Unit 5

Q. Explain activation tree and control stack with example?

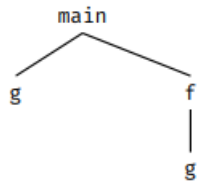
Ans:

Activation Tree:

• Depicts the way control enters and leaves activations
• Root represents the activation of main
• Each node represents activation of a procedure
• Node *a* is the parent of *b* if control flows from *a* to *b*
• Node *a* is to the left of *b* if lifetime of *a* occurs before *b*
• Flow of control in a program corresponds to depth-first traversal of activation tree

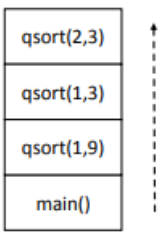
Example:

```
int g() { return 42; }
int f() { return g(); }
main() {
g();
f():
}
```



Control Stack:

• Procedure calls and returns are usually managed by a run-time stack called the control stack
• Each live activation has an activation record on the control stack
• Stores control information and data storage needed to manage the activation
• Also called a frame
• Frame is pushed when activation begins and popped when activation ends
• Suppose node *n* is at the top of the stack, then the stack contains the nodes along the path from *n* to the root



When will a control stack work?

• Once a function returns, its activation record cannot be referenced again
• We do not need to store old nodes in the activation tree
• Every activation record has either finished executing or is an ancestor of the current activation record

When will a control stack not work?

• Once a function returns, its activation record cannot be referenced again
• Function closures – procedure and run-time context to define free variables

****Q. Explain stack allocation with example?**

Ans:

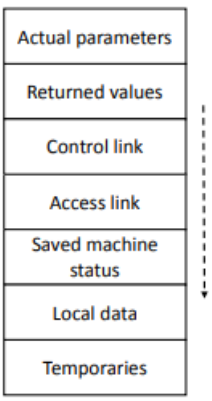
Q. What is activation record? Explain the contents of activation record?

Ans:

▪ Each live activation has an activation record (sometimes called a frame)
▪ The root of activation tree is at the bottom of the stack
▪ The current execution path specifies the content of the stack with the last
▪ Activation has record in the top of the stack.

Activation Record

• Temporary values • Local data
• A saved machine status
• An "access link"
• A control link
• Space for the return value of the called function
• The actual parameters used by the calling procedure



- Fields in an activation record
    - Temporaries – evaluation of expressions
    - Local data – field for local data
    - Saved machine status – information about the machine state before the procedure call
    - Return address (value of program counter)
    - Register contents
    - Access link – access nonlocal data
    - Control link – Points to the activation record of the caller
    - Returned values – Space for the value to be returned
    - Actual parameters – Space for actual parameters
- Contents and position of fields may vary with language and implementations

Q. Explain dynamic storage allocation strategies?

Ans:

Dynamic allocation – Storage allocation decisions are made when the program is running

- Stack allocation –

    Manage run-time allocation with a stack storage

    Local data are allocated on the stack

- Heap allocation –

    Memory allocation and deallocation can be done at any time

    Requires memory reclamation support

Stack vs Heap Allocation

Stack	Heap
• Allocation/deallocation is automatic	• Allocation/deallocation is explicit
• Less expensive	• More expensive
• Space for allocation is limited	• Challenge is fragmentation

Q. Explain following parameter passing methods:

1) Call by value

Ans:

- Convention where the caller evaluates the actual parameters and passes their r-values to the callee
- Formal parameter in the callee is treated like a local name
- Any modification of a value parameter in the callee is not visible in the caller
- Example: C and Pascal

Call-by-value (easy, no special compiler effort)

The arguments are evaluated at the time of the call and the value parameters are copied and either

- behave as *constant values* during the execution of the procedure (i.e., cannot be assigned to as in Ada), or
- are viewed as initialized *local* variables (in C or in Pascal)

Call-by-value

C uses call-by-value everywhere (except macros...)
Default mechanism in Pascal and in Ada

```
callByValue(int y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByValue(x);
    print(x);
}
```

output:
x = 42
y = 43
x = 42

x's value does *not* change when y's value is changed

2) Call by value result

Ans:

Call-by-value-result

The arguments are evaluated at call time and the value parameters are copied (as in call-by-value) and used as a local variables

The final values of these variables are copied back to the location of the arguments when the procedure exits (note that the activation record cannot be freed by the callee!)

