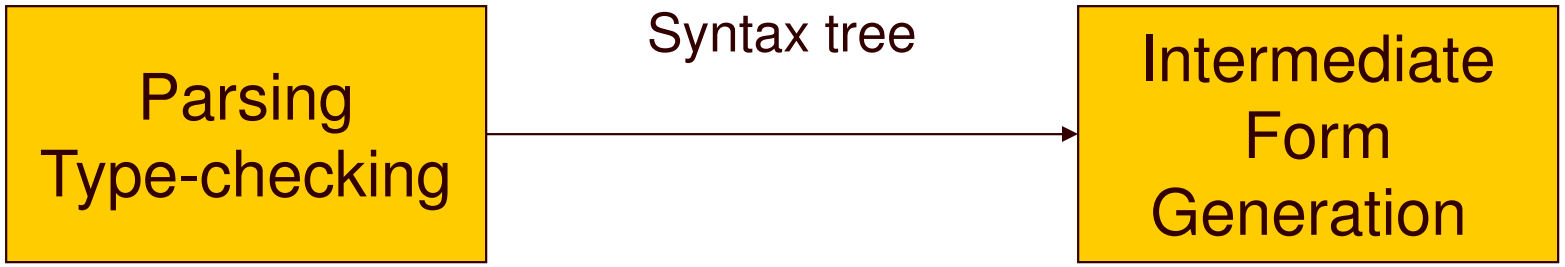
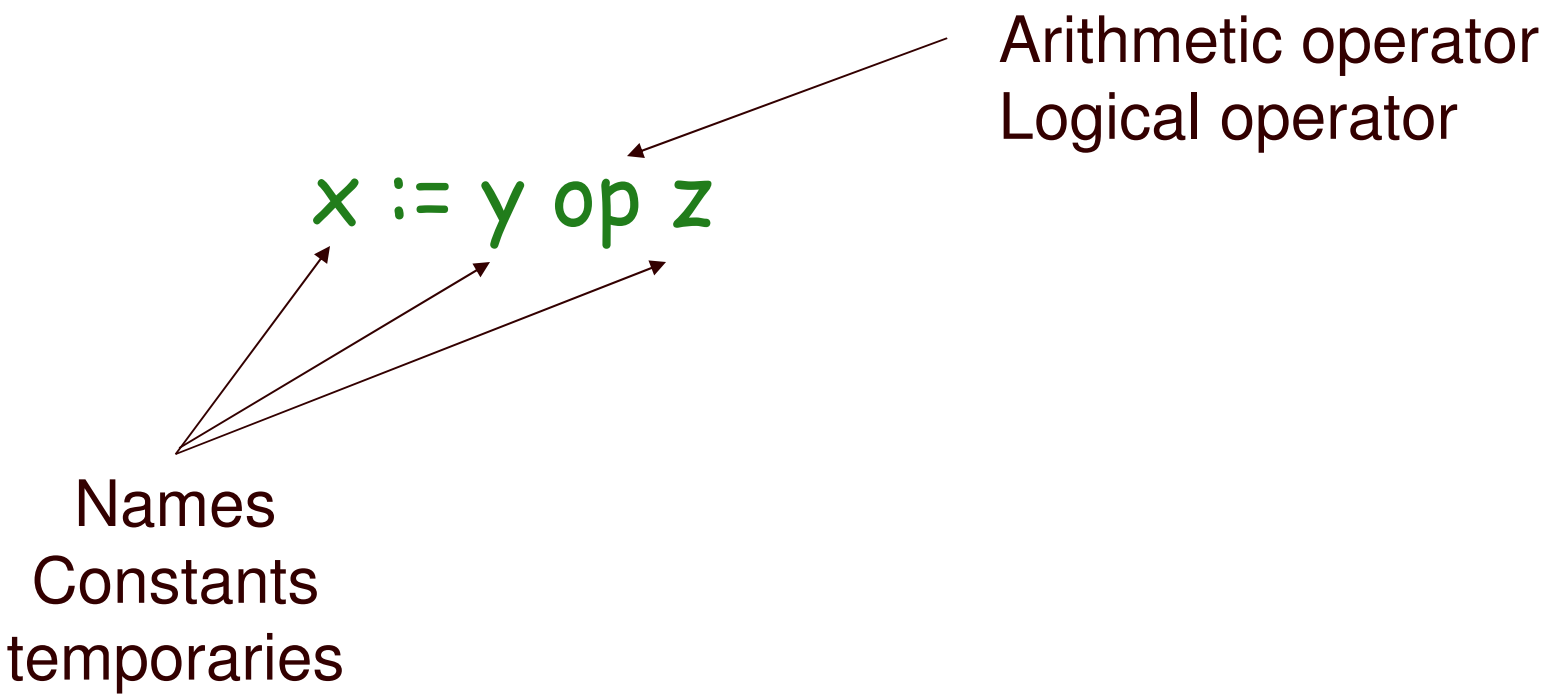
Intermediate Code Representation



