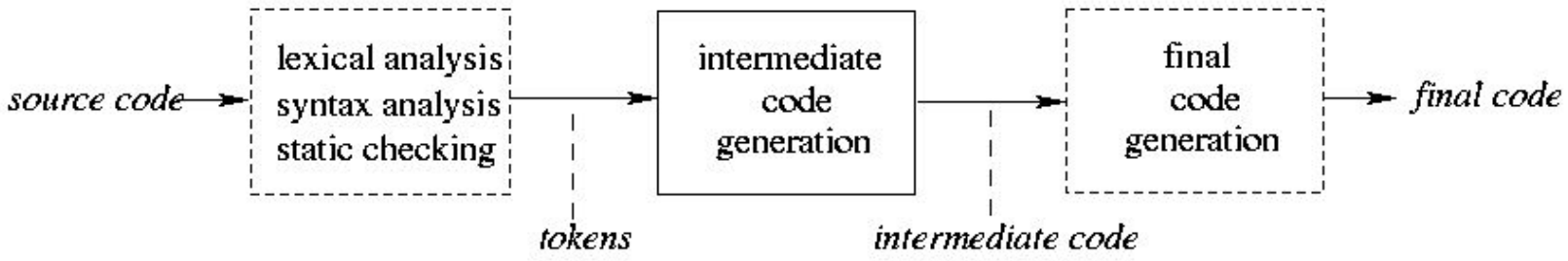
23Z602 COMPILER DESIGN
Unit-4
INTERMEDIATE CODE GENERATION

Dr.C.Kavitha
Associate Professor
Dept. of CSE
PSG College of Technology

INTERMEDIATE CODE GENERATION:

Benefits- Intermediate Languages - Generation of Three Address Code - Declarations - Assignment Statements - Arrays - Boolean Expressions - Backpatching - Flow of Control Statements –Procedure calls.

Overview



- Intermediate representations span the gap between the source and target languages
- **Intermediate code is:**
  - closer to target language
  - Relatively machine-independent and allows the compiler to be retargeted
  - allows many optimizations to be done in a machine-independent way.
- Implementable via syntax directed translation, so can be folded into the parsing process.
- In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form

Intermediate Code Generation

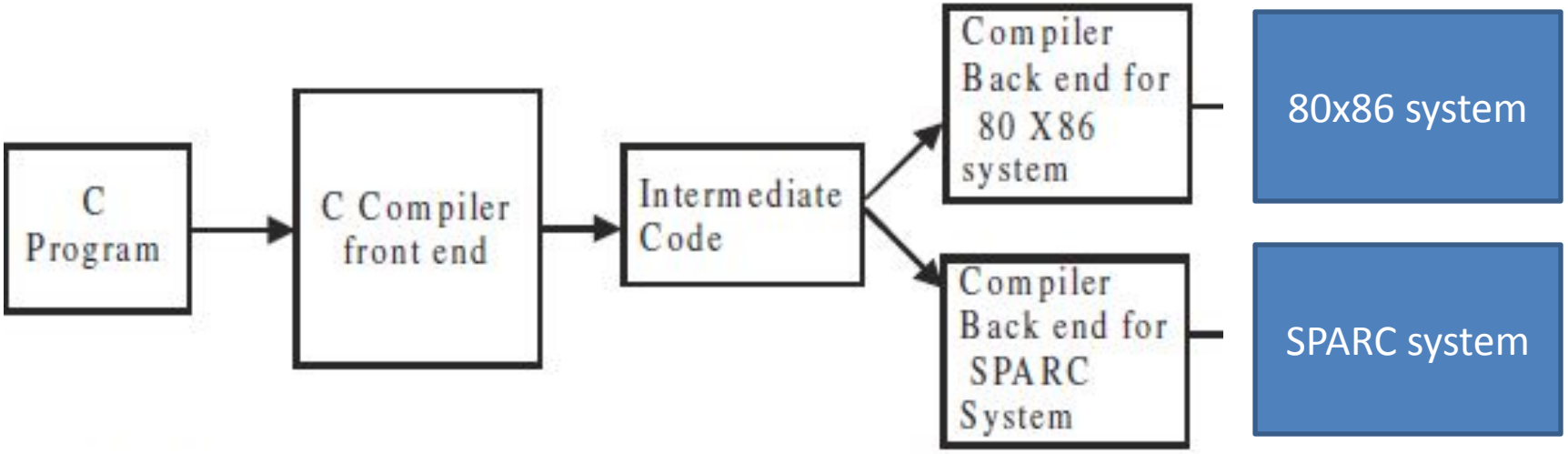
- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - abstract syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language

Benefits

1. Suppose We have n-source languages and m-Target languages. Without Intermediate code we will change each source language into target language directly. So, for each source-target pair we will need a compiler. Hence we will require (n*m) Compilers, one for each pair. If we Use Intermediate code We will require n-Compilers to convert each source language into Intermediate code and m-Compilers to convert Intermediate code into m-target languages. Thus We require only (n+m) Compilers.
2. Retargeting is facilitated; a compiler for a different machine can be created by attaching a Back-end (which generate Target Code) for the new machine to an existing Front-end (which generate Intermediate Code).
3. A machine Independent Code-Optimizer can be applied to the Intermediate code.
4. Intermediate code is simple enough to be easily converted to any target code.
5. Complex enough to represent all the complex structure of high level language.

Need for ICG

- It facilitates machine independent code optimization. ICG by the front end can be optimized by using several techniques.
- It is easy to re-target the compiler to generate code for newer and different processors.



Types of Intermediate Languages

- *High Level Representations* (e.g., syntax trees):
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward.
- *Low Level Representations* (e.g., 3-address code, RTL(Register Transfer Language)):
  - closer to the target machine;
  - easier for optimizations, final code generation

INTERMEDIATE LANGUAGES

- Three ways of intermediate representation:
  - Abstract Syntax Tree
  - Postfix notation
  - Three address code
- The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Postfix notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
- The postfix notation for the same expression places the operator at the right end as ab +.
- In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +.
- No parentheses are needed in postfix notation because the position and number of arguments of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.
- **Infix Postfix**

  a + b a b +

  a + b * c a b c * +

  (a + b) * c a b + c *

Evaluation of Postfix Notation

- Given an arithmetic expression in reverse Polish (Postfix) notation it is easy to evaluate directly from left to right.
- Often used in interpreters.
- We need a stack for storing intermediate results.
- If numeric value: Push the value onto the stack.
- If identifier: Push the value of the identifier (r-value) onto the stack.
- If binary operator: Pop the two uppermost elements , apply the operator to them and push the result.
- If unary operator: Apply the operator directly to the top of the stack.
- When the expression is completed, the result is on the top of the stack.

Assignment Statement

- x := 10 + k * 30 x 10 k 30 * + :=

Conditional Statement

- If-thenElse Statements
- if a+b then

  if c-d then

  x := 10

  else y := 20

  else z := 30;

Postfix Notation

a b + L1 JEQZ

c d - L2 JEQZ

x 10 := L3 JUMP

L2: y 20 := L4 JUMP

L1: z 30 := L3: L4:

postfix

Ex1: x = c ? a : b

x c a b : ? =

Ex2:

```
if (f < 6)
z = p / q;
else if ( f == 6)
z=p*q;
else
z=p-q
```

f 6 < | z p q / = | f 6 == z p q * = z p q - = : ? : ?

l jump

e l jeqz

e1 e2 l jlt

Stack is used for evaluation

Advantages:

a) It eliminates the need for having parenthesis and has an unambiguous order of evaluation

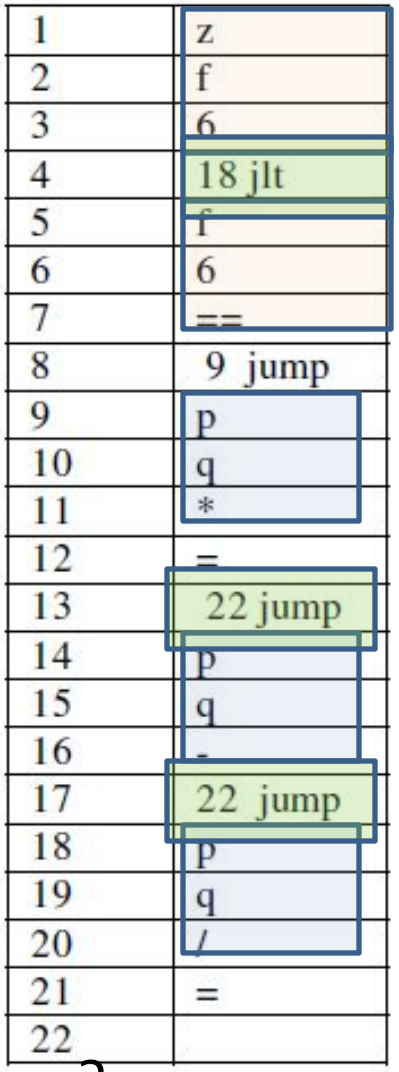
b) It is reasonably easy to generate postfix code from the input source pro gram.

c) It is compact in terms of storage for the instructions.