Call-by-value-result

Available in Ada for **in out** parameters

(code below in *C* syntax)

```
callByValueResult(int y, int z)
{
    y = y + 1;   z = z + 1;
    print(y);    print(z);
}

main()
{
    int x = 42;
    print(x);
    callByValueResult(x, x);
    print(x);
}
```

output:

x = 42

y = 43

z = 43

x = 43

Note that x's value is *different* from both using call-by-value and call-by-reference

3) Call by name

Ans:

• Reference to a formal parameter behaves as if the actual parameter had been textually substituted in its place

• Renaming is used in case of clashes

• Can update the given parameters

• Actual parameters are evaluated inside the called function

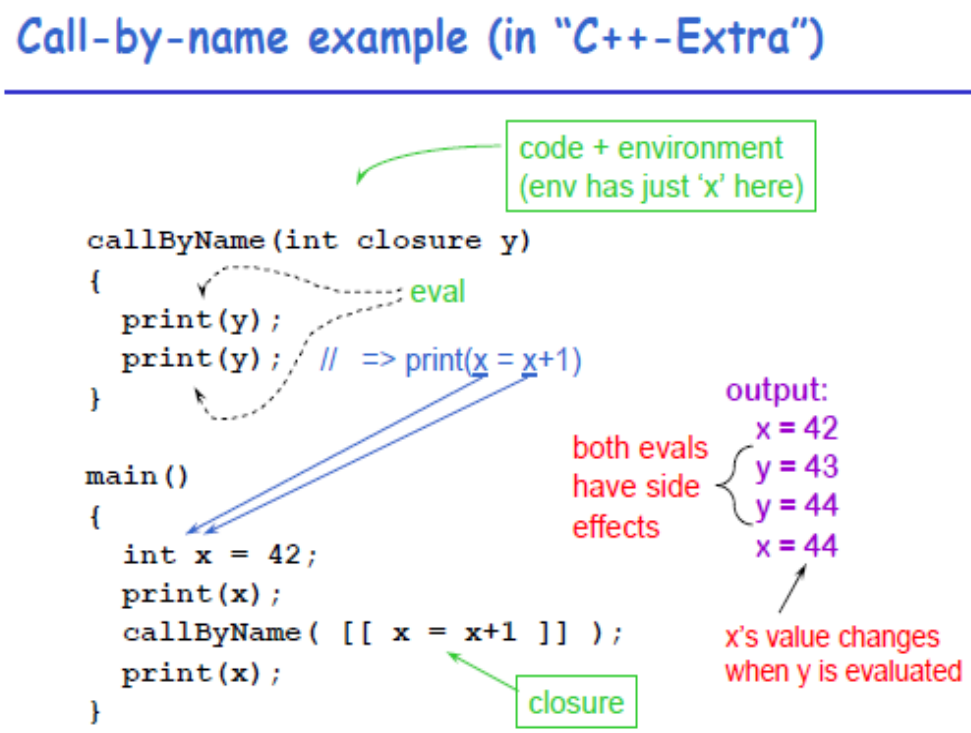
• Example: Algol-60

```
procedure double(x);
  real x;
begin
  x := x*2
end;

double(c[j]) ➡ c[j] := c[j]*2
```

Call-by-name

- Whole different ballgame: it's like passing the *text* of the argument expression, unevaluated
  - The text of the argument is viewed as a function in its own right
  - Also passes the environment, so free variables are still bound according to rules of static scoping
- The argument is not evaluated until it is actually used, *inside* the callee
  - Might not get evaluated at all!
- An optimized version of call-by-name is used in some functional languages (e.g. Haskell, Miranda, Lazy-ML) under the names *lazy evaluation* (or *call-by-need*)