Assumption: Source code is parsed and type-checked

TAC storage representation

- It is a sequence of statements of the general form



3-address code

/* source language expression */

x + y * z

/* 3-address statements */

t1 := y * z
t2 := x + t1

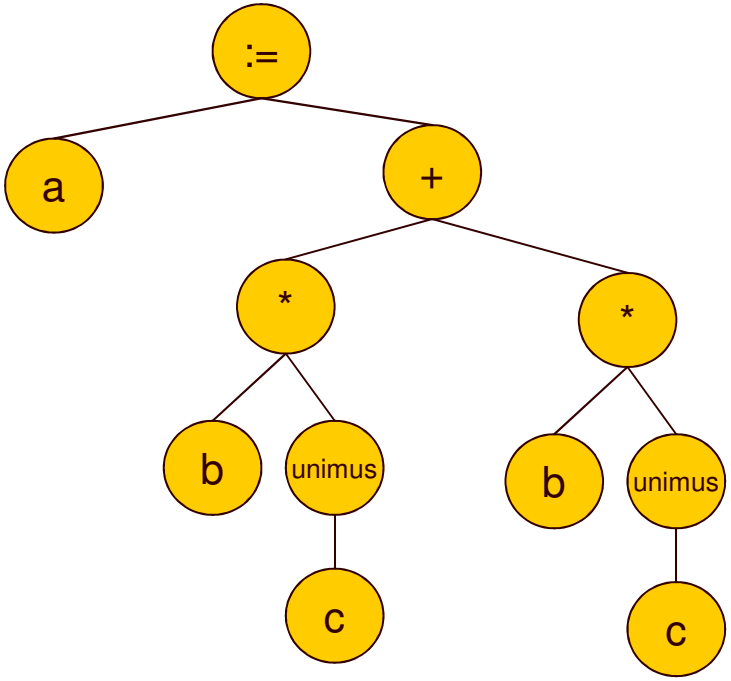
Where t1, t2 are compiler-generated temporary names

Variable names can appear directly in 3-address statements

3-address code

/* source language expression */

a := b * -c + b * -c



/* 3-address statements */

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

In other words, for each interior node in the syntax tree we create a new temporary variable

Types of 3-address statements

- Assignment statement
  x := y op z
- Assignment instruction
  x := op z
- Copy statement
  x := y
- Unconditional jump
  goto L
- Conditional jump
  if x relop y goto L

Procedural call

```
p(x1,x2,…,xn)
  param x1
  param x2
  param xn
  call p, n
```

Indexed assignments

x := y[i] , x[i] := y

Address and pointer assignments

x := &y, x := *y, *x := y

Implementation of 3-address

- 3-address statement is an abstract form of intermediate code
- Need a concrete form
  - Records with fields for operator and operands
    - Quadruples
    - Triples
    - Indirect triples

Quadruples

- It is a record structure with 4 fields
  - Operator
  - Argument 1
  - Argument 2
  - Result

a:= b * -c + b * -c

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

op	arg1	arg2	result
unimus	c		t1
*	b	t1	t2
unimus	c		t3
*	b	t3	t4
+	t2	t4	t5
:=	t5		a

Triples

- It is a record structure with 3 fields
  - Operator
  - Argument 1 (pointer to symbol table or triple)
  - Argument 2 (pointer to symbol table or triple)

– Idea
  – avoid entering temporary names to symbol table
  – Refer to a temp value by the position of that statement that computes it

a:= b * -c + b * -c

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

op	arg1	arg2
unimus	c	
*	b	(0)
unimus	c	
*	b	(2)
+	(1)	(3)
assign	a	(4)

Indirect Triples

- It is a record structure with 3 fields
  - Operator
  - Argument 1 (pointer to symbol table or triple)
  - Argument 2 (pointer to symbol table or triple)

– Idea
  - List pointers to triples instead listing triples

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

op	arg1	arg2
unimus	c	
*	b	(14)
unimus	c	
*	b	(16)
+	(15)	(17)
assign	a	(18)

In-class Exercise

- Show the 3-address code for the following statements:

  a := 5 + 6

  c := a + 3

- Try to optimize the 3-address code generated

Role of Storage in Compilers

- Source language issues
  - Activation trees, control stacks, scope of declaration, bindings
- Storage Organization
  - Activation records
- Storage Allocation Strategies
  - Static allocation, stack allocation, heap allocation

Source Language Issues

- Before considering code generation
  - Need to relate static program source to actions that must take place at runtime to implement this program
    - Relationship between names and data objects
    - Allocation and deallocation of data objects is managed by runtime support package
      - Routines loaded with the generated target code
    - Depending on the language, the runtime environment may be
      - Fully static, stack-based or dynamic
      - Type of environment determines the need to use stack, heap or both

Program and Procedures

Procedure definition

procedure name associated with procedure body

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
  FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
  PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Formal parameters

Procedure call

Actual parameters

Activation Record

Program and Procedures

Procedure Activation

An execution of a procedure body

Lifetime of Procedure Activation

Sequence of steps between the first and last statements in the execution of the procedure body, including time spent executing other procedures called by p

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Formal parameters

Actual parameters

Activation Record

Flow of control

- Assumptions about the flow of control among procedures during program execution
  - Control flows sequentially
  - Execution of a procedure starts at the beginning of procedure body and eventually returns control to the point immediately following the place where the procedure was called
    - In this case we can use trees to illustrate the flow of control in a program
    - goto statement (transfers control to another point)

Procedure Activation

- Each time control enters a procedure q from procedure p, it eventually returns to p (if no errors occur)
  - Activation of procedure p
  - Activation of procedure q
  - Return to activation of procedure p
- Consider
  - lifetime(p) and lifetime(q)
    - Lifetime(p) and lifetime(q) are non-overlapping
    - Lifetime(p) and lifetime(q) are nested
      - See example above

Recursive Procedures

- A recursive procedure is a procedure that calls itself
  - A new activation begins before an earlier activation of the same procedure has ended
  - Several activations for procedure p exist at time t
- A recursive procedure is a procedure p that calls another procedure q, which in turns calls procedure p
  - Procedure p
    - Call to procedure q

Procedure q

Call to procedure p

Activation tree

- We can use a tree (activation tree) to depict the way control enters and leaves activations
- How to build an activation tree
  - Each node represents an activation of a procedure
    - The root represents the activation of the main program
    - The node for a is the parent of node b iff control flows from activation a to b
    - The node for a is to the left of the node for b iff the lifetime(a) occurs before lifetime(b)

Activation Tree for a program

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Can you build the activation tree for this program?

