# 6.001 Recitation 23: Register Machines and Stack Frames

RI: Gerald Dalley, dalleyg@mit.edu, 8 May 2007
http://people.csail.mit.edu/dalleyg/6.001/SP2007/

## Miscellany

- `apply`: See the MIT Scheme documentation for a precise description.
- *Register machines:* read the textbook and/or study the lecture notes. We will only have time to touch on a few key points in recitation.
- *Project 5:* `printf` prints values to the screen. It does not return a value.

## Expression Types

`(const C)`: A constant value. It acts somewhat like quote. To get the number one, you would use `(const 1)`. To get the list `(1 2)` you would use `(const (1 2))`.

`(reg R)`: Retrieve the value of a register `R`. To get the value of the register `arg0`, you would use `(reg arg0)`.

`(label L)`: Retrieve the offset of the given label `L`. To get the value of label `loop-top`, you would use `(label loop-top)`.

`(op O)`: Perform operation `O` on some values. Following the `(op O)`, you should list the input arguments to the operation, which may be constants, registers, or labels. An expression may only contain 1 operation (*i.e.* nested expressions are not allowed). In order to compute the result of adding 1 to the register `arg0`, you would use `(op +) (reg arg0) (const 1)`. For this recitation, you may assume the following operations are available: `+`, `-`, `*`, `/`, `<`, `<=`, `=`, `>`, and `>=` (nearly all real CPUs have these operations built-in).

## Some Special Registers

`pc`: The program counter register `pc` tells the sequencer what instruction is currently being evaluated.

`cr`: The condition register is used to determine whether to take branches. Use a `(test expr)` instruction to set its value.

`sp`: Pointer to the top of the stack. The stack is a fixed-sized memory area for saving and restoring values.

## Instructions

`(assign reg expr)`: Sets register `reg` to be the result of expression `expr`. The assigned register doesn't need a tag because it is always a register being assigned. For example, to increment the result register:
`(assign result (op +) (reg result) (const 1))`.

`(test expr)`: This is equivalent to assigning the `cr`. The `cr` register is used to determine whether to take a `branch`. For example, to set the `cr` based on whether the register `x` is less than `10`:
`(test (op <) (reg x) (const 10))`

`(goto expr)`: Sets the `pc` to be the result of `expr`, which is usually a label or a register. Effectively continues the execution at another point in the code. To jump to the label `loop-top`:
`(goto (label loop-top))`

`(branch expr)`: If the value in the `cr` is true, acts like a `goto`. Otherwise it does nothing. To conditionally jump to the label `loop-done`:
`(branch (label loop-done))`

`(save reg)`: Place the value in a register `reg` on top of the stack. This will also increment the stack pointer `sp`. If the stack has no space left, a "stack overflow" error is signalled. To place the value in the register `result` on the stack: `(save result)`.

`(restore reg)` Take the top value off the stack and put it in the register `reg`. This will also decrement the stack pointer `sp`. If the stack has nothing on it, a "stack underflow" error is signalled. To remove the top element of the stack and place it in the register `result`: `(restore result)`.

## Writing Code

Write `double`: code to compute $2x$, given $x$ in `arg0`, and leave the output in `result`.

```
double
  (assign result (op *) (reg arg0) (const 2))
  (goto (reg continue))
```

Write `func`: code to compute $x^2 + y$, given $x$ in `arg0`, $y$ in `arg1`, and leave the output in `result`.

```
func
```

## General Contracts

When we first started learning Scheme, we initially evaluated isolated expressions and then quickly started working with procedure abstractions. With our register machines, we also want to build abstractions. Since the register machine is much lower-level, it's very important to clearly document the expectations and constraints of our subroutines. One way to do so is to specify the contract:

**Input**: Register(s) whose value is read and used before it is written.
**Output**: Register(s) designated as output.
**Modifies**: Register(s) whose value after the code block *could* differ from their original value.

## Interpreting Iterative Code

Consider the following code:

```
foo
  (assign a (const 1))
  (assign b (const 1))
bar
  (test (op <=) (reg n) (const 2))
  (branch (reg continue))
  (assign tmp (reg a))
  (assign a (reg b))
  (assign b (op +) (reg tmp) (reg a))
  (assign n (op -) (reg n) (const 1))
  (goto (label bar))
```

What is the contract:

- Input:
- Output:
- Modifies:

What does the code do?

Implement an equivalent procedure in Scheme:

## Subroutine Contracts

- **Input:** What register(s) hold the input values and the return point?
- **Output:** What register(s) will hold the output value (often `result`)?
- **Modifies:** What non-output register(s) might be modified?
- **Stack:** What happens to the stack?

## Subroutine Call Conventions

There are actually a number of design decisions to make when creating conventions for how to call a subroutine. For a given system, it's highly beneficial to come up with a boilerplate subroutine contract that all (normal) subroutines obey. Here's one such convention:

- Making the call
  1. Save the registers you don't want clobbered (including `continue`).
  2. Assign values to procedure input regs
  3. Assign the `continue` register to the return label.
  4. `goto` the start of the procedure.
- At the call's return label
  1. Use the output
  2. Restore the registers (in reverse order)

Another "calling convention" might be to have the callee (the procedure being called) save and restore everything that is important that it might modify, have the caller put the return address on the top of the stack, and have the callee remove the return address from the stack but otherwise leave the stack unchanged. Other conventions are possible (and each convention comes with its tradeoffs).
A "call frame" is the section of the stack which corresponds to a procedure call.

## Iterative Exponentiation

Consider an iterative implementation of exponentiation:

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1)
                   (* b product))))
  (expt-iter n 1))
```

Write the register machine code implementation:

What is the contract:
- Input:
- Output:
- Modifies:
- Stack:

## Recursive Exponentiation

Consider a recursive implementation of exponentiation:

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

Write the register machine code implementation:

What is the contract:

- Input:
- Output:
- Modifies:
- Stack:

Can you think of any optimizations that would speed up your procedure and/or allow for fewer lines of code?

Trace the execution of the register machine:

| pc | nextPC | cr | continue | b | n | result | stack |
|---|---|---|---|---|---|---|---|
| ??? | 1 | ??? | foo | 2 | 3 | ??? | empty |
| 1 | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

# Solutions

## Miscellany

- `apply`: See the MIT Scheme documentation for a precise description.
- *Register machines:* read the textbook and/or study the lecture notes. We will only have time to touch on a few key points in recitation.
- *Project 5:* `printf` prints values to the screen. It does not return a value.

## Expression Types

`(const C)`: A constant value. It acts somewhat like quote. To get the number one, you would use `(const 1)`. To get the list `(1 2)` you would use `(const (1 2))`.

`(reg R)`: Retrieve the value of a register `R`. To get the value of the register `arg0`, you would use `(reg arg0)`.

`(label L)`: Retrieve the offset of the given label `L`. To get the value of label `loop-top`, you would use `(label loop-top)`.

`(op O)`: Perform operation `O` on some values. Following the `(op O)`, you should list the input arguments to the operation, which may be constants, registers, or labels. An expression may only contain 1 operation (*i.e.* nested expressions are not allowed). In order to compute the result of adding 1 to the register `arg0`, you would use `(op +) (reg arg0) (const 1)`. For this recitation, you may assume the following operations are available: `+`, `-`, `*`, `/`, `<`, `<=`, `=`, `>`, and `>=` (nearly all real CPUs have these operations built-in).

## Some Special Registers

`pc`: The program counter register `pc` tells the sequencer what instruction is currently being evaluated.

`cr`: The condition register is used to determine whether to take branches. Use a `(test expr)` instruction to set its value.

`sp`: Pointer to the top of the stack. The stack is a fixed-sized memory area for saving and restoring values.

## Instructions

`(assign reg expr)`: Sets register `reg` to be the result of expression `expr`. The assigned register doesn't need a tag because it is always a register being assigned. For example, to increment the result register:
`(assign result (op +) (reg result) (const 1))`.

`(test expr)`: This is equivalent to assigning the `cr`. The `cr` register is used to determine whether to take a `branch`. For example, to set the `cr` based on whether the register `x` is less than `10`:
`(test (op <) (reg x) (const 10))`

`(goto expr)`: Sets the `pc` to be the result of `expr`, which is usually a label or a register. Effectively continues the execution at another point in the code. To jump to the label `loop-top`:
`(goto (label loop-top))`

`(branch expr)`: If the value in the `cr` is true, acts like a `goto`. Otherwise it does nothing. To conditionally jump to the label `loop-done`:
`(branch (label loop-done))`

`(save reg)`: Place the value in a register `reg` on top of the stack. This will also increment the stack pointer `sp`. If the stack has no space left, a "stack overflow" error is signalled. To place the value in the register `result` on the stack: `(save result)`.

`(restore reg)` Take the top value off the stack and put it in the register `reg`. This will also decrement the stack pointer `sp`. If the stack has nothing on it, a "stack underflow" error is signalled. To remove the top element of the stack and place it in the register `result`: `(restore result)`.

## Writing Code

Write `double`: code to compute $2x$, given $x$ in `arg0`, and leave the output in `result`.

```
double
  (assign result (op *) (reg arg0) (const 2))
  (goto (reg continue))
```

Write `func`: code to compute $x^2 + y$, given $x$ in `arg0`, $y$ in `arg1`, and leave the output in `result`.

```
func
  (assign tmp    (op *) (reg arg0) (reg arg0))
  (assign result (op +) (reg tmp)  (reg arg1))
  (goto (reg continue))
```

## General Contracts

When we first started learning Scheme, we initially evaluated isolated expressions and then quickly started working with procedure abstractions. With our register machines, we also want to build abstractions. Since the register machine is much lower-level, it's very important to clearly document the expectations and constraints of our subroutines. One way to do so is to specify the contract:

**Input**: Register(s) whose value is read and used before it is written.
**Output**: Register(s) designated as output.
**Modifies**: Register(s) whose value after the code block *could* differ from their original value.

## Interpreting Iterative Code

Consider the following code:

```
foo
  (assign a (const 1))
  (assign b (const 1))
bar
  (test (op <=) (reg n) (const 2))
  (branch (reg continue))
  (assign tmp (reg a))
  (assign a (reg b))
  (assign b (op +) (reg tmp) (reg a))
  (assign n (op -) (reg n) (const 1))
  (goto (label bar))
```

What is the contract:

- Input: `n, continue`
- Output: `b`
- Modifies: `a, n, tmp`

What does the code do?

It computes the $n^{th}$ Fibonacci number.

Implement an equivalent procedure in Scheme:

```
; This is a very direct translation (considered bad style for Scheme)
(define (fib n)
  (define result 1)
  (define prev   1)
  (define tmp    'undefined)
  (define (helper)
    (if (<= n 2)
        result
        (begin
          (set! tmp prev)
          (set! prev result)
          (set! result (+ tmp prev))
          (set! n (- n 1))
          (helper))))
  (helper))

; Here's a more stylistic translation
(define (fib n)
  (define (helper prev result)
    (if (<= n 2)
        result
        (begin
          (set! n (- n 1))
          (helper result (+ prev result)))))
  (helper 1 1))
```

## Subroutine Contracts

- **Input:** What register(s) hold the input values and the return point?
- **Output:** What register(s) will hold the output value (often `result`)?
- **Modifies:** What non-output register(s) might be modified?
- **Stack:** What happens to the stack?

## Subroutine Call Conventions

There are actually a number of design decisions to make when creating conventions for how to call a subroutine. For a given system, it's highly beneficial to come up with a boilerplate subroutine contract that all (normal) subroutines obey. Here's one such convention:

- Making the call
  1. Save the registers you don't want clobbered (including `continue`).
  2. Assign values to procedure input regs
  3. Assign the `continue` register to the return label.
  4. `goto` the start of the procedure.
- At the call's return label
  1. Use the output
  2. Restore the registers (in reverse order)

Another "calling convention" might be to have the callee (the procedure being called) save and restore everything that is important that it might modify, have the caller put the return address on the top of the stack, and have the callee remove the return address from the stack but otherwise leave the stack unchanged. Other conventions are possible (and each convention comes with its tradeoffs).
A "call frame" is the section of the stack which corresponds to a procedure call.

## Iterative Exponentiation

Consider an iterative implementation of exponentiation:

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1)
                   (* b product))))
  (expt-iter n 1))
```

Write the register machine code implementation:

```
; This implementation is basically a direct translation
;    Input: b n continue
;   Output: product
; Modifies: counter product
;    Stack: unchanged
expt
  (assign product (const 1))
  (assign counter (reg n))
expt-loop
  ; (if (= counter 0) product
  (test   (op =) (reg counter) (const 0))
  (branch (reg continue))

  (assign counter (op -) (reg counter) (const 1))
  (assign product (op *) (reg b)       (reg product))
  (goto (label expt-loop))

; We could also try to tighten up the contract, sacrificing some readability
; while removing the need for the counter register.
;    Input: b n
;   Output: result
; Modifies: n result
;    Stack: unchanged
expt
  (assign result (const 1))
expt-loop
  (test   (op =) (reg n) (const 0))
  (branch (reg continue))

  (assign n      (op -) (reg n) (const 1))
  (assign result (op *) (reg b) (reg result))
  (goto (label expt-loop))
```

What is the contract:

- Input: see answer in code
- Output: see answer in code
- Modifies: see answer in code
- Stack: see answer in code

## Recursive Exponentiation

Consider a recursive implementation of exponentiation:

```
  (define (expt b n)
    (if (= n 0)
        1
        (* b (expt b (- n 1)))))
```

Write the register machine code implementation:

```
   expt
       (test   (op =) (reg n) (const 0))
       (branch (label expt-base-case)) ; (if (= n 0)

       ; Compute (* b (expt b (- n 1))
       (assign n (op -) (reg n) (const 1))
       (save   b)
       (save   continue)
       (assign continue (label expt-after-recursion))
       (goto   (label expt)) ; (expt b (- n 1))
   expt-after-recursion ; result holds value from (expt b (- n 1))
       (restore continue)
       (restore b)
       (assign result (op *) (reg b) (reg result)) ; (* b ...)
       (goto (reg continue))

   expt-base-case
       (assign result (const 1))
       (goto (reg continue))

; Possible Optimizations:
;   The best optimization is to rewrite the algorithm as iterative instead
;   of recursive, but for now let's assume you still want a recursive one...
;
;   Note that we don't actually modify b anywhere.  We could relax our
;   "save anything you don't want clobbered" convention here and not
;   save and restore b.
;
```

What is the contract:

- Input: `b,n,continue`
- Output: `result`
- Modifies: `n`
- Stack: unchanged

Can you think of any optimizations that would speed up your procedure and/or allow for fewer lines of code?

see comments in the code

Trace the execution of the register machine:

Solution note: mutated values are highlighted for the convenience of the reader.

| pc | nextPC | cr | continue | b | n | result | stack |
|---|---|---|---|---|---|---|---|
| | 1 | ??? | foo | 2 | 3 | ??? | *empty* |
| 1 | 2 | **#f** | foo | 2 | 3 | ??? | *empty* |
| 2 | 3 | #f | foo | 2 | 3 | ??? | *empty* |
| 3 | 4 | #f | foo | 2 | **2** | ??? | *empty* |
| 4 | 5 | #f | foo | 2 | 2 | ??? | **2** |
| 5 | 6 | #f | foo | 2 | 2 | ??? | **foo 2** |
| 6 | 7 | #f | **expt-after-recursion** | 2 | 2 | ??? | foo 2 |
| 7 | 1 | #f | expt-after-recursion | 2 | 2 | ??? | foo 2 |
| 1 | 2 | **#f** | expt-after-recursion | 2 | 2 | ??? | foo 2 |
| 2 | 3 | #f | expt-after-recursion | 2 | 2 | ??? | foo 2 |
| 3 | 4 | #f | expt-after-recursion | 2 | **1** | ??? | foo 2 |
| 4 | 5 | #f | expt-after-recursion | 2 | 1 | ??? | **2 foo 2** |
| 5 | 6 | #f | expt-after-recursion | 2 | 1 | ??? | **e-a-r 2 foo 2** |
| 6 | 7 | #f | **expt-after-recursion** | 2 | 1 | ??? | e-a-r 2 foo 2 |
| 7 | 1 | #f | expt-after-recursion | 2 | 1 | ??? | e-a-r 2 foo 2 |
| 1 | 2 | **#f** | expt-after-recursion | 2 | 1 | ??? | e-a-r 2 foo 2 |
| 2 | 3 | #f | expt-after-recursion | 2 | 1 | ??? | e-a-r 2 foo 2 |
| 3 | 4 | #f | expt-after-recursion | 2 | **0** | ??? | e-a-r 2 foo 2 |
| 4 | 5 | #f | expt-after-recursion | 2 | 0 | ??? | **2 e-a-r 2 foo 2** |
| 5 | 6 | #f | expt-after-recursion | 2 | 0 | ??? | **e-a-r 2 e-a-r 2 foo 2** |
| 6 | 7 | #f | **expt-after-recursion** | 2 | 0 | ??? | e-a-r 2 e-a-r 2 foo 2 |
| 7 | 1 | #f | expt-after-recursion | 2 | 0 | ??? | e-a-r 2 e-a-r 2 foo 2 |
| 1 | 2 | **#t** | expt-after-recursion | 2 | 0 | ??? | e-a-r 2 e-a-r 2 foo 2 |
| 2 | 12 | #t | expt-after-recursion | 2 | 0 | ??? | e-a-r 2 e-a-r 2 foo 2 |
| 12 | 13 | #t | expt-after-recursion | 2 | 0 | **1** | e-a-r 2 e-a-r 2 foo 2 |
| 13 | 8 | #t | expt-after-recursion | 2 | 0 | 1 | e-a-r 2 e-a-r 2 foo 2 |
| 8 | 9 | #t | **expt-after-recursion** | 2 | 0 | 1 | **2 e-a-r 2 foo 2** |
| 9 | 10 | #t | expt-after-recursion | **2** | 0 | 1 | **e-a-r 2 foo 2** |
| 10 | 11 | #t | expt-after-recursion | 2 | 0 | **2** | e-a-r 2 foo 2 |
| 11 | 8 | #t | expt-after-recursion | 2 | 0 | 2 | e-a-r 2 foo 2 |
| 8 | 9 | #t | **expt-after-recursion** | 2 | 0 | 2 | **2 foo 2** |
| 9 | 10 | #t | expt-after-recursion | **2** | 0 | 2 | **foo 2** |
| 10 | 11 | #t | expt-after-recursion | 2 | 0 | **4** | foo 2 |
| 11 | 8 | #t | expt-after-recursion | 2 | 0 | 4 | foo 2 |
| 8 | 9 | #t | **foo** | 2 | 0 | 4 | **2** |
| 9 | 10 | #t | foo | **2** | 0 | 4 | ***empty*** |
| 10 | 11 | #t | foo | 2 | 0 | **8** | *empty* |
| 11 | **foo** | #t | foo | 2 | 0 | 8 | *empty* |