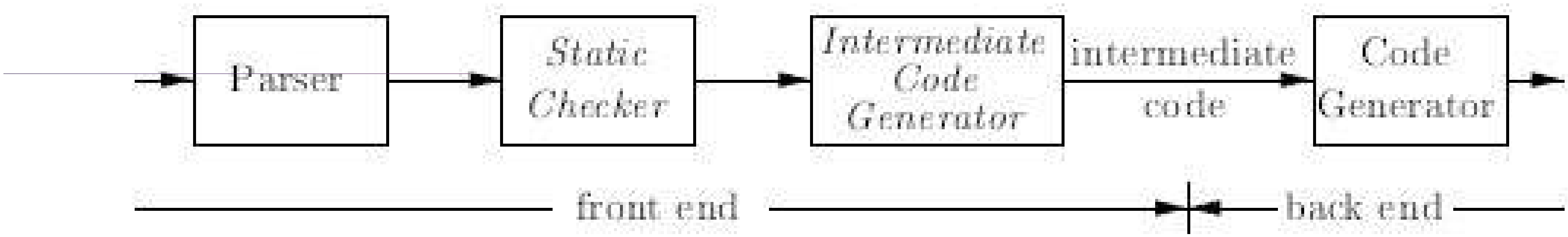
ntermediate Code Generation

UNIT 4

By
Dr. Sini Anna Alex

gical structure of a compiler front end

analysis-synthesis model of a compiler, the front end analyzes a source program and creat
nediate representation, from which the back end generates target code.



Logical structure of a compiler front end

ere parsing, static checking, and intermediate-code generation are done sequentially; som
y can be combined and folded into parsing.

ic checking includes type checking, which ensures that operators are applied to co
erands. For example, static checking assures that a break-statement in C is enclosed within
or switch-statement

ompiler might use a sequence of termediate representations

- e term **three-address code** comes from instructions of the general form
- **= y op z with three addresses**: two for the operands y and z and one for the result x.
- the process of translating a program in a given source language into code for a given targ achine, a compiler may construct a sequence of intermediate representations.
- low-level representation is suitable for machine-dependent tasks like register allocation a struction selection. High Level representation a tree like structure.

ariants of Syntax Trees - DAG

directed acyclic graph (hereafter called a DAG) for an expression identifies the co
bexpressions (subexpressions that occur more than once) of the expression. Nodes in a synt
oresent constructs in the source program; the children of a node represent the mea
omponents of a construct.

ted Acyclic Graphs for Expressions

e the syntax tree for an expression, a DAG has leaves corresponding to atomic operan
erior nodes corresponding to operators.

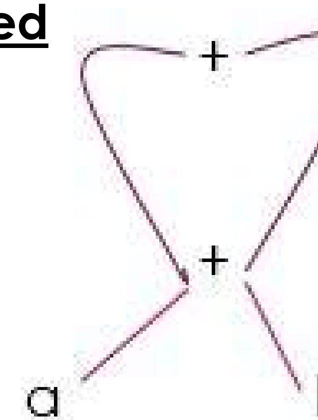
ntax-directed definition to produce syntax ees or DAG's

ODUCTION	SEMANTIC RULES
→ $E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
→ $E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
→ T	$E.node = T.node$
→ (E)	$T.node = E.node$
→ **id**	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
→ **num**	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

DAG

<u>**Input string to be represented**</u>

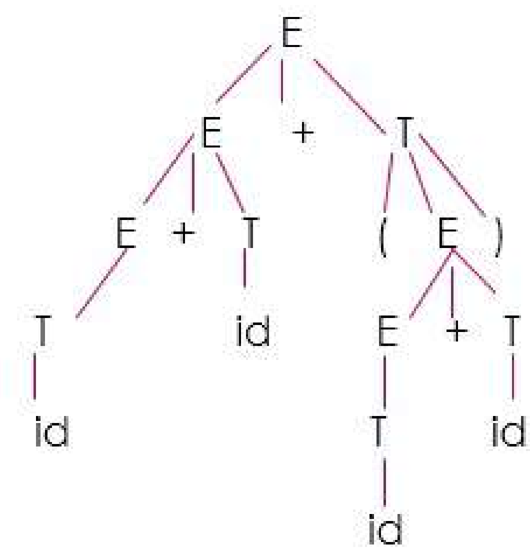
a + b + (a + b)



<u>**Syntax Tree**</u>

<u>**t string to be represented**</u>

b + (a + b)

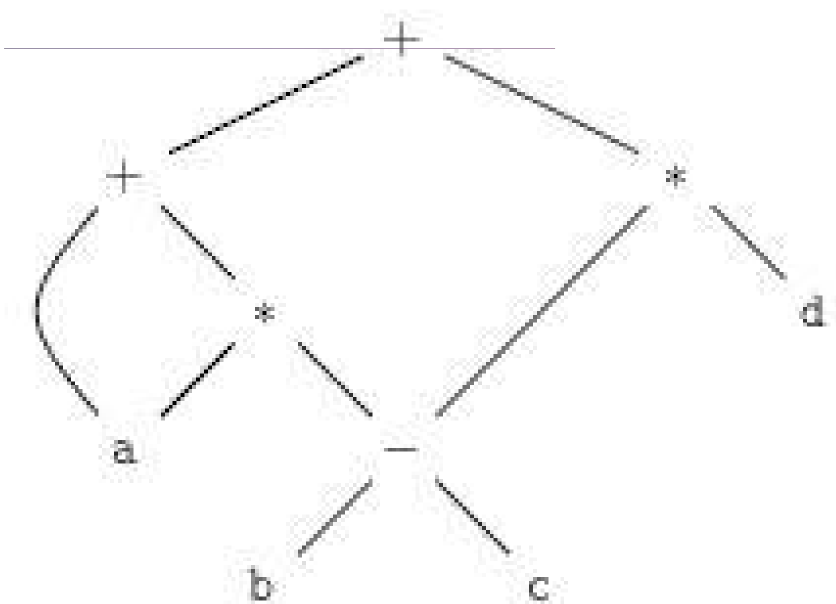


<u>**Steps for constructing the D**</u>

1) p1= Leaf(id, entry-a)
2) p2 = Leaf(id, entry-b)
3) p3= Node('+',p1,p2)
4) p4=Leaf(id, entry-a) = p
5) p5=Leaf(id, entry-b) = p
6) p6=Node('+',p1,p2) = p
7) p7= Node('+',p3,p3)

onstruct DAG for the expression
+a*(b-c)+ (b-c)*d

DAG



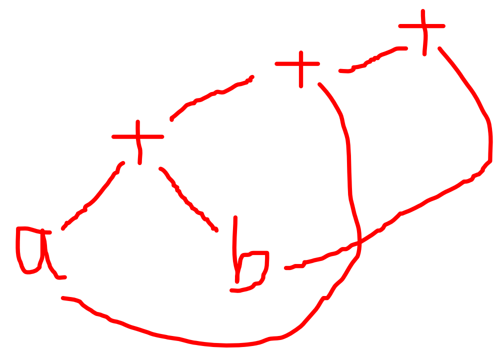
Steps for constructing the DA

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

onstruct the DAG for the expression assuming + associates from the left)

ab+ a+ b+

$$a + b + a + b.$$



$$a + a + (a + a + a + (a + a + a + a)).$$

aa+ aa+ a+ aa+ a+ a+ + +

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

utorial Questions

- 1. A-> L M { L.i=f(A.s); M.i=f(L.s); A.s=f(M.s); }

- 2. A->Q R { R.i=f(A.i);Q.i=f(R.i); A.s=f(Q.s); }

- **Is the above definitions S-Attributed or L-attributed?**

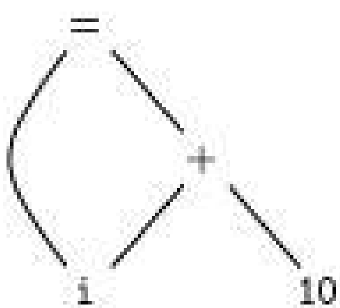
Intermediate Code Generation

Unit 4

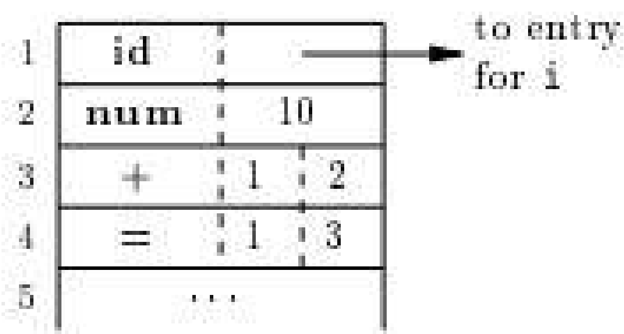
Dr. Sini Anna Alex

Value Number Method for Constructing DAG’s

- Nodes of Syntax Tree or DAG – stored in an array of records.
- Each row represents- one record(one node)
- Nodes of a DAG for i=i+10



(a) DAG



(b) Array.

Algorithm: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

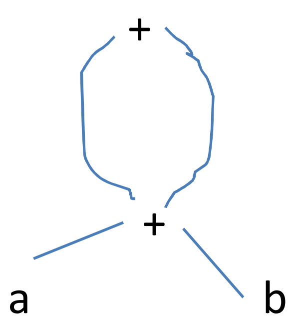
OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: 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 of 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 number.

Construct three address code and VNM for the given DAG

a+b+(a+b)

DAG



Three Address Code

t1= a+ b
t2=t1+t1

Value Number Method (VNM)

1	**id**		**Entry for a**
2	id		Entry for b
3	+	1	2
4	+	3	3

Three Address Code

- In three-address code, there is at most one operator on the right side of an instruction.
- source-language expression like x+y*z might be translated into the sequence of three-address instructions

$$t1 = y * z$$

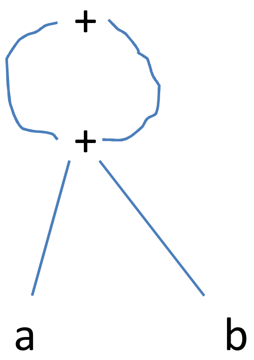
$$t2 = x + t1$$

where t1 and t2 are compiler-generated temporary names.

Three Address Code From DAG and From Syntax Tree

- a+b+(a+b)

DAG

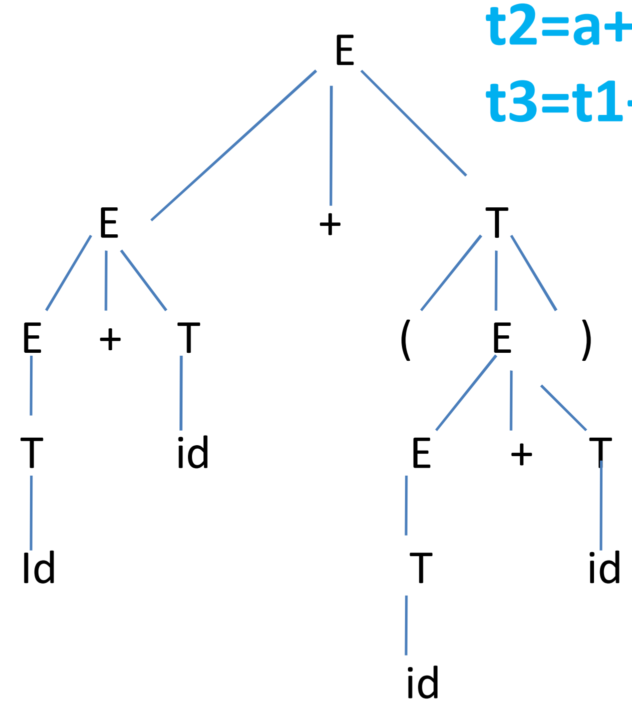


Three Address Code

t1= a+ b

t2=t1+t1

Syntax Tree



Three Address Code

t1=a+b

t2=a+b

t3=t1+t2

Addresses and Instructions

- Three-address code is built from two concepts: addresses and instructions.

An address can be one of the following:

- Name
- Constant
- A compiler-generated temporary

Common three-address instruction forms

- 1. Assignment instructions of the form x = y op z, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.
- 2. Assignments of the form x = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators, for example, convert an integer to a floating-point number.
- 3. Copy instructions of the form x = y, where x is assigned the value of y.

Common three-address instruction forms contd..

- 4. An unconditional jump **goto L**. The three-address instruction with label L is the next to be executed.
- 5. Conditional jumps of the form if x goto L and ifFalse x goto L.
- 6. Conditional jumps such as if x relop y goto L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y.

Common three-address instruction forms contd..

7. Procedure calls and returns are implemented using the following instructions: param x for parameters;

call p,n and y = call p,n for procedure and function calls, respectively;

return y, where y, representing a returned value, is optional.

p(x1,x2;..., xn)

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

Where n= no of arguments

8. Indexed copy instructions of the form x=y[i] and x[i]=y.
9. Address and pointer assignments of the form x =&y, x =*y, and *x = y. The instruction x =&y sets the r-value of x to be the location (l-value) of y.

Three Address Code Translation

1. a=b[i]+c[j]

2. a[i]=b*c + b*d

3. x=f(y+1)+2

Three address Translation of Control Statements

- Consider the statement

  do i = i+1; while (a[i] < v);

- Two possible translations of this statement are

```
L:    t1 = i + 1
      i = t1
      t2 = i * 8
      t3 = a [ t2 ]
      if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

The description of three-address instructions

- Three address instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called quadruples, triples, and indirect triples.
- Quadruples- A quadruple (or just quad) has four fields: op, arg1, arg2, and result.
- The op field contains an internal code for the operator.
- For instance, the three-address instruction $x = y + z$ is represented by placing + in op, y in arg1, z in arg2, and x in result.

Quadruple Representation

The following are some exceptions to this rule:

- 1. Instructions with unary operators like x = minus y or x = y do not use arg2. For a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.
- 2. Operators like param use neither arg2 nor result.
- 3. Conditional and unconditional jumps put the target label in result.

Quadruple representation of $a = b*-c + b*-c$

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

Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

Quadruples

Triples

A triple has only three fields, which we call op, arg1, and arg2. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

Triple Representation of a = b*-c + b*-c

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

Three-address code

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Triples

Indirect Triples

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without a directing the triples themselves

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

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Indirect triples representation of three-address code

THANK YOU

Intermediate Code Generation

Unit 4

Sini Anna Alex

The description of three-address instructions

- Three address instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called quadruples, triples, and indirect triples.
- Quadruples- A quadruple (or just quad) has four fields: op, arg1, arg2, and result.
- The op field contains an internal code for the operator.
- For instance, the three-address instruction $x = y + z$ is represented by placing + in op, y in arg1, z in arg2, and x in result.

Quadruple Representation

The following are some exceptions to this rule:

- 1. Instructions with unary operators like

  x = minus y or x = y do not use arg2.

For a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

- 2. Operators like param use neither arg2 nor result.
- 3. Conditional and unconditional jumps put the target label in result. goto L

Quadruple representation of a = b*-c + b*-c

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

Three-address code

	op	*arg_1*	*arg_2*	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

Quadruples

Triples

A triple has only three fields, which we call op, arg1, and arg2. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.

Triple Representation of a = b*-c + b*-c

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

Three-address code

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Triples

Indirect Triples

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without a directing the triples themselves

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

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Indirect triples representation of three-address code

Static Single Assignment Form(SSA)

- Intermediate Representation that facilitates certain code optimizations.
- Intermediate program in three-address code and SSA

$$
\begin{array}{l}
p = a + b \\
q = p - c \\
p = q * d \\
p = e - p \\
q = p + q
\end{array}
$$

(a) Three-address code.

$$
\begin{array}{l}
p_1 = a + b \\
q_1 = p_1 - c \\
p_2 = q_1 * d \\
p_3 = e - p_2 \\
q_2 = p_3 + q_1
\end{array}
$$

(b) Static single-assignment form.

ϕ-function

- The same variable may be defined in two different control-flow paths in a program.

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

- has two control-flow paths in which the variable x gets defined.

$$\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1\texttt{;}$$
$$x_3 = \phi(x_1, x_2)\texttt{;}$$

- Here, ϕ(x1,x2) has the value x1 if the control flow passes through the true part of the conditional and the value x2 if the control flow passes through the false part.

Types and Declarations

- The applications of types can be grouped under checking and translation:
- **Type checking:** uses logical rules to reason about the behavior of a program at run time.
  - For example, the && operator in Java expects its two operands to be booleans; the result is also of type boolean.

- **Translation Applications:** From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

  -For example, Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions.

Type Expressions

- Types have structure, which represent using type expressions:
  - a type expression is either a basic type or is by applying an operator called a type constructor to a type expression.
    - The array type int[2][3] can be read as “array of 2 arrays of 3 integers each" and written as a type expression array(2,array(3,integer)). The operator array takes two parameters, a number and a type.

Definition of type expressions

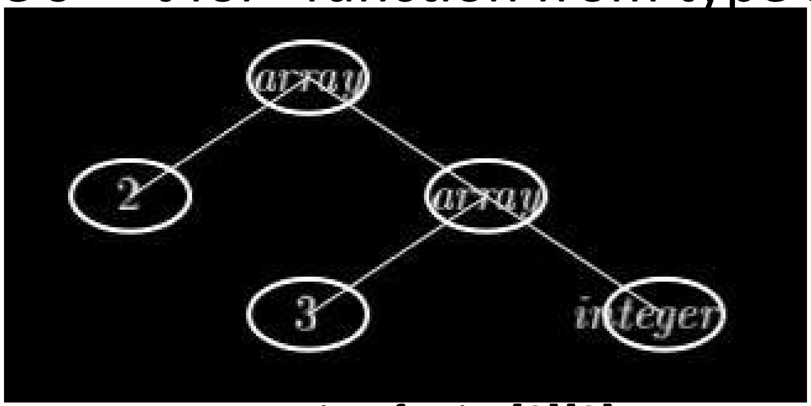
•A basic type is a type expression

•A type name is a type expression.

•A type expression can be formed by applying the array type constructor to a number and a type expression.

• A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types.

•A type expression can be formed by using the type constructor -> for function types. We write s -> t for “function from type s to type t."



Type expression for int[2][3]

Type Equivalence

- When are two type expressions equivalent?
  - Many type-checking rules have the form, “if two type expressions are equal then return a certain type else error.“
- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types.
  - One is a type name that denotes the other.
- The first two conditions in the above definition lead to name equivalence of type expressions. Name-equivalent expressions are assigned the same value number.

CFG for valid Declarations in C Language

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

* record -> can be a struct or union

Storage Layout for local names

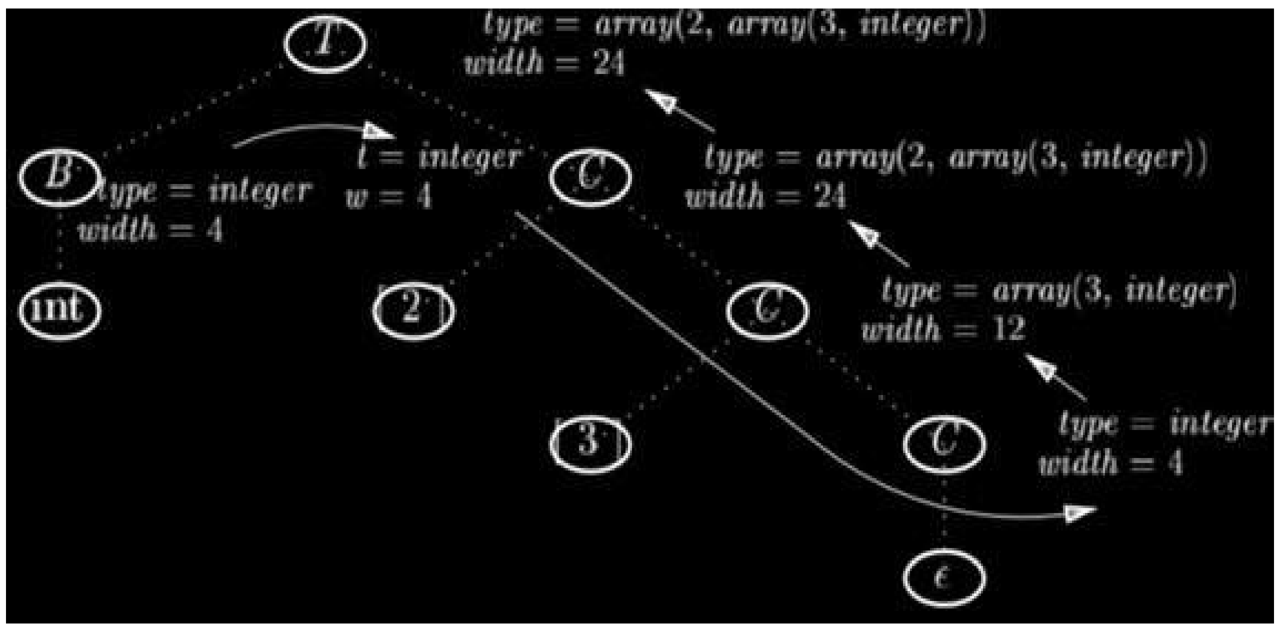
- From the type of a name, we can determine the amount of storage that will be needed for the name at run time.
  - Eg: integer means 4 bytes of storage, int A[10].
- At compile time, we can use these amounts to assign each name a relative address.
  - A[2] array reference can be accessed by base address(A)+2*4.
- The type and relative address are saved in the symbol-table entry for the name.
  - Symbol Table gets added with the additional specifications.
  - A is an identifier with type array(10,int)

Computing types and their widths
Syntax-directed translation of array types

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

The width of a type is the number of storage units needed for objects of that type.
A basic type, such as a character, integer, or float, requires an integral number of bytes.

SDT uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information.
In syntax-directed translation, t and w would be inherited attributes for C.



Sequences of Declarations

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group.
- Therefore, we can use a variable, say offset, to keep track of the next available relative address.
- For eg: We have a declaration in C as given below

