Procedure Calls

(Part 2)

March 4, 2013

Schedule for the rest of the quarter . . .

Assignments

- PA3 – due tonight at 11:59pm!
- PA4 – posted tomorrow, due Wed, March 13
- HW2 – posted Friday, due in class Fri, March 15
  (this will be very short)

Comprehensive final exam

- Tues, Mar 19, Noon-2pm
- in this room (STAG 203)
- review on Fri, March 15

Outline

Recursion

Memoization

Function pointers

Calling conventions review

Recursive algorithms

A recursive algorithm consists of two parts

1. **base case(s)**
   - “trivial” cases — usually just return a value
2. **recursive case**
   - “typical case” — defined in terms of a call to itself
   - recursive call should make progress toward the base case

Example: Computing factorials

1. fact(0) $\Rightarrow$ 1
2. fact(n) $\Rightarrow$ $n \times \text{fact}(n-1)$

Functional view of recursion

Example: Computing factorials

1. $\text{fact}(0) \;\Rightarrow\; 1$
2. $\text{fact}(n) \;\Rightarrow\; n \times \text{fact}(n-1)$

$$
\begin{array}{ll}
\text{fact}(5) & \\
5 \times \text{fact}(4) & \\
5 \times 4 \times \text{fact}(3) & \\
5 \times 4 \times 3 \times \text{fact}(2) & \\
5 \times 4 \times 3 \times 2 \times \text{fact}(1) & \\
5 \times 4 \times 3 \times 2 \times 1 \times \text{fact}(0) & \\
5 \times 4 \times 3 \times 2 \times 1 \times 1 & = 120
\end{array}
$$

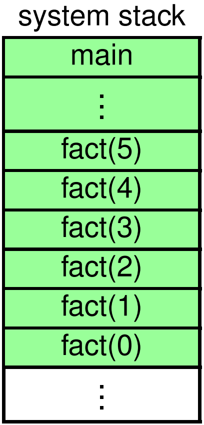
Imperative view of recursion

Factorial in pseudocode

```
# int fact(int n) {
#   if (n == 0) return 1
#   return n * fact(n-1)
# }
```

Expanded pseudocode

```
# int fact(int n) {
#   if (n == 0) return 1
#   m = fact(n-1)
#   m = n * m
#   return m
# }
```



system stack

main

⋮

fact(5)

fact(4)

fact(3)

fact(2)

fact(1)

fact(0)

⋮

Each recursive call pushes a new stack frame*

Really important to get calling conventions right!

*can avoid with tail recursion

Recursion in assembly

Recursive functions in assembly

- nothing special!
    - just `jal` to the same procedure
- calling conventions doubly important
    - potentially many stack frames
    - procedure will step on its own toes

(MARS demo: FactRec.asm)

Outline

Recursion

Memoization

Function pointers

Calling conventions review

Memoization

An **optimization** technique for recursive functions

- maintain a *global array* of previously computed values
- on each procedure call, lookup in array
    - if already computed, return it
    - otherwise, proceed as usual and *save result* in array

Neat trick:

- can often handle base cases by just pre-initializing the first few values in the array

Memoization strategy

Sketch of memoized recursive function

In data segment:

- declare array **memo** with length $\geq$ largest input
- possibly initialize base cases

Definition of **fun(n)** in text segment:

1. check if **memo[n]** is set
    - if yes, return **memo[n]**
2. (no) compute **fun(n)** as usual
3. store result in **memo[n]**
4. return result

(MARS demo: FactMemo.asm)

Memoization grab bag

Can't use memoization if . . .

- recursive function is **not pure**
  - it does I/O, sets global variables, etc.
- input does not map onto array indexes

Gotcha: Can your function produce 0?

- if so, need a smarter check than **if (memo[n] != 0)**

Big win: memoize functions with **multiple recursion**

- $\text{fib}(0) \Rightarrow 0$
- $\text{fib}(1) \Rightarrow 1$
- $\text{fib}(n) \Rightarrow \text{fib}(n-2) + \text{fib}(n-1)$

Outline

Recursion

Memoization

Function pointers

Calling conventions review

Function pointers

- In MIPS, a procedure is identified by an **address**
- When we say, `jal myProcedure`, we're saying:
  "jump and link to the address at label `myProcedure`"
- We can jump and link to an address in a register too!
  - example: `jalr $t0`
  - can pass addresses around, store them in arrays,
    or do whatever – they're just like other values

Jump and link register

To call a function pointer: `jalr $t5`

`jalr` – “jump and link register”

1. sets `$ra` to PC+4 (just like `jal`)
   - save the address of the next instruction of the caller
2. sets PC to the value in `$t5` (`$t5` can be any register)
   - jump to the address of the first instruction of the callee

Otherwise, exactly like any other procedure call!

