Chapter 5 – Implementation Model for Imperative Languages

Introduction to Programming Languages First Edition, 2013

Author: Arvind Bansal

ISBN: 978-146-6565142

Topics Covered

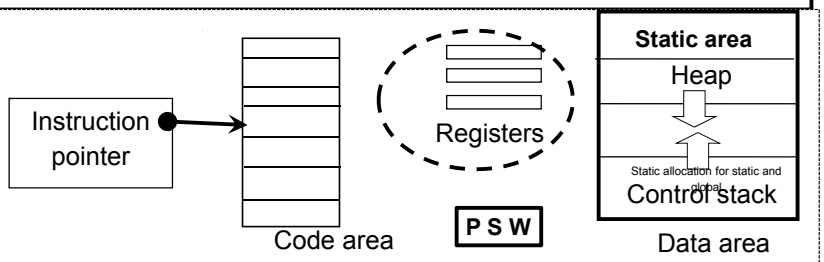
- Introduction
- Abstract Computing Machine
- Translating Control Abstractions
- Static Allocation
- Hybrid Allocation
- Implementing Parameter Passing
- Low Level Behavior of Recursive Procedures
- Implementing Exception Handlers
- Summary

Memory Allocation Schemes

- Static allocation
  - Fixed memory allocation at compile time; no runtime growth
  - Wastage of memory if not used; efficient due to direct memory access
  - Does not support recursive procedures, recursive, and dynamic data structures.
- Stack based allocation
  - Memory is allocated on control stack frame by frame at runtime
  - Supports recursive procedures, but not recursive data structures.
  - Uses pointer based access. Is slower than static allocation
- Heap based allocation
  - Memory allocation on demand in a common memory called heap.
  - Deallocated manually or automatically
  - Supports recursive procedures and recursive data structures
  - Memory recycling overhead up to 30%. Pointer access overhead
- Hybrid allocation
  - Combination of the above three schemes to exploit the advantages

Abstract Computing Machine

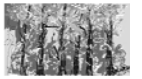
- Consists of data area, code area, instruction pointer, registers, and program status word – a set of flags
  - Code area is a sequence of low level instructions
  - Instruction pointer to step through the instructions in the code area
  - Data area is a pair of the linear structures (*heap*, *control stack*) growing in opposite direction
  - Control stack is a sequence of frames holding the store of procedures
  - Control changes either through IP or by jump statements



Data Area

- Each frame is indexible and contains
  - Local dynamic variables
  - Image of the registers that would be updated in the called subprogram
  - Various pointers to access data-entities and frames
  - The frozen state of the calling subprogram
  - Simple dynamic objects that do not outlive the subprogram
  - Parameters passed to the called subprogram.
- Memory access: direct, indirect, offset based
  - Memory location in a frame are accessed using d[FP + offset]. Indirect memory access is d[d[FP + Offset]].
- Control stack during program invocation
  - Calling programs frame is saved
  - Frame of the called program is placed on top of the stack
  - Various pointers are updated to link the code area and data area of the called subprogram and calling program
  - Code area jumps from the calling program to the called program

Code Area and Memory Access



- Code area is a sequence of low level instructions
- Instructions are accessed using c[index]
- Instructions are of different types
  - Arithmetic operations; logical operations; conditional jumps; unconditional jumps; accessing data from memory; storing data into memory; moving data from one register to another register
- Types of memory access
  - Direct access :- suitable for static variables
  - Indirect access :- uses a pointer, provides independence from fixed locations; overhead of an additional access
  - Offset based access :- used to access memory locations in frame
- Status flags
  - Negation bit, zero bit, overflow bit, Boolean flag are set by various arithmetic and logical instructions

Translating Control Abstractions

- Process
  - Make a control flow diagram of the high level control abstractions
  - Map control flow diagrams to low level instructions using a combination of conditional jumps and low level instructions
- Control flow diagram → low level abstract instructions
  - Control flow diagrams are two dimensional while low level instructions are single dimensional
  - Sequence of contiguous single dimensional statements are connected using conditional branches if needed
  - Registers are used for temporary storage and scratchpad computations

