Lecture 9: Procedures & Functions

CS 540

George Mason University
Procedures/Functions

• Control Abstraction
  – call/return semantics, parameters, recursion

• Controlled Namespace
  – Scope (local/non-local), binding, addressing

• External Interface
  – separate compilation, libraries (not dealing with here)
Procedures as Control Abstractions

Relationship between caller and callee is asymmetric

- Control flow (call and return)
- Data flow (call and return): parameters and return values
- Recursion
- Variable addressing

The way this data/control flow is achieved is strictly defined and the given rules must be adhered to by the compiler
Data Structure: Call Graph

A **call graph** is a directed multi-graph where:

- the nodes are the procedures of the program and
- the edges represent calls between these procedures.

Used in optimization phase.

Acyclic $\rightarrow$ no recursion in the program

Can be computed **statically**.
Example

```pascal
var a: array [0 .. 10] of integer;

procedure readarray
var i: integer
begin ... 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 i := partition(m,n); quicksort(m,i-1); quicksort(i+1,n) end

procedure main
begin readarray(); quicksort(1,9); end
```
Data Structure: Call Tree

• A call tree is a tree where:
  – the nodes are the procedure activations of the program and
  – the edges represent calls between these procedure activations.

• Dynamic – typically different every time the program is run
Run-time Control Flow

Call Tree - cannot be computed statically

```
main
  r()
  q (1,9)
    p (1,9)
    q (1,3)
    q (5,9)
      p (1,3)
      q (1,0)
      q (2,3)
      p (5,9)
      q (5,5)
      q (7,9)
        p (2,3)
        q (2,1)
        q (3,3)
        p (7,9)
        q (7,7)
        q (9,9)
```
Run-time Control Flow

Paths in the call tree from root to some node represent a sequence of active calls at runtime.
Static Allocation

• Historically, the first approach to solving the run-time control flow problem (Fortran)

• All space allocated at compile time $\rightarrow$ No recursion
  – Code area – machine instructions for each procedure
  – Static area –
    • single data area allocated for each procedure.
      – local vars, parameters, return value, saved registers
    • return address for each procedure.
Static Allocation

<table>
<thead>
<tr>
<th>Code area</th>
<th>Data area for procedure A</th>
<th>Data area for procedure B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code generated for A</td>
<td>return addr</td>
<td>Local data, parameters, return value, registers for A</td>
</tr>
<tr>
<td>Code generated for B</td>
<td></td>
<td>Local data, parameters, return value, registers for B</td>
</tr>
</tbody>
</table>
Call/Return processing in Static Allocation

• When A calls B:
  – in A: evaluate and save actual parameters, save any registers and status data needed, save RA, finally update the program counter (PC) to B’s code
  – In B: deal with parameters and RA (if needed)

• When the call returns
  – in B: save return value, update PC to value in RA
  – in A: get return value, restore any saved registers or status data

Save options: data areas (A or B), registers
Static Allocation

Call tree:
A

Code area

A:
call B
L1:

B:
...
return

Activation for procedure A

Activation for procedure B

Local data, parameters, return value, registers for A

Local data, parameters, return value, registers for B
Static Allocation

Code area

A:
    call B
L1:

B:
    ...
return

Activation for procedure A
Local data for A

Activation for procedure B
Local data for B

Call tree:
A
   B
Static Allocation

Code area

A:

call B

L1:

B:
... return

Activation for procedure A

Local data for A

Activation for procedure B

Local data for B

Call tree: A

PC
Managing Control Flow: Spim

1. Set aside extra space for return address and local info for each function.

FN_RA: .word 0
FN_Regs: .space 48
# save 8 $t + 4 $a registers
FN_Params: .word ...