```
int a;
struct{int b; float x;}p;
```

Total Storage allocation should be 4 + (4+8) =16 bytes.

(For int – 4 bytes , float – 8 bytes)

Computing the relative addresses of declared names

$$
\begin{array}{lll}
P & \rightarrow & \{\ offset = 0;\ \} \\
& & D \\
D & \rightarrow & T\ \mathbf{id}\ ;\quad \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
& & \quad offset\ =\ offset + T.width;\ \} \\
& & D_1 \\
D & \rightarrow & \epsilon
\end{array}
$$

D -> T id;D1 creates a symbol table entry by executing
top . put (id . lexeme, T . type, offset).
Here top denotes the current symbol table.

$$P \rightarrow \{\ offset = 0;\ \}\ D$$

Handling of field names in records

A record type has the form record(t), where record is the type constructor, and t is a symbol table object that holds information about the fields of this record type

$$
\begin{array}{lll}
T \rightarrow & \mathbf{record}\ '\{' & \{\ Env.push(top);\ top = \mathbf{new}\ Env(); \\
& & \ Stack.push(offset);\ offset = 0;\ \} \\
& D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
& & \ top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

Class Env implement symbol tables. The call Env.push(top) pushes the current symbol table denoted by top onto a stack. Variable top is then set to a new symbol table. Similarly, offset is pushed onto a stack called Stack.

Compute the type and relative address for the Declaration statement given below

float x;

record { float a; float b;} p;

Use the given Grammar for computation:

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

THANK YOU

Intermediate Code Generation

Unit 4

Sini Anna Alex

Subject: **Compiler Design**

Q 12. Consider the following intermediate program in three address code

$$
\begin{aligned}
p &= a - b\\
q &= p * c\\
p &= u * v\\
q &= p + q
\end{aligned}
$$

Which one of the following corresponds to a static single assignment form of the above code ?

(a)
$$
\begin{aligned}
p_1 &= a - b\\
q_1 &= p_1 * c\\
p_1 &= u * v\\
q_1 &= p_1 + q_1
\end{aligned}
$$

(b)
$$
\begin{aligned}
p_3 &= a - b\\
q_4 &= p_3 * c\\
p_4 &= u * v\\
q_5 &= p_4 + q_4
\end{aligned}
$$

(c)
$$
\begin{aligned}
p_1 &= a - b\\
q_1 &= p_2 * c\\
p_3 &= u * v\\
q_2 &= p_4 + q_3
\end{aligned}
$$

(d)
$$
\begin{aligned}
p_1 &= a - b\\
q_1 &= p * c\\
p_2 &= u * v\\
q_2 &= p + q
\end{aligned}
$$

What is the triple representation of x[i]=y (Assume x is an integer array)

Three address Code:

t1=i*4

x[t1]=y

Triple

	op	arg1	arg2
0	*	i	4
1	[]=	t1	y

Computing type and their widths

int [5][5]

$\Rightarrow$

type expression

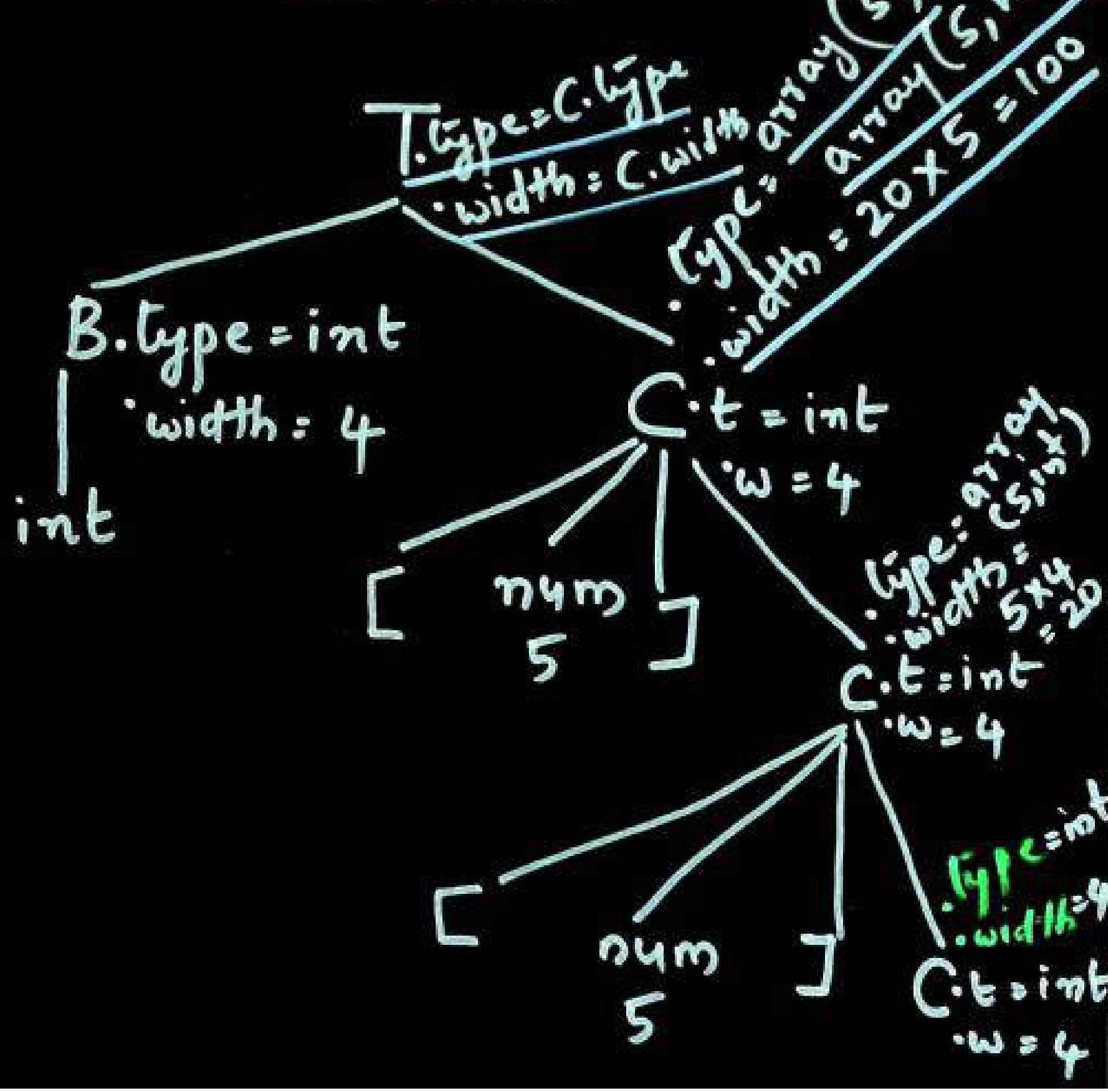
array(5, array(5, int))

Total storage units

$= 5 \times 5 \times 4$ (width of integer)

$= 100$

int [5] [5];



(Compute type and relative address

```
int x;
record { int a;
         float b;
       } p;
```

$P \rightarrow$ { offset = 0; } D

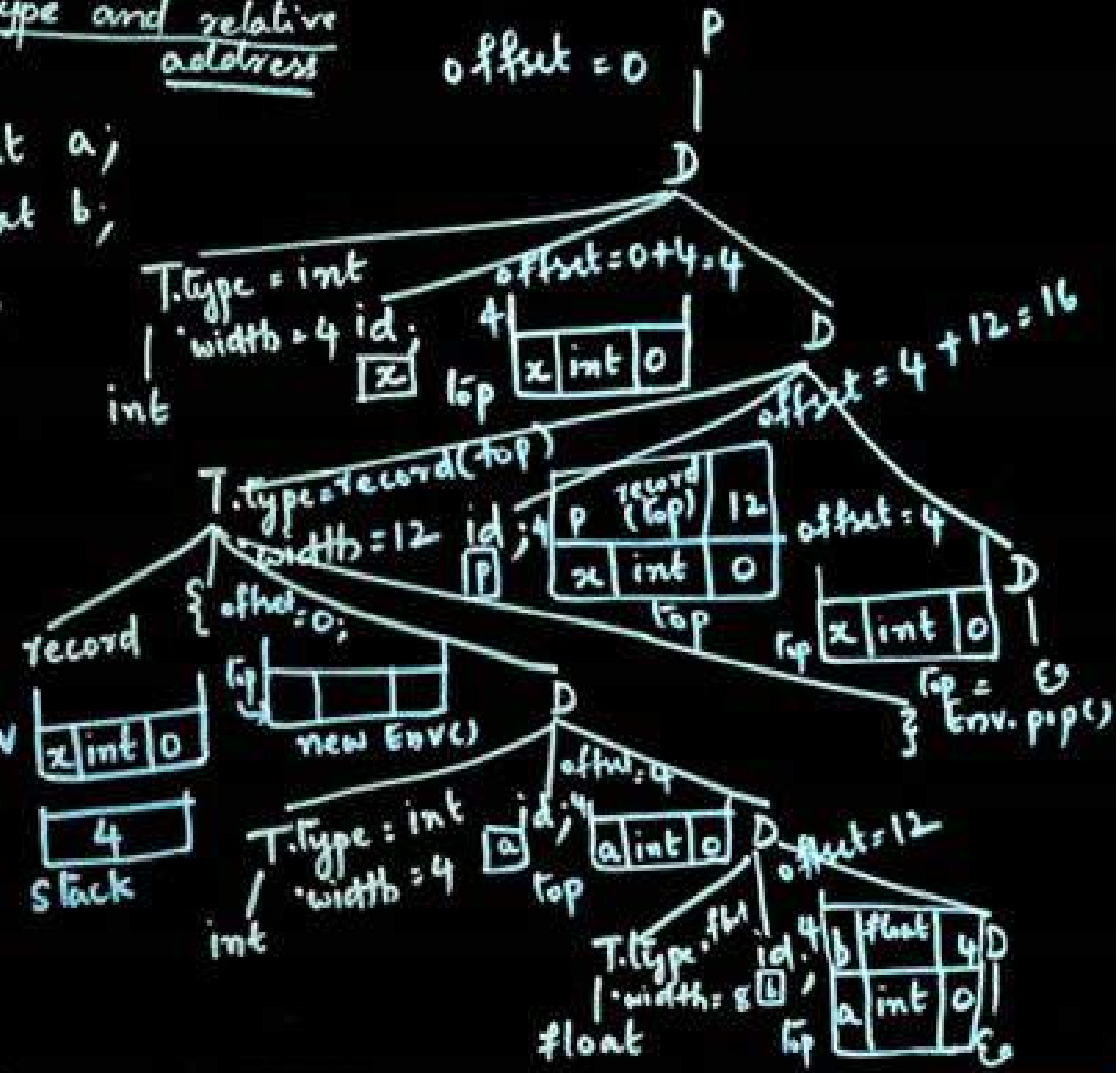
$D \rightarrow T$ id; { top.put(id.lexeme, T.type, offset); offset = offset + T.width; }

D_1

$D \rightarrow \varepsilon$

$T \rightarrow$ record '{' { Env.push(top); top = new Env(); Stack.push(offset); offset = 0; }

D '}' { T.type = record(top); T.width = offset; top = Env.pop(); offset = Stack.pop(); }



$a = b[i] + c[j]$

3AC

$t_1 = i * 4$

$t_2 = b[t_1]$

$t_3 = j * 4$

$t_4 = c[t_3]$

$t_5 = t_2 + t_4$

$a = t_5$

Quadruple Representation

	op	arg1	arg2	result
0	*	i	4	t_1
1	=[ ]	b	t_1	t_2
2	*	j	4	t_3
3	=[ ]	c	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

2. $a[i] = b * c + b * d$

<u>3AC</u>

$t_1 = i * 4$

$t_2 = b * c$

$t_3 = b * d$

$t_4 = t_2 + t_3$

$a[t_1] = t_4$

---

*

$a[t_1] = t_4$

$\boxed{\text{op :- } [\ ] =}$

$x = f(y+1) + 2$

<u>3AC</u>

$t_1 = y + 1$

param t_1

$t_2 = \text{call } f, 1$

$\boxed{\text{// return } t_2 \text{ (optional)}}$

$t_3 = t_2 + 2$

$x = t_3$

Quadruple Representation

	op	arg1	arg2	result
0	+	y	1	t_1
1	param	t_1		
2	Call	f	1	t_2
3	+	t_2	2	t_3
4	=	t_3		x

Intermediate Code Generation

Unit 4

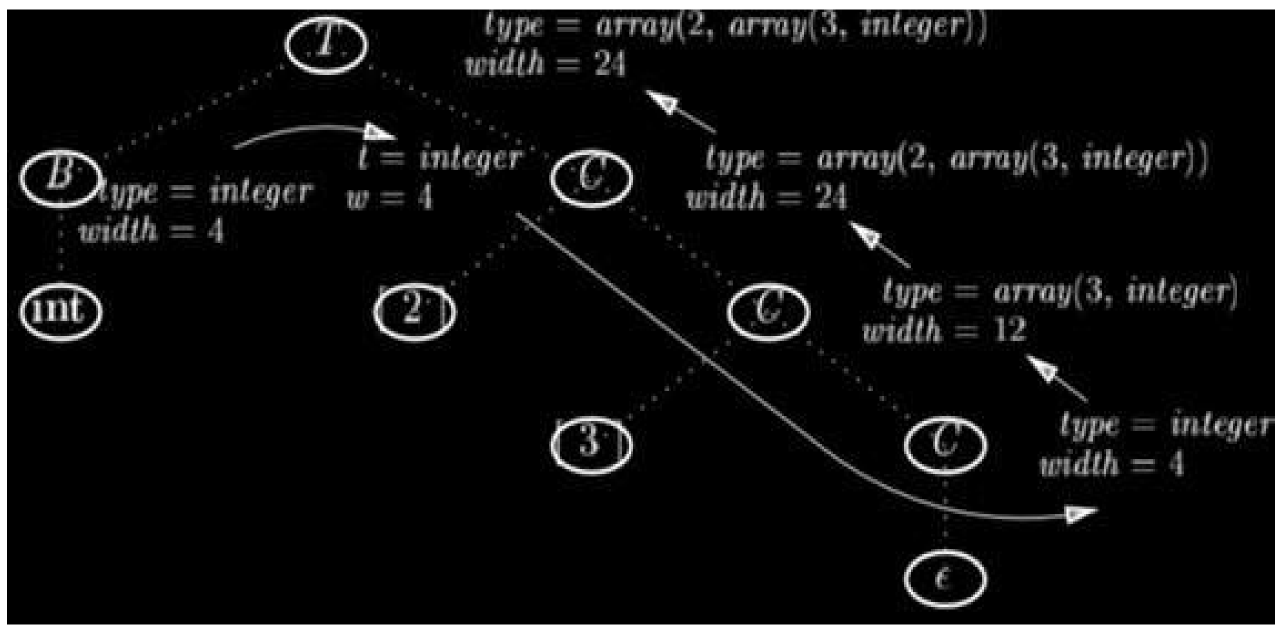
Sini Anna Alex

Computing types and their widths
Syntax-directed translation of array types

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

The width of a type is the number of storage units needed for objects of that type.
A basic type, such as a character, integer, or float, requires an integral number of bytes.

SDT uses synthesized attributes type and width for each nonterminal and two variables t and w to pass type and width information.
In syntax-directed translation, t and w would be inherited attributes for C.



Computing type and their widths

int [5][5]

$\Rightarrow$

type expression

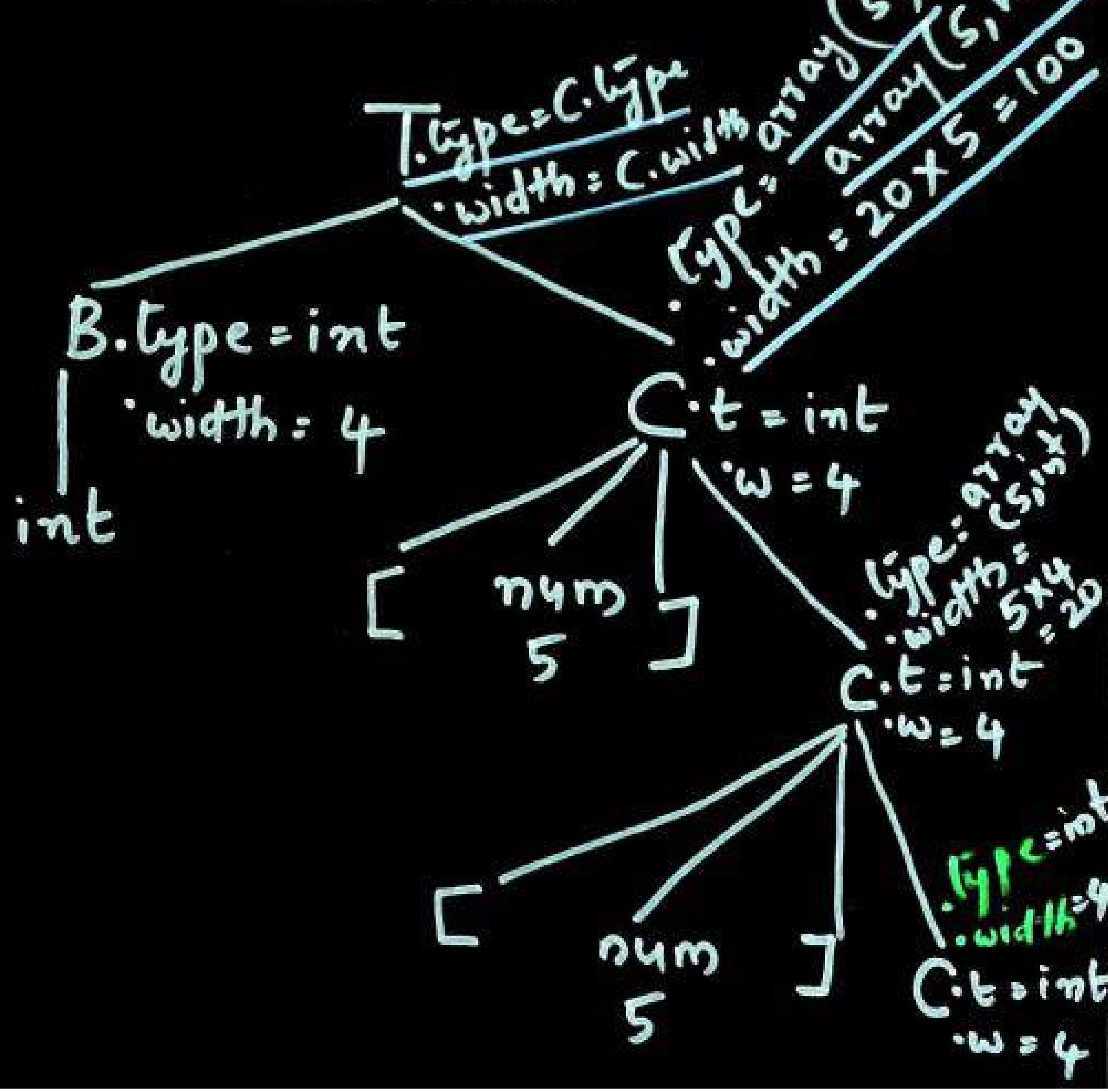
array(5, array(5, int))

Total storage units

$= 5 \times 5 \times 4$ (width of integer)

$= 100$

int [5] [5];



Sequences of Declarations

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group.
- Therefore, we can use a variable, say offset, to keep track of the next available relative address.
- For eg: We have a declaration in C as given below

```
int a;
struct{int b; float x;}p;
```

Total Storage allocation should be 4 + (4+8) =16 bytes.

(For int – 4 bytes , float – 8 bytes)

Computing the relative addresses of declared names

$$
\begin{array}{llll}
P & \rightarrow & & \{\ offset = 0;\ \} \\
 & & D & \\
D & \rightarrow & T\ \textbf{id}\ ; & \{\ top.put(\textbf{id}.lexeme,\ T.type,\ offset); \\
 & & & \quad offset\ =\ offset + T.width;\ \} \\
 & & D_1 & \\
D & \rightarrow & \epsilon &
\end{array}
$$

D -> T id;D1 creates a symbol table entry by executing
top . put (id . lexeme, T . type, offset).
Here top denotes the current symbol table.

$$P \rightarrow \{\ offset = 0;\ \}\ D$$

Handling of field names in records

A record type has the form record(t), where record is the type constructor, and t is a symbol table object that holds information about the fields of this record type

$$
\begin{array}{lll}
T \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); \\
 & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

Class Env implement symbol tables. The call Env.push(top) pushes the current symbol table denoted by top onto a stack. Variable top is then set to a new symbol table. Similarly, offset is pushed onto a stack called Stack.

Compute the type and relative address for the Declaration statement given below

int x;

record {   int a; float b;   } p;

Use the given Grammar for computation:

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

(Compute type and relative address

