Unit -5: Intermediate code generation and Code Genaration

Syllabus:

Intermediate Code Generation :

5.1: Introduction

- Variants of syntax trees -;
- Directed Acyclic graphs for expressions
- Syntax Directed Definitions to construct DAGs
- Value number method for constructing DAGS

5.2: Three-address code, Translation of expressions Control Flow statements

5.3: Code generation Algorithm

Self- Learning : Control flow : Translation of Boolean exprssion;

B-Intermediate Code Generation :

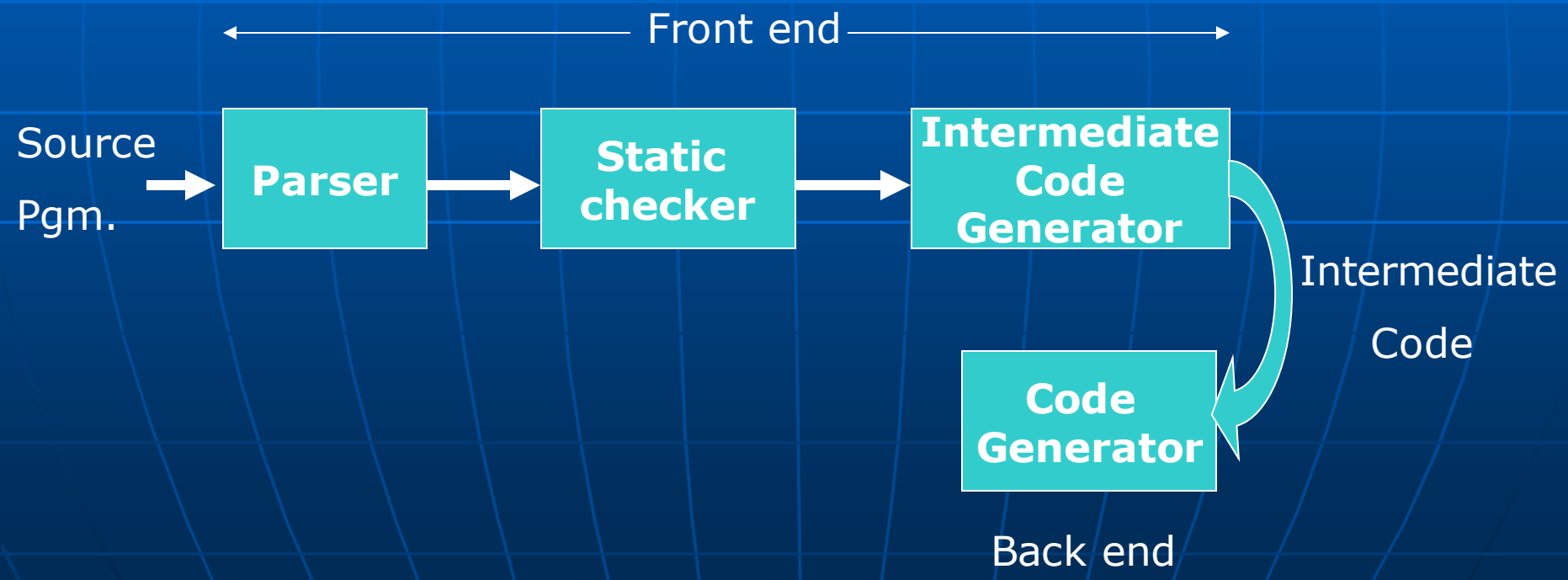
Dr. M.M.Math
Professor, CSE
GIT Belgaum

Agenda

- Introduction
- Benefits of Intermediate codes
- Different kinds of Intermediate codes
- Examples of writing Intermediate codes namely DAG, Three Address Codes
- Syntax Directed Translation for Intermediate code Generation.

We assume that a compiler front end is organized as in the figure shown below

Here parsing, static checking, and intermediate-code generation are done sequentially.



Benefits of Intermediate codes

1. Intermediate code is closer to the target machine than the source language, and hence easier to generate the code form
2. Unlike machine language, intermediate code is machine independent. This makes it easier to retarget compiler optimization.
3. It allows a variety of machine independent optimizations to be performed.
4. Typically, intermediate code generation can be implemented via SDTs and thus can be folded into parsing and Type checking.

