PUNE VIDYARTHI GRIHA's
COLLEGE OF ENGINEERING, NASHIK.

O"*SYNTAX DIRECTED TRANSLATION*"

PREPARED BY :

PROF. ANAND N. GHARU

ASSISTANT PROFESSOR

COMPUTER DEPARTMENT

SUBJECT – COMPILER (BE COMPUTER SPPU-2019)

Syntax-Directed Translation

Syntax-Directed Translation

1. We associate information with the programming language constructs by attaching attributes to grammar symbols.

2. Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

3. 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.

4. An attribute may hold almost any thing.
   - a string, a number, a memory location, a complex record.

Syntax-Directed Definitions and Translation Schemes

1. When we associate semantic rules with productions, we use two notations:
   - **Syntax-Directed Definitions**
   - **Translation Schemes**

A. 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.

B. 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 Translation

- Conceptually with both the syntax directed translation and translation scheme we
  - Parse the input token stream
  - Build the parse tree
  - Traverse the tree to evaluate the semantic rules at the parse tree nodes.

Input string ⟶ parse tree ⟶ dependency graph ⟶ evaluation order for semantic rules

Conceptual view of syntax directed translation

Syntax-Directed Definitions

1. 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.

2. The value of an attribute at a parse tree node is defined by the semantic rule associated with a production at that node.
3. The value of a **synthesized attribute** at a node is computed from the values of attributes at the children in that node of the parse tree
4. The value of an **inherited attribute** at a node is computed from the values of attributes at the siblings and parent of that node of the parse tree

Syntax-Directed Definitions

Examples:

Synthesized attribute : E→E1+E2 { E.val =E1.val + E2.val}

Inherited attribute :A→XYZ {Y.val = 2 * A.val}

1. *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.

2. This *dependency graph* determines the evaluation order of these semantic rules.

3. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

1. A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

2. Values of Attributes in nodes of annotated parse-tree are either,
   – initialized to constant values or by the lexical analyzer.
   – determined by the semantic-rules.

3. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

4. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

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

where *f* is a function and *b* can be one of the followings:

➔ *b* is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

OR

➔ *b* is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

Syntax-Directed Definition -- Example

Production	**Semantic Rules**
$L \rightarrow E$ **n**	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

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
3. Terminals are assumed to have synthesized attributes only. Values for attributes of terminals are usually supplied by the lexical analyzer.
4. The start symbol does not have any inherited attribute unless otherwise stated.

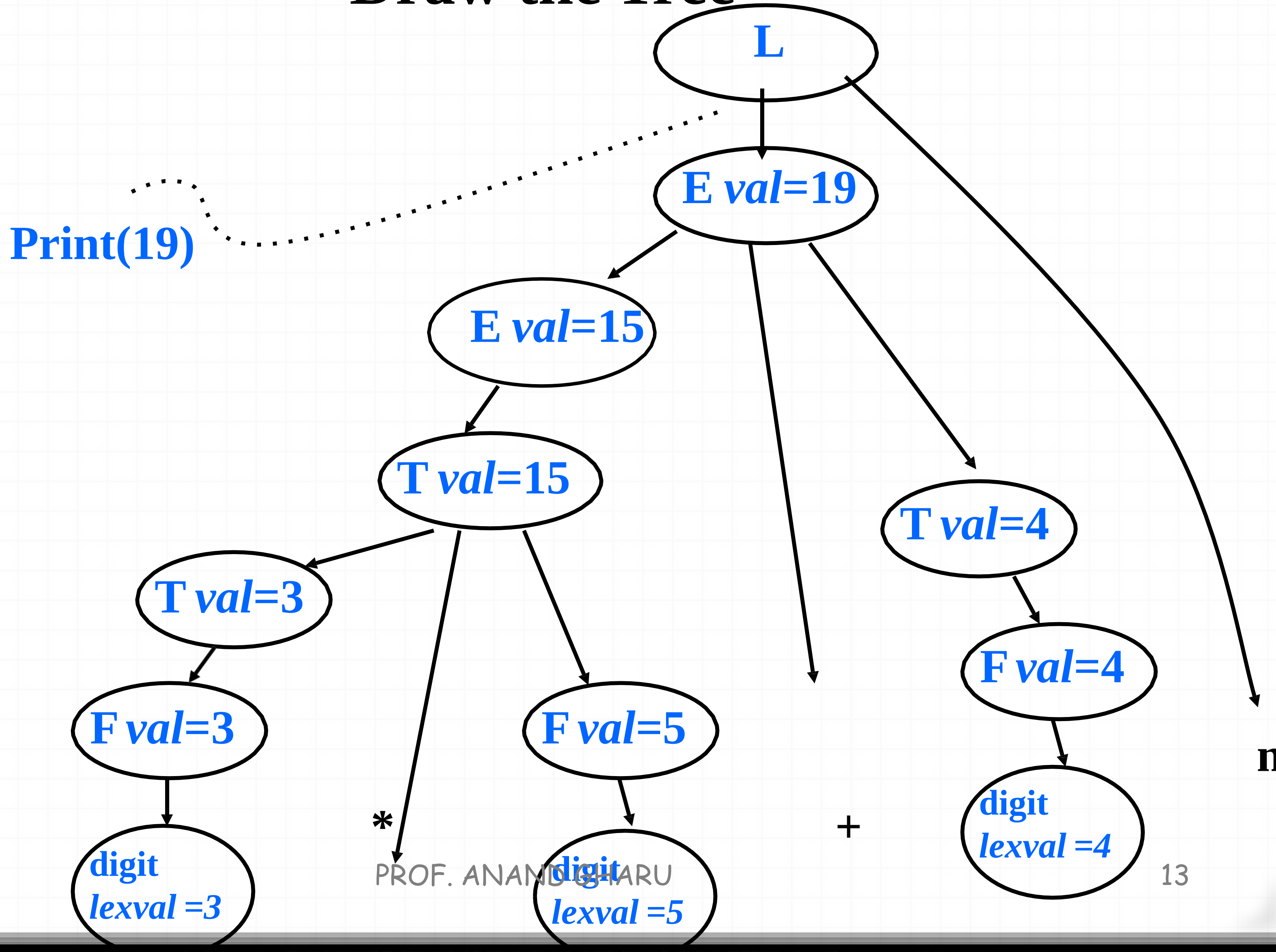
S-attributed definition

- A syntax directed translation that uses synthesized attributes exclusively is said to be a S-attributed definition.

- A parse tree for a S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node, bottom up from leaves to the root.

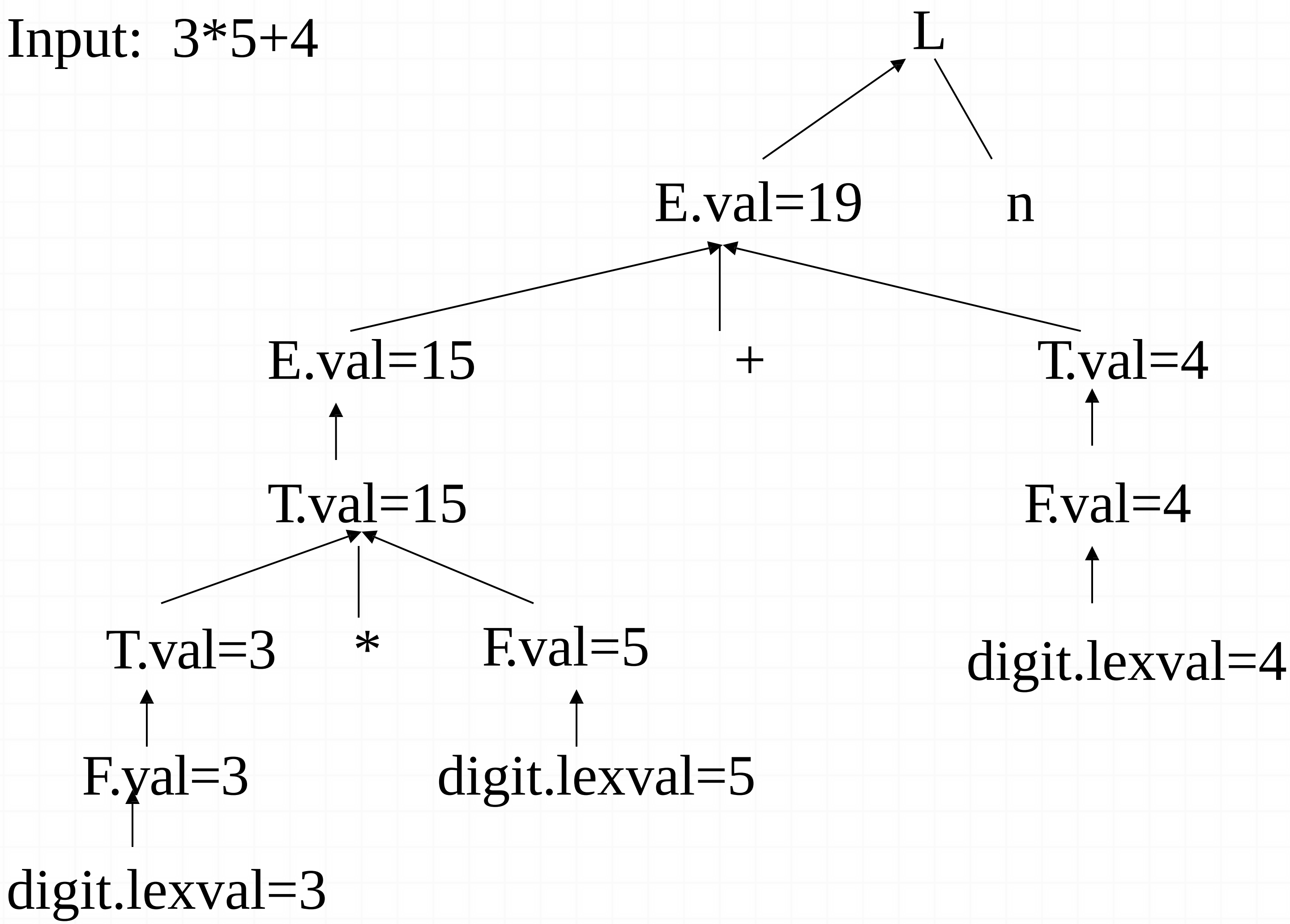
Draw the Tree

Example **3*5+4n**



Dependency Graph

Input: 3*5+4



Inherited attributes

- An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.

- Convenient way for expressing the dependency of a programming language construct on the context in which it appears.

- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.

- Example: The inherited attribute distributes type information to the various identifiers in a declaration.

Syntax-Directed Definition – Inherited Attributes

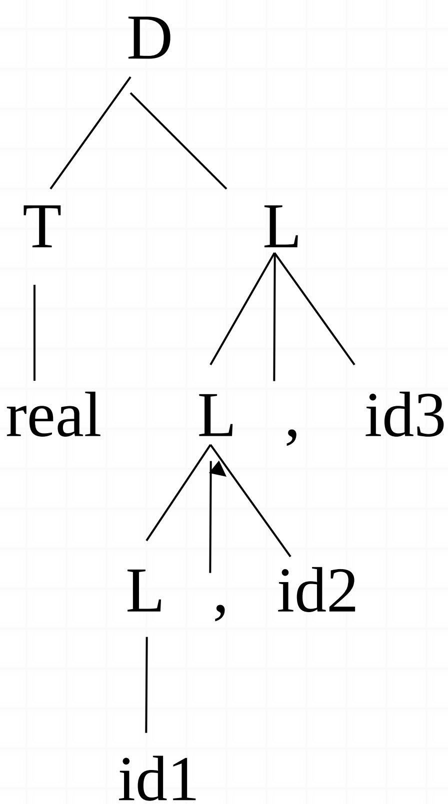
Production	**Semantic Rules**
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$ **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

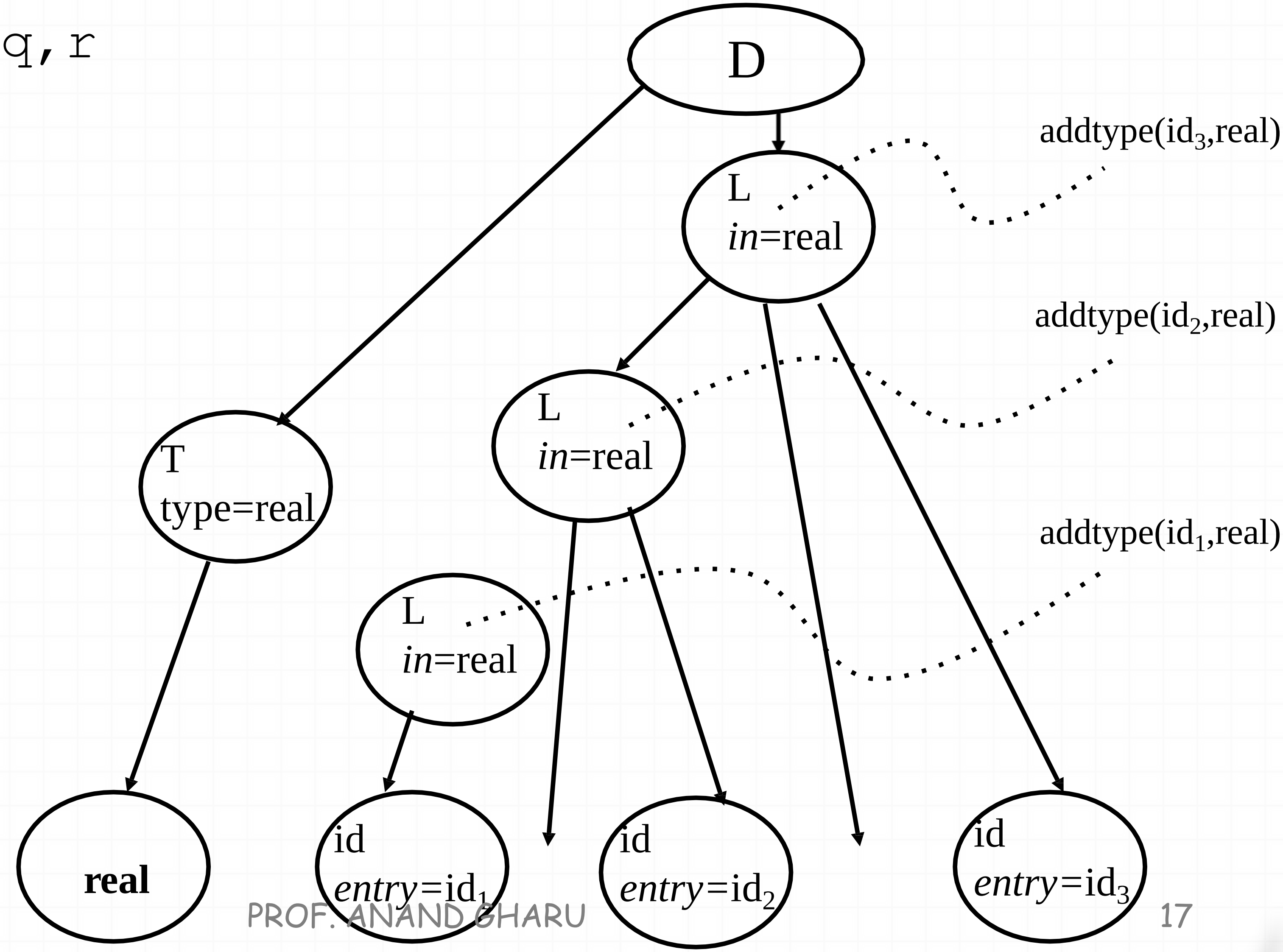
1. Symbol T is associated with a synthesized attribute *type*.
2. Symbol L is associated with an inherited attribute *in.*

