Intro to Procedures

CS4410: Spring 2013

Procedures

Let's augment Fish with procedures and local variables.

```
type exp = ... |
   Call of var * (exp list)

type stmt = ... | Let of var*exp*stmt

type func = { name : var, args : var list,
              body : stmt }

type prog = func list
```

Call & Return

Each procedure is just a Fish program beginning with a label (the function name).

The MIPS procedure calling convention is:

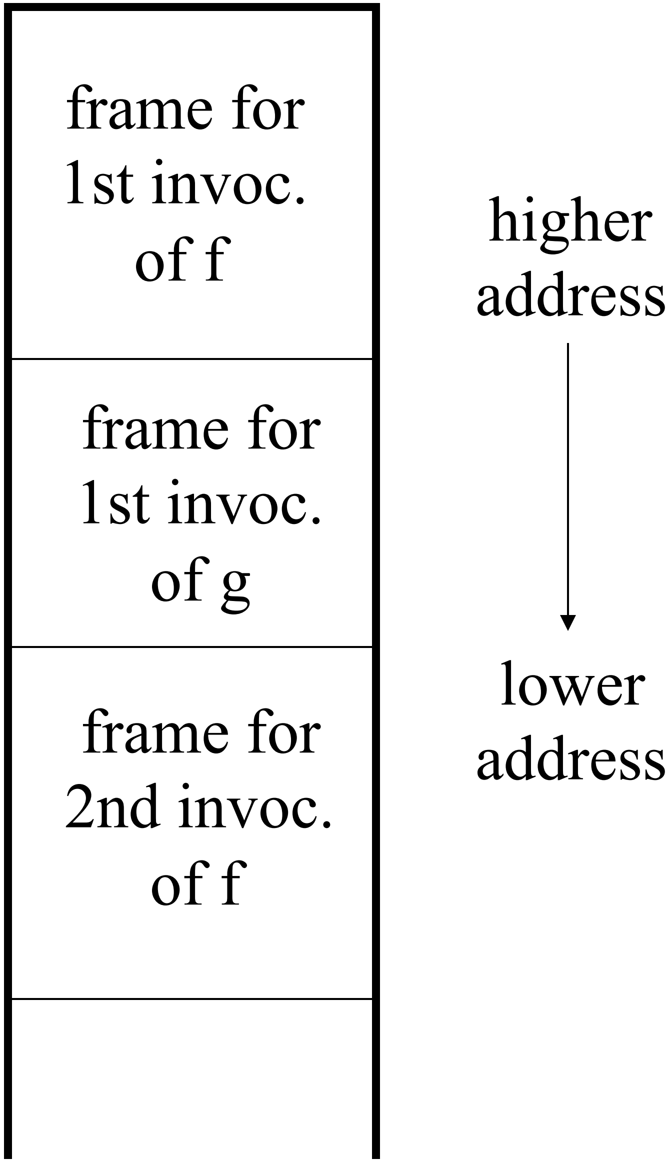
- To compile a call f(a,b,c,d),
  - we move results of a,b,c,d into $4-$7
  - jal f: this moves the return address into $31
- To return(e):
  - we move result of e into $2
  - jr $31: that is, jump to the return address.

What goes wrong?

- Oops, what if f calls g and g calls h?
  - g needs to save its return address.
  - (a caller-saves register)
  - Where do we save it?
  - One option: have a variable for each procedure (e.g., g_return) to hold the value.
- But what if f calls g and g calls f and f calls g and …?
  - we need a bunch of return addresses for f & g
  - (and also a bunch of locals, arguments, etc.)