Introduction

- Intermediate code is a form of representation of source statement that is very close to machine instructions and independent of machine code
- Analysis-synthesis model:
  - Front end analyses a source code and creates an intermediate representation
  - From this intermediate representation the back end generates the object code
  - The front end is program dependent and the back end is machine dependent

Different kinds of Intermediate codes

- There are Two kinds of intermediate forms namely

1. High level Intermediate representation

These representations expresses high level structure of the program which is closer to source program and can be generated easily form the input program

Example : Abstract syntax tree and DAG

2. Low level Intermediate representation

These representations expresses low level structure of the program which is closer to M/c program and generations may involve some work

Example : Three address code, P-code, Diana and Byte code

Varients of Syntax Tree

Directed Acyclic Graphs (DAG)

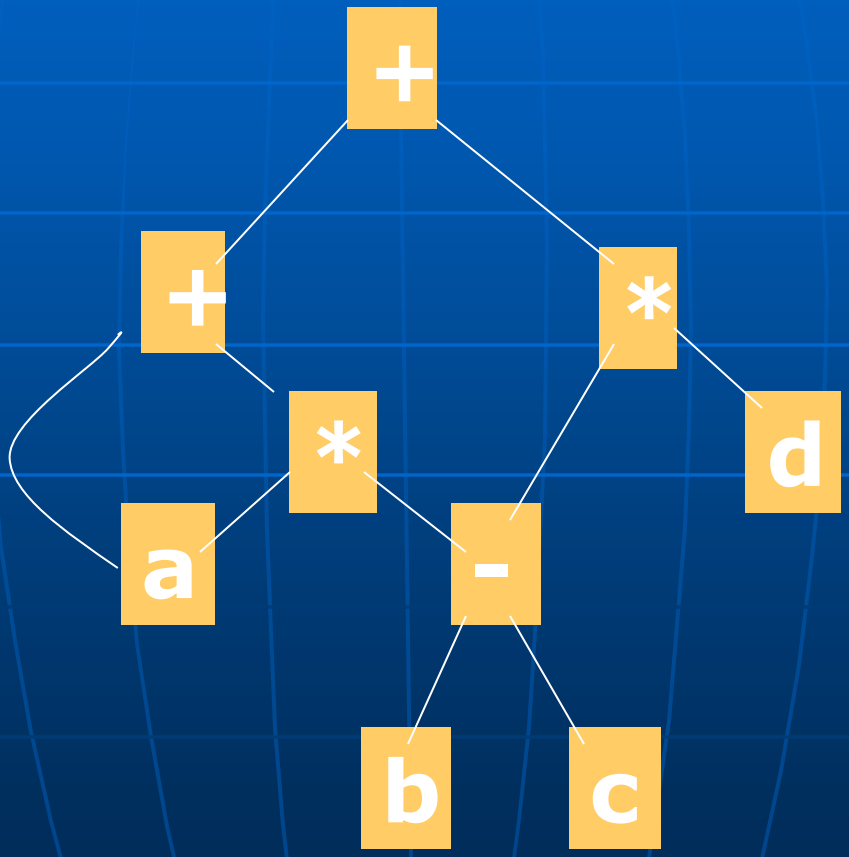
- Leaves correspond to atomic operands
- Interior nodes correspond to operators
- A node N in a DAG can have more than one parent if N represents a common subexpression

Advantages:

- Represents expressions more succinctly
- Gives the compiler more clues for generation of efficient code

Directed Acyclic Graphs for Expressions

```
a + a * ( b – c ) + ( b – c ) * d
```



Examples :

1. a+b+(a+b)
2. a+b+a+b
3. (a*b)+(c+d)*(a*b) + b
4. ((x+y)-((x+y)*(x-y))) + ((x+y)*(x-y))
5. a+a+((a+a+a+(a+a+a+a))

Constructing a DAG
(Refer SDD for Syntax Tree)

- A syntax directed definition is used to construct a DAG
- The steps are similar to the construction of syntax trees
- But before creating a new node we need to check whether an identical node already exists
- If such a node exists then a pointer to the existing node is returned
- else a new node is created and pointer to a newly created node is returned..

SDD for constructing DAG for arithamatic Exp

1. Like Syntax tree, Each node in the DAG is implemented as a record with several fields. One field identifies the operator and remaining fields contain pointers to the nodes for the operands.
2. We assume three functions to create the nodes for the DAG for expressions with binary operators.

   a) ***mknode(op,left,right)*** – This creates an operator node with label 'op' and two fields containing pointers to left and right child

   b)***mkleaf(id, id.entry)*** - This creates a leaf node with label id and a field containing entry, a pointer to the symbol entry to the identifier.

   c) ***mkleaf(digit, digit.val)*** - This creates a leaf node with label digit and a field containing val, the value of the number

note : The mkleaf() or mknode() functions , before creating a new node it needs to check whether an identical node already exists, If such a node exists then only the pointer to existing node is returned otherwise a new node is created and pointer to the created node is returned.