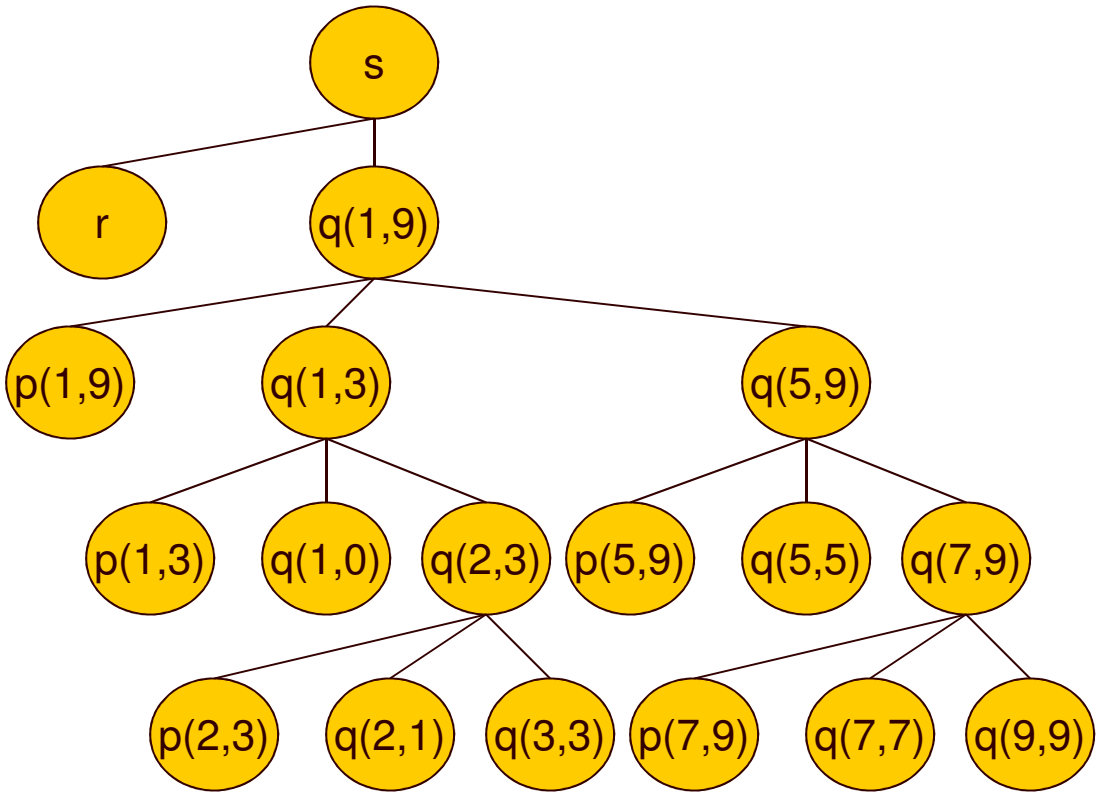
Use
s - activation of main program sort
r - activation of readarray
p - activation of partition
q(x,y) - activation of quicksort

Start with
s as the root of the tree
r and q(1,9) are its children, etc…

Activation Record

Activation Tree for a program

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

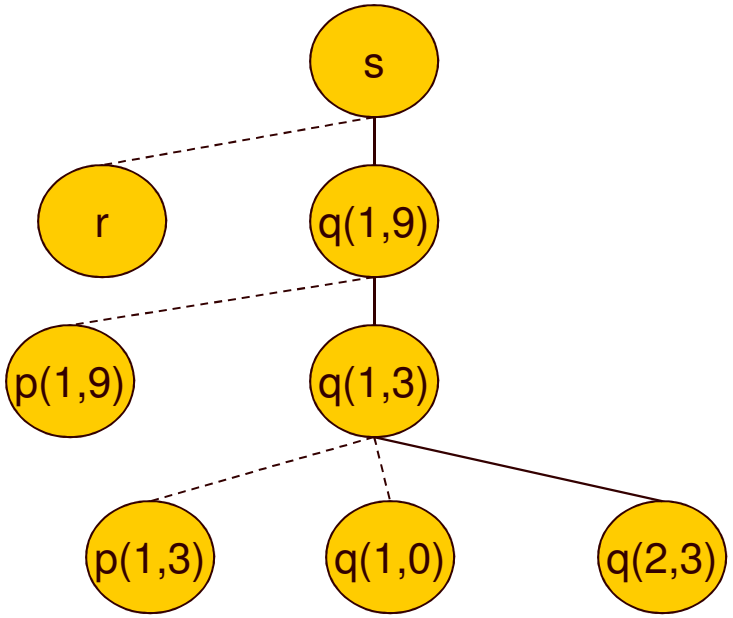


Activation Record

Control Stack and Activation Tree

- Flow of control in a program
  - Depth-first traversal of the activation tree
  - Use a stack called control stack to keep track of the live procedure activations
    - Push node for an activation onto the stack as the activation begins
    - Pop node for an activation off the stack when the activation ends
    - Control Stack contents
      - Node n is at top of the control stack
      - Stack contains the nodes along the path from n to the root

Activation Tree for a program



r, p(1,9), p(1,3) and q(1,0) have executed to completion (dashed lines)

Stack contains

q(2,3) ← top
q(1,3)
q(1,9)
s

Scope of declaration

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Scope of variables depends on the language scope rules
(1) Local
(2) Nonlocal