Managing Control Flow: Spim

2. Add code at the point of the call:

\[
\begin{align*}
\text{sw} & \quad \text{$t0, FN1\_Regs} \\
\text{sw} & \quad \text{$t1, FN1\_Regs+4} \\
\text{move} & \quad \text{$a0, t} \\
\text{jal} & \quad \text{function} \\
\text{lw} & \quad \text{$t0, FN1\_Regs} \\
\text{lw} & \quad \text{$t1, FN1\_Regs +4} \\
\text{move} & \quad \text{$t_x, v0}
\end{align*}
\]

jal – save the address of the next statement in $ra and then jump to the given label

Save and restore local registers in use

Pass in parameters and get the return value
Managing Control Flow: Spim

3. Add code in the called function
   
   – Prologue:
     
     ```
     sw $ra, FN2_RA
     sw $a0, FN2_Param1 ...
     ```

   – Epilogue
     
     ```
     move $v0,$t\_y
     lw $t\_x$, FN2_RA
     jr $t\_x
     ```

   jal put return address in $ra

If there is a return value…
result := f(a,bb);

.data
main_Regs: .word 0,0,0,0,0,0,0,0,0,0
main_RA: .word 0

.text
lw $t0,a
move $a0,$t0
lw $t0,bb
move $a1,$t0
sw $t0,main_Regs
sw $t1,main_Regs+4
sw $t2,main_Regs+8
jal label_f
lw $t0,main_Regs
lw $t1,main_Regs+4
lw $t2,main_Regs+8
move $t0,$v0
sw $t0,result
int f(int x, int y) {... return max; }

.data
f_RA: .word 0  # return addr
.text
label_f:
    sw $ra,F_RA
.data
x: .word 0   # param 1
    .text
    sw $a0,x
.data
y: .word 0   # param 2
    .text
    lw $v0,max  # return val
    lw $t0,f_RA
    jr $t0

body of f
Runtime Addressing in Static Allocation

• Variable addresses hard-coded, usually as offset from data area where variable is declared.
  – addr(x) = start of x's local scope + x's offset

• In Spim, we are going to save the local variable using a label and when needed, we can use `lw` to get the value (or `la` to get the address).
Static Allocation: Recursion?

What happens???

Code area

A:
call B

B:
call B
L2:

Activation for procedure A

return addr

Local data for A

Activation for procedure B

return addr

Local data for B
Static Allocation

Call tree:

A

Code area

A:
    call B

L1:

B:
    call A
    L2:
    return

Activation for procedure A

Local data for A

Activation for procedure B

Local data for B
Static Allocation

Code area

A:
call B
L1:

B:
call B
L2:
return

Activation for procedure A

Local data

Call tree:
A
B

Activation for procedure B

Local data

L1
Static Allocation

Code area

A:
   call B
L1:

B:
   call B
L2:
   return

Activation for procedure A

Local data for A

Activation for procedure B

Local data for B

Call tree:
A
   B
      B
Static Allocation

Code area

A:
  call B
L1:

B:
  call B
L2:
  return

Activation for procedure A

Activation for procedure B

Local data for A

Local data for B

Call tree:

A
  B
Static Allocation

Code area

A:
    call B
L1:

B:
    call B
L2:
    return

Activation for procedure A

L2
Local data for A

Activation for procedure B

L2
Local data for B

Call tree:

A
    B

We’ve lost the L1 label so we can’t get back to A
Stack Allocation

Need a different approach to handle recursion.

- Code area – machine code for procedures
- Static data – often not associated with procedures
- Stack – runtime information
  - Activation records – allocated at call time onto a runtime stack. Holds return addresses, local information
  - Dynamic – can grow and shrink
Process Address Space

• Each process has its own *address space*:
  – **Text section (text segment)** contains the executable code
  – **Data section (data segment)** contains the global variables
  – **Stack** contains temporary data (local variables, return addresses..)
  – **Heap**, which contains memory that is dynamically allocated at run-time.
Activation Records (frame)

Information needed by a single instance of a procedure.

• Local data
  – Parameter storage
  – Return value storage
• Saved registers
• Control links for stack
• Return address
Activation Records

Different procedures/functions will have different size activation records. Activation record size can be determined at compile time.