Annotated parse tree

Input: `real p,q,r`

parse tree





Dependency Graph

- Directed Graph
- Shows interdependencies between attributes.
- If an attribute b at a node depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
- Construction:
  - Put each semantic rule into the form $b=f(c_1,\ldots,c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a procedure call.
  - E.g.,
    - **L → E n** ***print*****(E.*****val*****)**
    - **Becomes:** ***dummy* = *print*****(E.*****val*****)**
  - The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

Dependency Graph Construction

for each node *n* in the parse tree do

  for each attribute a of the grammar symbol at *n* do

    construct a node in the dependency graph for a

for each node *n* in the parse tree do

  for each semantic rule $b = f(c_1, \ldots, c_n)$

    associated with the production used at *n* do

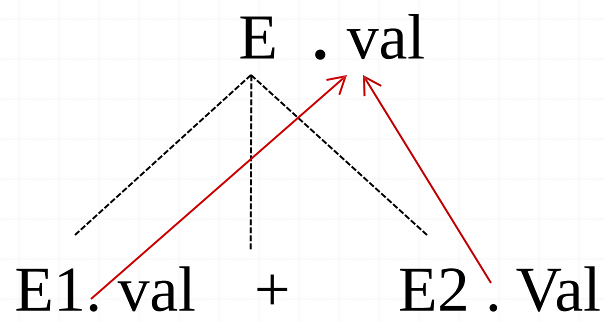
      for i= 1 to n do

        construct an edge from
        the node for c_i to the node for *b*

Dependency Graph Construction

- Example
- Production — Semantic Rule

$E \rightarrow E1 + E2$ — $E.val = E1.val + E2.val$



- E.val is synthesized from E1.val and E2.val
- The dotted lines represent the parse tree that is not part of the dependency graph.

Dependency Graph

D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1 **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

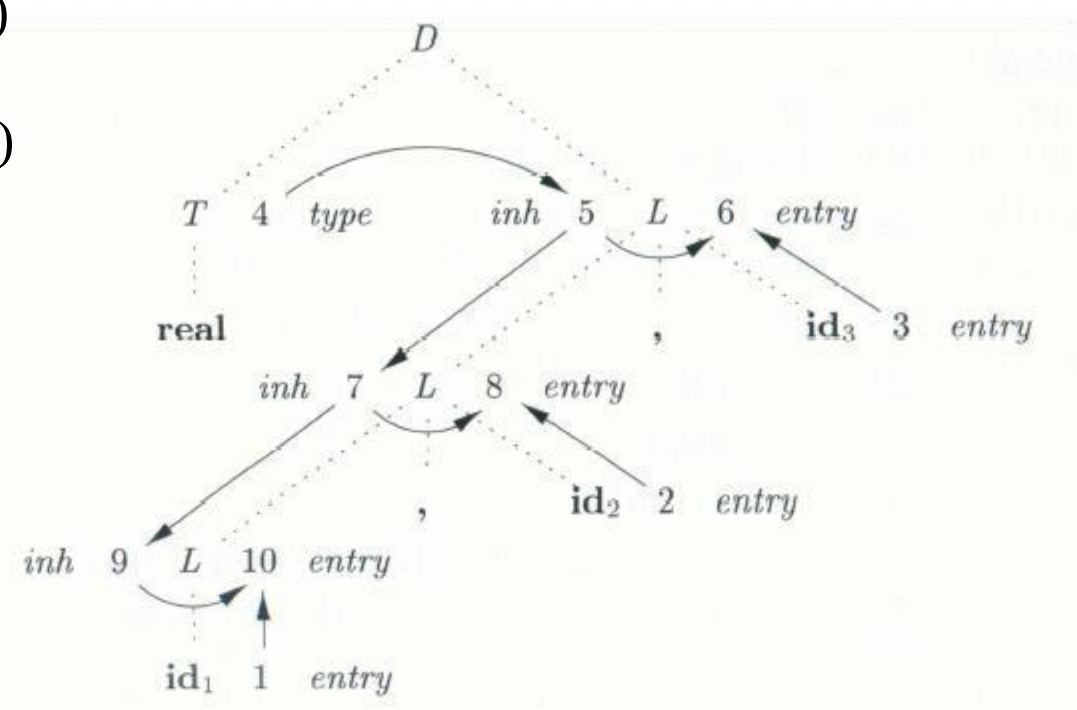


Figure 5.9: Dependency graph for a declaration **float** id_1 , id_2 , id_3

Evaluation Order

- A topological sort of a directed acyclic graph is any ordering m1,m2…mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.

  . i.e if there is an edge from m_i to m_j them m_i appears before m_j in the ordering

- Any topological sort of dependency graph gives a valid order for evaluation of semantic rules associated with the nodes of the parse tree.
  - The dependent attributes c1,c2….ck in b=f(c1,c2….ck) must be available before f is evaluated.

- Translation specified by Syntax Directed Definition

- Input string ⟶ parse tree ⟶ dependency graph ⟶ evaluation order for semantic rules

Evaluation Order

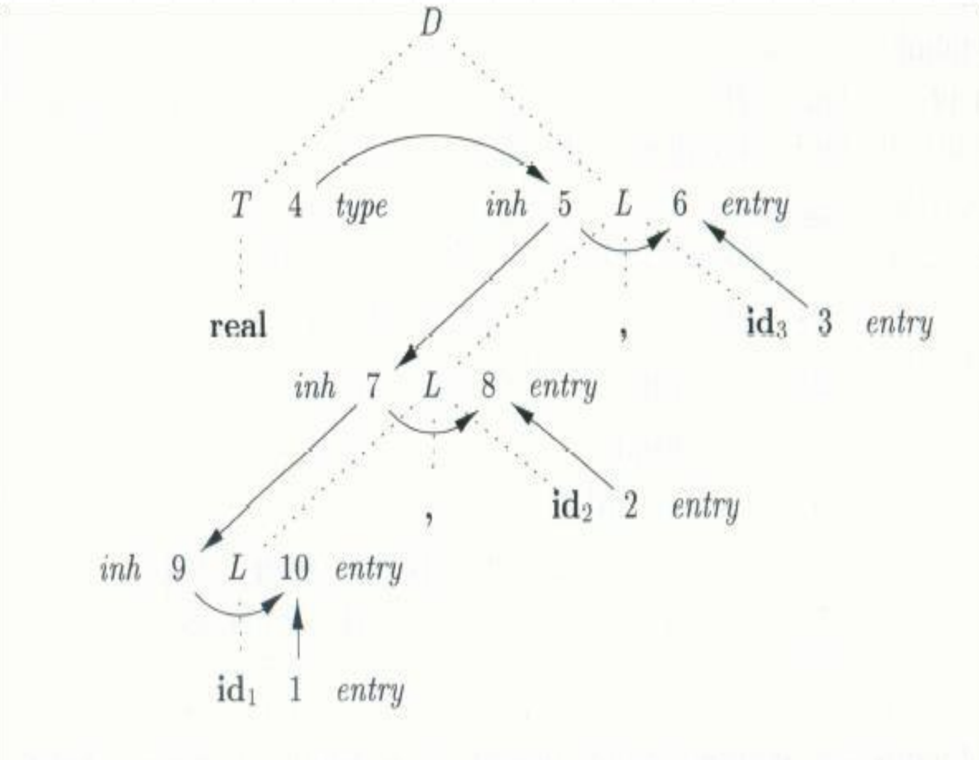


Figure 5.9: Dependency graph for a declaration **float** $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$

- a4=real;
- a5=a4;
- addtype(id3.entry,a5);
- a7=a5;
- addtype(id2.entry,a7);
- a9=a7;
- addtype(id1.entry,a5);

Evaluating Semantic Rules

- Parse Tree methods
  - At compile time evaluation order obtained from the topological sort of dependency graph.
  - Fails if dependency graph has a cycle
- Rule Based Methods
  - Semantic rules analyzed by hand or specialized tools at compiler construction time
  - Order of evaluation of attributes associated with a production is pre-determined at compiler construction time
- Oblivious Methods
  - Evaluation order is chosen without considering the semantic rules.
  - Restricts the class of syntax directed definitions that can be implemented.
  - If translation takes place during parsing order of evaluation is forced by parsing method.

Syntax Trees

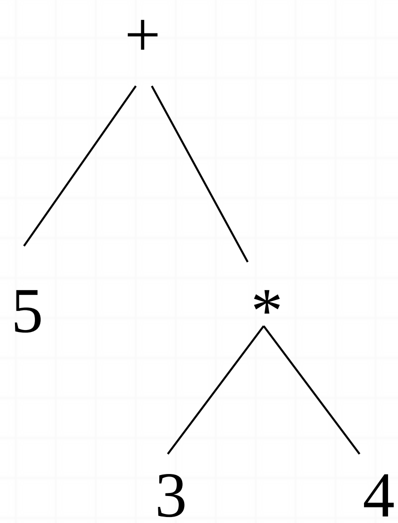
Syntax-Tree

- an intermediate representation of the compiler's input.
- A condensed form of the parse tree.
- Syntax tree shows the syntactic structure of the program while omitting irrelevant details.
- Operators and keywords are associated with the interior nodes.
- Chains of simple productions are collapsed.

Syntax directed translation can be based on syntax tree as well as parse tree.

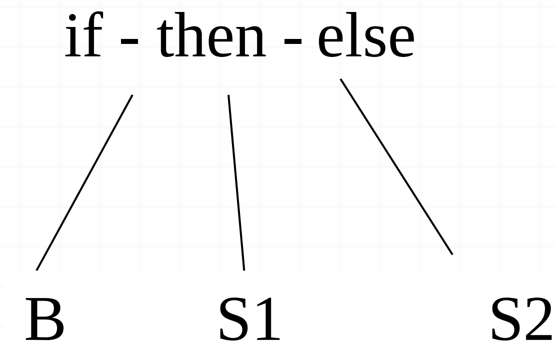
Syntax Tree-Examples

Expression:



- Leaves: identifiers or constants
- Internal nodes: labelled with operations
- Children: of a node are its operands

if B then S1 else S2



Statement:

- Node's label indicates what kind of a statement it is
- Children of a node correspond to the components of the statement

Constructing Syntax Tree for Expressions

- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called *label of the node*) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.

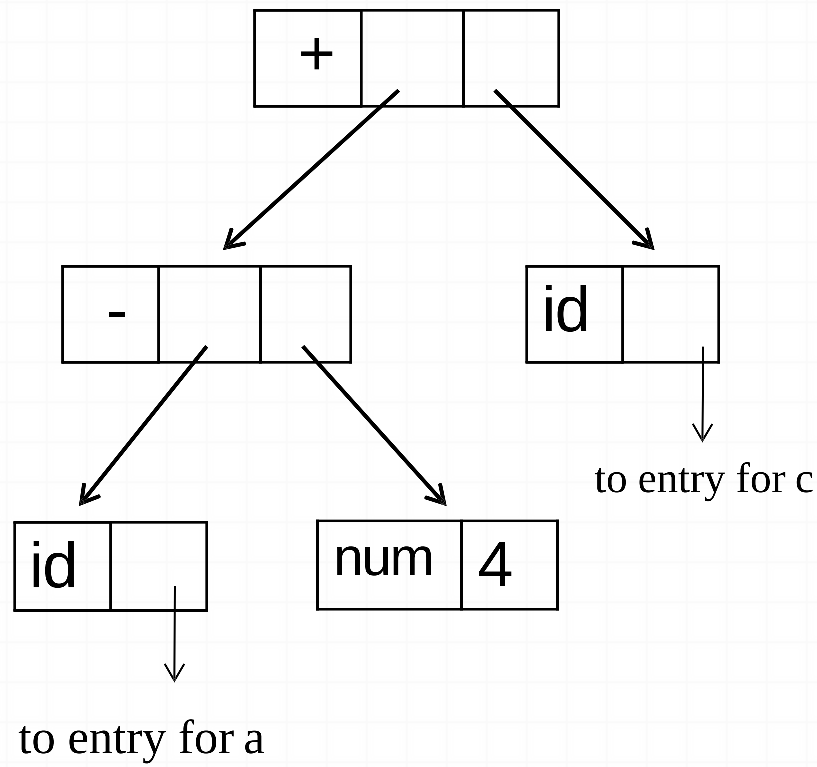
- Functions used to create nodes of syntax tree for expressions with binary operator are given below.
  - mknode(op,left,right)
  - mkleaf(id,entry)
  - mkleaf(num,val)

Each function returns a pointer to a newly created node.

