Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 6.847 — Dataflow and Reduction Architectures

Handout # 1 September 13, 1989

Professor Arvind, NE43-210, x3-6090, arvind@abp
TA: Madhu Sharma, NE43-237, x3-4103, sharma@abp.lcs.mit.edu, sharma@athena
TA: Andrew Shaw, NE43-251, x3-8860, shaw@abp.lcs.mit.edu, ashaw@athena
Preferred Mailing Address for TAs: course@bk.lcs.mit.edu
Course Secretary: Susan Hardy, NE43-206, x3-0240, sh@abp.lcs.mit.edu

Time and Place: 26-204, MW9:30-11:00

Dataflow concepts offer a new approach to building highly parallel computer systems geared to sound concepts of program structure and language design. Our goal is to relate the problems in designing and programming multiprocessor systems and their purported solutions in the dataflow framework. Since dataflow languages are closely related to functional languages, we will discuss design and implementation issues of functional languages, starting with Lambda Calculus, Combinatory Calculus and Term Rewriting Systems. A framework for discussing the denotational and operational semantics of dataflow graphs will be presented. The course will concentrate on the current research on declarative languages and dataflow architectures at MIT. We will also discuss reduction techniques for implementing functional languages and explore the relationships between parallel reduction and dataflow. Program examples will be given in the context of the high-level language Id, developed at MIT by the Computation Structures Group. A more detailed list of topics is as follows:

1. **Hardware and Software Issues in Parallel Computing:** Processor and memory design to tolerate both large latency in memory accesses and to provide efficient synchronization of processors. Multiprocessor packet communication architectures. Inadequacy of imperative languages to express parallelism. Need for dynamic resource allocation to fully exploit parallelism.

2. **Id – A High Level Language for Dataflow Computation:** Functional constructs– higher-order functions, tuples, functional arrays, lists and algebraic types. Polymorphic typing and type inferencing. Limitations of functional data structures. Logical variables as incorporated in I-structures to model array operations.

3. **Dataflow Concepts:** Dataflow program graphs. Determinacy of computation. Least fixpoint semantics of abstract dataflow networks. The unraveling-interpreter. Static and dynamic architectures. Data structure storage. Higher-order functions and demand-driven evaluation in dataflow frame work. Compilation schemas for language control structures into dataflow program graphs and optimizations.

4. **Theoretical Issues:** Foundations of functional languages. Term Rewriting Systems. The Lambda Calculus and Combinatory forms. Typed and untyped Lambda Calculus. Applicative and normal order interpreters. Infinite data structures and higher-order functions. Abstraction of variables to generate supercombinatory code.

5. **Reduction Concepts:** Combinator reduction and weak-normal forms. Wadsworth graph reduction and *mfe* abstraction. Short circuiting or the G-machine approach to compiling super combinators. Architectural requirements for parallel graph reduction.

6. **Non-determinism:** The need of non-determinism in expressing resource management problems. Incorporation of non-deterministic constructs such as *merge* and *managers* in functional languages. Programming with mutable arrays.

Grading: The grades will be assigned solely on the basis of homeworks.

Please see the policy on collaboration on problem sets (Handout 2).

[9] David E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 1988. Also: CSG Memo 280, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

[10] David E. Culler and Gregory M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, January 1991 (to appear). (Also: CSG Memo 312, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[11] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, November 1980.

[12] Jack B. Dennis. First Version of a Data Flow Procedure Language. In *Proceedings of the Programming Symposium, Paris, Springer-Verlag LNCS 19*, 1974. (Revised: MAC TM61, May 1975, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[13] Kei Hiraki, Kenji Nishida, Satoshi Sekiguchi, Toshio Shimada, and Toshitsugu Yuba. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing*, 10(4):219–226, 1987.

[14] R. J. M. Hughes. Super-Combinators: A New Implementation Method for Applicative Languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10. Association for Computing Machinery, August 1982.

[15] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69. Association for Computing Machinery, June 1984.

[16] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Springer-Verlag LNCS 201: Proc. Functional Programming Languages and Computer Architecture, Nancy, France*, September 1985.

[17] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing 74: Proceedings of the IFIP Congress*, pages 471–475. International Federation for Information Processing, August 1974.

[18] J.W. Klop. Term Rewriting Systems. Technical report, Center for Mathematics and Computer Science, Amsterdam, The Netherlands, September 1985.

[19] Rishiyur S. Nikhil. Id World Reference Manual. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987. Revised August 1988 by P. R. Fenstermacher and J. E. Hicks.

[20] Rishiyur S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[21] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th. Annual International Symposium on Computer Architecture, Jerusalem, Israel*, May 29-31 1989. Also: CSG Memo 292, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

[22] Rishiyur S. Nikhil and Arvind. *Programming in* Id*: a parallel programming language.* Nom de Qwerty, Inc., 1989. (book, in preparation).

[23] Keshav Pingali. Fixpoint Equations and Dataflow. Computation Structures Group Memo 256, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, December 1985.

[24] Keshav Pingali and Vinod Kathail. An Introduction to the Lambda Calculus. Computation Structures Group Memo 258, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge MA, March 1986.

[25] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, and Toshitsugu Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel*, pages 46–53, May28-June 1 1989.

[26] K. R. Traub. Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, London, England*, September 1989. Also: CSG Memo 291, MIT Laboratory for Computer Science 545 Technology Square, Cambridge, MA 02139, USA.

[27] D. A. Turner. A New Implementation Techique for Applicative Languages. *Software - Practice and Experience*, 9:31–49, 1979.

