# Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering &Technology

ESTD:1995

## *Department of Computer Science & Engineering*

**SUBJECT:** *Automata and Compiler Design*

Subject Code:***22PC1CB303***

***Topic Name: Phases of a compiler***
***III year-I sem, sec: B and D***


Dr. M.Gangappa

Associate Professor

*Email: gangappa_m@vnrvjiet*


*<Web link of your created resource if any>*

# Syllabus

Semantics: Syntax directed translation, S-attributed and L-attributed grammars, Intermediate code – abstract syntax tree, translation of simple statements and control flow statements.

# Agenda

ESTD:1995

**4.1**

**1.Syntax-Directed Definitions and Translation schemes**

**2.Synthesized and inherited attributes**

**3.SDD for desk calculator**

**4.Annotated parse tree**

**5.Attributed grammar**

**6.SDD for inherited attributes**

**7.Dependency graph**

**8.Abstract syntax trees and DAG**

**9. Syntax tree creation using SDD**

# Agenda

ESTD:1995

**4.2**

1. Intermediate code generation - benefits
2. Intermediate code representations – syntax tree, DAG, post fix notation and Three address code forms
3. Types of three address code
4. Implementation of three address code-quadruples, tripuls and indirect triples
5. Three address code for programming language constructs
   - 5.1 Three address code for assignment statement using SDD
   - 5.2 Three address code for Boolean expression using SDD
   - 5.3 Three address code for control flow statements using SDD

# Syntax-Directed Definitions and Translation Schemes

ESTD:1995

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.

# Syntax-Directed Definitions and Translation Schemes

ESTD:1995

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

# Syntax-Directed Definitions

ESTD:1995

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.

- SYNTHESIZED attributes are computed from the values of the CHILDREN of a node in the parse tree.
- INHERITED attributes are computed from the attributes of the parents and/or siblings of a node in the parse tree.

# Synthesized Attributes

A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition

A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

# Syntax-Directed Definition for Desk calculator -- Example

| **Production** | **Semantic Rules** |
|---|---|
| $L \rightarrow E$ **return** | print(E.val) |
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | $E.val = T.val$ |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | $T.val = F.val$ |
| $F \rightarrow ( E )$ | $F.val = E.val$ |
| $F \rightarrow$ **digit** | F.val = **digit**.lexval |

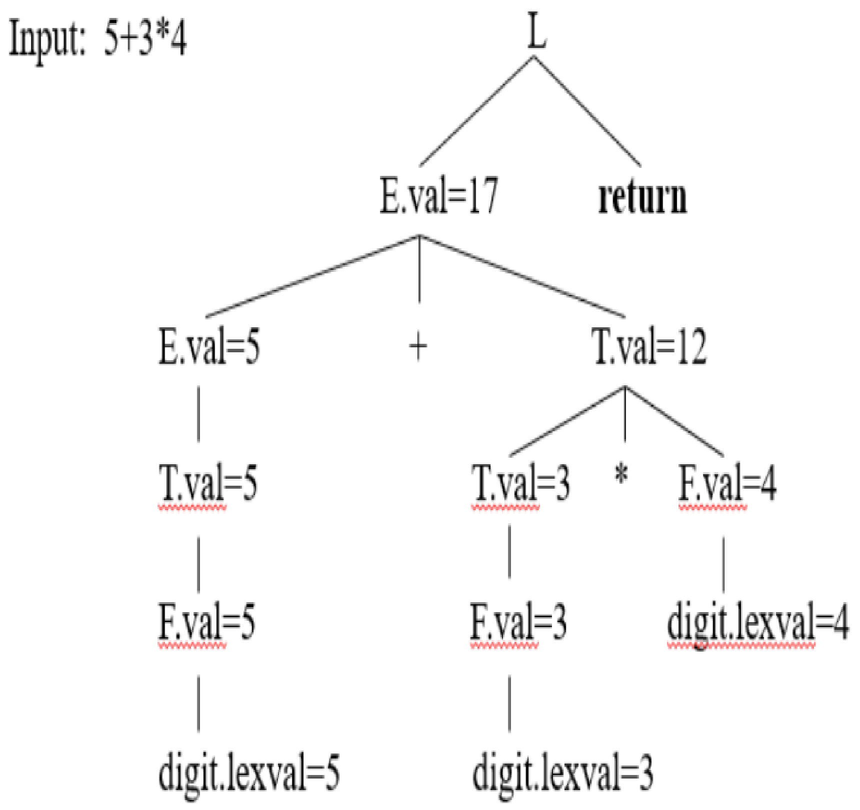
- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

# Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

# Parse tree for 5 + 3 * 5

## Annotated Parse Tree



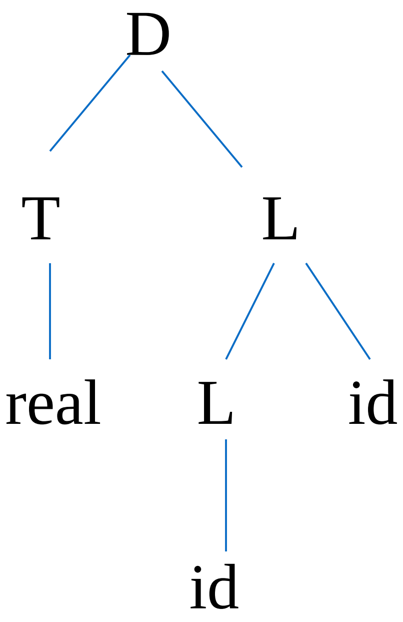
# Syntax-Directed Definition – Inherited Attributes