SDD for Directed Acyclic Graph

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 + T$	E.*nptr* = *mknode*("+",E_1.*nptr* ,T.*nptr*)
$E \rightarrow E_1 - T$	E.*nptr* = *mknode*("-",E_1.*nptr* ,T.*nptr*)
$E \rightarrow T$	E.*nptr* = T.*nptr*
$T \rightarrow T * F$	T.*nptr* = *mknode*("*",T_1.*nptr* ,F.*nptr*)
$T \rightarrow T / F$	T.*nptr* = *mknode*("/",T_1.*nptr* ,F.*nptr*)
$T \rightarrow F$	T.*nptr* = F.*nptr*
$F \rightarrow (E)$	F.*nptr* = E.*nptr*
$F \rightarrow id$	F.*nptr* = *mkleaf*(id, id.*entry*)
$F \rightarrow digit$	F.*nptr* = *mkleaf*(digit, digit.*val*)

Representation of DAG by value-number method

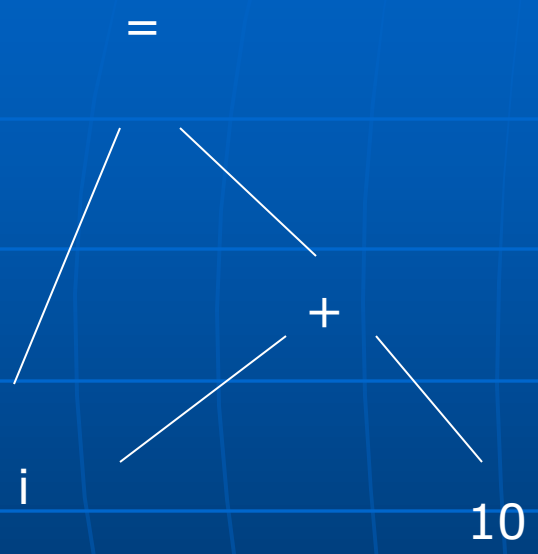
- Nodes of a DAG can be stored as an array of records. Each row of the array represents a node
- In each record the first field represents an operation code
- For leaves one additional field holds the lexical value
- For interior nodes there are two additional fields for left and right children
- We refer to each node with integer index of the array called the value number

Algorithm for value-number method

Suppose that nodes are stored in an array and each node is referred to by its value number.

- **Input**: label op,node l and node r
- **Output**:the value of node in the array with signature <op, l, r>
- **Method**:search the array for the node with label op,left child l & right child r.If found return its value number.If not found,we create in the array a new node with label op left child l and right child r & return its value number

DAG Construction
i = i + 10



1	id	To entry for I	
2	num	10	
3	+	1	2
4	=	1	3

Construct the DAG for the expression

- a+b+(a+b)
- a+b+a+b
- ((x+y)-((x+y)*(x-y))) + ((x+y)*(x-y))
- a+a+((a+a+a+(a+a+a+a))

Three-address code

- Three address code is built from two concepts: addresses and instructions
- In three-address code, there is at most one operator on the right side of an instuction
- If more than one operator is to be used then they are simplified

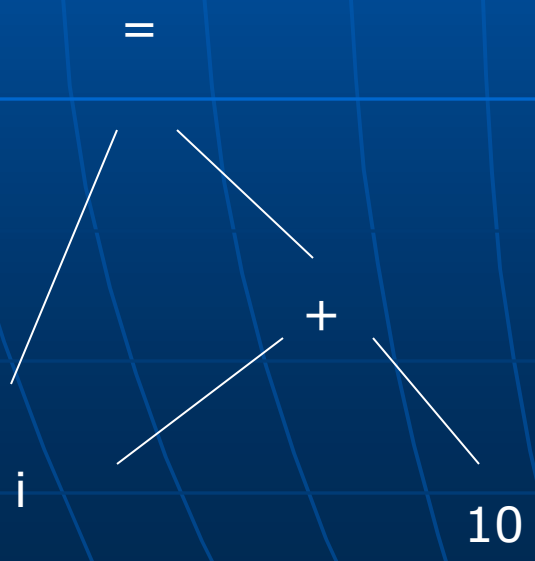
Eg.

X+y*z can be written as

T1=y*z

T2=x+T1

- Invariably the three address code is a linearized representation of syntax tree or DAG in which explicit names are given to the corresponding interior nodes of the graph.
- Example :
- DAG



Three Address code

t1 = i + 10

i = t1

Addresses

Address are of three types

- A name: source program names can appear as addresses in three adderss code
- A constant: compiler must be able to deal with different types of constants
- A compiler generated temporary are used as addresses

Three address instructions

- Assignment instructions: x=y op z;
- Assignments of the form: x=op y;
- Copy instructions: x=u;
- An unconditional jump: goto L
- Conditional jumps(1): if x goto L
- Conditional jumps(2): if x relop y goto L
- Conditional jumps(3): ifFalse x goto L
- For procedure calls and returns : param x for parameters; call p,n and y=call p,n for procedure calls and function calls; return y where y is return value (Optional)
- Indexed copy instructions : x=y[i] and x[i]=y
- Address and pointer assignments: x=&y, x=*y,*x=y