[28] Yoshinori Yamaguchi, Shuichi Sakai, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An Architectural Design of a Highly Parallel Dataflow Machine. In *Proc. Information Processing 89, San Francisco, USA*, pages 1155–1160, August 28-September 1 1989.

Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 6.847 — Dataflow Architectures and Languages

Handout # 3 September 17, 1990

Policy Regarding Collaboration on Problem Sets

In recent years there has been increasing concern regarding the use of old solution sets and collaboration between students on the problem sets in various subjects. This concern has been voiced primarily by students who feel that they put forth substantial effort on the problem sets and yet receive below average scores because of the collaboration and use of old solution sets that takes place by a sizable portion of participants in the course. Collaboration and referring to old solution sets clearly puts students who work all the problems out on their own at a disadvantage. This handout explicitly states our policy regarding collaboration and the use of old solution sets.

1. We feel that discussion of the problem sets among course participants is a good way to learn the course material, as long as this interaction does not focus on deriving solutions. Such discussions should be solely for the purpose of clarifying your understanding of course material or the problem statement. The following ground rules should be kept in mind:

   - It is *never* acceptable to cooperate toward the solution of a problem. Examples of "cooperation" toward a problem solution include jointly working out a solution on a blackboard and then writing up problem sets separately, paraphrasing or copying the solution of a classmate, and discussion of possible solutions to a problem.
   - When you discuss the material covered in a problem set with another student, you should make explicit mention of that person's name at the beginning of your written solution.
   - If you feel that you are completely stuck on a problem, you should turn to the teaching assistants for help rather than relying upon assistance from other students (or old solution sets).

2. Referring to old solution sets is strictly forbidden. If you have inadvertantly been exposed to an old solution (visually or verbally), then you should explicitly state this fact at the beginning of your problem solution.

3. When problem sets are to be worked on cooperatively in groups, grading and collaboration policy on them will be specified on the handout.

Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 6.847 — Dataflow Architectures and Languages

Handout # 4 September 17, 1990

Information About Computer Facilities

Computer Facilities

All programming in 6.847 this year will be done in Id. Id is a high-level, expression-oriented functional language augmented with parallel data structures called *I-structures* and *M-structures.* Details about Id will be given in the lectures and can be found in the readings, notably in *Programming in Id: a parallel programming language* and the *Id Version 90 Reference Manual.*

There will be two compilers for Id: one which runs on the Lisp Machines, and one which runs under UNIX. On the Lisp Machines, the programming environment for Id is called "Id World"; instructions for using Id World and Lisp Machines are contained in the *Id World Reference Manual.* The Lisp Machines reside on the second floor of building NE43 (the Laboratory for Computer Science and the Artificial Intelligence Laboratory). Students who do not have lobby keys to the building should see a teaching assistant.

The UNIX Id compiler is experimental: it will produce faster code, but it will compile slower. Because it is an experimental compiler, we may experience some minor difficulties incorporating it into the course work. It is unlikely that we will be able to port the compiler to your own UNIX box. We will probably run the UNIX Id Compiler from one of the machines in the Computation Structures Group. Because this compiler and much of the course this year will be experimental, we ask that you be patient when the inevitable problems occur.

Since the resources of the Computation Structures Group will be taxed by the requirements of this course, we may require that all problem set files be kept on your Athena account. This will also aid us in grading. If you do not have an Athena account, you should apply for one by going to any Athena cluster and registering on-line. Details on how to use Athena are contained in *Athena Basics* and *Essential Athena.* Athena workstations are located in rooms 1-142, 2-225, 4-035, 4-167, 11-116, 16-034, 37-318, 66-080, and the fifth floor of the student center.

Electronic Mail

Electronic mail will be used extensively in the course as a means of communication between you and the TA's. If you receive your mail at Athena, you should learn how to use the Athena mail handler, `mh`, as the Athena mail system is likely to be quite different from any other system you have used. Information on how to use `mh` is contained in *Essential Messages.*

If you wish to contact one of the course teaching assistants, please use the course address, not his or her personal address.

Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 6.847 — Dataflow Architectures and Languages

Handout # 5 September 17, 1990

Problem Sets and Grading Policy

Extensions

Problem sets are due in class at 11AM on the day indicated on the problem set handout. Problem sets turned in after that time are considered late, and will not necessarily receive any credit.

Occasionally, you may find you need extra time to do a problem set. The teaching assistant can grant an extension of up to a week, if necessary. **To receive an extension, you must ask for one either in person or via computer mail before 11AM on the day the problem set is due.** We will be very strict about this requirement. Depending on circumstances, we may grant an extension of less than a week. Because of the need to distribute solution sets in a timely fashion, we usually cannot grant an extension of more than a week. For obvious reasons, no extensions will be granted for the last problem set of the term.

All requests for extensions or questions about problem sets should be directed to one of the teaching assistants, not to Prof. Arvind.

Grading

Your grade will be based entirely on the homework assignments. There are no exams in this course. All assignments will be weighted equally.

We will grade programs based on the correctness of your algorithm and on your style of programming. *All programming exercises will be in Id.* People who have taken 6.001 will have a slight advantage, since elements of programming style learnt in 6.001 will be very useful. Include comments to explain what each section of your code accomplishes. If we cannot understand your program, we cannot give you much credit. In the beginning of the course, we will be lenient about the style of your Id code, but we expect you to heed any comments we make about style.

How To Turn In Your Problem Set

1. Answers to problems should be submitted in the proper sequence; *i.e.*, answers to Problem 1 before answers to Problem 2.

2. If the problem specifically states what the name of the function should be and how it should take its arguments, follow these instructions exactly.