Constructing Syntax Tree for Expressions-

Example: a-4+c

1.p1:=mkleaf(id,entrya);
2.p2:=mkleaf(num,4); 3.
p3:=mknode(-,p1,p2)
4. p4:=mkleaf(id,entryc);
5. p5:= mknode(+,p3,p4);



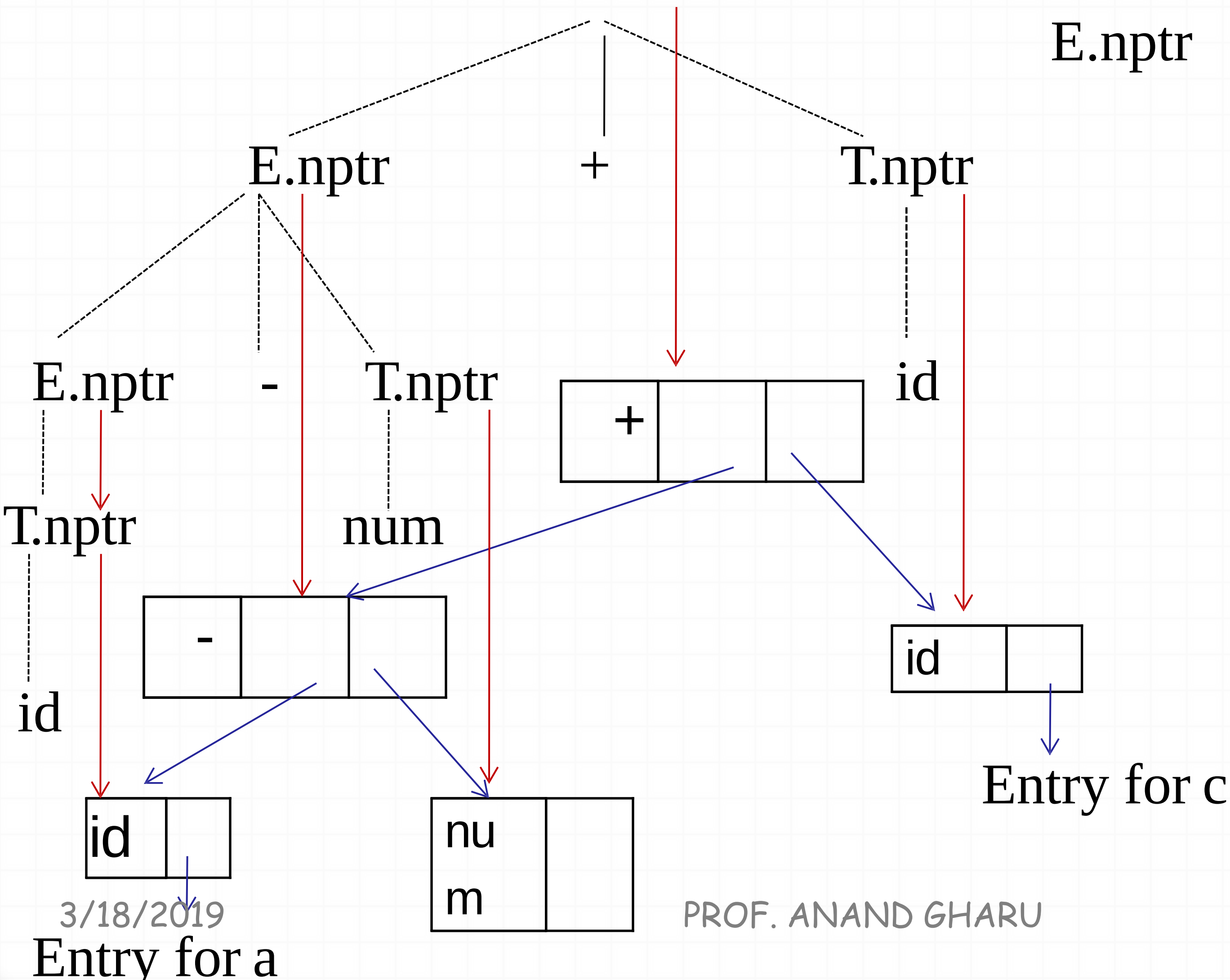
- **The tree is constructed bottom up.**

A syntax Directed Definition for Constructing Syntax Tree

1. It uses underlying productions of the grammar to schedule the calls of the functions *mkleaf* and *mknode* to construct the syntax tree
2. Employment of the synthesized attribute *nptr* (pointer) for E and T to keep track of the pointers returned by the function calls.

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
T → (E)	T.*nptr* = E.*nptr*
T → id	T.*nptr* = *mkleaf*(id, id.*lexval*)
T → num	T.*nptr* = *mkleaf*(num, num.*val*)

Annotated parse tree depicting construction of syntax tree for the expression a-4+c



S-Attributed Definitions

1. Syntax-directed definitions are used to specify syntax-directed translations.

2. To create a translator for an arbitrary syntax-directed definition can be difficult.

3. We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).

4. We will look at two sub-classes of the syntax-directed definitions:
   – **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
   – All actions occur on the right hand side of the production.
   – **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

1. To implement S-Attributed Definitions and L-Attributed Definitions we can evaluate semantic rules in a single pass during the parsing.

6. Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

Bottom-Up Evaluation of S-Attributed Definitions

- A translator for an S-attributed definition can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- We put the values of the synthesized attributes of the grammar symbols a stack that has extra fields to hold the values of attributes.
  - The stack is implemented by a pair of arrays *val & state*
  - If the i^{th} state symbol is A the val[i] will hold the value of the attribute associated with the parse tree node corresponding to this A.

Bottom-Up Evaluation of S-Attributed Definitions

- We evaluate the values of the attributes during reductions.

$A \rightarrow XYZ$ A.a=f(X.x,Y.y,Z.z)   where all attributes are synthesized.

top →

state	val
Z	Z.z
Y	Y.y
X	X.x
.	.

➔ top→

state	val
A	A.a
.	.

- Synthesized attributes are evaluated before each reduction.
- Before XYZ is reduced to A, the value of Z.z is in val[top], that of Y.y in val[top-1] and that of X.x in val[top-2].
- After reduction top is decremented by 2.
- If a symbol has no attribute the corresponding entry in the array is undefined.

Bottom-Up Evaluation of S-Attributed Definitions

Production	**Semantic Rules**
L → E **n**	print(val[top-1])
E → E_1 + T	val[ntop] = val[top-2] + val[top]
E → T	
T → T_1 * F	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → **digit**	

1. At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
2. At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Bottom-Up Evaluation -- Example

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

Input	state	val	semantic rule
5+3*4n	-	-	
+3*4n	5	5	
+3*4n	F	5	F → digit
+3*4n	T	5	T → F
+3*4 n	E	5	E → T
3*4n	E+	5-	
*4 n	E+3	5-3	
*4n	E+F	5-3	F → digit
*4n	E+T	5-3	T → F
4n	E+T*	5-3-	
n	E+T*4	5-3-4	
n	E+T*F	5-3-4	F → digit
n	E+T	5-12	$T \rightarrow T_1 * F$
n	E	17	$E \rightarrow E_1 + T$
	En	17-	L → E n
	L	17	

L-Attributed Definitions

- When translation takes place during parsing, order of evaluation is linked to the order in which the nodes of a parse tree are created by parsing method.
- A natural order can be obtained by applying the procedure *dfvisit* to the root of a parse tree.
- We call this evaluation order *depth first order*.
- L-attributed definition is a class of syntax directed definition whose attributes can always be evaluated in depth first order(L stands for left since attribute information flows from left to right).

```
dfvisit(node n)
{
  for each child m of n, from left to right
 {
 evaluate inherited attributes of m
  dfvisit(m)
}
evaluate synthesized attributes of n
}
```

L-Attributed Definitions

A syntax-directed definition is **L-attributed** if each inherited attribute of X_j, where $1 \le j \le n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on

1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of X_j in the production
2. The inherited attribute of A

Every S-attributed definition is L-attributed, since the restrictions apply only to the inherited attributes (not to synthesized attributes).

A Definition which is *not* L-Attributed

Productions	**Semantic Rules**
A → L M	L.in=l(A.i)
	M.in=m(L.s)
	A.s=f(M.s)
A → Q R	R.in=r(A.in)
	Q.in=q(R.s)
	A.s=f(Q.s)

This syntax-directed definition is not L-attributed because the semantic rule Q.in=q(R.s) violates the restrictions of L-attributed definitions.

- When Q.in must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of Q.in depends on R.s which will be available after we return from R. So, we are not be able to evaluate the value of Q.in before we enter to Q.

Top-down translation of L-Attributed Definition

E → T { E'.in = T.val } R { E.val = R.s}

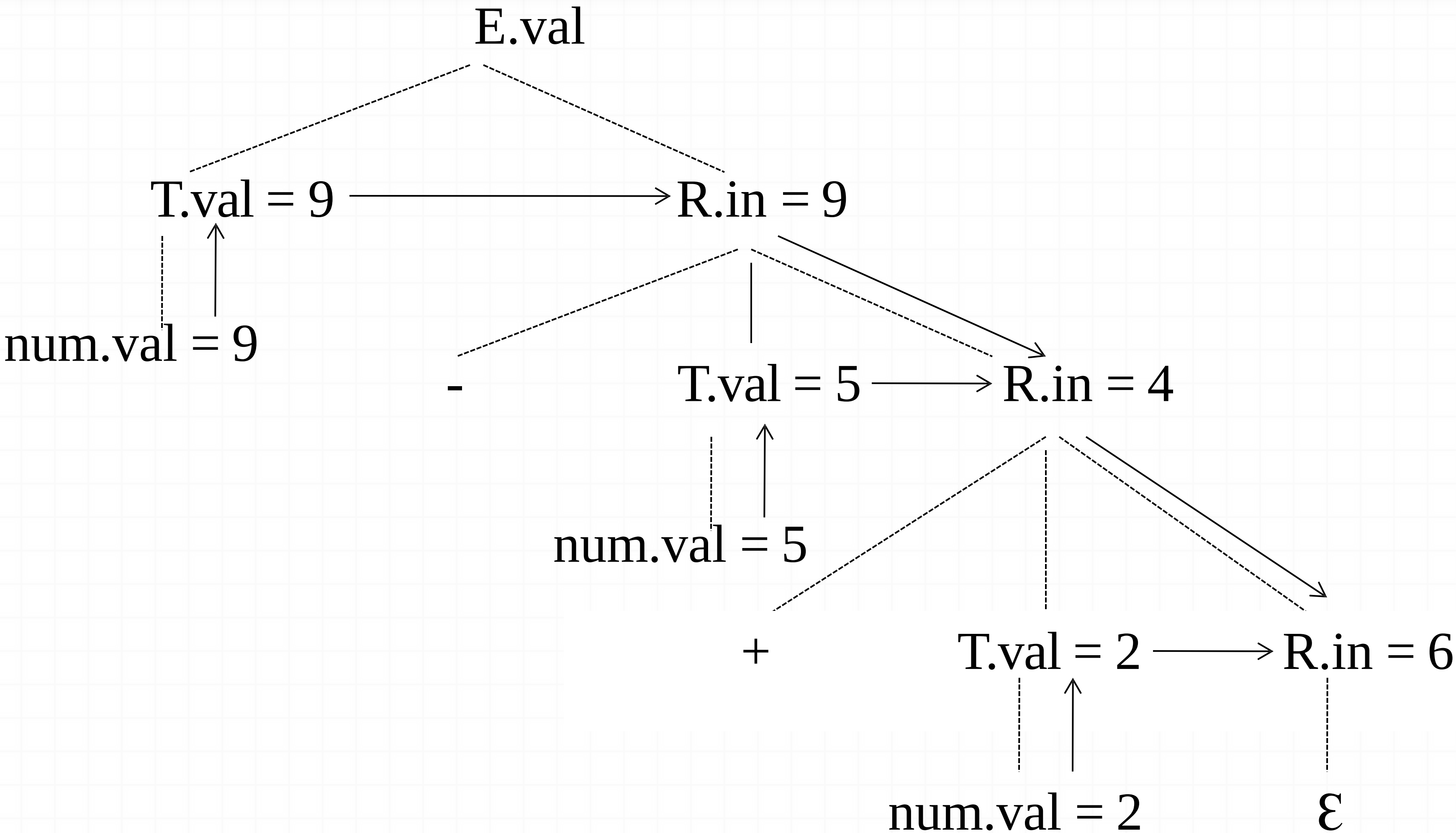
R → + T { R'.in = E.in + T.val } R' {R.s = R'.s}

R → + T { R'.in = E.in - T.val } R' {R.s = R'.s}

R → Ɛ { R.s = R.in}

T → (E){ T.val = E.val }

T → num { T.val = num.val}



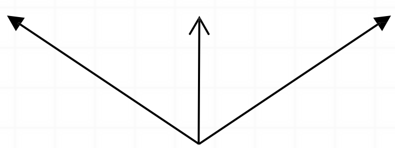
Top-down translation of L-Attributed Definition

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated).
- Translation schemes describe the order and timing of attribute computation.
- A **translation scheme** is a context-free grammar in which:
  - –attributes are associated with the grammar symbols and
  - –semantic actions enclosed between braces {} are inserted within the right sides of productions.

Each semantic rule can only use the information compute by already executed semantic rules.