Disadvantages

a) does not lend itself to be effectively for optimization.

b) This notation is ineffective for target code generation.



z f 6 < | p q / | f 6 == | p q * | p q - | : | ? | : | ? | =

```
while (e < 7)
{
if (f < 6)
   z = p/q;
else
   e = e+1;
}
```

1	e
2	7
3	5 jlt
4	20 jump
5	f
6	6
7	9 jlt
8	14 jump
9	z
10	p
11	q
12	/
13	=
14	e
15	e
16	1
17	+
18	=
19	1 jump
20	

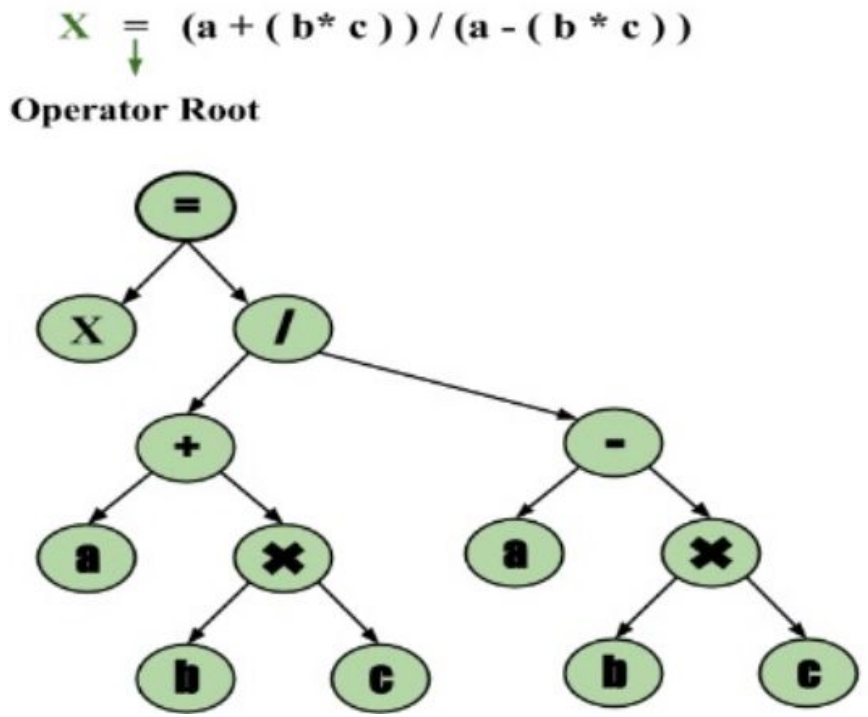
Syntax Trees

- Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first

Syntax Trees

Example –

x = (a + b * c) / (a – b * c)



Syntax Trees: Example

Grammar :

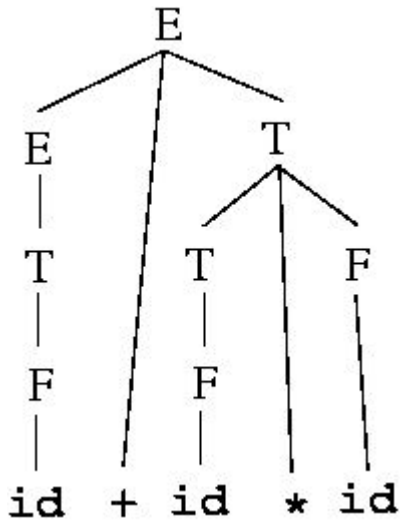
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

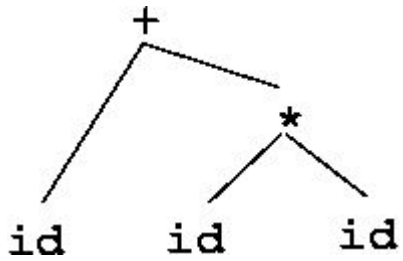
$F \rightarrow (E) \mid \mathbf{id}$

Input: **id + id * id**

Parse tree:



Syntax tree:

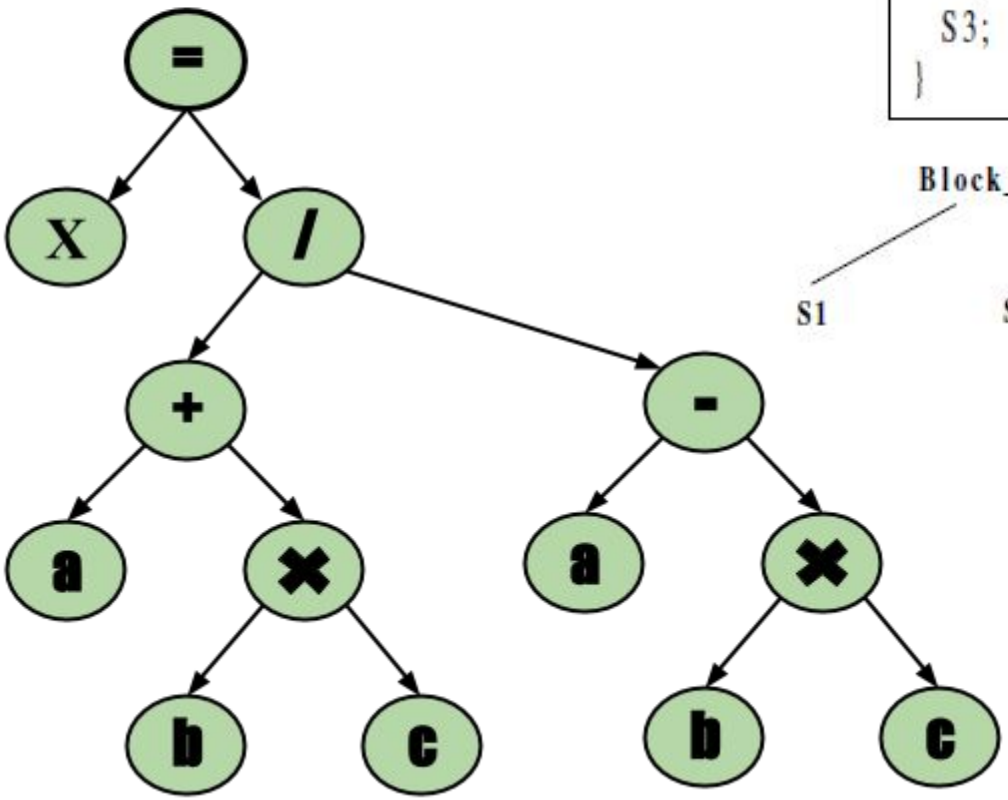


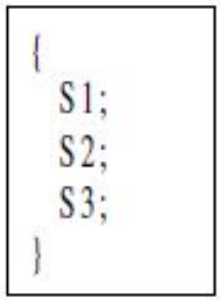
AST

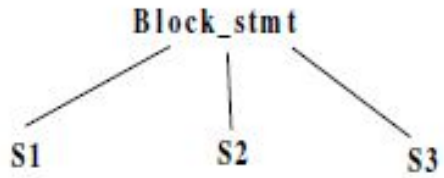
- Condensed form of parse tree

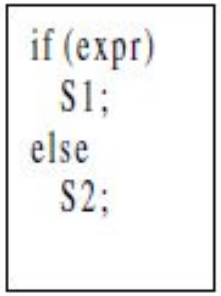
X = (a + (b* c)) / (a - (b * c)) jw.

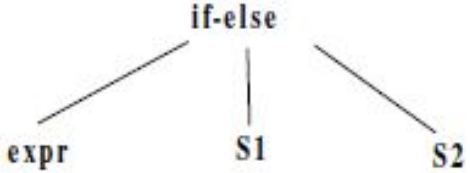
Operator Root



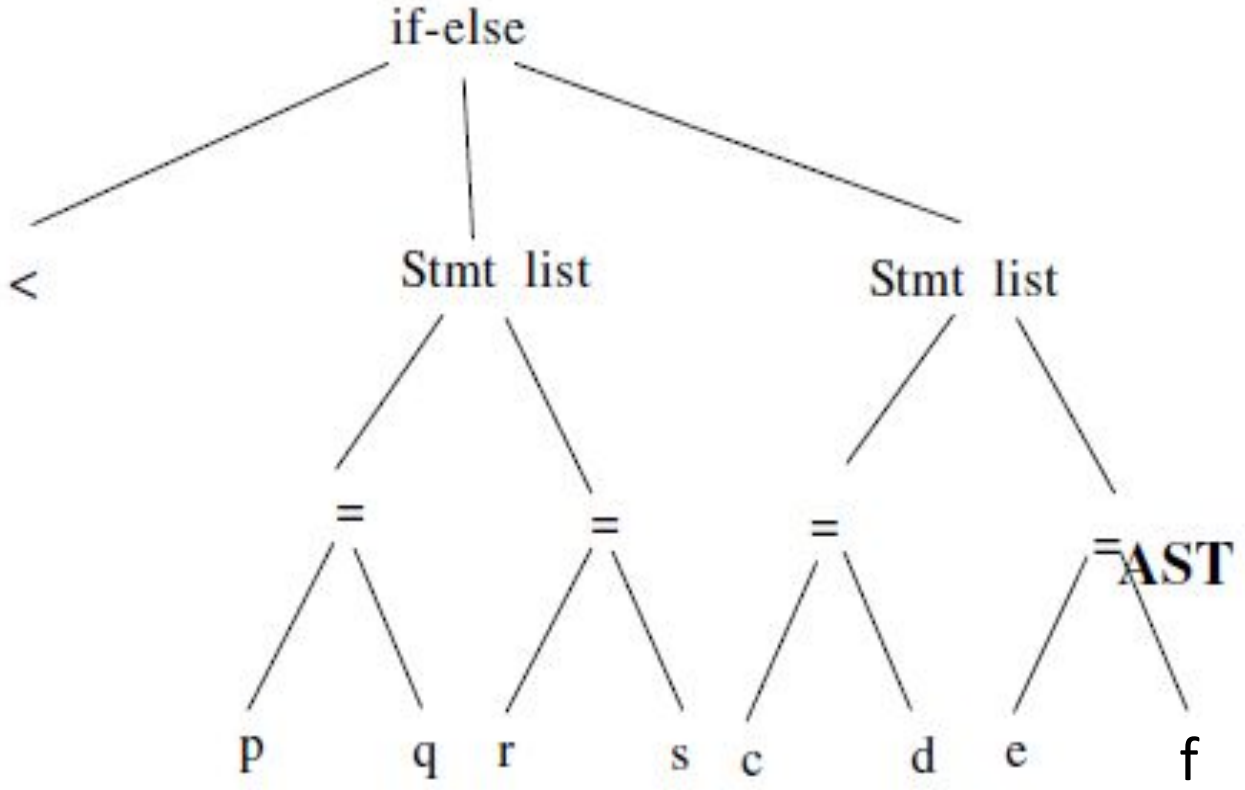








```
if(a<b)
{
    p = q;
    r = s;
}
else
{
   c = d;
   e = f;
}
```



