Chapter 3

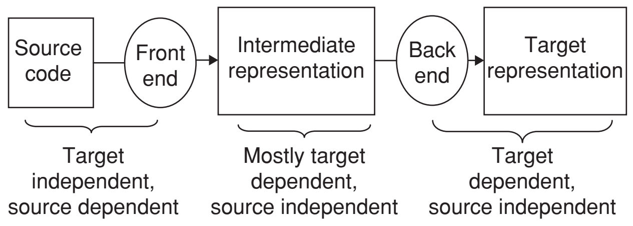
Intermediate Code Generation

LEARNING OBJECTIVES

- ☞ *Introduction*
- ☞ *Directed Acyclic Graphs (DAG)*
- ☞ *Three address code*
- ☞ *Symbol table operations*
- ☞ *Assignment statements*
- ☞ *Boolean expression*
- ☞ *Flow control of statements*
- ☞ *Procedure calls*
- ☞ *Code generation*
- ☞ *Next use information*
- ☞ *Run-time storage management*
- ☞ *DAG representations of basic blocks*
- ☞ *Peephole optimization*

INTRODUCTION

In the analysis–synthesis model, the front end translates a source program into an intermediate representation (IR). From IR the back end generates target code.


Source code
Front end
Intermediate representation
Back end
Target representation
Target independent, source dependent
Mostly target dependent, source independent
Target dependent, source independent

There are different types of intermediate representations:

- High level IR, i.e., AST (Abstract Syntax Tree)
- Medium level IR, i.e., Three address code
- Low level IR, i.e., DAG (Directed Acyclic Graph)
- Postfix Notation (Reverse Polish Notation, RPN).

In the previous sections already we have discussed about AST and RPN.

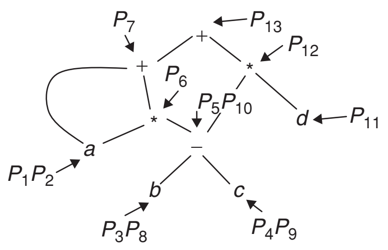
Benefits of Intermediate code generation: The benefits of ICG are

1. We can obtain an optimized code.
2. Compilers can be created for the different machines by attaching different backend to existing front end of each machine.
3. Compilers can be created for the different source languages.

Directed acyclic graphs for expression: (DAG)

- A DAG for an expression identifies the common sub expressions in the given expression.
- A node N in a DAG has more than one parent if N represents a common sub expression.
- DAG gives the compiler, important clues regarding the generation of efficient code to evaluate the expressions.

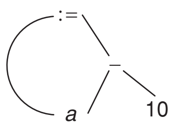
Example 1: DAG for $a + a*(b - c) + (b - c)*d$



P_1 = makeleaf (id, a)
P_2 = makeleaf (id, a) = P_1
P_3 = makeleaf (id, b)
P_4 = makeleaf (id, c)
P_5 = makenode (–, P_3, P_4)
P_6 = makenode (*, P_1, P_5)
P_7 = makenode (+, P_1, P_6)
P_8 = makeleaf (id, b) = P_3
P_9 = makeleaf (id, c) = P_4
P_{10} = makenode (–, P_8, P_9) = P_5

P_{11} = makeleaf (id, d)
P_{12} = makenode (*, P_{10}, P_{11})
P_{13} = makenode (+, P_7, P_{12})

Example 2: $a: = a - 10$



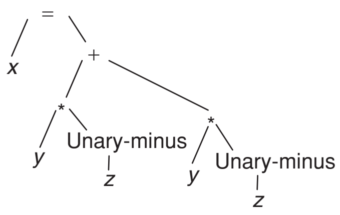
Three-Address Code

In three address codes, each statement usually contains 3 addresses, 2 for operands and 1 for the result.

Example: $-x = y$ OP z

- x, y, z are names, constants or complier generated temporaries,
- OP stands for any operator. Any arithmetic operator (or) Logical operator.

Example: Consider the statement $x = y * -z + y* -z$



The corresponding three address code will be like this:

Syntax Tree	DAG
$t_1 = -z$	$t_1 = -z$
$t_2 = y * t_1$	$t_2 = y * t_1$
$t_3 = -z$	$t_5 = t_2 + t_2$
$t_4 = y * t_3$	$X = t_5$
$t_5 = t_4 + t_2$	
$X = t_5$	

The postfix notation for syntax tree is: xyz unaryminus *yz unaryminus *+=.

- Three address code is a 'Linearized representation' of syntax tree.
- Basic data of all variables can be formulated as syntax directed translation. Add attributes whenever necessary.

Example: Consider below *SDD* with following specifications:
E might have *E*. place and E.code
E.place: the name that holds the value of *E*.
E.code: the sequence of intermediate code starts evaluating *E*.
Let Newtemp: returns a new temporary variable each time it is called.
New label: returns a new label.
Then the SDD to produce three–address code for expressions is given below:

Production	Semantic Rules
$S\rightarrow$ id ASN E	S. code = E.code \\ gen (ASN, id.place, *E*.place) E. Place = newtemp ();

$E\rightarrow E_1$ PLUS E_2	E. code = E_1. code || E_2. code || gen (PLUS, *E*. place, E_1. place, E_2. place);
	E. place = newtemp();

$E\rightarrow E_1$MUL E_2	E. code = E_1. code || E_2.code || gen (MUL, E. place, E_1. place, E_2. place);
	E. Place = Newtemp();

$E\rightarrow$ UMINUS E_1	E. code = E_1 code || gen (NEG, *E*. Place, E_1. place); E. code = E_1.code
$E\rightarrow$ LP E_1 RP	E. Place = E_1. Place

$E\rightarrow$ IDENT	E.place = id. place
	E. code = empty.list ();

Types of Three Address Statement

Assignment

- Binary assignment: $x: = y$ OP z Store the result of y OP z to x.
- Unary assignment: $x: =$ op y Store the result of unary operation on y to x.

Copy

- Simple Copy $x: = y$ Store y to x
- Indexed Copy $x: = y[i]$ Store the contents of $y[i]$ to x
- $x[i]:= y$ Store y to $(x + i)$th address.

Address and pointer manipulation

$x : = \&y$	Store address of y to x
$x : = *y$	Store the contents of y to x
$*x : = y$	Store y to location pointed by x .

Jump

- Unconditional jump:- goto L, jumps to L.
- Conditional:

```
if (x relop y)
goto L1;
else
```

```
{
goto L2;
}
```

Where relop is <, < =, >, > = , = or ≠.

Procedure call

Param x_1;

Param x_2;

.

.

.

Param x_n;

Call p, n, x;	Call procedure p with n parameters and store the result in x.
return x	Use x as result from procedure.

Declarations

- Global x, n_1, n_2: Declare a global variable named x at offset n_1 having n_2 bytes of space.
- Proc x, n_1, n_2: Declare a procedure x with n_1 bytes of parameter space and n_2 bytes of local variable space.
- Local x, m: Declare a local variable named x at offset m from the procedure frame.
- End: Declare the end of the current procedure.

Adaption for object oriented code

- $x = y$ field z: Lookup field named z within y, store address to x
- Class x, n_1, n_2: declare a class named x with n_1 bytes of class variables and n_2 bytes of class method pointers.
- Field x, n: Declare a field named x at offset n in the class frame.
- New x: Create a new instance of class name x.

Implementation of Three Address Statements

Three address statements can be implemented as records with fields for the operator and the operands. There are 3 types of representations:

1. Quadruples
2. Triples
3. Indirect triples

Quadruples

A quadruple has four fields: op, arg1, arg2 and result.