At call time, we push a new activation on the runtime stack. At call termination time, we pop the activation off the stack.
Stack Allocation - 1

Call Tree

Stack (growing downward)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td></td>
</tr>
<tr>
<td>a: array</td>
<td></td>
</tr>
<tr>
<td>readarray</td>
<td></td>
</tr>
<tr>
<td>i: integer</td>
<td></td>
</tr>
</tbody>
</table>
Stack Allocation - 2

Call Tree

Stack (growing downward)

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
</tr>
<tr>
<td>a: array</td>
</tr>
<tr>
<td>quick(1,9)</td>
</tr>
<tr>
<td>i: integer</td>
</tr>
</tbody>
</table>

Main

readarray quick(1,9)
Stack Allocation - 3

Call Tree

Main

readarray quick(1,9)

p(1,9) quick(1,3)

p(1,9) quick(1,0)

Stack (growing downward)

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: array</td>
</tr>
<tr>
<td>quick(1,9)</td>
</tr>
<tr>
<td>i: integer</td>
</tr>
<tr>
<td>quick(1,3)</td>
</tr>
<tr>
<td>i: integer</td>
</tr>
<tr>
<td>quick(1,0)</td>
</tr>
<tr>
<td>i: integer</td>
</tr>
</tbody>
</table>
Call Processing: Caller

- Create new (callee) activation record on stack.
- Evaluate actual parameters and place them on stack (or register).
- Save registers and other status data
- Store a return address, **dynamic link (= current stack pointer)**, and **static link information** into callee activation then
- **Make stack pointer (SP) point at new activation.**
- Update the program counter (PC) to the code area for the called procedure.

**Added at the point of the call**
Call Processing: Callee

• Initialize local data, including moving parameters (if needed)
• Save return address if needed
• Stack maintenance
• Begin local execution

Added at the start of the function
Return Processing: Callee

- Place return value (if any) in activation.
- Stack maintenance
- Restore PC (from saved RA).

Added at the ‘return’ point(s) of the function
Return Processing: Caller

- Restore the stack pointer
- Restore registers and status
- Copy the return value (if any) from activation
- Continue local execution

Added after the point of the call
## Spim Example: Activation

<table>
<thead>
<tr>
<th>Offset</th>
<th>Data</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Return address</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>old frame ptr</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$t$ registers</td>
<td>32</td>
</tr>
<tr>
<td>40</td>
<td>$a$ registers</td>
<td>16</td>
</tr>
<tr>
<td>56</td>
<td>local vars</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Spim Example

Assume a function f with two integer parameters named x and y that are passed by value. f returns a single integer. Also in function f are two local variables a and b.

Function f's prolog:

sw $ra,0($sp)  # save return address
sw $a0,56($sp) # parameter x
sw $a1,60($sp) # parameter y

NOTE: local variables a and b can be stored at offsets 64 and 68 respectively.

• Function f's epilog:
  move $v0,$t1
  lw $t1,0($sp)
  jr $t1
At \( f(a,b) \)

\[
\begin{align*}
\text{# saving registers} \\
\text{sw } & \textdollar t0,8(\textdollar sp) \\
\text{sw } & \textdollar t1,12(\textdollar sp) \\
\cdots \\
\text{sw } & \textdollar a0,40(\textdollar sp) \\
\cdots \\
\text{# get actual parameters} \\
\text{lw } & \textdollar a0,a\_global \\
\text{lw } & \textdollar a1,b\_global \\
\text{# stack maintenance} \\
\text{sw } & \textdollar fp,4(\textdollar sp) \\
\text{subu } & \textdollar sp,\textdollar sp,104 \\
\text{addiu } & \textdollar fp,\textdollar sp,100 \text{ # set frame ptr} \\
\text{jal } & \text{outputnums} \\
\text{move } & \textdollar t0,\$v0 \text{ # grab return value} \\
\text{addiu } & \textdollar sp,\textdollar sp,104 \text{ # reset sp} \\
\text{lw } & \textdollar fp,4(\textdollar sp) \text{ # reset frame pointer} \\
\text{lw } & \textdollar t0,8(\textdollar sp) \text{ # restoring registers} \\
\text{lw } & \textdollar t1,12(\textdollar sp) \\
\cdots \\
\text{lw } & \textdollar a0,40(\textdollar sp) \\
\cdots
\end{align*}
\]

http://cs.gmu.edu/~white/CS540/Slides/Semantic/runtime.html
Runtime Addressing

• Given a variable reference in the code, how can we find the correct instance of that variable?
• Things are trickier – variables can live on the stack.
• Tied to issues of scope
Types of Scoping

• Static – scope of a variable determined from the source code. Scope A is enclosed in scope B if A's source code is nested inside B's source code.