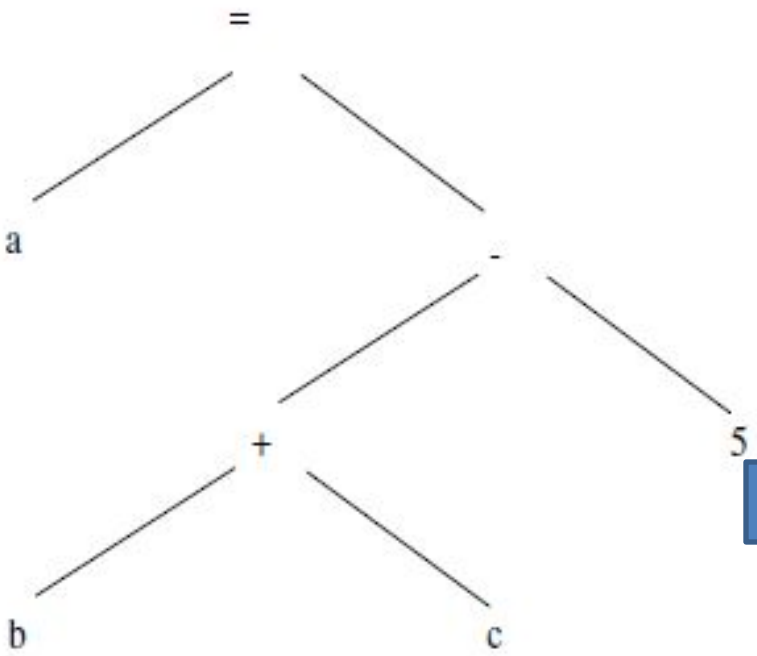
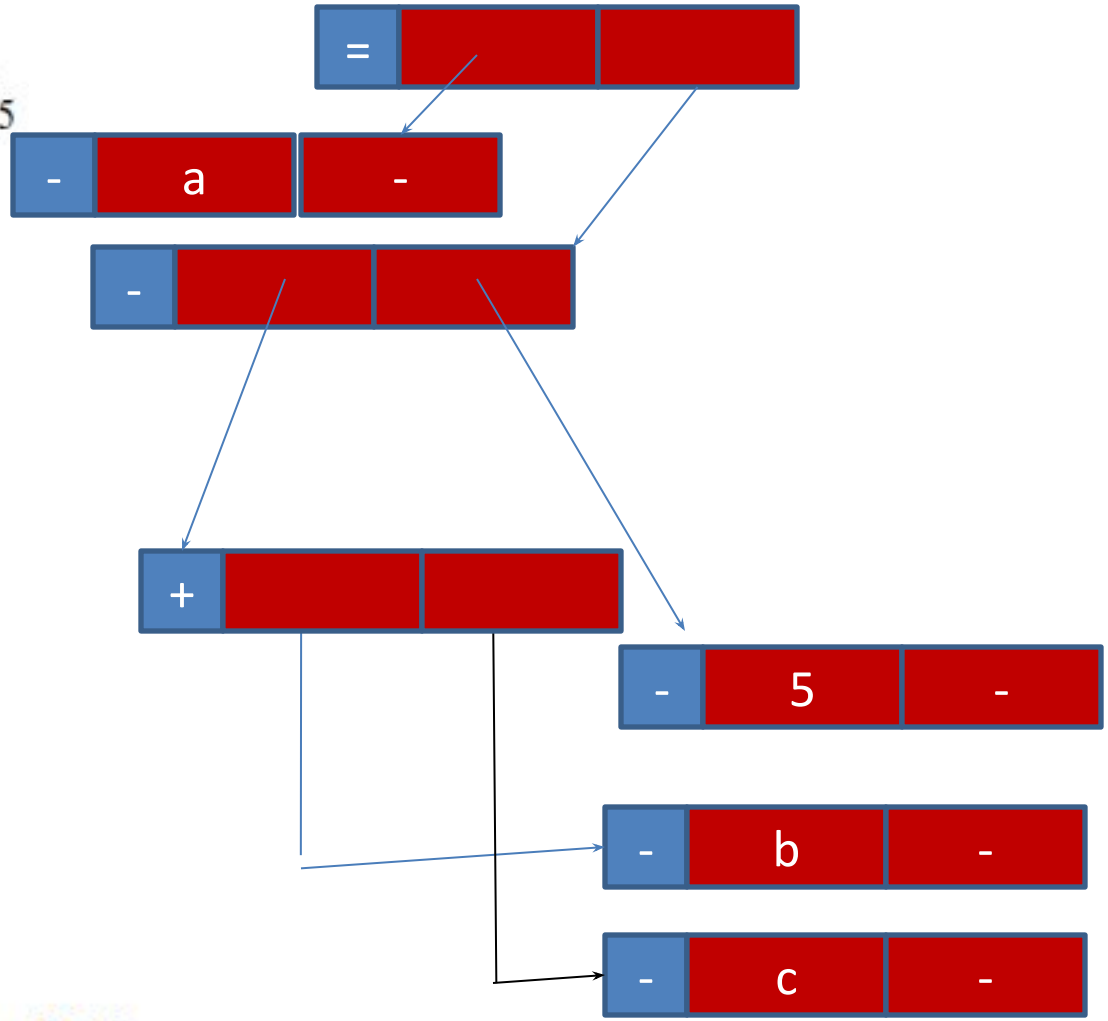


Fig 6.4 a: AST for a=b+c-5

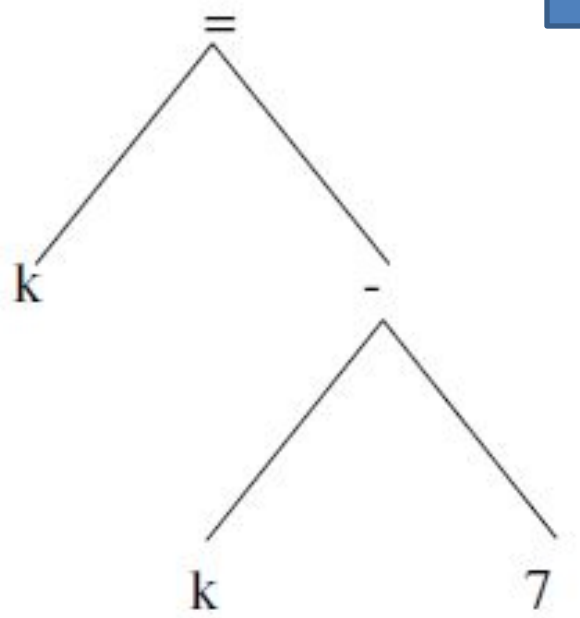


0	Identifier	a	
1	Identifier	b	
2	Identifier	c	
3	Number	5	
4	+	1	2
5	-	4	3
6	=	0	5

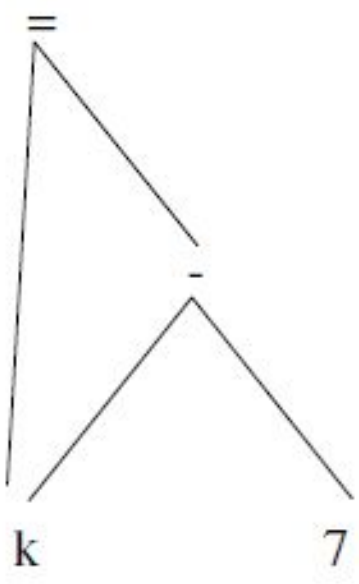
Fig 6.4 c: Array representation for AST

- Advantages
  - Lends itself for machine independent code optimization by code reorganisation
- Disadvantages
  - Consumes lot of memory , so DAG can be used