- Unary operators do not use arg2.
- Param use neither arg2 nor result.
- Jumps put the target label in result.
- The contents of the fields are pointers to the symbol table entries for the names represented by these fields.
- Easier to optimize and move code around.

Example 1: For the expression $x = y * -z + y * -z$, the quadruple representation is

	OP	Arg1	Arg2	Result
(0)	Uminus	z		t_1
(1)	*	y	t_1	t_2
(2)	Uminus	z		t_3
(3)	*	y	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Example 2: Read (x)

	Op	Arg1	Arg2	Result
(0)	Param	x		
(1)	Call	READ	(x)	

Example 3: WRITE ($A*B$, x +5)

	OP	Arg1	Arg2	Result
(0)	*	A	B	t_1
(1)	+	x	5	t_2
(2)	Param	t_1		
(3)	Param	t_2		
(4)	Call	Write	2	

Triples

Triples have three fields: OP, arg1, arg2.

- Temporaries are not used and instead references to instructions are made.
- Triples are also known as two address code.
- Triples takes less space when compared with Quadruples.
- Optimization by moving code around is difficult.
- The DAG and triple representations of expressions are equivalent.
- For the expression $a = y* -z + y*-z$ the Triple representation is

	Op	Arg1	Arg2
(0)	Uminus	z	
(1)	*	y	(0)
(2)	Uminus	z	
(3)	*	y	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Array – references

Example: For A [I]: = B, the quadruple representation is

	Op	Arg1	Arg2	Result
(0)	[ ] =	A	I	T_1
(1)	=	B		T_2

The same can be represented by Triple representation also.

[] = is called L-value, specifies the address to an element.

	Op	Arg1	Arg2
(0)	[ ] =	*A*	*I*
(1)	=	(0)	*B*

Example 2: $A := B[I]$

	Op	Arg1	Arg2
(0)	= [ ]	*B*	*I*
(1)	=	*A*	(0)

= [ ] is called r-value, specifies the value of an element.

Indirect Triples

- In indirect triples, pointers to triples will be there instead of triples.
- Optimization by moving code around is easy.
- Indirect triples takes less space when compared with Quadruples.
- Both indirect triples and Quadruples are almost equally efficient.

Example: Indirect Triple representation of 3-address code

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

	Op	Arg1	Arg2
(14)	Uminus	*z*	
(15)	*	*y*	(14)
(16)	Uminus	*z*	
(17)	*	*y*	(16)
(18)	+	(15)	(17)
(19)	=	*x*	(18)

Symbol Table Operations

Treat symbol tables as objects.

- Mktable (previous);
  - create a new symbol table.
  - Link it to the symbol table previous.
- Enter (table, name, and type, offset)
  - insert a new identifier name with type and offset into table
  - Check for possible duplication.
- Add width (table, width);
  - increase the size of symbol table by width.
- Enterproc (table, name, new table)
  - Enter a procedure name into table.
  - The symbol table of name is new table.
- Lookup (name, table);
  - Check whether name is declared in the symbol table, if it is in the table then return the entry.

Example:

Declaration $\rightarrow M_1D$

$M_1 \rightarrow \in$ `{TOP (Offset): = 0 ;}`

$D \rightarrow D$ ID

$D \rightarrow$ id: T `{enter (top (tblptr), id.name, T.type top (offset)); top (offset): = top (offset) + T. width ;}`

$T \rightarrow$ integer `{T.type : = integer; T. width: = 4 :}`

$T \rightarrow$ double `{T.type: = double; T.width = 8 ;}`

$T \rightarrow * T_1$ `{T. type: = pointer (T. type); T.width = 4;}`

Need to remember the current offset before entering the block, and to restore it after the block is closed.

Example: Block → begin M4 Declarations statements end `{pop (tblptr); pop (offset) ;}`

$M_4 \rightarrow \in$ `{t: = mktable (top (tblptr); push (t, tblptr); push (top (offset), offset) ;`

Can also use the block number technique to avoid creating a new symbol table.

Field names in records

- A record declaration is treated as entering a block in terms of offset is concerned.
- Need to use a new symbol table.

Example: $T \rightarrow$ record M_5 D end

```
{T. type: = (top (tblptr));
 T. width = top (offset);
 pop (tblptr);
 pop (offset) ;}
```

$M_5 \rightarrow \in$

```
{t: = mktable (null);
push (t, tblptr);
push {(o, offset) ;}
```

Assignment Statements

Expressions can be of type integer, real, array and record. As part of translation of assignments into three address code, we show how names can be looked up in the symbol table and how elements of array can be accessed.

Code generation for assignment statements gen ([address # 1], [assignment], [address #2], operator, address # 3);

Variable accessing Depending on the type of [address # i], generate different codes.

Types of [address # i]:

- Local temp space
- Parameter
- Local variable
- Non-local variable
- Global variable
- Registers, constants,…

Error handling routine error – msg (error information);

The error messages can be written and stored in other file. Temp space management:

- This is used for generating code for expressions.
- newtemp (): allocates a temp space.
- freetemp (): free t if it is allocated in the temp space

Label management

- This is needed in generating branching statements.
- newlabel (): generate a label in the target code that has never been used.

Names in the symbol table

```
S→ id: = E {p: = lookup (id-name, top (tblptr));
            If p is not null then gen (p, ":=",
            E.place);
            Else error ("var undefined", id. Name)
            ;}
            E→E1+ E2 {E. place = newtemp ();
            gen (E.place, ": = ", E1.place, "+",
            E2.Place);   free   temp   (E1.pace);
            freetemp
            (E2. place) ;}
            E→ –E1 {E. place = newtemp ();
            gen   (E.place,   ":   =",   "uminus",
            E1.place);
             Freetemp (E1. place ;)}
E→(E1) {E. place = E1. place ;}
E→ id {p: = lookup (id.name, top (tblptr);
If p≠ null then E.place = p. place else error
("var undefined", id. name) ;}
```

Type conversions

Assume there are only two data types: integer, float.

For the expression,

$$E \to E_1 + E_2$$

If E_1. type = E_2. type then
generate no conversion code
E.type = E_1. type;

Else

    E.type = float;
    temp1 = newtemp ();
    If E1. type = integer then
    gen (temp1,':=' int - to - float, E1.place);
    gen (E,':=' temp1, '+', E2.place);

Else

    gen (temp1,':=' int - to - float, E2. place);
    gen (E,':=' temp1, '+', E1. place);
    Free temp (temp1);

Addressing array elements

Let us assume

low: lower bound

w: element data width

Start_addr: starting address

1D Array: A[i]

- Start_addr + (i – low)* w = i * w + (start_addr – low *w)
- The value called base, (start_addr – low * w) can be computed at compile time and then stored at the symbol table.

Example: array [–8 …100] of integer.

To declare [–8] [–7] … [100] integer array in Pascal.

2D Array A [i_1, i_2]

Row major order: row by row. A [i] means the ith row.

1st row A [1, 1]
A [1, 2]

2nd row A [2, 1]
A [2, 2]
A [i, j] = A [i] [j]

Column major: column by column.

A [1, 1] ⋮ A [1, 2]
A [2, 1] ⋮ A [2, 2]
1st Column 2nd column

Address for A [i_1, i_2]:

Start _ addr + ((i, – low$_1$) *n_2 + (i_2 – low$_2$))*w

Where low$_1$ and low$_2$ are the lower bounds of i_1 and i_2. n_2 is the number of values that i_2 can take. High$_2$ is the upper bound on the valve of i_2. n_2 = high$_2$ – low$_2$ + 1

We can rewrite address for A [i_1, i_2] as (($i_1 \times n_2$) + i_2) $\times$ w + (start _ addr - ((low$_1$ $\times$ n_2) + low$_2$) $\times$ w). The value (start _ addr – low$_1$ $\times$ n_2 $\times$ w – low$_2$ $\times$ w) can be computed at compiler time and then stored in the symbol table.