- *Ex:* A → { ... } X { ... } Y { ... }



Semantic Actions

Translation Schemes for S-attributed Definitions

- useful notation for specifying translation during parsing.
- Can have both synthesized and inherited attributes.
- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production	Semantic Rule	
$E \rightarrow E1 + T$	E.val = E1.val + T.val	a production of a syntax directed definition
$\Downarrow$		
$E \rightarrow E1 + T$ { E.val = E1.val + T.val }		the production of the corresponding translation scheme

A Translation Scheme Example

• A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

E → T R

R → + T { print("+") } R1

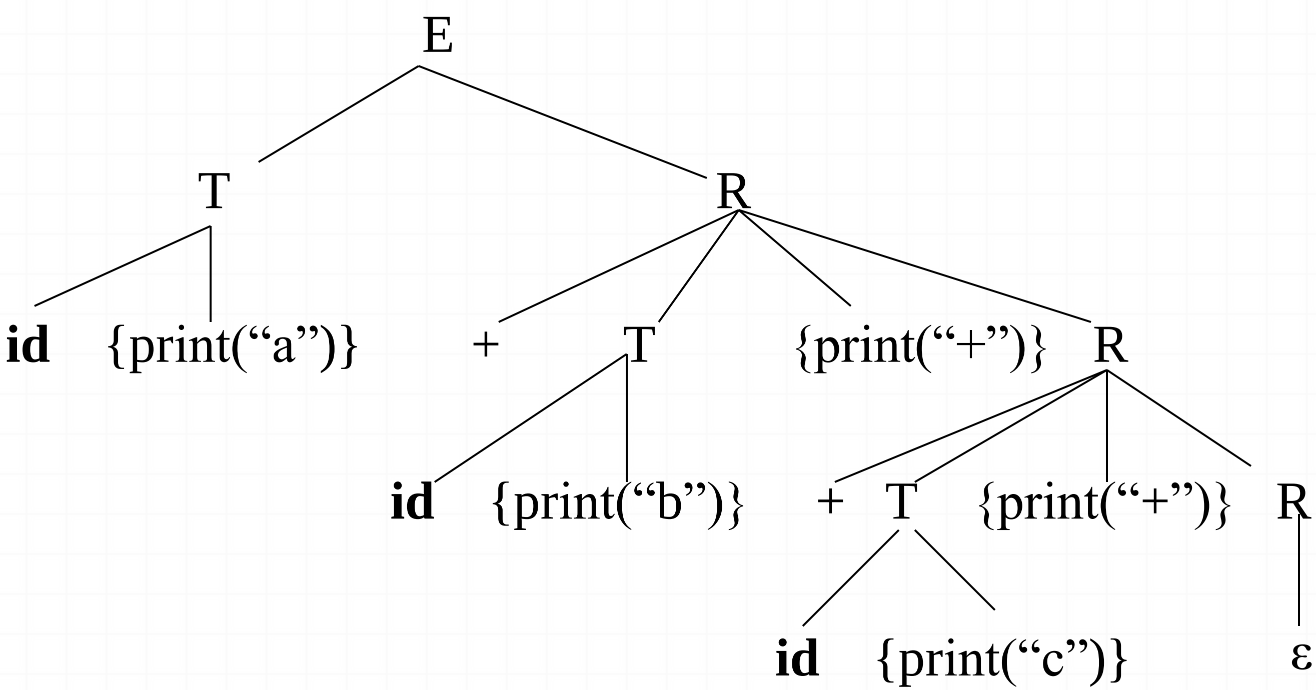
R → ε

T → **id** { print(**id**.name) }

a+b+c ab+c+

infix expression postfix expression

A Translation Scheme Example (cont.)



The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.

Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules to ensure that the attribute value is available when an action refers to it.

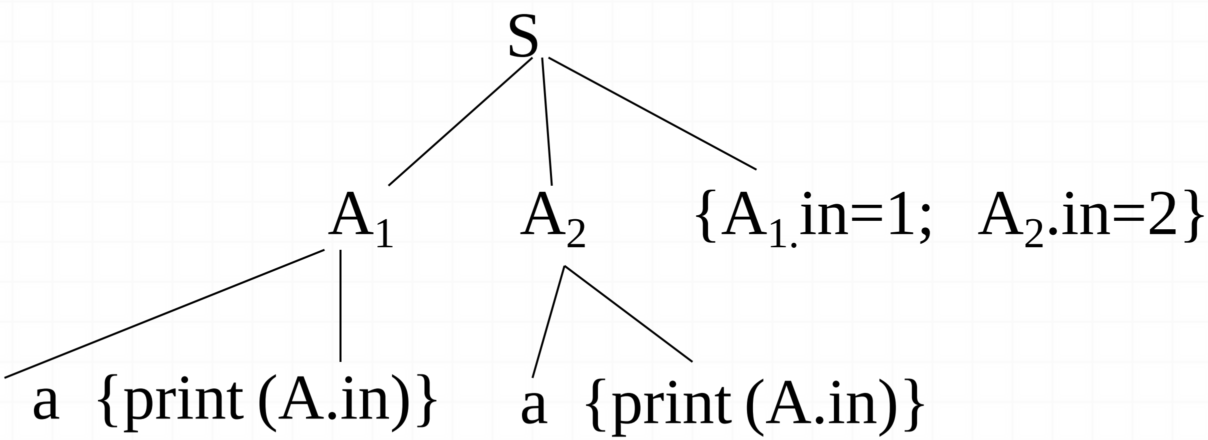
  1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
  2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
  3. A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).

- With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).

Inherited Attributes in Translation Schemes: Example

$S \rightarrow A_1 A_2 \quad \{A_1.in=1; \;\; A_2.in=2\}$

$A \rightarrow a \quad \{ print(A.in)\}$



A Translation Scheme with Inherited Attributes

D → T {L.in = T.type } L

T → **int** { T.type = integer }

T → **real** { T.type = real }

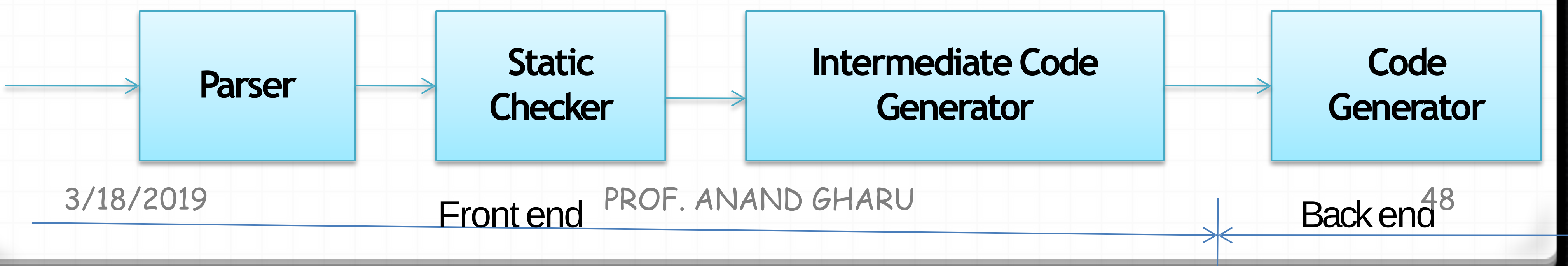
L → {L1.in = L.in } L1, **id {**addtype(**id**.entry,L.in)}

L → id **{**addtype(**id**.entry,L.in)}

- This is a translation scheme for an L-attributed definitions

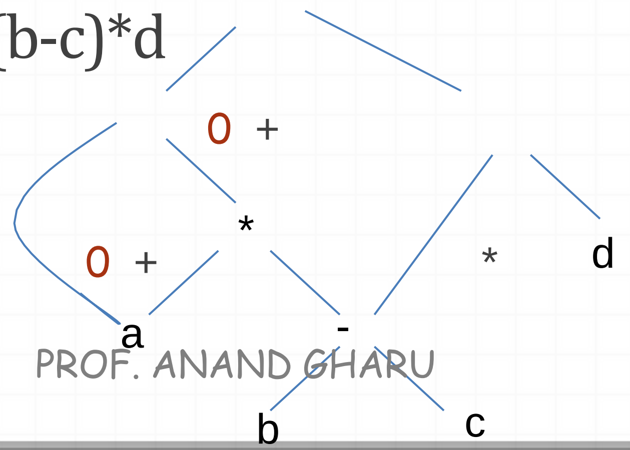
INTRODUCTION

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, intermediate code generation

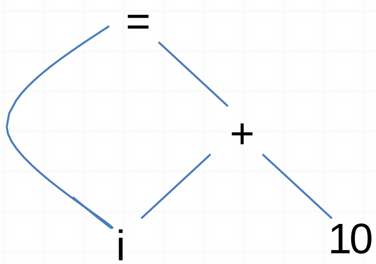


Variants of syntax trees

- It is sometimes beneficial to create a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- Example: a+a*(b-c)+(b-c)*d



Value-number method for constructing DAG's



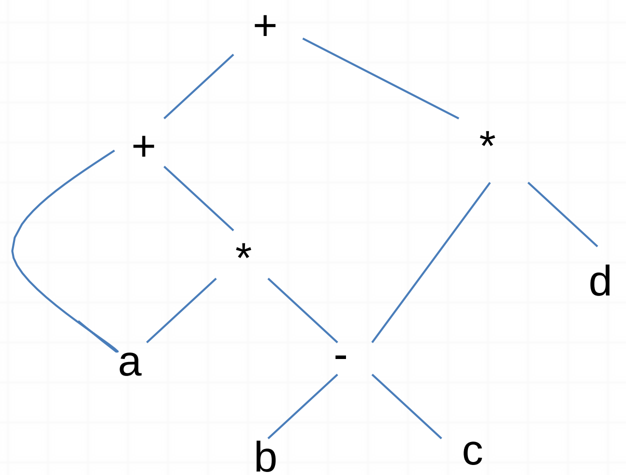
id		
num	10	
+	1	2
=	1	3

To entry for i

- Algorithm
  - Search the array for a node M with label op, left child l and right child r
  - If there is such a node, return the value number M
  - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table

Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example: (a + (a * b-c)) + ((b-c) * d)



t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

Forms of three address instructions

- Assignment statement : x = y op z
- Assignment instruction : x = op y
- Copy statement : x = y
- Unconditional Jump : goto L
- Conditional jump : if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- Indexed Assignments : x = y[i] and x[i] = y
- Address & Pointer Assignments : x = &y and x = *y and *x = y

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- b * minus c + b * minus c

op	Arg1	Agr2	Result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4

a = t5

Data structures for three address codes

- Triples
  - Temporaries are not used and instead references to instructions are made
- b * minus c + b * minus c

	op	Arg1	Agr2
35	minus	c	
36	*	b	(0)
37	Minus	c	
38	*	b	(2)
39	+	(1)	(3)
40	=	a	(4)

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Data structures for three address codes

- Indirect triples
  - In addition to triples we use a list of pointers to triples

- b * minus c + b * minus c

	op
(0)	(0)
(1)	(1)
(2)	(2)
(3)	(3)
(4)	(4)
(5)	(5)

	op	Arg1	Agr2
(0)	minus	c	
(1)	*	b	(0)
(2)	Minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

SDD for Array to Produce TAC

```
S-> L: = E   {if L.offset = null then
                  gen(L.place ' := 'E.place); /* Lis a id*/
             else
                gen(L.place '[' L.offset']' ':=' E.place);
             }

E-> E1+ E2   {E.place := newtemp
              gen(E.place ' := 'E1.place+ E2.place);
              }
```

SDD for Array to Produce TAC

```
E-> (E1)   {E.place := E1.place; }

E-> L   {if L.offset = null then
                gen(E.place ' := 'L.place);
         else begin
                E.place :=newtemp( );
                gen(E.place ':='L.place '[' L.offset']' );
end
        }

L-> id   {L.place := id.place;
          L.offset := null;
         }
```

SDD for Array to Produce TAC

```
L-> Elist] {L.offset =  newtemp;
              L.place:=newtemp;
              gen(L.place ' := 'c( Elist.array));
     gen(L.offset ':='Elist.place '*' width(Elist.array));   }

Elist-> Elist,E   {t:= newtemp();
                      m:=Elist.dim + 1;
          gen(t':=' Elist.place * limit(Elist.array,m);
          gen(t':=' t + E.place);
          Elist.array := Elist.array
          Elist.dim : =m;
          Elist.place :=t;
          }
```

SDD for Array to Produce TAC

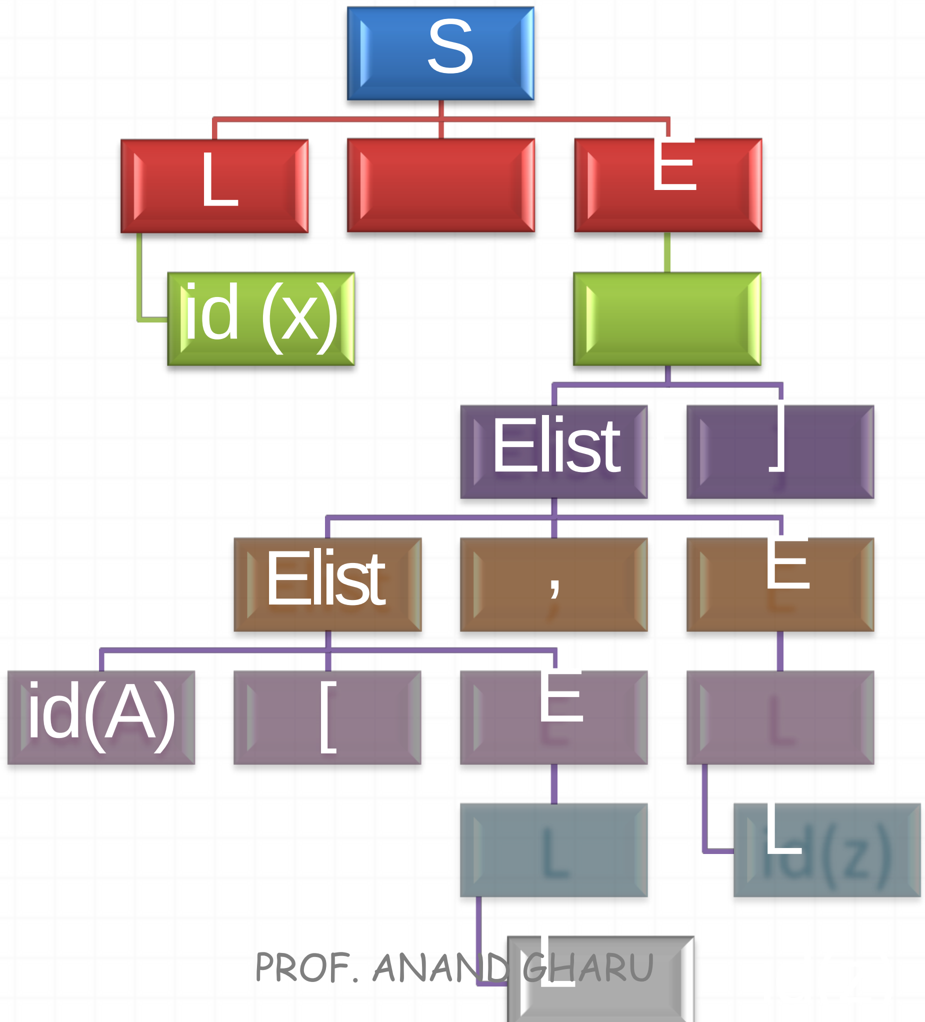
```
Elist-> id [E   {Elist.array := id.place;
                 Elist.dim : =1;
                 Elist.place :=E.place;
                 }
```