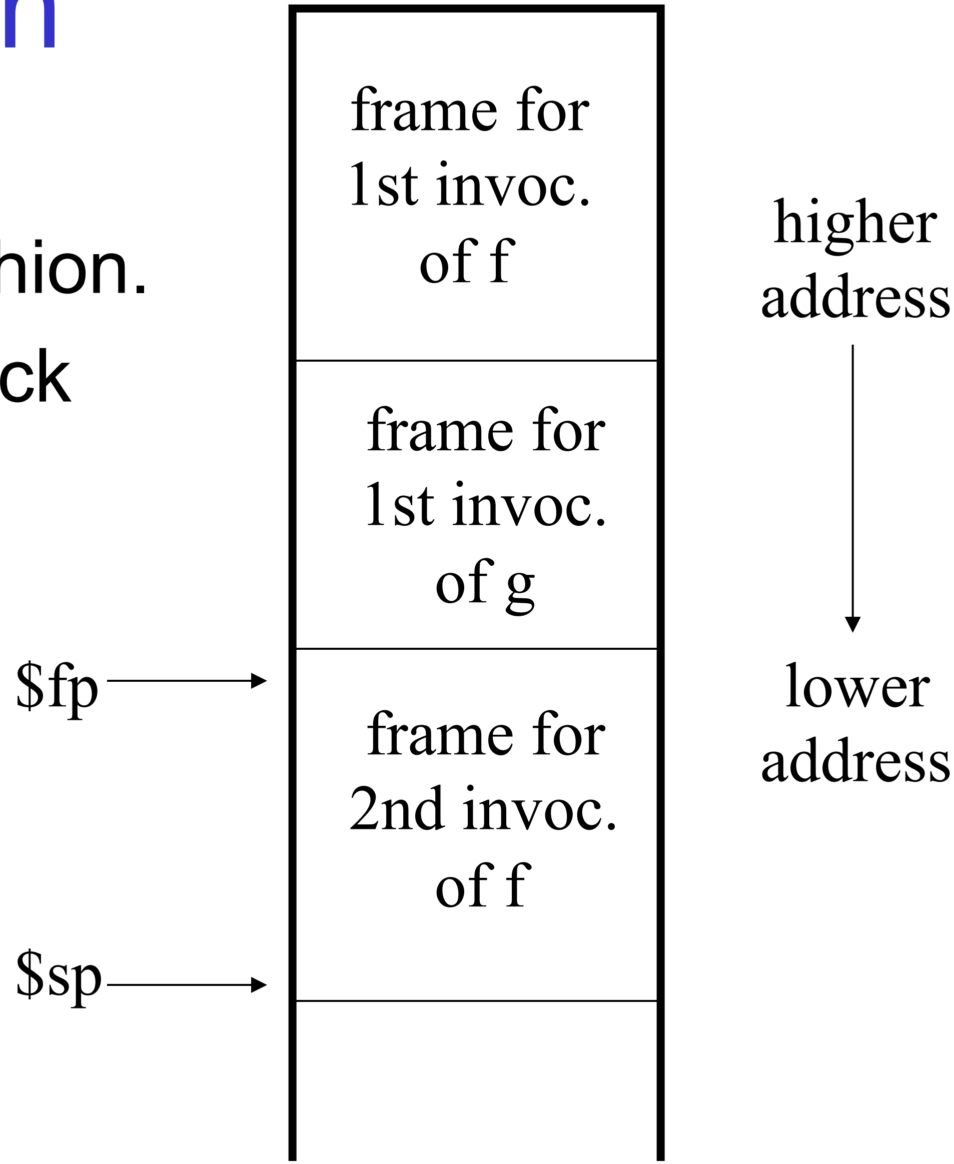
Stacks:

- The trick is to associate a frame with each *invocation* of a procedure.
- We store data belonging to the invocation (e.g., the return address) in the frame.



Frame Allocation

- Frames are allocated in a last-in-first-out fashion.
- We use \$29 as the stack pointer (aka \$sp).
- To allocate a frame with n bytes, we subtract n from \$sp.



Calling Convention in Detail:

To call **`f`** with arguments $a_1,\ldots,a_n$:

1. Save caller-saved registers.
   - These are registers that **`f`** is free to clobber, so to preserve their value, you must save them.
   - Registers \$8-\$15,\$24,\$25 (aka \$t0-t9) are the general-purpose caller-saved registers.
2. Move arguments:
   - Push extra arguments onto stack in reverse order.
   - Place 1st 4 args in \$a0-a3 (\$4-\$7).
   - Set aside space for 1st 4 args.
3. Execute **`jal f`**: return address placed in \$ra.
4. Upon return, pop arguments & restore caller-saved registers.