Translating Expressions

- Process involves
  - Reading the variables from the memory locations
  - Evaluating the subexpressions
  - Storing the common subexpressions in a register to avoid reevaluation of the common subexpressions
  - Laws of computation such as associativity, commutativity, and distributivity are used for simplification
- Example: (X + Y + 5) + 2 * (X + Y)

```
load   X, R1
add    Y, R1, R2
add    #5, R2, R3
multiply #2, R2, R4   %   looking up the common subexpression
add    R3, R4, R3
```

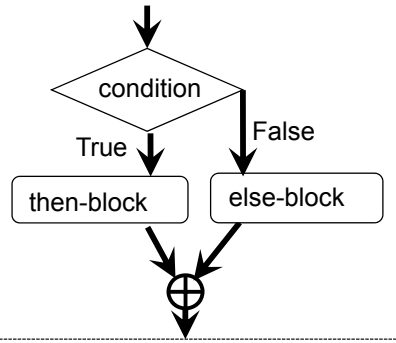
Translating Assignment Statement

- Assignment state includes
  - Evaluation of the right hand side expression
  - Writing into the memory-location of the variable on the left hand side
- Example: x = y + 5

```
load y, R0  % load the value of y in register R0
add # 5, R0, R0  % add 5 to the value in the register R0
store R0, x  % store the value of R0 in the variable X
```


Translating if-then-else Statement

- Process
  - Evaluate the predicate
  - Set the status flags based upon the pedicate evaluation
  - Based upon the status flags branch to execute either the block of statements in then-part or the block of statements in the else-part.

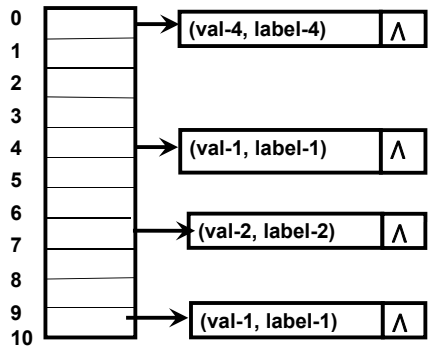


1. evaluate not (<condition>)
2. branch-on-true **else**
3. execute <then-block>
4. jump **exit**
5. **else**: execute <else-block>
6. **exit**:

Translating Case Statement

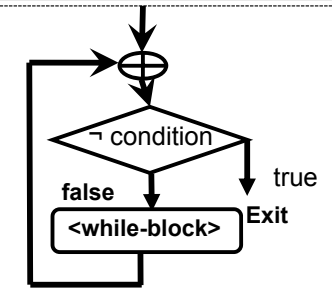
- Can be translated to low level instructions
  - Use a set of if-then-else statements, or
  - Use hash table. Hash table contains a triple of the form (expected-value, label-number, pointer to next triple). After evaluating the expression, the control jumps to the index given by the hash-value of the evaluated expression.

```
value = evaluate(<expression>);
lbl-id = hash(value);
if (lbl-id .ne. null)   jump lbl-id;
else jump default-label
lbl-1: { <block1>; jump exit; }
lbl-2: {<block2>; jump exit; }
…
lbl-4: {<blockN>; jump exit;}
default-label: <default-block>;
<exit>:
```