Same name, different scope

Bindings of names

- Environment
  - Function that maps a name to a storage location
    - f: name $\rightarrow$ storage location
- State
  - Function that maps a storage location to a value
    - g: storage location $\rightarrow$ value

g(f(name)) = value

Bindings of names

- Consider storage location 100 is associated with variable x and it currently holds value 0
  - Environment
    - x is bound to storage location 100
  - State
    - Value held is 0
- Consider assignment statement x:=10
  - Environment
    - x is still bound to storage location 100
  - State
    - Value held is 10

Bindings of names

- A binding is the dynamic counterpart of a declaration
  - Pascal local variable name in a procedure is bound to a different storage location in each activation of a procedure

STATIC NOTION	DYNAMIC COUNTERPART
Definition of procedure	Activations of the procedure
Declaration of name	Bindings of the name
Scope of declaration	Lifetime of binding

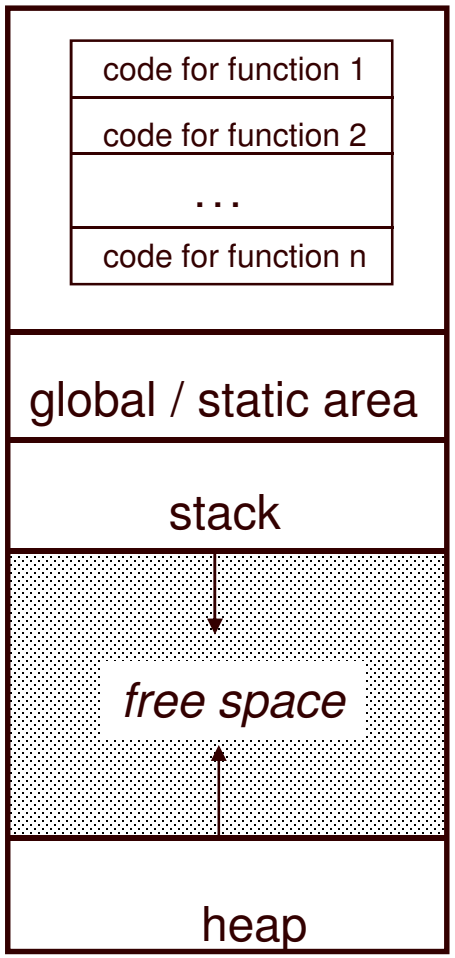
Lecture Outline

- Storage Organization
  - Activation records

Storage Organization

- Suppose that the compiler obtains memory from the OS so that it can execute the compiled program
  - Program gets loaded on a newly created process
- This runtime storage must hold
  - Generated target code
  - Data objects
  - A counterpart of the control stack to keep track of procedure activations

Runtime Memory



PASCAL and C use extensions of the control stack to manage activations of procedures

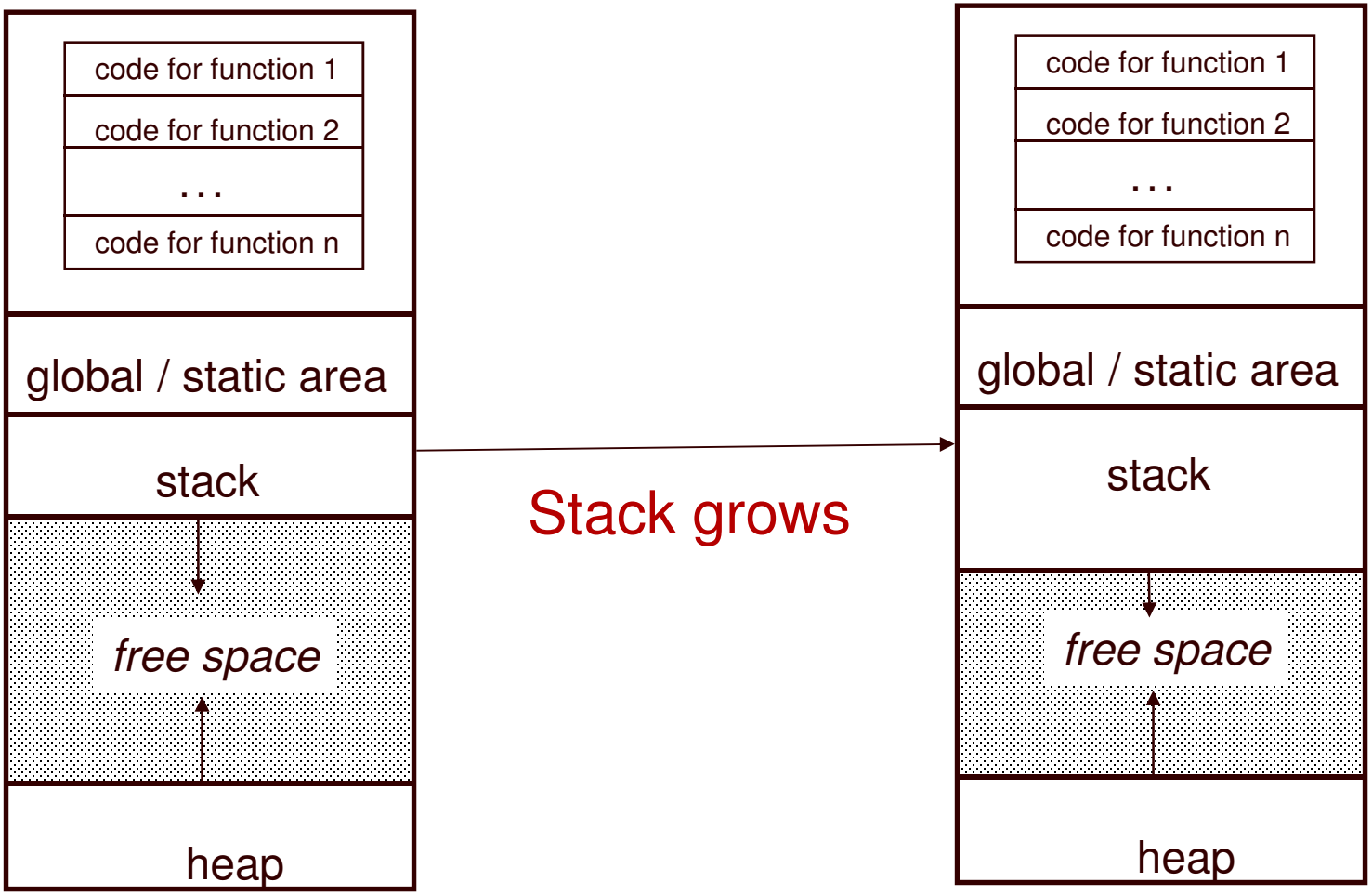
Stack contains information about register values, value of program counter and data objects whose lifetimes are contained in that of an activation

