

Tiger Runtime Environments

- **Compile-time environments** are just symbol tables; they are used to assist static semantic analysis, code generation and code optimization.
- **Run-time environments** are about how to map each runtime value into the memory? more specifically, here are the main issues

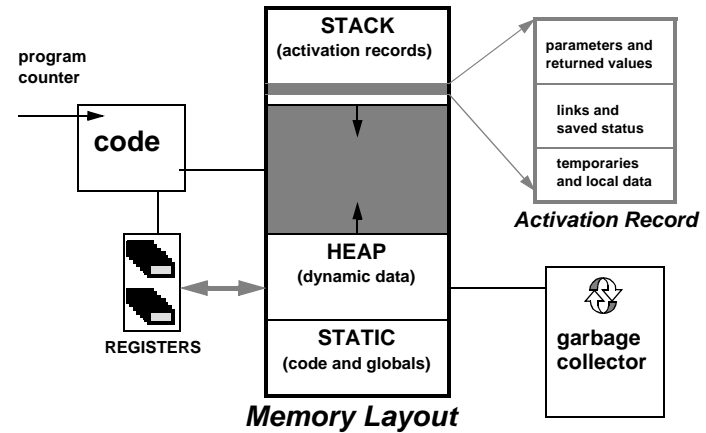
how to implement procedure (or function) call ? --- stack frames
(activation records, access to non-local variables, parameter passing,...)

what are the data representations ?
(primitive data type, records, arrays, dynamic data structure, ...)

what are the memory layout (i.e., storage organization) ?
(where to put the static data, code segments, stack, heap?)

how to do the memory allocation and de-allocation ?
(malloc-free package, garbage collection, ...)

Typical Runtime Layout



Example: Nested Functions

```

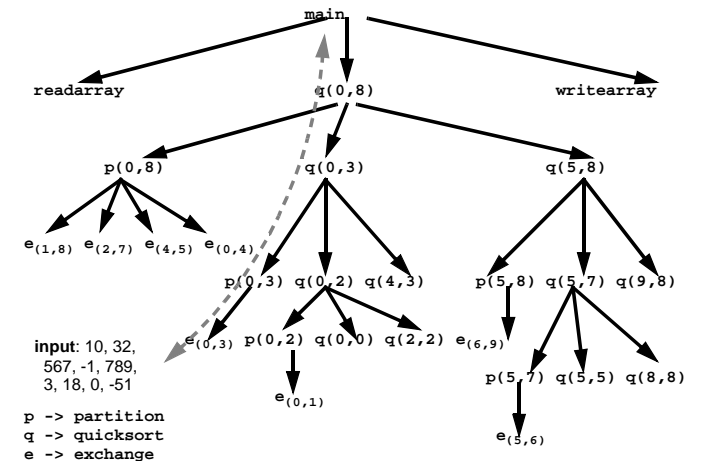
let type intArray = array of int
var a := intArray [9] of 0
function readarray () = ...
function writearray () = ...
function exchange(x : int, y : int) =
  let var z := a[x] in a[x] := a[y]; a[y] := z end

function quicksort(m : int, n : int) =
  let function partition(y : int, z : int) : int =
    let var i := y      var j := z + 1
    in (while (i < j) do
      (i := i+1; while a[i] < a[y] do i := i+1;
       j := j-1; while a[j] > a[y] do j := j-1;
       if i < j then exchange(i,j));
      exchange(y,j); j)
    end

    in if n > m then (let var i := partition(m,n)
                     in quicksort(m, i-1);
                     quicksort(i+1, n)
                    end)
  end
in readarray(); quicksort(0,8); writearray()
end
    
```

input: 10, 32, 567, -1, 789, 3, 18, 0, -51

Activation Trees



Activations

- Each function (or procedure) **declaration** associates a **name** with a function body ----- this binding is done at compile time.
- An **activation** is created during runtime execution when the function (or procedure) is invoked. The **lifetime** of an activation is the time between execution of the 1st operation of the body and through the last operation.
- Activations are either **nested** or **non-overlapping**. If two activations are **nested**, then one must be the descendant of another. If two activations are non-overlapping, then they must be the siblings.
- A function f is **recursive** if more than 2 activations of f is nested.
- Program execution is just **depth-first traversal** of activation tree !