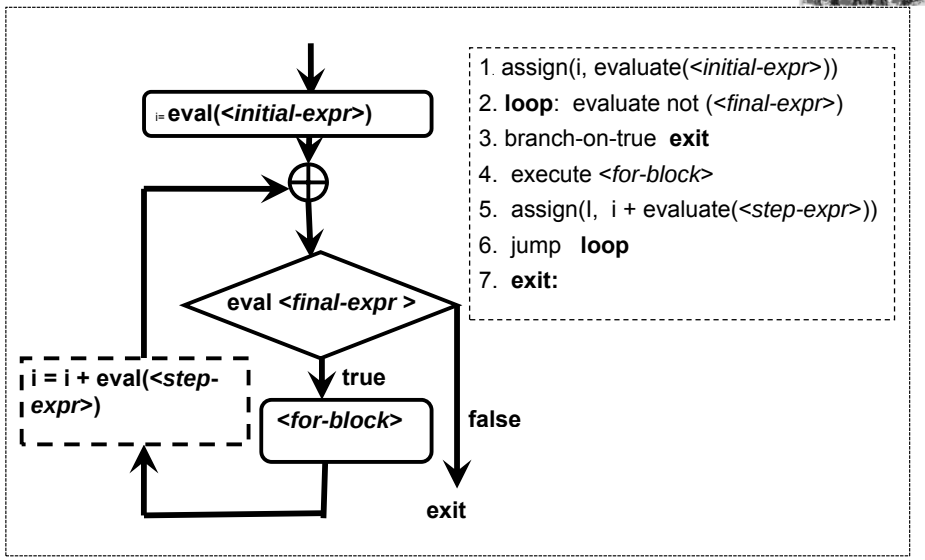
Translating While-loop

- Process
  - Make the control flow diagram using negation of the condition-test.
  - Label the entry point of while-loop and the exit-point.
  - Check for the negation of the condition and jump to the exit on success.
  - Fall through the then part.
  - Add an unconditional jump after then part to the exit-point.



1. **loop**: evaluate not (<condition>)
2. branch-on-true **exit**
3. execute <while-block>
4. jump-to **loop**
6. **exit:**

For-loop Translation



Static Allocation

- Allocation
  - Memory is allocated at compile time, and can not grow
  - Wastes memory space as all the allocated memory allocation may not be used in a single invocation
  - Does not support recursive procedure and recursive data structures
  - **Advantage**: direct memory allocation is efficient
- Data area represented as fixed one dimensional array
  - Data elements are accessed as d[index]
  - Code area is accessed as c[index]
- Program invocation
  - A return pointer that stores the next instruction in the calling program to be executed is stored in the activation record of the called procedure.
  - Unconditional jump to jump to first instruction of the called program
  - Return from the subroutine takes the location from the return pointer, and makes unconditional jump.

Example – Static Allocation

```
PROGRAM MAIN
DIMENSION M[4]
INTEGER I, J, Max
COMMON /DATA/ M[4], MAX
D0  20  I = 1, 4, 1
20 READ(M[I])
CALL FIND_MAX
END
```

```
0.  assign(d[5], 1)
1.  cmp(d[5], 4)
2.  brgt (ip + 4) % ip = 6
3.  read(d[0 + d[5]])
4.  assign(d[5],  d[5] + 1)
5.  jump(ip – 4)
6.  assign(d[<R>], ip + 2)
7.  jump(ip + <S>)
8.  halt
```

0	M[1]
1	M[2]
2	M[3]
3	M[4]
4	MAX
5	I
6	J

```
SUBROUTINE FIND_MAX
DIMENSION M[4]
INTEGER MAX, I
COMMON /DATA/, M[4], MAX
MAX = M[1]
DO 10  I = 2, 4, 1
10  IF (M[I] > MAX)  MAX = M[I]
RETURN
```

```
0.  assign(d[4], d[0])
1.  assign(d[6], 2)
2.  cmp(d[6], 4)
3.  brgt (ip + 6)  % ip = 9
4.  cmp(d[0 + d[6]], d[4])
5.  brle (ip + 2) % ip = 7
6.  assign(d[4], d[0 + d[6]])
7.  assign(d[6], d[6] + 1)
8.  jump(ip – 6)
9.  jump(d[5])
```

0	M[1]
1	M[2]
2	M[3]
3	M[4]
4	MAX
5	return
6	I

Linking Compiled Code

- Multiple program units are compiled separately
- Separately compiled units lack
  - Information about where to jump when a subprogram is called
  - How much to jump when a subprogram is called
- Program linking joins the
  - Code areas of the complied programs into one single large code area
  - Data areas of the compiled programs into one single large data area
  - The order of compiled code is independent of the calling order of the programs. Rather it depends on the order in the linking commands
- Effect of linking
  - The return location and the location of the first instruction is the called routine is known, and filled in
  - The memory location of variables of the subprograms in the linked programs gets shifted
  - Common area is put first followed by the sequence of remaining activation records: <**common**, (R_1, L_1), …, (R_N, L_N)>