Heap holds all other information. For example, activations that cannot be represented as a tree.

By convention, stack grows down and the top of the stack is drawn towards the bottom of this slide (value of top is usually kept in a register)

Activation Record

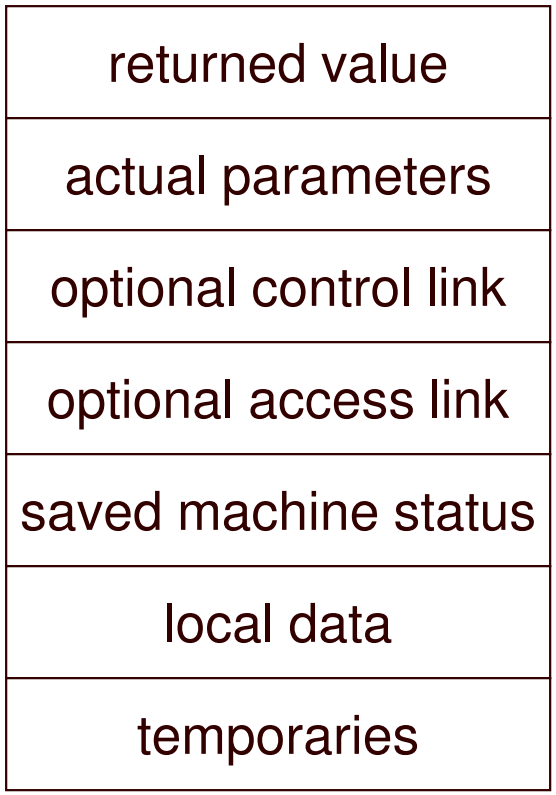
Runtime Memory



Activation Record

Activation Record

- Information needed by a single execution of a procedure is managed using an activation record or frame
  - Not all compilers use all of the fields
  - Pascal and C push activation record on the runtime stack when procedure is called and pop the activation record off the stack when control returns to the caller



Activation Record

1) Temporary values

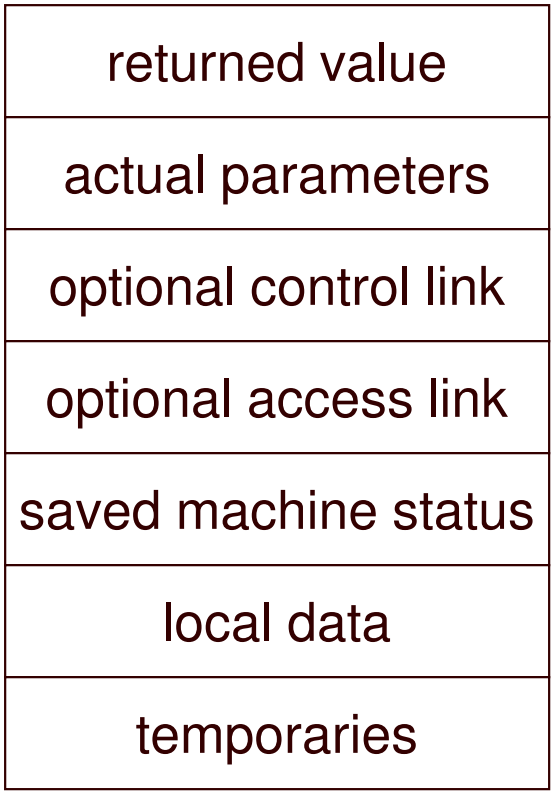
   e.g. those arising in the evaluation of expressions

2) Local data

   Data that is local to an execution of the procedure

3) Saved machine status

   State of the machine info before procedure is called. Values of program counter and machine registers that have to be restored when control returns from the procedure



Activation Record

4) Access Link

   refer to non-local data held in other activation records

5) Control link

   points to the activation record of the caller

6) Actual parameters

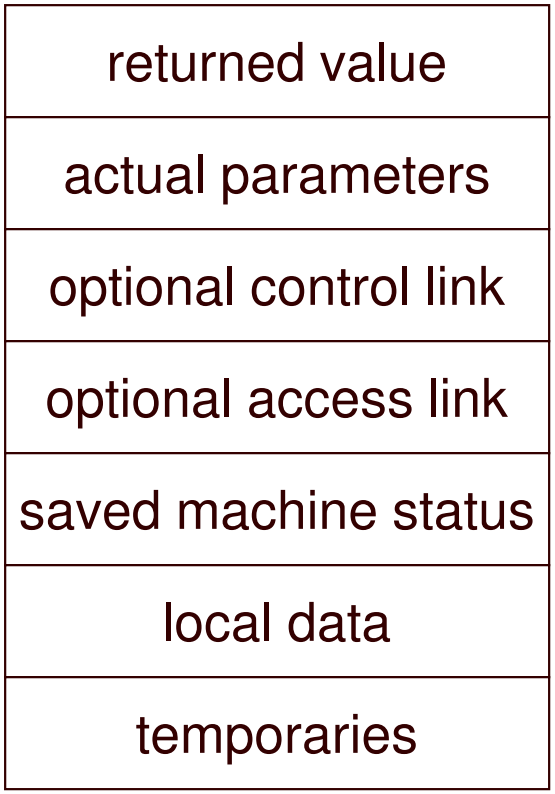
   used by the calling procedure to supply parameters to the called procedure