Function pointers in pseudocode

Syntax of a function pointer in C

- **`int (*foo)(int)`**
  - **`*foo`** is a pointer to a function from **`int`** to **`int`**
- **`int x = (*foo)(n)`**
  - apply the function foo points at to **`n`**

Example in C-like pseudocode

```
# Pseudocode:
#   void test(int (*foo)(int), int n) {
#     int x = (*foo)(n)
#     printInt(x)
#   }
#
#   void main() {
#     test(&myProc, 5)
#   }
```

Function pointers in pseudocode

Syntax of a function pointer in C

- `int (*foo)(int)`
    - `*foo` is a pointer to a function from `int` to `int`
- `int x = (*foo)(n)`
    - apply the function foo points at to `n`

Kind of tricky to get right . . . OK to fudge it, as long as it's clear

Example in simpler pseudocode

```
# Pseudocode:
#   void test(foo, int n) {
#     x = foo(n)
#     printInt(x)
#   }
```

Example

(MARS demo: FunPointers.asm)

Outline

Recursion

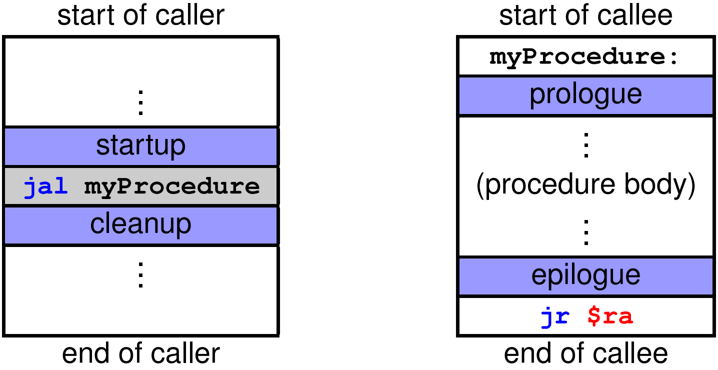
Memoization

Function pointers

Calling conventions review

Subroutine linkage

The boilerplate code related to the calling conventions



What to do in the caller

Caller startup sequence

1. save non-$s registers needed after call (local var section)
2. setup args to send to procedure ($a0–$a3, arg section)

Caller cleanup sequence

1. retrieve result of procedure ($v0–$v1)
2. restore non-$s registers saved in startup

```
# Pseudocode: ... x = myProcedure(n) ...
# Registers: n => $t0, x = $t1
  ...
sw    $t0, 20($sp)  # (startup) save n
move  $a0, $t0      # setup arg = n
jal   myProcedure   # myProcedure(arg)
move  $t1, $v0      # save result in x
lw    $t0, 20($sp) # (cleanup) restore n
  ...
```

What to do in the callee

Callee procedure prologue

1. retrieve arguments from stack (prev arg section)
2. push new stack frame
3. save `$s` registers used in body (saved register section)
4. save `$ra` (return address)

Callee procedure epilogue

1. restore `$s` registers saved in prologue
2. restore `$ra`
3. pop stack frame

What to do in the callee

```
myProcedure:
  addiu $sp, $sp, -24    # push stack frame
  sw    $ra, 20($sp)     # save $ra
  sw    $s0, 16($sp)     # save $s0
    ...
    (procedure body that uses $s0)
    ...
  lw    $s0, 16($sp)     # restore $s0
  lw    $ra, 20($sp)     # restore $ra
  addiu $sp, $sp, 24     # pop stack frame
  jr    $ra              # return
```

Responsibilities of a procedure

Remember: non-leaf procedure can be both a callee and caller!

```
myProcedure:
  # (procedure prologue, as callee)
  ...
  # (caller startup)
  jal subRoutine1
  # (caller cleanup)
  ...
  # (caller startup)
  jal subRoutine2
  # (caller cleanup)
  ...
  # (procedure epilogue, as callee)
  jr  $ra
```