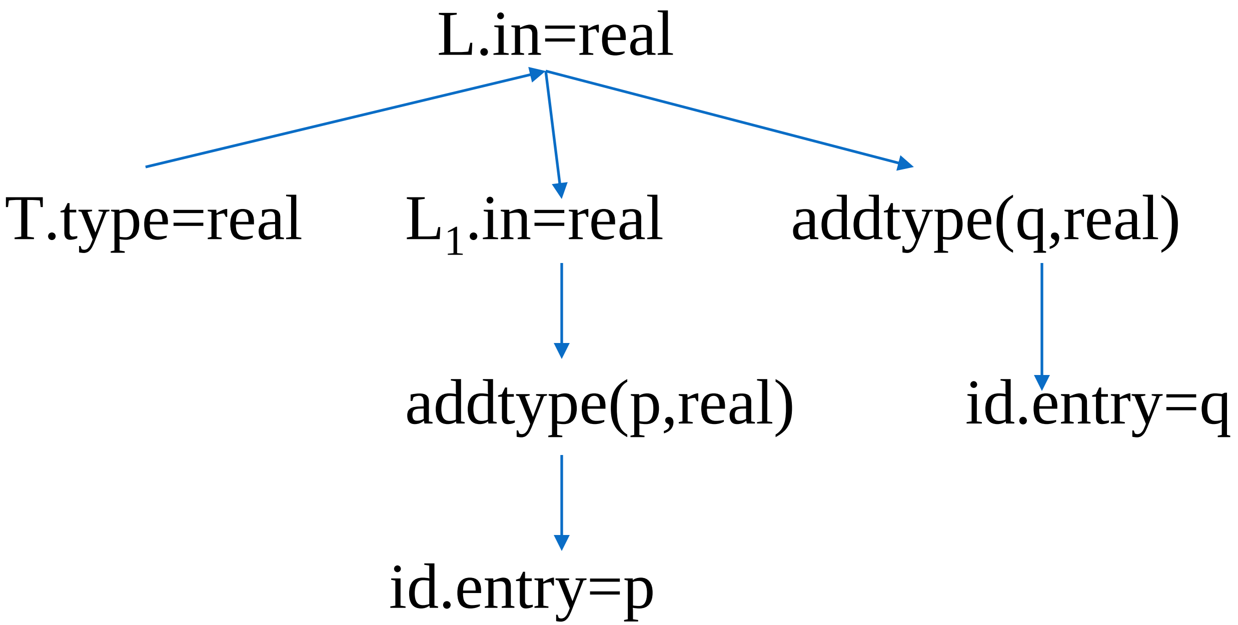
| **Production** | **Semantic Rules** |
|---|---|
| $D \rightarrow T\ L$ | L.in = T.type |
| $T \rightarrow$ **int** | T.type = integer |
| $T \rightarrow$ **real** | T.type = real |
| $L \rightarrow L_1$ **id** | $L_1$.in = L.in, addtype(**id**.entry,L.in) |
| $L \rightarrow$ **id** | addtype(**id**.entry,L.in) |

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in.*

# A Dependency Graph – Inherited Attributes

Input: `real p q`




*parse tree*

*dependency graph*

# S-Attributed Definitions

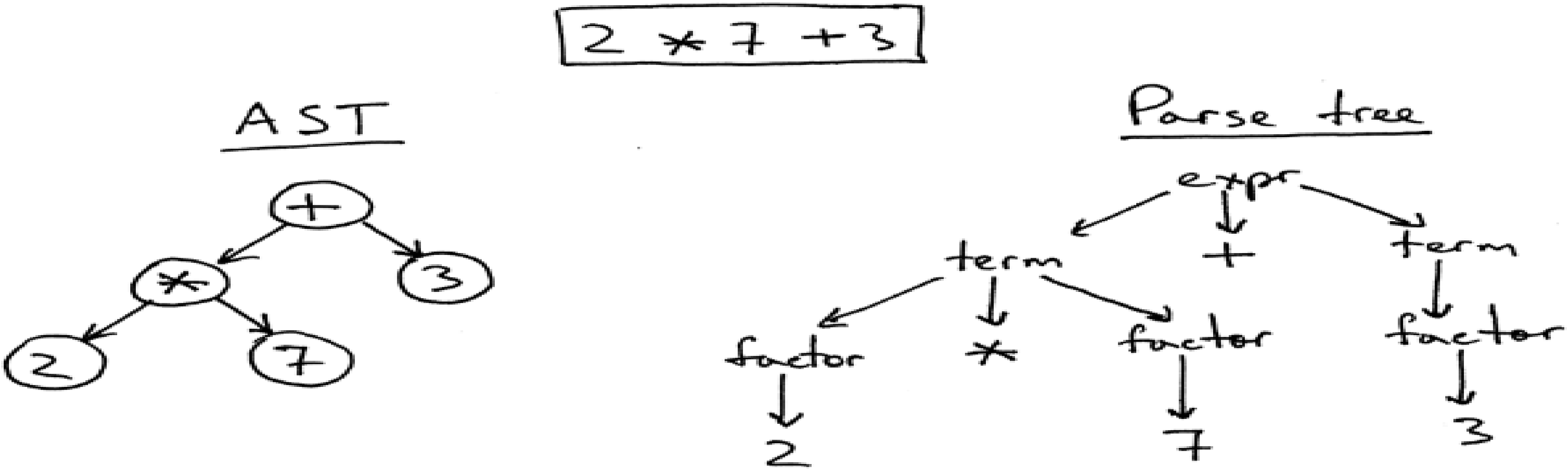
- We will look at two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

# Abstract Syntax tree

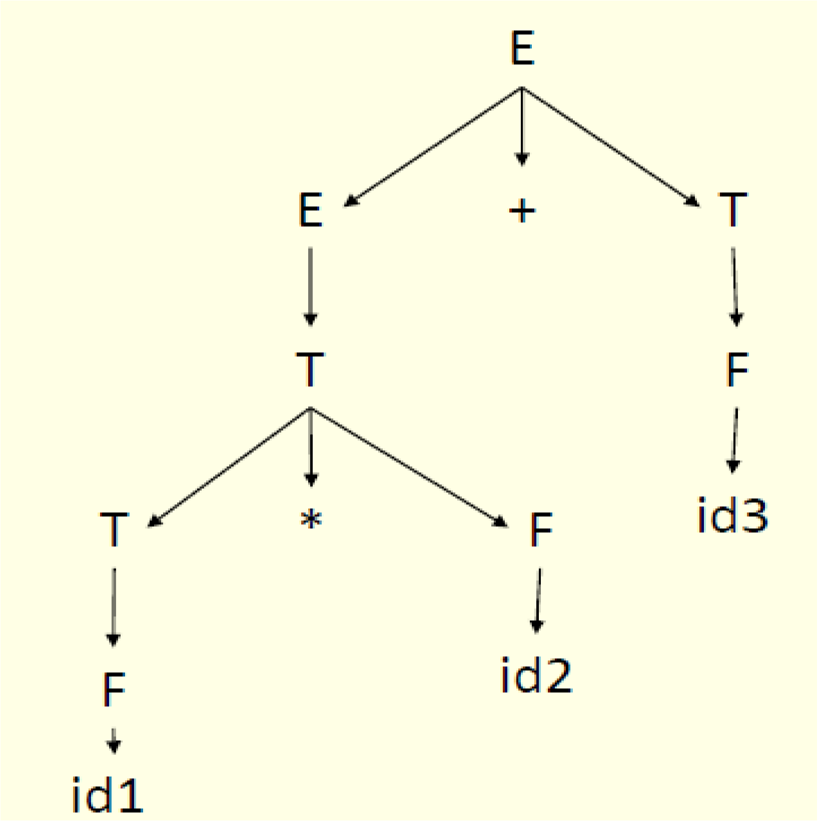
- Condensed form of parse tree
- useful for representing language constructs.
- A **concrete syntax** tree represents the source text exactly in parsed form.
- The **abstract syntax tree** is the result of simplifying the concrete syntax tree
- Uses of AST
- AST is used in Intermediate code representation(IR)