3. Include output to demonstrate that your functions work properly.

4. Formatting (like LaTeXor Scribe) is not necessary, and will not affect your grade in any way. However, if your handwriting is bad, we would appreciate a typed problem set: raw text is fine. Be reasonable.

Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science 6.847 — Dataflow Architectures and Languages

Handout # 6 September 17, 1990

Problem Set 1 – Due September 24, 1990

This problem set is intended to get you familiar with Id and Id World. We'll concern ourselves here only with the functional subset of Id, *i.e.*, Id without I-structures. The solutions are not long (no solution should be more than 20 lines of code), but those of you unfamiliar with the functional programming style may find it a bit tricky. We strongly suggest that you start on the problem set as soon as possible since there are only a relatively small number of Lisp machines available for classroom use.

Problem 1 **15 Points**

Part a:

The composite function of two given functions f and g is defined as follows—

$$(f \circ g)x = f(gx)$$

Complete the following definition of **compose** as written in Id.

```
def compose f g = ···
```

Part b:

Write a function **repeat** that given two arguments n and f, returns a function that composes f over its single argument n times. *E.g.*,

`(repeat 3 f) x` $\Longrightarrow$ `f(f(f(x)))`

Your function should utilize **compose**.

Part c:

The derivative of a function f is given by the formula:

$$f\prime(x) = \frac{f(x + dx) - f(x)}{dx}$$

Write an Id function **(deriv f dx)** which returns the derivative of f. Note that **deriv** should return a function.

Part d: Write an Id function **(nth_deriv f n dx)** which calculates the nth derivative of f. Your function should use **repeat** and **deriv**. How do you get around the fact that the function passed to **repeat** expects only one argument, while **deriv** expects two arguments?

In order to test **nth_deriv**, write a function **cube** which cubes a number. What do you get for the first, second, and third derivatives of **cube** for $x = 5$ and $dx = .1$? Try using different values for dx. Does reducing the value of dx passed to **nth_deriv** increase the accuracy? Explain.

Problem 2

20 Points

In this problem, we will study two different methods to find the root of a univariate continuous function f, *i.e.*, find an x such that,

$$f(x) = 0.$$

Section 1: The Newton-Raphson method

Part a:

The Newton-Raphson Method for finding the root of a function $f(x)$ is shown below. An initial guess is repeatedly iterated using the following equation until it is good enough.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

where $f'(x)$ is the derivative of $f(x)$.

Write a function `improve_guess` that given two arguments, a function f and its derivative f', returns a function that does one Newton-Raphson iteration. *E.g.*,

```
def f x = x*x - 2;
def ff x = 2*x;
step = improve_guess f ff;
...
> step 1.414 ⟹ 1.4142135
> repeat 2 step 1 ⟹ 1.4166666
> repeat 4 step 1 ⟹ 1.4142135
```

Use such iterators for various functions and their derivatives along with the `repeat` function in problem 1 to run a given number of iterations over an initial guess (as shown).

Part b:

Write a function `test_gen` which accepts a function f as an argument. It should return a function that given a guess x_k, returns true if $f(x_k)$ is within 6 decimal places of 0 and returns false otherwise. *E.g.*,

```
good_guess? = test_gen f;
...
> good_guess? 1.414 ⟹ false
> good_guess? 1.4142135 ⟹ true
```

Part c:

Using the functions defined in parts a and b above, write an Id procedure `find_root` to find the roots of a function using Newton-Raphson Method. `find_root` should take the function and its derivative as arguments and return a function that finds the closest root to a given initial guess as shown below.

```
def find_root f df = ...
sqrt_2 = find_root f ff;
...
> sqrt_2 1 ⟹ 1.4142135
> sqrt_2 -1 ⟹ -1.4142135
```

Note – You should test your program on other multi-root functions also.

Part 2: The half-interval method In the half-interval method, we are given a function f, a tolerance level t, and two points a and b such that

$$f(a) < 0 < f(b).$$

Since f is continuous, it must have at least one zero between a and b. Let x be the average of a and b. If

$$f(x) > 0$$

then f must have a zero between a and x. Similarly, if

$$f(x) < 0$$

then f must have a zero between x and b. We can use this procedure to cut the interval we are searching in half each time. When the size of the interval is less than t, we are done. The number of steps is clearly:

$$O(log(\frac{|a-b|}{t})).$$

Write a function `(half_interval_method f a b t)` which calculates a zero of f where a, b, and t are as defined above. You should use the following functions in your solution:

- `(close_enough x y)`: Returns true iff the length of the interval defined by x and y is less than t.
- `(search f neg_point pos_point)`: If the interval is small enough, `search` should return its midpoint. Otherwise, `search` should call itself recursively on the proper half of the interval.

Use `half_interval_method` to approximate π as a root of the sine (known as `sin` in Id) function between 2 and 4. You must first load the transcendental library in order use trigonometric functions in Gita. Mouse the `load` icon and click the middle button. When the menu pops up, mouse the file:

```
LIVE-OAK:>Id-world>csg>LIBRARY-90>TRANSCENDENTAL-LIBRARY.TTDAB
```

Problem 3 **30 Points**

In this problem we will study some Integration methods. The problem is to numerically integrate a given function $f(x)$ over an interval $[a, b]$.

Part a:

The Trapezoidal Rule for integration is as follows—

$$I = \int_a^b f(x)\,dx = h[f(a) + f(a + 2h)]$$

where,

$$h = \frac{b - a}{2}$$

Write a function `trap_int` with arguments f, h and a that computes the quadrature using the trapezoidal rule above.

Part b:

Simpson's Rule is more accurate than the trapezoidal rule and is computed as follows—

$$I = \frac{h}{3}[f(a) + 4f(a + h) + f(a + 2h)]$$

where h is as before.

Write a function `simp_int` with arguments f, h and a that computes the quadrature using Simpson's rule.

Part c:

For better accuracy, the given interval of integration $[a, b]$ is divided into several smaller intervals. The rule is applied to each of the smaller intervals and the results added to give the total area. There are be several ways to partition the interval.

If the partitions are all of the same size p, where $p = 2h$, then we get what is known as the **Composite rule** strategy.

Write a function `comp_quad` with arguments f, (a, b), a *quadrature_rule* and n such that $h = (b - a)/(2n)$ (n is the number of partitions). The function should compute the integral of f over (a, b) by adding the areas obtained by using the *quadrature_rule* over each of the partitions.

Use this function to integrate simple functions (*e.g.*, the function f in the previous problem) using different quadrature rules and partition sizes. Compare the accuracy of the Trapezoidal rule versus the Simpson's rule.

Part d:

If the partitions in the above computations are not all equal and can be changed as required, then we obtain the **Adaptive Quadrature** strategy.

A simple recursive adaptive quadrature method is outlined below—

1. First, the integral is calculated over the entire given interval $[a, b]$ using the given quadrature rule. This is the *old_guess*.

2. The midpoint of the interval is found: $x = (b - a)/2$.

3. A *new_guess* is computed by applying the rule to $[a, x]$ and $[x, b]$ and adding the result.

4. The *new_guess* is tested against *old_guess*. If they agree within limits then *new_guess* is returned. Otherwise, the procedure is recursively called over the two partitions and the sum is returned.

Write a function **adapt_quad** with arguments f, (a, b) and a *quadrature_rule* that implements the above method. Use this with the rules given above and compare the results with the Composite strategy.

Part e:

Integrals can also be calculated using the following formula:

$$I = \int_a^b f(x)\,dx = (f(a + \frac{dx}{2}) + f(a + \frac{3dx}{2}) + f(a + \frac{5dx}{2}) + \ldots) * dx$$

An Id function **integrate** which uses this formula is given in chapter two of the notes. Implement **integrate** in Id and test it out for various functions.

In order to get more accurate integrals, smaller values of dx need to be specified. Our goal is to write a function which will call **integrate** using progressively smaller values of dx until the calculated integral converges to an acceptably accurate estimate.

Given an initial dx and a quantity *epsilon*, we can do the following:

1. Calculate the integral `(integrate dx a b f)`.

2. Calculate the integral `(integrate dx/2 a b f)`.

3. If the two integrals differ by less than *epsilon*, return the interval calculated at step 2. Otherwise, repeat the process with $dx/2$ substituted for dx.

Write a function `(call_integrate dx a b f epsilon)` which implements this algorithm.

Part f: Test the different integration methods on several functions. Which strategy works best? How do the instruction counts compare?

Problem 4

35 Points

In this problem we will develop a higher order representation for vectors and matrices and write functions to manipulate them.

Recall that Matrix multiplication is defined as follows. Given a matrix A of size $n \times m$ and another matrix B of size $m \times l$, it returns a new matrix C of size $n \times l$, whose elements are defined by the following equation—

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$$

In other words, an element c_{ij} of the matrix C is the vector inner-product of the i-th row of matrix A with the j-th column of matrix B.

Section 1 : Using higher-order notation

A straightforward translation into Id code gives the following—

```
def mat_mult A B = { (la1,ua1),_ = 2d_bounds A;
                     _,(lb2,ub2) = 2d_bounds B;
                     def select (i,j) = vip (row i A) (column j B);
                    in
                     make_matrix ((la1,ua1),(lb2,ub2)) select };
def row i X = { _,(lx2,ux2) = 2d_bounds X;
                def vsel j = X[i,j];
               in
                make_array (lx2,ux2) vsel };
def column j X = { (lx1,ux1),_ = 2d_bounds X;
                   def vsel i = X[i,j];
                  in
                   make_array (lx1,ux1) vsel };
```

Part a:

Write the function `vip` that would complete the above code.

Section 2 : Efficient matrix multiplication

The problem with the above program is that in order to access a given row or a column of a matrix as a 1-dimensional array, we have to copy it into a vector. Clearly, it is possible to write a program that does not do any copying, but then it would not reflect the computation mechanism so clearly as the above procedure does.

Part b:

Write an efficient version of matrix multiply, `mat_mult2`, that does not do any copying of rows and columns into vectors.

Section 3 : Generalised Vectors

The question is—Can we use the clean, higher-order notation and yet not lose on efficiency? The answer lies in the way we look at a vector.

When we think of a *vector*, or 1-dimensional array, in a language, what we have is a data structure that has a lower bound and upper bound, and a selection function which takes an integer i within

the bounds, and returns a value corresponding to that integer. We write this selection function as `A[i]` for a vector `A`. For mathematical purposes, we don't even require the lower and the upper bound explicitly, instead, we only need to look at the size of the vector, *i.e.*, the number of elements in the vector. The indices for the selection function are assumed to start from 1 in that case.

We now define an abstraction called a *generalized vector*. A generalized vector would have a number representing the size of the vector and a selection function. The generalization is that we will allow the function to be *any* arbitrary function defined over the range, not just the primitive function which selects from 1-dimensional arrays. The selection operation is to be viewed as selection-by-position and not by index value. So an ordinary vector with a given lower and upper bound would appear to have been normalised to lower bound 1 in the generalised vector form.

Note that the choice of keeping just the size of the generalized vector and not its actual bounds is purely a matter of convenience. Most algebraic manipulations do not concern themselves with the actual lower and upper bounds and use the relative positions of the elements alone, and hence the choice.

We will represent a generalized vector as a 2-tuple containing the size and the selection function. A few examples are shown below.

```
A = make_array (1,n) fa;
genA = n,fa;
B = make_matrix ((1,n),(1,m)) fb;
genB = (n,m),fb;
...
Asize,Asel = genA;
> Asel 2 ⟹ A[2]
...
Bsize,Bsel = genB;
> Bsel (3,4) ⟹ B[3,4]
```

As shown in the examples above, it is straightforward to extend this notion of generalised vectors to matrices and arrays of higher dimensions. The representation would be similar—a 2-tuple whose first element would now itself be a d-tuple representing the size in each of the d dimensions and a selector function over d-tuples as indices.

Part c:

Write a function **gen_vec** which creates a generalised vector out of a given vector. Take care that its bounds are properly translated to start from 1. Similarly, write a function **gen_mat** that converts a matrix into a generalized matrix.

Note – The generalised vector does not create a copy of the given vector. It only logically reorganises the data storage already available by changing its accessing function.

Part d:

Write functions **vcopy** and **matcopy** that copy a given generalised vector (matrix) into an ordinary vector (matrix) as shown below.

```
def vcopy genX (l,u) = ...;
def matcopy genX ((l1,u1),(l2,u2)) = ...;
```

This is to enable you to see the contents of a generalized vector (matrix) since GITA only knows how to print ordinary vectors (matrices).

Note – Assume that the given bounds are compatible with the size of the generalised vector (matrix).

Section 3a : Manipulating Generalised Vectors

Now we would write a library of functions to manipulate generalized vectors and matrices to solve our original problem—to be able to do matrix multiplication using higher-order functions and still retain the efficiency of direct methods.

Note – In the following parts you may assume that the vectors you operate upon have the correct sizes, so there is no need of bounds checking. Also note that the code shown below is merely suggestive of the naming, arity and the type of the function to be written. You are free to incorporate any pattern-matching.

Part e:

Write the functions `row_vector` and `column_vector` which return a generalized vector that accesses a given row (column) of a generalized matrix as shown below.

```
defsubst row_vector genA i = ...;
defsubst column_vector genA j = ...;
```

Note – Make sure the generalized vector you return has the correct size! You are advised to test both `row_vector` and `column_vector` on non-square matrices.

Part f:

Write functions `vec_add` and `vec_sub` that return a generalised vector that represents the element by element sum (difference) of the two given generalised vectors.

```
defsubst vec_add genA genB = ...;
defsubst vec_sub genA genB = ...;
```

Part g:

Now write an inner-product function `gen_vip` for generalized vectors.

```
defsubst gen_vip genA genB = ...;
```

Part h:

Write a function `gen_mat_vec_mult` that multiplies a generalized matrix with a generalised vector and produces another generalised vector.

```
defsubst gen_mat_vec_mult genA genX = ...;
```

Part i:

Write a function `gen_mat_mult` that multiplies two generalized matrices together and produces a third.

```
defsubst gen_mat_mult genA genB = ...;
```

Note – Make sure that it does not copy rows and columns out of the given matrices and it works for non-square matrices as well.

Section 4 : Memoization in generalized vectors

The concept of generalised vectors does not necessitate the use of data storage as it is clear from the above discussion. But, still we would want to memoize the element values into ordinary vectors and matrices, simply because repeating computation each time we select an element is too inefficient.

The point to notice is that if there is already a data storage embedded in the selector function, then it is not necessary to create copies of it. This is where the generalized vectors win over our previous example involving ordinary vectors. They allow the flexibility of functions while still retaining the efficiency of data storage embedded within their selector functions.

The point can be illustrated with the following example—

```
def vmult1 (n,Asel) (_,Bsel) = { defsubst Vsel i = Asel i * Bsel i;
                                 in
                                  n,Vsel };
def vmult2 (n,Asel) (_,Bsel) = { defsubst Vsel i = Asel i * Bsel i;
                                  V = make_array (1,n) Vsel;
                                 in
                                  gen_vec V };
c = gen_vec (make_array (lc,uc) fc);
c_sqr1 = vmult1 c c;
c_sqr2 = vmult2 c c;
sum_of_4pow1 = gen_vip c_sqr1 c_sqr1;
sum_of_4pow2 = gen_vip c_sqr2 c_sqr2;
```

It should be clear that `sum_of_4pow1` would execute the inner multiplication for elements of `c_sqr1` each time they are used. While the same information is memoized into `c_sqr2` and hence is computed only once and used as many times as desired.

It may be argued that this memoization may be costlier if, for example, the inner computation was cheap enough, for instance in `vec_add`. Indeed, this flexibility of space-time trade-off is the primary property that make the concept of generalized vectors interesting.

Part j:

Write functions `1D_memoize` and `2D_memoize` that simply transform a given generalized vector (matrix) into the corresponding memoized generalized vector (matrix).

```
defsubst 1D_memoize genX = ...;
defsubst 2D_memoize genA = ...;
```

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.847 — Dataflow Architectures and Languages

Handout # 6s Sept 24, 1990

Problem Set 1 – Solutions

Problem 1 **15 Points**

Part a:

The composite function of two given functions f and g is defined as follows—

$$(f \circ g)x = f(gx)$$

Complete the following definition of **compose** as written in Id.

```
def compose f g = ···
```

```
defsubst compose f g = {fun x = f (g x)};
```

Part b:

Write a function **repeat** that given two arguments n and f, returns a function that composes f over its single argument n times. *E.g.*,

```
(repeat 3 f) x ⟹ f(f(f(x)))
```

Your function should utilise **compose**.

```
defsubst repeat 0 f = {fun x = x}
      |..repeat n f = compose f (repeat (n-1) f);
```

Part c:

The derivative of a function f is given by the formula:

$$f'(x) = \frac{f(x+dx) - f(x)}{dx}$$

Write an Id function (**deriv f dx**) which returns the derivative of f. Note that **deriv** should return a function.

```
def deriv f dx =
  {fun x = ((f (x + dx)) - (f x))/ dx};
```

Part d: Write an Id function (`nth_deriv f n dx`) which calculates the nth derivative of f. Your function should use `repeat` and `deriv`. How do you get around the fact that the function passed to `repeat` expects only one argument, while `deriv` expects two arguments?

In order to test `nth_deriv`, write a function `cube` which cubes a number. What do you get for the first, second, and third derivatives of `cube` for $x = 5$ and $dx = .1$? Try using different values for dx. Does reducing the value of dx passed to `nth_deriv` increase the accuracy? Explain.

```
def nth_deriv f n dx =
  ((repeat n {fun f = deriv f dx}) f);
```

Problem 2 **20 Points**

In this problem, we will study two different methods to find the root of a univariate continuous function f, *i.e.*, find an x such that,

$$f(x) = 0.$$

Section 1: The Newton-Raphson method

Part a:

The Newton-Raphson Method for finding the root of a function $f(x)$ is shown below. An initial guess is repeatedly iterated using the following equation until it is good enough.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

where $f'(x)$ is the derivative of $f(x)$.

Write a function `improve_guess` that given two arguments, a function f and its derivative f', returns a function that does one Newton-Raphson iteration. *E.g.*,

```
def f x = x*x - 2;
def ff x = 2*x;
step = improve_guess f ff;
...
> step 1.414 ⟹ 1.4142135
> repeat 2 step 1 ⟹ 1.4166666
> repeat 4 step 1 ⟹ 1.4142135
```

Use such iterators for various functions and their derivatives along with the `repeat` function in problem 1 to run a given number of iterations over an initial guess (as shown).

```
defsubst improve_guess f df =
  { defsubst iteration x = x - (f x)/(df x);
   in
     iteration};
```

Part b:

Write a function `test_gen` which accepts a function f as an argument. It should return a function that given a guess x_k, returns true if $f(x_k)$ is within 6 decimal places of 0 and returns false otherwise. *E.g.*,

```
good_guess? = test_gen f;
...
> good_guess? 1.414 ⟹ false
> good_guess? 1.4142135 ⟹ true
```

```
defsubst test_gen f =
  { defsubst test x = abs (f x) < 1e-6;
   in
     test};
```

Part c:

Using the functions defined in parts a and b above, write an Id procedure `find_root` to find the roots of a function using Newton-Raphson Method. `find_root` should take the function and its derivative as arguments and return a function that finds the closest root to a given initial guess as shown below.

```
def find_root f df = ···
sqrt_2 = find_root f ff;
...
> sqrt_2 1 ⟹ 1.4142135
> sqrt_2 -1 ⟹ -1.4142135
```

Note – You should test your program on other multi-root functions also.

```
defsubst find_root f df =
  { iterator = improve_guess f df;
    tester = test_gen f;
    defsubst root x = if (tester x) then x
      else root (iterator x);
    in
      root};
```

Part 2: The half-interval method In the half-interval method, we are given a function f, a tolerance level t, and two points a and b such that

$$f(a) < 0 < f(b).$$

Since f is continuous, it must have at least one zero between a and b. Let x be the average of a and b. If

$$f(x) > 0$$

then f must have a zero between a and x. Similarly, if

$$f(x) < 0$$

then f must have a zero between x and b. We can use this procedure to cut the interval we are searching in half each time. When the size of the interval is less than t, we are done. The number of steps is clearly:

$$O(log(\frac{|a-b|}{t})).$$

Write a function (`half_interval_method f a b t`) which calculates a zero of f where a, b, and t are as defined above. You should use the following functions in your solution:

- (`close_enough x y`): Returns true iff the length of the interval defined by x and y is less than t.
- (`search f neg_point pos_point`): If the interval is small enough, `search` should return its midpoint. Otherwise, `search` should call itself recursively on the proper half of the interval.

Use `half_interval_method` to approximate π as a root of the sine (known as `sin` in Id) function between 2 and 4. You must first load the transcendental library in order use trigonometric functions in Gita. Mouse the `load` icon and click the middle button. When the menu pops up, mouse the file:

`LIVE-OAK:>Id-world>csg>LIBRARY-90>TRANSCENDENTAL-LIBRARY.TTDAB`

```
def half_interval_method f a b t =
  {def close_enough x y =
     (abs (x - y)) < t;
   def search f neg_point pos_point =
     {midpoint = (neg_point + pos_point)/2.0
      in
if close_enough neg_point pos_point then
  midpoint
else
  {test_value = f midpoint
   in
     if test_value > 0 then
       search f neg_point midpoint
     else
       if test_value < 0 then
 search f midpoint pos_point
       else midpoint}};
   a_value = f a;
```

```
  b_value = f b
  in
    if (a_value < 0) and (b_value > 0) then
      search f a b
    else
      if (a_value > 0) and (b_value < 0) then
search f b a
      else
9999.0  % an error value
 };
```

---

30 Points

Problem 3

In this problem we will study some Integration methods. The problem is to numerically integrate a given function $f(x)$ over an interval $[a, b]$.

Part a:

The Trapesoidal Rule for integration is as follows—

$$I = \int_a^b f(x)\,dx = h[f(a) + f(a+2h)]$$

where,

$$h = \frac{b-a}{2}$$

Write a function **trap_int** with arguments f, h and a that computes the quadrature using the trapesoidal rule above.

```
defsubst trap_int f h a = h*(f a + f (a+2*h));
```

Part b:

Simpson's Rule is more accurate than the trapesoidal rule and is computed as follows—

$$I = \frac{h}{3}[f(a) + 4f(a+h) + f(a+2h)]$$

where h is as before.

Write a function **simp_int** with arguments f, h and a that computes the quadrature using Simpson's rule.

```
defsubst simp_int f h a = h*(f a + 4.0 * f (a+h) + f (a+2.0*h))/3.0;
```

Part c:

For better accuracy, the given interval of integration $[a, b]$ is divided into several smaller intervals. The rule is applied to each of the smaller intervals and the results added to give the total area. There are be several ways to partition the interval.

If the partitions are all of the same size p, where $p = 2h$, then we get what is known as the **Composite rule strategy.**

Write a function comp_quad with arguments f, (a, b), a *quadrature_rule* and n such that $h = (b - a)/(2n)$ (n is the number of partitions). The function should compute the integral of f over (a, b) by adding the areas obtained by using the *quadrature_rule* over each of the partitions.

Use this function to integrate simple functions (*e.g.*, the function f in the previous problem) using different quadrature rules and partition sizes. Compare the accuracy of the Trapezoidal rule versus the Simpson's rule.

```
defsubst comp_quad f (a,b) r n =
  { h = (b-a)/(2.0*n);
    defsubst sum s x = if abs (x-b)<1e-6 then s
 else sum (s + r f h x) (x+2.0*h);
    in
      sum 0.0 a};
```

Part d:

If the partitions in the above computations are not all equal and can be changed as required, then we obtain the **Adaptive Quadrature strategy.**

A simple recursive adaptive quadrature method is outlined below—

1. First, the integral is calculated over the entire given interval $[a, b]$ using the given quadrature rule. This is the *old_guess*.
2. The midpoint of the interval is found: $x = (b - a)/2$.
3. A *new_guess* is computed by applying the rule to $[a, x]$ and $[x, b]$ and adding the result.
4. The *new_guess* is tested against *old_guess*. If they agree within limits then *new_guess* is returned. Otherwise, the procedure is recursively called over the two partitions and the sum is returned.

Write a function adapt_quad with arguments f, (a, b) and a *quadrature_rule* that implements the above method. Use this with the rules given above and compare the results with the Composite strategy.

```
defsubst adapt_quad f (a,b) r =
  { h = (b-a)/2.0;
    old_guess = r f h a;
    x = a+h;
    new_guess = r f (h/2.0) a + r f (h/2.0) x;
    in
      if abs (new_guess-old_guess) < 1e-2 then new_guess
      else adapt_quad f (a,x) r + adapt_quad f (x,b) r};
```

6.847 – Problem Set 1 – Solutions

Part e:

Integrals can also be calculated using the following formula:

$$I = \int_a^b f(x)\,dx = (f(a + \frac{dx}{2}) + f(a + \frac{3dx}{2}) + f(a + \frac{5dx}{2}) + \ldots) * dx$$

An Id function `integrate` which uses this formula is given in chapter two of the notes. Implement `integrate` in Id and test it out for various functions.

In order to get more accurate integrals, smaller values of dx need to be specified. Our goal is to write a function which will call `integrate` using progressively smaller values of dx until the calculated integral converges to an acceptably accurate estimate.

Given an initial dx and a quantity *epsilon*, we can do the following:

1. Calculate the integral (`integrate dx a b f`).
2. Calculate the integral (`integrate dx/2 a b f`).
3. If the two integrals differ by less than *epsilon*, return the interval calculated at step 2. Otherwise, repeat the process with $dx/2$ substituted for dx.

Write a function (`call_integrate dx a b f epsilon`) which implements this algorithm.

```
def call_integrate dx a b f epsilon =
  {half_dx = dx/2.0;
   typeof integral_1 = F;
   integral_1 = integrate dx a b f;
   integral_2 = integrate half_dx a b f
   in
     if abs (integral_1 - integral_2) < epsilon then
       integral_2
     else
       call_integrate half_dx a b f epsilon};
```

Part f: Test the different integration methods on several functions. Which strategy works best? How do the instruction counts compare?

Problem 4 **35 Points**

In this problem we will develop a higher order representation for vectors and matrices and write functions to manipulate them.

Recall that Matrix multiplication is defined as follows. Given a matrix A of size $n \times m$ and another matrix B of size $m \times l$, it returns a new matrix C of size $n \times l$, whose elements are defined by the following equation—

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j}$$

In other words, an element c_{ij} of the matrix C is the vector inner-product of the i-th row of matrix A with the j-th column of matrix B.

Section 1 : Using higher-order notation

A straightforward translation into Id code gives the following—

```
def mat_mult A B = { (la1,ua1),_ = 2d_bounds A;
                     _,(lb2,ub2) = 2d_bounds B;
                     def select (i,j) = vip (row i A) (column j B);
                    in
                     make_matrix ((la1,ua1),(lb2,ub2)) select };
def row i X = { _,(lx2,ux2) = 2d_bounds X;
                def vsel j = X[i,j];
               in
                make_array (lx2,ux2) vsel };
def column j X = { (lx1,ux1),_ = 2d_bounds X;
                   def vsel i = X[i,j];
                  in
                   make_array (lx1,ux1) vsel };
```

Part a:

Write the function vip that would complete the above code.

```
defsubst vip A B =
  { la,ua = bounds A;
    typeof A = 1d_array F;
    typeof B = 1d_array F;
    defsubst sum s i = if i>ua then s
       else sum (s+A[i]*B[i]) (i+1);
    in
      sum 0.0 la};
```

Section 2 : Efficient matrix multiplication

The problem with the above program is that in order to access a given row or a column of a matrix as a 1-dimensional array, we have to copy it into a vector. Clearly, it is possible to write a program that does not do any copying, but then it would not reflect the computation mechanism so clearly as the above procedure does.

Part b:

Write an efficient version of matrix multiply, mat_mult2, that does not do any copying of rows and columns into vectors.

```
defsubst mat_mult2 A B =
  { (la1,ua1),(la2,ua2) = 2d_bounds A;
    _,(lb2,ub2) = 2d_bounds B;
    defsubst vip i j s k = if k>ua2 then s
   else vip i j (s+A[i,k]*B[k,j]) (k+1);
    defsubst select (i,j) = vip i j 0 la2;
    in
      make_matrix ((la1,ua1),(lb2,ub2)) select};
```

Section 3 : Generalised Vectors

The question is—Can we use the clean, higher-order notation and yet not lose on efficiency? The answer lies in the way we look at a vector.

When we think of a *vector*, or 1-dimensional array, in a language, what we have is a data structure that has a lower bound and upper bound, and a selection function which takes an integer i within the bounds, and returns a value corresponding to that integer. We write this selection function as `A[i]` for a vector `A`. For mathematical purposes, we don't even require the lower and the upper bound explicitly, instead, we only need to look at the size of the vector, *i.e.*, the number of elements in the vector. The indices for the selection function are assumed to start from 1 in that case.

We now define an abstraction called a *generalized vector*. A generalised vector would have a number representing the size of the vector and a selection function. The generalisation is that we will allow the function to be *any* arbitrary function defined over the range, not just the primitive function which selects from 1-dimensional arrays. The selection operation is to be viewed as selection-by-position and not by index value. So an ordinary vector with a given lower and upper bound would appear to have been normalised to lower bound 1 in the generalised vector form.

Note that the choice of keeping just the size of the generalised vector and not its actual bounds is purely a matter of convenience. Most algebraic manipulations do not concern themselves with the actual lower and upper bounds and use the relative positions of the elements alone, and hence the choice.

We will represent a generalised vector as a 2-tuple containing the size and the selection function. A few examples are shown below.

```
A = make_array (1,n) fa;
genA = n,fa;
B = make_matrix ((1,n),(1,m)) fb;
genB = (n,m),fb;
...
Asize,Asel = genA;
> Asel 2 ⟹ A[2]
...
Bsize,Bsel = genB;
> Bsel (3,4) ⟹ B[3,4]
```

As shown in the examples above, it is straightforward to extend this notion of generalised vectors to matrices and arrays of higher dimensions. The representation would be similar—a 2-tuple whose first element would now itself be a d-tuple representing the size in each of the d dimensions and a selector function over d-tuples as indices.

Part c:

Write a function `gen_vec` which creates a generalised vector out of a given vector. Take care that its bounds are properly translated to start from 1. Similarly, write a function `gen_mat` that converts a matrix into a generalised matrix.

Note – The generalised vector does not create a copy of the given vector. It only logically reorganises the data storage already available by changing its accessing function.

```
defsubst gen_vec X =
  { lx,ux = bounds X;
    typeof X = 1d_array F;
    defsubst sel i = X[lx+i-1];
    in
```

```
      (ux-lx+1),sel};

defsubst gen_mat A =
  { (l1,u1),(l2,u2) = 2d_bounds A;
    defsubst sel (i,j) = A[l1+i-1,l2+j-1];
    in
      (u1-l1+1,u2-l2+1),sel};
```

Part d:

Write functions vcopy and matcopy that copy a given generalised vector (matrix) into an ordinary vector (matrix) as shown below.

```
def vcopy genX (l,u) = ...;
def matcopy genX ((l1,u1),(l2,u2)) = ...;
```

This is to enable you to see the contents of a generalised vector (matrix) since GITA only knows how to print ordinary vectors (matrices).

Note – Assume that the given bounds are compatible with the size of the generalised vector (matrix).

```
defsubst vcopy (n,f) (l,u) =
  {array (l,u)
   | [l+i-1] = f i || i <- 1 to n};

defsubst matcopy ((n,m),f) ((l1,u1),(l2,u2)) =
  {matrix ((l1,u1),(l2,u2))
   | [l1+i-1,l2+j-1] = f (i,j) || i <- 1 to n & j <- 1 to m};
```

Section 3a : Manipulating Generalised Vectors

Now we would write a library of functions to manipulate generalised vectors and matrices to solve our original problem—to be able to do matrix multiplication using higher-order functions and still retain the efficiency of direct methods.

Note – In the following parts you may assume that the vectors you operate upon have the correct sizes, so there is no need of bounds checking. Also note that the code shown below is merely suggestive of the naming, arity and the type of the function to be written. You are free to incorporate any pattern-matching.

Part e:

Write the functions row_vector and column_vector which return a generalised vector that accesses a given row (column) of a generalized matrix as shown below.