Examples

1. X=a*(b-c)+(b-c)*d
2. A=b*-c+b*-c
3. If (x>y) then z=x+y;
   Else z=x-y;
4. While (x<y) Do
   begin
   if c>d x=x+y
   else x=x-y
   End
5. Do i=i+1; while(a[i]<v);
6. If (x<100 || x >200 && x !=y) x=0;
7. (a>b) && ((b>c) || a>c))

Example -1

- X=a*(b-c)+(b-c)*d
- Three address code statements

  t1=b-c

  t2=a*t1

  t3=b-c

  t4=t3*d

  t5=t2+t4

  X= t5

Example - 3

3. If (x>y) then z=x+y;
   Else z=x-y;

Three address code statements :

```
    t1=x>y
    if t1 goto L1
    t2=x-y
    z=t2
    goto L2
L1: t2=x+y
    z=t2
L2: ---
    ---
```

Example - 4

```
While (x<y) Do
    begin
        if c>d x=x+y
        else x=x-y
    End
```

Three address codes:

```
L1 :if x<y goto L2
       goto L4
L2 : if c > d goto L3
        t1=x-y
         x=t1
          goto L1
L3 : t1 = x+y
        x=t1
        goto L1
L4 : ---
        ---
```

Example - 5

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

```
L1 : t1=i+1
     i=t1
     t2=i*8
     t3=a[t2]
     if t3 < v goto L1 L2 : -
--
     ---
```

TAC with Symbolic Labels

```
10 : t1=i+1
11 : i=t1
12 :  t2=i*8
13 : t3=a[t2]
14 : if t3 < v goto 10
15 : ---
     ---
```

TAC with Position Numbers

For above construct two types of codes can be genarated. Three address code viz with Symbolic labels or with position numbers. In case of array reference first array index value is calculated in some temporary by considering space occupied by each element which is either 4 or 8 in case of integer elements or 16 units in case of float

Example -6

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

  Three address code statements :

  Refer Page No : 400 figure6.34 for solution

Implementation of Three Address code -TAC

- The description of three address code specifies the components of each type of instructions but it does not specifies the representation of these instructions in a data structure.
- In the compiler implementation these instructions are implemented by Quadruple, triple and indirect triples

Quadruples

- Quadruples are used to implement the three address instructions in compilers. Here Each quadruple is implemented as a object or record containing four fields: op,arg1,arg2 & result.

For Instance three address instruction t= x + y is represented by placing '+' is in OP field , x is in arg1, y is in arg2 and t is in result.

Exceptions are:

- Instructions with unary operators Eg x= -y
  do not use arg2
- Conditional & unconditional jumps put the target label in result.
- Operators like param use neither arg2 nor result

Example on Quadruple representation of TAC a=b* -c + b* -c

position	Op	Arg1	Arg2	result
1	U-minus	c		t1
2	*	b	t1	t2
3	U-minus	c		t3
4	*	b	t3	t4
5	+	t2	t4	t5
6	=	t3		a
7				

Quadruple representation if x > y goto L and if x goto L

	Op	Arg1	Arg2	result
1	>	x	y	t1
2	goto	t1		L

	Op	Arg1	Arg2	result
2	goto	x		L

Quadruple representation x[i]=y and x =y[i]

	op	Arg1	arg2	result
0	[]=	x	i	t1
1	assign	y		t1

	op	Arg1	arg2	result
0	[]=	y	i	t1
1	assign	t1		x

TRIPLES

- They are also used in the implemention of three adress instructions but use only three fields. The result field is missing here
- Using the triples we refer to the result of an operation by its position rather than by a temporary name.
- When instructions are moved around we need to change all references to that result

Triple representation
b*-c + b*-c

position	op	Arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Triple representation
if x > y goto L and if x goto L

position	op	Arg1	arg2
0	>	x	y
1	goto	(0)	L

position	op	Arg1	arg2
0	goto	x	L

Triple representation x[i]=y and x =y[i]

position	op	Arg1	arg2
0	[]=	x	i
1	assign	(0)	y

Position	Op	Arg1	Arg2
0	[]=	Y	I
1	Assign	x	(0)

Indirect Triples

- They consist of listing of pointers to tripples.
- Here we can move an instruction by reordering the instruction list without affecting the tripples themselves.

Indirect Triples representation
b*-c + b*-c

Instructions	
30	(0)
31	(1)
32	(2)
33	(3)
34	(4)
35	(5)

position	op	Arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Translation of an assignment statement

- **S→id = E, E→ E + E | E*E | (E) | -E | id**

1. S stands for assignment statement made up of two operators and all the operands denote primitive data type namely integer.
2. Intermediate code that is to be generated is sequence of Three address code statements.
3. Each Grammar symbol has a synthesize attribute. Hence synthesize attribute S.code represents a sequence of three codes for the assignment S.