• Dynamic – current call tree determines the relevant declaration of a variable use.
Static Scoping: Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.
Example Program

Program main;
a, b, c: real;

procedure sub1(a: real);
d: int;

procedure sub2(c: int);
d: real;
body of sub2

procedure sub3(a: int)
body of sub3

body of sub1

body of main
Example Program: Static

Program main;
  a, b, c: real;

  procedure sub1(a: real);
    d: int;

    procedure sub2(c: int);
      d: real;

    body of sub2
  
  procedure sub3(a: int)

    body of sub3

  body of sub1

body of main

What is visible at this point (globally)?
What is visible at this point (sub1)?
Example Program: Static

Program main;
  a, b, c: real;

  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
      d: real;
    body of sub2
  body of sub1

  procedure sub3(a: int)
    body of sub3

body of main

What is visible at this point (sub3)?
Example Program: Static

Program main;
a, b, c: real;

procedure sub1(a: real);
d: int;

procedure sub2(c: int);
d: real;
body of sub2

procedure sub3(a: int)
body of sub3

body of sub1

body of main

What is visible at this point (sub2)?
## Variables from Example

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Enclosing</th>
<th>Local: addr = offset</th>
<th>Non-local: addr = (scope, offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>-</td>
<td>a:0, b:1, c:2</td>
<td>-</td>
</tr>
<tr>
<td>sub1</td>
<td>main</td>
<td>a:0, d:1</td>
<td>b: (main, 1), c: (main, 2)</td>
</tr>
<tr>
<td>sub2</td>
<td>sub1</td>
<td>c:0, d:1</td>
<td>b: (main, 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a: (sub1, 0)</td>
</tr>
<tr>
<td>sub3</td>
<td>sub1</td>
<td>a:0</td>
<td>b: (main, 1), c: (main, 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>d: (sub1, 1)</td>
</tr>
</tbody>
</table>
Control Links in Stack Allocation

• Dynamic – points to caller’s activation (old frame pointer)
• Static (access) link – points to enclosing scope
Static Chain Maintenance

• How to set the static link?
  – Let $P_{sd}$ be the static_depth of P, and $Q_{sd}$ be the static_depth of Q
  – Assume Q calls P
  – There are three possible cases:
    1. $Q_{sd} = P_{sd}$
    2. $Q_{sd} < P_{sd}$
    3. $Q_{sd} > P_{sd}$
Static Chain Maintenance: Q calls P

\[ Q_{sd} = P_{sd} \] - They are at same static depth

P’s static link should be the same as Q’s since they must occur in same enclosing scope – Q copies its link to P
Static Chain Maintenance: Q calls P

\[ Q_{sd} < P_{sd} \] - P must be enclosed directly in Q
P’s static link should point at Q’s activation
Static Chain Maintenance: Q calls P

\[ Q_{sd} > P_{sd} \] - Q is \( n \) levels down in the nesting – must follow Q’s static chain \( n \) levels and copy that pointer

Suppose \( n \) is 2
Runtime Addressing in Stack Allocation

• At runtime, we can’t know where the relevant activation record holding the variable exists on the stack
• Use static (access) links to enable quick location
  – $\text{addr}(x) = \# \text{ static links} + x's \ offset$
  – Local: (0,offset)
  – Immediately enclosing scope: (1,offset)
Example Program

Program main;

procedure sub1(a: int,b:int);
procedure sub2(c: int);
   if c > 0 call sub2(c-1)
procedure sub3()
   body of sub3
end procedure;
call sub3(b); call sub2(a);
call sub1(3,4);
Example Program at runtime 1

main:
call sub1(3,4)
s1:

sub1:
call sub3(b)
s2: call sub2(a)
s3:

sub2:
call sub2(c-1)
s4:

sub3:
Example Program at runtime 2

```
main:
call sub1(3,4)
s1:
sub1:
call sub3(b)
s2: call sub2(a)
s3:
sub2:
call sub2(c-1)
s4:
sub3:
```

```
Code area

main:
call sub1(3,4)
s1:
sub1:
call sub3(b)
s2: call sub2(a)
s3:
sub2:
call sub2(c-1)
s4:
sub3:
```
Example Program at runtime 3

Code area
main:
call sub1(3,4)
s1:
sub1:
call sub3(b)
s2:
call sub2(a)
s3:
sub2:
call sub2(c-1)
s4:
sub3:
Example Program at runtime 4