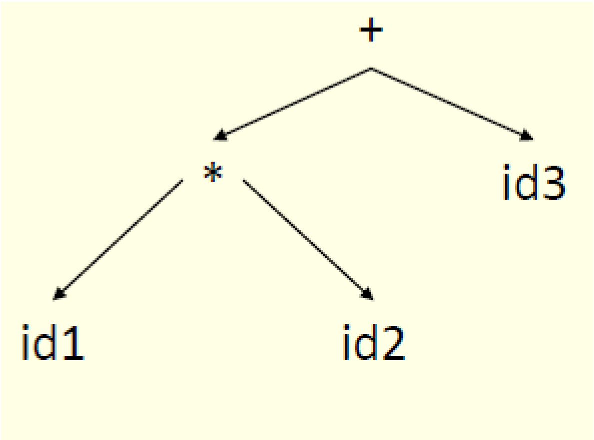
# Abstract Syntax tree-definition

An ***abstract syntax tree*** (*AST*) is a tree that represents the abstract syntactic structure of a language construct where each interior node and the root node represents an operator, and the children of the node represent the operands of that operator.



# Abstract Syntax tree …



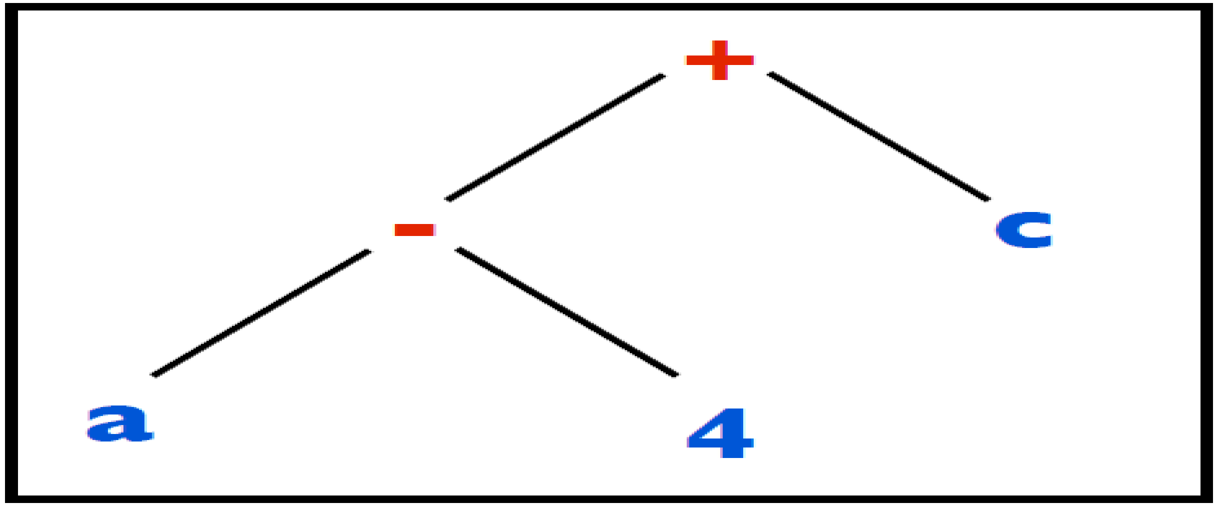



# Abstract Syntax tree …

ESTD:1995

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $E \rightarrow E_1 + T$ | |
| 2) | $E \rightarrow E_1 - T$ | |
| 3) | $E \rightarrow T$ | |
| 4) | $T \rightarrow ( E )$ | ? |
| 5) | $T \rightarrow \textbf{id}$ | |
| 6) | $T \rightarrow \textbf{num}$ | |

a−4+c



# Constructing Abstract Syntax Tree for expression

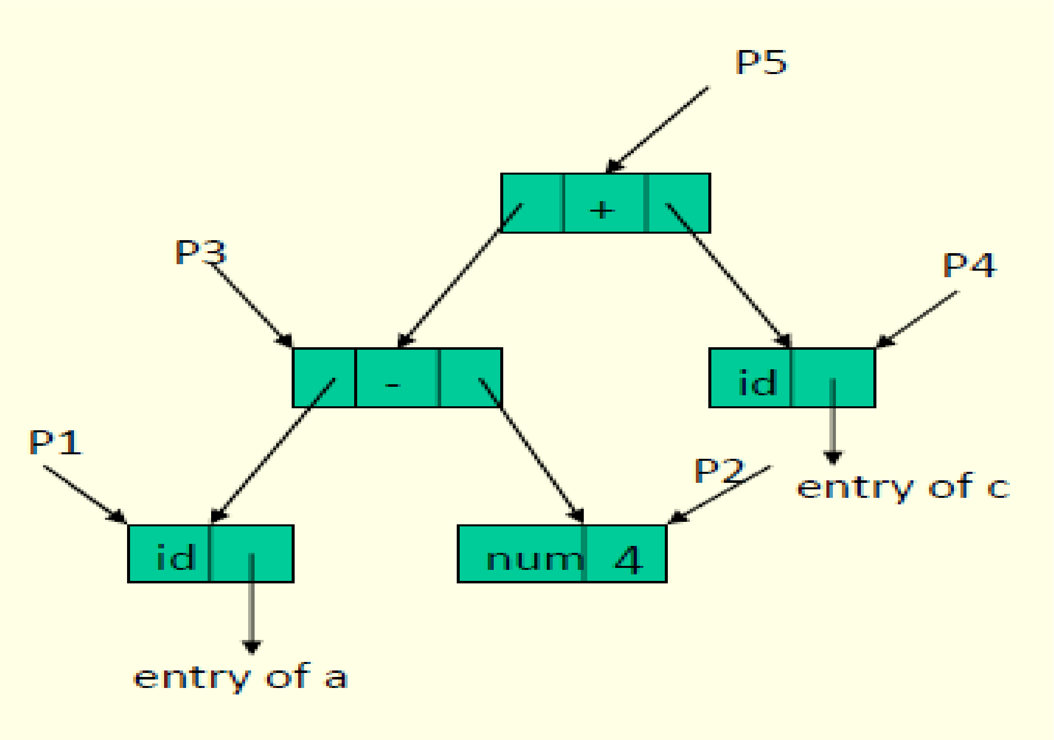
Each node can be represented as a record