- Example :
- X=a*(b-c)+(b-c)*d
- Three address code statements

t1=b-c

t2=a*t1

t3=b-c

t4=t3*d

t5=t2+t4

X= t5

Attributes of Grammar Symbols

- The Nonterminal E has two attributes namely,
  1. E.code - denotes the sequence of three address codes evaluating E.
  2. E.addr – denotes the address that hold the value of sequence of TAC represented by E.code Here the address can be name or compiler generated temporary

S→id = E If this production is used for reduction then we must have sequence of TAC (E.code) to evaluate E into some temporary and then semantic rules will generate the TAC X=E.addrs where x is the name corresponds to id and finally it has to concatenate the sequence of TAC for E.code with the TAC generated which will be S.code.

S→id = E S.code =E.code || gen(top.get(id.lexme) '=' E.addr)

E → E1 + E2

In this case we must have sequence of TAC (E1.code and E2.code) to evaluate E1 and E2 into some temp(E1.addr and E2.addr).

First we must have a semantic rule that must request some temporary for E.addr to store the value of E.code and secondly Semantic Rule must generate the new TAC E.addr=E1.addr + E2.addr and concatenate with E1.code and E2.code which will be the TAC for E

E → E1 + E2

E.addr = new Temp()

E.code = E1.code || E2.code || Gen(E.addr '=' E1.addr '+' E2.addr)

E→ID

In this case the expression is a single identifier say x and x itself holds the value of the expression. The semantic rule for this production must define E.addr to point to the symbol table entry for some instance of ID.

E→ID E.addr = top.get(id.lexme)

E.code =" "

Top denote the current symbol table and top.get retrieves entry form the symbol table

Translation of expressions

Production	Semantic rules
S→id = E	S.CODE = E.CODE|| GEN(TOP.GET(ID.LEXEME) '=' E.ADDR))
E → E1 + E2	E.ADDR = NEW TEMP() E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '=' E1.ADDR '+' E2.ADDR)
E → E1 * E2	E.ADDR = NEW TEMP() E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '=' E1.ADDR '*' E2.ADDR)
| -E1	E.ADDR = NEW TEMP() E.CODE = E1.CODE || GEN(E.ADDR '=' 'MINUS' E1.ADDR)
| (E1)	E.ADDR = E1.ADDR E.CODE = E1.CODE
| ID	E.ADDR = TOP.GET(ID.LEXEME) E.CODE = ""

Translation of mixed type operation in an Expression

1. Expression may involve different types of variables and constants, so the compiler must either reject the mixed type operations or generate appropriate type conversion statements.
2. Let us consider the assignment statement with two types namely float and integer
3. For example: `x= y + i * j`

   assume x and y have type float and i and j have type integer

- Sequence of three address code could be generated as follows.

```
t1= i int* j
t2 = inttofloat t1
t3 = y real+ t2
 x= t3
```

In addition to two attributes E.code and E.addr of grammar symbol E we have another attribute called E.type that represents type of operation

The semantic rules must check the type and generate the appropriate Three address code.

```
E → E1 + E2
        E.ADDR = NEW TEMP();
        if E1.type = integer and E2.type = integer then
        Begin
        E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '='
                    E1.ADDR  int '+' E2.ADDR)
        E.type = integer
        end
        else if E1.type = float and E2.type = float then
        Begin
        E.CODE = E1.CODE || E2.CODE || GEN(E.ADDR '='
                    E1.ADDR  float '+' E2.ADDR)
        E.type = float
        end
```

```
else if E1.type = integer and E2.type = float then
        Begin
         u=newtemp();
        E.CODE = E1.CODE || E2.CODE || u '=' inttofloat  E1.addr ||
                    GEN(E.ADDR '='  E1.ADDR  float '+' E2.ADDR)
        E.type = float
        end
else if E1.type = float and E2.type = integer then
        Begin
         u=newtemp();
        E.CODE = E1.CODE || E2.CODE || u'='inttofloat E2.addr
        GEN(E.ADDR '='  E1.ADDR  real '+' E2.ADDR)
        E.type = float
        end
Else E.type = type_error
```

Flow of control statements

Consider the following statements:

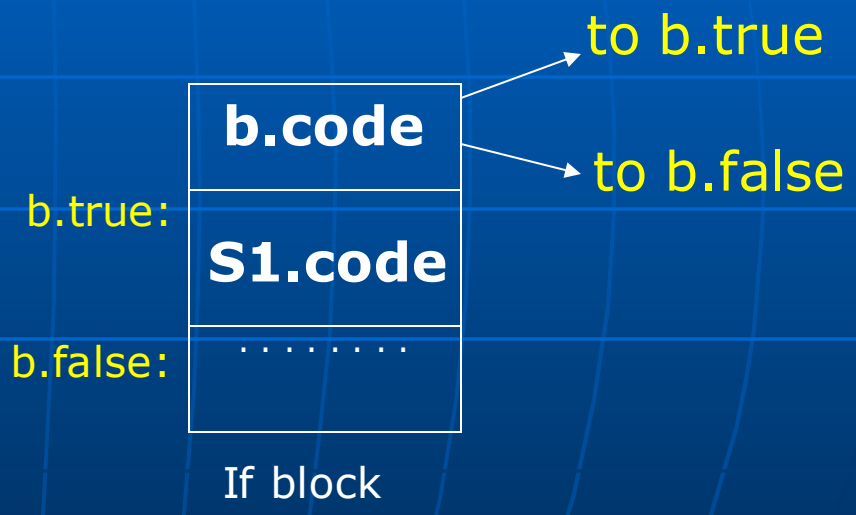
S->if (b) s1

where s represents statements and b represents Boolean expressions.

The translation of this statement consists of b.code followed by s1.code as shown.

Based on the values of b, there are jumps within b.code.

Similar are the other flow control statements



- With boolean expression B we associate two label (attributes) namely

  B.true : The label for first statement of TAC to which control flows if B is true.

  B.false : The label for first statement of TAC to which control flows if B is false.

  we also have the following attributes

  B.code : sequence of three address codes evaluating Boolean expression.

- With statement S S1, S2 (any statement) we associate label (attributes) and string attribute namely

  S.next : The label to Three address code statement that follows three address code statement for S

  S1.next and S2.next : The label to three address code statement that follows three address code statement for S1 and S2

  S.code : sequence of three address codes of statement S.

  S1.code and S2.code : sequence of three address codes

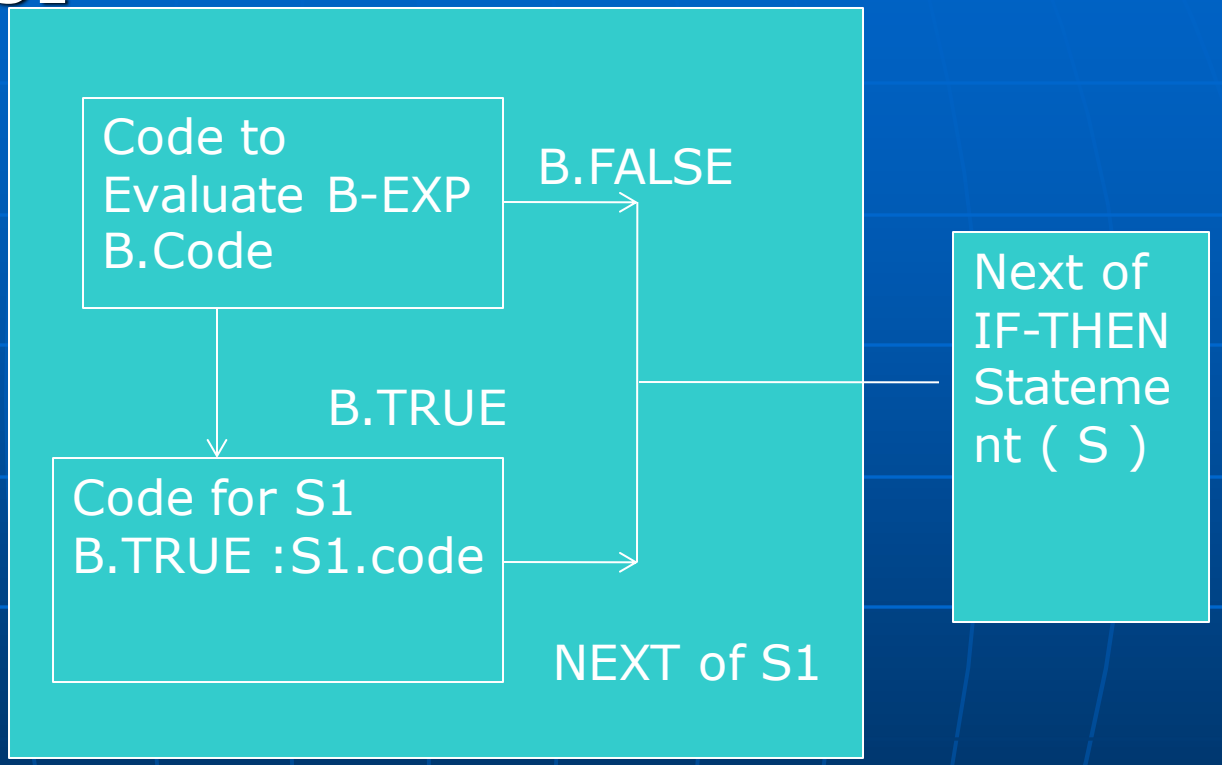
Semantic rule for boolean expression

B → E1 rel E2 B.code = E1.code || E2.CODE ||
GEN(‘if’ E1.addr rel.op E2.addr ‘goto’ B.true
|| GEN (‘goto’ B.false)