Code After Linking

Linked code area

```
0.   assign(d[5], 1)
1.   cmp(d[5], 4)
2.   brgt(IP + 3)
3.   read(d[0 + d[5]])
4.   d[5] = d[5] + 1
5.   jump(ip – 4)
6.   assign(d[7], IP + 2)
7.   jump(IP + 2)
8.   halt
9.   assign(d[4], d[0])
10.  assign(d[8], 2)
11.  cmp(d[8], 4)
12.  brgt (IP + 5)
13.  cmp(d[0 + d[8]], d[4])
14.  brle(IP + 2)
15.  d[4] = d[0 + d[8]]
16.  jump(IP – 5)
17.  jump(d[7])
```

Linked data area

0	M[1]
1	M[2]
2	M[3]
3	M[4]
4	MAX
5	I-main
6	J
7	Return- sub
8	I-sub

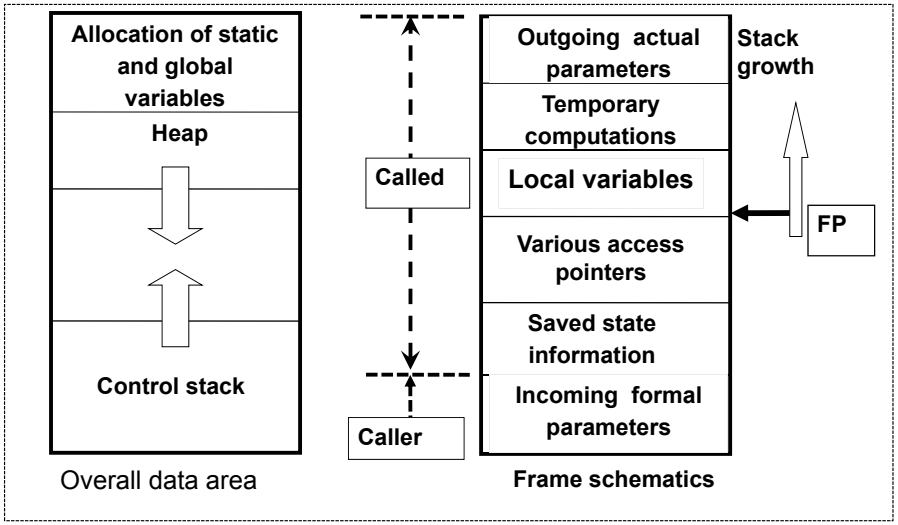
Common Area

Common area has been merged , and subroutine's remaining activation-record has been shifted resulting into update of the code area

Hybrid Allocation

- Hybrid allocation combines various allocation schemes
  - Static memory that remains fixed during runtime, and is efficiently accessed using direct access. Static variables can be local or global
  - Frames allocated on the stack when a subprogram is invoked, and released when the subprogram is over
  - Dynamic data structures in heap
- Frame contains
  - Local dynamic stack variables
  - Saved state information of the calling program such as pointers, registers to be modified in the called programs
  - Incoming parameter space and outgoing parameter space
  - Space for scratchpad computation
- Pointers
  - 1) frame pointer, 2) dynamic link, 3) static link, 4) return pointer, 5) top of the stack pointer
- Incoming parameter space of the called subprogram and the outgoing parameter space of the calling programs overlap