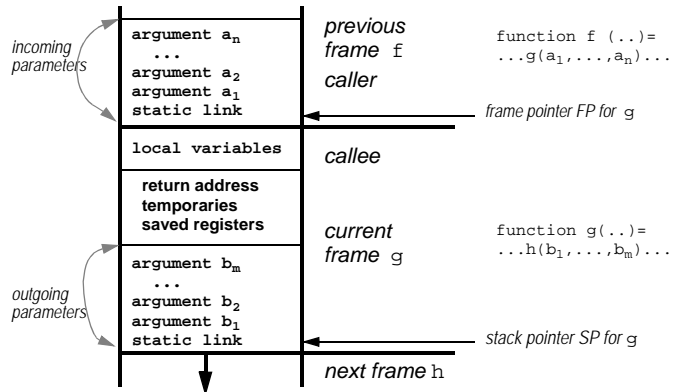
How to implement depth-first traversal ?

Activation Record

- An **activation record** is **constructed** when a function (or a procedure) is called (activated); it is **destroyed** when the function returns; the interim is the **lifetime** of the activation.
- The activation record often contains the following:
 - relevant machine state (saved registers, return address)
 - space for local data, including temporaries
 - space for return value
 - space for outgoing arguments
 - control link**: pointer to caller's activation record (optional)
 - static link**: pointer to activation for accessing **non-local** data
- **Main problem**: how to **layout** the activation record so that the caller and callee can **communicate** properly ?

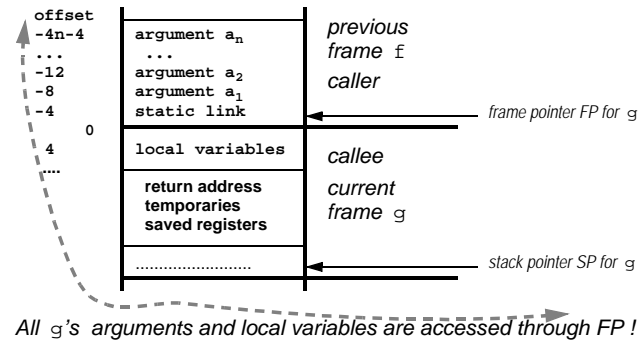
Stack Frames

- The most common (and standard) way is to allocate activation records on a sequential stack --- using the following **standard frame layout**.



Stack Frames (cont'd)

- **Frame Pointer (FP)** is a pointer that points to the start of the **current** frame; **Stack Pointer (SP)** --- referring to the top of the stack ---- points to the end of the **current** frame.



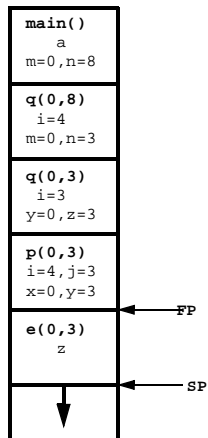
Typical Calling Sequence

- **Question:** Suppose function f calls function $g(a_1, \dots, a_n)$, what will happen at runtime? how f and g communicate? Assuming FP and SP are in two registers.
- **1. Call sequence** (done by the caller f before entering g)
 - f puts arguments a_1, \dots, a_n onto the stack (or in registers)
 - f puts function g 's static link onto the stack (or in a register)
 - f puts the return address of this call to g onto the stack (or in a register)
 - f puts current FP onto the stack (i.e., control link, optional)
 - Jump to g 's code
- **2. Entry sequence** (the first thing done after entering the callee g)
 - move SP to FP
 - decrement SP by the frame size of g (stack grows downwards!!!)
 - (optional: save callee-save registers if any)

Typical Calling Sequence (cont'd)

- **3. Return sequence** (the callee g exits and returns back to f)
 - put the return result into a designated register
 - (optional: restore callee-save registers if any)
 - fetch the return address to a register (if in register, do nothing)
 - fetch the saved FP of f back to the FP register
 - increment SP by the frame size of g (pop off the activation of g)
 - Return back to f
- **Tiger Specifics** (also true for many other modern compilers)
 - return address is put in a designated register
 - only maintain SP at runtime (FP is a "virtual" reg. = SP - framesize)
 - (when implementing Tiger, frame-size of each function is a compile-time constant)
- **Must maintain a separate FP and SP if**
 - (1) the frame size of a function may vary
 - (2) the frames are not always contiguous (e.g., linked list)

A Snapshot of Running Quicksort



- **Remaining questions:** how to find the value of local variables and non-local variables?
- **Local variables** are allocated in the current stack frame --- we can access them through the **Frame Pointer** (notice, the actual value of FP is unknown until runtime, but the each local-variable's offset to FP is known at compile time)
- **Non-local variables** must be accessed through the **static link**, or by using some other tricks