SDD for Array to Produce TAC

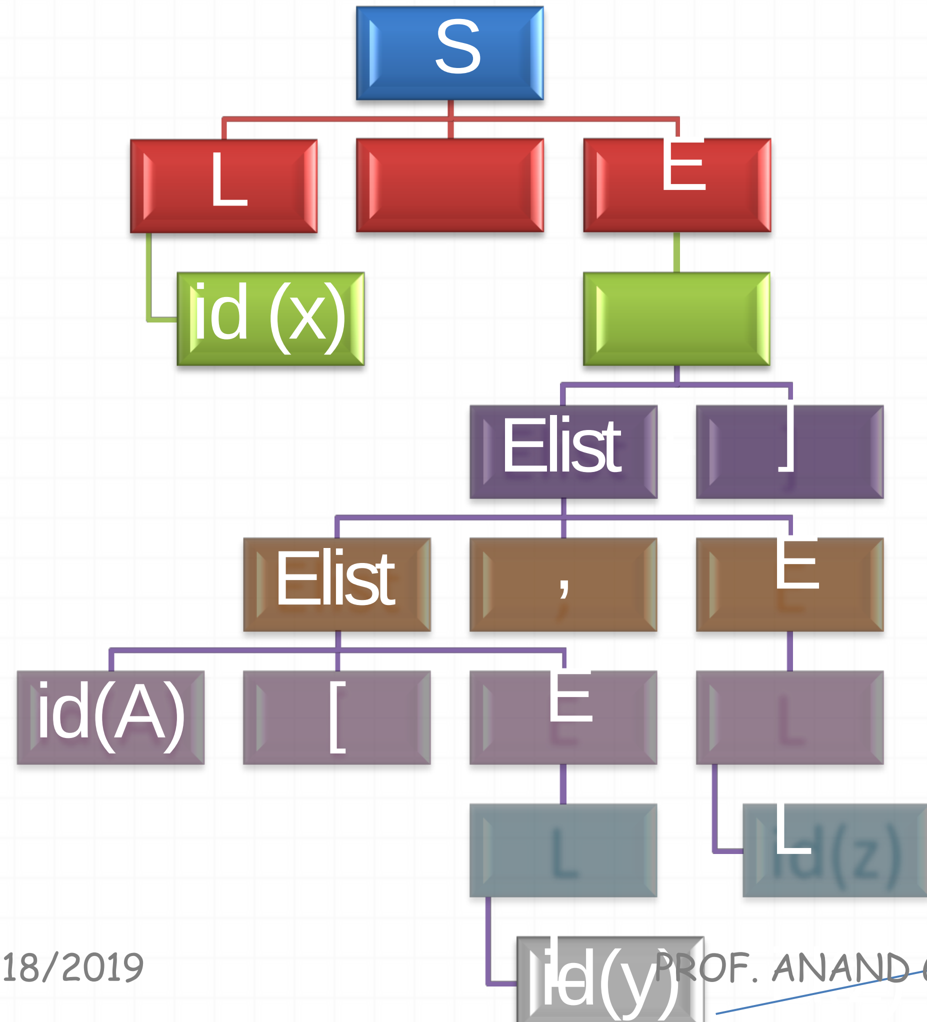
For Eg: - x:= A[y,z] dimensions 10 * 20 and width of A = 4 Produce TAC using SDD of Array to produce TAC

SDD for Array to Produce TAC

First drawing parse tree we obtain :-



SDD for Array to Produce TAC

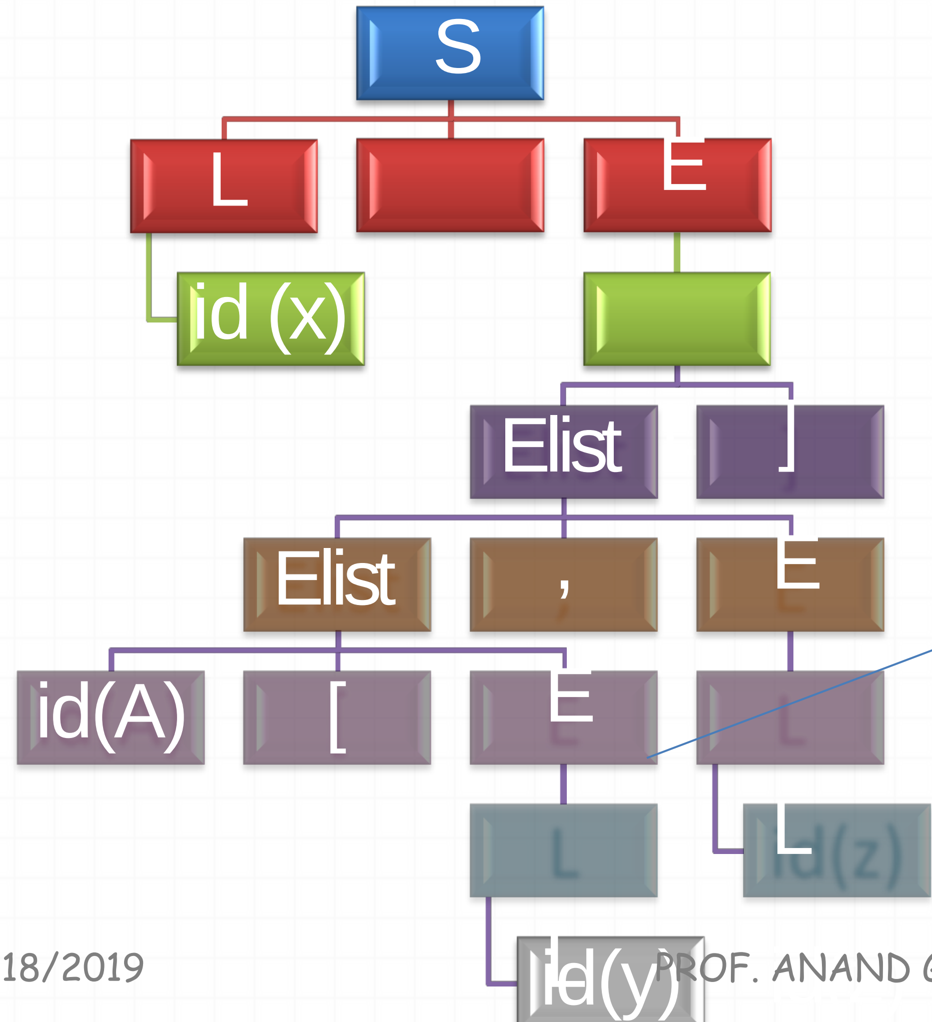


L-> id has semantic rule i.e

L.offset = null

L.place =id.place =y

SDD for Array to Produce TAC



E-> L has semantic rule i.e

{if L.offset = null then
gen(E.place ' := 'L.place);
else begin
E.place :=newtemp( );
gen(E.place ':='L.place '[' L.offset']'
); end
}