k=k-7

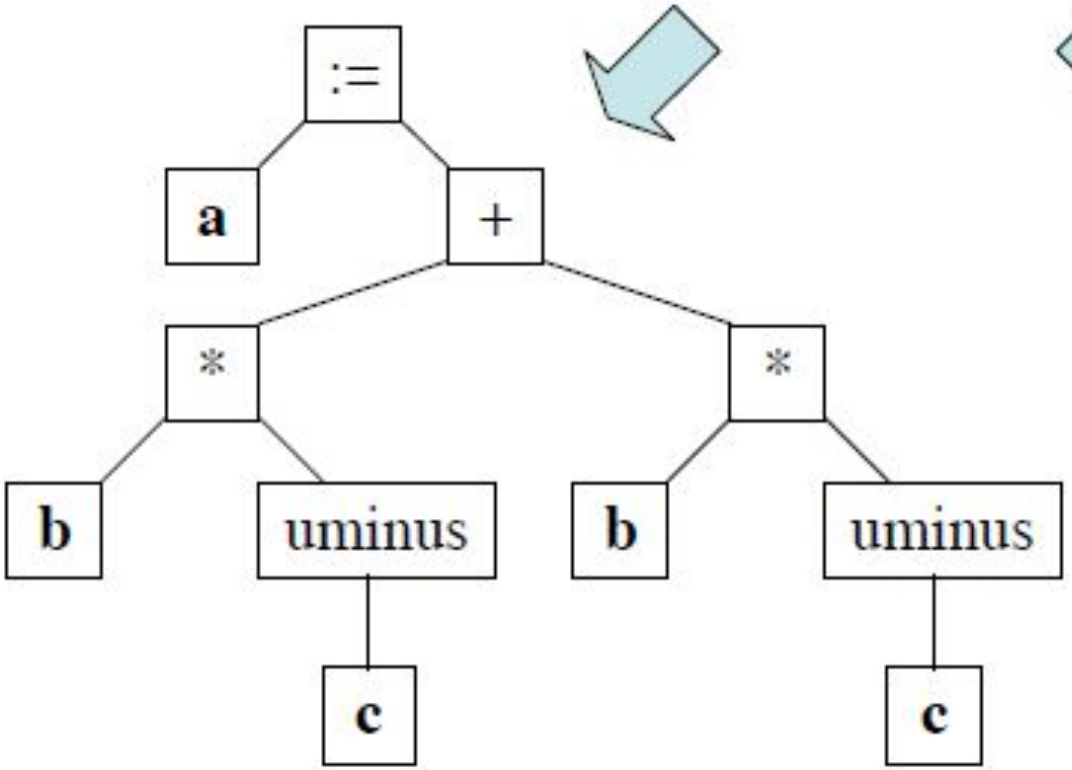


AST

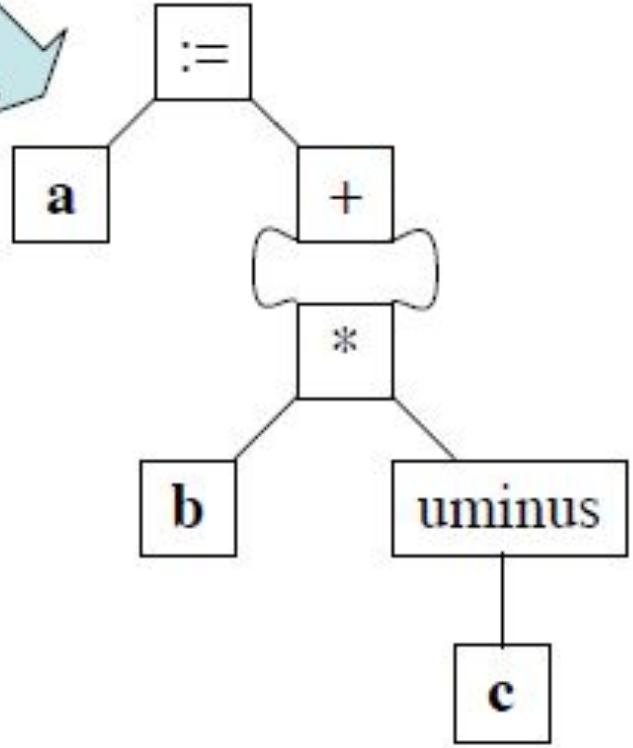


DAG

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



Tree



DAG

Three Address Code

- **A=B op C** …. each statement consists of three addresses, two for the source operands (A and B) and one for the result (C).
- Unary: C: = op B

- Assignment statements: $x := y\ op\ z$, $x := op\ y$
- Indexed assignments: $x := y[i]$, $x[i] := y$
- Pointer assignments: $x := \&y$, $x := *y$, $*x := y$
- Copy statements: $x := y$
- Unconditional jumps: `goto` *lab*
- Conditional jumps: `if` *x relop y* `goto` *lab*
- Function calls: `param` *x*… `call` *p, n*
  `return` *y*

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

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

TAC for AST

```
t1 := - c
t2 := b * t1
t3 := t2 + t2
a := t3
```

TAC for DAG

```
i := 2 * n + k
while i do
        i := i – k
```

```
t1 := 2
t2 := t1 * n
t3 := t2 + k
i := t3
L1: if i = 0 goto L2
t4 := i - k
i := t4
goto L1
L2:
```

Implementation of TAC

- **Quadruples**: A quadruple is a record with 4 fields. The fields are operator code, argument1, argument2 and the result

#	*Op*	*Arg1*	*Arg2*	*Res*

- Pro: easy to rearrange code for global optimization
- Cons: lots of temporaries

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

t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Triples

- Record has three fields - operator code, argument1 and argument2.

#	*Op*	*Arg1*	*Arg2*

- Pro: temporaries are implicit
- Cons: difficult to rearrange code

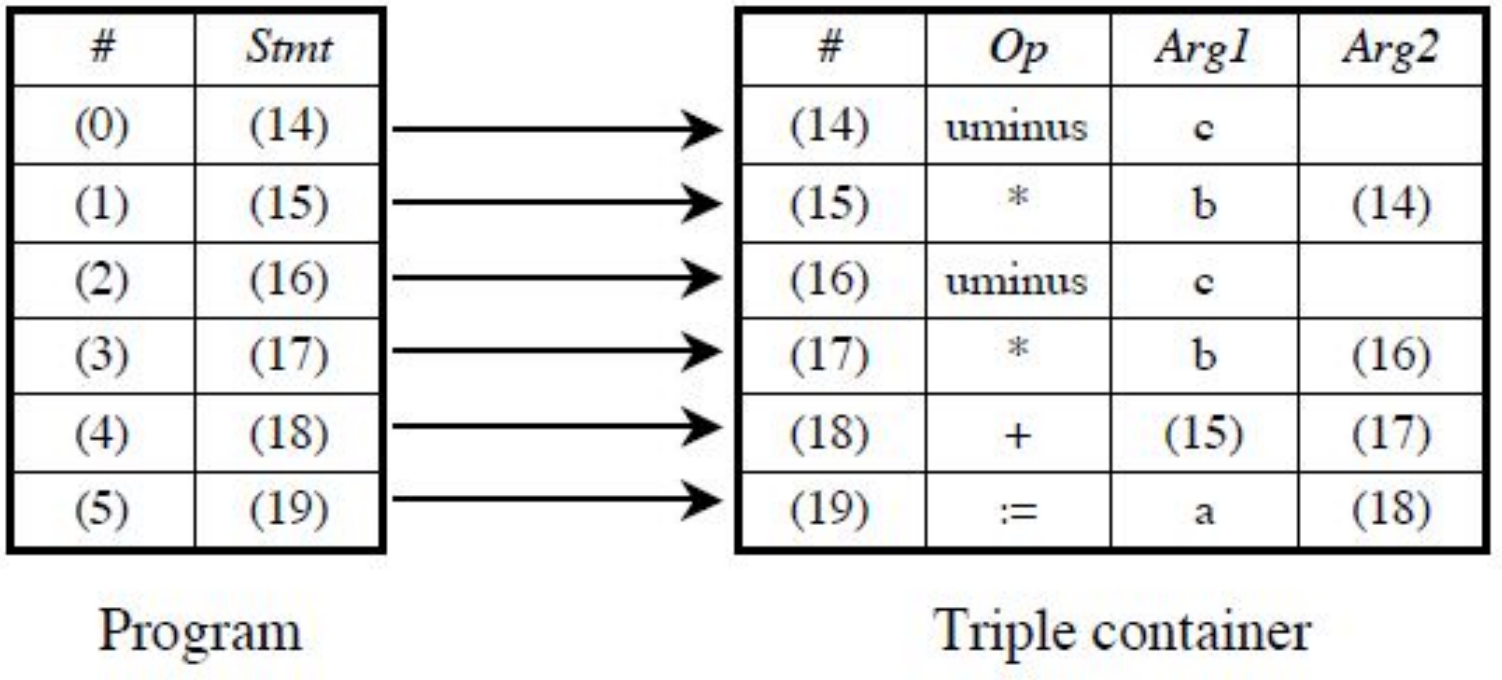
a := b * -c + b * -c

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Indirect Triples

- three address code, the listing consists of pointers to triples rather than the triples themselves
- Pro: temporaries are implicit & easier to rearrange code
- **a := b * -c + b * -c**



#	Stmt
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

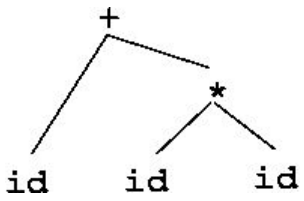
Triple container

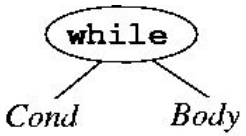
Comparison

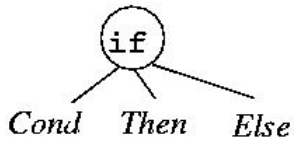
- Indirection – Quadruples (no); Triples (some extent) – best is indirect Triples
- Suitability for optimisation – Quadruples (good); Triples (not good)
- Space – Quadruples (more); Triples (least)

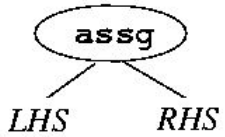
Syntax Trees: Structure

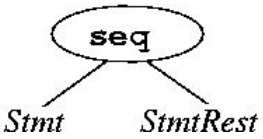
- Expressions:
  - leaves: identifiers or constants;
  - internal nodes are labeled with operators;
  - the children of a node are its operands.
- Statements:
  - a node's label indicates what kind of statement it is;
  - the children correspond to the components of the statement.