```
int x;
record { int a;
         float b;
       } p;
```

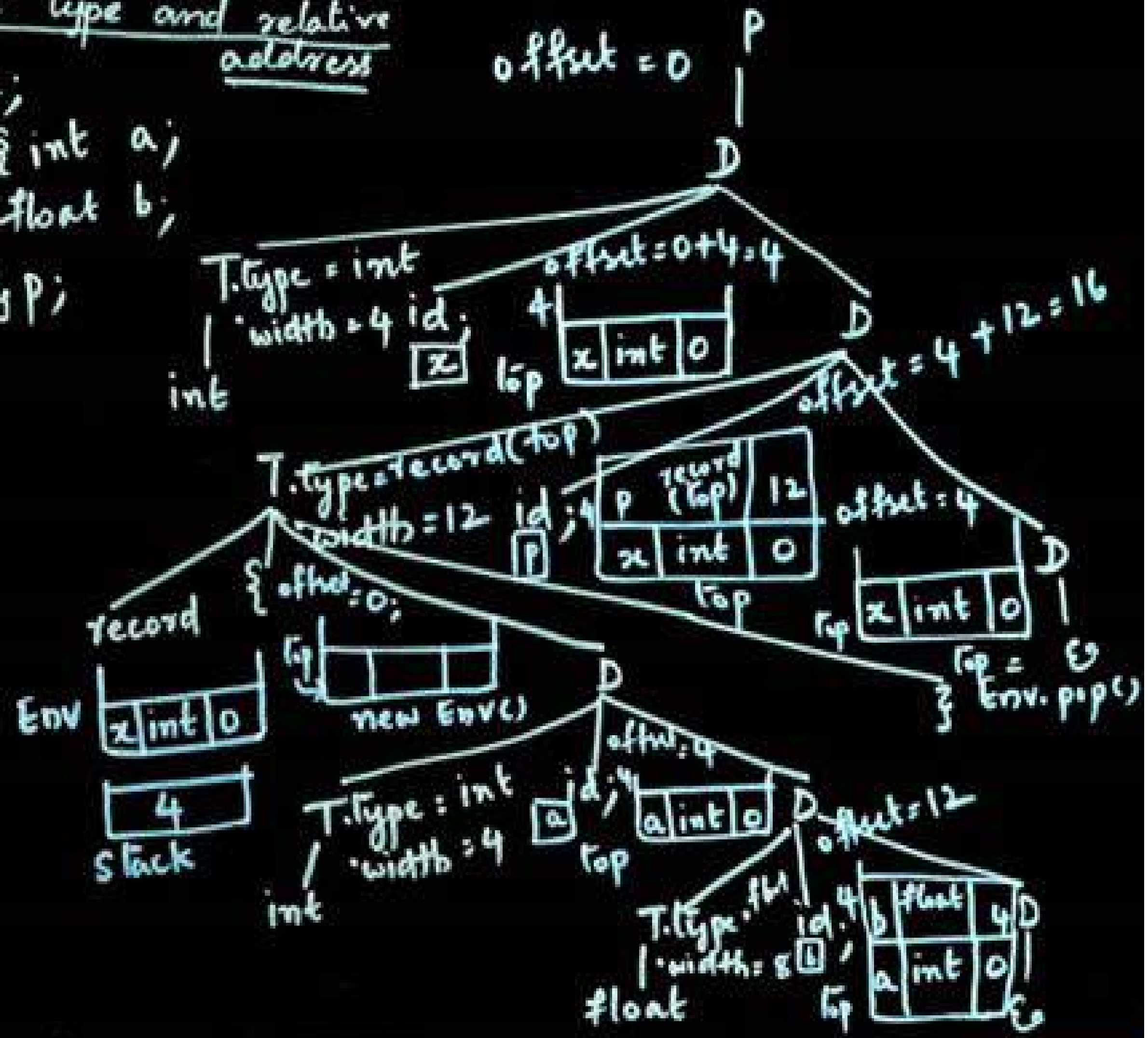
P → {offset = 0;} D

D → T id; {top.put(
  id.lexeme, T.type, offset);
 offset = offset + T.width;}

D₁

D → ε

T → record '{' {Env.push(top);
 top = new Env();
 Stack.push(offset);
  offset = 0; }
  D '}'   {T.type = record(top);
         T.width = offset;
         top = Env.pop();
   offset = Stack.pop(); }

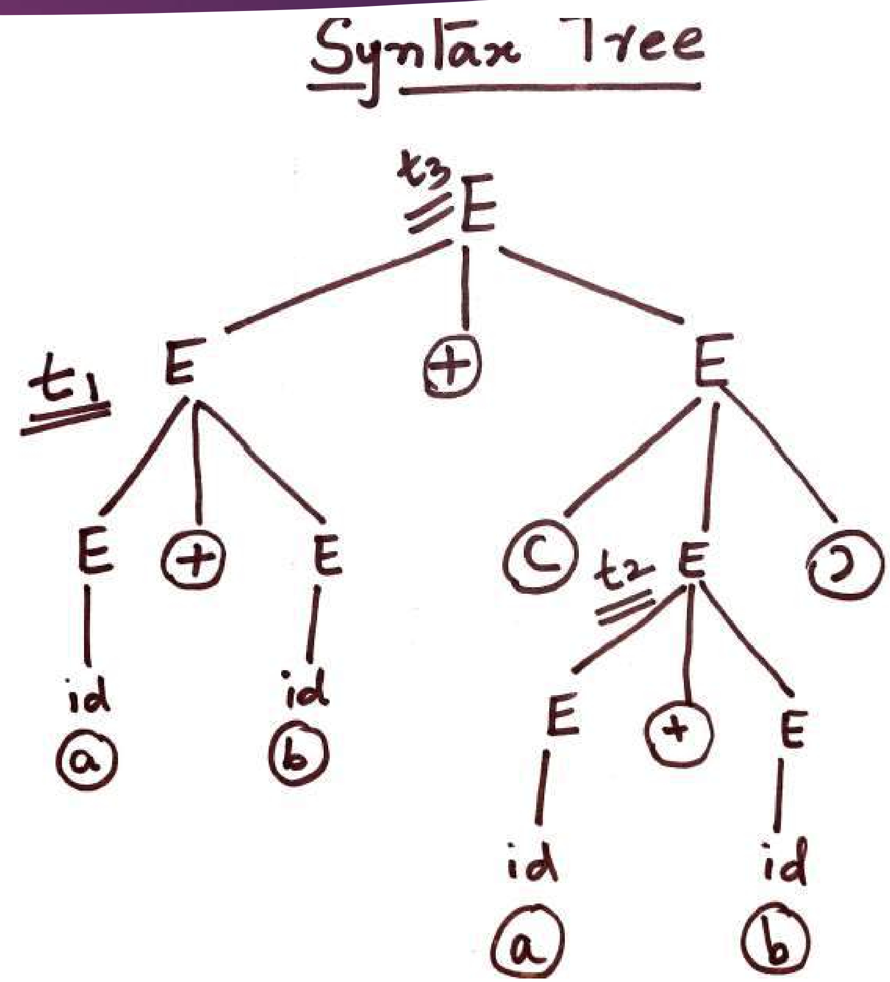


THANK YOU

Translation of Expressions

- **Translation of expressions into three-address code**
  - An expression with more than one operator, like a + b * c, will translate into instructions with at most one operator per instruction.
- The compiler decides the order of operation given by three address code.
- **Three address code** is a linearized representation of a syntax tree, where the names of the temporaries correspond to the nodes.

- **<u>Three Address Code for a + b + (a + b)</u>**
- t1=a + b
- t2=a + b
- t3=t1 + t2



Operations within Expressions

Attributes **S.code and E.code** denote the three-address code for S and E, respectively.

Attribute **E.addr** denotes the address that will hold the value of E. An address can be a nam a constant, or a compiler-generated temporary.

The notation **gen(x = y + z)** to represent the three-address instruction **x = y + z**. Expressio appearing in place of variables like x, y, and z are evaluated when passed to gen, ar quoted strings like "=" are taken literally.

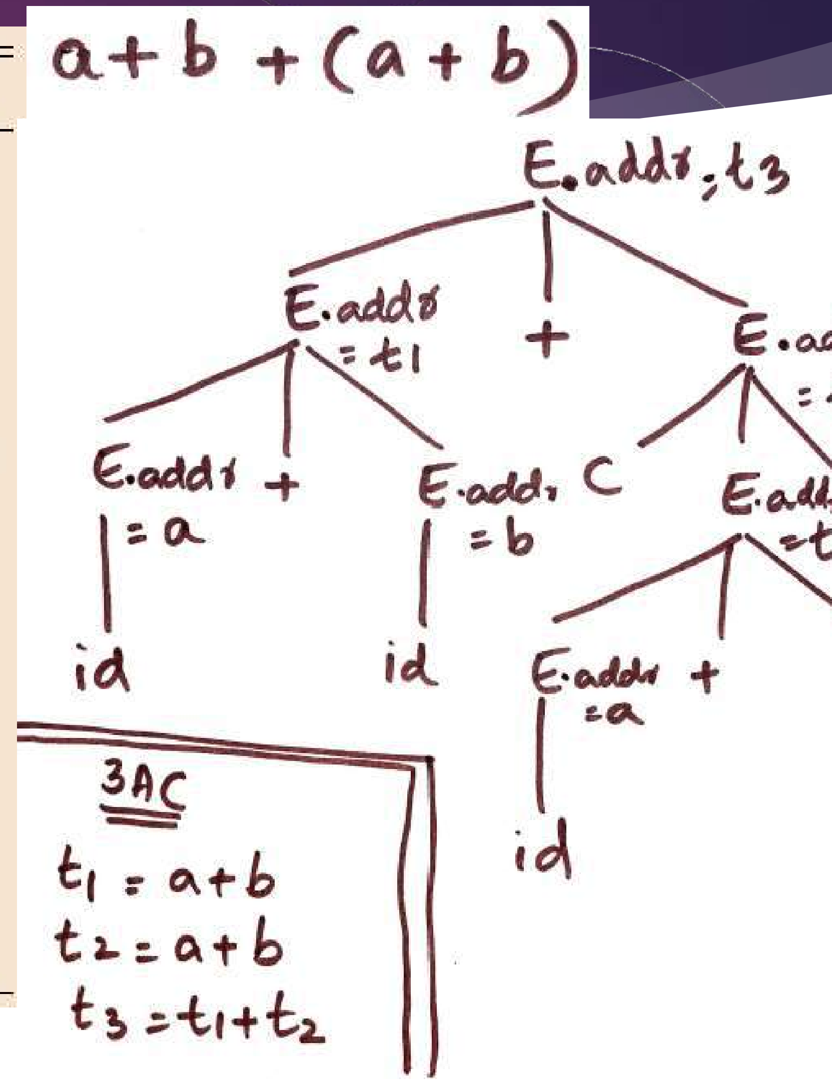
- In syntax-directed definitions, gen builds an instruction and returns it. In translation schemes, g builds an instruction and incrementally emits it by putting it into the stream of generated instruction

A sequence of distinct temporary names t1,t2, … is created by successively executing ne Temp().

E.addr to point to the symbol-table entry for the instance of id. Let top denote the curre symbol table. Function **top.get** retrieves the entry when it is applied to the stri representation **id.lexeme** of the instance of id. E.code is set to the empty string.

hree-address code for expressions

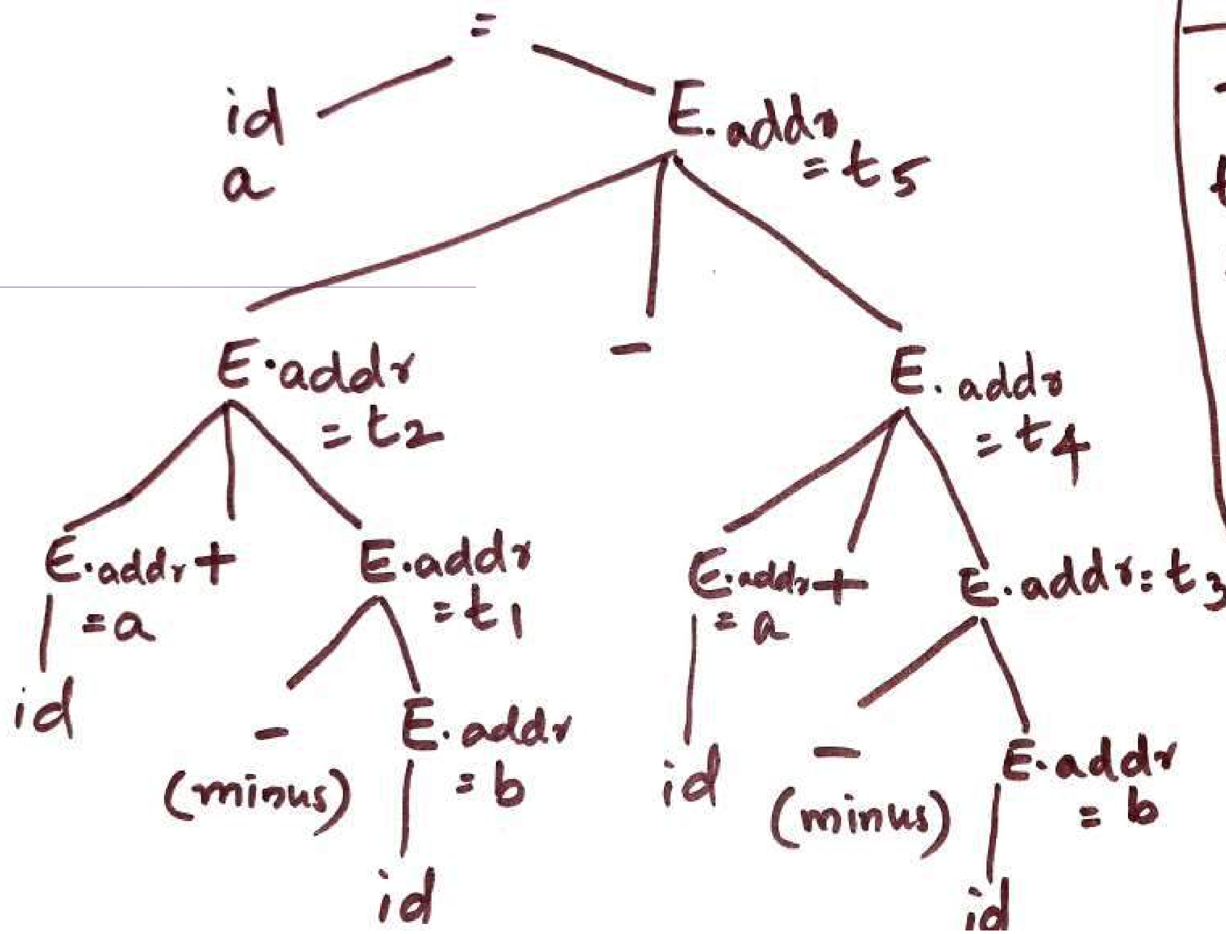
ODUCTION	SEMANTIC RULES
$\rightarrow$ **id** = E ;	$S.code = E.code \;\|\|$ $gen(top.get(\textbf{id}.lexeme)\;'='\;E.addr)$
$\rightarrow$ E_1 + E_2	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $gen(E.addr\;'='\;E_1.addr\;'+'\;E_2.addr)$
| - E_1	$E.addr = \textbf{new}\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



Generate Three Address Code for the following using Translation Scheme

- $a = a + -b - a + -b$

a = a + -b - a + -b



3AC
t_1 = minus b
$t_2 = a + t_1$
t_3 = minus b
$t_4 = a + t_3$
$t_5 = t_2 - t_4$
$a = t_5$

Incremental Translation

- Code attributes can be long strings, so they are usually generated incrementally.
- Thus, instead of building up E.code generate only the new three-address instructions, as in the translation scheme.
- In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally.
- Here, attribute **addr** represents the address of a node rather than a variable or constant.

Generating three-address code for expressions incrementally

$$
\begin{array}{lcll}
S & \rightarrow & \mathbf{id} = E \; ; & \{\; gen(\; top.get(\mathbf{id}.lexeme) \; '=' \; E.addr); \;\} \\[2ex]
E & \rightarrow & E_1 + E_2 & \{\; E.addr = \mathbf{new} \;\; Temp\,(); \\
& & & \;\;\; gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \;\} \\[2ex]
& | & - \, E_1 & \{\; E.addr = \mathbf{new} \;\; Temp\,(); \\
& & & \;\;\; gen(E.addr \; '=' \; '\mathbf{minus}' \; E_1.addr); \;\} \\[2ex]
& | & (\; E_1 \;) & \{\; E.addr = E_1.addr; \;\} \\[2ex]
& | & \mathbf{id} & \{\; E.addr = top.get(\mathbf{id}.lexeme); \;\}
\end{array}
$$

Answer the question

- **The least number of temporary variables required to create a three-address code in static single assignment form for the expression**

  $q + r/3 + s - t * 5 + u * v/w$ is ________________ ?

Answer: 8

Addressing Array Elements

- Array elements can be accessed quickly if they are stored in a block of consecutiv locations. In C and Java array elements are numbered 0,1,...n -1, for an array with elements.
- If the width of each array element is w, then the i^{th} element of array A begins location. For a one dimensional array, address calculation will be

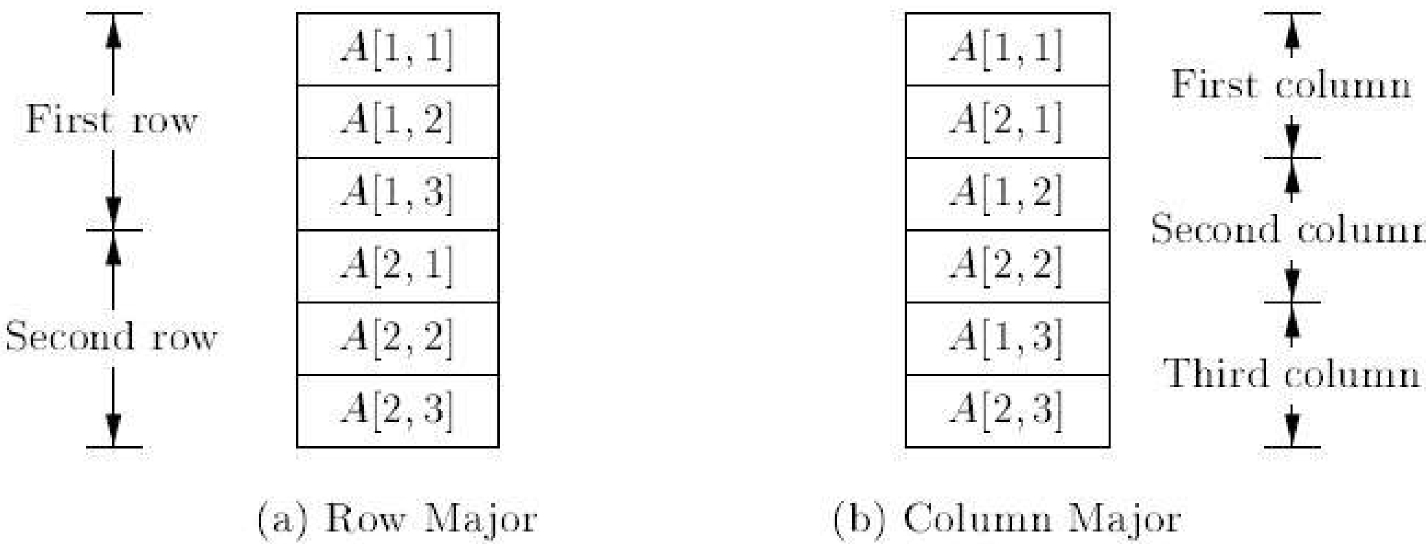
base+ i x w

  - where base is the relative address of the storage allocated for the array. That is, base is th relative address of A[0].
- For a two dimensional array, the relative address of A[i1][i2] can then be calculated b the formula $base + i_1 \times w_1 + i_2 \times w_2$
- In two dimensions, the location for A[i1][i2] is given by

base+(i_1x n_2 +i_2)x w

Layouts for a two dimensional array

The address calculations used here are based on row-major layout for arrays, which is used
Column major form is used in the Fortran family of languages.



Translation of Array References

nonterminal L generate an array name followed by a sequence of index expressions:

$$L \rightarrow L\ [\ E\ ]\ \mid\ \mathbf{id}\ [\ E\ ]$$

In C and Java , the lowest numbered array index element is 0.

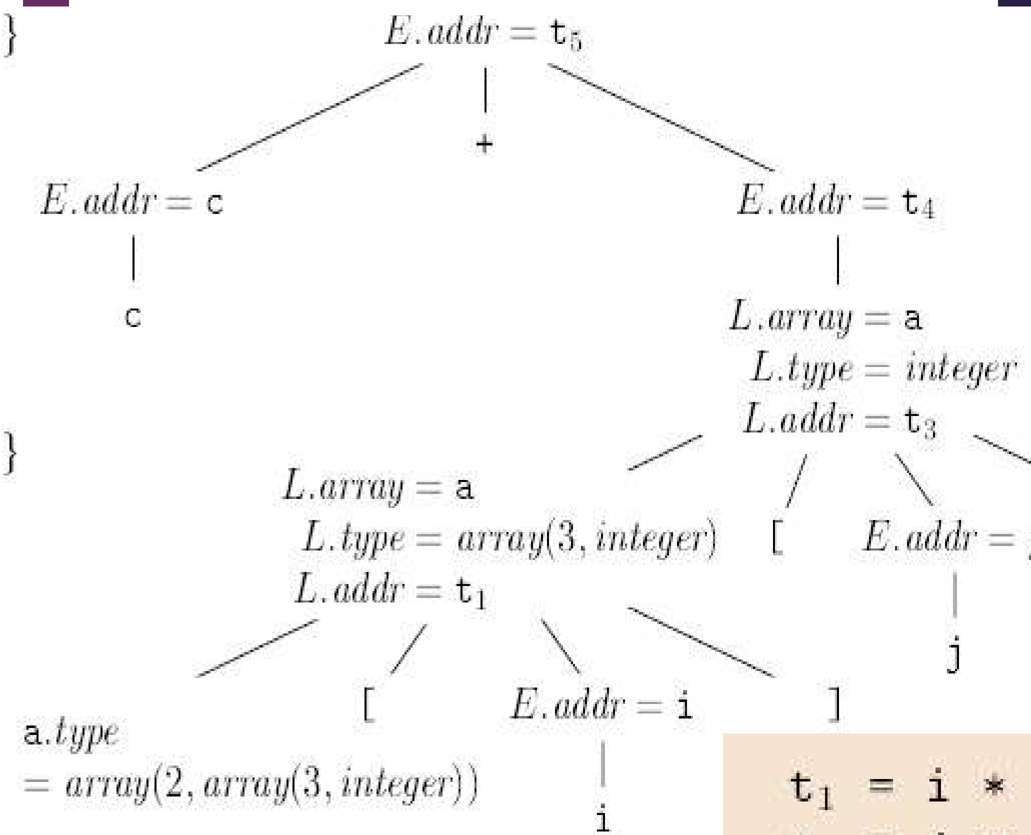
Nonterminal L has three synthesized attributes:

1. $L.addr$ denotes a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$
2. $L.array$ is a pointer to the symbol-table entry for the array name. The base address of the array, say, $L.array.base$ is used to determine the actual l-value of an array reference after all the index expressions are analyzed.
3. $L.type$ is the type of the subarray generated by L. For any type t, we assume that its width is given by $t.width$. We use types as attributes, rather than widths, since types are needed anyway for type checking. For any array type t, suppose that $t.elem$ gives the element type.

Semantic Actions for Array References

id = E ;	{ $gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }
L = E ;	{ $gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);$ }
E_1 + E_2	{ $E.addr = \mathbf{new}\ Temp\,();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
id	{ $E.addr = top.get(\mathbf{id}.lexeme);$ }
L	{ $E.addr = \mathbf{new}\ Temp\,();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }
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]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \mathbf{new}\ Temp\,();$ $L.addr = \mathbf{new}\ Temp\,();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

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



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

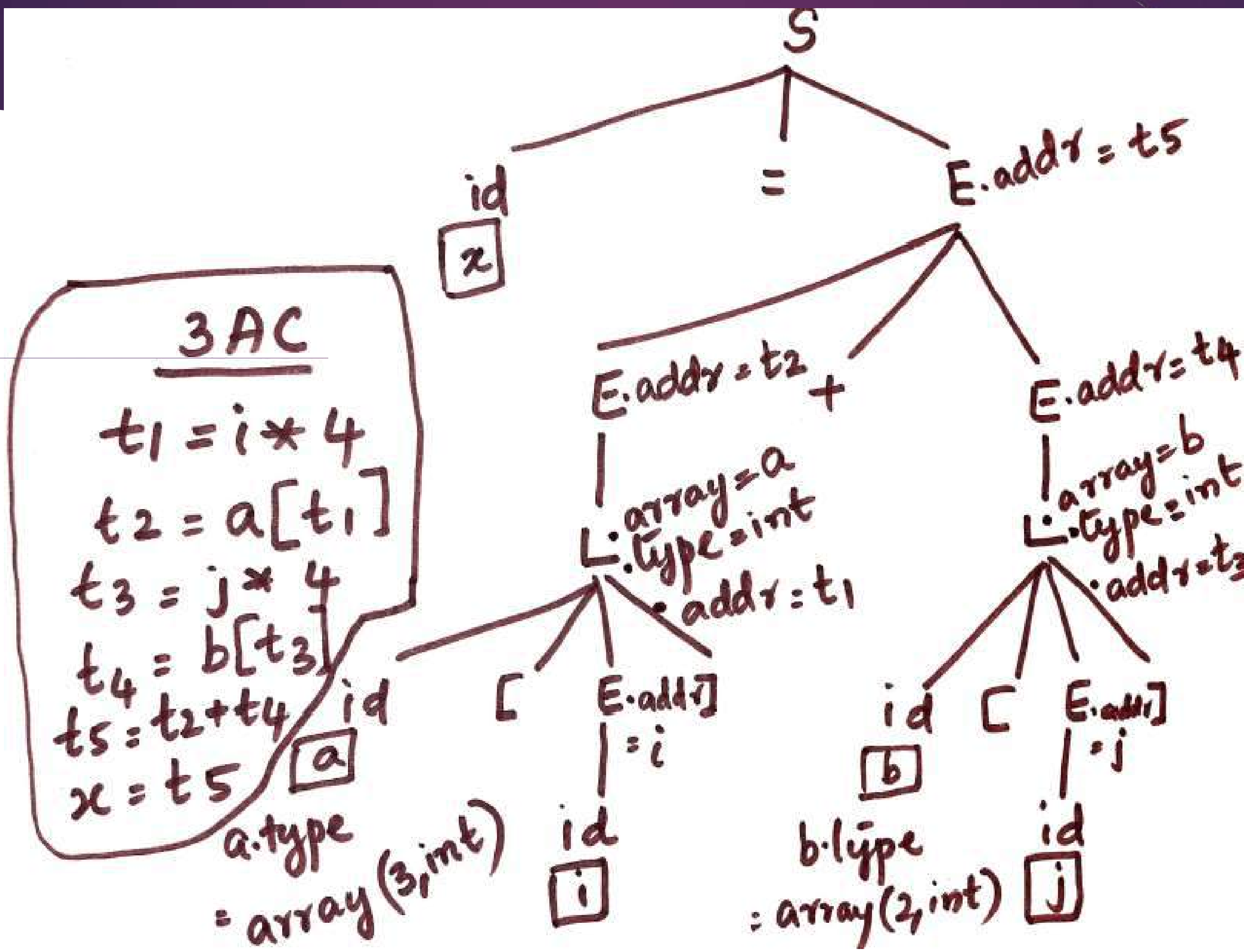
t_1 = i *
t_2 = j *
t_3 = t_1 +
t_4 = a [
t_5 = c +

e the Translation Scheme and translate
e following assignment statements

x=a[i] + b[j]

- Both a and b are integer arrays a[3] and b[2] respectively.

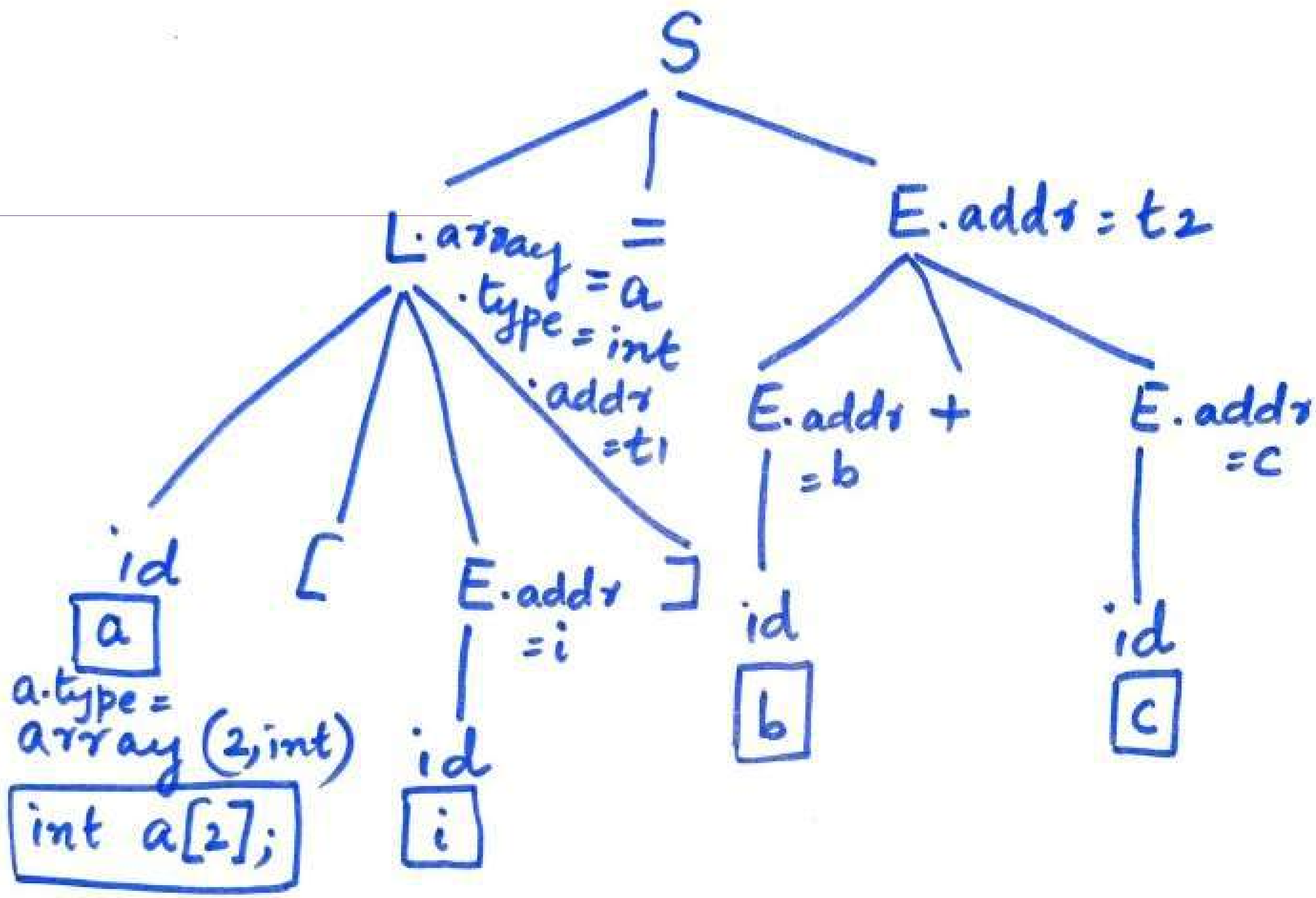
Translation of $x = a[i] + b[j]$



Semantic Actions for Array References

$$
\begin{array}{llll}
S & \rightarrow & \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
 & | & L = E\ ; & \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & | & \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\
 & | & L & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L & \rightarrow & \mathbf{id}\ [\ E\ ] & \{\ L.array = top.get(\mathbf{id}.lexeme); \\
 & & & \quad L.type = L.array.type.elem; \\
 & & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & | & L_1\ [\ E\ ] & \{\ L.array = L_1.array; \\
 & & & \quad L.type = L_1.type.elem; \\
 & & & \quad t = \mathbf{new}\ Temp\,(); \\
 & & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(t\ '='\ E.addr\ '*'\ L.type.width); \\
 & & & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Annotated parse tree for $a[i] = b+c$



<u>3AC</u>

$t_1 = i * 4$

$t_2 = b + c$

$a[t_1] = t_2$

Type Checking

- To do type checking a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.
- Type checking has the potential for catching errors in programs.
- A **sound type system** eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.
- An implementation of a language is **strongly typed** if a compiler guarantees that the programs it accepts will run without type errors.

Type Checking- 2 forms

➢Type checking can take on **two** forms: **synthesis and inference**. **Type synthesis** builds up the type of an expression from the types of **its subexpressions**.

$E \rightarrow E_1 + E_2$

if f has type $s \rightarrow t$ **and** x has type s,
then expression $f(x)$ has type t

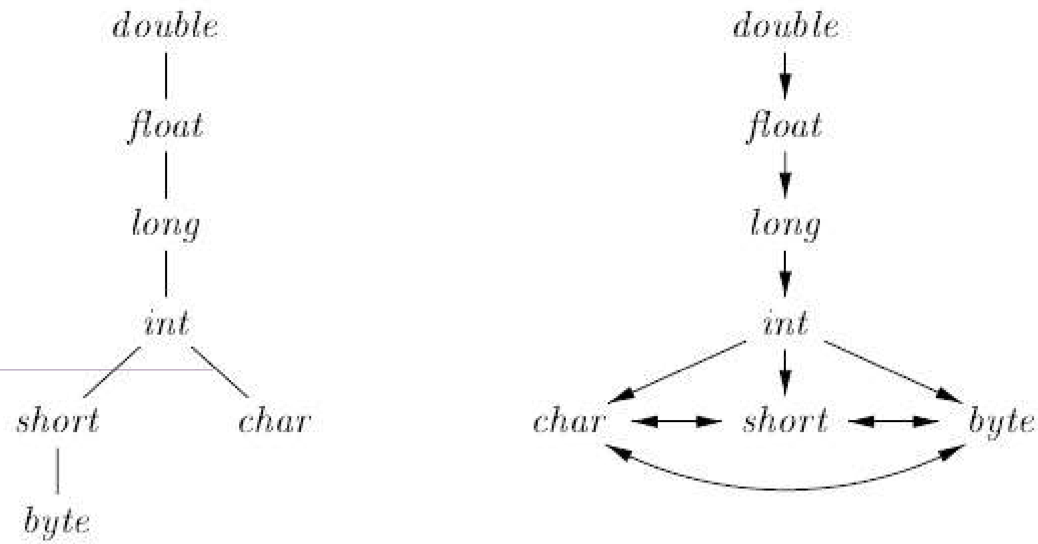
➢Here, f and x denote expressions, and s -> t denotes a function from s to t.

➢**Type inference** determines the type of a language construct from the way it is used.

if $f(x)$ is an expression,
then for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

➢ Let null be a function that tests whether a list is empty. Then, from the usage null(x), we can tell that x must be a list.

Type Conversions



(a) Widening conversions (b) Narrowing conversions

- **Widening conversions**, which are intended to preserve information, and **narrowing conversions**, which can lose information.
- A type s can be narrowed to a type t if there is a path from s to t. Conversion from one type to another is said to be implicit if it is done automatically by the compiler.
- **Implicit type** conversions, also called **coercions.**

Type Conversions

- Conversion is said to be explicit if the programmer must write something to cause the conversion. **Explicit conversions are also called casts**.
- The semantic action for checking E -> E1 + E2 uses two functions:
- **max(t1,t2)** takes two types t1 and t2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy.
- **widen(a , t , w)** generates type conversions if needed to widen the contents of an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary, which is returned as the result.

Pseudocode for function *widen*

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

ntroducing type conversions into expression evaluation

$$\textbf{if } (\ E_1.type = integer \textbf{ and } E_2.type = integer\ )\ \ E.type = integer;$$
$$\textbf{else if } (\ E_1.type = float \textbf{ and } E_2.type = integer\ )\ \cdots$$
$$\cdots$$

$$\begin{array}{lcll} E & \rightarrow & E_1 + E_2 & \{\ E.type \ = \ max(E_1.type, E_2.type); \\ & & & \quad a_1 \ = \ widen(E_1.addr, E_1.type, E.type); \\ & & & \quad a_2 \ = \ widen(E_2.addr, E_2.type, E.type); \\ & & & \quad E.addr = \textbf{new } Temp\,(); \\ & & & \quad gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \} \end{array}$$

Let us now translate
the expression
2*3.14

=>

<u>**Three Address Translation**</u>

t1 = (float) 2;
t2 = t1 * 3.14;

Algorithm for unification

- Unification is the problem of determining whether two type expressions s and t can be made identical by substituting expressions for the variables in s and t
- If s and t have constants, but no variables then s and t unify, if and only if they are identical.
- The unification algorithm extends to graphs with cycles, so we can test structural equivalence.
- Type variables are represented by leaves and type constructors by interior nodes. Nodes are grouped into equivalence classes. If two nodes are in same equivalence class, then the type expressions they represent must unify.

Two operations on nodes

- find(m)-returns the representative node of the equivalence class currently containing node n.
- union(m,n)-merges the equivalence classes containing nodes m and n. If one of the representatives of m and n is a non variable node, union operation helps in simply changing the set field of one equivalence class substitutes the other.

Unification Algorithm

$boolean\ unify(Node\ m, Node\ n)$ {
    $s = find(m);\ t = find(n);$
    if ($s = t$) **return** $true$;
    else if (nodes s and t represent the same basic type) **return** $true$;
    else if (s is an op-node with children s_1 and s_2 **and**
            t is an op-node with children t_1 and t_2) {
        $union(s, t);$
        return $unify(s_1, t_1)$ **and** $unify(s_2, t_2)$;
    }
    else if (s or t represents a variable) {
        $union(s, t);$
        return $true$;
    }
    else return $false$;
}

Thank you

Control Flow

- The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions. In programming languages, boolean expressions are often used to
- 1. Alter the flow of control. Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program.
- 2.Compute logical values. A boolean expression can represent true or false as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.
- Boolean expressions are composed of the boolean operators (which we denote &&, ||, and !), using the C convention for the operators AND, OR, and NOT respectively.
- B->B || B | B && B | !B | (B) | E rel E | true |false
- rel-> < |<= |> |>= |!= |=

Short-Circuit Code

- In short-circuit (or jumping) code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.
- The statement

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

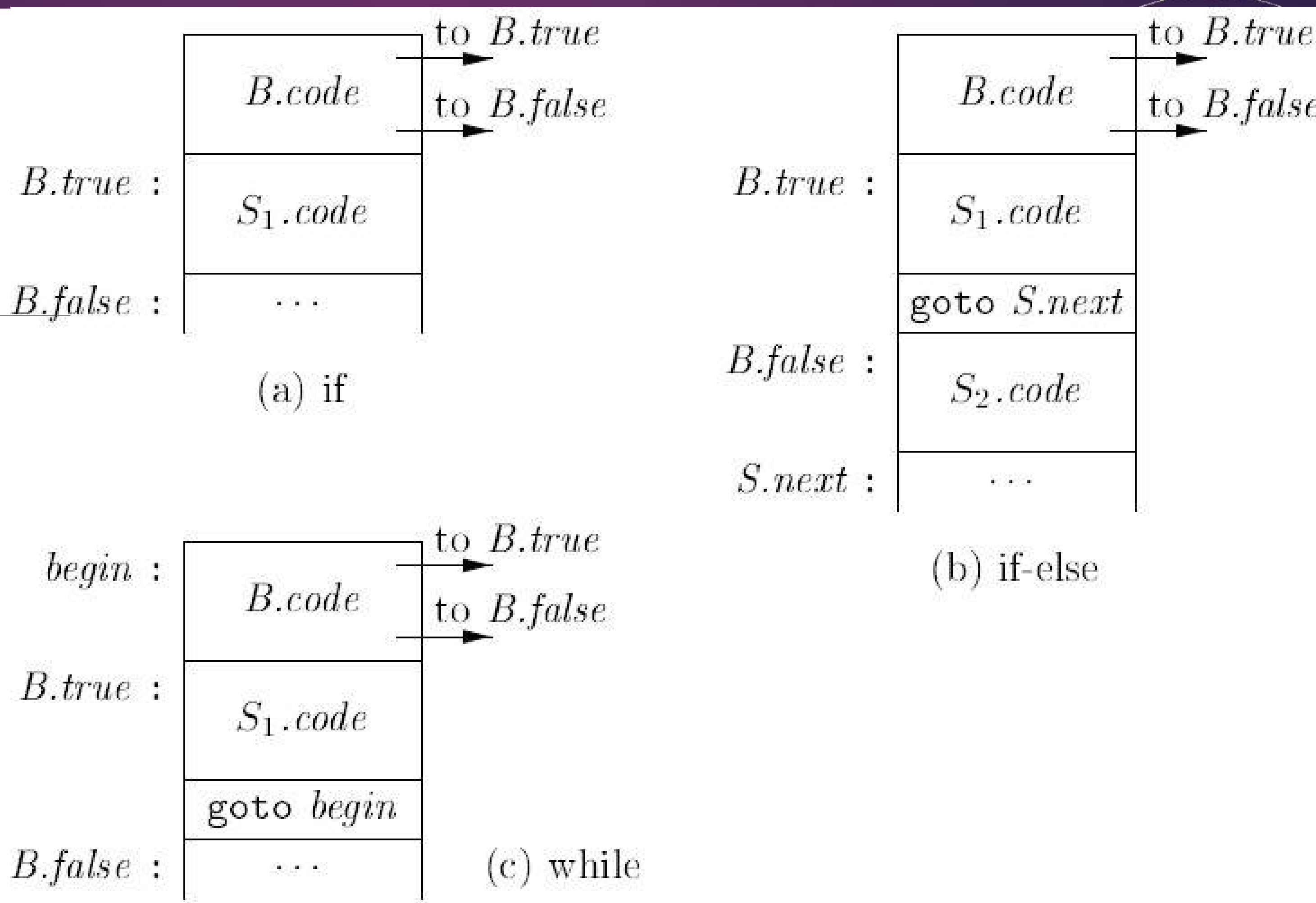
can be translated into

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

Jumping Code

Flow of Control Statements

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



Syntax Directed Definition for flow of control statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

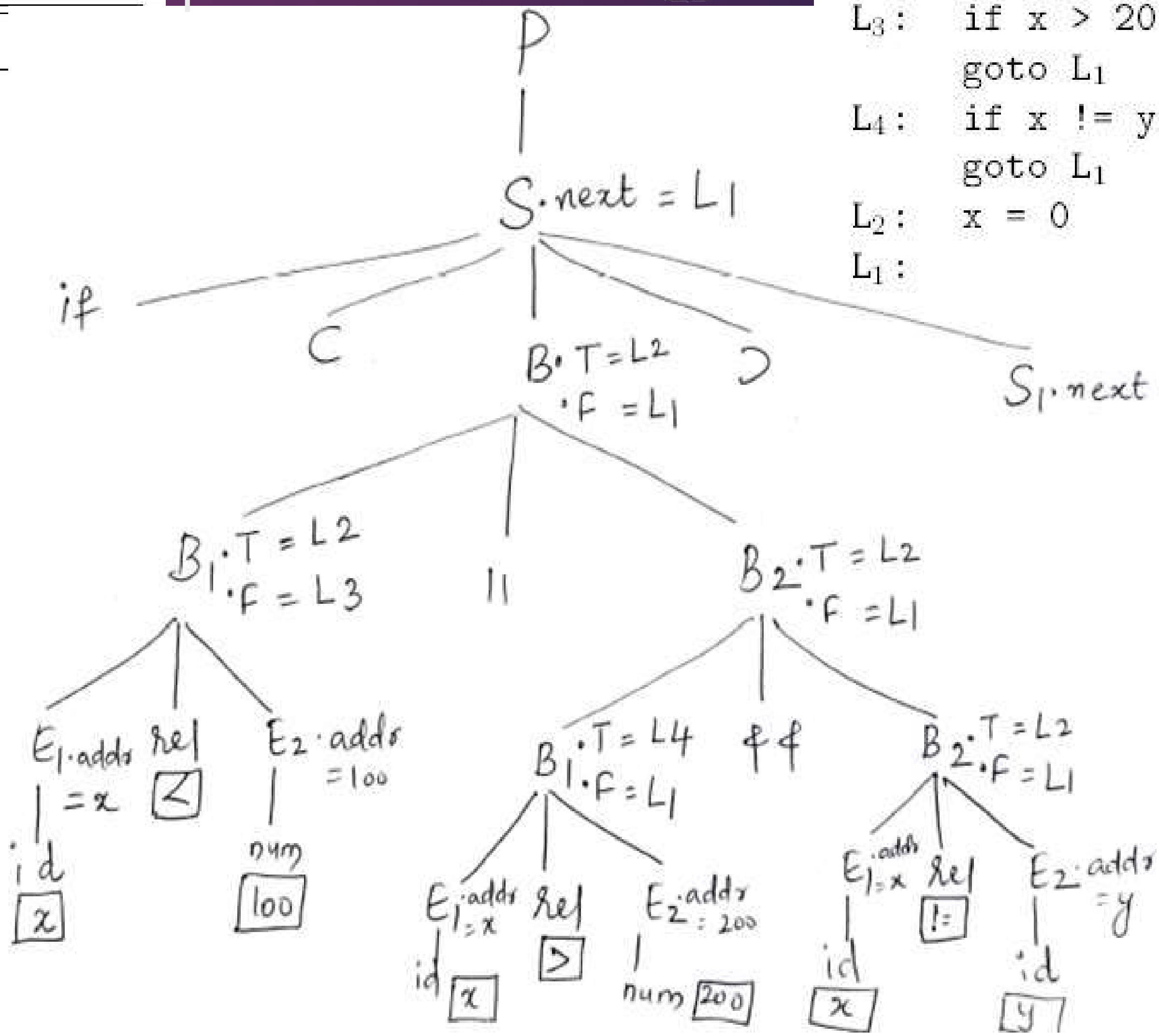
next, true, false-inherited attributes
code- synthesized attribute

ntrol-Flow Translation Boolean Expressions

Control-Flow Translation of a simple-if statement

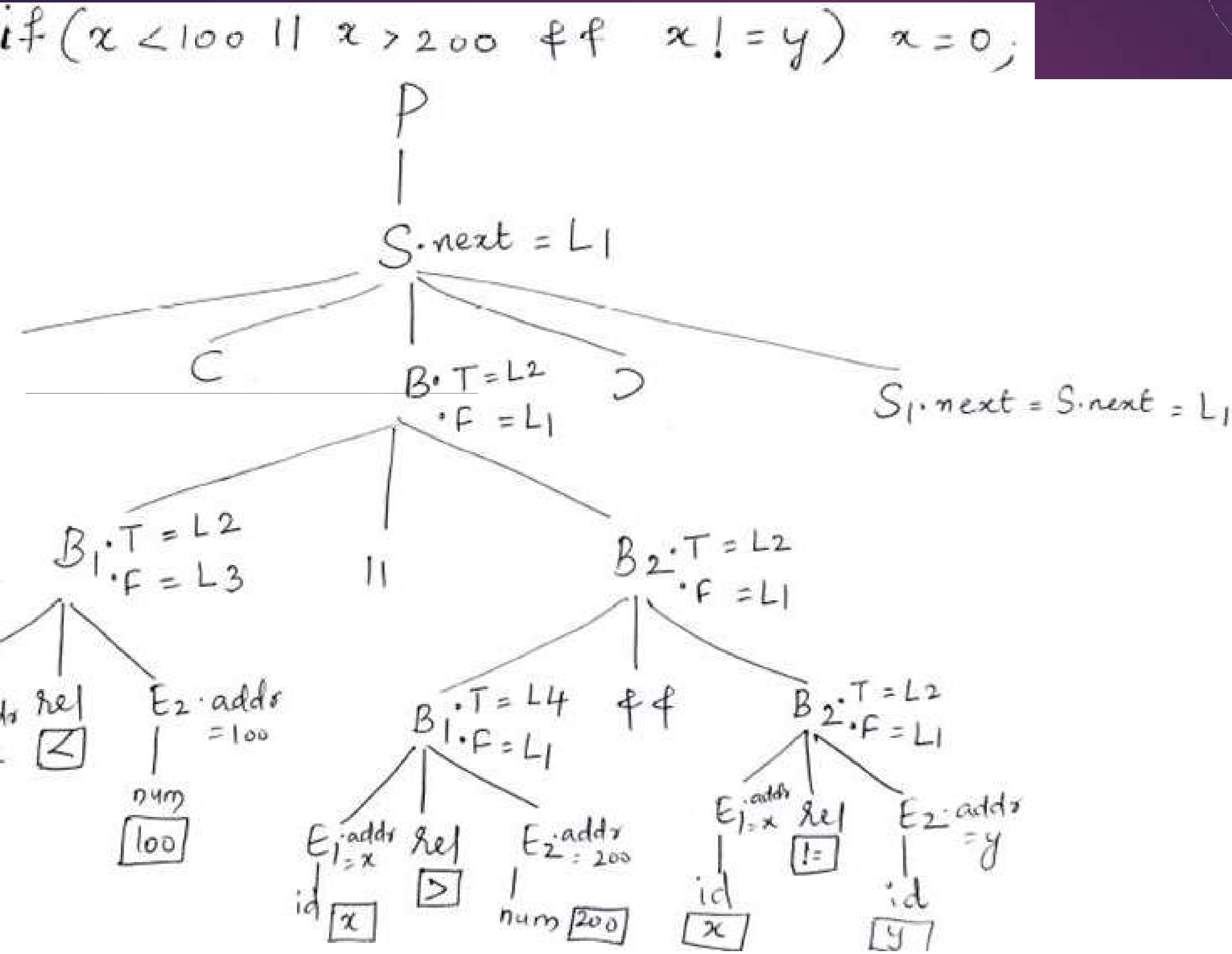
Three Addres

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