   (in practice these are passed in registers)

7) Returned value

   used by the called procedure to return a value to the calling procedure

   (in practice it is returned in a register)



Local data

- The field for local data is set when declarations in a procedure are examined during compile time
  - Variable-length data is not stored here
  - Keep a count of the memory locations that have been allocated so far
  - Determine a relative address (offset) of the storage for a local with respect to some position (e.g. beginning of the frame)
    - Multibyte objects are stored in consecutive bytes and given the address of the first byte

Lecture Outline

- Storage Allocation Strategies
  - Static allocation, stack allocation, heap allocation

Storage-allocation strategies

- There are three storage allocation strategies:
  - Static allocation
  - Stack-based allocation
  - Dynamic (or heap-based) allocation

Static Allocation

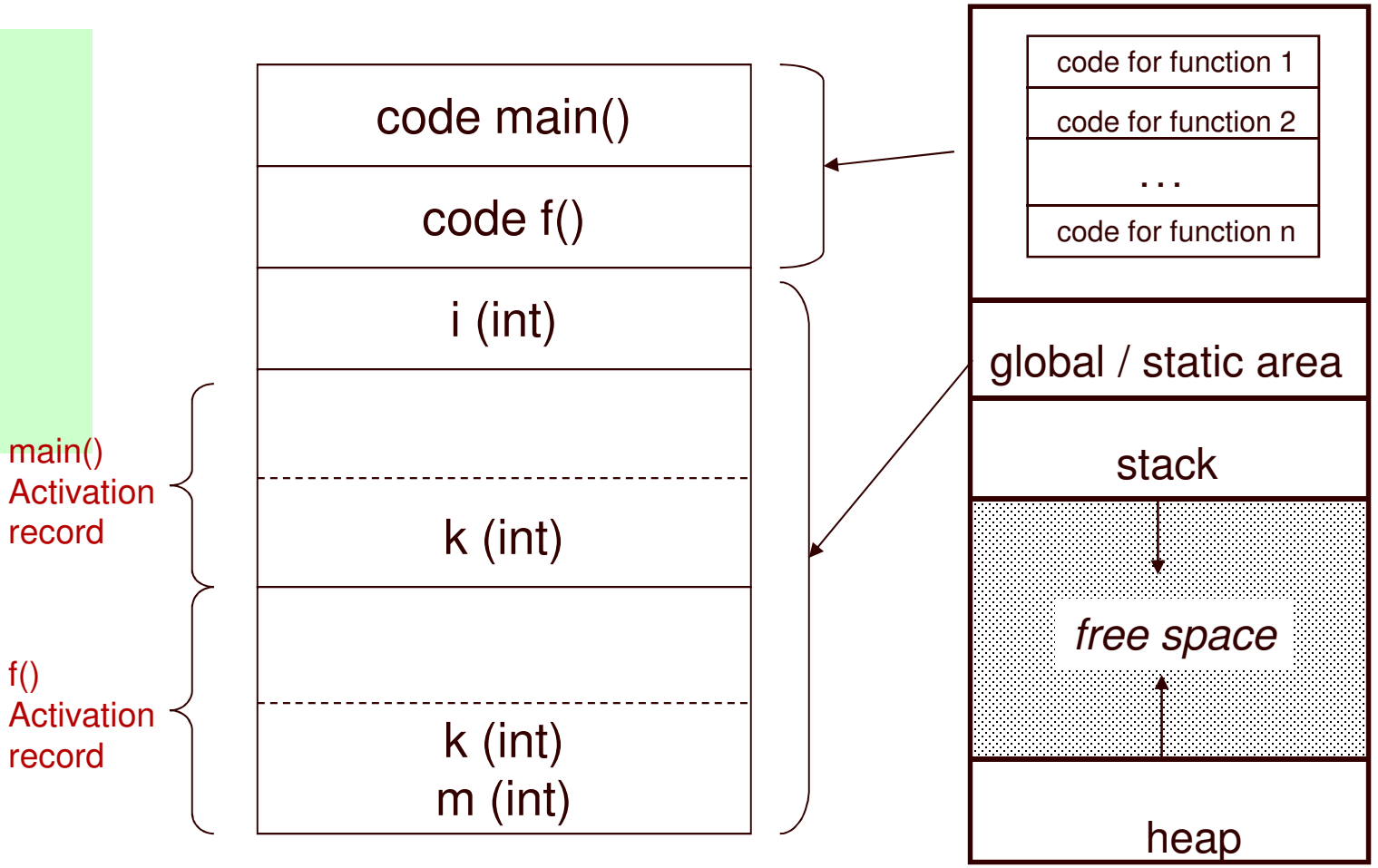
- In a static environment (Fortran 77) there are a number of restrictions:
  - Size of data objects are known at compile time
  - No recursive procedures
  - No dynamic memory allocation
- Only one copy of each procedure activation record exists at time t
  - We can allocate storage at compile time
    - Bindings do not change at runtime
    - Every time a procedure is called, the same bindings occur

Static Allocation

```
int i = 10;

int f(int j)
{
  int k;
  int m;
  …
}

main()
{
  int k;
  f(k);
}
```