- *operators*: one field for operator, remaining fields ptrs to operands mknode(op,left,right )
- *identifier*: one field with label id and another ptr to symbol table mkleaf(id,entry)
- *number*: one field with label num and another to keep the value of the number  mkleaf(num,val)

# Example

The following sequence of function calls creates a parse tree for a- 4 + c

P1 = mkleaf(id, entry.a)
P2 = mkleaf(num, 4)
P3 = mknode(-, P1, P2)
P4 = mkleaf(id, entry.c)
P5 = mknode(+, P3, P4)

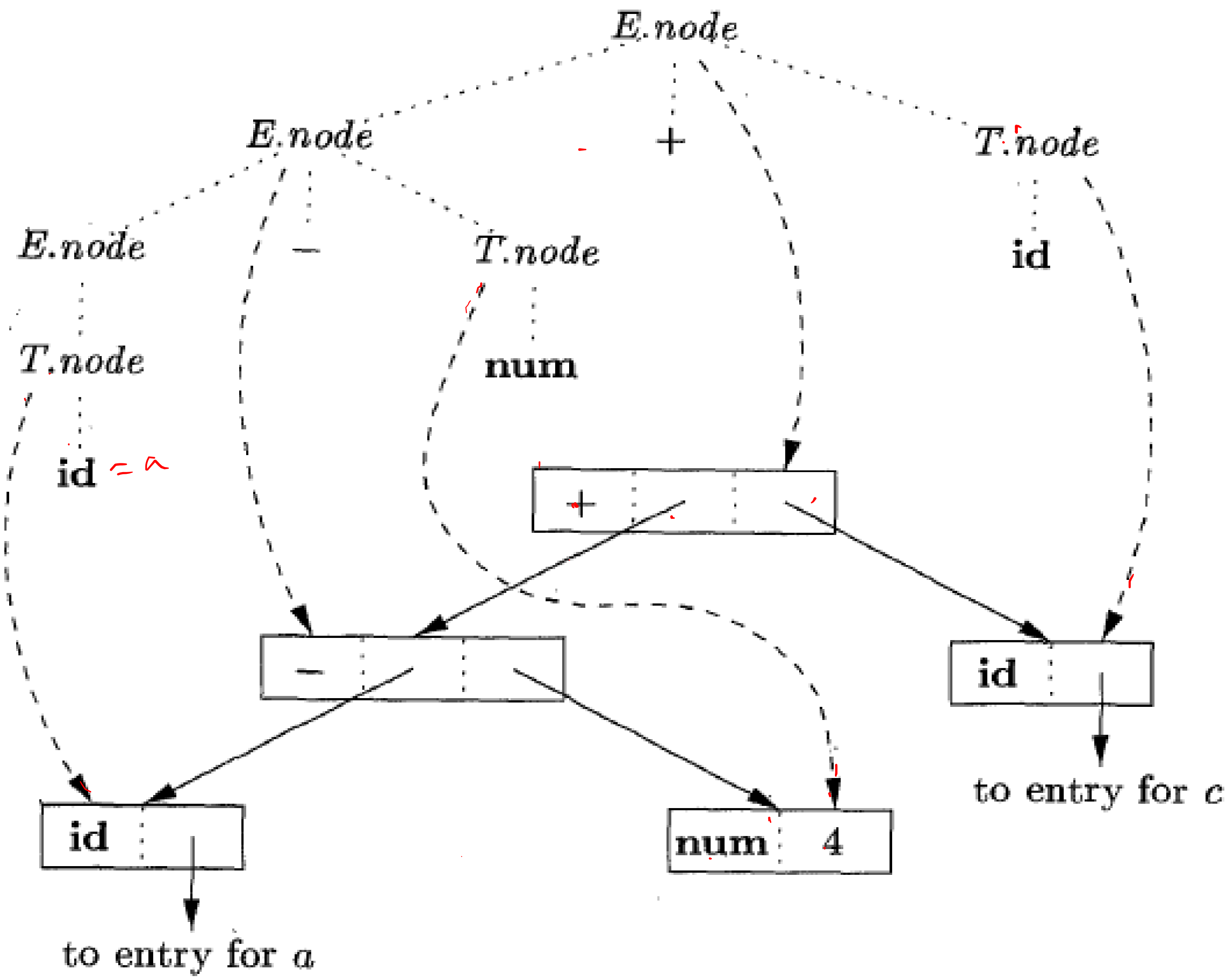


# A syntax directed definition for constructing syntax tree or DAG(directed acyclic graph)

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E1 + T$ | E.ptr = mknode(+, E1.ptr, T.ptr) |
| $E \rightarrow T$ | E.ptr = T.ptr |
| $T \rightarrow T1 * F$ | T.ptr := mknode(*, T1.ptr, F.ptr) |
| $T \rightarrow F$ | T.ptr := F.ptr |
| $F \rightarrow (E)$ | F.ptr := E.ptr |
| $F \rightarrow id$ | F.ptr := mkleaf(id, entry.id) |
| $F \rightarrow num$ | F.ptr := mkleaf(num,val) |

# Syntax Tree Using SDD

for $\boxed{a-4+c}$



# A syntax directed definition for constructing DAG(directed acyclic graph)

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E1 + T$ | E.ptr = mknode(+, E1.ptr, T.ptr) |
| $E \rightarrow T$ | E.ptr = T.ptr |
| $T \rightarrow T1 * F$ | T.ptr := mknode(*, T1.ptr, F.ptr) |
| $T \rightarrow F$ | T.ptr := F.ptr |
| $F \rightarrow (E)$ | F.ptr := E.ptr |
| $F \rightarrow id$ | F.ptr := mkleaf(id, entry.id) |
| $F \rightarrow num$ | F.ptr := mkleaf(num,val) |