```
     if x < 10
     goto L3
L3 : if x > 20
     goto L1
L4 : if x != y
     goto L1
L2 : x = 0
L1 :
```

ontrol-Flow Translation of a simple-if statement



Three Address Code

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

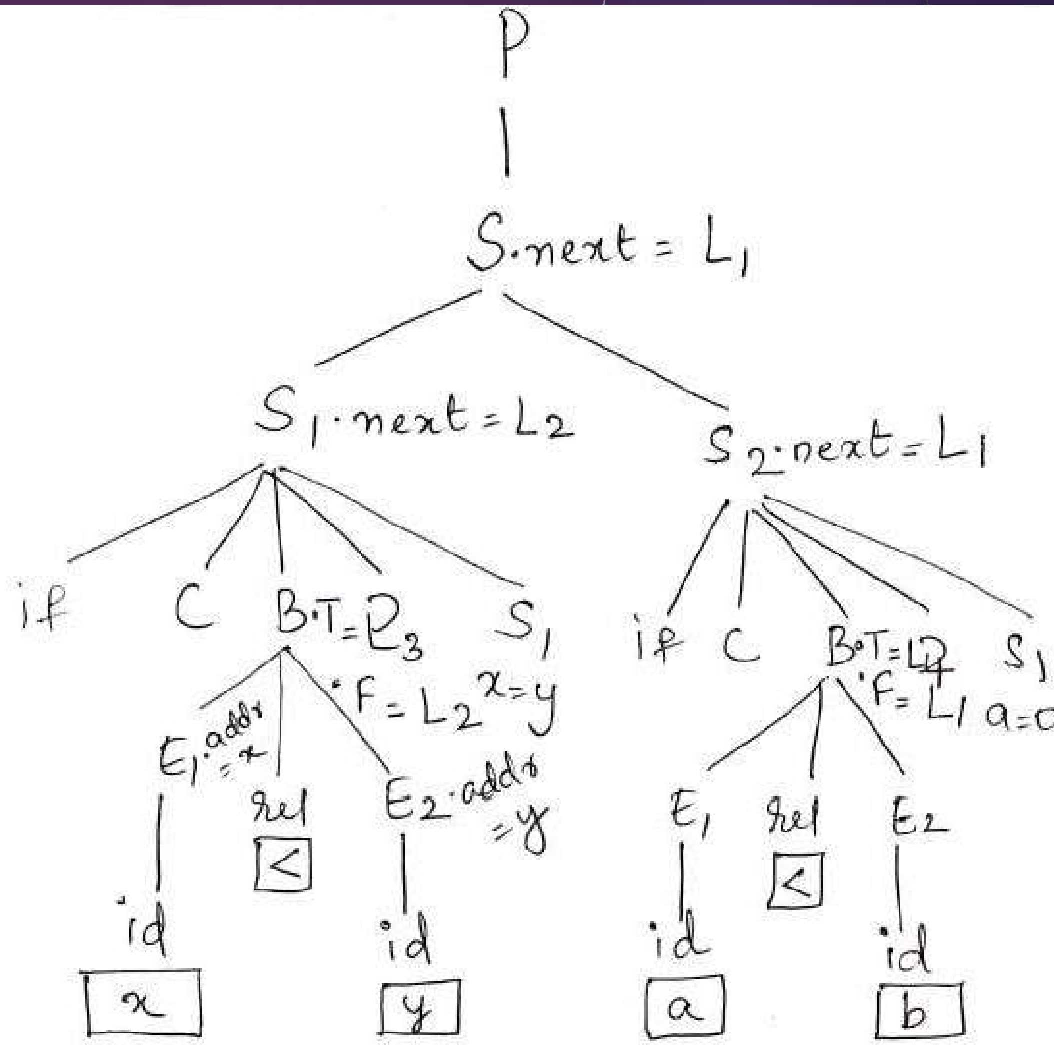
Syntax Directed Definition for flow of control statements

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

(x < y) x = y;
(a < b) a = 0;

Three Address Code

if x < y goto L3
goto L2
L3 : x = y
-2 : if a < b goto L4
goto L1
-4 : a = 0
1 : — — —



Syntax Directed Definition for flow of control statements

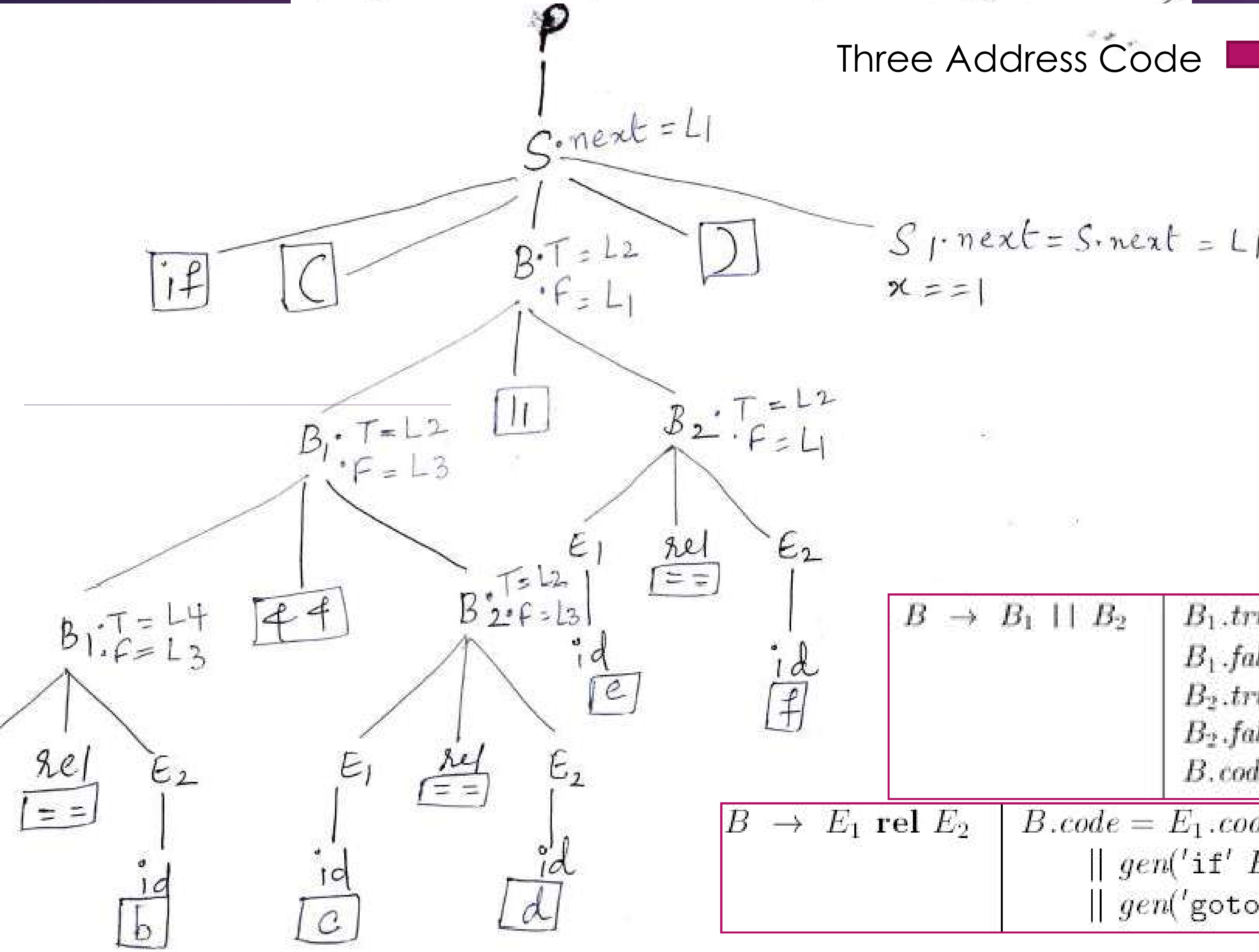
PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$

next, true, false-inherited attributes
code- synthesized attribute

ontrol-Flow Translation of Boolean Expressions

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

if (a==b && c==d || e==f) x==1;



Three Address Code

if a==b goto
goto L3
L4: if c==d goto
goto L3
L3: if e==f goto
goto L1
L2: x==1
L1: ---

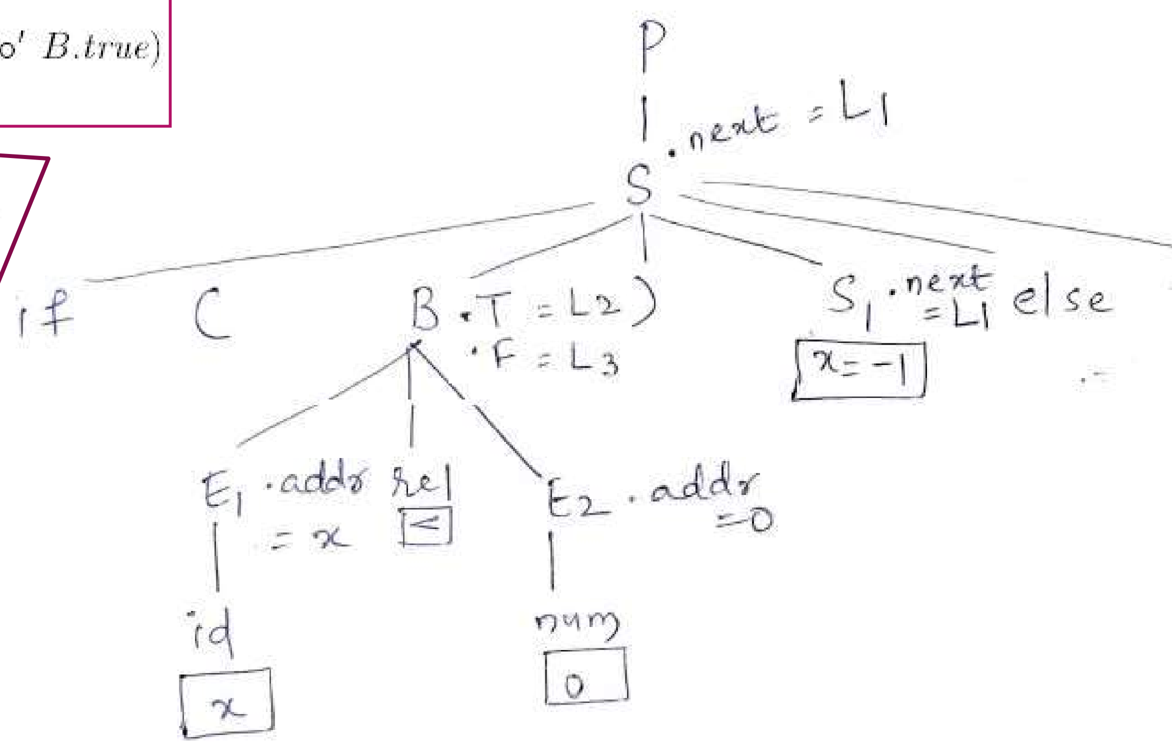
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'go $\mid\mid gen($'goto' $B.false)$

UCTION	SEMANTIC RULES
S	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
if (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \|\| \; label(B.true) \;\|\|\; S_1.code$ $\quad \|\| \; gen(\texttt{'goto'}\; S.next)$ $\quad \|\| \; label(B.false) \;\|\|\; S_2.code$
E_1 **rel** E_2	$B.code = E_1.code \;\|\|\; E_2.code$ $\quad \|\| \; gen(\texttt{'if'}\; E_1.addr\; \textbf{rel}.op\; E_2.addr\; \texttt{'goto'}\; B.true)$ $\quad \|\| \; gen(\texttt{'goto'}\; B.false)$

if (x < 0)   x = -1   else   x = 1

dress Code:

```
    if x < 0 goto L2
    goto L3
L2: x = -1
    goto L1
L3: x = 1
L1: - - -
```

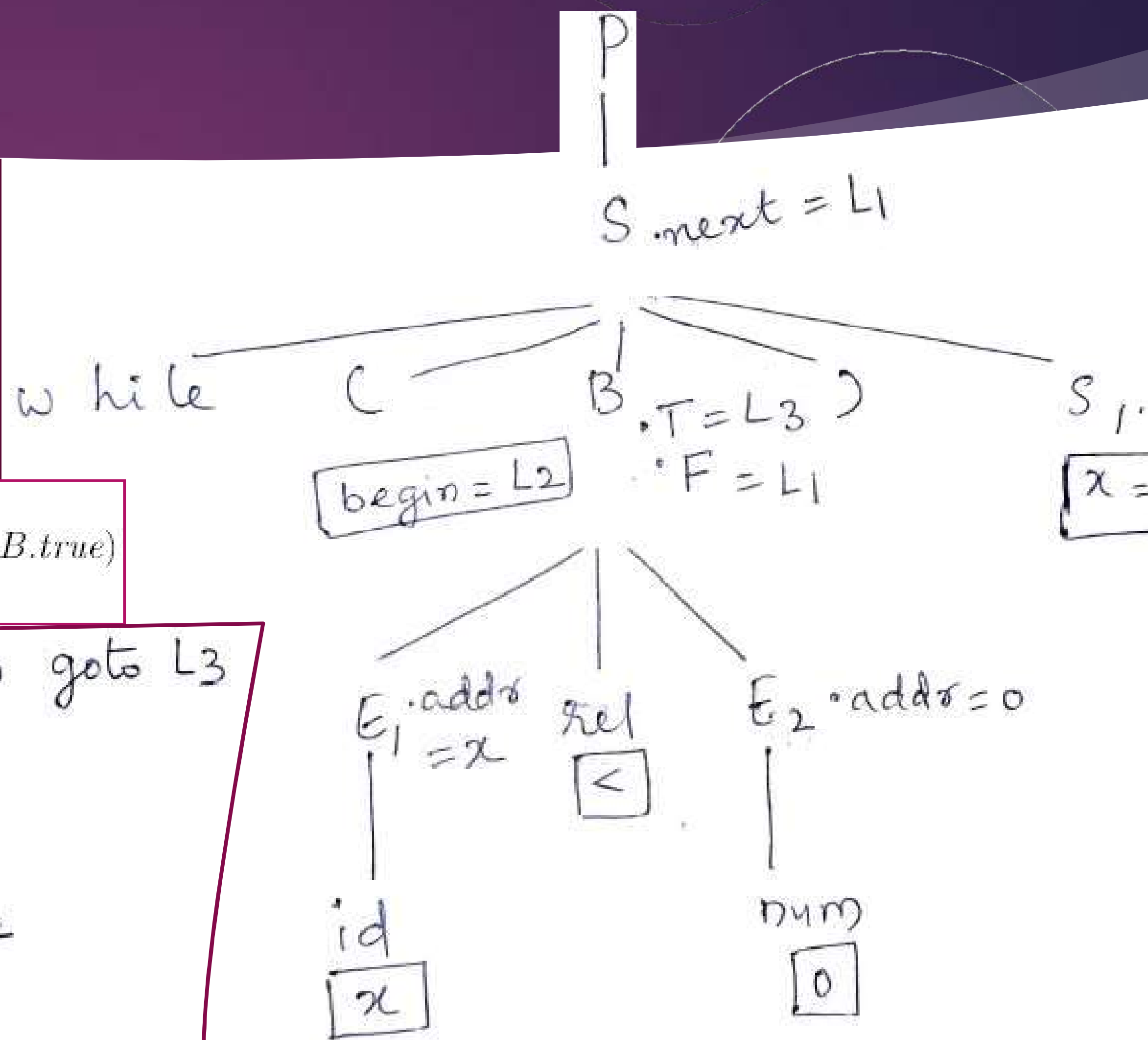


while $(x < 0)$ $x = -1;$

…TION	SEMANTIC RULES
	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
…hile (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen($'goto' $begin)$
…rel E_2	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\| \; gen($'if' $E_1.addr$ **rel**$.op$ $E_2.addr$ 'goto' $B.true)$ $\|\| \; gen($'goto' $B.false)$