Multi-Dimensional Array A [i_1, i_2,…i_k]

Address for A [i_1, i_2,…i_k]

$$= i_1 * \pi_{i=2}^{k}{}^{n_i} + i_2 * \pi_{i=3}^{k}{}^{n_i} + \cdots + i_k \Big) * w$$
$$+ \Big(start\_addr - low_1 * w * \pi_{i=2}^{k} ni$$
$$- low_2 * w * \pi_{i=3}^{k}{}^{n_i} - \cdots - low_k * w \Big)$$

It can be computed incrementally in grammar rules:

f (1) = i$_1$;

f (j) = f (j –1) * n_j + i_j;

f (k) is the value we wanted to compute.

Attributes needed in the translation scheme for addressing array elements:

Elegize: size of each element in the array

Array: a pointer to the symbol table entry containing information about the array declaration.

Ndim: the current dimension index

Base: base address of this array

Place: where a variable is stored.

Limit (array, n) = n_m is the number of elements in the mth coordinate.

Translation scheme for array elements

Consider the grammar

$S \to L: = E$

$E \to L$

$L \rightarrow$ id

$L \rightarrow$ [Elist]

Elist$\rightarrow$ Elist$_1$, E

Elist$\rightarrow$ id [E]

$E \rightarrow$ id

$E \rightarrow E + E$

$E \rightarrow (E)$

- $S \rightarrow L := E$ `{if L. offset = null then /* L is a simple id */ gen (L. place, ":=", E.place);`
```
Else
gen (L. place, "[", L. offset, "]",":=",
E.place);
```
- $E \rightarrow E_1 + E_2$ `{E.place = newtemp ();`
```
gen (E. place, ":=", E1.place, "+", E2.
place) ;}
```
- $E \rightarrow (E_1)$ `{E.place= E1.place}`
- $E \rightarrow L$ `{if L. offset = null then /* L is a simple id */ E.place:= L .place);`
```
Else begin
E.place:=newtemp();
gen (E.place, ":=",L.place, "[",L.offset,
']");
end }
```
- $L \rightarrow$ id `{P! = lookup (id.name, top (tblptr));`
```
      If P ≠ null then
      Begin
      L.place: = P.place:
      L.offset:= null;
      End
      Else
      Error ("Var underfined", id. Name) ;}
```
- $L \rightarrow$ Elist `{L. offset: = newtemp ();`
```
    gen (L. offset, ":=", Elist.elesize,
    "*", Elist.place );
    freetemp (Elist.place);
    L.Place := Elist . base ;}
```
- Elist$\rightarrow$ Elist$_1$, E `{t: =newtemp (); m: = Elist1.`
```
ndim+1;
gen (t, ":=" Elist1.place, "*", limit (Elist1.
array, m));
Gen (t, ":=", t"+", E.place); freetemp
(E.place);
Elist.array: = Elist.array;
Elist.place:= t; Elist.ndim:= m ;}
```

Elist $\rightarrow$ id [E `{Elist.Place:= E.place; Elist.`
```
ndim:=1;
P! = lookup (id.name, top (tblptr)); check
for id errors;
Elist.elesize:= P.size; Elist.base: = p.base;
Elist.array:= p.place ;}
```
- $E \rightarrow$ id `{P:= lookup (id,name, top (tblptr);`

```
Check for id errors; E. Place: = Populace ;}
```

Boolean Expressions

There are two choices for implementation of Boolean expressions:

1. Numerical representation
2. Flow of control

Numerical representation

Encode true and false values.
Numerically, 1:true 0: false.
Flow of control: Representing the value of a Boolean expression by a position reached in a program.

Short circuit code: Generate the code to evaluate a Boolean expression in such a way that it is not necessary for the code to evaluate the entire expression.

- If a_1 or a_2
  a_1 is true then a_2 is not evaluated.
- If a_1 and a_2
  a_1 is false then a_2 is not evaluated.

Numerical representation

$E \rightarrow$ id$_1$ relop id2
```
{B.place:= newtemp ();
gen ("if", id1.place, relop.op, id2.
place,"goto", next stat +3);
gen (B.place,":=", "0");
gen ("goto", nextstat+2);
gen (B.place,":=", "1")'}
```

Example 1: Translate the statement (if $a < b$ or $c < d$ and $e < f$) without short circuit evaluation.

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

Flow of Control Statements

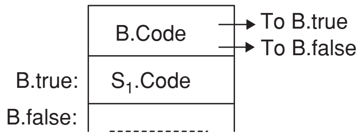
$B\rightarrow$ id$_1$ relop id$_2$
```
{
B.true: = newlabel ();
B.false:= newlabel ();
B.code:= gen ("if", id1. relop, id2, "goto",
B.true, "else", "goto", B. false) ||
gen (B.true, ":")
}
```

S→if *B* then S_1 S.code:= B.code || S_1 .code ||gen (B.false, ':')

|| is the code concatenation operator.

1. **If – then implementation:**

   S →if *B* then *S*1 `{gen (Befalls," :");}`



2. **If – then – else**

   P→*S* `{S.next:= newlabel ();`

   `P.code:= S.code || gen (S.next," :")}`

   S → if *B* then S_1 else S_2 `{`S_1`.next:= S.next;`

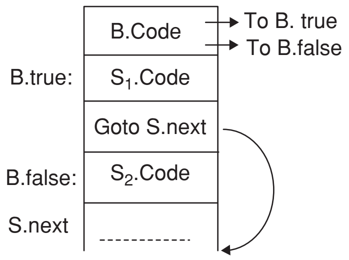
   S_2`.next:= S.next;`

   `Secede: = B.code || `S_1`.code ||.`

   `Gen ("goto" S.next) || B. false," :")`

   `||`S_2`.code}`

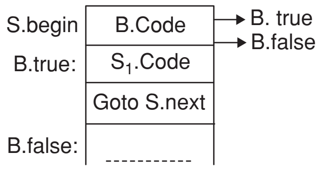
   Need to use inherited attributes of *S* to define the attributes of S_1 and S_2



3. **While loop:**

   B→ id_1 relop id_2 B.true:= newlabel ();
   B.false:= newlabel ();
   B.code:=gen ('if', id.relop,
   id_2, 'goto', B.true 'else', 'goto', B. false) ||
   gen (B.true ':');

   S→ while *B* do S_1 S.begin:= newlabel ();
   S.code:=gen (S.begin,':')||
   B.code||S1.code || gen
   ('goto', S.begin) || gen (B.false, ':');



4. **Switch/case statement:**

   The c - like syntax of switch case is

   switch epr {

```
case V [1]: S [1]
.
.
.
case V [k]: S[k]
default: S[d]
              }
```

Translation sequence

- Evaluate the expression.
- Find which value in the list matches the value of the expression, match default only if there is no match.
- Execute the statement associated with the matched value.

How to find the matched value? The matched value can be found in the following ways:

1. Sequential test
2. Lookup table
3. Hash table
4. Back patching

Two different translation schemes for sequential test are shown below:

1. Code to evaluate E into t

Goto test

L[*i*]: code for S [1]

goto next

L[*k*]: code for *S*[*k*]

goto next

L[*d*]: code for *S*[*d*]

Go to next test:

If $t = V$ [1]: goto *L* [1]

.

.

