 + t_2$

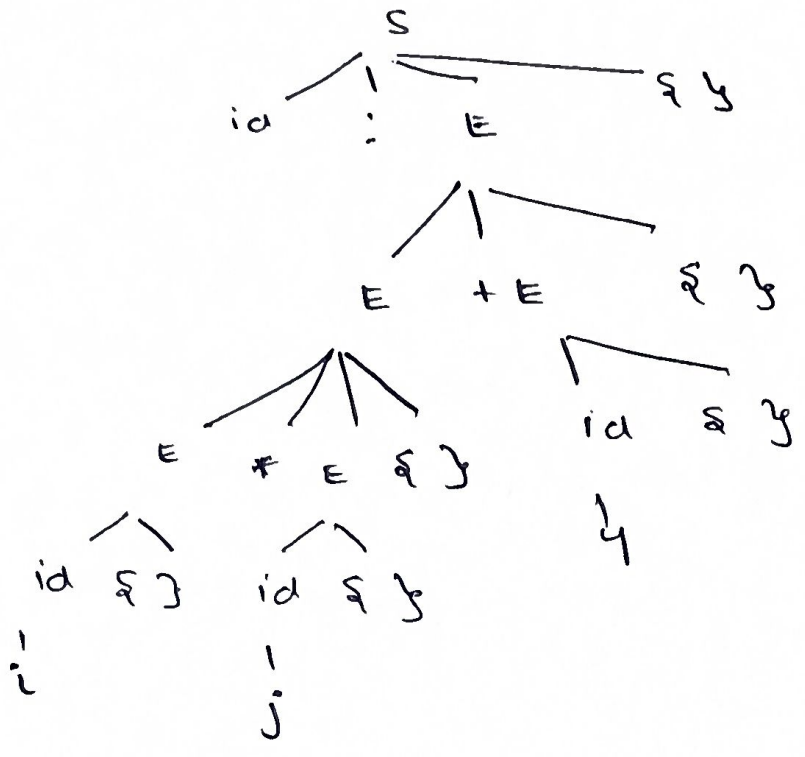
$a = t_3$

(6) Example : Assignment and Arithmetic Statements

real x, y

int i, j

x = y + i * j



E1.place = i

E2.place j

E1.type = int

E2.type = int

E.place = t1

t1 = i * j

E1.type - int

E.place = y

E2.type = real

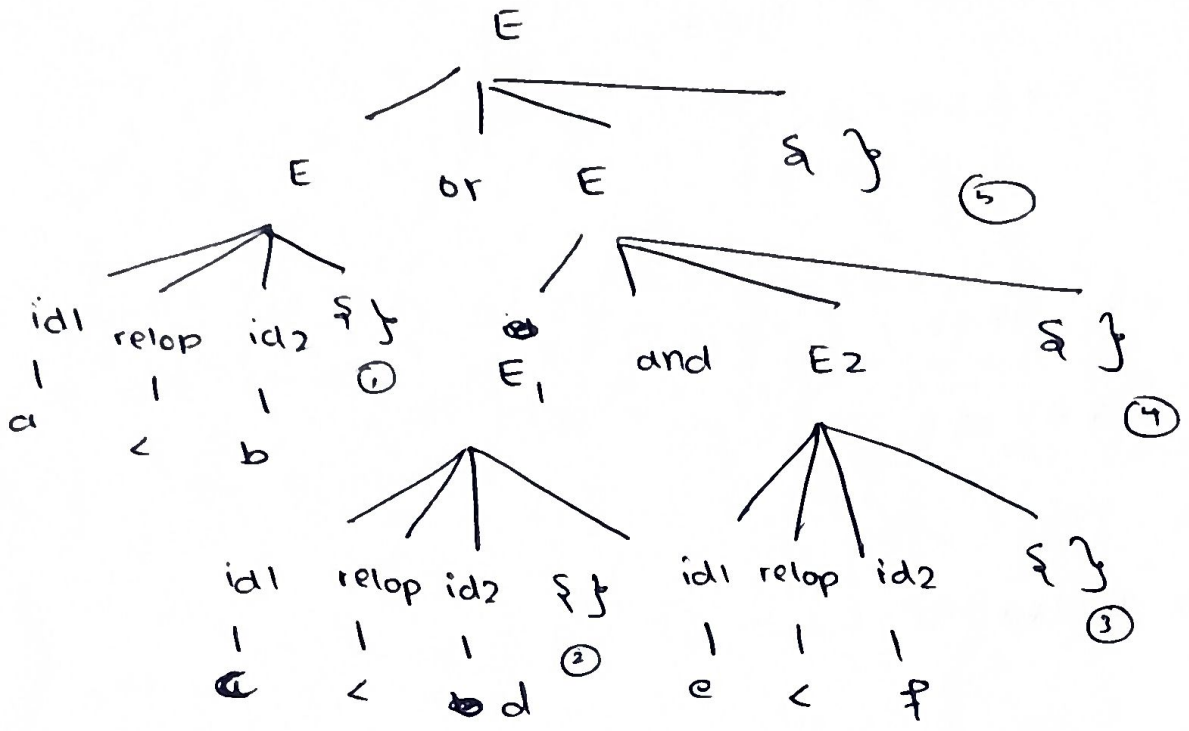
t2 = int to real t1

t3 = t1 + y

x = t3

① Example : Boolean Expressions

a < b or c < d and e < f



E.place = t1

```
100  if a<b goto 103
101  t1 = 0
102  goto 104
103  t1 = 1
104: if c<d goto 107      <-- E1.place = t2
105  t2 = 0
106  goto 108
107  t2 = 1
108  if e<f goto 111      <-- E2.place = t3
109  t3 = 0
110  goto 112
111  t3 = 1
112  t4 = t2 and t3
113: t5 = t1 or t4
```

(8) Examples for conditional and looping statements

For if-then statements:

E.true → inside if block

E.false → else block

example1 if a<b then

x = y + z * c

Ans if a<b goto E.true

goto E.false

E.true :

t1 = z * c

t2 = y + t1

x = t2

goto S.next

E.false : ——

S.next :

example2 if a<b then

x = y + z * c

else

x = y - z / c

if a<b goto E.true

goto E.false