```
L2:  if x < 0 goto L3
     goto L1
L3:  x = -1
     goto L2
L1:  _ _ _
```

…Address Code



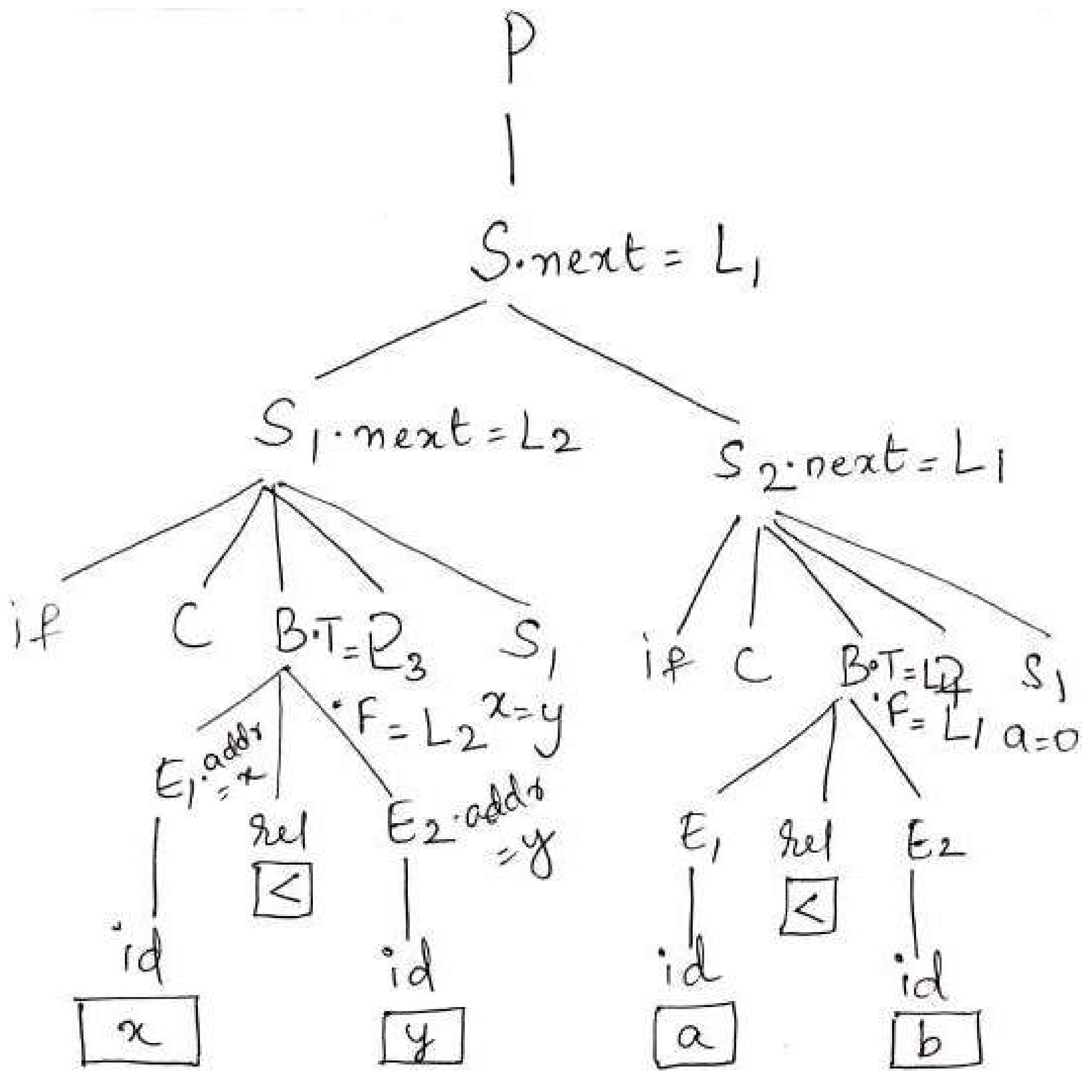
Syntax Directed Definition for flow of control statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

$(x < y)$ $x = y;$
$(a < b)$ $a = 0;$

Three Address Code

if $x < y$ goto L_3
goto L_2
L_3 : $x = y$
L_2 : if $a < b$ goto L_4
goto L_1
L_4 : $a = 0$
L_1 : — — —



Avoiding Redundant Gotos

if x > 200 goto L4

goto L1

L4: ……

Instead, consider the instruction:

ifFalse x > 200 goto L1

L4: ……..

Short-circuit code/Jumping Code

This ifFalse instruction takes advantage of the natural flow from one instruction to the next in sequence, so control simply "falls through" to label L4

if x > 200, thereby avoiding a jump.

Fall – Through Technique

- By using a special label fall (i.e., “don't generate any jump"), we can adapt the semantic rules to allow control to fall through from the code for B to the code for S1.
- The new rules for S -> if (B) S1 is set B:true to fall

$$
\begin{aligned}
B.true &= fall \\
B.false &= S_1.next \;=\; S.next \\
S.code &= B.code \;||\; S_1.code
\end{aligned}
$$

Similarly, the rules for if-else- and while-statements also set $B.true$ to $fall$.

- We now adapt the semantic rules for boolean expressions to allow control to fall throug whenever possible. Suppose B:true is fall ; i.e, control falls through B, if B evaluates to true.

Semantic Rules for B-> E_1 *rel* E_2

The rules for P -> S create label L1.

$$test = E_1.addr \ \mathbf{rel}.op \ E_2.addr$$

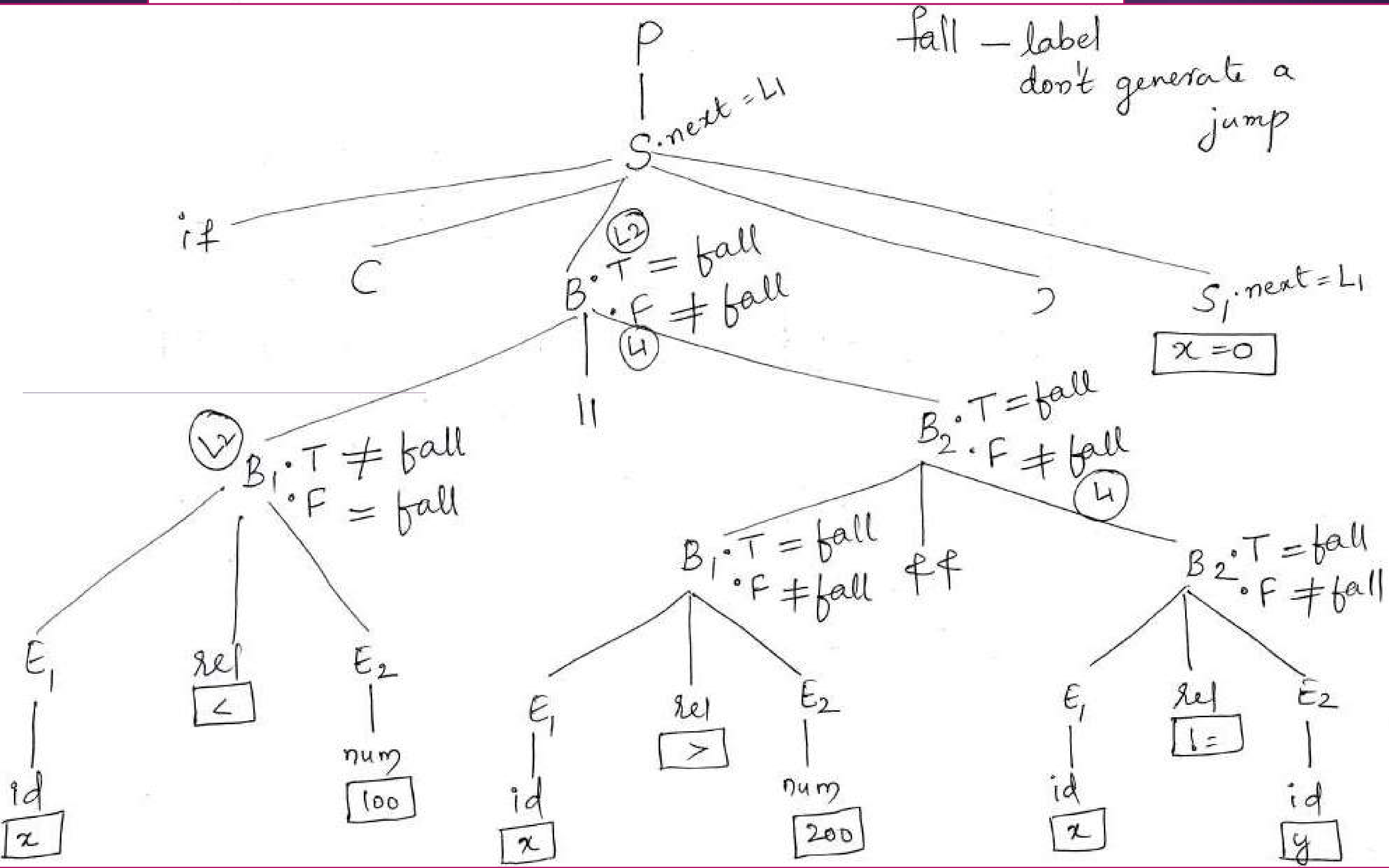
$$\begin{aligned} s = \ & \mathbf{if} \ B.true \neq fall \ \mathbf{and} \ B.false \neq fall \ \mathbf{then} \\ & \quad gen('\texttt{if}' \ test \ '\texttt{goto}' \ B.true) \ || \ gen('\texttt{goto}' \ B.false) \\ & \mathbf{else \ if} \ B.true \neq fall \ \mathbf{then} \ gen('\texttt{if}' \ test \ '\texttt{goto}' \ B.true) \\ & \mathbf{else \ if} \ B.false \neq fall \ \mathbf{then} \ gen('\texttt{ifFalse}' \ test \ '\texttt{goto}' \ B.false) \\ & \mathbf{else} \ '' \end{aligned}$$

$$B.code = E_1.code \ || \ E_2.code \ || \ s$$

Semantic Rules for B-> B_1 || B_2

$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \;||\; B_2.code \\
&\quad\;\; \textbf{else } B_1.code \;||\; B_2.code \;||\; label(B_1.true)
\end{aligned}
$$

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



B.code = B.true = fall

Yes B.true = fall then

B_1.code ⇒

if x < 100 goto L2

B_2.code ⇒

iffalse x > 200 goto L1

ifFalse x != y goto L1

B.code = B_1.code || B_2.code || label(B_1.true)

if x < 100 goto L2

iffalse x > 200 goto L1

iffalse x != y goto L1

L2 :

$B \rightarrow B_1 \,||\, B_2$

test = E_1.addr rel.op E_2

① E_1.addr = x; rel.op = <
E_2.addr = 100

B.T ≠ fall then

if x < 100 goto L2

② E.addr = x; rel.op = >
E_2.addr = 200

B.F ≠ fall then

iffalse x > 200 goto L

③ E.addr = x; rel.op = !=
E.addr = y

B.F ≠ fall then

iffalse x != y goto L

ewriting using 'fall' label: fall through echnique/ Short-circuit code

$S \to$ if (B) S_1

B.code is computed

S_1.code $\Rightarrow$ $\boxed{x = 0}$

if x <100 goto L2

iffalse x > 200 goto L1

iffalse x != y goto L1

L2 : x = 0

L1 : — — —

S.code = B.code || S_1.code

B.code

L2 : x = 0

$P \to S$

P.code = S.code || label(S.next)

S.code

L1 : ____

statement translated using the fall-through chnique

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

translates into the code of

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

Boolean Values and Jumping Code

A clean way of handling both roles of boolean expressions is to first build a yntax tree for expressions, using either of the following approaches:

1. *Use two passes.* Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.

2. *Use one pass for statements, but two passes for expressions.* With this approach, we would translate E in **while** (E) S_1 before S_1 is examined. The translation of E, however, would be done by building its syntax tree and then walking the tree.

The following grammar has a single nonterminal E for expressions:

$$S \rightarrow \textbf{id} = E \; ; \;|\; \textbf{if} \; (\; E \;) \; S \;|\; \textbf{while} \; (\; E \;) \; S \;|\; S \; S$$

$$E \rightarrow E \,||\, E \;|\; E \,\&\&\, E \;|\; E \; \textbf{rel} \; E \;|\; E + E \;|\; (E) \;|\; \textbf{id} \;|\; \textbf{true} \;|\; \textbf{false}$$

Nonterminal E governs the flow of control in $S \rightarrow$ **while** (E) S_1. The same nonterminal E denotes a value in $S \rightarrow \textbf{id} = E$; and $E \rightarrow E + E$.

A boolean expression ma be evaluated for its value, a assignment statements suc x = true; or x =

ethod jump generate jumping code at an expression
ode, and let method rvalue generate code to compute
e value of the node into a temporary

When E appears in $S \rightarrow$ **while** (E) S_1, method *jump* is called at no
$E.n$. The implementation of *jump* is based on the rules for boolean expressio

Specifically, jumping code is generated by calling $E.n.jump(t, f$
where t is a new label for the first instruction of $S_1.code$ and f is the lab
$S.next$.

When E appears in $S \rightarrow$ **id** $=$ E ;, method *rvalue* is called at node $E.n$. If
has the form $E_1 + E_2$, the method call $E.n.rvalue()$ generates code

If E has the form E_1 && E_2, we first generate jumping code fo
E and then assign true or false to a new temporary `t` at the true and false exit
respectively, from the jumping code.

Translating a boolean assignment by computing the value of a temporary

For example, the assignment x = a<b && c<d can be implemented by the code

```
     ifFalse a < b goto L1
     ifFalse c < d goto L1
     t = true
     goto L2
L1:  t = false
L2:  x = t
```

Backpatching

- A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump.
- For example S-> if (B) S1, S contains a jump when B is false.
- We followed an approach like this before, Pass labels as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.
- In a one-pass translation, B must be translated before S is examined.
- Backpatching is a one-pass translation approach.
- In backpatching list of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified.
- Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label. We follow here the translation using position numbers.

One-Pass Code Generation using Backpatching

- Backpatching can be used to generate code for boolean expressions and flow of-control statements in one pass.
- Synthesized attributes truelist and falselist of nonterminal B are used to manage labels in jumping code for boolean expressions.
- In particular, B:truelist will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true.
- B:falselist likewise is the list of instructions that eventually get the label to which control goes when B is false.
- As code is generated for B, jumps to the true and false exits are left incomplete, with the label field unfilled.
- Statement S has a synthesized attribute, S.nextlist denoting a list of jumps to the instruction immediately following the code for S.

To manipulate list of jumps, three functions are used

$makelist(i)$ creates a new list containing only i, an index into the array of instructions; $makelist$ returns a pointer to the newly created list.

$merge(p_1, p_2)$ concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.

$backpatch(p, i)$ inserts i as the target label for each of the instructions on the list pointed to by p.

Backpatching for Boolean Expressions

$$\rightarrow B_1 \;||\; M\, B_2 \mid B_1 \;\&\&\; M\, B_2 \mid !\, B_1 \mid (\, B_1\,) \mid E_1 \textbf{ rel } E_2 \mid \textbf{true} \mid \textbf{false}$$

$$\rightarrow \epsilon$$

Now we will design a translation scheme suitable for generating code for boole expression during bottom up parsing.

A marker nonterminal M in the grammar causes a semantic action to pick up, appropriate times, the index of the next instruction to be generated.

slation
eme for
ean
essions

1) $B \rightarrow B_1 \mid\mid M \; B_2$ $\{$ $backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist; \}$

2) $B \rightarrow B_1 \; \&\& \; M \; B_2$ $\{$ $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist); \}$

3) $B \rightarrow \; ! \; B_1$ $\{$ $B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist; \}$

4) $B \rightarrow (\; B_1 \;)$ $\{$ $B.truelist = B_1.truelist;$
$B.falselist = B_1.falselist; \}$

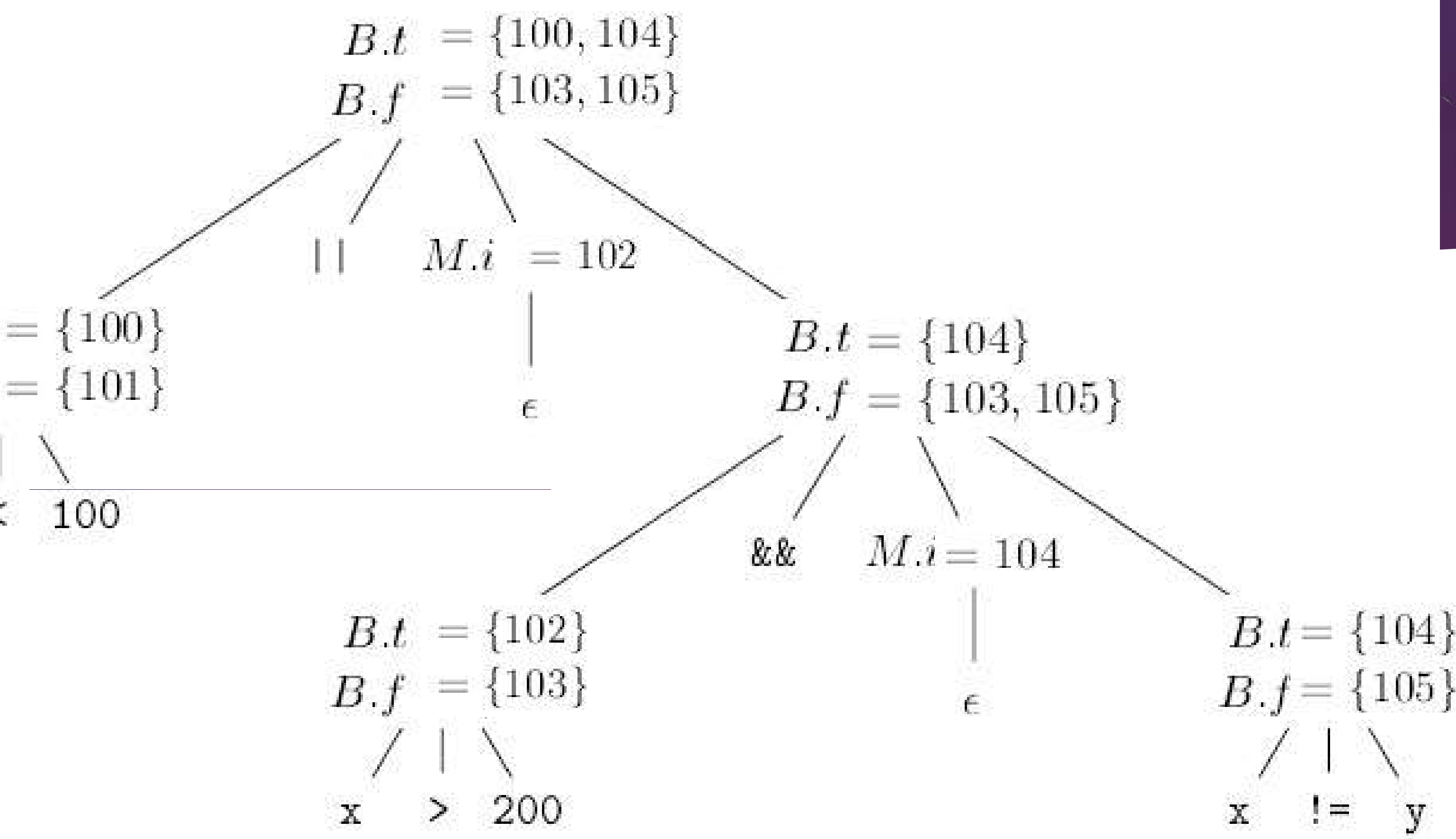
5) $B \rightarrow E_1 \; \mathbf{rel} \; E_2$ $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen('\texttt{if}' \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; '\texttt{goto \_}');$
$gen('\texttt{goto \_}'); \}$

6) $B \rightarrow \mathbf{true}$ $\{$ $B.truelist = makelist(nextinstr);$
$gen('\texttt{goto \_}'); \}$

7) $B \rightarrow \mathbf{false}$ $\{$ $B.falselist = makelist(nextinstr);$
$gen('\texttt{goto \_}'); \}$

8) $M \rightarrow \epsilon$ $\{$ $M.instr = nextinstr; \}$

otated parse tree for $x < 100 \;||\; x > 200 \;\&\&\; x \;!=\; y$



tes truelist, falselist, instr are
ented by t,f and i respectively

Control Flow Translation