.

goto L[d]

Next:

2. Can easily be converted into look up table

   If *t* <> *V* [*i*] 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 [*k* – 1]: if *t* < > *V* [*k*] goto *L*[*k*]

   Code for *S*[*k*]

   Goto next

   .

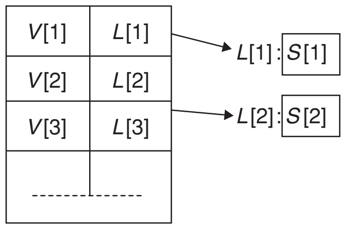
   .

   .

   L[*k*]: code for *S*[*d*]

   Next:

Use a table and a loop to find the address to jump



3. **Hash table:** When there are more than two entries use a hash table to find the correct table entry.
4. **Back patching:**
    - Generate a series of branching statements with the targets of jumps temporarily left unspecified.
    - To determine label table: each entry contains a list of places that need to be back patched.
    - Can also be used to implement labels and gotos.

Procedure Calls

- Space must be allocated for the activation record of the called procedure.
- Arguments are evaluated and made available to the called procedure in a known place.
- Save current machine status.
- When a procedure returns:
    - Place returns value in a known place.
    - Restore activation record.

Example: $S \rightarrow$ call id (Elist)

```
{for each item P on the queue Elist.
Queue do gen ('PARAM', q);
gen ('call:', id.place) ;}
```

Elist $\rightarrow$ Elist, E {append E.place to the end of Elist.queue}

Elist $\rightarrow$ E {initialize Elist.queue to contain only E.place}

Use a queue to hold parameters, then generate codes for params.

Code for E_1, store in t_1

.

.

.

Code for E_k, store in t_k

PARAM t1

:

.

.

PARAM tk

Call P

Terminology:

Procedure declaration:

Parameters, formal parameters

Procedure call:

Arguments, actual parameters.

The values of a variable: $x = y$

r – value: value of the variable, i.e., on the right side of assignment. Ex: y, in above assignment.

l – value: The location/address of the variable, i.e., on the leftside of assignment. Ex: x, in above assignment.

There are different modes of parameter passing

1. call-by-value
2. call-by-reference
3. call-by-value-result (copy-restore)
4. call-by-name

Call by value

Calling procedure copies the r values of the arguments into the called procedure's Activation Record.

Changing a formal parameter has no effect on the actual parameter.

Example: void add (int C)

```
{
C = C+ 10;
printf ('\nc = %d', &C);
}
main ()
{
int a = 5;
printf ('a=%d', &a);
add (a);
printf ('\na = %d', &a);
}
```

In main a will not be affected by calling add (a)

It prints $a = 5$

$a = 5$

Only the value of C in add ( ) will be changed to 15.

Usage:

1. Used by PASCAL and C++ if we use non-var parameters.
2. The only thing used in C.

Advantages:

1. No aliasing.
2. Easier for static optimization analysis.
3. Faster execution because of no need for redirecting.

Call by reference

Calling procedure copies the l-values of the arguments into the called procedure's activation record. i.e., address will be passed to the called procedure.

- Changing formal parameter affects the corresponding actual parameter.
- It will have some side effects.

Example: void add (int *c)

```
{
*c = *c + 10;
printf('\nc=%d', *c);
```

```
}
void main()
{
int a = 5;
printf ('\na = %d', a);
add (&a);
printf ('\na = %d', a);
```

output: a = 5
c = 15
a = 15

That is, here the actual parameter is also modified.

Advantages

1. Efficiency in passing large objects.
2. Only need to copy addresses.

Call-by-value-result

Equivalent to call-by-reference except when there is aliasing. That is, the program produces the same result, but not the same code will be generated.

Aliasing: Two expressions that have the same l-values are called aliases. They access the same location from different places.

Aliasing happens through pointer manipulation.

1. Call by reference with global variable as an argument.
2. Call by reference with the same expression as argument twice.

Example: test (x,y,x)

Advantages:

1. If there is no aliasing, we can implement it by using call – by – reference for large objects.
2. No implicit side effect if pointers are not passed.

Call by-name

used in Algol.

- Procedure body is substituted for the call in calling procedure.
- Each occurrence of a parameter in the called procedure is replaced with the corresponding argument.
- Similar to macro expansion.
- A parameter is not evaluated unless its value is needed during computation.

Example:

```
void show (int x)
{
for (int y = 0; y < 10; y++)
x++;
}
main ()
{
int j;
j = -1;
show (j);
}
Actually it will be like this
main ()
{
int j;
j = - 1;
For (in y= 0; y < 10; y ++)
x ++;
}
```

- Instead of passing values or address as arguments, a function is passed for each argument.
- These functions are called **thunks.**
- Each time a parameter is used, the thunk is called, then the address returned by the thunk is used.

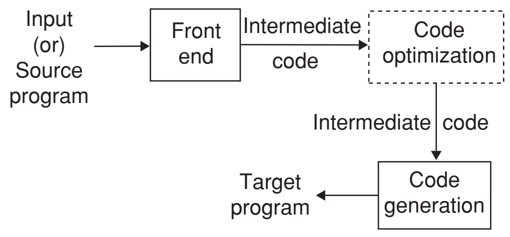
$y = 0$: use return value of thunk for y as the ℓ-value.

Advantages

- More efficient when passing parameters that are never used.
- This saves lot of time because evaluating unused parameter takes a longtime.

Code Generation

Code generation is the final phase of the compiler model.



The requirements imposed on a code generator are

1. Output code must be correct.
2. Output code must be of high quality.
3. Code generator should run efficiently.

Issues in the Design of a Code Generator

The generic issues in the design of code generators are

- Input to the code generator
- Target programs
- Memory Management
- Instruction selection
- Register Allocation
- Choice of Evaluation order

Input to the code generator

Intermediate representation with symbol table will be the input for the code generator.

- High Level Intermediate representation

Example: Abstract Syntax Tree (AST)

- Medium – level intermediate representation

Example: control flow graph of complex operations

- Low – Level Intermediate representation

Example: Quadruples, DAGS

- Code for abstract stack machine, i.e., postfix code.

Target programs

The output of the code generator is the target program. The output may take on a variety of forms:

1. Absolute machine language
2. Relocatable machine language
3. Assembly language

Absolute machine language

- Final memory area for a program is statically known.
- Hard coded addresses.
- Sufficient for very simple systems.

Advantages:

- Fast for small programs
- No separate compilation

Disadvantages: Can not call modules from other languages/ compliers.

Relocatable code It Needs

- Relocation table
- Relocating linker + loader (or) runtime relocation in Memory management Unit (MMU).

Advantage: More flexible.

Assembly language Generates assembly code and use an assembler tool to convert this to binary (object) code. It needs (i) assembler (ii) linker and loader.

Advantage: Easier to handle and closer to machine.

Memory management

Mapping names in the source program to addresses of data objects in runtime memory is done by the front end and the code generator.

- A name in a three address statement refers to a symbol entry for the name.
- Stack, heap, garbage collection is done here.

Instruction selection

Instruction selection depends on the factors like

- Uniformity
- Completeness of the instruction
- Instruction speed
- Machine idioms

- Choose set of instructions equivalent to intermediate representation code.
- Minimize execution time, used registers and code size.

Example: $x = y + z$ in three address statements:

MOV y, R_0 / * load y into R_0 * /
ADD z, R_0
MOV R_0, x /* store R_0 into x*/

Register allocation

- Instructions with register operands are faster. So, keep frequently used values in registers.
- Some registers are reserved.

Example: *SP*, *PC* … etc.
Minimize number of loads and stores.