Code area:
- **main:**
  - call sub1(3,4)
  - s1:
    - call sub3(b)
    - s2: call sub2(a)
    - s3: call sub2(c-1)
  - s4:
    - sub2:
      - call sub2(c-1)
    - sub3:
Example Program at runtime

main:
call sub1(3,4)
s1:
call sub3(b)
s2:
call sub2(a)
s3:
call sub2(c-1)
s4:

Code area

sub1

RA
b
4
s1
DP
SP

sub2

RA
s3
c
3
DP
SP

sub2

RA
s4
c
2
DP
SP

stack

PC

main:
call sub1(3,4)
s1:
sub1:
call sub3(b)
s2:call sub2(a)
s3:
sub2:
call sub2(c-1)
s4:
sub3:
Display
Alternate representation for access information.

- The current static chain kept in an array – entry $n$ points to the activation record at level $n$ in the static chain.

```
Q’s ari
...
R’s ari
...
P’s ari
S’s ari
```
Display Maintenance

For Q calls P …

\[ Q_{sd} < P_{sd} \quad - \quad P \text{ must be enclosed directly in } Q \]

If entry \( n \) points at Q, entry \( n+1 \) points at P
Display Maintenance

For Q calls P …

\[ Q_{sd} = P_{sd} \] - They are at same static depth

Update entry n to point to P

What happens when P ends? Must save the old entry n in P’s ar to make things work properly.

Q’s ar

becomes

P’s ar

Q’s ar

becomes
Display Maintenance

$Q_{sd} > P_{sd}$ - Q is $n$ levels down in the nesting – update that pointer $n$ levels down
Display

• Advantages: faster addressing
• Disadvantages: additional data structure to store and maintain.
Parameter Passing

Various approaches to passing data into and out of a procedure via parameters

1. **Call-by-value** – data is copied at the callee into activation and any item changes do not affect values in the caller.

2. **Call-by-reference** – pointer to data is placed in the callee activation and any changes made by the callee are indirect references to the actual value in the caller.

3. **Call-by-value-result** (copy-restore) – hybrid of call-by-value and call-by-reference. Data copied at the callee. During the call, changes do not affect the actual parameter. After the call, the actual value is updated.

4. **Call-by-name** – the actual parameter is in-line substituted into the called procedure. This means it is not evaluated until it is used.
Call-by-value

var a, b : integer
procedure s (x, y : integer);
var t : integer;
begint := x; x := y; y := t;end;
a := 1; b := 2;
s (a, b);
write ('a = ',a);
write ('b = ',b);
end.

1. x and y are integers initialized to a’s and b’s values
2. At end of s, x = 2 and y = 1
3. No change to a and b on return
Call-by-reference

var a, b : integer
    procedure s (x, y : integer);
        var t : integer;
        begin  t := x; x := y; y := t;  end;
    begin
        a := 1;  b := 2;
        s (a, b);
        write ('a = ',a);
        write ('b = ',b);
    end.

1. Pointers x and y are initialized to a’s and b’s addresses
2. At end of s, x (and a) = 2 and y (and b) = 1
Call-by-value/result

var a, b : integer

procedure s (x, y : integer);
  var t: integer;
  begin  t := x; x := y; y := t;  end;
begin
  begin
    a := 1;  b := 2;
    s (a, b);
    write ('a = ',a);
    write ('b = ',b);
  end.
end.

1. x and y are integers initialized to a’s and b’s values
2. At end of s, x = 2 and y = 1
3. At return, a is given x’s value and b is given y’s value
Call-by-name

var a, b : integer

procedure s (x, y : integer);
var t : integer;
begin  t := x; x := y; y := t;  end;
begin
  a := 1;  b := 2;
s (a, b);
write ('a = ', a);
write ('b = ', b);
end.

begin
  a := 1;  b := 2;
s (a, b);
write ('a = ', a);
write ('b = ', b);
end.
Call-by-value-result vs. Call-by-reference

var a: integer
    procedure foo(x: integer);
    begin a := a + 1; x := x + 1; end;
begin
a := 1;
foo(a);
write ('a = ',a);
end.

<table>
<thead>
<tr>
<th>write(a)</th>
<th>Value-result</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>