Function Prologue

At the beginning of a function **`f`**:

1. Allocate memory for a frame by subtracting the frame's size (say *n*) from $sp.
   - Space for local var's, return address, frame pointer, etc.
2. Save any callee-saved registers:
   - Registers the caller expects to be preserved.
   - Includes $fp, $ra, and $s0-$s7 ($16-$23).
   - Don't need to save a register you don't clobber…
3. Set new frame pointer to $sp + *n* - 4.

During a Function:

- Variables access relative to frame pointer:
  - must keep track of each var's offset
- Temporary values can be pushed on the stack and then popped back off.
  - Push(r): subu $sp,$sp,4; sw r,0($sp)
  - Pop(r): lw r,0($sp); addu $sp,$sp,4
  - e.g., when compiling e1+e2, we can evaluate e1, push it on the stack, evaluate e2, pop e1's value and then add the results.

Function Epilogue

At a return:

1. Place the result in $v0 ($r2).
2. Restore the callee-saved registers saved in the prologue (including caller's frame pointer and the return address.)
3. Pop the stack frame by adding the frame size (n) to $sp.
4. Return by jumping to the return address.

Example (from SPIM docs):

```
int fact(int n) {
  if (n < 1) return 1;
  else return n * fact(n-1);
}

int main() {
  return fact(10)+42;
}
```

Main

```
main: subu  $sp,$sp,32   # allocate frame
      sw    $ra,20($sp) # save caller return address
      sw    $fp,16($sp) # save caller frame pointer
      addiu $fp,$sp,28   # set up new frame pointer
      li    $a0,10       # set up argument (10)
      jal   fact         # call fact
      addi  $v0,v0,42    # add 42 to result
      lw    $ra,20($sp) # restore return address
      lw    $fp,16($sp) # restore frame pointer
      addiu $sp,$sp,32   # pop frame
      jr    $ra          # return to caller
```

Fact

```
fact:  subu   $sp,$sp,32     # allocate frame
       sw     $ra,20($sp)    # save caller return address
       sw     $fp,16($sp)    # save caller frame pointer
       addiu $fp,$sp,28      # set up new frame pointer
-----------------------------------------------------------
       bgtz   $a0,L2         # if n > 0 goto L2
       li     $v0,1          # set return value to 1
       j      L1             # goto epilogue
L2:    sw     $a0,0($fp)     # save n
       addi   $a0,$a0,-1     # subtract 1 from n
       jal    fact           # call fact(n-1)
       lw     $v1,0($fp)     # load n
       mul    $v0,$v0,$v1    # calculcate n*fact(n-1)
-----------------------------------------------------------
L1:    lw     $ra,20($sp)    # restore ra
       lw     $fp,16($sp)    # restore frame pointer
       addiu  $sp,$sp,32     # pop frame from stack
       jr     $ra            # return
```

Fact Animation:

0x100	
0x0FC	
0x0F8	
0x0F4	
0x0F0	
0x0EC	
0x0E8	
0x0E4	
0x0E0	
0x0DC	
0x0D8	
0x0D4	
0x0D0	
0x0CC	
0x0C8	
0x0C4	

main's fp

main's sp

Fact Animation:

0x100	
0x0FC	saved argument 10
0x0F8	(filler to align to multiple of 8)
0x0F4	main's return address
0x0F0	main's frame pointer
0x0EC	(space for $a3)
0x0E8	(space for $a2)
0x0E4	(space for $a1)
0x0E0	(space for $a0)
0x0DC	
0x0D8	
0x0D4	
0x0D0	
0x0CC	
0x0C8	
0x0C4	

fact(10)

fact(10)'s fp

fact(10)'s sp

Fact Animation:

0x100	
0x0FC	saved argument 10
0x0F8	(filler to align to multiple of 8)
0x0F4	main's return address
0x0F0	main's frame pointer
0x0EC	(space for $a3)
0x0E8	(space for $a2)
0x0E4	(space for $a1)
0x0E0	(space for $a0)
0x0DC	saved argument 9
0x0D8	(fillller to align to multiple of 8)
0x0D4	fact(10)'s return address
0x0D0	fact(10)'s frame pointer
0x0CC	(space for $a3)
0x0C8	(space for $a2)
0x0C4	(space for $a1)

fact(10)

fact(9)

fact(9)'s fp

fact(9)'s sp (0x0C0)

Notes:

- Frame pointers aren't necessary:
  - can calculate variable offsets relative to $sp
  - this works until values of unknown size are allocated on the stack (e.g., via alloca.)
  - furthermore, debuggers like having saved frame pointers around (can crawl up the stack).
- There are 2 conventions for the MIPS:
  - GCC: uses frame pointer
  - SGI: doesn't use frame pointer

Varargs

The convention is designed to support functions in C such as printf or scanf that take a variable number of arguments.

In particular, the callee can always write out $a0-$a3 and then has a contiguous vector of arguments.

In the case of printf, the 1st argument is a pointer to a string describing how many other arguments were pushed on the stack (hopefully.)

How to Compile a Procedure:

- Need to generate prologue & epilogue
  - need to know how much space frame occupies.
  - roughly $c + 4*v$ where c is the constant overhead to save things like the caller's frame pointer, return address, etc. and v is the number of local variables (including params.)
- When translating the body, we need to know the offset of each variable.
  - Keep an environment that maps variables to offsets.
  - Access variables relative to the frame pointer.
- When we encounter a return, need to move the result in to $v0 and jump to the epilogue.
  - Keep epilogue's label in environment as well.

Environments:

```
type varmap
val empty_varmap : unit -> varmap
val insert_var : varmap -> var -> int ->
                              varmap
val lookup_var : varmap -> var -> int

type env = {epilogue : label,
            varmap : varmap}
```

How to Implement Varmaps?

One option:

```
type varmap = var -> int

exception NotFound

fun empty_varmap() = fn y => raise NotFound

fun insert_var vm x i =
  fn y => if (y = x) then i else vm y

fun lookup_var vm x = vm x
```

Other options?

- Immutable Association list: (var * int) list
  - O(1) insert, O(n) lookup, O(1) copy, O(n) del
- Mutable Association list:
  - O(1) insert, O(n) lookup, O(n) copy, O(1) del
- Hashtable
  - O(1) insert, O(1) lookup, O(n) copy, O(1) del
- Immutable Balanced tree (e.g., red/black):
  - O(lg n) insert, O(lg n) lookup, O(1) copy, O(lg n) del

What about temps?

Option 1 (do this or option 2 or 3 for next project):

- when evaluating a compound expression x + y:
  - generate code to evaluate x and place it in $v0, then push $v0 on the stack.
  - generate code to evaluate y and place it in $v0.
  - pop x's value into a temporary register (e.g., $t0).
  - add $t0 and $v0 and put the result in $v0.
- Bad news: lots of overhead for individual pushes and pops.
- Good news: don't have to do any pre- or post-processing to figure out how many temps you need, and it's dirt simple.

For Example: 20 instructions

```
a  :=  (x + y) + (z + w)
lw    $v0, <xoff>($fp)    #       evaluate x
push $v0                  #       push x's value
lw    $v0, <yoff>($fp)    #       evaluate y
pop   $v1                 #       pop x's value
add   $v0,$v1,$v0         #    add x and y's values
push $v0                  #    push value of x+y
lw    $v0, <zoff>($fp)    #       evaluate z
push $v0                  #       push z's value
lw    $v0, <woff>($fp)    #       evaluate w
pop   $v1                 #       pop z's value
add   $v0,$v1,$v0         #    add z and w's values
pop   $v1                 #    pop x+y
add   $v0,$v1,$v0         # add (x+y) and (z+w)'s values
sw    $v0,<aoff>($fp)     # store result in a
```

Option 2:

- We have to push every time we have a nested expression.
- So eliminate nested expressions!
  - Introduce new variables to hold intermediate results
- For example, **`a := (x + y) + (z + w)`** might be translated to:

```
t0 := x + y;
t1 := z + w;
a := t0 + t1;
```

- Add the temps to the local variables.
  - So we allocate space for temps once in the prologue and deallocate the space once in the epilogue.