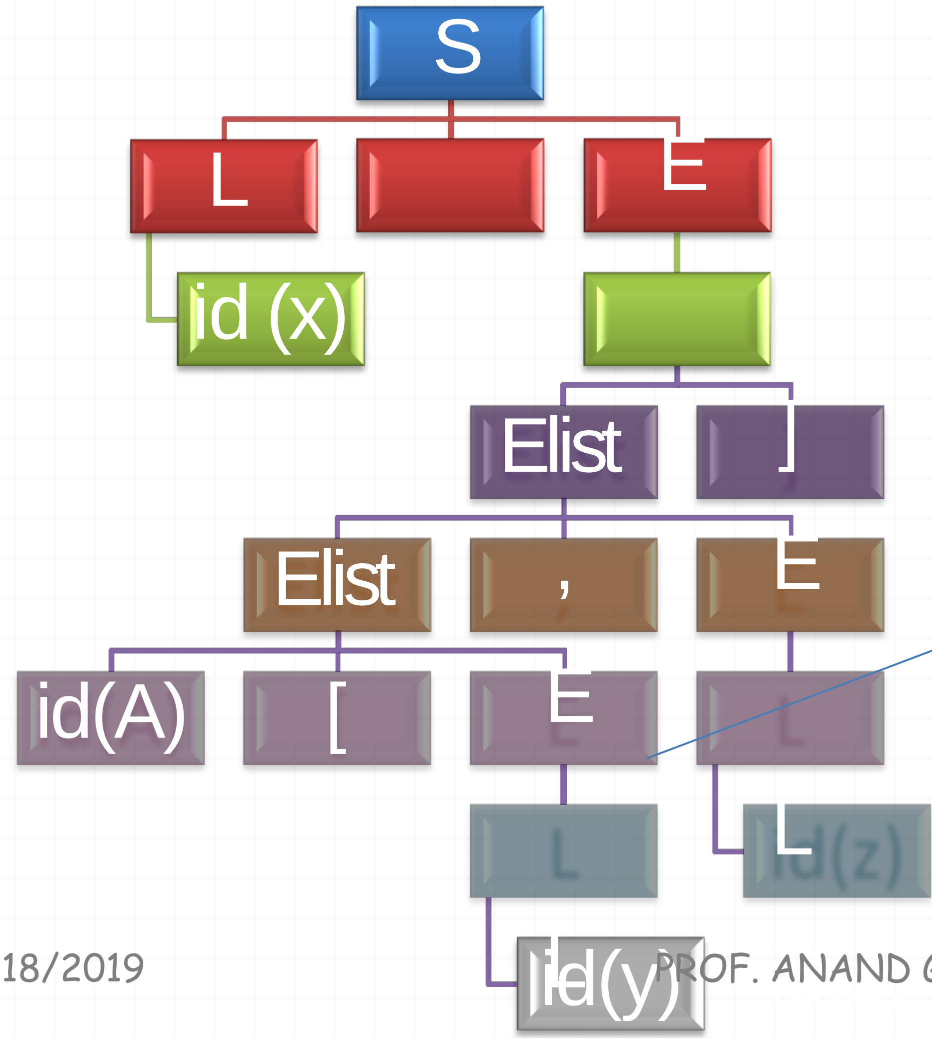
Now, we have

L.offset = null

Thus,

E.Place : = L.place :=y

SDD for Array to Produce TAC



Elist-> id [E {Elist.array := id.place;
Elist.dim : =1;
Elist.place :=E.place;
}

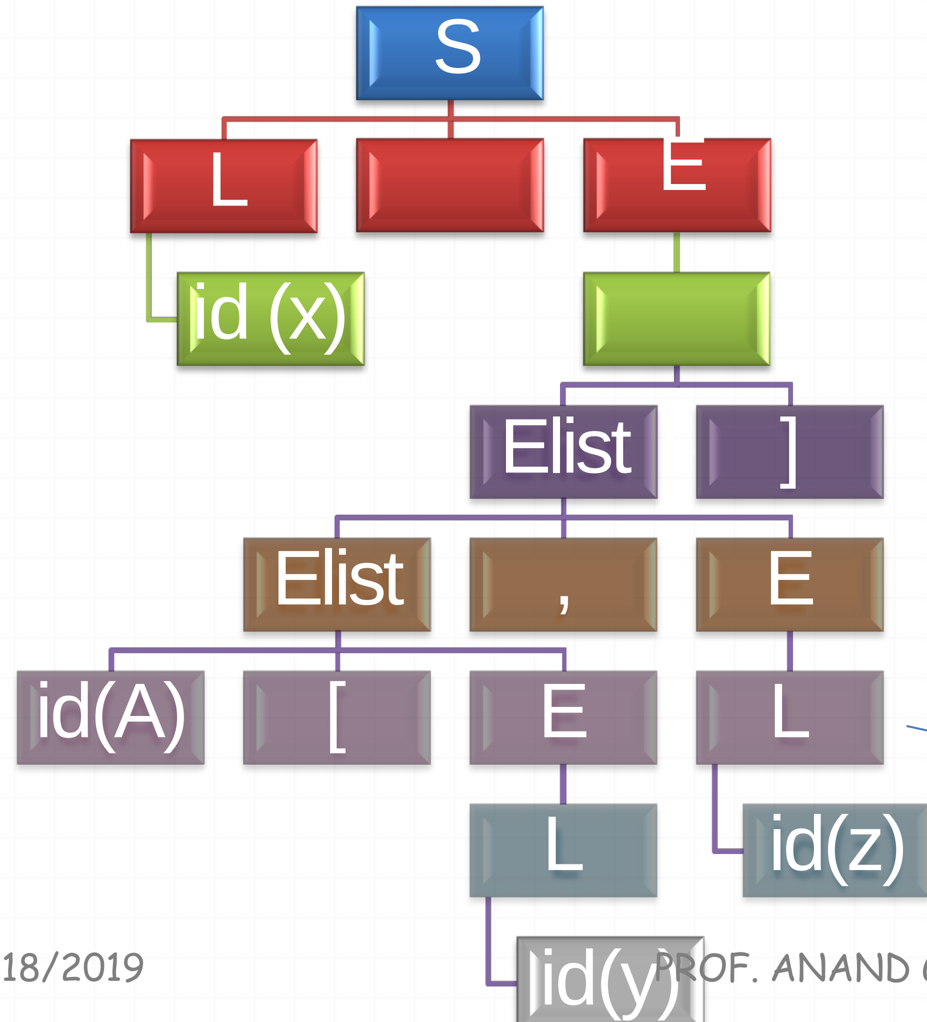
Now, we have

Elist.array = A

Elist.Place : = y

Elist.ndim = 1

SDD for Array to Produce TAC

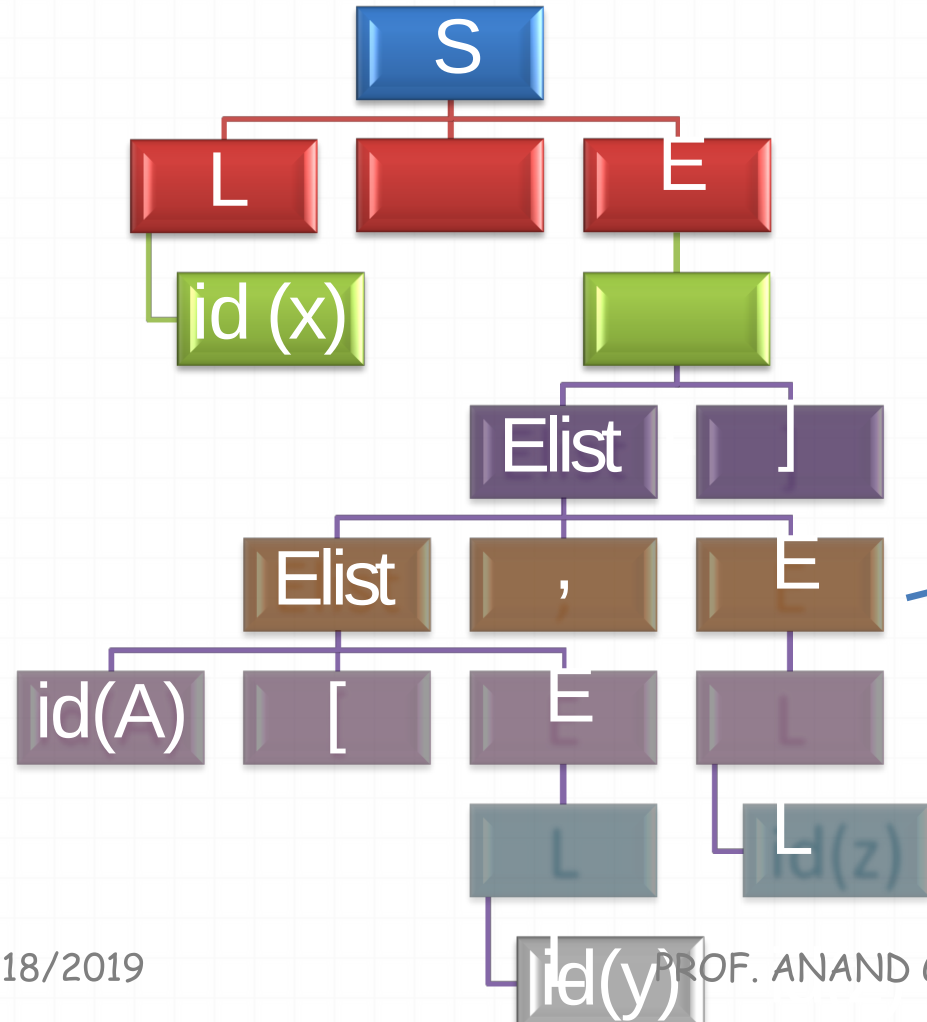


L-> id has semantic rule i.e

L.offset = null

L.place =id.place =z

SDD for Array to Produce TAC

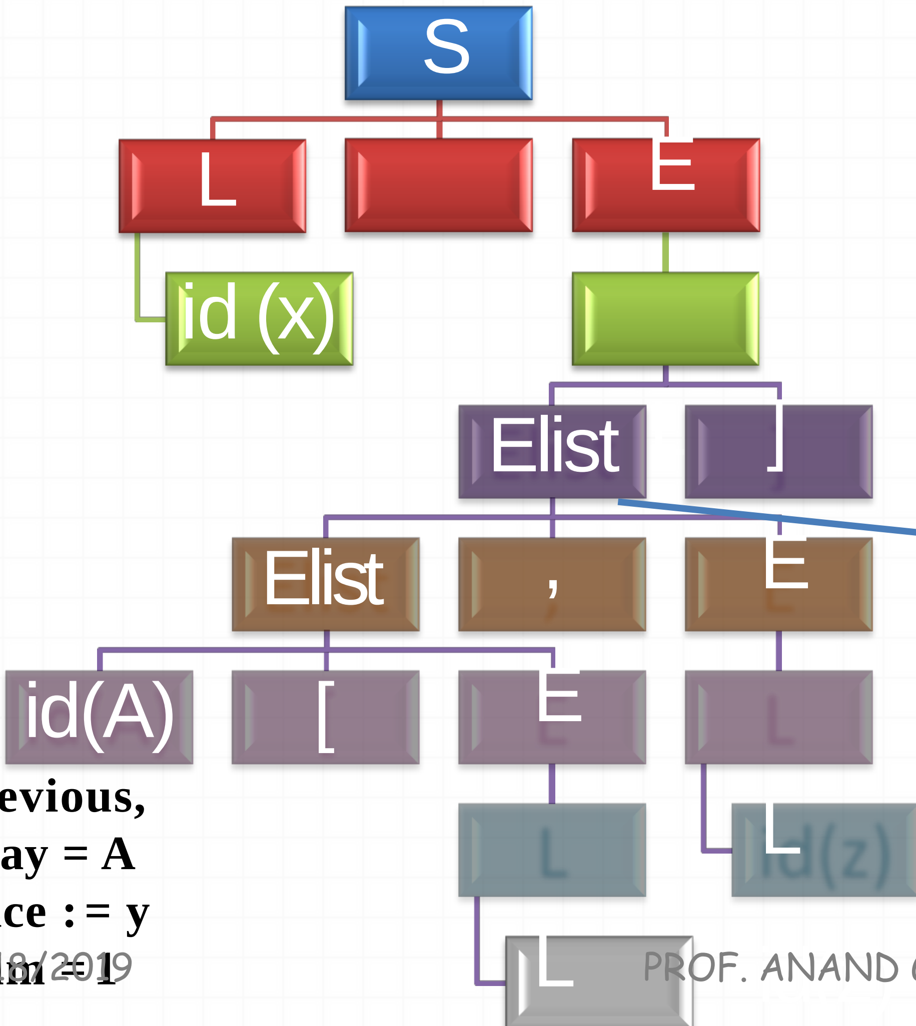


E-> L has semantic rule i.e

{if L.offset = null then
gen(E.place ' := 'L.place);
else begin
E.place :=newtemp( );
gen(E.place ':='L.place '[' L.offset']'
); end
}

Now, we have
L.offset = null
Thus,
E.Place : = L.place :=z

SDD for Array to Produce TAC




```
Elist-> Elist,E   {t:= newtemp();
            m:=Elist.dim + 1;
         gen(t':=' Elist.place *
limit(Elist.array,m);
         gen(t':=' t + E.place);
         Elist.array := Elist.array
         Elist.dim : =m;
         Elist.place :=t;
         }
```

Now, we have

t = t1

m = 1 (Elist.dim) **+ 1 = 2**

t1 = t1 * 20 (limit(A,2))

t1 = t1 + z

And,

Elist.array = A

Elist.dim : = m = 2

Elist.place = t1

From previous,

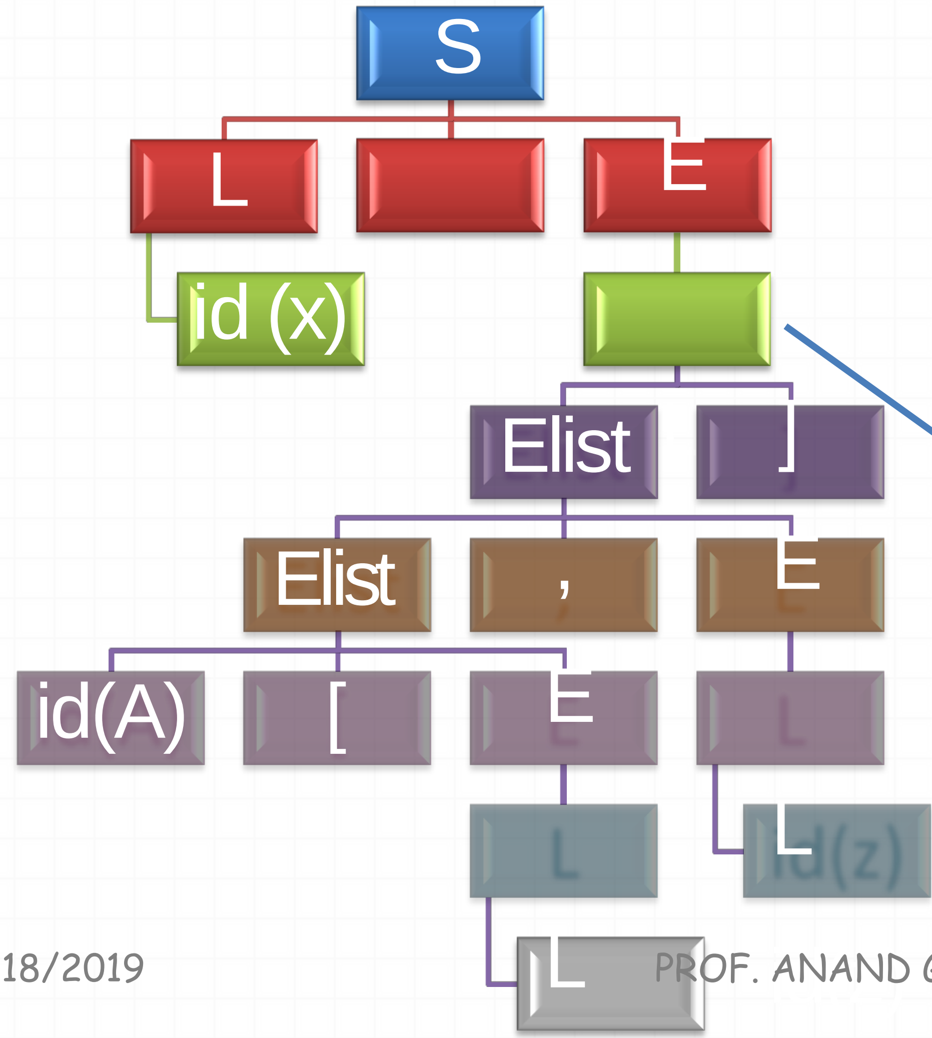
Elist.array = A

Elist.Place : = y

Elist.ndim = 1

E.Place : = z

SDD for Array to Produce TAC

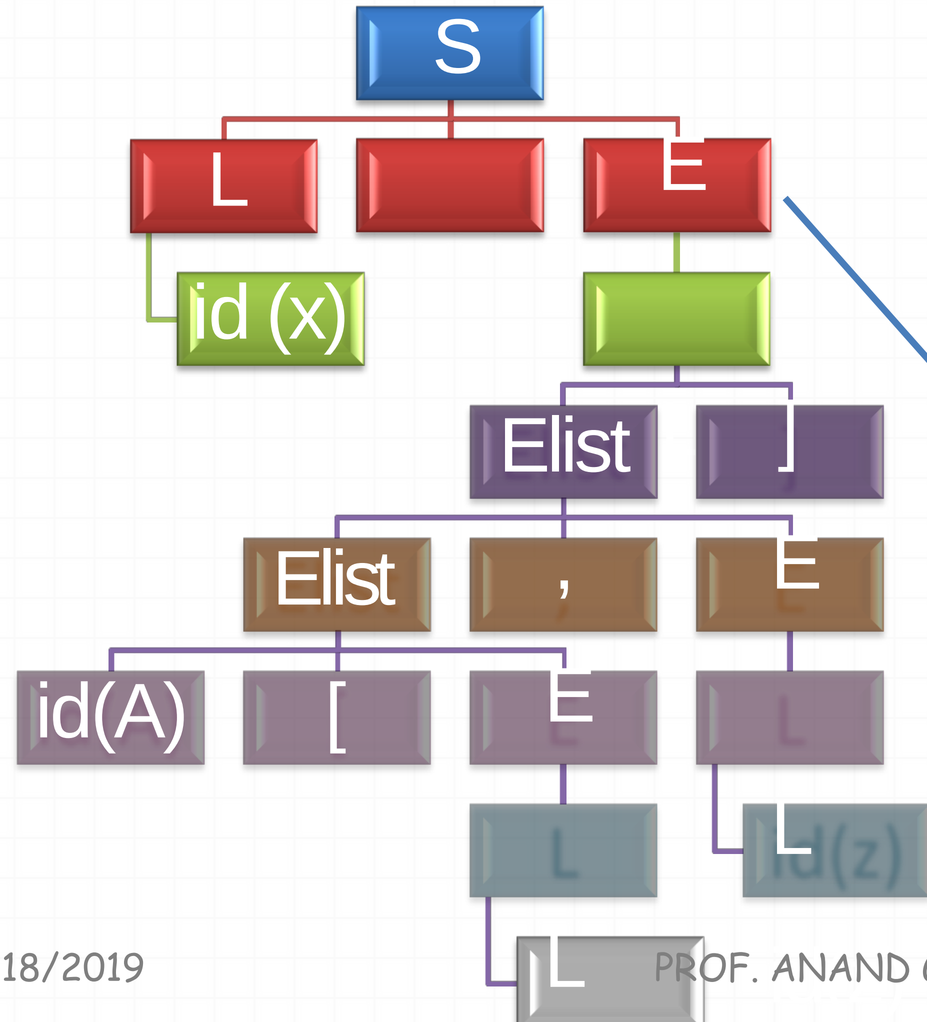


L-> Elist] {L.offset = newtemp;
L.place:=newtemp;
gen(L.place ' :=' c(Elist.array));
gen(L.offset ':=' Elist.place '*' width(Elist.array)); }