Evaluation order

- The order of evaluation can affect the efficiency of the target code.
- Some orders require fewer registers to hold intermediate results.

Target Machine

Lets us assume, the target computer is

- Byte addressable with 4 bytes per word
- It has n general purpose registers $R_0, R_1, R_2, \ldots R_{n-1}$
- It has 2 address instructions of the form

  OP source, destination
  [cost: 1 + added]

Example: The op may be MOV, ADD, MUL.
Generally cost will be like this

Source	Destination	Cost
Register	Register	1
Register	Memory	2
Memory	Register	2
Memory	Memory	3

Addressing modes:

Mode	Form	Address	Cost
Absolute	M	M	2
Register	R	R	1
Indexed	C(R)	C+contents(R)	2
Indirect register	*R	Contents (R)	1
Indirect indexed	*C(R)	Contents (C+contents (R))	2

Example: $x: = y - z$
MOV y, R0 $\rightarrow$ cost = 2
SUB z, R0 $\rightarrow$ cost = 2
MOV R_0, x $\rightarrow$ cost = 2
6

Runtime Storage Management

Storage Organization

To run a compiled program, compiler will demand the operating system for the block of memory. This block of memory is called runtime storage.

This run time storage is subdivided into the generated target code, Data objects and Information which keeps track of procedure activations.

The fixed data (generated code) is stored at the statically determined area of the memory. The Target code is placed at the lower end of the memory.

The data objects are stored at the statically determined area as its size is known at the compile time. Compiler stores these data objects at statically determined area because these are compiled into target code. This static data area is placed on the top of the code area.

The runtime storage contains stack and the heap. Stack contains activation records and program counter, data object within this activation record are also stored in this stack with relevant information.

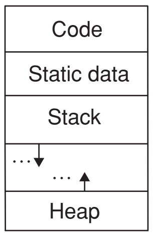
The heap area allocates the memory for the dynamic data (for example some data items are allocated under the program control)

The size of stack and heap will grow or shrink according to the program execution.

Activation Record

Information needed during an execution of a procedure is kept in a block of storage called an activation record.

- Storage for names local to the procedures appears in the activation record.
- Each execution of a procedure is referred as activation of the procedure.
- If the procedure is recursive, several of its activation might be alive at a given time.
- Runtime storage is subdivided into
  1. Generated target code area
  2. Data objects area
  3. Stack
  4. Heap



- Sizes of stack and heap can change during program execution.

For code generation there are two standard storage allocations:

1. **Static allocation:** The position of an activation record in memory is fixed at compile time.
2. **Stack allocation:** A new activation record is pushed on to the stack for each execution of the procedure.

The record is poped when the activation ends.

Control stack The control stack is used for managing active procedures, which means when a call occurs, the execution of activation is interrupted and status information of the stack is saved on the stack.

When control is returned from a call, the suspended activation is resumed after storing the values of relevant registers it also includes program counter which sets to point immediately after the call.

The size of stack is not fixed.

Scope of declarations Declaration scope refers to the certain program text portion, in which rules are defined by the language.

Within the defined scope, entity can access legally to declared entities.

The scope of declaration contains immediate scope always. Immediate scope is a region of declarative portion with enclosure of declaration immediately.

Scope starts at the beginning of declaration and scope continues till the end of declaration. Whereas in the over loadable declaration, the immediate scope will begin, when the callable entity profile was determined.

The visible part refers text portion of declaration, which is visible from outside.

Flow Graph

A flow graph is a graph representation of three address statement sequences.

- Useful for code generation algorithms.
- Nodes in the flow graph represents computations.
- Edges represent flow of control.

Basic Blocks

Basic blocks are sequences of consecutive statements in which flow of control enters at the beginning and leaves at the end without a halt or branching.

1. First determine the set of leaders
   - First statement is leader
   - Any target of goto is a leader
   - Any statement that follows a goto is a leader.
2. For each leader its basic block consists of the leader and all statements up to next leader.

Initial node: Block with first statement is leader.

Example: consider the following fragment of code that computes dot product of two vectors *x* and *y* of length 10.
begin
Prod: = 0;

```
i: = 1;
repeat
begin
Prod: = Prod + x [i] * y [i];
i: = i + 1;
end
until i < = 10;
end
```

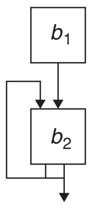
B_1

(1)	Prod : = 0
(2)	I: = 1

B_2

(3)	t_1:= 4*i
(4)	t_2: =x[t_1]
(5)	t_3: =4 * i
(6)	t_4: =y [t_3]
(7)	t_5: =t_2* t_4
(8)	t_6; =Prod + t_5
(9)	Prod := t_6
(10)	t_7: = i+1
(11)	i:= t_7
(12)	if i < = 10 goto (3)

∴ The flow graph for this code will be



Here b_1 is the initial node/block.

- Once the basic blocks have been defined, a number of transformations can be applied to them to improve the quality of code.
  1. **Global:** Data flow analysis
  2. **Local:**
     - Structure preserving transformations
     - Algebraic transformations
- Basic blocks compute a set of expressions. These expressions are the values of the names live on exit from the block.
- Two basic blocks are equivalent if they compute the same set of expressions.

Structure preserving transformations:

1. *Common sub-expression elimination:*

$$\begin{array}{l} a := b + c \\ b := a - d \\ c := b + c \\ d := a - d \end{array} \quad \Rightarrow \quad \begin{array}{l} a := b + c \\ b := a - d \\ c := b + c \\ d := b \end{array}$$

2. *Dead code elimination:* Code that computes values for names that will be dead i.e., never subsequently used can be removed.
3. *Renaming of temporary variables*
4. *Interchange of two independent adjacent statements*

Algebraic Transformations

Algebraic identities represent other important class optimizations on basic blocks. For example, we may apply arithmetic identities, such as $x + 0 = 0 + x = x$,

$$x * 1 = 1 * x = x$$
$$x - 0 = x$$
$$x/1 = x$$

Next-Use Information

- Next-use info used in code generation and register allocation.
- Remove variables from registers if not used.
- Statement of the form $A = B$ or C defines A and uses B and C.
- Scan each basic block backwards.
- Assume all temporaries are dead or exit and all user variables are live or exit.

Algorithm to compute next use information

Suppose we are scanning

i: x: = y op z

in backward scan

- attach to i, information in symbol table about x, y, z.
- set x to not live and no next-use in symbol table
- set y and z to be live and next-use in symbol table.

Consider the following code:

1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_3 = 2 * t_2$

4: $t_4 = t_1 + t_2$

5: $t_5 = b * b$

6: $t_6 = t_4 + t_5$

7: $x = t_6$

Statements:

7: no temporary is live

6: t_6: use (7) t_4 t_5 not live

5: t_5: use (6)

4: t_4: use (6), t_1 t_3 not live

3: t_3: use (4) t_2 not live

2: t_2: use (3)

1: t_1: use (4)

Symbol Table:

t_1 dead use in 4

t_2 dead use in 3
t_3 dead use in 4
t_4 dead use in 6
t_5 dead use in 6
t_6 dead use in 7
The six temporaries in the basic block can be packed into two locations t_1 and t_2:
1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_2 = 2 * t_2$
4: $t_1 = t_1 + t_2$
5: $t_2 = b * b$
6: $t_1 = t_1 + t_2$
7: $x = t_1$

Code Generator

- Consider each statement
- Remember if operand is in a register
- Descriptors are used to keep track of register contents and address for names
- There are 2 types of descriptors
  1. Register Descriptor
  2. Address Descriptor

Register Descriptor

Keep track of what is currently in each register. Initially all registers are empty.