Non-Local Variables

```

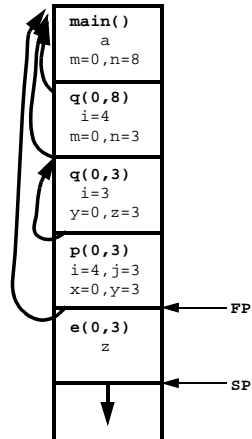
let type intArray = array of int
var a := intArray [9] of 0
2 function readarray () = ...
2 function writearray () = ...
2 function exchange(x : int, y : int) =
  let var z := a[x] in a[x] := a[y]; a[y] := z end

2 function quicksort(m : int, n : int) =
  let function partition(y : int, z : int) : int =
    3 let var i := y var j := z + 1
      in (while (i < j) do
          (i := i+1; while a[i] < a[y] do i := i+1;
           j := j-1; while a[j] > a[y] do j := j-1;
           if i < j then exchange(i,j));
          exchange(y,j); j)
        end
      in if n > m then (let var i := partition(m,n)
                      in quicksort(m, i-1);
                      quicksort(i+1, n)
                     end)
    end
  in readarray(); quicksort(0,8); writearray()
end
    
```

nesting depth

input: 10, 32,
567, -1, 789,
3, 18, 0, -51

Static Link

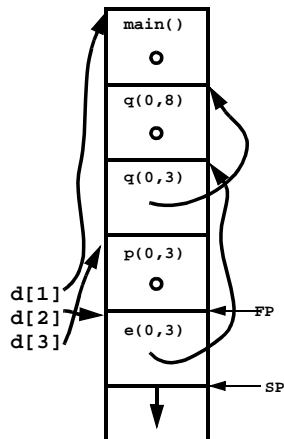


- **Static link** (also called **access link**) is used to implement lexical scoping.
- If function **p** is nested **immediately** within **q** in the source code, then the static link in activation of **p** is a point to the **most recent** activation of **q**.
- Non-local variable **v** is found by following static links to an activation (i.e, frame) that contains **v**
- If **v** is declared at depth n_v and accessed in **p** declared at depth n_p , then we need follow $n_p - n_v$ static links

Static Link (cont'd)

- Suppose function **q** at depth n_q calls function **p** at depth n_p . The question is how to access non-local variables once we are inside **p**? or what is the static link inside **p**'s activation.
- If $n_q < n_p$, then $n_q = n_p - 1$, **p** is nested within **q**; the static link in **p**'s activation = **q**'s activation; e.g., quicksort₍₂₎ calls partition₍₃₎.
- If $n_q \geq n_p$, **p** and **q** must have common calling "prefix" of functions at depths 1, ..., $n_p - 1$; the static link in **p**'s activation is the activation found by following $n_q - n_p + 1$ access links in caller **q**; e.g., partition₍₃₎ calls exchange₍₂₎ --- follow $3 - 2 + 1 = 2$ links inside partition's activation.
- Two alternatives to the **static link** method for accessing non-local variables:
 1. **Display**
 2. **Lambda-Lifting**

Display



- One alternative to static link is to maintain pointers to the current activation at depth **k** using a **display** array **d[1...]**.
- Upon **entry** to **p** at depth **k**: save **d[k]** in **p**'s activation; **d[k] = p**'s activation
- Upon **exit** from **p** at depth **k**: **d[k] = saved "d[k]"** inside **p**'s activation
- **display** ---- **pros**: faster access, constant call/return cost; **cons**: uses up registers, awkward when functions being passed as arguments.

Lambda-Lifting

```

let type intArray = array of int
var a := intArray [9] of 0
function readarray (a : intArray) = ...
function writearray (a : intArray) = ...
function exchange(a : intArray, x : int, y : int) =
  let var z := a[x] in a[x] := a[y]; a[y] := z end
function quicksort(a: intArray, m : int, n : int) =
  let function partition(a: intArray, y : int, z : int) : int =
    let var i := y          var j := z + 1
      in (while (i < j) do
          (i := i+1; while a[i] < a[y] do i := i+1;
           j := j-1; while a[j] > a[y] do j := j-1;
           if i < j then exchange(i,j));
          exchange(y,j); j)
        end
      in if n > m then (let var i := partition(a,m,n)
                       in quicksort(a, m, i-1);
                       quicksort(a, i+1, n)
                      end)
    end
  in readarray(a); quicksort(a,0,8); writearray(a)
end
    
```

Rewriting the program by treating non-local variables as formal parameter

Parameter Passing

how to map actual parameters to formal parameters?