# Draw the DAG for the expression

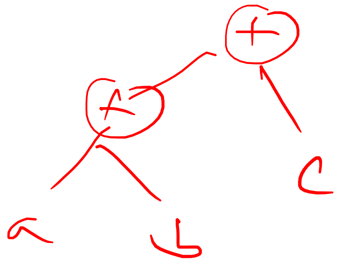
ESTD:1995

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

Monday

(18/01/2021)

for a+b+c



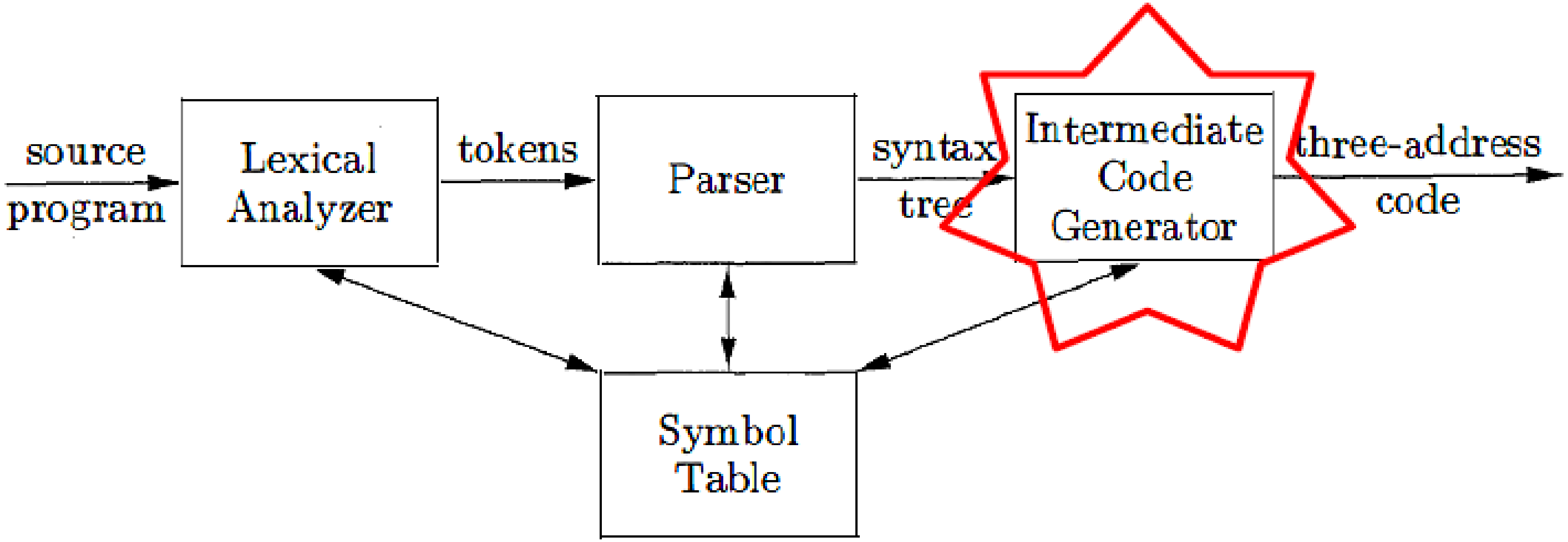

instructions

$p_1$ = mkleaf(id, entry.a)
$p_2$ = mkleaf(id, entry.b)
$p_3$ = mknode(+, $p_1$, $p_2$)
$p_4$ = mkleaf(id, entry.c)
$p_5$ = mknode(+, $p_3$, $p_4$)

1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}a)$
2) $p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}a) = p_1$
3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}b)$
4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}c)$
5) $p_5 = \mathit{Node}('-', p_3, p_4)$
6) $p_6 = \mathit{Node}('*', p_1, p_5)$
7) $p_7 = \mathit{Node}('+', p_1, p_6)$
8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}b) = p_3$
9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}c) = p_4$
10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}d)$
12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

Steps for constructing the DAG

# Intermediate code generation




The front end translates a source program into an intermediate representation from which the back end generates target code.

# Intermediate code generation

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.

2. A machine-independent code optimizer can be applied to the intermediate representation.

# Intermediate code representations

Three ways of intermediate representation:

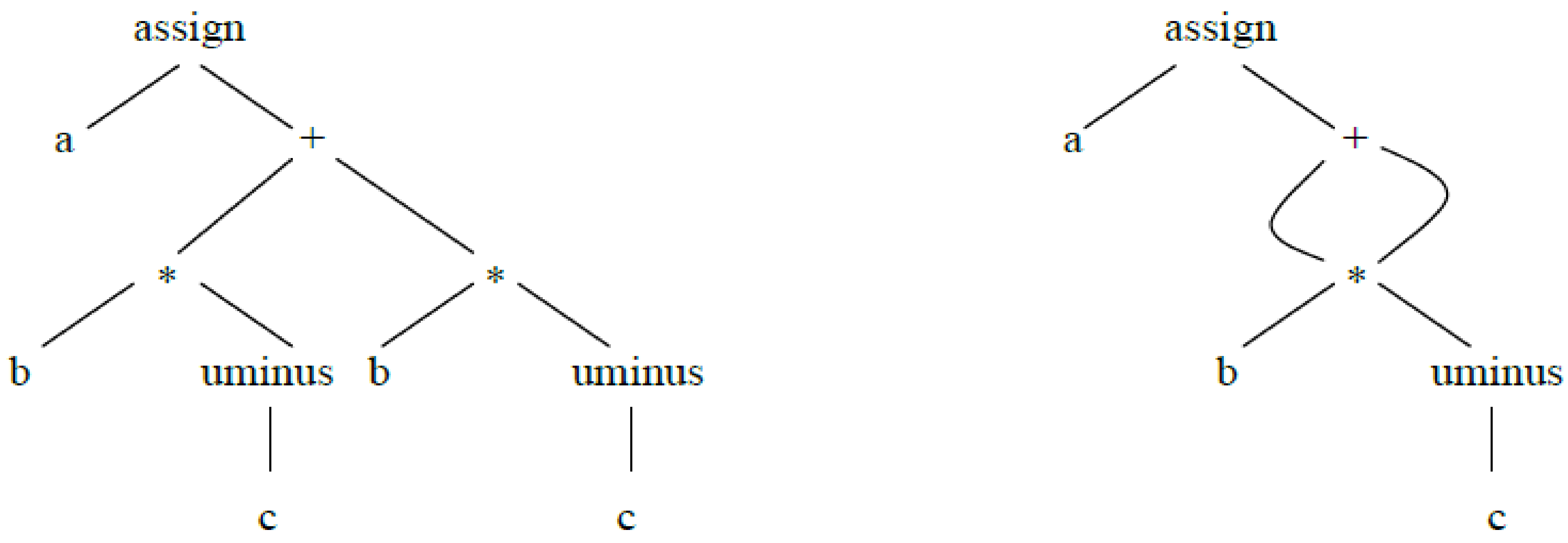
- Syntax tree and DAG
- Postfix notation
- Three address code

# Syntax tree and DAG

A syntax tree depicts the natural hierarchical structure of a source program. A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.