- Three-address code is a sequence of statements of the general form x : = y *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
- t_1 : = y * z t_2 : = x + t_1
- where t_1 and t_2 are compiler-generated temporary names.

- Some three-address statements that will be used are:
  - Assignment statements:
    - With a binary operation: x := y op z
    - With a unary operation: x:= op y
    - With no operation(copy) : x := y
  - Branch statements
    - Unconditional jump: goto L
    - Conditional jumps: if x relop y goto L
  - Statement for procedure calls
    - Param x, set a parameter for a procedure call
    - Call p, n call procedure p with n parameters
    - Return y return from a procedure with return value y

- Example: instructions for procedure call: p(x1, x2, x3, …, xn):

```
param x1
param x2
…
param xn
call p, n
```

- Indexed assignments:
  - x := y[i] and x[i] := y
- Address and pointer assignments
  - x := &y, x := *y

An Intermediate Instruction Set

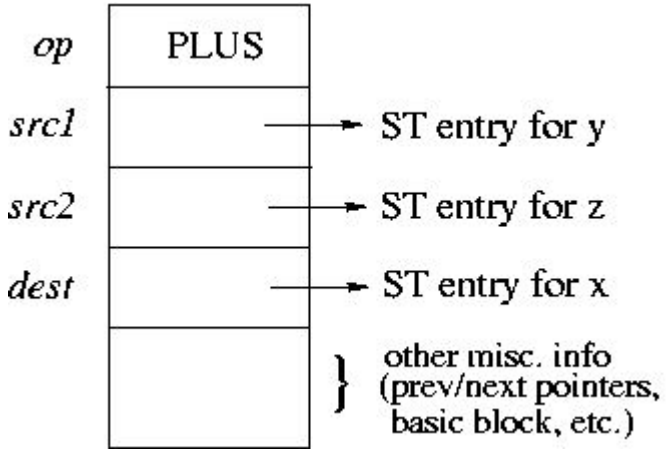
- *Assignment*:
  - x = y *op* z (*op* binary)
  - x = *op* y (*op* unary);
  - x = y
- *Jumps*:
  - if (x *op* y) goto L (L a label);
  - goto L
- *Pointer and indexed assignments*:
  - x = y[z]
  - y[z] = x
  - x = &y
  - x = *y
  - *y = x.
- *Procedure call/return*:
  - param x, k (x is the k^{th} param)
  - retval x
  - call p
  - enter p
  - leave p
  - return
  - retrieve x
- *Type Conversion*:
  - x = cvt_A_to_B y (A, B base types) e.g.: cvt_int_to_float
- *Miscellaneous*
  - label L

Three Address Code: Representation

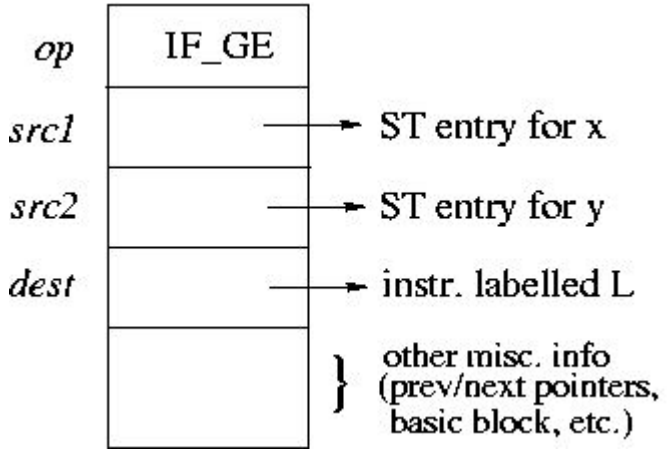
- Each instruction represented as a structure called a *quadruple* (or "*quad*"):
  - contains info about the operation, up to 3 operands.
  - for operands: use a bit to indicate whether constant or ST pointer.

E.g.:

x = y + z



if (x ≥ y) goto L



Representation of three-address codes

- Three-address code can be represented in various forms viz**. Quadruples, Triples and Indirect Triples**. These forms are demonstrated by way of an example below.

Example:

- A = -B * (C + D)

Three-Address code is as follows:

T1 = -B

T2 = C + D

T3 = T1 * T2

A = T3

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

- b * minus c + b * minus c

Three address code

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

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Quadruple:

	Operator	*Operand 1*	*Operand 2*	*Result*
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

Triple:

	Operator	*Operand 1*	*Operand 2*
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	Statement
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	Operator	*Operand 1*	*Operand 2*
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

- 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.

Production	Semantic Rule
E->E1+T	E.code=E1.code||T.code||'+'

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions

- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be of many kinds: numbers, types, table references, strings, etc.
- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and at N it
- Inherited attributes
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings

Example of S-attributed SDD

	Production	Semantic Rules
1)	L -> E n	L.val = E.val
2)	E -> E1 + T	E.val = E1.val + T.val
3)	E -> T	E.val = T.val
4)	T -> T1 * F	T.val = T1.val * F.val
5)	T -> F	T.val = F.val
6)	F -> (E)	F.val = E.val
7)	F -> digit	F.val = digit.lexval

Example of mixed attributes

Production	Semantic Rules
1) T -> FT’	T’.inh = F.val T.val = T’.syn
2) T’ -> *FT’$_1$	T’$_1$.inh = T’.inh*F.val T’.syn = T’$_1$.syn
3) T’ -> ε	T’.syn = T’.inh
1) F -> digit	F.val = F.val = digit.lexval

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.c to B.c

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then $i<j$
- Such an ordering is called a topological sortof a graph

S-Attributed definitions

- An SDD is S-attributed if every attribute is synthesized
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

```
postorder(N) {
        for (each child C of N, from the left) postorder(C);
        evaluate the attributes associated with node N;
}
```

- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there us a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

Syntax directed translation

- Conversion of source code to intermediate code based on the syntax of the language is called as Syntax Directed Translation
- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically SDT's are implemented during parsing without building a parse tree

Three-address code for expressions

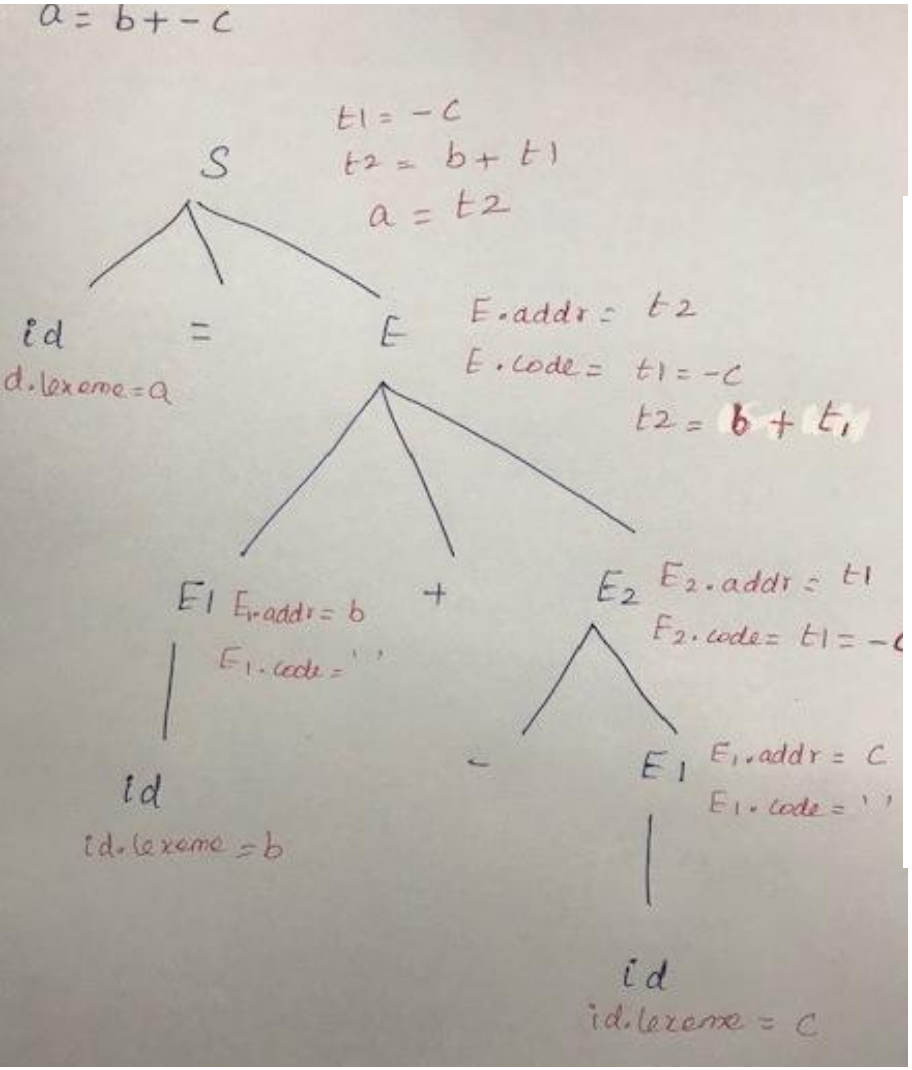
PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\ ()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\ ()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

- Attribute *code* for S and *code* and *addr* for E
- Attributes **S.code** and **E.code** denote the three-address code for S and E.
- **top** denotes the current symbol table.
- **top.get(id.lexeme)** –retrieves the symbol table entry for the id.