- **call-by-value:** values of the actual arguments are passed and established as values of formal parameters. Modification to formals have no effect on actuals. **Tiger, ML, C** always use call-by-value.

```
function swap(x : int, y : int) =
  let var t : int := x in x := y; y := t end
```

- **call-by-reference:** locations of the actuals are passed; references to the formals include implicit indirection to access values of the actuals. Modifications to formals do change actuals. (supported in PASCAL, but not in Tiger)

```
function swap(var x : int, var y : int) =
  let var t : int := x in x := y; y := t end
```

Use of Registers

- To avoid memory traffic, modern compilers often pass arguments, return results, and allocate local variables in **machine registers**.

- **Typical parameter-passing convention on modern machines:**

the first k arguments ($k = 4$ or 6) of a function are passed in registers R_p, \dots, R_{p+k-1} , the rest are passed on the stack.

- **Problem :** extra memory traffic caused by passing args. in registers

```
function g(x : int, y : int, z :int) : int = x*y*z
```

```
function f(x : int, y : int, z : int) =
  let val a := g(z+3, y+3, x+4) in a*x+y+z end
```

Suppose function f and g pass their arguments in R_1, R_2, R_3 ; then f must save R_1, R_2 , and R_3 to the stack frame before calling g ,

Use of Registers (cont'd)

how to avoid extra memory traffic?

- **Leaf procedures** (or functions) are procedures that do not call other procedures; e.g. the function `exchange`. The parameters of **leaf procedures** can be allocated in registers without causing any extra memory traffic.
- Use **global register allocation**, different functions use different set of registers to pass their arguments.
- Use **register windows** (as on SPARC) --- each function invocation can allocate a fresh set of registers.
- Use **callee-save registers**
- When all fails --- save to the corresponding slots in the stack frame.

Callee-save Registers

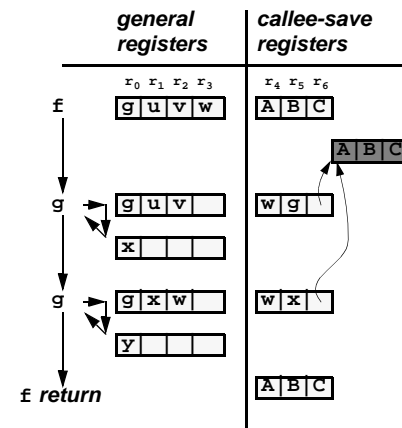
Convention :

Reserve k special registers !

Every function promises to always preserve these registers !

Example : $k=3$ (r_4, r_5, r_6)

```
fun f(u,v,w) =
  let val x = g(u,v)
      val y = g(x,w)
  in x+y+w
  end
```



Frame Resident Variables

Certain values must be allocated in stack frames because

- the value is too big to fit in a single register
- the variable is passed by reference --- must have a memory address
- the variable is an array -- need address arithmetic to extract components
- the register that the variable stays needs to be used for other purpose!
- just too many local variables and arguments --- there are not enough registers !!! ----- **SPILLING**

Open research problem: When to allocate local variables or passing arguments in registers ?

Needs good heuristics !

Stack Frames in Tiger

- Using **abstraction** to avoid the machine-level details
- What do we need to know about each stack frame at compile time ?

1. offsets of incoming arguments and the static links
2. offsets of all local variables
3. the frame size

```
signature FRAME =
sig type frame
  val newFrame : int -> frame * int list
  val allocLocal : frame -> int
  (* other stuff, eventually ... *)
end

structure PowerPCFrame : FRAME =
struct
  type frame = {formals: int, offlst: int list,
                locals: int ref, maxargs: int ref}
  .....
end

structure SparcFrame : FRAME = ...
```

Stack Frames in Tiger (cont'd)

- In the **static environment** (i.e., the symbol table), associate each variable with the **access** information; associate each function with the layout information of its activation record (i.e, frame), a static link, and the caller's frame.

```
type offset = int

datatype level
  = LEVEL of {frame: Frame.frame,
              slink_offset: offset,
              parent: level} * unit ref
  | TOP

type access = level * offset
```

- When converting the **absyn** into intermediate code --- generate accessing code for each local or non-local variable, plus calling sequences for each function call.

Limitation of Stack Frames

- It does not support **higher-order functions** ---- it cannot support "nested functions" and "procedure passed as arguments and results" at the same time.

C --- functions passed as args and results, but no nested functions;
PASCAL --- nested functions, but cannot be passed as args or res.

- **Alternative to the standard stack allocation scheme** ----

1. use a linked list of chunks to represent the stack
2. allocate the activation record on the heap --- no stack frame pop !

advantages: support higher-order functions and parallel programming well

(will be discussed several weeks later !)