Data Area – Block Structured Language



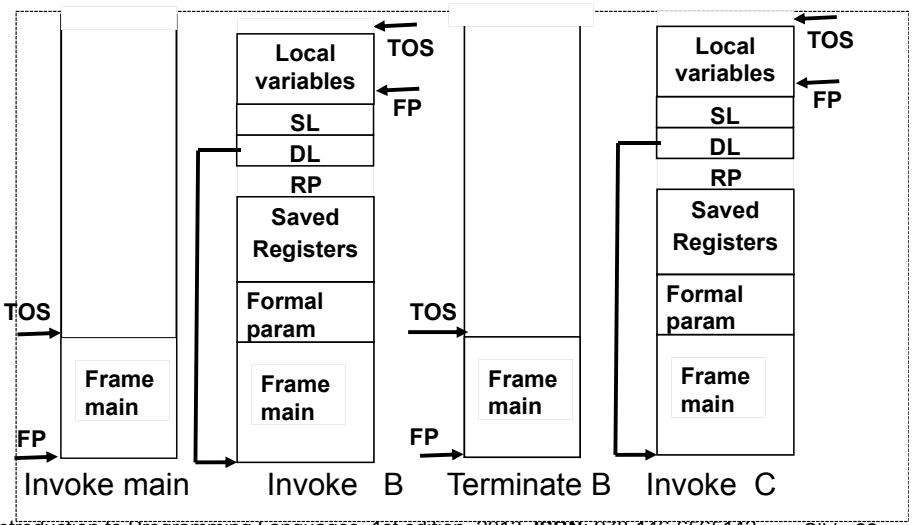
Roles of Various Pointers

- Current Frame pointer (FP)
  - Used to access the data area of the currently executing subprogram.
  - Points to the first memory location of the frame.
- Top of the stack pointer (ToS)
  - Used to point to the first free location in the stack.
- Return Pointer (RP)
  - Used to access the next instruction of the calling subprogram after return from the called subprogram.
  - Stored in the frame of the called subprogram since the same subprogram can be called multiple times.
- Dynamic Link (Previous Frame Pointer)
  - Points to the first memory location of the frame of the calling subprogram
  - Used to return the access of the frame of the calling subprogram back
- Static link (SL)
  - Points to the frame of the subprogram under which currently executing subprogram is nested to access the nonlocal dynamic variables

Calling Subprograms

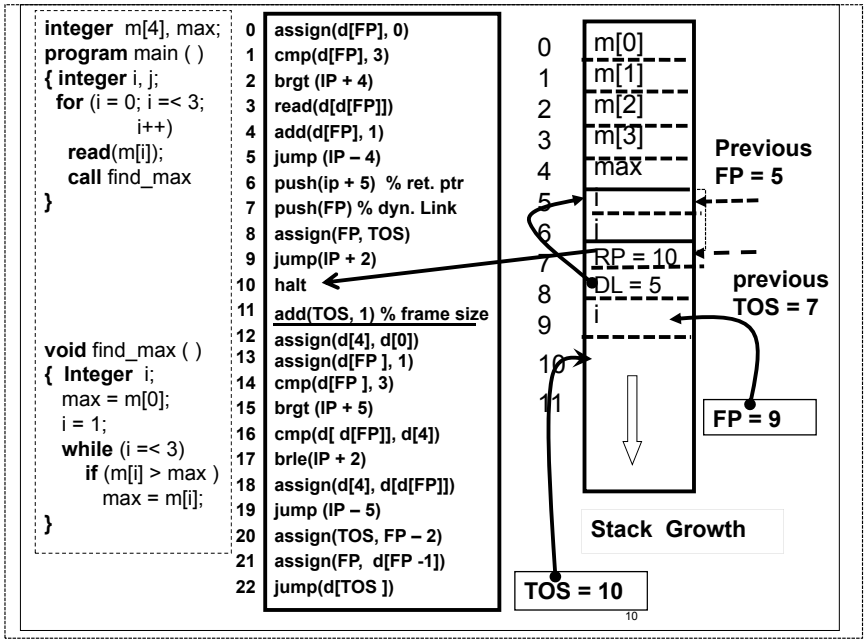
- Program invocation
  - Exchange the information between the calling and the called subprogram.
  - Save the PSW and registers to be modified.
  - The memory location of next instruction is pushed the control stack.
  - Calling program's frame pointer is pushed in the control stack.
  - The static link is set and pushed in the control stack.
  - The current TOS value is copied in the frame pointer.
  - IP is set to the location of the first instruction in the called subprogram.
  - TOS is incremented by activation record size in the called subprogram
- Return from the subprogram
  - Copy the saved registers back of the calling subprogram.
  - Set the FP back to the previous frame pointer using dynamic link.
  - Reclaim the data area of the called subprogram by resetting the TOS pointer back to the TOS of the calling subprogram's TOS.
  - Set IP to the value stored in the return pointer to jump back to the next instruction in the calling program.
  - Copy the parameter values from outgoing parameter locations.