Address Descriptors

- Keep track of location where current value of the name can be found at runtime.
- The location might be a register, stack, memory address or a set of all these.

Issues in design of code generation The issues in the design of code generation are

1. Intermediate representation
2. Target code
3. Address mapping
4. Instruction set.

Intermediate Representation It is represented in post fix, 3-address code (or) quadruples and syntax tree (or) DAG.

Target Code The Target Code could be absolute code, relocatable machine code (or) assembly language code. Absolute code will execute immediately as it is having fixed address relocatable, requires linker and loader to get the code from appropriate location for the assembly code, assemblers are required to convert it into machine level code before execution.

Address mapping In this, mapping is defined between intermediate representations to target code address.

It is based on run time environment like static, stack or heap.

Instruction set It should provide a complete set in such a way that all its operations can be implemented.

Code Generation Algorithm

For each three address statement $x = y$ op z do

- Invoke a function getreg to determine location L where x must be stored. Usually L is a register.
- Consult address descriptor of y to determine y'. Prefer a register for y'. If value of y is not already in L generate MOV y', L.
- Generate

  OP z', L

  Again prefer a register for z. Update address descriptor of x to indicate x is in L. If L is a register update its descriptor to indicate that it contains x and remove x from all other register descriptors.
- If current value of y and/or z have no next use and are dead or exit from block and are in registers then change the register descriptor to indicate that it no longer contain y and /or z.

Function getreg

1. If y is in register and y is not live and has no next use after $x = y$ OP z then return register of y for L.
2. Failing (1) return an empty register.
3. Failing (2) if x has a next use in the block or OP requires register then get a register R, store its contents into M and use it.
4. Else select memory location x as L.

Example: $D: = (a - b) + (a - c) + (a - c)$

Stmt	Code Generated	reg desc	addr desc
$t = a - b$	MOV a, R_0	R_0 contains t	t in R_0
	SUB b, R_0		
$u = a - c$	MOV a, R_1	R_0 contains t	t in R_0
	SUB c, R_1	R_1 contains u	u in R_1
$v = t + u$	ADD R_1, R_0	R_0 contains v	u in R_0
		R_1 contains u	v in R_0
$d = v + u$	ADD R_1, R_0	R_o contains d	d in R_0
	MOV R_0,d		d in R_0 and memory

Conditional Statements

Machines implement conditional jumps in 2 ways:

1. Based on the value of the designated register (R) Branch if values of R meets one of six conditions.

(i) Negative	(ii) Zero
(iii) Positive	(iv) Non-negative
(v) Non-zero	(vi) Non-positive

Example: Three address statement: if $x < y$ goto z
It can be implemented by subtracting y from x in R, then jump to z if value of R is negative.

2. Based on a set of **condition codes** to indicate whether last quantity computed or loaded into a location is negative (or) Zero (or) Positive.
   - compare instruction set codes without actually computing the value.

Example: CMP x, y
CJL Z.

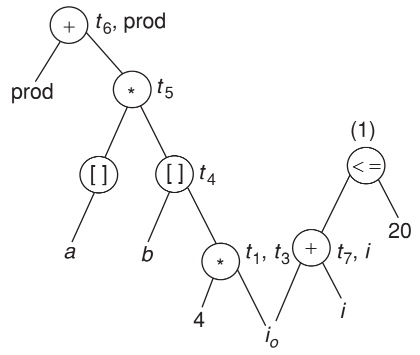
- Maintains a condition code descriptor, which tells the name that last sets the condition codes.

Example: X: = $y + z$
If $x < 0$ goto z
By
MOV y, R_o
ADD z, R_o
MOV R_o, x
CJN z.

DAG Representation of Basic Blocks

- DAGS are useful data structures for implementing transformations on basic blocks.
- Tells, how value computed by a statement is used in subsequent statements.
- It is a good way of determining common sub expressions.
- A DAG for a basic block has following labels on the nodes:
  - Leaves are labeled by unique identifiers, either variable names or constants.
  - Interior nodes are labeled by an operator symbol.
  - Nodes are also optionally given as a sequence of identifiers for labels.

Example: 1: t_1:= 4 * i
2: t_2:= a [t_1]
3: t_3:= 4 * i
4: t_4:= b [t_3]
5: t_5:= t_2 * t_4
6: t_6:= prod + t_5
7: prod: = t_6
8: t_7:= i + 1
9: i= t_7
10: if i < = 20 got (1)

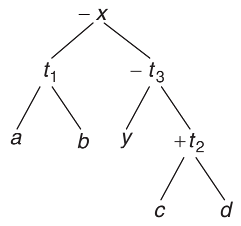


Code Generation from DAG:

$S_1 = 4 * i$	$S_1 = 4 * i$
S_2 = add(A) – 4	S_2 = add(A) – 4
$S_3 = S_2 [S_1]$	$S_3 = S_2 [S_1]$
$S_4 = 4 * i$	
S_5 = add(B) – 4	S_5 = add(B) – 4
$S_6 = S_5[S_4]$	$S_6 = S_5[S_4]$
$S_7 = S_3 * S_6$	$S_7 = S_3 * S_6$
S_8 = prod + S_7	prod = prod + S_7
prod = S_8	
$S_9 = I + 1$	
$I = S_9$	$I = I + 1$
if I < = 20 got (1)	if I < = 20 got (1)

Rearranging order of the code

Consider the following basic block
t_1:= $a + b$
t_2:= $c + d$
t_3:= $e - t_2$
$x = t_1 - t_3$ and its DAG



Three address code for the DAG:
(Assuming only two registers are available)

MOV a, R_o	
ADD b, R_o	
MOV c, R_1	
MOV R_o, t_1	Register Spilling
MOV e, R_o	Register Reloading
SUB R_1, R_o	
MOV t_1, R_1	
SUB R_o, R_1	
MOV R_1, x	

Rearranging the code as
t_2:= $c + d$
t_3:= $e - t_2$
t_1:= $a + b$
$x = t_1 - t_3$
The rearrangement gives the code:
MOV c, R_o
ADD d, R_o
MOV e, R_1
SUB R_o, R_1

MOV a, R_o
ADD b, R_o
SUB R_1, R_0
MOV R_1, x

Error detection and Recovery The errors that arise while compiling

1. Lexical errors
2. Syntactic errors
3. Semantic errors
4. Run-time errors

Lexical errors If the variable (or) constants are declared (or) defined, not according to the rules of language, special symbols are included which were not part of the language, etc is the lexical error.

Lexical analyzer is constructed based on pattern recognizing rules to form a token, when a source code is made into tokens and if these tokens are not according to rules then errors are generated.

Consider a c program statement
printf ('Hello World');

Main printf, (, ', Hello world,' ,),; are tokens.

Printf is not recognizable pattern, actually it should be printf. It generates an error.

Syntactic error These errors include semi colons, missing braces etc. which are according to language rules.

The parser reports the errors

Semantic errors This type of errors arises, when operation is performed over incompatible type of variables, double declaration, assigning values to undefined variables etc.

Runtime errors The Runtime errors are the one which are detected at runtime. These include pointers assigned with NULL values and accessing a variable which is out of its boundary, unlegible arithmetic operations etc.

After the detection of errors. The following recovery strategies should be implemented.

1. Panic mode recovery
2. Phrase level recovery
3. Error production
4. Global correction.

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs.
- Improving the performance of the target program by examining a short sequence of target instructions (peephole) and replacing these instructions by a shorter or faster sequence is peephole optimization.
- The peephole is a small, moving window on the target program. Some well known peephole optimizations are

1. Eliminating redundant instructions
2. Eliminating unreachable code
3. Flow of control optimizations or Eliminating jumps over jumps
4. Algebraic simplifications
5. Strength reduction
6. Use of machine idioms

Elimination of Redundant Loads and stores

Example 1: (1) MOV R_o, a
(2) MOV a, R_o

We can delete instruction (2), because the value of a is already in R_0.