12 instructions (9 memory)

```
t0 := x + y;        lw $v0, <xoff>($fp)
                    lw $v1, <yoff>($fp)
                    add $v0, $v0, $v1
                    sw $v0, <t0off>($fp)
t1 := z + w;        lw $v0, <zoff>($fp)
                    lw $v1, <woff>($fp)
                    add $v0, $v0, $v1
                    sw $v0, <t1off>($fp)
a := t0 + t1;       lw $v0, <t0off>($fp)
                    lw $v1, <t1off>($fp)
                    add $v0, $v0, $v1
                    sw $v0, <aoff>($fp)
```

Still…

We're doing a lot of stupid loads and stores.

- We shouldn't need to load/store from temps!
- (Nor variables, but we'll deal with them later…)

So another idea is to use registers to hold the intermediate values instead of variables.

- For now, assume we have an infinite # of registers.
- We want to keep a distinction between temps and variables: variables require loading/storing, but temps do not.

For example:

```
t0 := x;          # load variable
t1 := y;          # load variable
t2 := t0 + t1;    # add
t3 := z;          # load variable
t4 := w;          # load variable
t5 := t3 + t4;    # add
t6 := t2 + t5;    # add
a := t6;          # store result
```

Then: 8 instructions (5 mem!)

- Notice that each little statement can be directly translated to MIPs instructions:

```
t0 := x;          -->  lw $t0,<xoff>($fp)
t1 := y;          -->  lw $t1,<yoff>($fp)
t2 := t0 + t1;    -->  add $t2,$t0,$t1
t3 := z;          -->  lw $t3,<zoff>($fp)
t4 := w;          -->  lw $t4,<woff>($fp)
t5 := t3 + t4;    -->  add $t5,$t3,$t4
t6 := t2 + t5;    -->  add $t6,$t2,$t5
a := t6;          -->  sw $t6,<aoff>($fp)
```

Recycling:

- Sometimes we can recycle a temp:

```
t0 := x;            t0 taken
t1 := y;            t0,t1 taken
t2 := t0 + t1;      t2 taken (t0,t1 free)
t3 := z;            t2,t3 taken
t4 := w;            t2,t3,t4 taken
t5 := t3 + t4;      t2,t5 taken (t3,t4 free)
t6 := t2 + t5;      t6 taken (t2,t5 free)
a := t6;            (t6 free)
```

Tracking Available Temps:

Aha! Use a *compile-time* stack of registers instead of a run-time stack…

```
t0 := x;              t0
t1 := y;              t1,t0
t0 := t0 + t1;        t0
t1 := z;              t1,t0
t2 := w;              t2,t1,t0
t1 := t1 + t2;        t1,t0
t1 := t0 + t1;        t1
a := t1;              <empty>
```

Option 3:

- When the compile-time stack overflows:
  - Generate code to "spill" (push) all of the temps.
  - (Can do one subtract on $sp).
  - Reset the compile-time stack to <empty>
- When the compile-time stack underflows:
  - Generate code to pop all of the temps.
  - (Can do one add on $sp).
  - Reset the compile-time stack to full.
- So what's really happening is that we're caching the "hot" end of the run-time stack in registers.
  - Some architectures (e.g., SPARC, Itanium) can do the spilling/restoring with 1 instruction.

Pros and Cons:

Compared to the previous approach:

- We don't end up pushing/popping when expressions are small.
- Eliminates a lot of memory traffic and amortizes the cost of stack adjustment.

But it's still far from optimal:

- Consider a+(b+(c+(d+…+(y+z)…))) versus (…((((a+b)+c)+d)+ … +y)+z.
- If order of evaluation doesn't matter, then we want to pick one that minimizes the depth of the stack (less likely to overflow.)