Frames During Procedure Calls



Generating Data and Code Areas

- Individual programs are compiled to generate the low level compiled code with embedded activation record.
- During linking code area is joined together in the order of user specification to make a bigger code area.
- The information about the return pointer and the first instruction pointer is set up from symbol table during linking process.
- Access to data area is embedded in the code area.
- During program invocation frame of the called program is placed on top of the stack.
- After program termination, the memory location of the frame is recovered.
- The offset of the pointers RP, DL, SL are negative.
- FP points to the first memory location where the local variables start.

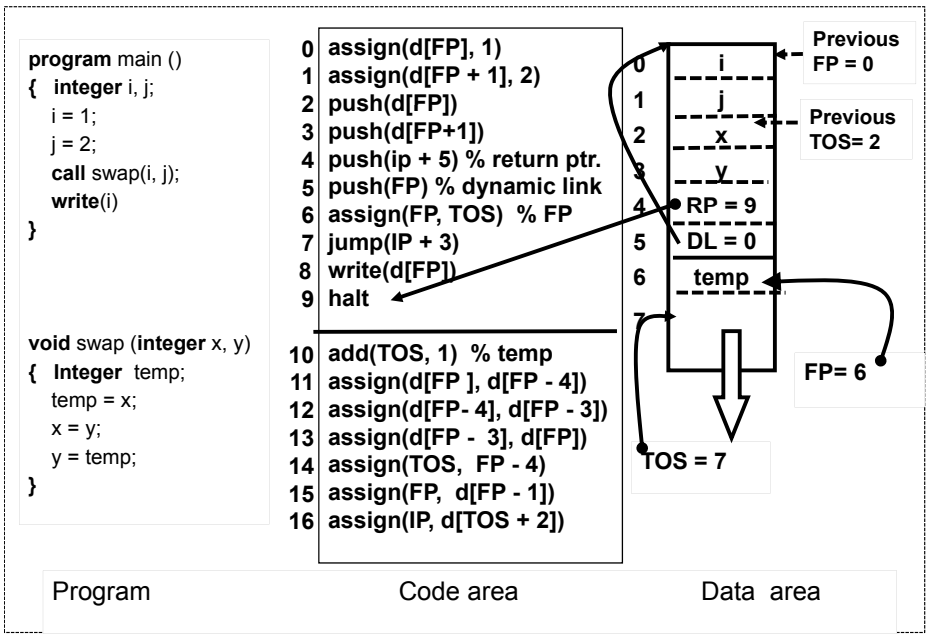


Implementing Parameter Passing

- Outgoing parameter space of the calling subprogram and incoming parameter space of the called subprogram overlap
  - Calling program writes into its outgoing parameter space
  - Called program reads and updates its incoming parameter space
- Formal parameters are allocated in incoming parameter space of the called program
  - Call by value, call by result and call by value-result creates local variables for formal parameters in incoming parameter space
  - Call by reference creates pointers as formal parameters in the incoming parameter space
- Offset scheme
  - The offset of the formal parameter space , the saved register space and the pointers is negative.
  - The offset of locally declared variables and scratch pad computation space is positive

Implementing Call by Value

- The expression is evaluated and copied into the outgoing parameter space of the calling program
- The outgoing parameter space of the calling program becomes incoming formal parameter space of the called subprogram
  - Actual parameter is copied into formal parameter space using a statement assign([FP + offset1], [FP + offset2]) where offset1 is the offset of the actual parameter and offset2 is the offset of the outgoing parameter location.
- Offset of the parameter space
  - The offset of the formal parameter space in called subprogram is negative with respect to the frame pointer.
  - The access to formal parameters is d[FP + Offset] where offset is negative.