E.true:

t1 = z * c

t2 = 4 + t1

x = t2

goto S.next

E.false :

t1 = z / c

t2 = 4 + t1

x = t2

S.next

★★★

example 3 while i<10 do

begin x = 0

c = c+1

end

Ans S.begin:

If i<10 goto E.true

goto E.false

E.true :

t1 = 0

x = t1

t2 = i+1

i = t2

goto S.begin

E.false

★★★

example 4 switch (i+j) {

case1 : x = y + z

break

case 2 : u = v + w

break

default : p = q + r

}

Ans

t1 = i+j

goto test

L1 : t2 = y + z

x = t2

goto next

L2 : t3 = v + w

u = t3

goto next

L3 : t4 = q + r

P = t4

goto next

test: If t1 = 1 goto L1

If t2 = 2 goto L2

goto L3

next :

example 5 while (i<n)

i = i+5

c = i * c

Ans origin:

If i<n goto L1

goto L2

L1: t1 = i+5

i = t

t2 = i * c

c = t2 goto origin

L2 :

Example 6 if (a<b && x==4)

z = c * d / - f

a = f - id

Ans

if a<b goto L1

goto E.false

L1: if x==4 goto E.true

goto E.false

E.true

t1 = -f

t2 = c * d

t3 = t2 / t1

z = t3

t4 = f - d

a = t4

E.false

Example 7 a<b and b>c and d<c

if a<b goto L1

goto E.false

L1: if b>c goto L2

goto E.false

L2: if d<c goto E.true

goto E.false

E.true:

E.false: