Lab 11
JAM2000, translating Scheme to C

Comp 210

Spring 1996

Contents

1 Running the JAM2000 simulator

For your homework, you will need to run the JAM2000 machine simulator. DrScheme does not support the simulator, so you must must use Chez Scheme. You can run Chez under Emacs (via M-x run-scheme) or from a shell (by typing `scheme`).

To load and start the simulator, type

(*load* "~comp210/Hw/hw9/jam.ss")

You will see a prompt such as

```
[00000]:  00000000 (halt) %
```

There are three parts to this prompt. First comes the current address (in brackets, followed by a colon): you are examining memory location 0. Second is the contents of the current address: this memory location contains 00000000. Third is a mnemonic description of the contents, interpreted as a machine instruction (followed by a percent sign): 00000000 stands for the "halt" instruction.

Type "h" (and press return) for a list of commands. To set the currently-examined memory location, just type a number. To set an arbitrary memory location, move to that location (using the "p" command) and then set it. Since setting a memory location also advances you to the next location, you can enter an entire program by moving to the location that contains its first element and then typing the numbers representing the program one after the other. You can also enter mnemonics instead of raw numbers: for instance, typing `(bnez 3 1)` is equivalent to typing `134`.

If you run the simulator under Emacs, then you can only enter one piece of information at a time. For instance, you would have to type `p RET 9 RET` (where `RET` stands for the return key) instead of `p 9 RET` to set the program counter to 9, and would have to type `149 RET 9 RET 419 RET` instead of `149 9 419 RET` to set three memory locations. (Another way of stating this restriction is that your commands can't contain spaces.) If you do include spaces in a command when running under Emacs, the simulator will just wait until you either interrupt it or back up over all but the first piece of information, then press return again.

If you get stuck, you can interrupt execution by typing C-c (in Emacs, type C-c twice); restart the simulation, after you exit or it errs, by executing (*walk*).

We will practice by typing in and running the program in file `~comp210/Hw/hw9/sample-1.a`. You should also try `sample-2.a`, but you will have to translate the labels into physical addresses yourself.

One thing to notice about JAM2000 (and every real computer) is that it has only one underlying representation — numbers — for all of its data. How to interpret a number depends on how it is being used. For instance, a particular memory location might contain 00000134. If we load that into a register and perform arithmetic on it, we are interpreting it as the number 134. If we set the PC to the location and start the machine, then we are interpreting the location's contents (00000134) as an instruction, namely `(bnez 3 1)`. Even in that mnemonic, the different numbers mean different things. The first number, 3, specifies a particular register — $R3$. The second number, 1, specifies a machine address to jump to if $R3$ contains the value 0. Some instructions, such as `ldi`, interpret an argument as simply a number. How to interpret the arguments to a machine instruction depends on the particular instruction.

More information on the JAM2000 machine architecture and a quick reference card for the JAM2000 instruction set are available on the Comp 210 WWW page.

2 Translating Scheme to C

Any calculation that can be performed in Scheme can also be performed in C; for every Scheme program, there exists a C program that does the same thing. Likewise, Scheme can express any computation that C can express. In some cases, the Scheme and C versions of a program are very similar, and we can easily translate between the two. In other cases, more substantial rewriting is required, particularly since C does not support some features that Scheme does, such as dynamic (run-time) creation of procedures and using a single variable to hold values of multiple types (at different times). Here we will examine the easier translations.

When translating Scheme into C, it is important to keep in mind the distinction in C between *expressions* and *statements*. A statement has no value, but generally has a side effect. An expression computes a value (and may have a side effect as well). Scheme makes no such distinction: it has only expressions, so every Scheme construct has a value. Some Scheme expressions also have side effects, and some of those side-effecting expressions produce an uninteresting value: you may think of such Scheme expressions as the equivalent of C statements. However, Scheme makes no syntactic distinction between expressions and statements, while C does.

A second concern when translating from Scheme to C is whether a value is a procedure or not. Generally speaking, Scheme expressions that do not involve procedure values (that is, procedures appear only at the head of applications, not as data) have a straightforward translation into C. Function-valued expressions, on the other hand, may require some manipulation in order to be translated.

A final, minor detail is that C does not permit any top-level statements or expressions: a C program consists of function and variable definitions, with an implicit call to the `main` procedure. The Scheme program may have to be manipulated to move top-level statements into a *main* function.