Example 2: Load x, R_0
Store R_0, x

If no modifications to R_0/x then store instruction can be deleted

Example 3: (1) Load x, R_0
(2) Store R_0, x

Example 4: (1) store R_0, x
(2) Load x, R_0

Second instruction can be deleted from both examples 3 and 4.

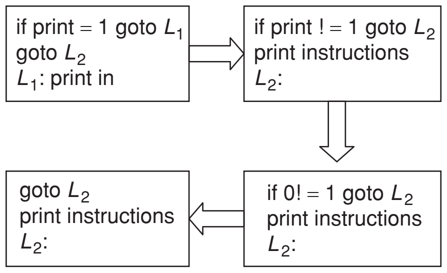
Example 5: Store R_0, x
Load x, R_0
Here load instruction can be deleted.

Eliminating Unreachable code

An unlabeled instruction immediately following and unconditional jump may be removed.

- May be produced due to debugging code introduced during development.
- May be due to updates in programs without considering the whole program segment.

Example: Let print = 0



```
if print = 1 goto L1
goto L2
L1: print in
```

```
if print ! = 1 goto L2
print instructions
L2:
```

```
if 0! = 1 goto L2
print instructions
L2:
```

```
goto L2
print instructions
L2:
```

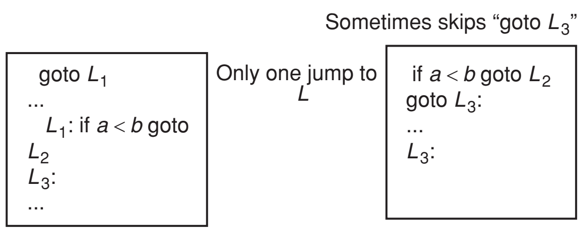
In all of the above cases print instructions are unreachable.
∴ Print instructions can be eliminated.

Example: goto L_2
…
L_2:

Flow of control optimizations The unnecessary jumps can be eliminated.
Jumps like:
Jumps to jumps,
Jumps to conditional jumps,
Conditional jumps to jumps.

Example 1: we can replace the jump sequence
goto L_1
…
L_1: got L_2
By the sequence
Got L_2
L_1: got $L_{2,}$
…
If there are no jumps to L_1 then it may be possible to eliminate the statement L_1: goto L_2.

Example 2:


Sometimes skips "goto L_3"

```
goto L1
...
L1: if a < b goto
L2
L3:
...
```

Only one jump to L

```
if a < b goto L2
goto L3:
...
L3:
```

Reduction in strength

- x^2 is cheaper to implement as $x * x$ than as a call to exponentiation routine.
- Replacement of multiplication by left shift.

Example: $x * 2^3 \Rightarrow x << 3$

- Replace division by right shift.

Example: $x >> 2$ (is $x/2^2$)

Use of machine Idioms

- Auto increment and auto decrement addressing modes can be used whenever possible.

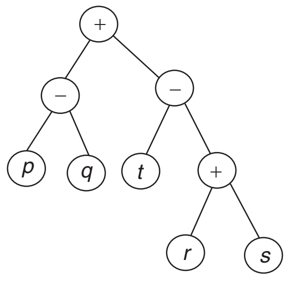
Example: replace add #1, R by INC R

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices

1. Consider the following expression tree on a machine with bad store architecture in which memory can be accessed only through load and store instructions. The variables p, q, r, s and t are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when the operands are in registers. The instructions produce result only in a register if no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression?



(A) 2 (B) 9
(C) 5 (D) 3

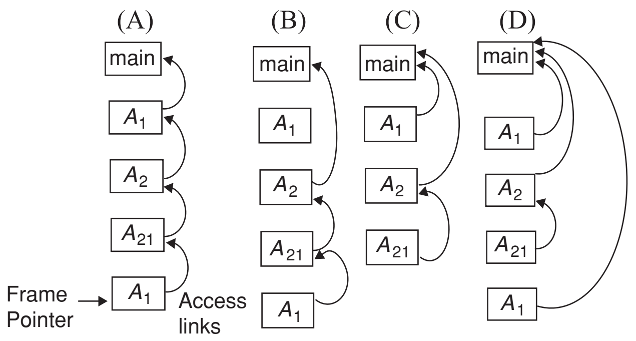
2. Consider the program given below with lexical scoping and nesting of procedures permitted.

Program main ( )

```
{
Var …
Procedure A1 ( )
{
Var …
call A2;
}
Procedure A2 ( )
{
Var..
Procedure A21 ( )
{
Var…
call A21 ( );
}
Call A1;
}
Call A1;
}
```

Consider the calling chain: main ( )$\rightarrow A_1$ ( ) $\rightarrow A_2$ ( ) $\rightarrow$ A_{21} ( ) $\rightarrow A_1$ ( ).

The correct set of activation records along with their access links is given by



3. Consider the program fragment:

```
sum = 0;
For (i = 1; i < = 20; i++)
sum = sum + a[i] +b[i];
```

How many instructions are there in the three-address code for this?

(A) 15 (B) 16
(C) 17 (D) 18

4. Suppose the instruction set of the processor has only two registers. The code optimization allowed is code motion. What is the minimum number of spills to memory in the complied code?

$c = a + b$;

$d = c*a$;

$e = c + a$;

$x = c*c$;

```
If (x > a)
{
y = a*a;
Else
{
d = d*d; e = e*e;
}
```

(A) 0 (B) 1
(C) 2 (D) 3

5. What is the minimum number of registers needed to compile the above problem's code segment without any spill to memory?

(A) 3 (B) 4
(C) 5 (D) 6

6. Convert the following expression into postfix notation:

$a = (-a + 2*b)/a$

(A) $aa - 2b$ *+a/= (B) $a - 2ba$ */+ =
(C) $a2b * a/+$ (D) $a2b - * a/+$

7. In the quadruple representation of the following program, how many temporaries are used?

int $a = 2, b = 8, c = 4, d$;

For ($j = 0; j <= 10; j$++)

$a = a * (j* (b/c))$;

$d = a * (j* (b/c))$;

(A) 4 (B) 7
(C) 8 (D) 10

8. Let $A = 2, B = 3, C = 4$ and $D = 5$, what is the final value of the prefix expression: + * $AB - CD$

(A) 5 (B) 10
(C) –10 (D) –5

9. Which of the following is a valid expression?

(A) $BC * D - +$ (B) $* ABC -$
(C) BBB ***– + (D) $-*/bc$

10. What is the final value of the postfix expression $B\ C\ D\ A\ D - + - +$ where $A = 2, B = 3, C = 4, D = 5$?

(A) 5 (B) 4
(C) 6 (D) 7

11. Consider the expression $x = (a + b)* -C/D$. In the quadruple representation of this expression in which instruction '/' operation is used?

(A) 3rd (B) 4th
(C) 5th (D) 8th

12. In the triple representation of $x = (a + b)* - c/d$, in which instruction $(a + b) * - c/d$ result will be assigned to x?

(A) 3rd (B) 4th
(C) 5th (D) 8th

13. Consider the three address code for the following program:

While ($A < C$ and $B > D$) do

If ($A == 1$) then $C = C + 1$;

Else

While ($A <= D$) do

$A = A + 3$;

How many temporaries are used?