A syntax tree and dag for the assignment statement a : = b * - c + b * - c are as follows:

# Syntax tree and DAG




(a) Syntax tree (b) Dag

# Postfix notation

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children.

The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

# Three-Address Code:

Three-address code is a sequence of statements of the general form

$$x := y \text{ op } z$$

where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean valued data. Thus a source language expression like x+ y*z might be translated into a sequence

$$t1 := y * z$$
$$t2 := x + t1$$

where t1 and t2 are compiler-generated temporary names

# Three-address code corresponding to the syntax tree and DAG the statement : a : = b * - c + b * - c

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

**(a) Code for the syntax tree**

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

**(b) Code for the dag**

# Types of Three-Address Statements

ESTD:1995

**Assignments:**

Two possible forms:

1. ❑ x:= y op z where op is a binary arithmetic or logical operation,
2. ❑ x:= op y where op is a unary operation (minus, negation)

**Copy statements:**

3. ❑ They have the form x := y

**Un-conditional jumps:**

4. ❑ They have the form goto L where L is a symbolic label of a statement.

**Conditional jumps:**

5 ❑ They have the form if x relop y goto L where statement L is executed if x and y are in relation relop.

(1) x = y op z
(2) x = op z
(3) x = y
(4) goto L
(5) if x relop y goto L

# Types of Three-Address Statements

**Procedure calls:**

They have the form

param $x_1$

param $x_2$

.

.

.

param $x_n$

call p, n corresponding to the procedure call $p(x_1, x_2, ..., x_n)$

**Return statement:**

They have the form return y where y representing a returned value is optional.

# Types of Three-Address Statements

ESTD:1995

**Indexed assignments**:

- They have the form x := y[i]  or   x[i] := y

**Address assignments**:

- They have the form x := &y which sets x to the location of y.

**Pointer assignments**:

- They have the form
  - x := *y where y is a pointer and which sets x to the value pointed to by y
  - *x := y which changes the location of the value pointed to by x.

# Data structures to hold Three address code
# OR
# Three address code implementation Techniques

Three address code implementations methods:

- ❑ Quadruples
- ❑ Triples
- ❑ Indirect triples

# Three address code implementation Techniques

In quadruples representation, each instruction is split into the following 4 different fields : **op, arg1, arg2, result**

The op field is used for storing the internal code of the operator. The arg1 and arg2 fields are used for storing the two operands used. The result field is used for storing the result of the expression.

**For example: x = y + z**

- ❑ Op (+)
- ❑ Arg1 (y)
- ❑ Arg2 (z)
- ❑ Result (x)

**Triples-**

In triples representation,

- References to the instructions are made.
- Temporary variables are not used.

# Indirect Triples-

- This representation is an enhancement over triples representation.
- It uses an additional instruction array to list the pointers to the triples in the desired order.
- Thus, instead of position, pointers are used to store the results.
- It allows the optimizers to easily re-position the sub-expression for producing the optimized code.

# Translate the following expression to quadruple, triple and indirect triple: a + b x c / e ↑ f + b x a

Three Address Code for the given expression is-

T1 = e ↑ f
T2 = b x c
T3 = T2 / T1
T4 = b x a
T5 = a + T3
T6 = T5 + T4

## Quadruple Representation-

| Location | Op | Arg1 | Arg2 | Result |
|---|---|---|---|---|
| (0) | ↑ | e | f | T1 |
| (1) | x | b | c | T2 |
| (2) | / | T2 | T1 | T3 |
| (3) | x | b | a | T4 |
| (4) | + | a | T3 | T5 |
| (5) | + | T5 | T4 | T6 |

# Triple Representation-

| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | ↑ | e | f |
| (1) | x | b | c |
| (2) | / | (1) | (0) |
| (3) | x | b | a |
| (4) | + | a | (2) |
| (5) | + | (4) | (3) |

# Indirect Triple Representation-

ESTD:1995

| | Statement |
|---|---|
| 35 | (0) |
| 36 | (1) |
| 37 | (2) 1869-9391 |
| 38 | (3) |
| 39 | (4) |
| 40 | (5) |

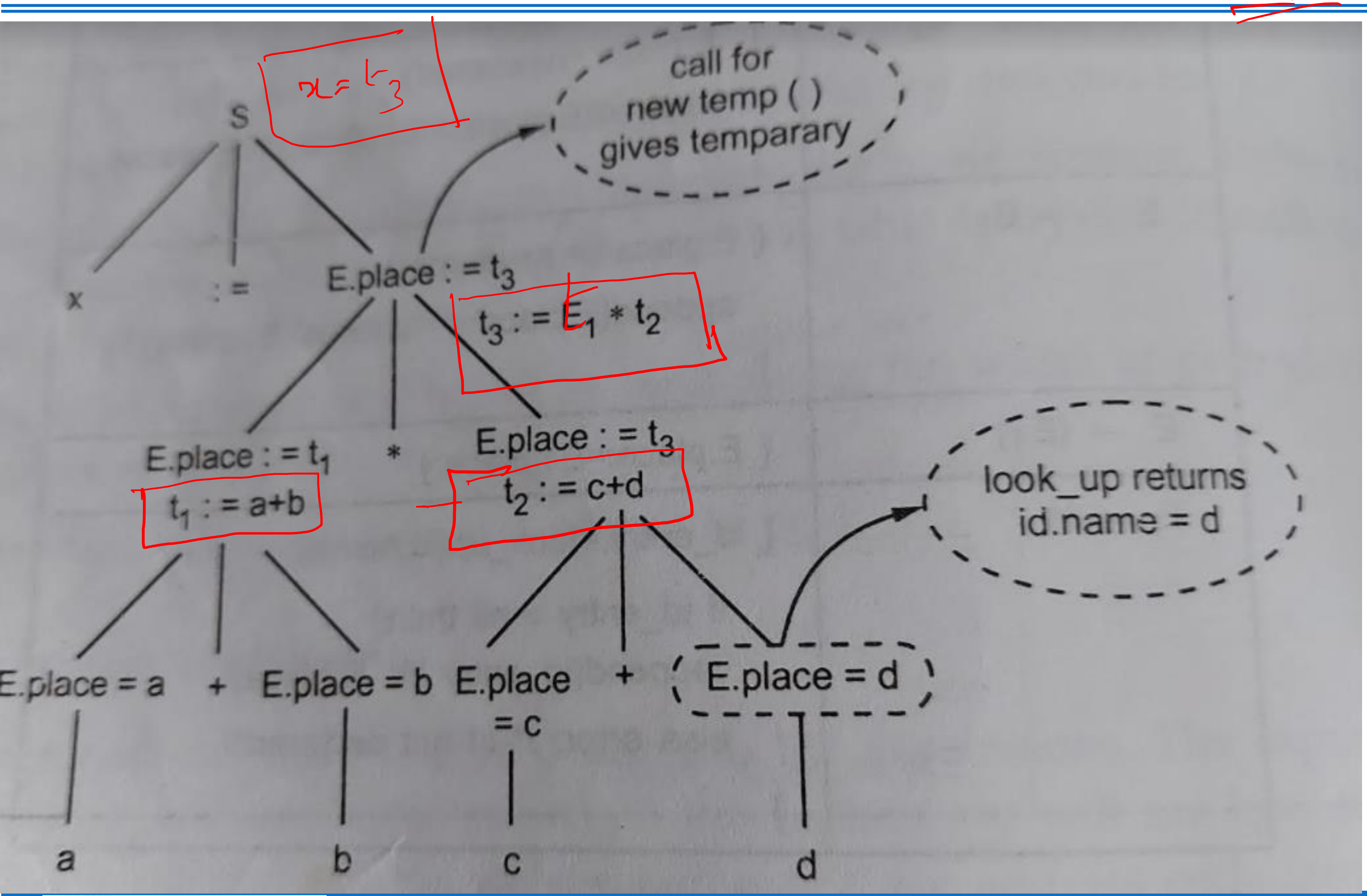
| Location | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | ↑ | e | f |
| (1) | x | b | e |
| (2) | / | (1) | (0) |
| (3) | x | b | a |
| (4) | + | a | (2) |
| (5) | + | (4) | (3) |