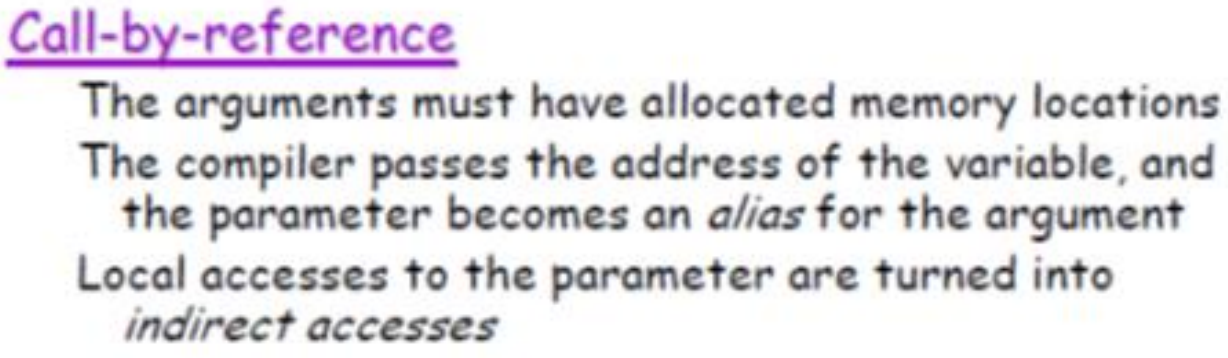
4) Call by reference

Ans:

• Convention where the compiler passes an address for the formal parameter to the callee

• Any redefinition of a reference formal parameter is reflected in the corresponding actual parameter

• A formal parameter requires an extra indirection



Call-by-reference

Available in C++ with the '&' type constructor
(and in Pascal with the **var** keyword)

```
callByRef(int &y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByRef(x);
    print(x);
}
```

output:
x = 42
y = 43
x = 43

x's value changes when y's value is changed

Unit 6

Q. Explain following machine independent transformation techniques:

1) Common sub expression and dead code elimination.

Ans:

Common Sub expressions elimination:

• An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

• For example

t1: =4*i t2: =a [t1] t3: =4*j t4:=4*i t5: =n

t 6: =b [t 4] +t 5

The above code can be optimized using the common sub-expression elimination as

t1: =4*i t2: =a [t1] t3: =4*j t5: =n

t6: =b [t1] +t5

The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
    a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied

2) Copy propagation and constant folding,

Ans:

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example: x=Pi;

……

A=x*r*r;

The optimization using copy propagation can be done as follows:
A=Pi*r*r;
Here the variable x is eliminated

Constant folding:

o We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

o One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Q. Explain how code motion and frequency reduction used for loop optimizations?

Ans:

Code Motion:

• An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop.

For example,

evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change Limit*/ Code motion will result in the equivalent of
t= limit-2;
while (i<=t) /* statement does not change limit or t */

Frequency Reduction:

Q. Explain global data flow analysis using data flow equations?

Ans:

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

out [S] = gen [S] U (in [S] – kill [S])

This equation can be read as “the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.

✓The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

✓Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Q. Explain different code generation issues with example?

Ans:

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

- Intermediate representation can be :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

- Labels in three-address statements have to be converted to addresses of instructions. For example,

j:gotoi generates jump instruction as follows :

- ifi<j, a backward jump instruction with target address equal to location of code for quadrupleiis generated.
- ifi>j, the jump is forward. We must store on a list for quadrupleithe location of the first machine instruction generated for quadruplej.Wheniis processed, the machine locations for all instructions that forward jumps toi are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.

- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

- The quality of the generated code is determined by its speed and size.

- The former statement can be translated into the latter statement as shown below:

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.

- The use of registers is subdivided into two subproblems :

 Register allocation– the set of variables that will reside in registers at a point in the program is selected.

 Register assignment– the specific register that a variable will reside in is picked.