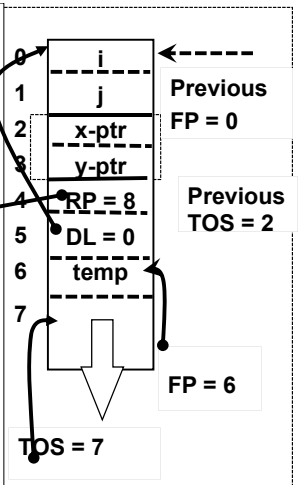


Implementing Call-by-Reference

- A pointer is created in the outgoing parameter space of the calling procedure that
  - Points to the first memory location of the actual parameter.
  - Elements in actual parameters are accessed using d[d[FP + $offset_1$] + $offset_2$] where $offset_1$ is negative and $offset_2$ is positive.
  - $Offset_1$ is negative because the formal parameter is allocated before the activation record.
  - $Offset_2$ is positive as it is used to access the fields or elements in the actual parameter that is part of the activation record of the calling subprogram.
- Setting up of the formal parameter
  - Formal parameter is set by the statements assign(d[FP + $offset_1$], FP + $offset_2$) where $offset_1$ is the offset of the outgoing parameter and $offset_2$ is the offset of the actual parameter in the frame of the calling program.
  - Formal parameters are set in the calling program.
- There is no need to copy any result back

```
program main ( )
{  integer i, j;
   i = 1;
   j = 2;
   swap(& i, & j);
   write(i)
}

void swap (integer
*x, *y)
{  integer  temp;
   temp = * x;
   *x = *y;
   *y = temp;
}
```

```
0  assign(d[FP], 1)
1  assign(d[FP + 1], 2)
2  push(FP)
3  push(FP +1)
4  push(ip + 5) % return pointer
5  push(FP) % dynamic link
6  assign(FP, TOS)
7  jump(IP + 3)
8  write(d[FP])
9  halt

10 add(TOS, 1) % add frame-size
11 assign(d[FP ], d[d[FP-4]])
12 assign(d[d[FP-4], d[d[FP-3]])
13 assign(d[d[FP-3]], d[FP])
14 assign(TOS, FP – 4)
15 assign(FP,  d[FP - 1])
16 jump(d[TOS + 2])
```



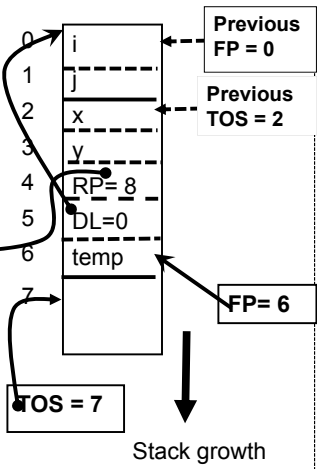
Program Code area Data area

Implementing Call by Value-Result

- The expression is evaluated and copied into the outgoing parameter space of the calling program
- The outgoing parameter space of the calling program becomes incoming formal parameter space of the called subprogram
  - Actual parameter is copied into formal parameter space using a statement assign([FP + $offset_1$], [FP + $offset_2$]) where $offset_1$ is the offset of the actual parameter and $offset_2$ is the offset of the outgoing parameter location
- The offset of the formal parameter space in called subprogram is negative with respect to the frame pointer.
- The access to formal parameters is d[FP + Offset] where offset is negative.
- After coming from the called subprogram
  - Formal parameter is copied into actual parameter using a statement assign([FP + $offset_1$], [FP + $offset_2$]) where $offset_1$ is the offset of the formal parameter in outgoing parameter space and $offset_2$ is the offset of the actual parameter.


```
program main ( )
{integer i, j;
   i = 1;
   j = 2;
   swap(# i, # j);
   write(i);
}

void swap (integer x, y)
{  integer temp;
   temp = x;
   x = y;
   y = temp;
}
```