(A) 2 (B) 3
(C) 4 (D) 0

14. Code generation can be done by

(A) DAG (B) Labeled tree
(C) Both (A) and (B) (D) None of these

15. Live variables analysis is used as a technique for

(A) Code generation (B) Code optimization
(C) Type checking (D) Run time management

Practice Problems 2

Directions for questions 1 to 19: Select the correct alternative from the given choices

1. Match the correct code optimization technique to the corresponding code:

(i) $i = i * 1$ $j = 2 * i$	⇒ $j = 2 * i$	(p) Reduction in strength
(ii) $A = B + C$ $D = 10 + B + C$	⇒ $A = B + C$ $D = 10 + A$	(q) Machine Idioms
(iii) For $i = 1$ to 10 $A[i] = B + C$	⇒ for $i = 1$ to 10 $t = B + C$ $A[i] = t$;	(r) Common sub expression elimination.
(iv) $x = 2 * y$	⇒ $y << 2$;	(s) Code motion

(A) i – r, iii – s, iv – p, ii – q
(B) i – q, ii – r, iii – s, iv –p
(C) i – s, iii – p, iii – q, iv – r
(D) i – q, ii – p, iii – r, iv – s

2. What will be the optimized code for the following expression represented in DAG?

$a = q * -r + q * -r$

(A) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = a * t_1$
$t_4 = t_2 + t_3$
$a = t_4$

(B) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

(C) $t_1 = -r$
$t_2 = q$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_3$
$a = t_4$

(D) All of these

3. In static allocation, names are bound to storage at ________ time.

(A) Compile (B) Runtime
(C) Debugging (D) Both (A) and (B)

4. The actual parameters are evaluate d and their r-values are passed to the called procedure is known as

(A) call-by-reference
(B) call-by-name
(C) call-by-value
(D) copy-restore

5. If the expression $-(a + b) *(c + d) + (a + b + c)$ is translated into quadruple representation, then how many temporaries are required?

(A) 5 (B) 6
(C) 7 (D) 8

6. If the above expression is translated into triples representation, then how many instructions are there?

(A) 6 (B) 10
(C) 5 (D) 8

7. In the indirect triple representation for the expression $A = (E/F) * (C - D)$. The first pointer address refers to

(A) $C - D$
(B) E/F
(C) Both (A) and (B)
(D) $(E/F) * (C - D)$

8. For the given assembly language, what is the cost for it?

MOV b, a

ADD c, a

(A) 3 (B) 4
(C) 6 (D) 2

9. Consider the expression

((4 + 2 * 3 + 7) + 8 * 5). The polish postfix notation for this expression is

(A) 423* + 7 + 85*+ (B) 423* + 7 + 8 + 5*
(C) 42 + 37 + *85* + (D) 42 + 37 + 85** +

Common data for questions 10 to 15: Consider the following basic block, in which all variables are integers, and ** denotes exponentiation.

$a: = b + c$
$z: = a ** 2$
$x: = 0 * b$
$y: = b + c$
$w: = y * y$
$u: = x + 3$
$v: = u + w$

Assume that the only variables that are live at the exit of this block are v and z. In order, apply the following optimization to this basic block.

10. After applying algebraic simplification, how many instructions will be modified?

(A) 1 (B) 2
(C) 4 (D) 5

11. After applying common sub expression elimination to the above code. Which of the following are true?

(A) $a: = b + c$ (B) $y: = a$
(C) $z = a + a$ (D) None of these

12. Among the following instructions, which will be modified after applying copy propagation?

(A) $a: = b + c$ (B) $z: = a * a$
(C) $y: = a$ (D) $w: = y * y$

13. Which of the following is obtained after constant folding?

(A) $u: = 3$ (B) $v: = u + w$
(C) $x: = 0$ (D) Both (A) and (C)

14. In order to apply dead code elimination, what are the statements to be eliminated?

(A) $x = 0$
(B) $y = b + c$
(C) Both (A) and (B)
(D) None of these

15. How many instructions will be there after optimizing the above result further?

(A) 1 (B) 2
(C) 3 (D) 4

16. Consider the following program:

L_0: $e: = 0$
$b: = 1$
$d: = 2$
L_1: $a: = b + 2$
$c: = d + 5$
$e: = e + c$
$f: a*a$
If $f < c$ goto L_3
L_2: $e: = e + f$
goto L_4
L_3: $e: = e + 2$
L_4: $d: = d + 4$
$b: = b - 4$
If $b! = d$ goto 4
L_5:

How many blocks are there in the flow graph for the above code?

(A) 5
(B) 6
(C) 8
(D) 7

17. A basic block can be analyzed by

(A) Flow graph
(B) A graph with cycles
(C) DAG
(D) None of these

18. In call by value the actual parameters are evaluated. What type of values is passed to the called procedure?

(A) l-values
(B) r-values
(C) Text of actual parameters
(D) None of these

19. Which of the following is FALSE regarding a Block?

(A) The first statement is a leader.
(B) Any statement that is a target of conditional / unconditional goto is a leader.
(C) Immediately next statement of goto is a leader.
(D) The last statement is a leader.

Previous Years' Questions

1. 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 ________ **[2015]**

2. Consider the intermediate code given below.

(1) $i = 1$
(2) $j = 1$
(3) $t_1 = 5 * i$
(4) $t_2 = t_1 + j$
(5) $t_3 = 4 * t_2$
(6) $t_4 = t_3$
(7) $a[t_4] = -1$
(8) $j = j + 1$
(9) if $j <= 5$ goto (3)
(10) $i = i + 1$
(11) if $i < 5$ goto (2)

The number of nodes and edges in the control-flow-graph constructed for the above code, respectively, are **[2015]**

(A) 5 and 7 (B) 6 and 7
(C) 5 and 5 (D) 7 and 8

3. Consider the following code segment. **[2016]**

$x = u - t;$
$y = x * v;$
$x = y + w;$
$y = t - z;$
$y = x * y;$

The minimum number of total variables required to convert the above code segment *to static single assignment form is* _____ .

4. What will be the output of the following pseudo-code when parameters are passed by reference and dynamic scoping is assumed? **[2016]**

$a = 3;$
void $n(x)$ { $x = x* a$; print (x);}
void $m(y)$ {$a = 1$; $a = y - a$; $n(a)$; print (a)}
void main( ) {$m(a)$;}

(A) 6,2 (B) 6,6
(C) 4,2 (D) 4,4

5. Consider the following intermediate program in three address code

$p = a - b$
$q = p * c$
$p = u * v$
$q = p + q$

Which one of the following corresponds to a *static single assignment* form of the above code? **[2017]**

(A) $p_1 = a - b$
$q_1 = p_1 * c$
$p_1 = u * v$
$q_1 = p_1 + q_1$

(B) $p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 + q_4$

(C) $p_1 = a - b$
$q_1 = p_2 * c$
$p_3 = u * v$
$q_2 = p_4 + q_3$

(D) $p_1 = a - b$
$q_1 = p * c$
$p_2 = u * v$
$q_2 = p + q$

Answer Keys

Exercises

Practice Problems 1

1. D	**2.** D	**3.** C	**4.** C	**5.** B	**6.** A	**7.** B	**8.** A	**9.** A	**10.** A
11. B	**12.** C	**13.** A	**14.** C	**15.** B					

Practice Problems 2

1. B	**2.** B	**3.** A	**4.** B	**5.** B	**6.** A	**7.** B	**8.** C	**9.** A	**10.** A
11. B	**12.** D	**13.** A	**14.** C	**15.** C	**16.** A	**17.** C	**18.** B	**19.** D	

Previous Years' Questions

1. 8	**2.** B	**3.** 10	**4.** D	**5.** B