Activation Record

Stack-based Allocation

- In a stack-based allocation, the previous restrictions are lifted (Pascal, C, etc)
  - procedures are allowed to be called recursively
    - Need to hold multiple activation records for the same procedure
    - Created as required and placed on the stack
      - Each record will maintain a pointer to the record that activated it
      - On completion, the current record will be deleted from the stack and control is passed to the calling record
  - Dynamic memory allocation is allowed
  - Pointers to data locations are allowed

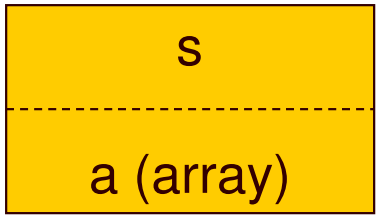
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree

s

Activation Records On Stack



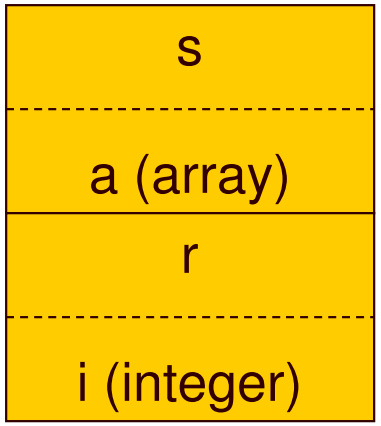
Activation Record

Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree

Activation Records On Stack

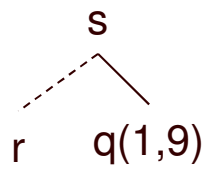


Activation Record

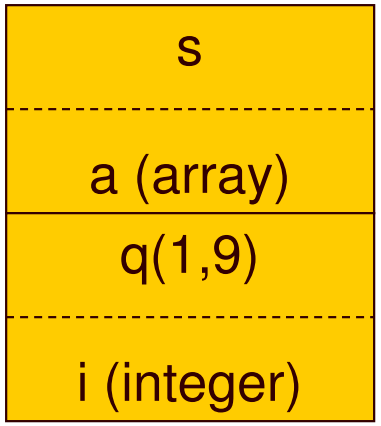
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree



Activation Records On Stack

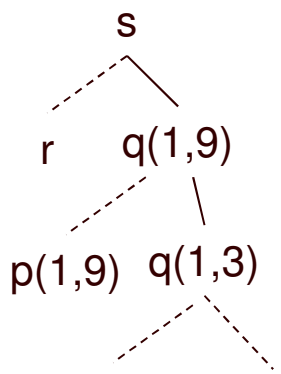


Activation Record

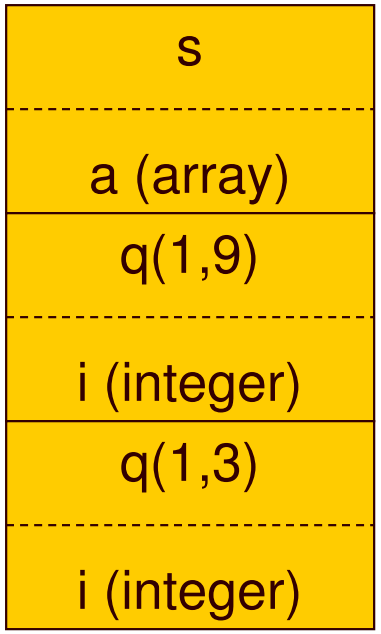
Stack-based Allocation

```
PROGRAM sort(input,output);
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    BEGIN
      for i:= 1 to 9 do read(a[i]);
    END;
 FUNCTION partition(y,z : Integer): Integer;
    VAR i,j,x,v : Integer;
    BEGIN
      …
    END;
 PROCEDURE quicksort(m,n : Integer);
    VAR i : Integer;
    BEGIN
      if (n > m) then BEGIN
          i := partition(m,n);
          quicksort(m, i-1);
          quicksort(i+1,n)
        END
    END;
  BEGIN  /* of main */
    a[0] := -9999; a[10] := 9999;
    readarray;
    quicksort(1,9)
  END.
```