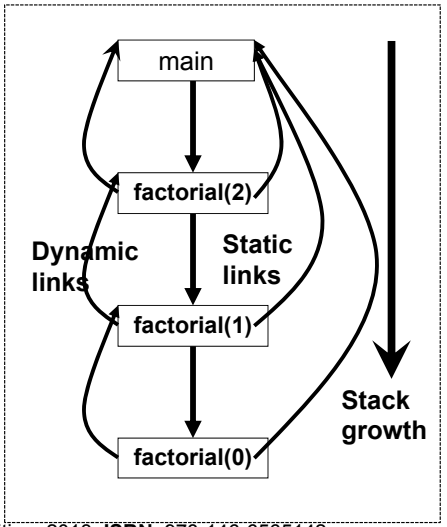
```
0  assign(d[FP], 1)
1  assign(d[FP + 1], 2)
2  push(d[FP])
3  push(d[FP+1])
4  push(IP + 5)  % return ptr.
5  push(FP)  % save dyn. Link
6  assign(FP, TOS)  % new FP
7  jump(IP +  5)  % jump to swap
8  assign(d[FP], d[TOS])
9  assign(d[FP+1], d[TOS + 1])
10 write(d[FP])
11 halt

12 add(TOS, 1)  % frame-size
13 assign(d[FP ], d[FP- 4)
14 assign(d[FP- 4], d[FP- 3])
15 assign(d[FP- 3], d[FP])
16 assign(TOS,  FP – 4)
17 assign(FP,  d[FP  - 1])
18 assign(IP, d[TOS + 2])
```



Program Code area Data area

Behavior of Recursive Procedures

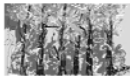
- Every invocation creates a new frame on top of the stack
- There is no access of the environment of multiple invocations of recursive procedures
- Static link of all invocation of a self recursive procedure point to the subprogram under which it is nested
- Dynamic link of a of a self recursive procedure points to the frame of previous invocation
- Recursive procedures have additional overhead of
  - Saving computational state of previous invocation



Implementing Exception Handlers

- Exception handling techniques
  - First technique is using the same control stack, exception flag and the use of branching instructions
  - Second technique is to keep a separate exception stack
- First technique
  - Use an exception flag that is set in case of the exceptional condition
  - After the exception flag is set the control returns to the calling subprogram with exceptional handler, and exception handler is executed
  - If exception flag is not set then a branch instruction is used to skip the exception handler
  - After the successful execution of an exception handler, the control remains in the same subprogram with exception handler
- Second technique
  - The address of the first instruction to the exception handler and the topmost activation frame is pushed on the execution stack.
  - After executing the exception-handler, all the frames above the frame of the successful exception handler are removed from the control stack.

Summary I



- Abstract machine contains code area, data area instruction pointer, and registers
- High level instructions are transferred to a sequence of low level instructions
- Conditional and iterative constructs are transferred using the control flow diagram and translation to low level instructions
- Memory allocation scheme is a combination of static allocation, stack based dynamic allocation, and heap based dynamic allocation
  - Static allocation has advantage of direct access but does not support recursive procedures and recursive data structures
  - Stack based allocation supports memory reuse and recursive procedures. However, does not support recursive and dynamic data structures
  - Heap based allocation is most general. However, it has overhead of memory recycling
- Block structured languages are implemented using hybrid allocation

Summary II

- Hybrid allocation uses
  - Static allocation for static local variables and global variables
  - Uses stack for dynamic local variables and saved computational state
  - Heap for the recursive data structures and dynamic objects
- Parameter passing implementation
  - Outgoing parameter space of the calling program is the same as incoming parameter space of the called program
  - The offset of the incoming parameter space and the saved computational state on the stack is negative
  - The offset of the local dynamic variable of the called subprogram is positive
  - Call by value and call by value-result evaluates the expression, and copies the value from the evaluated expression to outgoing parameter space
  - Call by reference copies the location of the actual parameter to the outgoing parameter space