```
if x < 100 goto _
goto _
```

are generated. (We arbitrarily start instruction numbers at 100.) The marker nonterminal M in the production

$$B \rightarrow B_1 \; || \; M \, B_2$$

records the value of *nextinstr*, which at this time is 102. The reduction of $x > 200$ to B by production (5) generates the instructions

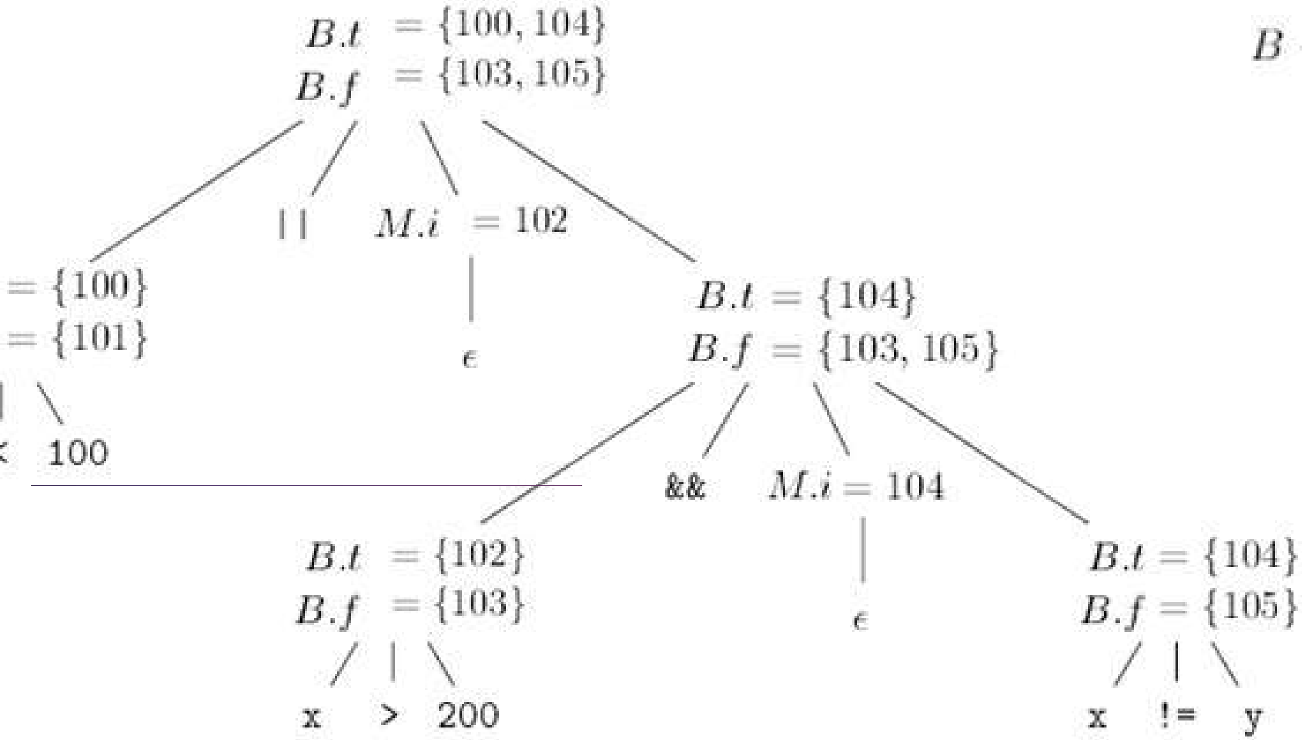
```
:   if x > 200 goto _
:   goto _
```

subexpression $x > 200$ corresponds to B_1 in the production $B \rightarrow B_1 \; \&\& \; M \; B_2$

he marker nonterminal M records the current value of *nextinstr*, which is now)4. Reducing $x \; != \; y$ into B by production (5) generates

```
104:   if x != y goto _
105:   goto _
```

anslation using backpatching



$B \rightarrow B_1 \; || \; M \; B_2 \quad \{ \; backpatch(B_1.falselist$

```
100:   if x < 100 goto _
101:   goto 102
102:   if x > 200 goto 10
103:   goto _
104:   if x != y goto _
105:   goto _
```

$B \rightarrow B_1 \; \&\& \; M \; B_2 \quad \{ \; backpatch(B_1.truel$

s in the
kpatching
ess

```
100:   if x < 100 goto _
101:   goto _
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

(a) After backpatching 104 into instruction 102.

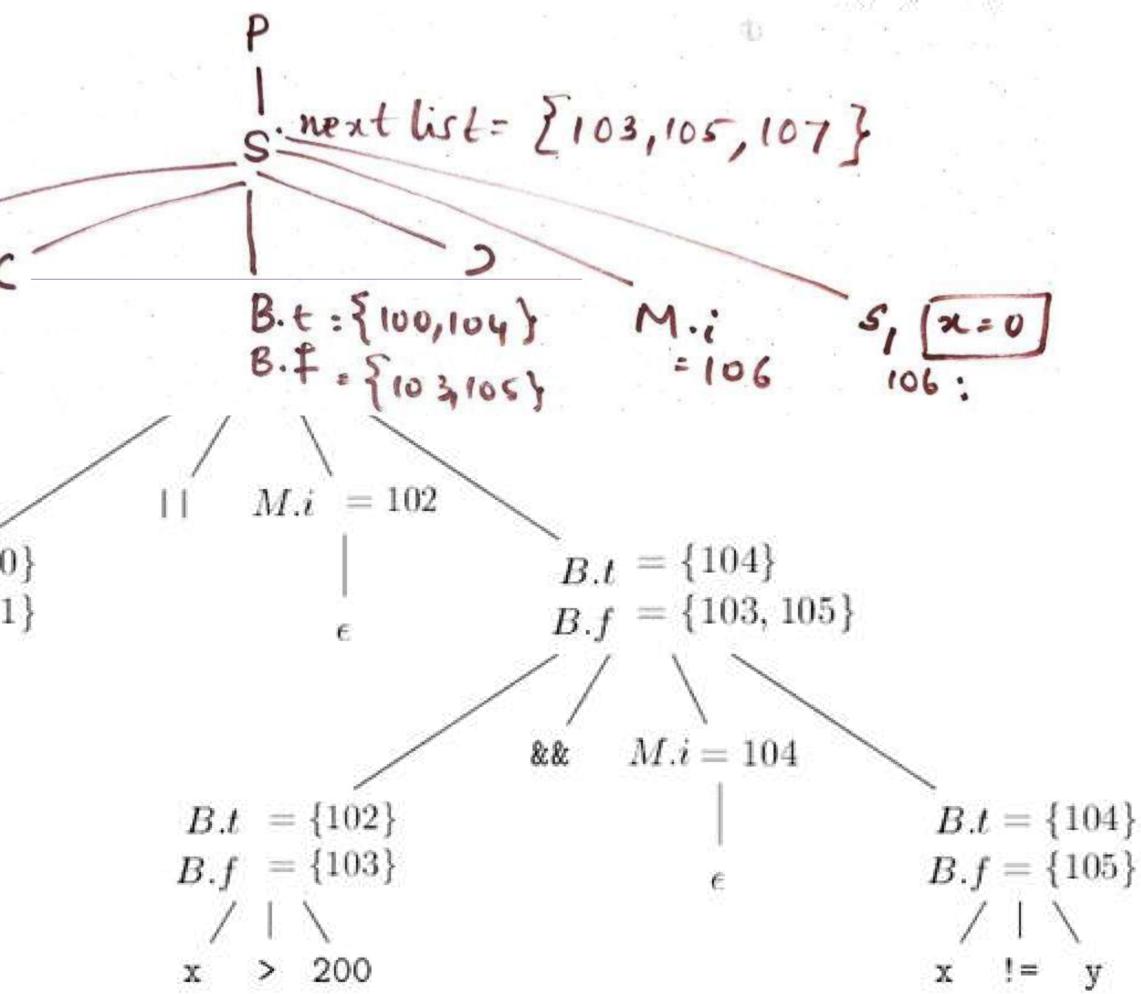
```
100:   if x < 100 goto _
101:   goto 102
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

(b) After backpatching 102 into instru

anslation of if statement through backpatchin

$\rightarrow$ **if** $(B)\ M\ S_1$ $\{$ $backpatch(B.truelist,\ M.instr);$

$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist);$ $\}$

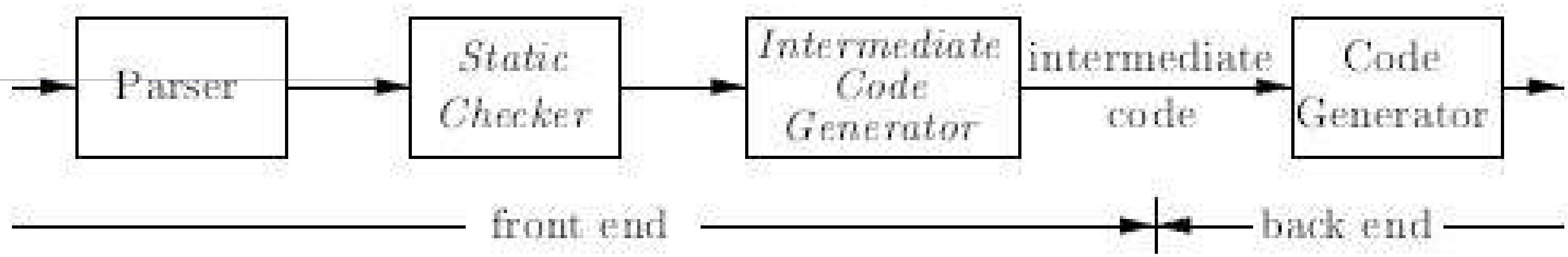



```
100:   if x < 100 goto
101:   goto 102
102:   if x > 200 goto 10
103:   goto 107
104:   if x != y goto 106
105:   goto 107
```

termediate Code Generation

gical structure of a compiler front end

analysis-synthesis model of a compiler, the front end analyzes a source program and creat
nediate representation, from which the back end generates target code.



Logical structure of a compiler front end

ere parsing, static checking, and intermediate-code generation are done sequentially; som
y can be combined and folded into parsing.

ic checking includes type checking, which ensures that operators are applied to co
erands. For example, static checking assures that a break-statement in C is enclosed within
or switch-statement

ompiler might use a sequence of termediate representations

e term **three-address code** comes from instructions of the general form

= y op z with three addresses: two for the operands y and z and one for the result x.

the process of translating a program in a given source language into code for a given targ achine, a compiler may construct a sequence of intermediate representations.

low-level representation is suitable for machine-dependent tasks like register allocation a struction selection. High Level representation a tree like structure.

ariants of Syntax Trees - DAG

directed acyclic graph (hereafter called a DAG) for an expression identifies the co bexpressions (subexpressions that occur more than once) of the expression. Nodes in a synt oresent constructs in the source program; the children of a node represent the mea omponents of a construct.

ted Acyclic Graphs for Expressions

ke the syntax tree for an expression, a DAG has leaves corresponding to atomic operan erior nodes corresponding to operators.

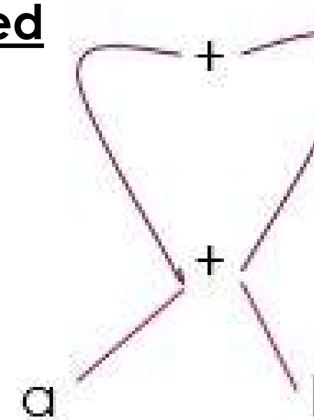
ntax-directed definition to produce syntax ees or DAG's

ODUCTION	SEMANTIC RULES
→ $E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
→ $E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
→ T	$E.node = T.node$
→ (E)	$T.node = E.node$
→ **id**	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
→ **num**	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

<u>Input string to be represented</u>

a + b + (a + b)

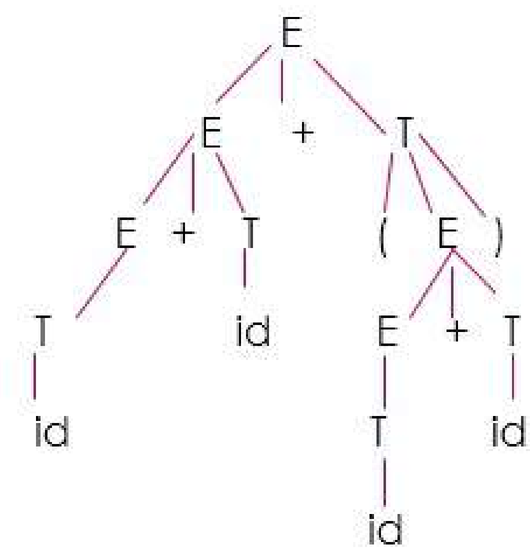
<u>DAG</u>



<u>Syntax Tree</u>

<u>t string to be represented</u>

b + (a + b)

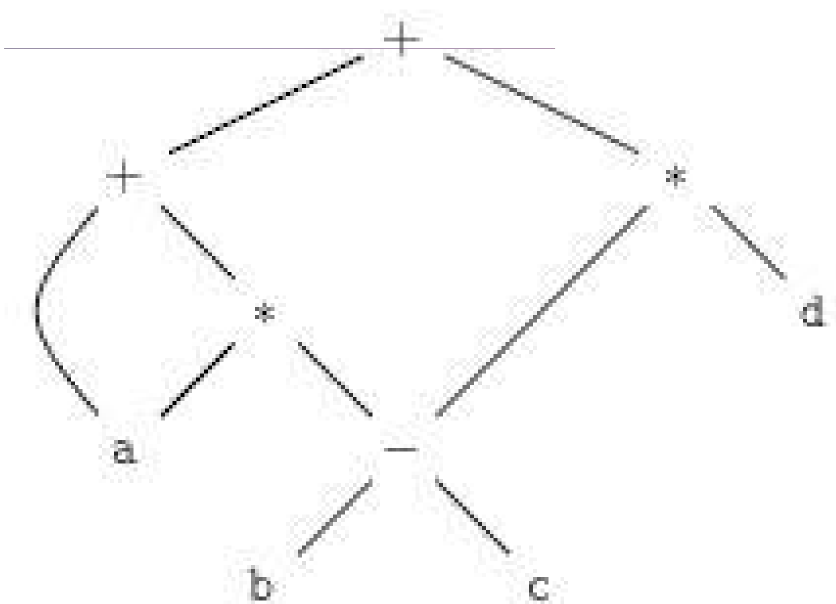


<u>Steps for constructing the D</u>

1) p1= Leaf(id, entry-a)
2) p2 = Leaf(id, entry-b)
3) p3= Node('+',p1,p2)
4) p4=Leaf(id, entry-a) = p
5) p5=Leaf(id, entry-b) = p
6) p6=Node('+',p1,p2) = p
7) p7= Node('+',p3,p3)

onstruct DAG for the expression
+a*(b-c)+ (b-c)*d

DAG



Steps for constructing the DA

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

onstruct the DAG for the expression assuming + associates from the left)

$$((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))$$

$$a+b+a+b.$$

$$a+a+(a+a+a+(a+a+a+a)).$$

utorial Questions

- 1. A-> L M { L.i=f(A.s); M.i=f(L.s); A.s=f(M.s); }

- 2. A->Q R { R.i=f(A.i);Q.i=f(R.i); A.s=f(Q.s); }

- **Is the above definitions S-Attributed or L-attributed?**

onstruct Activation Tree and Activation ecord for the given program

```
ain(){
intf("%d  ", fib(5));

 fib(int num)

if(num == 0 | | num == 1)
    return num;
else
    return fib(num-1) + fib(num-2);
```

anslation Scheme using Backpatching

- One pass Translation
- Only synthesized attributes - list of jumps are passed as synthesized attributes
- Suitable for bottom up parsing
- Uses position numbers

Backpatching for Boolean Expressions

$$\rightarrow B_1 \text{ || } M\ B_2 \mid B_1 \ \&\&\ M\ B_2 \mid !\ B_1 \mid (\ B_1\ ) \mid E_1 \ \mathbf{rel}\ E_2 \mid \mathbf{true} \mid \mathbf{false}$$

$$\rightarrow \epsilon$$

Now we will design a translation scheme suitable for generating code for boolea expression during bottom up parsing.

A marker nonterminal M in the grammar causes a semantic action to pick up, a appropriate times, the index of the next instruction to be generated.

$$M \rightarrow \epsilon \qquad \{\ M.instr = nextinstr;\ \}$$

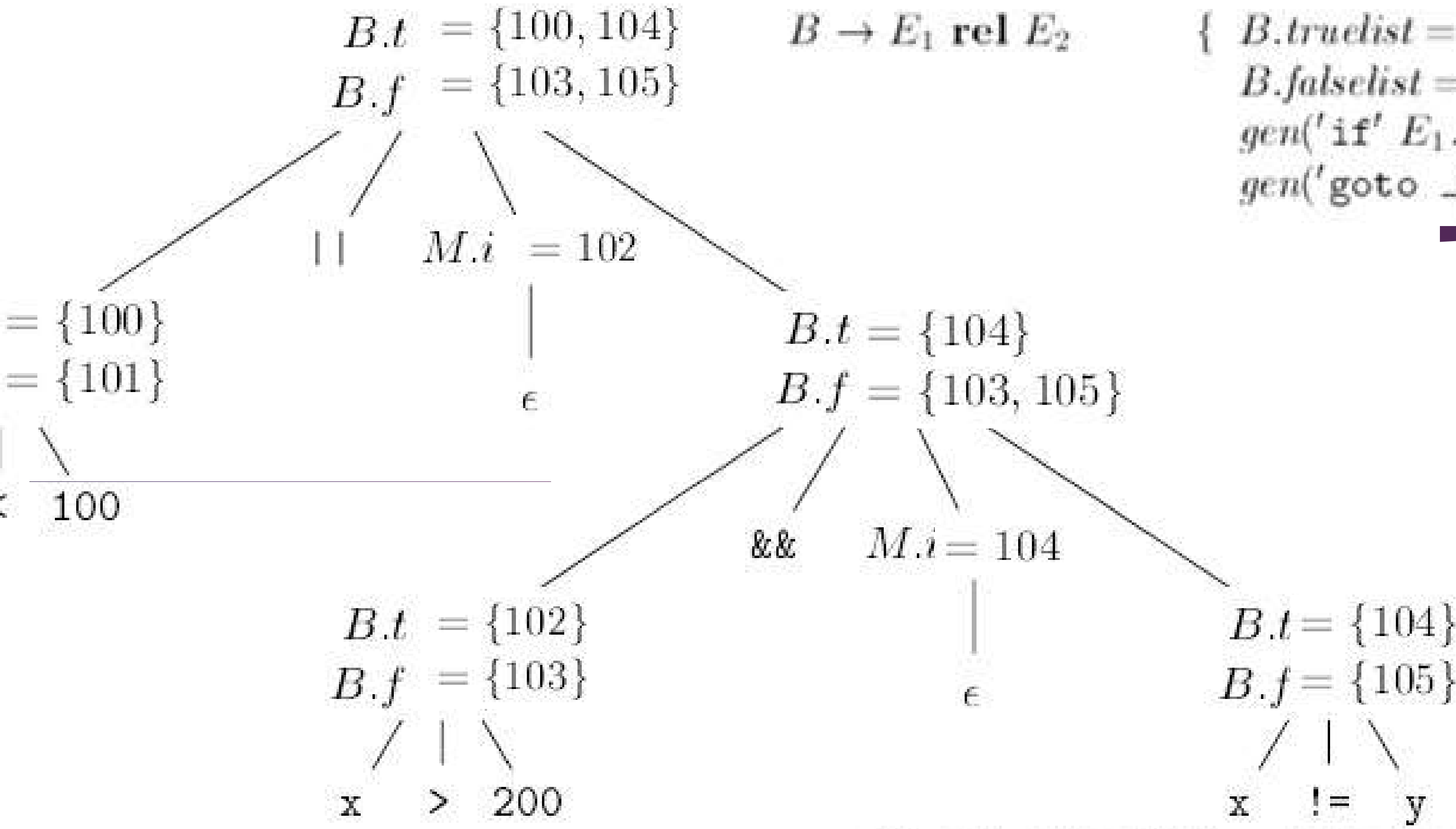
To manipulate list of jumps, three functions are used

$makelist(i)$ creates a new list containing only i, an index into the array of instructions; $makelist$ returns a pointer to the newly created list.

$merge(p_1, p_2)$ concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.

$backpatch(p, i)$ inserts i as the target label for each of the instructions on the list pointed to by p.

notated parse tree for $x < 100 \;||\; x > 200 \;\&\&\; x \;!= \; y$



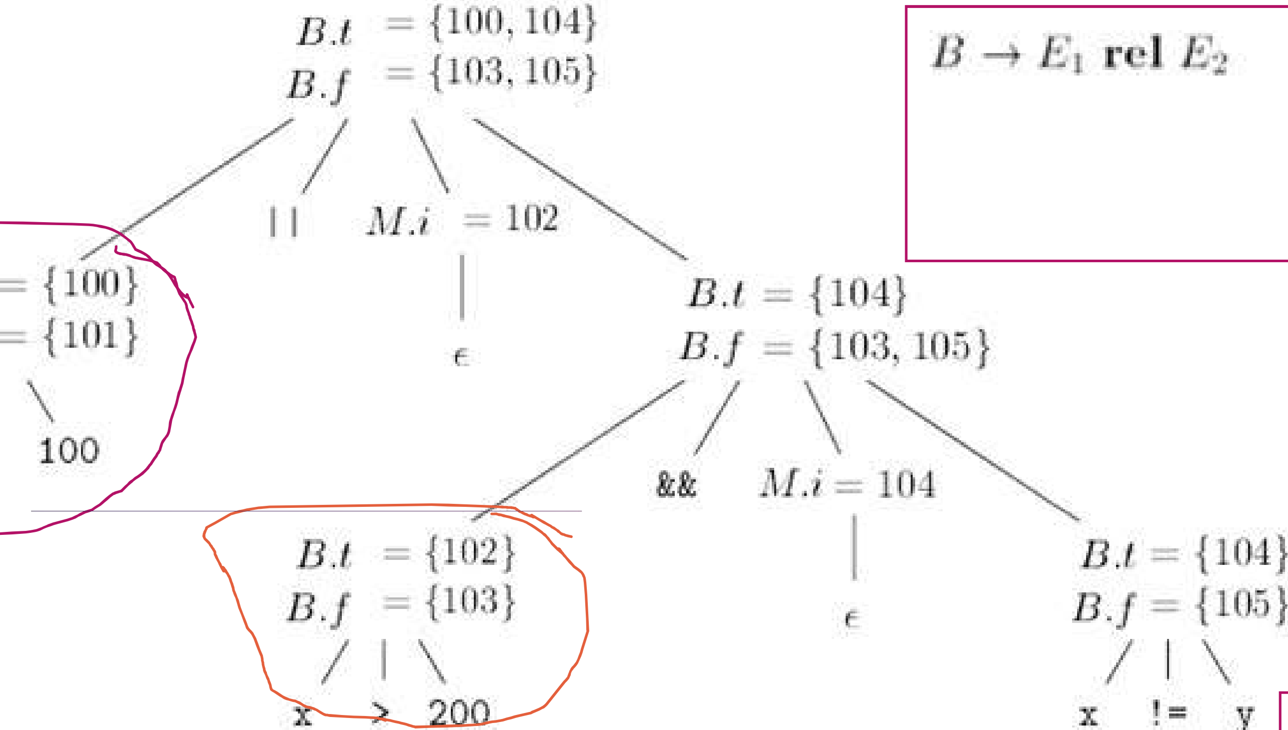
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$ $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen($'if' $E_1.addr$ $\mathbf{rel}.op$ $E_2.addr$ 'goto _'$)$
$gen($'goto _'$); \}$

Attributes truelist, falselist, instr are represented by t,f and i respectively

$B \rightarrow B_1 \ \&\& \ M \ B_2$ $\{$ $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.$

$| \ M \ B_2$ $\{$ $backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist; \}$

anslation using backpatching



$B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen($'if' $E_1.addr$ **rel**$.op$ $E_2.addr$ 'got
$gen($'goto _'$);$ }

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 10
103:  goto _
104:  if x != y goto _
105:  goto _
```