Example

- **Children attributes are computed before parents for a grammar having all synthesized attributes.**
- Move bottom up in left to right fashion for computing translation rules
- The flow of information happens **bottom-up** and **all the children attributes are computed before parents.**
- **Attributes are evaluated during bottom up parsing**

Example



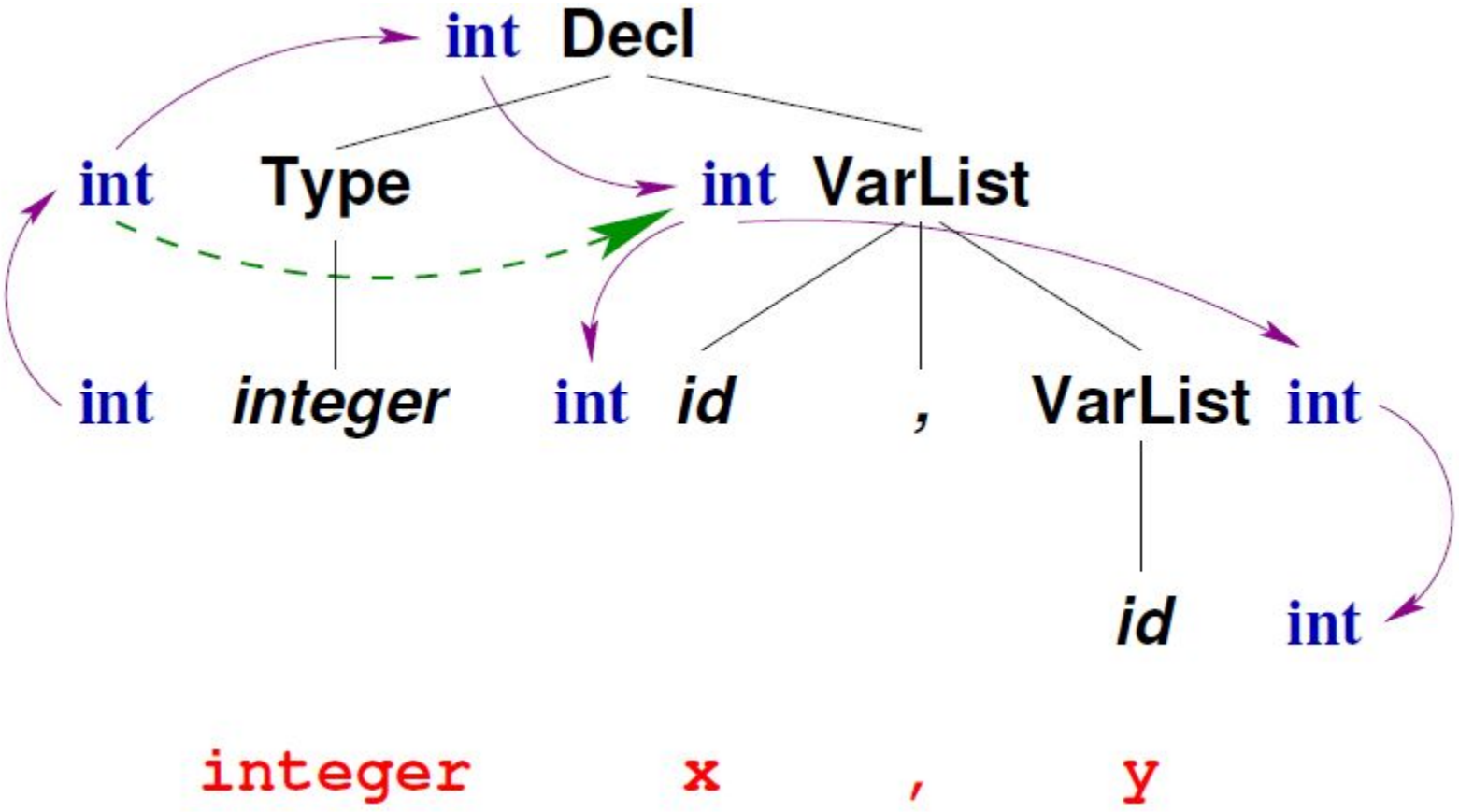
PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\ ()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp\ ()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Variable Declaration

$$
\begin{array}{rcl}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} \\
\mathit{Type} & \longrightarrow & \texttt{integer} \\
\mathit{Type} & \longrightarrow & \texttt{float} \\
\mathit{VarList} & \longrightarrow & \texttt{id}\ \texttt{,}\ \mathit{VarList} \\
\mathit{VarList} & \longrightarrow & \texttt{id}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} & \{\mathit{VarList}.\mathit{type} := \mathit{Type}.\mathit{type}\} \\
\mathit{Type} & \longrightarrow & \texttt{integer} & \{\mathit{Type}.\mathit{type} := \mathit{int}\} \\
\mathit{Type} & \longrightarrow & \texttt{float} & \{\mathit{Type}.\mathit{type} := \mathit{float}\} \\
\mathit{VarList} & \longrightarrow & \texttt{id}\ \texttt{,}\ \mathit{VarList}_1 & \{\mathit{VarList}_1.\mathit{type} := \mathit{VarList}.\mathit{type}; \\
 & & & \quad \texttt{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\} \\
\mathit{VarList} & \longrightarrow & \texttt{id} & \{\texttt{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\}
\end{array}
$$

Information Floe for **‘Type’**



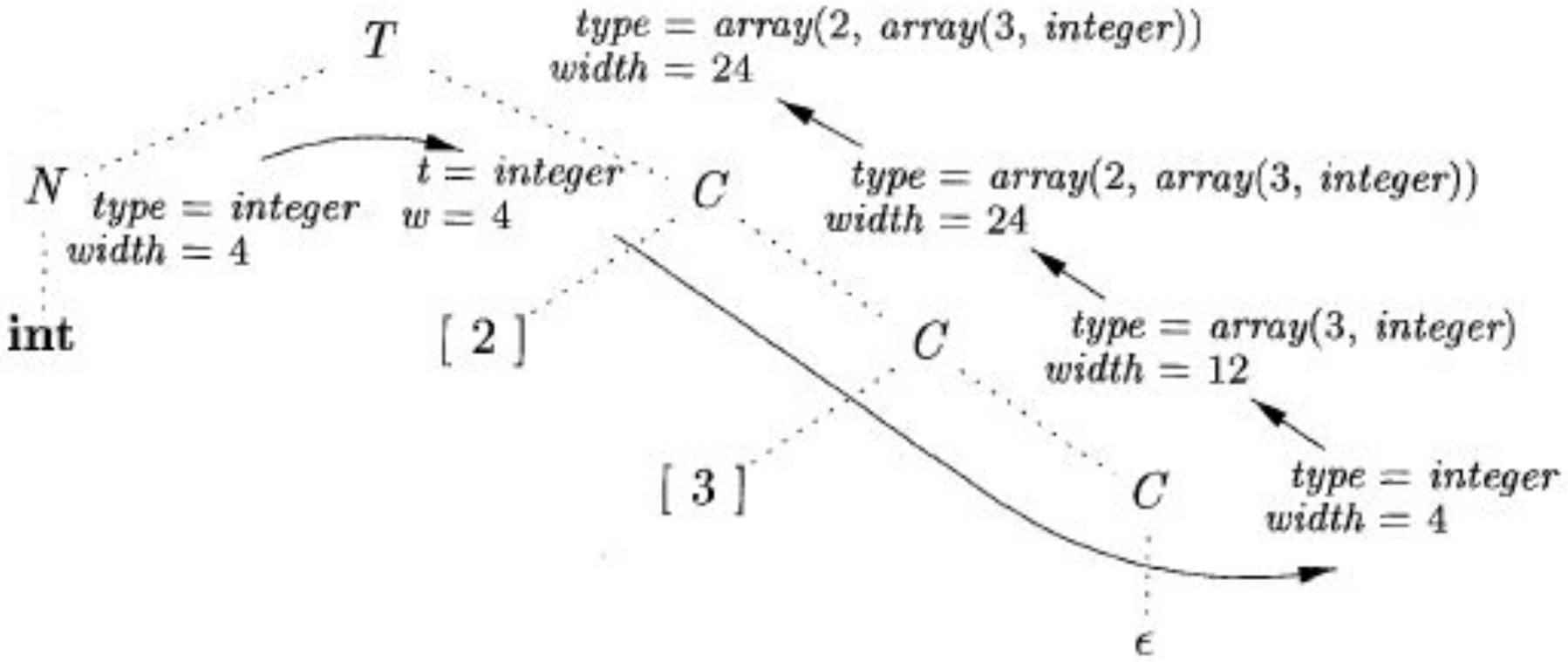
Storage Layout for Local Names

Computing types and their widths

$$
\begin{array}{lll}
T \rightarrow B & \{\ t = B.type;\ w = B.width;\ \} \\
\quad\ C & \\
B \rightarrow \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B \rightarrow \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C \rightarrow \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C \rightarrow [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