Now, we have
L.place= t2
L.offset = t3

t2 = c(A) as Elist.array = A
t3 = t1 * 4 as Elist.place = t1
And width of Elist.array is 4 as mentioned before

SDD for Array to Produce TAC



E-> L has semantic rule i.e

{if L.offset = null then
gen(E.place ' := 'L.place);
else begin
E.place :=newtemp( );
gen(E.place ':='L.place '[' L.offset']'
); end
}

Now, we have

L.offset ≠ null

Thus,

E.Place : t4

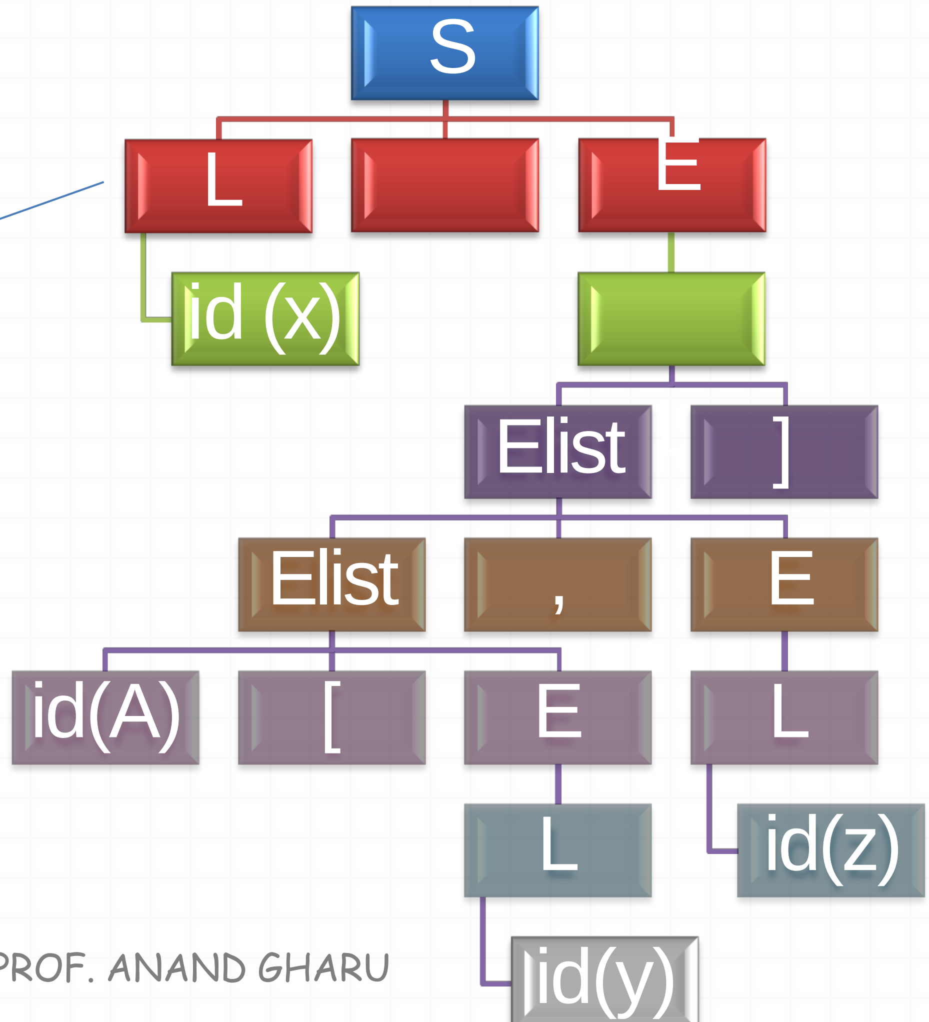
t4 = t2 [t3]

as L.place= t2

L.offset = t3

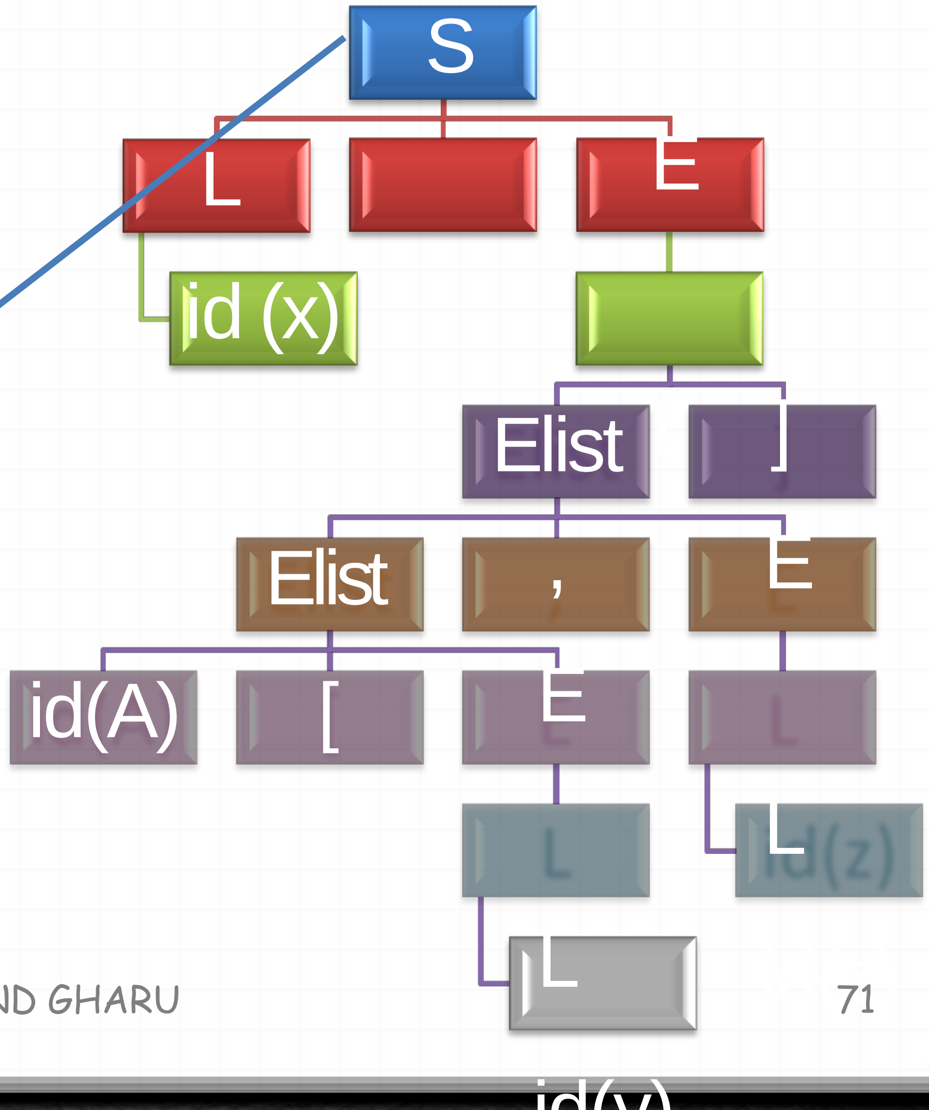
SDD for Array to Produce TAC

L-> id has semantic rule
i.e
L.offset = null
L.place =id.place =x



SDD for Array to Produce TAC

S-> L: = E {if L.offset = null then
gen(L.place ‘ :=‘E.place);
/* Lis a id*/
else
gen(L.place ‘[‘ L.offset’]’ ‘:=‘
E.place);
}



Now ,
L.offset = null
And L.place = x

Thus,

x = t4 as E.place = t4

SDD for Array to Produce TAC

Thus, finally the TAC generated for x = A[y,z] with dimensions 10 * 20 and width 4 is :	t1 = y * 20 t1 = t1 + z; t2 = c(A) t3 = t1 * 4 t4 = t2[t3] x = t4

SDD for Assignments Statements to Produce TAC

S→id : =E	{ p := lookup (id.name)
	if p ≠ NIL then
	gen(p = E.place)
	else
	error /*id not declared */
	}

E→E1 + E2	{ Eplace := newtemp; gen(E.place := E1.place '+'E2.place }
E->E1 * E2	{ Eplace := newtemp; gen(E.place := E1.place '*'E2.place}

SDD for Assignments Statements to Produce TAC

E→ - E1	{ Eplace := newtemp; gen(E.place := 'minus' E1.place }
E→ (E1)	{ Eplace := E1.place}

E-> id	{ p := lookup (id.name)
	if p ≠ NIL then
	E.place = p
	else
	error
	}

Produce TAC for x : = a * b + c * d + e * f

SDD for Array to Produce TAC

Thus, finally the TAC generated for x = a*b + c* d + e* f	t1 = a * b t2 = c * d; t3 = t1 + t2 t4 = e * f t5 = t3 + t4 x = t5

SDD for Boolean Expressions as Arithmetic Expressions to Produce TAC

E→ E1 or E2	{ Eplace := newtemp; E.place := E1.place 'OR' E2.place }
E→ E1 and E2	{ Eplace := newtemp; E.place := E1.place 'AND' E2.place }
E-> not E1	{ Eplace := newtemp; E.place := 'NOT' E1.place }
E-> (E1)	{E.place:= E1.place; }
E->id1 relop id2	{E.place := newtemp; gen('if' id1.place RELOP id2.place 'goto' stmt +3); gen(E.place :=0); gen('goto' stmt+2); gen(E.place :=1); }

SDD for Boolean Expressions as Arithmetic Expressions to Produce TAC

E->true	{ Eplace := newtemp; gen(E.place ‘:= ‘ 1); }
E->false	{ Eplace := newtemp; gen(E.place ‘:= ‘ 0); }

Produce TAC for a or b and c < d and e < f

SDD for Boolean Expressions as Arithmetic Expressions to Produce TAC

Thus, finally the TAC generated for

a or b and c < d and e < f

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

SDD for Boolean Expressions As Control Flow to Produce TAC

E→E1 or E2	{ E1.true := E.true; E1.false := newlabel; E2.true:= E.true; E2.false := E.false; E.code := E1.code || gen(E1.false,':') || E2.code }
E→E1 and E2	{ E1.true := newlabel; E1.false := E.false; E2.true:= E.true; E2.false := E.false; E.code := E1.code || gen(E1.true,':') || E2.code }
E->not E1	{ E1.true := E.false; E1.false := E.true; E.code := E1.code }

SDD for Boolean Expressions As Control Flow to Produce TAC

E-> (E1)	{ E1.true := E.true; E1.false := E.false; E.code := E1.code }
E->id1 relop id2	E.Code := gen('if' id1.place relop.op id2.place 'goto' E.true) || gen('goto' E.false))
E->true	E.Code := gen('goto' E.true)
E->false	E.Code := gen('goto' E.false)

SDD for Boolean Expressions As Control Flow to Produce TAC

Code for a < b or c < d and e < f

```
        if a < b goto Ltrue
        goto L1
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse


Ltrue:
Lfalse:
```

Control flow translation of boolean expression …

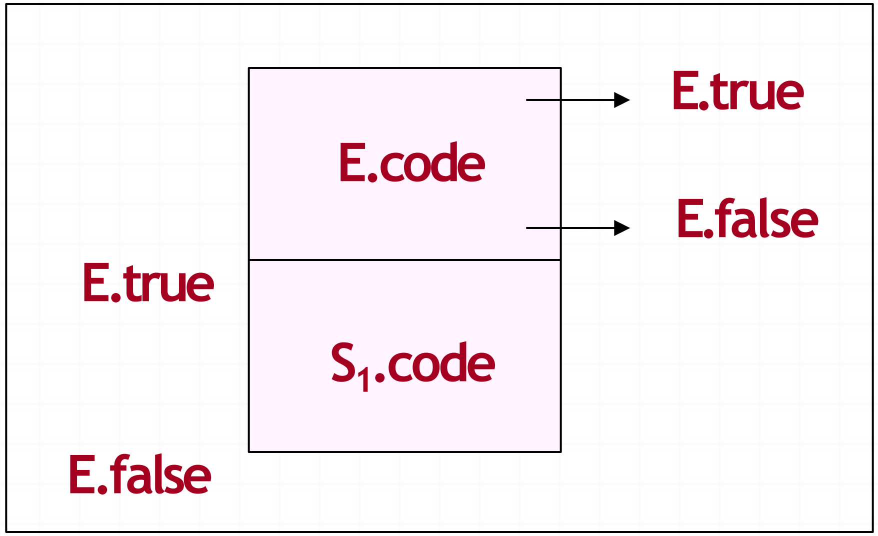
- Translate boolean expressions without:
  - generating code for boolean operators
  - evaluating the entire expression