Example : if a<b goto B.true
goto B.false

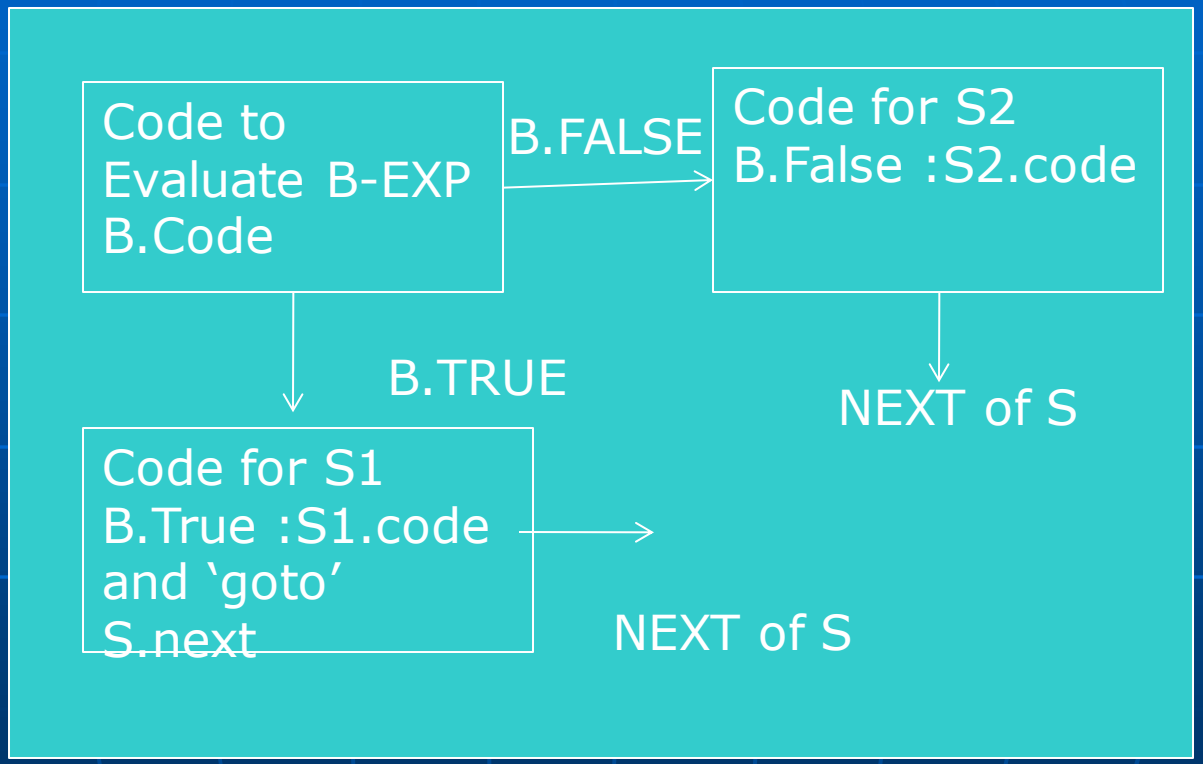
- Newlabel() – It creates a newlabel each time it is called
- Label(L)- It attaches label L to the next three address code statement

- S→IF (B) S1



B.true=newlabel()
B.FALSE=S1.next=S.next
S.code= B.code || label(B.true) || s1.code