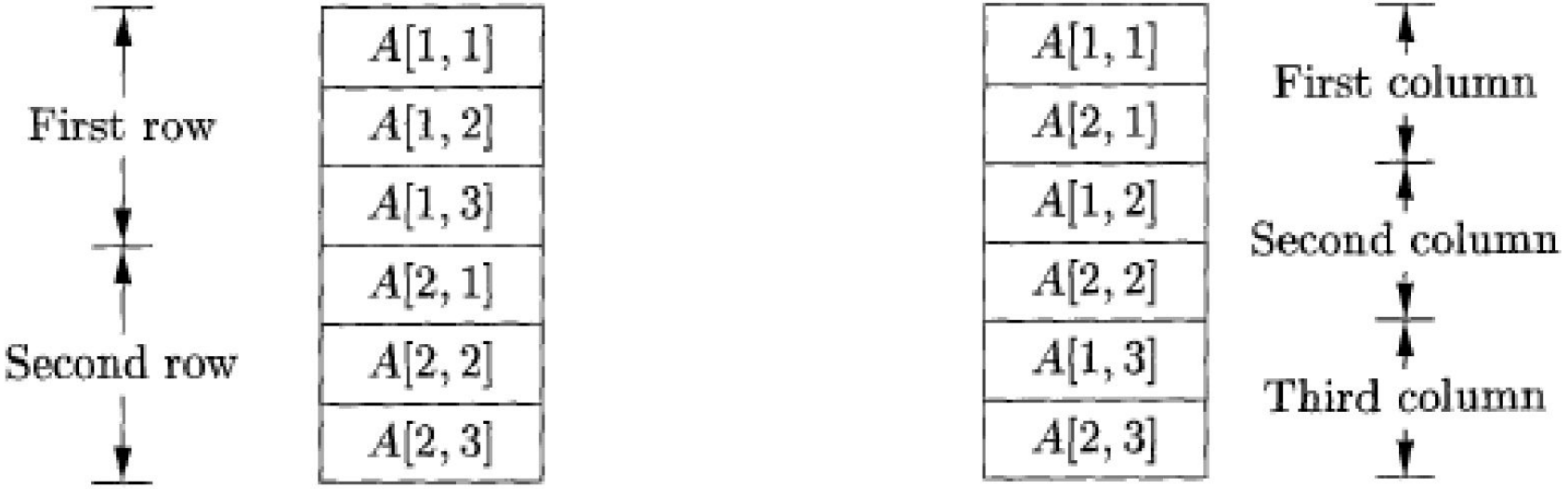
Syntax-directed translation of array types

int [2][3]



Addressing Array Elements

- Layouts for a two-dimensional array:



If the width of the array element is w, ith element of the Array A begins in the location base+I * w

A[i1,i2]--- base+i1*w1+i2*w2

For array A of size n×m×p

A[i][j][k] is (i×m×p+j×p+k) * size.

Or

(i×m×p×size) + (j×p× size) + (k×size)

or

((i*m + j)*p + k)*size

Semantic actions for Array reference

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }

| $L = E\ ;$ { $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);$ }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp\ ();$
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }

| $\mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme);$ }

| L { $E.addr = \mathbf{new}\ Temp\ ();$
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme);$
$L.type = L.array.type.elem;$
$L.addr = \mathbf{new}\ Temp\ ();$
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }

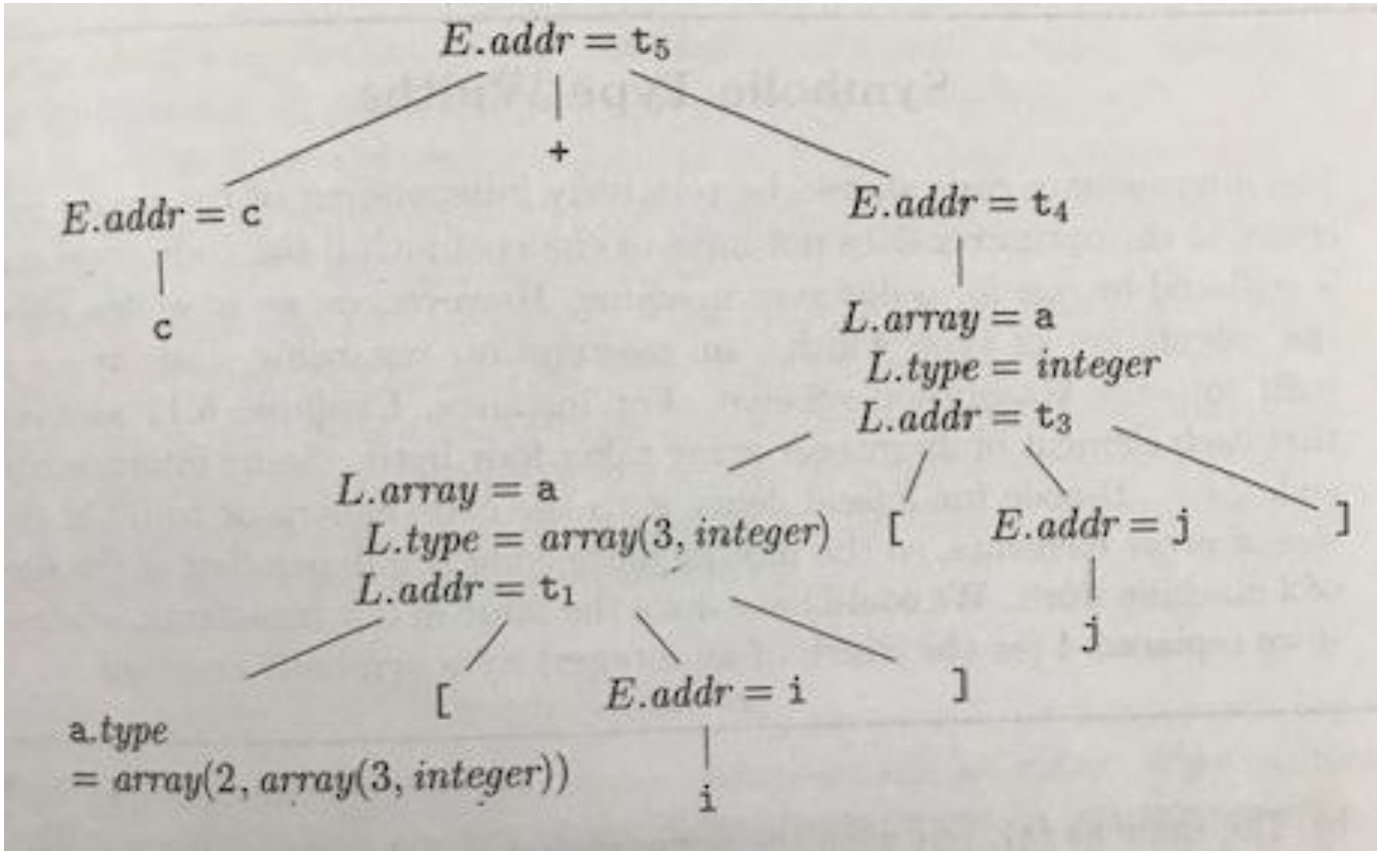
| $L_1\ [\ E\ ]$ { $L.array = L_1.array;$
$L.type = L_1.type.elem;$
$t = \mathbf{new}\ Temp\ ();$
$L.addr = \mathbf{new}\ Temp\ ();$
$gen(t\ '='\ E.addr\ '*'\ L.type.width);$ }
$gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translation of Array References

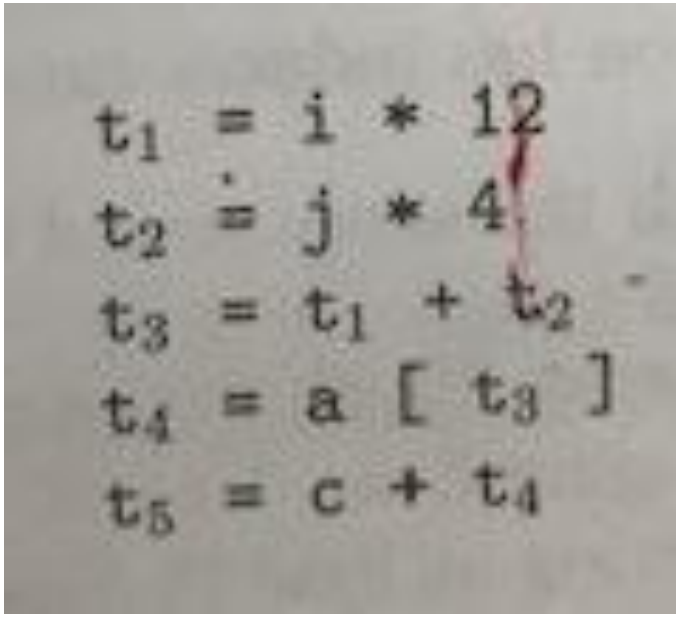
Nonterminal *L* has three synthesized attributes:

- ***L.addr***- denotes a temporary that is used while computing the offset for the array reference
- **L.array**- is a pointer to the symbol table entry for the array name
- **L.type** – is the type of the subarray generated by L

Annotated parse tree for c+a[i][j]



Three-adress code for expression c+a[i][j]



Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

Control Flow

boolean expressions are often used to:

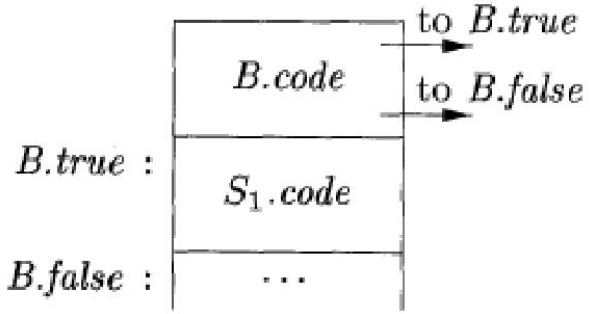
- *Alter the flow of control.*
- *Compute logical values.*