# Homework problem :

Implement a three address code for a = b * -c + b * -c using quadrapuples,triples and indirect tripples

| Production Rule | Semantic actions |
| --- | --- |
| S → id:= E | { id_entry:=look_up(id.name); if id_entry ≠ nil then gen (id_entry ':=' E.place) else error; /* id not declared*/ } |
| E → $E_1$ + $E_2$ | { E.place:=newtemp(); gen (E.place':='$E_1$.place '+' $E_2$.place) } |
| E → $E_1$ * $E_2$ | { E.place:=newtemp(); gen (E.place':='$E_1$.place '*' $E_2$.place) } |
| E → − $E_1$ | { E.place := newtemp(); gen (E.place':=' 'uminus' $E_1$.place) } |
| E → ($E_1$) | { E.place:=E1.place } |
| E → id | { id_entry:=look_up(id.name); if id_entry ≠ nil then E.place := id_entry else error; /* id not declared*/ } |

# Three address code for $x = (a+b) * (c+d)$




3AC

$t_1 = a+b$

$t_2 = c+d$

$t_3 = t_1 * t_2$

$x = t_3$

# Three address code(3AC) for Boolean expressions using SDD

Production | Semantic

| | |
|---|---|
| E→E1 or E2 | { E.place : = newtemp; gen ( E.place ': =' E1.place 'or' E2.place ) } |
| E→E1 and E2 | { E.place : = newtemp; gen( E.place ': =' E1.place 'and' E2.place ) } |
| E→not E1 | { E.place : = newtemp; gen( E.place ': =' 'not' E1.place ) } |
| E→ ( E1 ) | { E.place : = E1.place } |
| E→id1 relop id2 | { E.place : = newtemp; gen ( 'if' id1.place relop.op id2.place 'goto' nextstat + 3); gen ( E.place ': =' '0' ); gen ('goto' nextstat +2); gen( E.place ': =' '1') } |
| E→true | { E.place : = newtemp; gen ( E.place ': =' '1') } |
| E→false | { E.place : = newtemp; gen ( E.place ': =' '0') } |

# Translation scheme using a numerical representation for booleans

```
E→E1 orE2          { E.place : = newtemp;
                   emit( E.place ': =' E1.place 'or' E2.place ) }
E→E1 andE2         { E.place : = newtemp;
                   emit( E.place ': =' E1.place 'and' E2.place ) }
E→notE1            { E.place : = newtemp;
                   emit( E.place ': =' 'not' E1.place ) }
E→ ( E1)           { E.place : = E1.place }

E→id1 relop id2    { E.place : = newtemp;
        emit( 'if' id1.place relop.op id2.place 'goto' nextstat + 3); emit( E.place ': =' '0' );
                   emit('goto' nextstat +2);
                   emit( E.place ': =' '1') }
E→true             { E.place : = newtemp;
                   emit( E.place ': =' '1') }
E→false            { E.place : = newtemp;
                   emit( E.place ': =' '0') }
```

## Example: translating $(a < b$ **or** $c < d$ **and** $e < f)$ into 3AC using SDD

ESTD:1995

$E \rightarrow id_1 \; relop \; id_2$

- {E.place := newtemp();
- gen("if", $id_1$.place,relop.op, $id_2$.place,"goto",nextstat+3);
- gen(E.place,":=","0");
- gen("goto",nextstat+2);
- gen(E.place,":=","1");}

### Three address code

a<b

```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1 /* true */
104: if c < d goto 107
105: t2 := 0 /* false */
106: goto 108
107: t2 := 1
108: if e < f goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
113: t3 := t1 or t4
```

t1

c<d

t2

e<f

t3

and

or

ESTD:1995

Generate a three address code for x =a && b ||c

# Three address code for flow  of control statements

ESTD:1995

| Production | Semantic Rules |
|---|---|
| S → if E then S1 | E.true := newlabel()<br>E. false := s.next<br>S1.next := s.next<br>S.code := E.code \|\| gen(E.true ":") \|\| s1.code |
| S → if E then S1 else S2 | E.true := newlabel()<br>E. false := newlabel()<br>S1.next := s.next<br>S2.next := s.next<br>S.code := E.code \|\| gen(E.true ":") \|\| s1.code \|\|<br>gen("goto" S.next) \|\|<br>gen(E.false ":") \|\| s2.code |
| S → while E then S1 | S.begin := newlabel()<br>E.true := newlabel()<br>E. false := s.next<br>S1.next := S.begin<br>S.code := gen(S.begin ":") \|\| E.code \|\|<br>gen(E.true ":") \|\| s1.code \|\|<br>gen("goto" S.begin) |