$B \rightarrow B_1 \ \&\& \ M \ B_2$ { $backpatch(B_1.truelist, M.instr);$

$B \rightarrow B_1 \ || \ M \ B_2$ { $backpatch(B_1.falselis$

s in

kpa
ng
cess

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

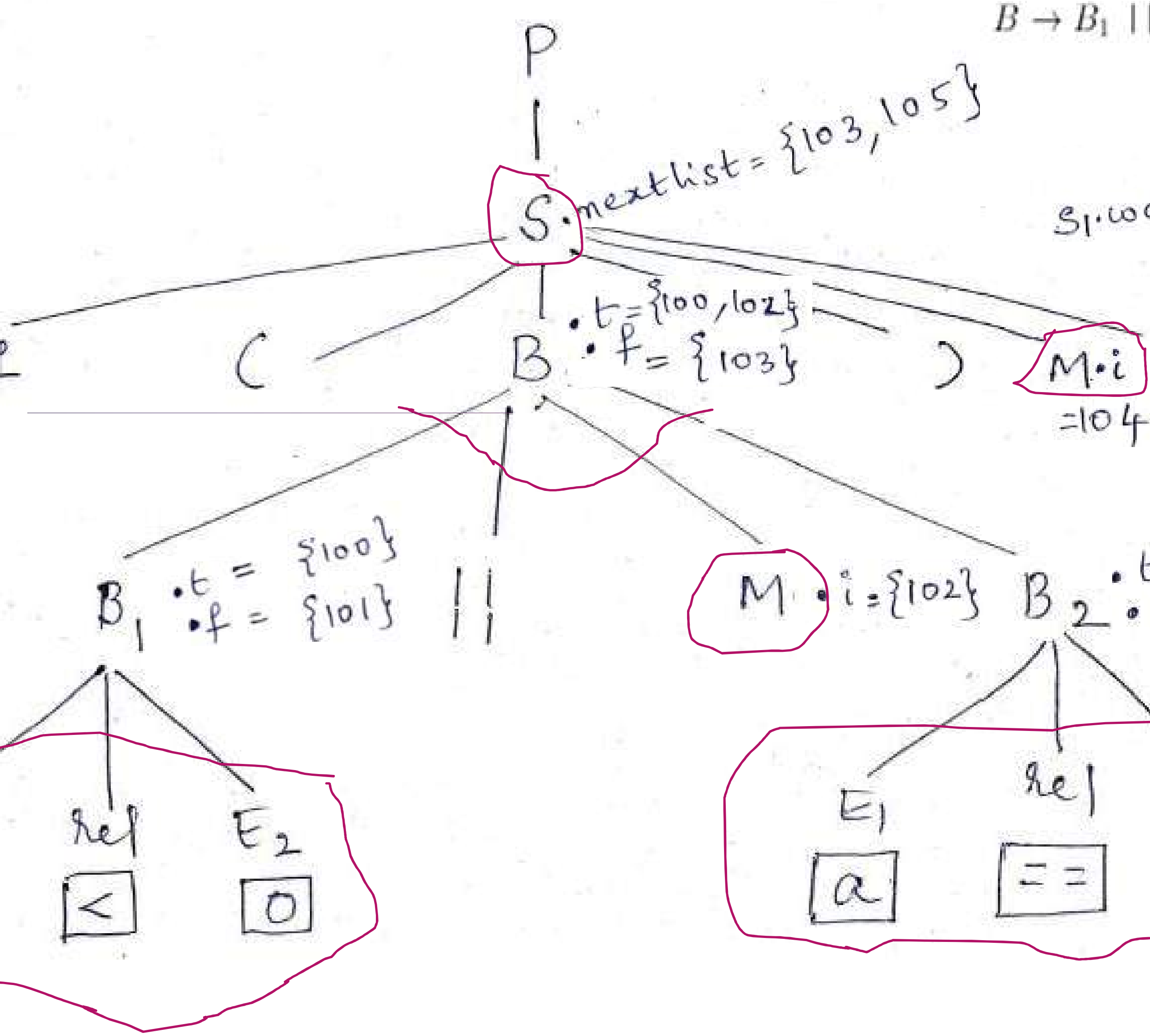
```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 10
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instr

$(a<0 \ || \ a==0) \quad a=1$

$S \rightarrow \mathbf{if}\,(\,B\,)\;M\;S_1\ \{\ backpatch(B.truelist,\ \ M.instr);$
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlis$

$B \rightarrow B_1\ ||\ M\ B_2\quad \{\ backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.t$
$B.falselist = B_2.falselist;\ \}$



S_1.code in 104

$\boxed{a=1}$

S_1. nextlist = 105

100: if a<0 goto
101: goto 102
102: if a==0 g
103: goto 105
104: a=1
105: ___

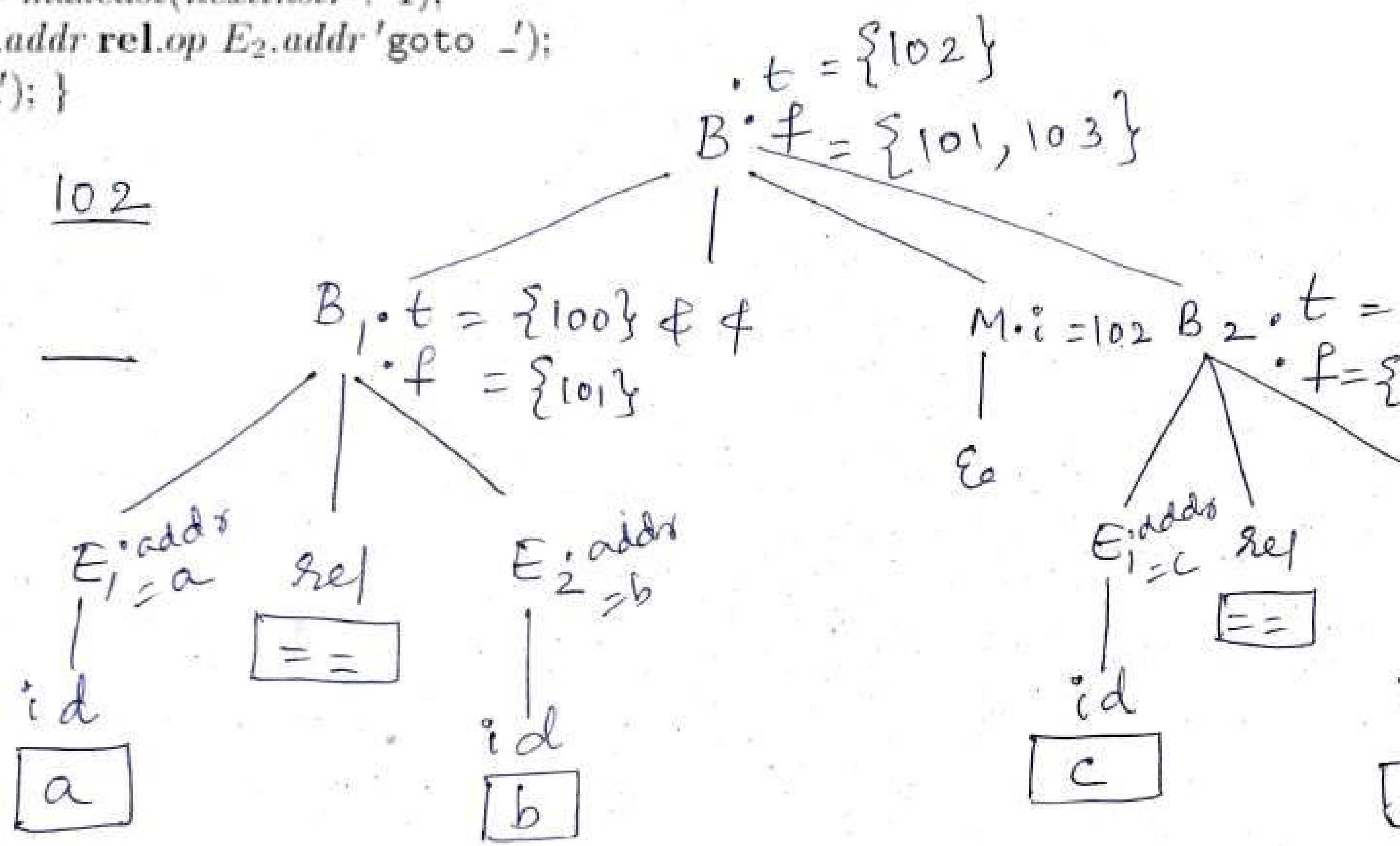
anslation of $a==b \;\&\&\; c==d$

$B \rightarrow B_1 \;\&\&\; M \; B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

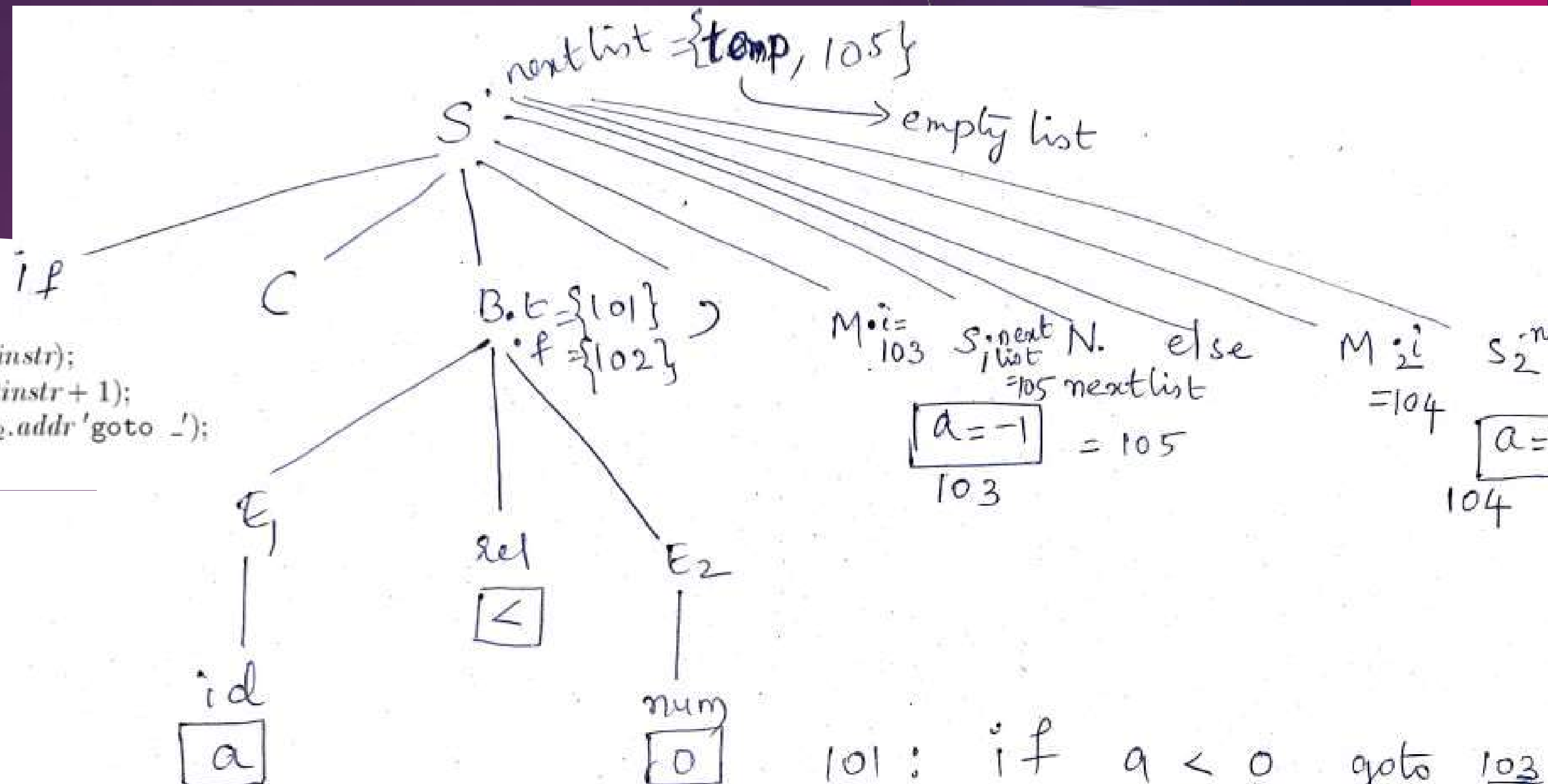
$B \rightarrow E_1 \; \mathbf{rel} \; E_2$ { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$gen($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto _'$)$;
$gen($'goto _'$)$; }

: if a==b goto 102
: goto ___
: if c==d goto ___
: goto ___

e Address Code



else statement



{ *B.truelist* = *makelist*(*nextinstr*);
B.falselist = *makelist*(*nextinstr* + 1);
gen('if' E_1.*addr* **rel**.*op* E_2.*addr* 'goto _');
gen('goto _'); }

101 : if a < 0 goto 103
102: goto 104
103 : a = -1
104 : a = 1
105 : ___

if (B) M_1 S_1 N **else** M_2 S_2

{ *backpatch*(B.*truelist*, M_1.*instr*);
backpatch(B.*falselist*, M_2.*instr*);
temp = *merge*(S_1.*nextlist*, N.*nextlist*);
S.*nextlist* = *merge*(*temp*, S_2.*nextlist*); }

Break, Continue, Goto statements

- If S is the enclosing construct, then a break statement is a jump to the first instruction after the code for S.

- We can generate code for break by

  (1)Keeping track of the enclosing statement S

  (2)Generating an unfilled jump for the break-statement, and

  (3) Putting this unfilled jump on S.nextlist

Switch-statement syntax

$$
\begin{array}{l}
\textbf{switch} \ (\ E \ ) \ \{ \\
\qquad \textbf{case} \ V_1\text{:} \ S_1 \\
\qquad \textbf{case} \ V_2\text{:} \ S_2 \\
\qquad \qquad \cdots \\
\qquad \textbf{case} \ V_{n-1}\text{:} \ S_{n-1} \\
\qquad \textbf{default}\text{:} \ S_n \\
\}
\end{array}
$$

The intended translation of a switch is code to:

1. Evaluate the expression E.
2. Find the value V_j in the list of cases that is the same as the value of expression. Recall that the default value matches the expression if nc of the values explicitly mentioned in cases does.
3. Execute the statement S_j associated with the value found.

Implementation of case statements

- Use a table and a loop to find the address to jump.
- Hash Table: If the number of values exceeds 10 or so, it is more efficient to construct a hash table for the values, with the labels of the various statements as entries. If no entry for the value possessed by the switch expression is found, a jump to the default statement is generated.
- Do Backpatching to generate a series of branching statements with the targets of the label left unspecified. To-be determined label table can be used for this purpose.

yntax-Directed Translation of Switch-tatements

```
witch ( E ) {
    case V1: S1
    case V2: S2
    ...
    case Vn-1: Sn-1
    default: Sn
```

Syntax

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

other translation of a switch statement

```
witch ( E ) {
     case V1: S1
     case V2: S2
          ...
     case Vn-1: Sn-1
     default: Sn
```

Syntax

	code to evaluate E into t
	if t != V_1 goto L_1
	code for S_1
	goto next
L_1:	if t != V_2 goto L_2
	code for S_2
	goto next
L_2:	
	...
L_{n-2}:	if t != V_{n-1} goto L_{n-1}
	code for S_{n-1}
	goto next
L_{n-1}:	code for S_n
next:	

ase three-address-code instructions used translate a switch statement

Reading the queue of value-label pairs, we can generate a sequence of three-address statements of the form

case t V_1 L_1
case t V_2 L_2
...
case t V_{n-1} L_{n-1}
case t t L_n
next:

There, t is the temporary holding the value of the selector expression E, and L_n is the label for the default statement.

The **case t Vi Li** instruction is a synonym for **if t=Vi goto Li**

Intermediate code for procedures

Assume that the parameters are passed by value.
Suppose that a is an array of integers, and that f is a function from integers to integers. Then, the assignment

```
n = f(a[i]);
```

might translate into the following three-address code:

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Intermediate code for procedures

$$
\begin{array}{lcl}
D & \rightarrow & \textbf{define } T \textbf{ id } (F) \{ S \} \\
F & \rightarrow & \epsilon \mid T \textbf{ id } , F \\
S & \rightarrow & \textbf{return } E \; ; \\
E & \rightarrow & \textbf{id } (A) \\
A & \rightarrow & \epsilon \mid E , A
\end{array}
$$

Symbol tables:
Let s be the top symbol table when the function definition is reached. The function name is entered into s for use in the rest of the program.

In the production for D, after seeing define and the function name, we push s and set up a new symbol table

Env.push(top); top = new Env(top);
Call the new symbol table, t. Note that top is passed as a parameter in new Env(top), so the new symbol table t can be linked to the previous one, s. The new table t is used to translate the function body.

Translate these statements

- 1. f=min(1,n-1,n+1)

Three address code:

```
t1=n-1
t2=n+1
param 1
param t1
param t2
t3= call min , 3
f = t3
```

- 2. switch(a + b){

    case 1: a=b;

    case 0: b=a;

    default: a=0;

  }

Three address code:

```
t1= a + b
If t1!= 1 goto L1
a=b
goto next
L1: if t1 != 0 goto L2
b=a
goto next
L2: a=0
next:
```