- Flow of control statements

$S \rightarrow$ if E then S_1

| if E then S_1 else S_2

| while E do S_1



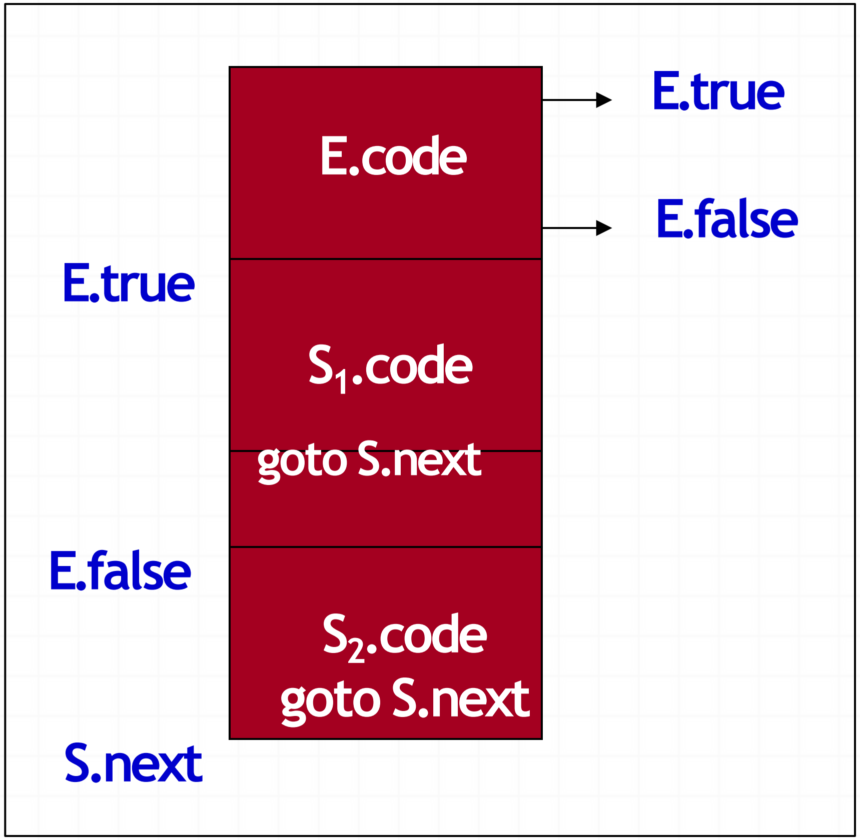
$$S \rightarrow \textbf{if } E \textbf{ then } S_1$$

$$E.true = newlabel$$

$$E.false = S.next$$

$$S_1.next = S.next$$

$$S.code = E.code \;||\; gen(E.true \; ':') \;||\; S_1.code$$



$S \rightarrow$ if E then S_1 else S_2

E.true = newlabel

E.false = newlabel

S_1.next = S.next

S_2.next = S.next

S.code = E.code | |

gen(E.true ':') | |

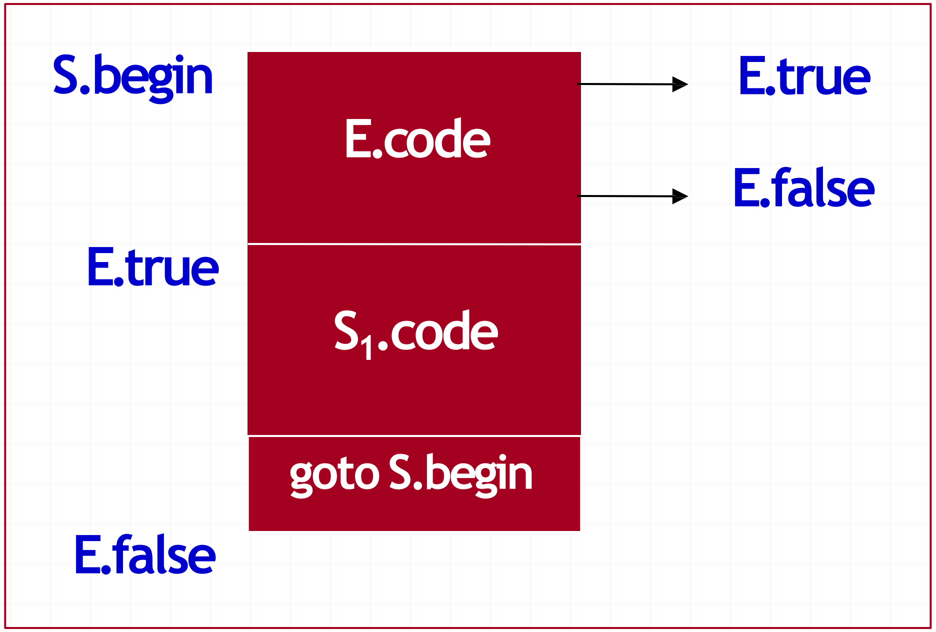
S_1.code | |

gen(goto S.next) | |

gen(E.false ':') | |

S_2.code||gen(goto S.next)

| | gen(S.next.':')



S→while E do S_1

S.begin = newlabel

E.true = newlabel

E.false = S.next

S_1.next = S.begin

S.ocde = gen(S.begin ':') | |
E.code | |
gen(E.true ':') | |
S_1.code | |
gen(goto S.begin) | |
gen(S.next, ':')

Flow of Control

$S \rightarrow$ while E do S_1

S.begin :

E.code

if E.place = 0 goto S.after

S_1.code

goto S.begin

S.after :

S.begin := newlabel

S.after := newlabel

S.code := gen(S.begin:) ||
E.code ||
gen(if E.place = 0 goto S.after) ||
S_1.code ||
gen(goto S.begin) ||
gen(S.after:)

Flow of Control …

$S \rightarrow$ if E then S_1 else S_2

E.code

if E.place = 0 goto S.else

S_1.code

goto S.after

S.else:

S_2.code

S.after:

S.else := newlabel

S.after := newlabel

S.code = E.code ||
gen(if E.place = 0 goto S.else) ||
S_1.code ||
gen(goto S.after) ||
gen(S.else :) ||
S_2.code ||
gen(S.after :)

Example ...

Code for while a < b do

if c<d then

x=y+z

else

x=y-z

```
L1:      if a < b goto L2
         goto Lnext
L2:      if c < d goto L3
         goto L4
L3:      t1 = Y+ Z
         X= t1
         goto S'next
L4:      t1 = Y- Z X=
         t1
         goto S'next
S'next : goto L1
Lnext:
```

FOLLOWING SLIDES NOT IN SYLLABUS 2012 …But JUST FOR REFERENCE

Case Statement

- switch expression
  begin
  case value: statement
  case value: statement
  ….
  case value: statement
  default: statement
  end

- evaluate the expression

- find which value in the list of cases is the same as the value of the expression.
  - Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression

- execute the statement associated with the value found

Translation

```
            code to evaluate E into t
            if t <> V1 goto L1
            code for S1
            goto next
L1          if t <> V2 goto L2
            code for S2
            goto next
L2:     .....
Ln-2        if t <> Vn-l goto Ln-l
            code for Sn-l
            goto next
Ln-1:       code for Sn
next:
```

```
            code to evaluate E into t
            goto test
L1: code for S1
            goto next
L2: code for S2
            goto next
            ......
Ln: code for Sn
            goto next
test:       if t = V1 goto L1
            if t = V2 goto L2
            ....
            if t = Vn-1 goto Ln-1
            goto Ln
next:
```

Efficient for n-way branch

Back Patching

- way to implement boolean expressions and flow of control statements in one pass
- code is generated as quadruples into an array
- labels are indices into this array
- makelist(i): create a newlist containing only i, return a pointer to the list.
- merge(p1,p2): merge lists pointed to by p1 and p2 and return a pointer to the concatenated list
- backpatch(p,i): insert i as the target label for the statements in the list pointed to by p

Boolean Expressions

$E \rightarrow E_1 \text{ or } M\ E_2$
$\quad |\ E_1 \text{ and } M\ E_2$
$\quad |\ \text{not } E_1$
$\quad |\ (E_1)$
$\quad |\ id_1 \text{ relop } id_2$
$\quad |\ \text{true}$
$\quad |\ \text{false}$
$M \rightarrow \epsilon$

- Insert a marker non terminal M into the grammar to pick up index of next quadruple.
- attributes truelist and falselist are used to generate jump code for boolean expressions
- incomplete jumps are placed on lists pointed to by E.truelist and E.falselist

Boolean expressions …

- Consider $E \rightarrow E_1$ and M E_2
  - if E_1 is false then E is also false so statements in E_1.falselist become part of E.falselist
  - if E_1 is true then E_2 must be tested so target of E_1.truelist is beginning of E_2
  - target is obtained by marker M
  - attribute M.quad records the number of the first statement of E_2.code

$E \rightarrow E_1$ or $M\ E_2$

backpatch(E_1.falselist, M.quad)

E.truelist = merge(E_1.truelist, E_2.truelist)

E.falselist = E_2.falselist

$E \rightarrow E_1$ and $M\ E_2$

backpatch(E_1.truelist, M.quad)

E.truelist = E_2.truelist

E.falselist = merge(E_1.falselist, E_2.falselist)

$E \rightarrow$ not E_1

E.truelist = E_1falselist

E.falselist = E_1.truelist

$E \rightarrow (E_1)$

E.truelist = E_1.truelist

E.falselist = E_1.falselist

$E \rightarrow id_1$ relop id_2

E.truelist = makelist(nextquad)

E.falselist = makelist(nextquad+ 1)

emit(if id_1 relop id_2 goto ---)

emit(goto ---)

$E \rightarrow$ true

E.truelist = makelist(nextquad)

emit(goto ---)

$E \rightarrow$ false

E.falselist = makelist(nextquad)

emit(goto ---)

$M \rightarrow \epsilon$

M.quad = nextquad

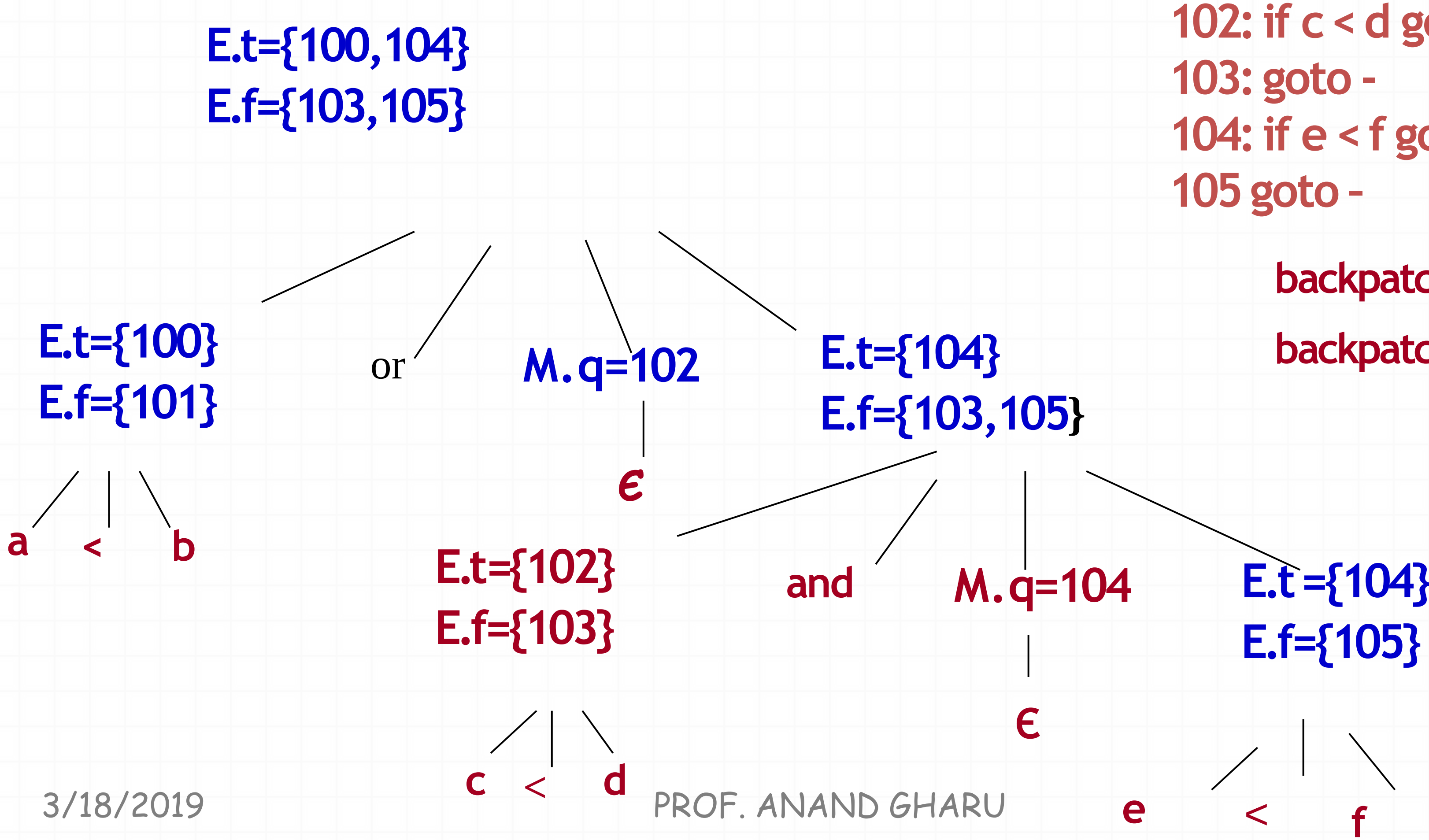
Generate code for
a<b or c<d and e<f

Initialize nextquad to 100

100: if a < b goto -
101: goto - 102
102: if c < d goto - 104
103: goto -
104: if e < f goto -
105 goto -

backpatch(102,104)

backpatch(101,102)



Procedure Calls

S → call id (Elist)

Elist → Elist , E

Elist → E

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address
  - jump to the beginning of the procedure

Procedure Calls …

Example

- parameters are passed by reference
- storage is statically allocated
- use param statement as place holder for the arguments
- called procedure is passed a pointer to the first parameter
- pointers to any argument can be obtained by using proper offsets

Code Generation

- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

S→ call id (Elist)
  { count = 0;
  for each item p on queue do {
    emit('param' p) ; count = count +1; }
  emit('call' id.place,count)

Elist → Elist , E
  append E.place to the end of queue

Elist → E
  initialize queue to contain E.place

THANK YOU!!!!!!!!!!!

My Blog : anandgharu.wordpress.com