# flow of control statements

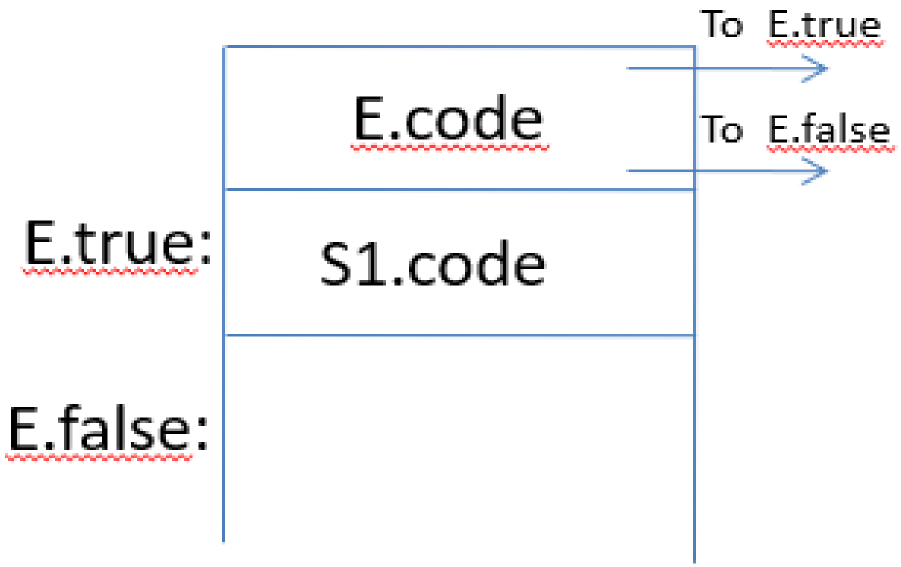



fig (a): if - then

# flow of control statements

ESTD:1995

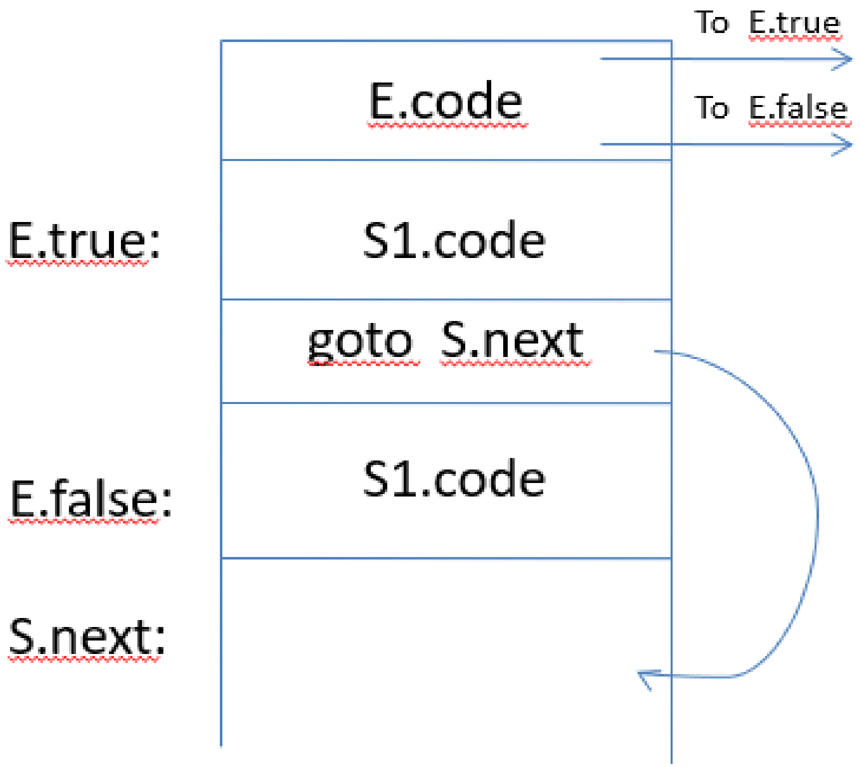



fig (b): if - then - else

# flow of control statements

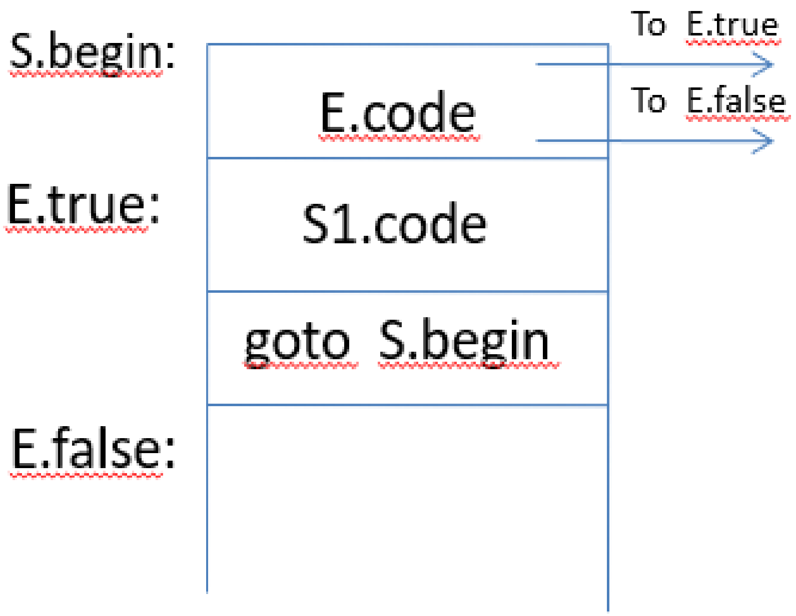



fig (c): while - do

HW submit on or before 23/01/2021

(1) Write a Three Address code for the following code fragment using SDD

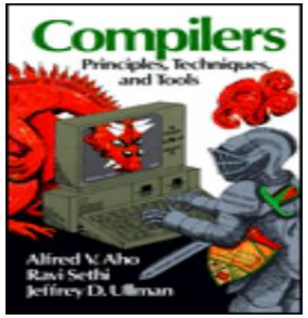
```
c = 0
do {
    if (a < b)
    {
        x = x + 1
    }
    else {
        x = x - 1
    }
    c = c + 1
} while (c < 5)
```

(2) Write a 3AC for

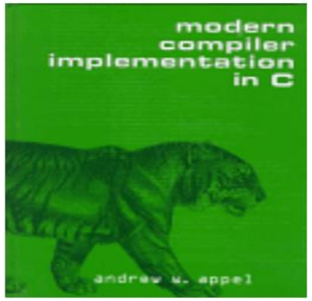
```
for (i = 1 ; i < 10 ; i++)
{
    a[i] = x * 5
}
```

# References

- Compilers principles ,tools and techniques by Aho, Sethi, and Ullman, Chapters 1, 2, 3




- S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufman, 1997
- Andrew W. Appel : Modern Compiler Implementation in C



# End of Unit -3

THANK YOU