We will define the following operators:

$\mathcal{CE}[\![\text{foo}]\!]$	the translation of Scheme expression *foo* into a C expression
$\mathcal{CS}[\![\text{foo}]\!]$	the translation of Scheme expression *foo* into a C statement

We will assume that implicit **begin**s are everywhere made explicit, omit parenthesization in the C translation for brevity, and use v to stand for variables, e for expressions, d for definitions, and t for types.

2.1 Constants

Numbers:	translated unchanged, for numbers representable by C; use an approximation for ratios, very large numbers, etc.
Strings:	translated unchanged
Booleans:	$\mathcal{CE}[\![\#\mathsf{t}]\!]$ = `true` $\mathcal{CE}[\![\#\mathsf{f}]\!]$ = `false`
Symbols:	our C dialect does not support an analog to Scheme symbols
Lists:	our C dialect does not support an analog to Scheme lists

2.2 Variables

$\mathcal{CE}[\![v]\!]$	=	`v`	variables are translated unchanged
$\mathcal{CS}[\![(\textbf{define}\ v\ e)]\!]$	=	`t v =` $\mathcal{CE}[\![e]\!]$`;`	`t` is the (non-procedure) type of $\mathcal{CE}[\![e]\!]$, and thus of `v`
$\mathcal{CS}[\![(\textbf{set!}\ v\ e)]\!]$	=	`v =` $\mathcal{CE}[\![e]\!]$`;`	$\mathcal{CE}[\![e]\!]$ and `v` have the same non-procedure type

Variables may need to be renamed to make them valid C identifiers, which may not contain punctuation (except for the underscore character); for instance, *a-b* is a valid Scheme variable name, but in C `a-b` calls for the subtraction, so we would use `a_b` instead.

$\mathcal{CS}[\![(\mathit{local}\ (d1\ \ldots\ dn)\ \mathit{body})]\!]$ = `{` $\mathcal{CS}[\![d1]\!]\ \ldots\ \mathcal{CS}[\![dn]\!]\ \mathcal{CS}[\![\mathit{body}]\!]$ `}` *di* are non-procedure definitions

The rule for *local* depends on whether the locals are in a procedure or not.

2.3 Arithmetic

$\mathcal{CE}[\![(op\ e1\ e2)]\!]$ = $\mathcal{CE}[\![e1]\!]$ `op` $\mathcal{CE}[\![e2]\!]$ where C's `op` corresponds to Scheme's *op*
Scheme may produce a more accurate result than C

2.4 Control flow

$\mathcal{CS}[\![(\textbf{if}\ e1\ e2\ e3)]\!]$	=	`if` $\mathcal{CE}[\![e1]\!]$ $\mathcal{CS}[\![e2]\!]$ `else` $\mathcal{CS}[\![e3]\!]$
$\mathcal{CS}[\![(\textbf{begin}\ e1\ \ldots\ en)]\!]$	=	`{` $\mathcal{CS}[\![e1]\!]\ \ldots\ \mathcal{CS}[\![en]\!]$ `}`

Convert **cond** forms into the equivalent **if**, then translate them to C.

2.5 Structures and arrays

$\mathcal{CS}[\![(\textit{define-structure}\ s\ f1\ \ldots\ fn)]\!]$	=	`struct s { t1 f1; ... tn fn; };`	`t`i is the type of field `f`i
$\mathcal{CE}[\![(\textit{s-fi}\ e)]\!]$	=	$\mathcal{CE}[\![e]\!]$`.fi`	*fi* is a field of structure *s*
$\mathcal{CS}[\![(\textit{set-s-fi!}\ e1\ e2)]\!]$	=	$\mathcal{CE}[\![e1]\!]$`.fi =` $\mathcal{CE}[\![e2]\!]$`;`	
$\mathcal{CE}[\![(\textit{make-vector}\ e1\ e2)]\!]$	=	`{` $\mathcal{CE}[\![e2]\!]$`, ...` $\mathcal{CE}[\![e2]\!]$ `}`	*e1* copies; *e1*, *e2* constants only in variable definitions
$\mathcal{CE}[\![(\textit{vector-ref}\ e1\ e2)]\!]$	=	$\mathcal{CE}[\![e1]\!]$`[`$\mathcal{CE}[\![e2]\!]$`]`	
$\mathcal{CS}[\![(\textit{vector-set!}\ e1\ e2\ e3)]\!]$	=	$\mathcal{CE}[\![e1]\!]$`[`$\mathcal{CE}[\![e2]\!]$`] =` $\mathcal{CE}[\![e3]\!]$	

C has no analog to Scheme's (*make-s* ...) where *s* is a structure. Instead, create a new variable of the type without initializing it, then set the fields individually and use the variable in place of the (*make-s* ...) expression.

$\mathcal{CE}[\![\ldots\ (\textit{make-s}\ e1\ e2)\ \ldots]\!]$ = `s v; v.f1 =` $\mathcal{CE}[\![e1]\!]$`; v.b =` $\mathcal{CE}[\![e2]\!]$`; ... v ...`

2.6 Functions

$\mathcal{CE}[\![(f\ e1\ \ldots\ en)]\!]$	=	`f(`$\mathcal{CE}[\![e1]\!]$`, ...` $\mathcal{CE}[\![en]\!]$`)`
$\mathcal{CS}[\![($**define** f (**lambda** $(v1\ \ldots\ vn)\ \textit{body}))]\!]$	=	`void f(t1 v1, ... tn vn) {` $\mathcal{CS}[\![\textit{body}]\!]$ `}`
$\mathcal{CS}[\![($**define** f (**lambda** $(v1\ \ldots\ vn)\ \textit{body}))]\!]$	=	`t0 f(t1 v1, ... tn vn) { return` $\mathcal{CE}[\![\textit{body}]\!]$`; }`

Select the appropriate rule for function definitions depending on whether the function returns a value or not; `t`i is the type of parameter `v`i and `t0` is the function's return type.

C does not support nested procedures, so an attempt to translate a locally-defined procedure, or any other use of **lambda** within a procedure, fails. Internal procedures should be lifted to top level, and additional parameters added for any internal variables they depended upon.

2.7 Expression statements

If $\mathcal{CE}[\![\textit{foo}]\!]$ is defined for some Scheme expression *foo*, but $\mathcal{CS}[\![\textit{foo}]\!]$ is not defined, then we can define $\mathcal{CS}[\![\cdot]\!]$ in terms of $\mathcal{CE}[\![\cdot]\!]$:

$\mathcal{CS}[\![\textit{foo}]\!]$ = $\mathcal{CE}[\![\textit{foo}]\!]$`;`

2.8 Values of if, begin, and *local* expressions

Scheme expressions do not always translate directly into C expressions; examples include **if**, **begin**, and *local* expressions. These often appear as the bodies of function definitions or in other contexts where a value is required (not just a side effect, which can be obtained from a C statement): the value supplied to **define** or **set!**, arguments to arithmetic operators, etc. In such situations, the body must be transformed in order to supply the requisite value. Our rules use `return` for concreteness, but they can be extended to other expression contexts; the rules duplicate the context. (We did a similar thing above for *make-s*, where *s* is a structure.)

`return` $\mathcal{CE}[\![($**if** $e1\ e2\ e3)]\!]$`;`	$\Rightarrow$	`if` $\mathcal{CE}[\![e1]\!]$ `return` $\mathcal{CE}[\![e2]\!]$`; else return` $\mathcal{CS}[\![e3]\!]$`;`
`return` $\mathcal{CE}[\![($**begin** $e1\ \ldots\ en\ e)]\!]$`;`	$\Rightarrow$	`{` $\mathcal{CS}[\![e1]\!]$ `...` $\mathcal{CS}[\![en]\!]$ `return` $\mathcal{CE}[\![e]\!]$`; }`
`return` $\mathcal{CE}[\![(\textit{local}\ (d1\ \ldots\ dn)\ e)]\!]$`;`	$\Rightarrow$	`{` $\mathcal{CS}[\![d1]\!]$ `...` $\mathcal{CS}[\![dn]\!]$ `return` $\mathcal{CE}[\![e]\!]$`; }`

Definitions are a special case, because C permits only a single definition for any particular variable, and because the location of the definition determines its context. When the definition would be duplicated or placed inside a compound statement (curly braces), instead define the variable without an initial value, then set its value:

`t v =` $\mathcal{CE}[\![e]\!]$`;` $\Rightarrow$ `t v; v =` $\mathcal{CE}[\![e]\!]$`;`

(This approach is always valid, but it is stylistically preferable to initialize variables at definition time if possible.) An alternate approach to avoiding placing a C variable definition inside a restricting scope (curly braces) is to rename local variables and omit the scope.

3 Exercises

Write the following procedures recursively; then convert them to tail-recursive form, into *while-fn* form, into C, and finally into JAM2000 machine language.

1. Compute the nth Fibonacci number.
2. Compute the sum of the first n natural numbers.
3. Compute the average of $\sin(0), \sin(1), \ldots, \sin(n)$.
4. Compute the average, minimum, and maximum of $\sin(0), \sin(1), \ldots, \sin(n)$.
5. Compute approximate integrals via the adaptive quadrature algorithm.

We will work out the fourth problem as an example, and will give a start toward the fifth.

3.1 Sine statistics

3.1.1 Recursive solution

Here is a recursive solution, which falls directly out of the program schema. (Don't forget program schemas; they come in handy in all sorts of situations!)

```
;; argument: a non-negative integer n
;; result: a list of the minimum, maximum, and average values of
;; sin(0), sin(1), ..., sin(n)
(define sin-stats-recursive
  (lambda (n)
    (if (< n 0)
        (list 1 -1 0)
        (local ((define recursive-result (sin-stats-recursive (- n 1)))
                (define recursive-min (car recursive-result))
                (define recursive-max (car (cdr recursive-result)))
                (define recursive-avg (car (cdr (cdr recursive-result))))
                (define this-sin (sin n)))
          (list (if (< this-sin recursive-min) this-sin recursive-min)
                (if (> this-sin recursive-max) this-sin recursive-max)
                (/ (+ this-sin (* recursive-avg n))
                   (+ n 1)))))))
```

There are several interesting things to notice about this definition. First, we have interesting work to do for the case of $n = 0$; for this problem, the base case is $n - 1$. In particular, for argument n, there are $n + 1$ calls to *sin* to be performed. You should get used to seeing base cases that are not always 0 or *null*.

Second, the repeated call to *sin* was lifted out of the body of function and made into a locally-defined variable, for efficiency. This version calls *sin* only $n + 1$ times when called on argument n.

Third, the recursive calls returns something that isn't just an ordinary number or list, as we are used to seeing. Thus, we need to decompose the result of the recursive call in order to get at the useful parts. We moved the recursive call into the definition part of the *local* in order to save repeated calls (the performance of this function goes from linear to exponential if repeated calls to *sin-stats-recursive* are made). We moved the extraction of the pieces from the value of the recursive call into the *local* definitions primarily to give them more descriptive names (but also to save a little time). It would arguably have been nicer to write this function to return a structure rather than a list; then we could get at the pieces with structure selectors instead of having to use (*car* (*cdr* ...)).

3.1.2 Tail-recursive solutions

We can convert our function to the following tail-recursive version. Notice that this version does not use **set!** at all; you should definitely get used to writing tail-recursive functions that do not use **set!**. The technique is to pass the new values as arguments to the tail-recursive function.

```
(define sin-stats-tail-recursive-1
  (lambda (n)
    (local ((define helper
                (lambda (counter min-so-far max-so-far avg-so-far)
                  (if (< counter 0)
                      (list min-so-far max-so-far avg-so-far)
                      (local ((define this-sin (sin counter)))
                        (helper (- counter 1)
                                (if (< this-sin min-so-far) this-sin min-so-far)
                                (if (> this-sin max-so-far) this-sin max-so-far)
                                (/ (+ this-sin (* avg-so-far (- n counter)))
                                   (+ (- n counter) 1))))))))
      (helper n 1 -1 0))))
```

The use of ($-$ *n* *counter*) reflects the fact that when this version computes values in the other direction from the original recursive function. When (*sin* x) is computed, only $n - x + 1$ values, not $x + 1$ values, have been computed, because (*sin* n) is executed before (*sin* ($-$ n 1)), and so forth. The original recursive function computed (*sin* 0), then (*sin* 1), and so forth. Here is a tail-recursive version that computes values in the upwards direction. It starts the internal counter at 0 and works it up to the argument n rather than vice-versa.

```
(define sin-stats-tail-recursive-2
  (lambda (n)
    (local ((define helper
                (lambda (counter min-so-far max-so-far avg-so-far)
                  (if (> counter n)
                      (list min-so-far max-so-far avg-so-far)
                      (local ((define this-sin (sin counter)))
                        (helper (+ counter 1)
                                (if (< this-sin min-so-far) this-sin min-so-far)
                                (if (> this-sin max-so-far) this-sin max-so-far)
                                (/ (+ this-sin (* avg-so-far counter))
                                   (+ counter 1))))))))
      (helper 0 1 -1 0))))
```

We could also have computed, instead of a running average, a running total. This version is more efficient because it performs only one division (instead of $n + 1$) and no multiplications.

```
(define sin-stats-tail-recursive-3
  (lambda (n)
    (local ((define helper
              (lambda (counter min-so-far max-so-far total-so-far)
                (if (< counter 0)
                    (list min-so-far max-so-far (/ total-so-far (+ n 1)))
                    (local ((define this-sin (sin counter)))
                      (helper (- counter 1)
                              (if (< this-sin min-so-far) this-sin min-so-far)
                              (if (> this-sin max-so-far) this-sin max-so-far)
                              (+ total-so-far this-sin)))))))
      (helper n 1 -1 0))))
```

This version is easy to convert to a tail-recursive form in which the helper function, instead of taking many arguments, takes just a single counter argument while side-effecting local variables that hold the other quantities.

```
(define sin-stats-tail-recursive-4
  (lambda (n)
    (local ((define min-so-far 1)
            (define max-so-far -1)
            (define total 0)
            (define helper
              (lambda (counter)
                (if (< counter 0)
                    (list min-so-far max-so-far (/ total (+ n 1)))
                    (local ((define this-sin (sin counter)))
                      (if (< this-sin min-so-far)
                          (set! min-so-far this-sin))
                      (if (> this-sin max-so-far)
                          (set! max-so-far this-sin))
                      (set! total (+ total this-sin))
                      (helper (- counter 1)))))))
      (helper n))))
```

This function definition uses the two-argument form of **if**. Ordinarily you provide three arguments to **if**; but if the last one would have been just (*void*), then you may omit it, and Scheme will automatically supply that for you.

We can even make the counter into a side-effected local variable, so the helper function takes no arguments at all (and returns no result; the caller must use the local variables to extract the values).

```
(define sin-stats-tail-recursive-5
  (lambda (n)
    (local ((define min-so-far 1)
            (define max-so-far -1)
            (define total 0)
            (define counter n)
            (define helper
              (lambda ()
```

```
            (if (>= counter 0)
                (local ((define this-sin (sin counter)))
                  (if (< this-sin min-so-far)
                      (set! min-so-far this-sin))
                  (if (> this-sin max-so-far)
                      (set! max-so-far this-sin))
                  (set! total (+ total this-sin))
                  (set! counter (- counter 1))
                  (helper)))))
      (helper)
      (list min-so-far max-so-far (/ total (+ n 1))))))
```

Notice that the counter is modified *after* the other quantities are. This isn't important in our case, because we lifted the *sin* computation into a local definition, but if we had used (*sin counter*) in place of *this-sin* and had set *counter* before performing those computations, the wrong thing would have happened. (If you don't understand why, then study the problem or run it under Donkey to see exactly what happens.)

3.1.3 While loop solution

This function is in exactly the form we need to convert it to a *while-fn* loop.

```
(define while-fn
  (lambda (test body)
    (if (test)
        (begin
          (body)
          (while-fn test body))
        (void))))

(define sin-stats-while
  (lambda (n)
    (local ((define min-so-far 1)
            (define max-so-far -1)
            (define total 0)
            (define counter n))
      (while-fn
       (lambda () (>= counter 0))
       (lambda ()
         (local ((define this-sin (sin counter)))
           (if (< this-sin min-so-far)
               (set! min-so-far this-sin))
           (if (> this-sin max-so-far)
               (set! max-so-far this-sin))
           (set! total (+ total this-sin))
           (set! counter (- counter 1)))))
      (list min-so-far max-so-far (/ total (+ n 1))))))
```

3.1.4 C solutions

Now we can convert the function to C. Here is an almost literal translation from the *while-fn* version.

```
#include <iostream.h>

main()
{
  double min = 1;
  double max = -1;
  double total = 0;
  int n;

  cout << "Enter a positive integer: ";
  cin >> n;

  int counter = n;

  while (counter >= 0)
    {
      double this_sin = sin(counter);
      if (this_sin < min)
        min = this_sin;
      if (this_sin > max)
        max = this_sin;
      total = total + this_sin;
      counter = counter - 1;
    }
  cout << "Statistics for sin(0) .. sin(" << n << "):" << endl;
  cout << " minimum = " << min << "  maximum = " << max
       << "  average = " << total / (n+1) << endl;
}
```

We have used the `>>` operator, along with the standard input stream `cin`, to ask the user for a number, and have printed a prompt and the statistics in a more informative form than simply a list of three numbers.

We can do essentially the same thing with a `for` loop, which is specially designed to make iterating over a sequence of numbers (or other objects) convenient.

```
#include <iostream.h>

main()
{
  double min = 1;
  double max = -1;
  double total = 0;
  int n;

  cout << "Enter a positive integer: ";
  cin >> n;

  for (int counter = n; counter >= 0; counter--)
    {
```

```
      double this_sin = sin(counter);
      if (this_sin < min)
        min = this_sin;
      if (this_sin > max)
        max = this_sin;
      total = total + this_sin;
    }
  cout << "Statistics for sin(0) .. sin(" << n << "):" << endl;
  cout << " minimum = " << min << "  maximum = " << max
       << "  average = " << total / (n+1) << endl;
}
```

3.1.5 JAM2000 solution

Here is a translation of the C program that uses `while` into JAM2000 machine code. Rather than translating the mnemonics into numbers yourself, you can depend on the JAM2000 simulator to do so.

```
;; Function statistics program
;; Assumptions:
;;  the program starts at location 1
;;  memory location 0 contains a memory address x at which n is stored.
;;            the following n+1 addresses (i.e., x+1..x+n+2) contain f(0),
;;            f(1), ... f(n), which are in the range -1,000,000..1,000,000

;; R0: n
;; R1: min
;; R2: max
;; R3: total
;; R4: counter
;; R5: location of f array in memory
;; R6: this_val
;; R7: the number 1
;; R8: not used
;; R9: scratch, temporary work


;; The program:

[00001]:  00000099 (ldi 9 0)    ; R9 = 0
[00002]:  00109560 (ld 5 9)     ; floc = mem[R9] = mem[0]
                ; The next two lines implement min = 1000000, but we are
                ; only able to specify immediate numbers in -99,999..99,999.
                ; Another way to do this would have been to store the values
                ; 1,000,000 and -1,000,000 in memory somewhere and load them in.
[00003]:  00105060 (ld 0 5)     ; n = mem[floc] = mem[mem[0]]
[00004]:  00100019 (ldi 1 1000) ; min = 1000
[00005]:  00011130 (mul 1 1 1)  ; min = min * min = 1000 * 1000 = 1000000
[00006]: -00000129 (ldi 2 -1)   ; max = -1
[00007]:  00012230 (mul 2 2 1)  ; max = max * min = -1 * 1000000 = -1000000
[00008]:  00000039 (ldi 3 0)    ; total = 0
[00009]:  00004090 (mov 0 4)    ; counter = n
[00010]:  00000179 (ldi 7 1)    ; R7 = 1
[00011]:  00075510 (add 5 5 7)  ; floc = floc + 1 = mem[0]+1
```

```
;; LOOP
[00012]:  00002345 (blz 4 23)   ; if (counter < 0) goto END
[00013]:  00045660 (ldx 6 5 4)  ; this_val = f(counter) = floc[counter]
[00014]:  00061920 (sub 9 1 6)  ; R9 = min - this_val
[00015]:  00001796 (blez 9 17)  ; if (R9 <= 0) goto NOTLESS
                                ; that is, if (min <= this_val) goto NOTLESS
[00016]:  00001690 (mov 6 1)    ; min = this_val
;; NOTLESS
[00017]:  00026920 (sub 9 6 2)  ; R9 = this_val - max
[00018]:  00002096 (blez 9 20)  ; if (R9 <= 0) goto NOTGREATER
                                ; that is, if (this_val <= max) goto NOTGREATER
[00019]:  00002690 (mov 6 2)    ; max = this_val
;; NOTGREATER
[00020]:  00063310 (add 3 3 6)  ; total = total + this_val
[00021]:  00074420 (sub 4 4 7)  ; counter = counter - 1
[00022]:  00001201 (jmpi 12)    ; goto LOOP
;; END
[00023]:  00001100 (print 1)    ; print min
[00024]:  00000200 (newline)
[00025]:  00002100 (print 2)    ; print max
[00026]:  00000200 (newline)
[00027]:  00070910 (add 9 0 7)  ; R9 = n + 1
[00028]:  00093940 (div 9 3 9)  ; R9 = total / R9 = total / (n+1)
[00029]:  00009100 (print 9)    ; print R9 = total / (n+1)
[00030]:  00000200 (newline)
[00031]:  00000000 (halt)

;; The data:

[00000]:  00000099              ; where to find n (and the array of n+1 values)

[00099]:  00000010              ; n
[00100]:  00000000              ; f(0) = 100000 * sin(0)
[00101]:  00841471              ; f(1) = 100000 * sin(1)
[00102]:  00909297              ; f(2) = 100000 * sin(2)
[00103]:  00141120
[00104]: -00756802
[00105]: -00958924
[00106]: -00279415
[00107]:  00656987
[00108]:  00989358
[00109]:  00412118
[00110]: -00544021              ; f(10) = 100000 * sin(3)
```

When writing a JAM2000 program, you are **strongly** encouraged to write it out on paper, then step through execution by hand, before typing it into the computer. Debugging machine programs on the fly can be difficult and frustrating, but if you take care, they can run on the first try.

When writing your program, decide (and document) what values will be stored in each register, and also comment each line of code. If you don't do this, you are unlikely to be able to understand your code later, nor will others be able to comprehend it.

3.2 Adaptive Quadrature

We will now translate a Scheme program for numerical integration program into C. The program uses the adaptive quadrature algorithm.

Given a non-negative function f defined on the closed interval $[a, b]$ of real numbers, the integral for f from a to b

$$\int_a^b f(x)\,dx$$

is simply the area enclosed between the curve formed by $f(x)$, the x-axis, and vertical lines passing through a and b. The area is often called the "area under the curve f between a and b."

Given a Scheme or C procedure `f` that computes f, we can compute an approximate value for the integral by

- subdividing $[a, b]$ into many small sub-intervals

$$[x_i, x_{i+1}], \quad i = 0, 1, \ldots, n-1$$

  where $x_0 = a$, $x_n = b$; and
- approximating the area of each subinterval $[x_i, x_{i+1}]$ by the area of the trapezoid with corners (vertices) $(x_i, 0)$, $(x_{i+1}, 0)$, $(x_{i+1}, f(x_{i+1}))$, $(x_i, f(x_i))$.

The approximate value for

$$\int_a^b f(x)\,dx$$

is the sum of the areas of the trapezoids for the intervals $[x_0, x_1], \ldots, [x_{n-1}, x_n]$.

To compute the area A of the trapezoid with corners $(x_i, 0)$, $(x_{i+1}, 0)$, $(x_{i+1}, f(x_{i+1}))$, $(x_i, f(x_i))$, we use the formula

$$A = (x_{i+1} - x_i) * \frac{f(x_{i+1}) + f(x_i)}{2},$$

which is the average height times the base.

The choice of sub-intervals has a significant effect on the accuracy of the approximation. The simplest choice is to subdivide $[a, b]$ into n equal sub-intervals. Unfortunately, this approach is wasteful for functions which have sharp peaks or valleys; obtaining an accurate approximation requires a large number of sub-intervals.

A better scheme is to "adaptively" subdivide $[a, b]$ into intervals until the difference between trapezoid area for the interval $[x_i, x_{i+1}]$ and the sum of the trapezoid areas for $[x_i, x_{mid}]$ and $[x_{mid}, x_{i+1}]$—where x_{mid} is the midpoint of $[x_i, x_{i+1}]$—is less than $(x_{i+1} - x_i) \cdot tol$. The constant *tol* is called the "unit error bound"; it is usually set to the value $error/(b-a)$ where *error* is the desired error bound on the final answer. Note that the bound is not guaranteed, since we are comparing the differences between successive approximations rather than the differences between approximations and exact answers.

3.2.1 Scheme Solution

Load the following program into Scheme and test it using *test*.

```
(define integrate
  ;; arguments: f: the function to be integrated
  ;; a: the left endpoint of the integration interval [a,b]
```

```
  ;; b: the right endpoint of the integration interval [a,b]
  ;; tol: the unit error bound for the computed result;
  ;; the approximate error in the answer is tol * (b - a)
  (lambda (f a b tolerance)
    (local
      [(define trapezoid
         (lambda (base left right)
           (* .5 base (+ left right))))
       (define integrate1
         (lambda (a b fa fb)
           (local
             [(define mid (* .5 (+ a b)))
              (define fmid (f mid))
              (define b-a (- b a))
              (define large-trap-area (trapezoid b-a fa fb))
              (define small-trap-area (+ (trapezoid (- mid a) fa fmid)
                                         (trapezoid (- b mid) fmid fb)))]
             (if (<= (abs (- large-trap-area small-trap-area)) (* tolerance b-a))
                 small-trap-area
                 (+ (integrate1 a mid fa fmid)
                    (integrate1 mid b fmid fb))))))]
      (integrate1 a b (f a) (f b)))))

(define pi 3.141592653589793)
(define e 2.718281828459045)

(define test
  (lambda ()
    (printf
      "~nExact answer is: 0~nComputed answer is:  ~s~n"
      (integrate cos 0 (* 2 pi) 1.e-8))
    (printf
      "~nExact answer is: 486~nComputed answer is:  ~s~n"
      (integrate sqrt 0 81 1.e-8))
    (printf
      "~nExact answer is near .5*sqrt(pi) = ~s ~nComputed answer is:  ~s~n"
      (* .5 (sqrt pi))
      (integrate (lambda (x) (expt e (- (* x x)))) 0 100 1.e-8))
    (printf
      "~nExact answer is near 1~nComputed answer is:  ~s~n"
      (integrate (lambda (x) (expt e (- x))) 0 10000 1.e-8))
    (printf
      "~nExact answer is pi/4 = ~s ~nComputed answer is:  ~s~n"
      (/ pi 4)
      (integrate (lambda (x) (sqrt (- 1 (* x x)))) 0 1 1.e-8))))
```

3.2.2 C solution

Write the equivalent C program; you may find the below template helpful. Since C does not support nested procedures, the help function `integrate1` cannot be nested inside `integrate` in your C program.

Like the Scheme function *integrate*, your C integration routine will divide the given interval $[a, b]$ in half creating two new sub-intervals $[a, mid]$ and $[mid, b]$. Then it will compute the area of the trapezoids determined by the original interval $[a, b]$ and the sum s of the areas of the trapezoids determined by the sub-intervals $[a, mid]$ and $[mid, b]$. If the two areas differ by no more than $tol * (b - a)$, then return the sum s as the answer (it's more accurate than the size of the large trapezoid). Otherwise, call the integration routine recursively to compute the area of curve f over the intervals $[a, mid]$ and $[mid, b]$ and return the sum.

Like the Scheme procedure `test` in `~comp210/Hw/10.ss`, the main routine determines approximate values for the following integrals:

1. $\int_0^{\pi} sin\, x\, dx$
2. $\int_0^{81} \sqrt{x}\, dx$
3. $\int_0^{100} e^{-x^2}\, dx$
4. $\int_0^{10000} e^{-x}\, dx$
5. $\int_0^{1} \sqrt{1 - x^2}\, dx$

```
#include <iostream.h>
#include <math.h>

double f1(double x)
{
  return exp(-(x*x));
}

double f2(double x)
{
  return exp(-x);
}

double f3(double x)
{
  return sqrt(1 - x*x);
}

// NEW AUXILIARY FUNCTION DEFINITIONS GO HERE

// Inputs:  f:  the function mapping double to double to be integrated
//          a:  the left end-point
//          b:  the right end-point
//          tol: the approximate relative error permitted in the result
// Result:  the approximate value of the integral of f over the interval [a,b]
//
// Method: adaptive integration is used to compute the area under the
// curve f over the interval [a,b].  The area over a given subinterval is
// computed using the trapezoidal rule.  The routine divides the
// interval [a,b] in half and recurs unless the difference between the
// area using the midpoint (a+b)/2 and the area without a midpoint (the
```

```
// trapezoid with base [a,b]) is less than equal to tol*(b-a).  Hence,
// tol serves as an estimate of the relative error in the answer.
double integrate(double f(double), double a, double b, double tol)
{
  // YOUR CODE GOES HERE
}

main()
{
  double pi = 3.141592653589793;

  cout.precision(16);           // set output precision

  cout << "Problem 1:" << endl << "The exact answer is 0" << endl;
  cout << "The computed answer is: " << integrate(sin, 0, 2*pi, 1e-8) <<
    endl << endl;

  cout << "Problem 2: " << endl << "The exact answer is 486" << endl;
  cout << "The computed answer is: " << integrate(sqrt, 0, 81, 1e-8) <<
    endl << endl;

  cout << "Problem 3: " << endl <<
    "The exact answer is near sqrt(pi)/2 = " << sqrt(pi)/2 << endl;
  cout << "The computed answer is:               " << integrate(f1, 0, 100, 1e-8) <<
    endl << endl;

  cout << "Problem 4: " << endl << "The exact answer is near 1" << endl;
  cout << "The computed answer is: " << integrate(f2, 0, 10000, 1e-8) <<
    endl << endl;

  cout << "Problem 5: " << endl << "The exact answer is pi/4 = " <<
    pi/4 << endl;
  cout << "The computed answer is:    " << integrate(f3, 0, 1, 1e-8) <<
    endl << endl;
}
```