- If (B) S1 else S2

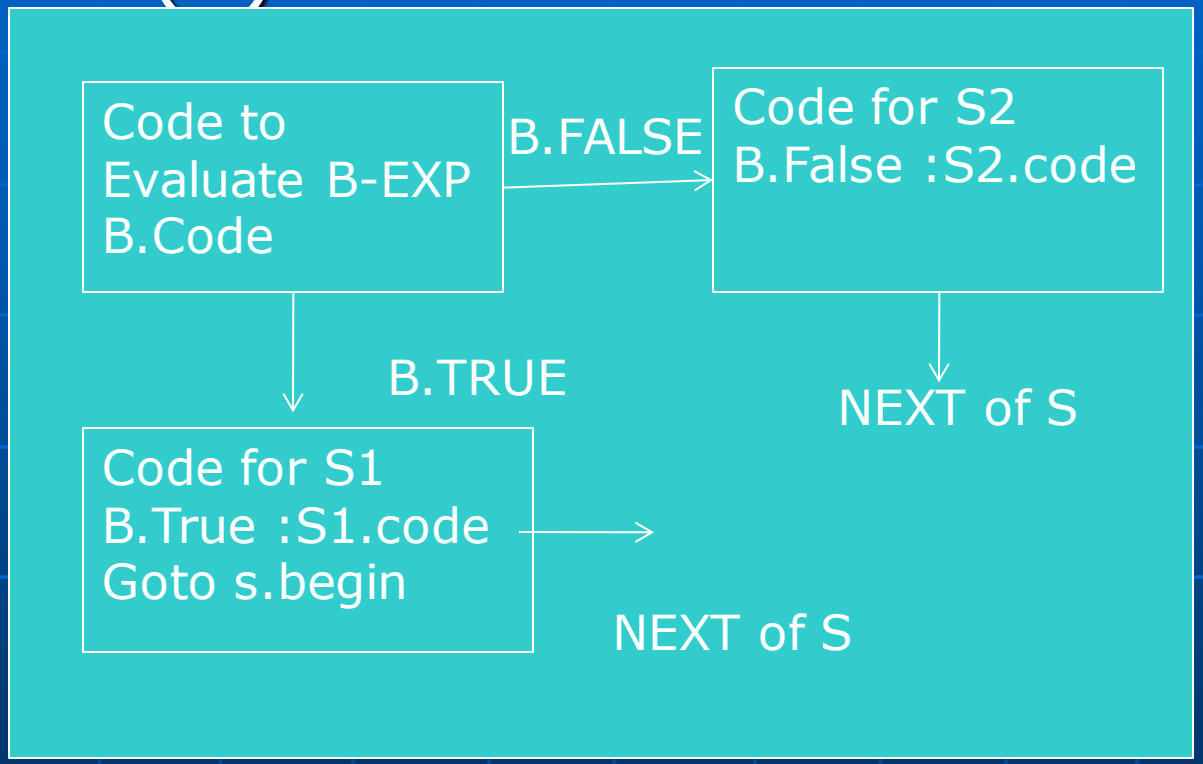


B.true=newlabel()
B.False=newlabel()

S2.next=S1.next=S.next

S.code= B.code || label(B.true) || s1.code || Gen(‘goto’ s.next) ||
label(B.False) || s2.code

- S→ while (b) Do s1



S->If (b) s1

b.code	
b.true:	s1.code
s.next:	……

If (b) then s1 else s2

	b.code
b.true:	s1.code
	goto s.next
b.false:	s2.code
s.next:	……

- S-> while (b) Do s1

s.begin :	b.code
b.true :	s1.code
	goto s.begin
b.false:	……..

The semantic rule for translating if control statement must first contain sequence of TAC's of Boolean expression(b.code) followed by sequence TAC's of statement S1 with label b.true for the first statement

b.true = newlabel

b.false = s1.next = s.next

i.e s.code = b.code || label b.true || s1.code

Sdd for some control statements

production	Semantic rules
S->if(b) s1	b.true= newlabel() b.false=s1.next=s.next s.code=b.code||label(b.true ‘ : ‘) ||s1.code
S->if(b)s1 else s2	b.true=newlabel() b.false=newlabel() S1.next=s2.next=s.next s.code=b.code || label(b.true ‘ : ‘)|| s1.code ||gen(‘goto’ s.next) || label(b.false ‘ : ‘)|| s2.code

Sdd for some control statements

contd

production	Semantic rules
S->while(b) s1	s.begin=newlabel() b.true=newlabel() b.false=s.next s1.next=s.begin s.code=label(s.begin :) || b.code || label (b.true ` : `)||s1.code || gen(‘goto’ s.begin)

B → E1 rel E2 B.code = E1.code || E2.CODE ||
GEN(‘if’ E1.addr rel.op E2.addr ‘goto’ B.true
|| GEN (‘goto’ B.false)

Example : if a<b goto B.true
goto B.false

Switch Statements

- There is a selector expression which needs to be evaluated followed by a set of values that it can take.
- The expression is evaluated and depending on the value generated particular set of statements are executed
- There is always a set of default statements which is executed if no other value matches the expression

Translation of a switch statement

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

Translation of switch-statement

- If the number of cases is small say 10 then we use a sequence of conditional jumps
- If the number of values exceeds 10 it is more efficient to construct a hash table for the values with labels of the various statements as entries.

Intermediate code procedures

In three address code a function call is unraveled into the evaluation of parameters in preparation of a call followed by the call itself.

the statement: n=f(a[i]); is translated to:

```
1)    t1=i * 4
2)    t2=a[t1]
3)    param t2    /* makes t2 an actual parameter */
4)    t3=call f,1
5)    n=t3
```