- Certain machine requires even-oddregister pairsfor some operands and results. For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair y – divisor
even register holds the remainder odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

Q. Explain labeling algorithm with example?

Ans:

(1) **if** n is a leaf **then**
(2) **if** n is the leftmost child of its parent **then**
(3) $label(n) := 1$
(4) **else** $label(n) := 0$
else begin /* n is an interior node */
(5) let $c_1, c_2, \cdots, c_k$ be the children of n ordered by $label$
so that $label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$
(6) $label(n) := \max_{1 \leq i \leq k}(label(c_i) + i - 1)$
end

$label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$

If k = 1 (a node with two children), then the following relation

$label(n_1) := \max_{1 \leq i \leq k}(label(c_i) + i - 1)$

becomes:

$$label(n) = \begin{cases} \max[label(c_1), label(c_2)] & \text{if } label(c_1) \neq label(c_2) \\ label(c_1) + 1 & \text{if } label(c_1) = label(c_2) \end{cases}$$

Q. Explain structure preserving transformation techniques with example

Ans:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

1. Common sub-expression elimination:

   Common sub expressions need not be computed over and over again. Instead they can be computed
   once and kept in store from where it's referenced when encountered again – of course providing the
   variable values in the expression still remain constant.

   Example:

   a: =b+c
   b: =a-d
   c: =b+c
   d: =a-d

   The 2nd and 4th statements compute the same expression: b+c and a-d

   Basic block can be transformed to

   a: =b+c
   b: =a-d
   c: =a
   d: =b

2. Dead code elimination

   It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error - correction of a program – once

declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

3. Renaming of temporary variables
   • A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
   • In this we can transform a basic block to its equivalent block called normal-form block.

4. Interchange of two independent adjacent statements.
   Two statements
   t1:=b+c
   t2:=x+y
   can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Q. Explain register allocation and assignment with the suitable example?

Ans:

• Instructions involving register operands are shorter and faster than those involving operands in memory.

• The use of registers is subdivided into two subproblems :

 Register allocation– the set of variables that will reside in registers at a point in the program is selected.

 Register assignment– the specific register that a variable will reside in is picked.

• Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair y – divisor
even register holds the remainder odd register holds the quotient

Q. Explain node listing and labeling algorithm with example?

Ans:

Labeling Algorithm:

(1) **if** n is a leaf **then**
(2) **if** n is the leftmost child of its parent **then**
(3) $label(n) := 1$
(4) **else** $label(n) := 0$
else begin / * n is an interior node * /
(5) let $c_1, c_2, \cdots, c_k$ be the children of n ordered by $label$
so that $label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$
(6) $label(n) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$
end

$label(c_1) \geq label(c_2) \geq \cdots \geq label(c_k)$

If $k = 1$ (a node with two children), then the following relation

$label(n_1) := \max_{1 \leq i \leq k} (label(c_i) + i - 1)$

becomes:

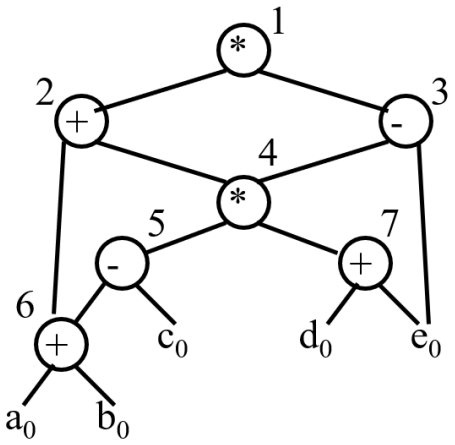
$$label(n) = \begin{cases} \max[label(c_1), label(c_2)] & \text{if } label(c_1) \neq label(c_2) \\ label(c_1) + 1 & \text{if } label(c_1) = label(c_2) \end{cases}$$

Node Listing:

Node Listing Algorithm

while unlisted interior nodes remain **do begin**
select an unlisted node n, all of whose
parents have been listed;
list n;
while the leftmost child m of n has no unlisted
parents and is not a leaf **do begin**
list m;
$n := m$;
end
end

An Example



t7 := d + e
t6 := a + b
t5 := t6 - c
t4 := t5 * t7
t3 := t4 - e
t2 := t6 + t4
t1 := t2 * t3

****Q. Explain principles sources of optimization.**

Ans:

***Q. Discuss in detail about global data flow analysis. (detail?)**

Ans:

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

out [S] = gen [S] U (in [S] – kill [S])

This equation can be read as “the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.

- ✓The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- ✓Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- ✓There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.