Flow-of-Control Statements

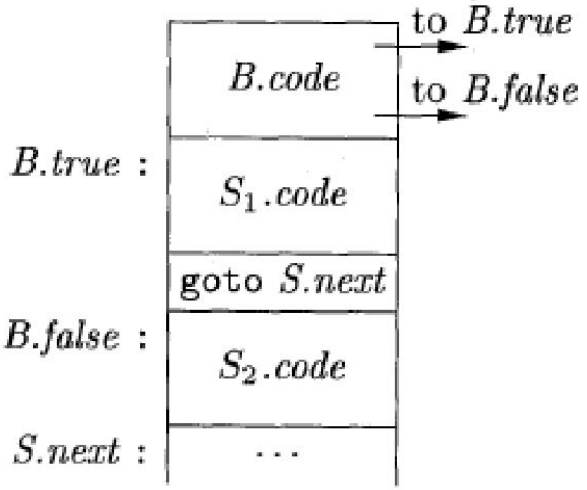
$S \rightarrow$ **if** (B) S_1

$S \rightarrow$ **if** (B) S_1 **else** S_2

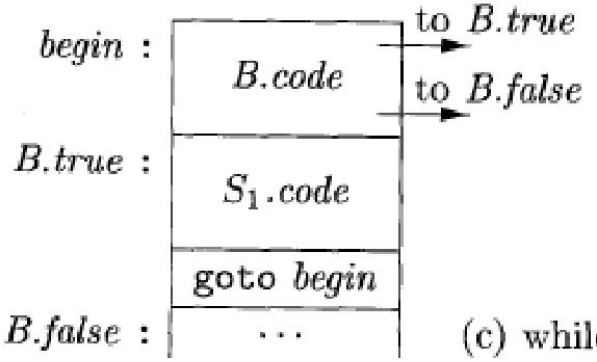
$S \rightarrow$ **while** (B) S_1



(a) if



(b) if-else



(c) while

Syntax-directed definition

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

- B and S have a Synthesized attribute ***code***
- Labels for the jumps in B.code and S.code are managed using inherited attributes.
- For S associate inherited attribute S.next denoting a label for the instruction immediately after the code for S.
- Newlabel()-creates a newlabel each time it is called.

Syntax-directed definition

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Example:
If(B) S1 else S2
A=D+E

P->S
S.next=L1
B.Code
L3: S1.Code
goto L2
L4: S2.code
L2: assign.code
L1:

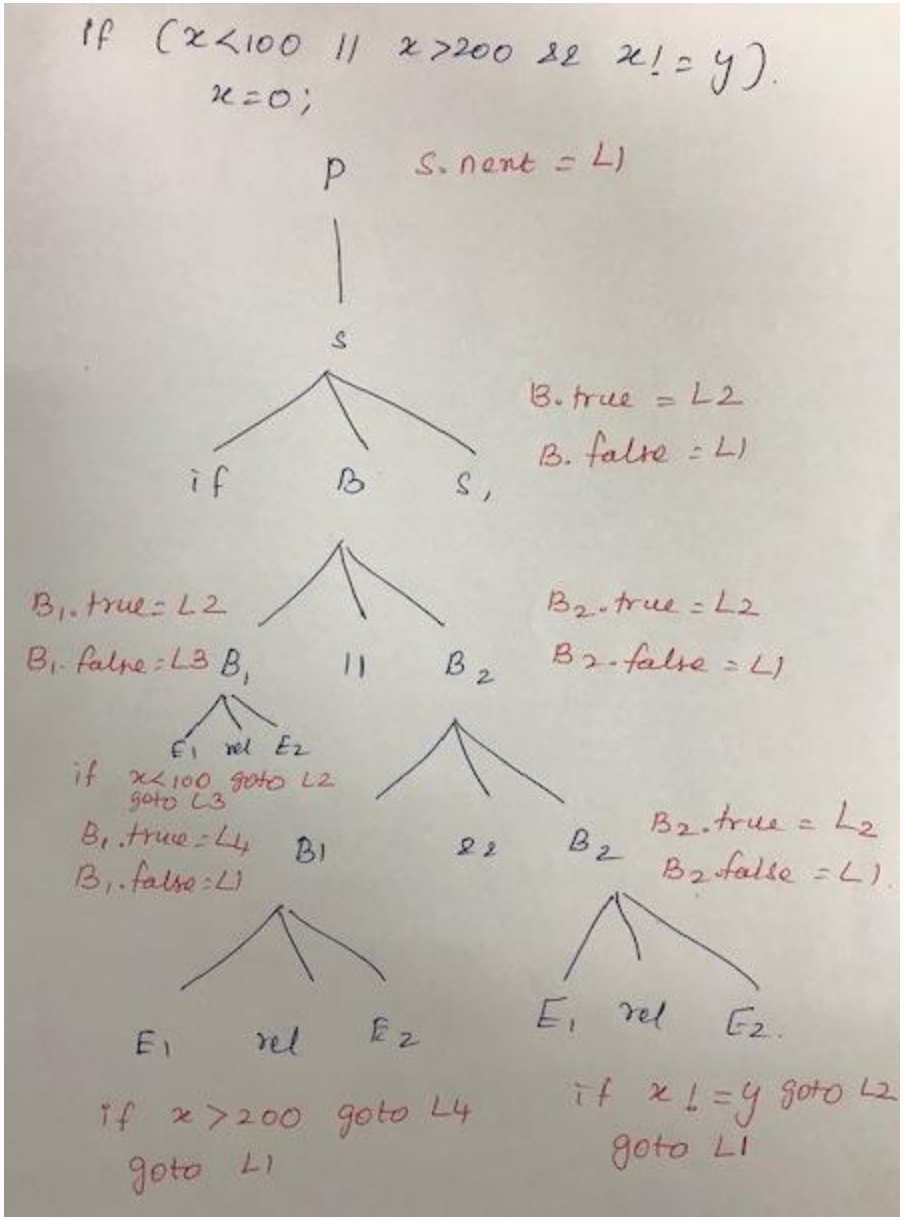
S-> S1S2
S1.next=L2
S2.next=L1
B.Code
L3: S1.Code
goto L2
L4: S2.code
L2: assign.code

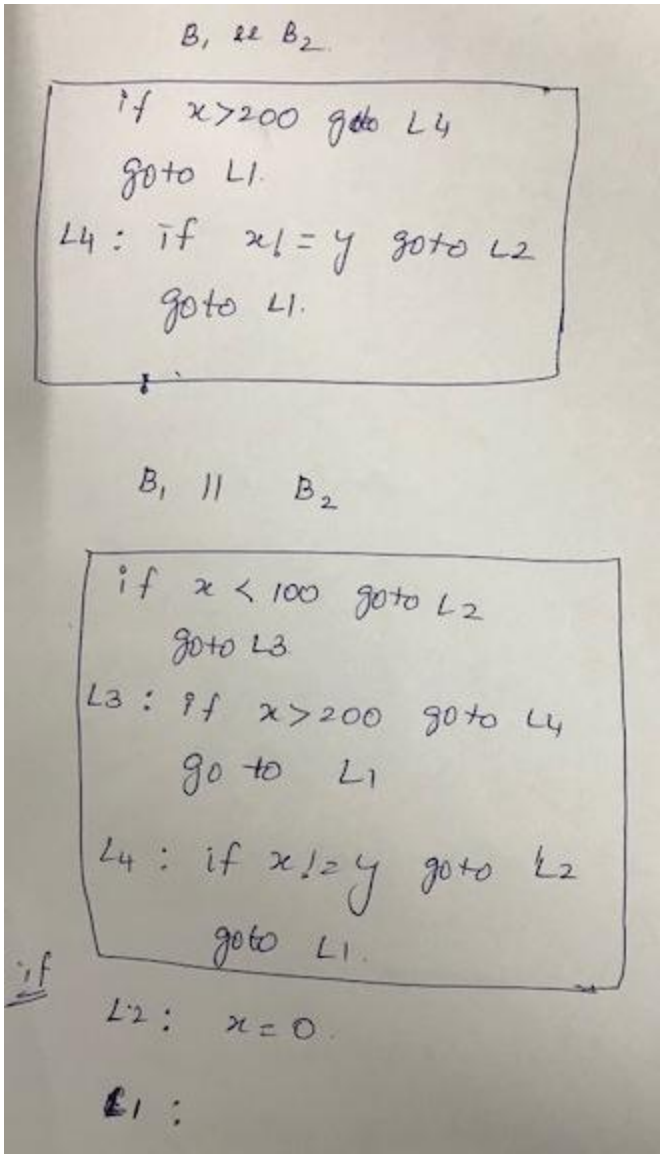
S->If (B) S1 else S2
B.T=L3
B.F=L4
S1.next=S2.next=L2
B.Code
L3: S1.Code
goto L2
L4: S2.code

S->assign
S.code=assign.code

Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \;\mathbf{rel}\; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$


if (x<100 || x>200 && x!=y)
x=0;
P
S.next = L1
S
if
B
S1
B.true = L2
B.false = L1
B1.true = L2
B1.false = L3
B1
||
B2
B2.true = L2
B2.false = L1
E1 rel E2
if x<100 goto L2
goto L3
B1.true = L4
B1.false = L1
B1
&&
B2
B2.true = L2
B2.false = L1
E1 rel E2
if x>200 goto L4
goto L1
E1 rel E2
if x!=y goto L2
goto L1


B1 && B2
if x>200 goto L4
goto L1
L4 : if x!=y goto L2
goto L1
B1 || B2
if x<100 goto L2
goto L3
L3 : if x>200 goto L4
go to L1
L4 : if x!=y goto L2
goto L1
if
L2 : x=0
L1 :

Translation of a simple if-statement

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```