Position in Activation Tree



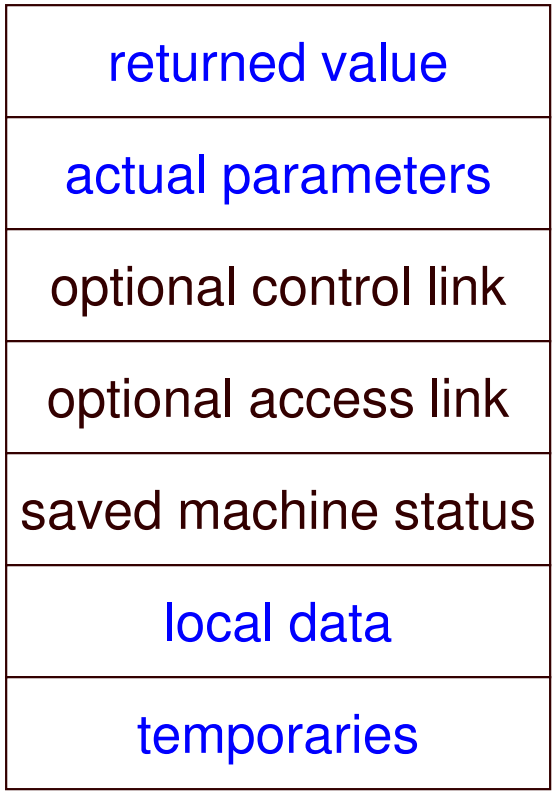
Activation Records On Stack



Activation Record

Calling Sequences

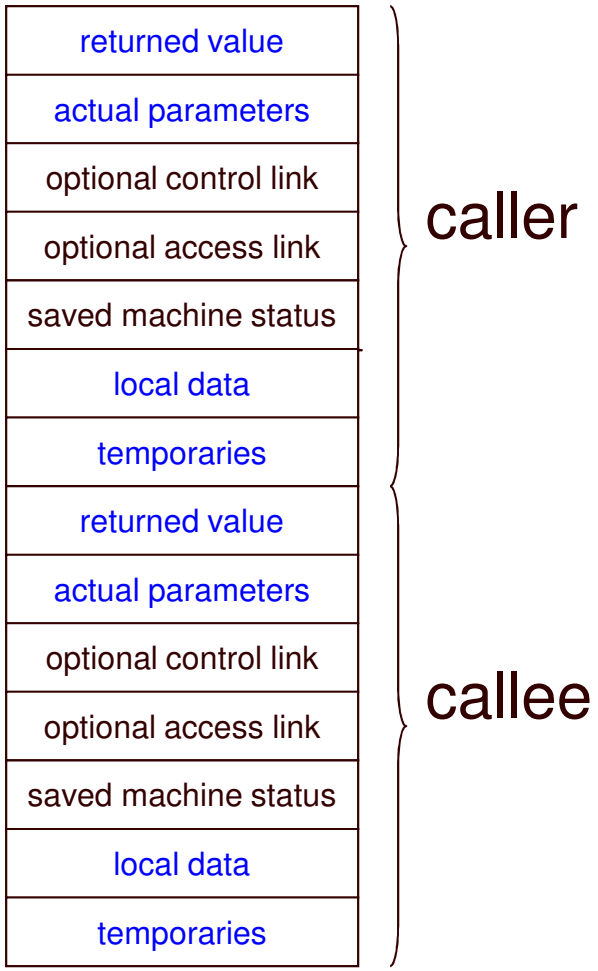
- Procedure calls are implemented by generating calling sequences in the target code
  - Call sequence: allocates activation record and enters information into fields
  - Return sequence: restores the state of the machine so that the calling procedure can continue execution



Activation Record

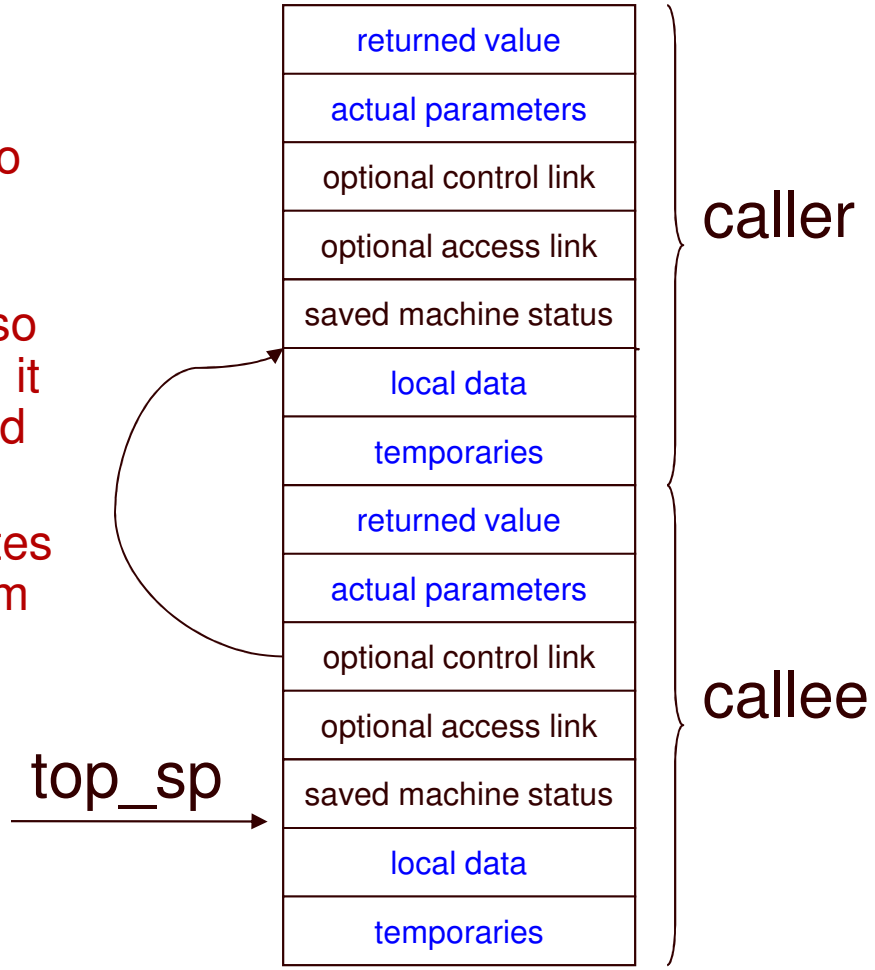
Calling Sequences

- Why placing returned value and actual parameters next to the activation record of the caller?
  - Caller can access these values using offsets from its own activation record
  - No need to know the middle part of the callee's activation record



Calling Sequences

- How do we calculate offset?
  - Maintain a register that points to the end of the machine status field in an activation record
  - Top_sp is known to the caller, so it can be responsible for setting it before control flows to the called procedure
  - Callee can access its temporaites and local data using offsets from top_sp



Call Sequence

- The caller evaluates actuals
- The caller
  - stores a return address and the old value of top_sp into the callee's activation record
  - increments top_sp; that is moved past the caller's local data and temporaries and the callee's parameter and status fields
- The callee
  - saves register values and other status information
- The callee
  - initializes its local data and begins execution

Return Sequence

- The callee places a return value next to the activation record of the caller
- Using the information in the status field, the callee
  - restores top_sp and other registers
  - braches to a return address in the caller's code
- Although top_sp has been decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression