Using Miranda as a first programming language

TIM LAMBERT
School of Computer Science and Engineering, University of New South Wales, P.O. Box 1, Kensington NSW 2033, Australia (*e-mail*: lambert@cs.unsw.oz.au)

PETER LINDSAY
Department of Computer Science, University of Queensland, Queensland 4072, Australia (*e-mail*: pal@cs.uq.oz.au)

KEN ROBINSON
School of Computer Science and Engineering, University of New South Wales, P.O. Box 1, Kensington NSW 2033, Australia (*e-mail*: kenr@cs.unsw.oz.au)

Abstract

The functional programming language Miranda has been used as a first programming language at the University of NSW since the beginning of 1989, when a new computer engineering course and a revised computer science course were introduced. This paper explains the reasons for choosing the language, and describes the subject in which Miranda is introduced. Examples of the presentation of the material, and of exercises and assignment used in the course, are given. Finally, an assessment of the experience is given.

1 Introduction

Towards the end of 1986, the Department of Computer Science commenced a revision of the programming content of the first year computing subjects. This was part of the development of a new four year course in computer engineering, and the consequent revision of the computer science course. At that time there was one computing subject in first year. A large part of that subject was devoted to the teaching of programming, using Pascal as the programming language. The subject was a fairly standard introductory subject in programming, and rather than being an introduction to programming it was more often thought of as being an introduction to *programming in Pascal.* This first subject was followed by a second year subject in which programming was taught more formally, introducing the concept of a formal specification, and design and implementation via a sequence of refinement steps.

We wanted the new subject to fulfil a number of needs that the old subject did not meet very satisfactorily, or even at all. As a first subject it obviously should provide a basis for subsequent computing subjects, but in particular, we wanted a sound basis

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

for subsequent programming. We wanted to introduce the notion of reasoning about programs and so a degree of formality was required. Thus the broad requirements of the new subject were:

- to identify and distinguish the notions of specification, design and implementation.
- to present a sound basis for programming as a systematic activity.
- to develop good problem solving methods such as functional decomposition, generalisation, and the appropriate use of data structures.
- to cover the elementary data types: numbers, booleans, characteristics etc. and the structures of sets, lists and trees.

This paper discusses our choice of programming language and the syllabus for the programming component of the new subject; presents two longer examples taken from that programming component; and contrasts our experiences with teaching using Pascal. We include three appendices that give details of assignments, tutorial exercises and lab exercises, respectively.

2 Choice of programming language

Many of the requirements are not well assisted by a programming language such as Pascal. Pascal is verbose, it is not modular, and it imposes too much language-specific implementation detail, especially when dealing with structures such as lists and trees. These criticisms are not specific to Pascal, but apply to most procedural programming languages. Other classes of programming language such as logic and functional were considered. A good, modern functional language was thought to be better suited to first year teaching, mainly due to the expectation of greater familiarity with functions compared with relations, and the greater simplicity of the evaluation model of functions. Functional programs are easier to reason about than either procedural or logic programs, because of the simple rewriting model for the execution of a functional program.

A number of properties of the chosen language Miranda[1] were considered to be very important:

- the use of an equational style of function definition is familiar, clear and concise.
- its non-strict (lazy) evaluation strategy aids the modular development of programs.
- the static type checking of functions assists the development of a set of functions. The type discipline ensures that functions can be composed in only a limited number of ways. This is in marked contrast to a procedural programming language where there is much greater freedom of composition of programs.
- polymorphism helps the modular development of programs.
- algebraic types provide a simple, clear and consistent route to the development of recursive structures such as trees.

In addition to those language features, a number of implementation details simplified the use of Miranda:

[1] Miranda is a trademark of Research Software Ltd.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

- functions can be evaluated interactively with direct input of any data type (with the exception of abstract data types), and conversely the automatic listing of any data type simplifies the testing of individual functions.
- functions do not need to contain special input or output functions, since files are mapped onto lists.
- definitions can be given in any order, thus there is no particular order imposed on the development of a program.
- the provision of a *literate programming* format allows the development of program scripts that are self-contained documents. Lines that start with > are program; everything else is comment. The TEX source for this document is also a Miranda script.

As a consequence of these features, Miranda is able to express, clearly and concisely, solution to a broad range of problems. The solutions can be developed with great generality and modularity. The literate script facility allows the development of programs to be presented as a story; indeed the TEX source of this paper is a Miranda script, and the examples presented in the later part of the paper display the style of presentation used in the subject. Many of these aspects of programming in Miranda are illustrated in the remainder of the paper.

3 The course

Full-time students at the University of NSW do four subjects each semester. There are two 13 week semesters in each year.

There are two first year computer science subjects. Computer science majors do Computing 1A in the first semester and Computing 1B in the second semester. Each of these subjects has two strands. The programming strand of Computing 1A uses Miranda to introduce students to programming. The rest of this paper describes the programming strand. The literacy strand of Computing 1A introduces students to computing systems and applications. Miranda is used in the literacy strand to present simple interactive models of computer systems (Piotrowski, 1989). Enrolment in Computing 1A amounts to roughly 450 students each session.

The programming strand of Computing 1B continues the introduction to programming. Students learn how to program procedurally using Modula-2, and how to use abstract data types in Modula-2 and Miranda. The hardware strand of Computing 1B introduces students to computer organisation and assembly language programming.

Miranda is also used in later second, third and fourth year subjects.

4 Programming strand outline

4.1 Textbook

The textbook used is Bird and Wadler's (1988) *Introduction to Functional Programming*, supplemented by lecture notes (Lambert, 1990). Copies of the lecture notes can be obtained from the University of NSW through any of the authors.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

4.2 Lectures

Two (one hour) lectures a week are used for teaching programming. References below are to sections in the text book.

Basics (Two lectures) expressions, values, notion of type, functions; function definitions [1.1–1.4].

Types (Two lectures) the basic types: numbers, booleans, characters, strings, tuples, patterns, functional composition, type synonyms [2.1–2.7].

Lists (Three lectures) lists, list comprehensions, list operations: take, drop, zip2, list indexing, map, filter, foldr [3.1–3.5.1].

Example Program (One lecture) print a supermarket receipt, given a list of bar codes and a database describing each code.

Logic (Three lectures) propositions and operators, truth tables, logical identities, simplification, predicates, quantifiers.

Recursion (Two lectures) list patterns [3.6]; recursion and induction [5.1–5.3].

Example Program (One lecture) find misspelled words, sorting.

Auxiliaries and Constructors (One lecture) auxiliary functions [5.4]; constructors [8.1].

Graphics (Two lectures) graphics primitives; plotting a graph, recursive pictures.

Recursive Types (Two lectures) recursive types [8.3], expression trees [8.3.3], simple interpreter.

Efficiency (two lectures) efficiency [6.1], reduction step [6.2] divide and conquer (merge sort, binary search) [6.4].

Infinite Lists (Two lectures) infinite lists [7.1–7.3].

4.3 Tutorials

Students attend a one hour small group tutorial each week, usually containing about 15 students. They work on a set of exercises beforehand and then discuss the solutions with the tutor. Some sample exercises can be found in Appendix B. The tutorial exercises are designed to support the lecture material.

4.4 Practical Work

Laboratory work is an essential part of the subject. Students spend about four hours a week in the workstation lab. One lab hour is scheduled for them, and a lab demonstrator is available at that time to help them. Students use the on-line booking system to reserve a workstation at other times.

Labs are small programming exercises intended to be completed over the course of a week. Assignments are larger programs that students have three weeks to complete. Here is the current sequence of labs and assignments.

- manipulate windows and files with the Display Manager and UNIX.
- run Miranda: correct a program containing errors.
- complete a program to find the day of the week for a given date.
- find perfect numbers.

- mastermind: given a code word and a guess, print a message indicating the number of bulls (correct letters in correct position) and cows (correct letters in incorrect positions).
- assignment 1: simple transaction processing and formatting program (see Appendix).
- write some recursive functions to calculate the moving average of a list of numbers (none of Miranda's standard functions are allowed to be used in this lab).
- complete a Miranda definition of a finite state machine.
- assignment 2: graph plotting program (see Appendix).
- translate a function to find the length of a list into a machine code program.

Students use a network of Apollo DN3000 and DN2500 workstations to do their labs. They use a friendly Miranda environment (written in an ugly mixture of Pascal and Apollo DM editor command language) for testing and editing their programs. They interact with two windows: an edit window, displaying their program, and a transcript window, showing all the expressions they have evaluated and error messages from the compiler. Saving a Miranda program causes it to be recompiled. If a program has errors, typing **/e** in the transcript window causes the cursor to be placed on the line containing the error.

All submission, testing, marking and returning of labs and assignments is carried out on line (Carrington *et al.*, 1984; Whale, 1991).

5 Example: graphics

The traditional introductory computer science course has not included computer graphics. This has been because the mini and mainframe programming environments and the languages used did not support it. We have included graphics because:

- bitmapped displays are replacing character displays, so graphical output from programs is becoming more common.
- pictures usually have an obvious decomposition into sub-pictures, helping students to understand functional decomposition of problems.
- plotting pretty pictures motivates students much more than typing tedious text.
- drawing fractals provides a lovely example of the use of recursion.
- it demonstrates the use of Miranda's composite types (equivalent to Pascal variant record).
- pictures are a good example of the use of a data structure to represent the object we are computing.

5.1 Alternative graphic systems

The following two systems were considered and rejected:

Turtle graphics

Those introductory courses that have used graphics have often used *turtle graphics* (Elenbogen and O'Kennon, 1988; Liss and McMillan, 1987), first used in LOGO to teach computing to children (Papert, 1980). A picture can be drawn by giving a

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

sequence of commands to a robot turtle, of the form move Forward distance, turn Right angle, lift Pen Up, and put Pen Down.

While a turtle graphics system is easy to understand and very simple, it is rather clumsy for creating pictures. Each command changes the state (position, orientation and pen state) of the turtle. This makes putting a picture together from functions that draw subpictures rather difficult, since each function will have a side-effect of changing the turtle's state. The functions must be carefully assembled in the right order, and then changing one of them can cause changes to the pictures produced by subsequent functions. The only solution to this problem is to make each function restore the turtle state at the end to what it was when it started. This adds extra complications to each function and is still error-prone.

Procedural systems

Traditional systems such as GKS (ISO, 1985) consist of libraries of over a hundred subroutine calls to add primitives to the picture and to set and inquire attribute values (which control how the primitives are drawn). The result of executing *polygon* (n, x, y) is very state dependent – you might get a polygon if the system has been initialized without errors, and the polygon vertices are mapped onto the current clipping region after being transformed by the current transformation matrix and the current colour is not the current background colour and the current fill pattern is not all zeroes, and ... Subroutines are provided to save and restore *some* of the state, but this only goes partway towards solving this problem.

Operations such as translating part of a picture are accomplished by modifying the current transformation matrix. This is awkward to use and rather complicated to explain to first year students.

5.2 A simple graphics system

The system described here is roughly equivalent in power to GKS level 0a (ISO, 1985) without the cell array primitive and without attributes for the other primitives.

Coordinate system

We use ordered pairs of numbers to represent points

$$point \quad == \quad (num, num)$$

Any values can be used for co-ordinates; the picture is scaled to fit in a window on the display. This makes things simpler than using device coordinates, where we have to know the size in pixels of our window, or normalized device coordinates, where it is necessary to scale the picture so that it fits in a unit square.

Output primitives

There are three output primitives: polylines, filled polygons, and text. For polylines and polygons, we just specify a list of points specifying the vertices. Text has a fixed size and orientation, we specify a reference point and whether the text string lies

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

above, below, to the left, to the right, or some combination. If no alignment is specified, the text is centred about the reference point.

We use a composite type to represent a primitive object; a picture is then just a list of primitive objects.

```
align    ::=  Left | Right | Above | Below
string   ==   [char]
object   ::=  Line [point] | Polygon [point] | Text string [align] point
picture  ==   [object]
```

So the labelled square in fig. 1 could be defined by

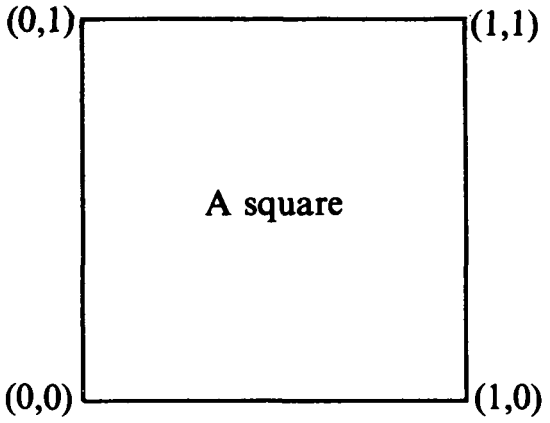

Fig. 1

```
square  =  [Line [(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)],
            Text "(0, 0)" [Below] (0, 0),
            Text "(1, 0)" [Below, Right] (1, 0),
            Text "(1, 1)" [Right] (1, 1),
            Text "(0, 1)" [Above, Left] (0, 1),
            Text "A square" [] (1/2, 1/2)]
```

Picture operations

The **draw** function takes a picture and displays it in a window. It is implemented by creating a new process and sending a description of the picture to it via a pipe. The new process (running a Pascal program) creates a new window and uses Apollo GPR calls to show the picture. If the window is resized, the picture is scaled to fit.

Since pictures are just lists, the list concatenation operator ++ can be used to combine pictures. This corresponds to drawing one picture on top of the other one.

Pictures can be translated, scaled and rotated. We can implement these operations by transforming the points in the objects making up the picture. Translating a point (x, y) by a displacement (a, b) gives $(a+x, b+y)$

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

```
shift_point              :: point → point → point
shift_point (a, b) (x, y) =  (a+x, b+y)
```

An object can be translated by translating each of its points

```
shift_object                 :: point → object → object
shift_object t (Line ps)     =  Line (map (shift_point t) ps)
shift_object t (Polygon ps)  =  Polygon (map (shift_point t) ps)
shift_object t (Text s j p)  =  Text s j (shift_point t p)
```

and we can translate a picture by translating each of its objects

```
shift   :: point → picture → picture
shift t =  map (shift_object t)
```

Scaling a point (x, y) about the origin by x and y scale factors (a, b) gives (ax, by)

```
scale_point              :: point → point → point
scale_point (a, b) (x, y) =  (a * x, b * y)
```

We could go on to define *scale_object* and *scale* similarly to *shift_object* and *object* but Miranda encourages us to generalise. We will create a function analogous to *map* called *map_picture* that applies a function to each point in a picture, so we can then say

```
scale   :: point → picture → picture
scale s =  map_picture (scale_point s)
```

Here is *map_picture*:

```
transform                 == point → point
map_picture               :: transform → picture → picture
map_picture f             =  map (map_object f)
map_object                :: transform → object → object
map_object f (Line ps)    =  Line (map f ps)
map_object f (Polygon ps) =  Polygon (map f ps)
map_object f (Text s j p) =  Text s j (f p)
```

Now, a little trigonometry yields a definition for *rotate_point*, which rotates a point about the origin by a given angle in degrees

```
rotate_point           :: num → point → point
rotate_point angle (x, y) =  (x * cos theta − y * sin theta,
                              x * sin theta + y * cos theta)
                              where theta = angle * pi/180
```

And here is *rotate* which rotates pictures

```
rotate   :: num → picture → picture
rotate a =  map_picture (rotate_point a)
```

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

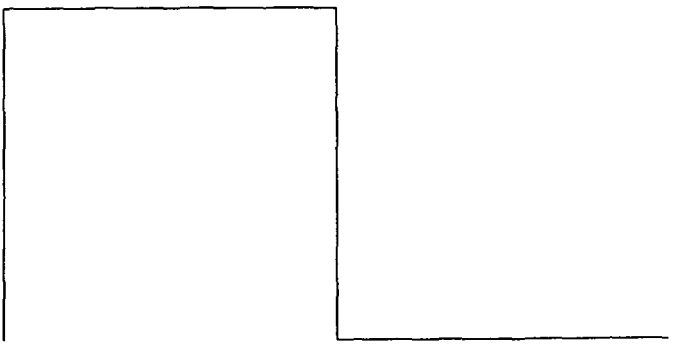

Fig. 2

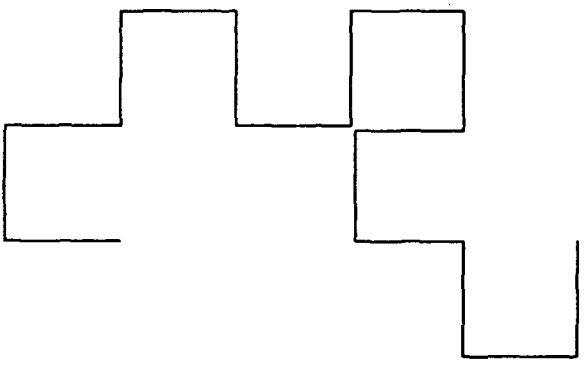

Fig. 3

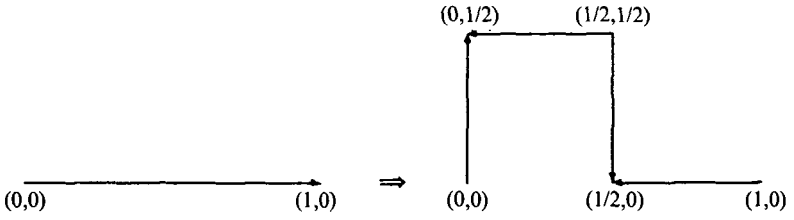

Fig. 4

5.3 Fractals

Drawing fractals makes a nice exercise in recursive programming since they are so visually appealing. Our graphics functions provide a very natural way to draw fractals.

The dragon curve is a fractal formed by folding a strip of paper. Fold a strip of paper in half and then fold it in half again (parallel to the first fold). Unfold it so that all the folds form right angles and look at it edge on. The result looks like that shown in fig. 2. If you fold it in half two more times[2] you get fig. 3. Repeating this an arbitrary number of times gives the dragon curve. The rule in fig. 4 shows how to go from one stage to the next in generating the dragon curve.

[2] Make sure you do all the folding from the same side!

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

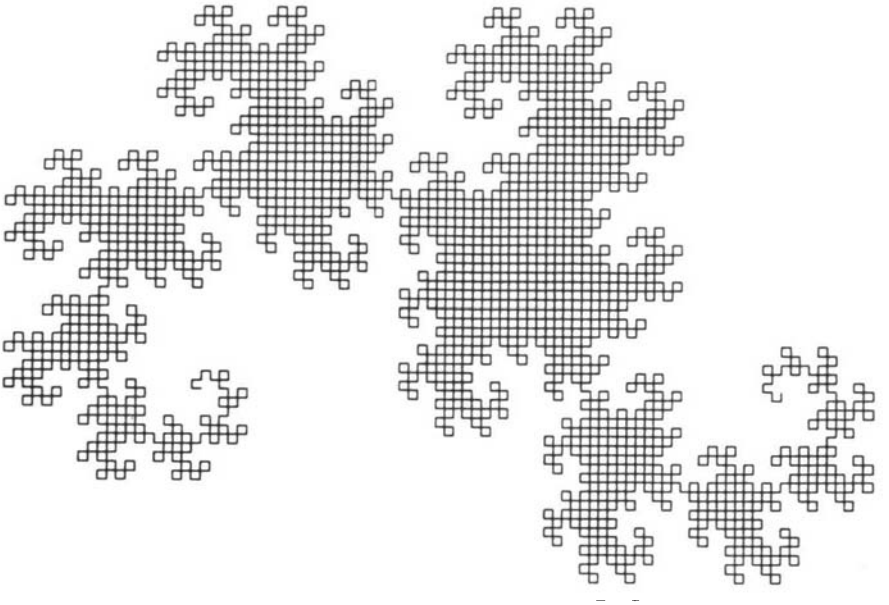

Fig. 5.

The program is almost a direct translation of the rule shown in fig. 4. A level $n+1$ dragon curve is made from four copies of the level n curve

```
dragon          ::  num→picture
dragon 0        =   [Line [(0, 0), (1, 0)]]
dragon (n+1)    =   rotate 90 drag ++
                    shift (1/2, 1/2) (rotate 180 drag) ++
                    shift (1/2, 1/2) (rotate (−90) drag) ++
                    shift (1, 0) (rotate 180 drag)
                    where drag  =  scale (1/2, 1/2) (dragon n)
```

5.4 Why Miranda is good for graphics

While a similar approach to graphics (representing a picture as a list of primitives) could be implemented in a procedural language, the necessity of allocating nodes and filling in pointers would make it much more difficult to use, and the implementation of operations like *rotate* much less illuminating.

Furthermore, it would not be possible to do until list data structures were covered – done much later with a procedural language based course.

6 Example: A simple interpreter

This section defines an interpreter for a simple language of arithmetical functions. It was felt to be worth including in this paper for a number of reasons:

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

- the language can be regarded as a subset of Miranda itself, so it reinforces the students' informal picture of how Miranda works.
- it demonstrates the use of recursive data types.
- it introduces the notions of abstract syntax and syntax trees.

6.1 A simple language of arithmetical functions

The language extends simple arithmetic (with addition, subtraction, multiplication and division) by adding user-definable constants (abbreviations) and user-definable unary functions. For example, we might define a constant pi, and functions for squaring numbers and for finding the area of a circle, thus

```
pi        = 3·14159
square x  = x*x
area r    = pi*(square r)
```

The language consists of (non-recursive) definitions of constants and unary functions. It is actually introduced to the students as a series of small extensions, but due to space limitations here we restrict ourselves to the final version.

First, *vnames* are used to represent 'variables' (user-defined constants and formal parameters to function definitions), and *fnames* are used for 'functions'. Collectively, *names* are simply lists of characters:

```
name  == [char]
vname == name
fname == name
```

To represent simple arithmetical expressions, we can use abstract syntax trees. For example, $2+3*4$ can be represented by the tree shown in fig. 6. The root of the syntax

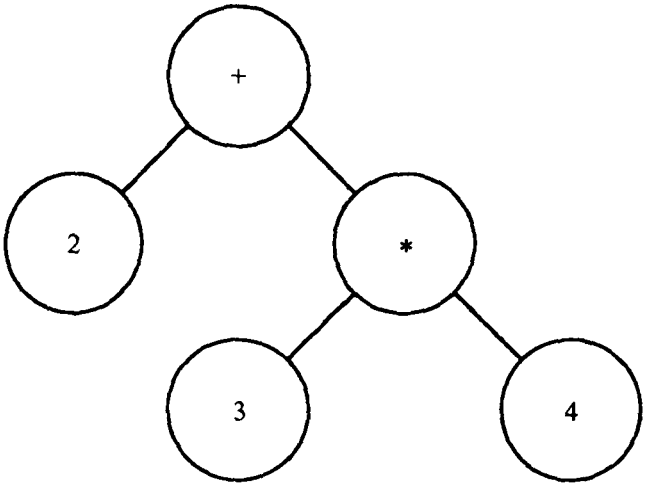

Fig. 6

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

tree (paradoxically drawn at the top in computer science!) is the 'outermost' operator – the last thing to be evaluated. Now each branch of the tree is itself a tree, so we are dealing with a recursively-defined data type. For the sample case above we could define the data type as

$$tree \quad ::= \quad Num\ num \mid Add\ tree\ tree \mid Mult\ tree\ tree$$

so the example would be represented by

$$Add\ (Num\ 2)\ (Mult\ (Num\ 3)\ (Num\ 4))$$

In the more general case, arithmetical expressions (*aexp*s) are built up from numerals and variables by arithmetical operations and function applications, so a more appropriate definition would be

$$\begin{aligned} aexp \quad &::= \quad Num\ num \mid Var\ vname \mid Exp\ aexp\ aop\ aexp \mid Func\ fname\ aexp \\ aop \quad &::= \quad Plus \mid Minus \mid Times \mid Divide \end{aligned}$$

Formally, a *variable definition* (or binding) is a pair consisting of the constant being defined (a *vname*) and the expressions which defines it (an *aexp*)

$$varDef \quad == \quad (vname,\ aexp)$$

A *function definition* is a pair consisting of the name of the function being defined (an *fname*) and a pair embodying the function's definition – namely, the formal parameter (a *vname*) and the *definiendum* (an *aexp*):

$$fnDef \quad == \quad (fname,\ (vname,\ aexp))$$

Finally, a *script* is made up of abbreviations and definitions of functions

$$script \quad == \quad ([varDef],\ [fnDef])$$

That concludes the definition of the language.

The simple example above would be written as

```
smallScript :: script
smallScript = ([p], [s, a])
              where p = ("pi", Num 3·14159)
                    s = ("square",
                         ("x",
                            Exp (Var "x") Times (Var "x")))
                    a = ("area",
                         ("r",
                            Exp (Var "pi")
                            Times
                            (Func "square" (Var "r"))))
```

in this language.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

6.2 Looking up values

How do we get the definition of a constant or function from a script? Rather than writing two separate functions, Miranda encourages us to generalise the problem, and to write a single function that can be used in either situation

```
lookup              :: [(name, *)] → name → *
lookup pairs item   =  hd values,                          if values ≠ [ ]
                    =  error (item ++ "is undefined"), otherwise
                       where values = [y | (x, y) ← pairs; x = item]
```

6.3 Evaluating expressions

We are ready to define the interpreter. The language operates in just the same way as Miranda itself: the user supplies a script and an expression to be evaluated

```
eval :: script → aexp → num
```

Numbers need no further evaluation

```
eval s (Num n) = n
```

Constants are evaluated by looking up their definitions in the appropriate part of the script and evaluating the result

```
eval s (Var x) = eval s def
                 where def = lookup (fst s) x
```

Arithmetical expressions are evaluated by finding the outermost operator, evaluating its arguments recursively, then putting the results together in the appropriate way

```
eval s (Exp a op b) = apply op (eval s a) (eval s b)
```

An auxiliary function is needed to apply the arithmetical operations

```
apply              :: aop → num → num → num
apply Plus m n     =  m+n
apply Minus m n    =  m−n
apply Times m n    =  m*n
apply Divide m n   =  m/n
```

To evaluate a function application *Func f arg*, we first look up the function's definition. This gives a pair consisting of the formal parameter, *v* say, and the *definiendum*, *rhs* say. Application causes *v* to be bound to the argument *arg* in the expression *rhs*. Finally, *rhs* is evaluated in an 'extended environment' in which *v* is an abbreviation for *arg*

```
eval (cs, fs) (Func f arg) = eval (cs', fs) rhs
                             where (v, rhs) = lookup fs f
                                   cs'      = (v, arg) : cs
```

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

That concludes the definition of the evaluator.[3]

At this stage lazy evaluation can be easily explained: if v is not actually used in *rhs*, then *arg* need never be evaluated. Closer inspection also reveals the variable bindings can be overwritten during the course of evaluation.

Students can convince themselves of the correctness of the model by stepping through the evaluation of, e.g.

$$eval\ smallScript\ (Func\ \text{"}square\text{"}\ (Func\ \text{"}square\text{"}\ (Num\ 3)))$$

on paper, to give 81.

6.4 Discussion points

From here there are many points of departure for further discussion. For example, a good exercise to set would be to write a syntax for boolean expressions and the corresponding evaluator. A more advanced exercise – but still within the students' reach – would be to design a pretty printer for arithmetical expressions (with all parentheses displayed, say)

$$unparse\ ::\ aexp \rightarrow [char]$$

The problem of parsing can also be discussed, although a full solution is beyond the scope of a first programming course

$$\begin{aligned} parseAExp &:: [char] \rightarrow aexp \\ parseScript &:: [char] \rightarrow script \end{aligned}$$

It could be pointed out that parsing is usually considered the first step an interpreter performs, so a full interpreter would be something like

$$\begin{aligned} interp &:: [char] \rightarrow [char] \rightarrow num \\ interp\ filename\ exp &= eval\ s\ e \\ &\quad \textbf{where}\ s = parseScript\ (read\ filename) \\ &\qquad\quad e = parseAExp\ exp \end{aligned}$$

In a quite different direction, another exercise would be to write auxiliary functions for checking context conditions, *viz.* that:

- abbreviations should contain no 'free variables' (what does this mean anyway?),
- and only one free variable is allowed in a function definition, namely the function's formal parameter.

Finally, the textbook (Bird and Wadler, 1988, section 8.3.3) has an example of a simple compiler for arithmetical expressions.

[3] Strictly, this definition is inadequate because it does not respect local scoping of variables. (To see this, try evaluating, e.g. $f(g\ 2)$ where $f\,x = g\,x$ and $g\,x = x$, or where $f\,x = x$ and $g\,y = x$.) To avoid such problems we must assume that, in a legal script, names are unique. In a real interpreter, more complex solutions are necessary. First year students are told that such considerations are the subject of a later subject in compilers.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

7 Problems

Student response to the subject has generally been positive. (In a survey of student feelings about all subjects, students gave it a higher rating than most other first year subjects.) There were, however, some areas that caused difficulties:

- many students found the textbook too difficult. They found it too terse and too abstract. For example, the textbook uses the higher order functions *foldr* and *foldl* extensively. These elegantly generalise a common pattern of recursion, but until students are familiar with and comfortable with this recursion pattern they do not really understand what *foldr* does, and when it should be used. Students are more comfortable with the directly recursive definition of insertion sort

$$\begin{aligned} isort\ [\,] &= [\,] \\ isort\ (x{:}xs) &= insert\ x\ (isort\ xs) \end{aligned}$$

  than with the definition using *foldr*

$$isort' = foldr\ insert\ [\,]$$

- students found currying difficult, and many never get confident enough to use it in their own programs. Fortunately, you can avoid currying in your programs at the price of defining names for partially applied functions.
- there were more students in the subject than had been planned for, making access to the workstations in peak times very difficult.
- some error messages from the Miranda compiler are truly awful. The frequently obtained type unification error message provides students with little real help as to what is wrong, other than a notification that there is a type error somewhere.

Many students would like to get Miranda for their personal computers, but implementations are not yet available. Viable alternatives for Miranda are expected to become available; we shall follow the development of Haskell in particular.

8 A comparison with Pascal

Having taught first year students programming using Pascal (the old subject) and Miranda (the new subject), a large number of points emerged where Miranda was superior. Some of these do not become obvious until one attempts to explain a particular concept to the students.

The first group of points follow from the higher-level nature of Miranda

- Pascal programming encourages students to think about implementations rather than more abstract specifications.
- Declarative Miranda implementations can be much closer to specifications. For example, the value of the binary number with digits $b_{n-1}\,b_{n-2}\ldots b_1\,b_0$ is

$$\sum_{i=0}^{n-1} 2^i b_i,$$

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

or in set notation

$$\text{sum}\{2^t b_i, \text{ where } i \in \{0, 1, \ldots, n-1\}\}.$$

In Miranda, one would write

$$\begin{aligned} value\ s &= sum\ [2\hat{}i * b!i \mid i \leftarrow [0..n-1]] \\ &\textbf{where } n = \#s \\ &\qquad b = reverse\ s \end{aligned}$$

- Solutions are developed directly in Pascal, discouraging high level description. Top down design tries to get around this difficulty, but works only with sequences of procedures and does not help with control structures.
- ○ In Miranda, there is only one way to decompose a problem – into functions. (Pascal allows many different control structures.) Strong typing helps avoid errors in functional composition. (Pascal's strong typing applies only to expressions, thus assignments and procedure calls are subject to type constraints, but sequential composition of any two statements is not subject to a constraint.)
- Reasoning about Pascal (or any imperative language) programs is possible, but difficult and rarely taught at the introductory level.
- ○ We can reason about Miranda programs by simple rewriting rules and induction

$$\begin{aligned} square\ (3+4) &\Rightarrow (3+4)*(3+4) \\ &\Rightarrow \end{aligned}$$

Or we can prove $(reverse\ s)!i = s!(\#s-1-i)$ and transform *value* to *value'*:

$$value'\ s = sum\ [2\hat{}i * s!(\#s-1-i) \mid i \leftarrow [0..\#s-1]]$$

Note that referential transparency (no side-effects) is needed for this to work.

- Students tend to understand an algorithm written in Pascal as a sequence of computations and miss the fundamental ideas – think of the Pascal implementations of quick sort, binary search and heap sort.
- ○ Algorithms can be expressed in Miranda so that the fundamental ideas are not obscured.

$$\begin{aligned} qsort\ [\,] &= [\,] \\ qsort\ (x:xs) &= qsort\ [y \mid y \leftarrow xs;\ y < x] +\!\!+ \\ &\quad\ [x] +\!\!+ \\ &\quad\ qsort\ [y \mid y \leftarrow xs;\ y \geq x] \end{aligned}$$

- The amount of skill students are able to acquire in Pascal by the end of the subject only allows them to solve toy problems.
- ○ The greater expressive power of Miranda allows more practical problems to be tackled by students (see Appendix A).
- Pascal has a number of procedures with their own special syntax for handling files. Special semantics are needed to handle interactive files. Strings are almost useless. Strings and the contents of text files are conceptually identical but have completely different syntax and semantics.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

- In Miranda, file contents and strings are just lists of characters. No special rules or syntax are required. *show* converts anything into a string for printing; *read* takes a file name and returns its contents

$$read\ ``thismonth''\ ++\ show\ (3 * 2)$$

Laziness lets you use files interactively and read large files efficiently.

- Pascal's strong typing aids in writing correct programs, but its lack of polymorphism makes writing general procedures impossible. a general matrix multiplication procedure, or a procedure to concatenate two arbitrary lists cannot be written.
- Miranda is also strongly typed, but polymorphism allows more general functions to be written. A matrix multiplication function would be of type $[[num] \rightarrow [[num]] \rightarrow [[num]]$, $++$ is of type $[*] \rightarrow [*] \rightarrow [*]$.
- In Pascal, pointers must be used to implement recursive data structures, obscuring the essentials.
- Miranda provides transparent definition of recursive types (see section 6).

The next group of points are concerned with the cleaner syntax and better programming environment of Miranda. Of course, cleaner languages such as Turing, and programming environments for Pascal with language sensitive editors and incremental compilation address some of these points, so these are not necessarily arguments for using a functional language, but they certainly can be stumbling blocks when teaching students:

- Teaching the syntax and idiosyncrasies of Pascal absorbs a lot of time that would be better devoted to the goals of the subject.
- Miranda has a simple equational syntax for definitions, and an orthogonal set of functions. (The simple rules do have a bad point: *square* -3 means subtract 3 from the *square* function.)
- The rigid structure of Pascal makes effective documentation (such as *literate programming*) difficult.
- Functions can be declared in an arbitrary order, allowing for more literate programming.
- Testing components of a Pascal program is awkward. The programmer must construct extra programs to exercise individual procedures and display the results.
- Miranda provides interactive access to all the top-level functions in a program.
- Many person-years have been lost in debates over where to put the semi-colons, *begins* and *ends* in Pascal programs.
- Miranda allows one to eschew semi-colons by using indentation, saving endless hours of arguments about where they should go and allowing the concrete syntax debate to concentrate on indentation style!

9 Conclusions

We have now been teaching Miranda as a first programming language since the beginning of 1989. During those eight sessions some 2700 students have taken the new subject. We have achieved our goals in terms of the coverage of the subject, teaching

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

much more material than has ever been covered in the equivalent subject in the past. We have also been able to set more substantial laboratory exercises and assignments than we believe would have been possible using a procedural programming language. Students have been observed to be more inclined to pay attention to small details in the solution to assignments – details that would have been neglected if a more verbose and less powerful language had been used.

This paper was written at the end of 1990. In 1992 Miranda is being used in a second year Data Structures subject; in a second year Concurrent Programming subject where it is used to implement a subset of CSP (Olszewski 1992); and in a third year Parsing and Translation subject, where it is used for assignments and also to implement tolls for context free grammar experiments.

We are pleased with the results obtained so far, and believe such an approach to teaching a first programming subject should be encouraged.

Appendix A: Assignments

The two assignments are from session 1 1990.

A.1 Assignment 1

Write a function called *phone* that prints a phone bill.[4]

Miranda phone "2425098"

Phone bill for Reynold Perin

Phone number 2425098

IDD Calls

Date	*Time*	*Number called*	*Amount*
28/01/90	23:48	001112135541212	23.94
28/03/90	04:48	001144917681234	19.74

STD Calls

Date	*Time*	*Number called*	*Amount*
12/01/90	11:30	047459812	0.42
15/01/90	12:56	043231213	2.94
13/02/90	11:45	049676545	1.89

Local Calls	3 *calls at* 0.21 *each*	0.63
Phone Rental		34.95
Total		95.64

The function *phone* is given the phone number of a person and produces a string containing the phone bill.

[4] The convention used below is that *Miranda* is the prompt printed by Miranda, the text following on the same line is the expression that is evaluated, and the following lines give the value.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

Here is a type declaration for phone

$$\begin{array}{lll} \textit{phone} & :: & \textit{string} \rightarrow \textit{string} \\ \textit{string} & == & [\textit{char}] \end{array}$$

The file *whitepages* contains the name and phone number of each person. Here is an example of what might be in it

```
4518543Paul Chung
2425098Reynold Perin
6595674Neil Willita
 569865Miechee Lee
```

Each line in this file consists of a phone number followed by the name.

There is a transaction file of the same name as each phone number. For example, the file 2425098 looks like this

```
9001022302 6643476          1
9001121130 047459812        2
9001121602 314159           1
9001151256 043231213        14
9001282348 001112135541212 114
9001311256 6643746          1
9002131145 049676545        9
9003280448 001144917681234 94
```

Each line in this file corresponds to a phone call from that phone. The first 10 characters give the date and time of the phone call in the form *YYMMDDHHMM* (two digits for the year, two digits for the month, two digits for the day of the month, two digits for the hour and two digits for the minute). Then there is a space. The next 16 characters give the number called, and the rest of the line gives the number of call units for the call.

If the number called starts with 0011, then the call is an IDD (International Direct Dial) call. If it starts with 0 (but not 0011) then it is an STD (Subscriber Trunk Dial) call. Otherwise it is a local call. You should just print the number of local calls and not the date, time and number called for each one.

Each call unit costs 21 cents. The phone rental is $34.95.

A.2 Assignment 2

In this assignment you will write some functions useful for graphing data.

The first one should be called *scatter*, and have type $\textit{string} \rightarrow [\textit{num}] \rightarrow [\textit{num}] \rightarrow \textit{picture}$. The function *scatter* takes a string, a list of x co-ordinates and a list of y co-ordinates and draws a scatter plot by drawing the string at each (x, y) point.

The result of

Miranda draw(*scatter* "+" [11, 19, 33, 17] [3·2, 12, 7, 8])

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

should be as shown in fig. A1.

Fig. A1

The second function should be called *histogram*, and have type [*num*] → *picture*. The function *histogram* draws a histogram, with the width of each bar being 0·5 and the heights given in the list of numbers.

The result of

Miranda draw (*histogram* [2, 4, 3, 1])

should be as shown in fig. A2.

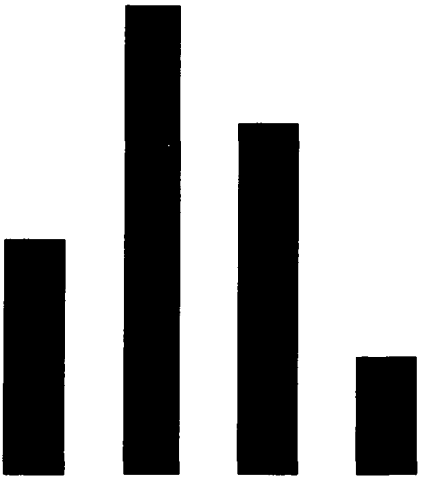

Fig. A2

The third function should be called *labelled*, and have type *picture* → *picture*, *labelled* takes any picture, draws a box around it, and adds tick marks and labels.

The result of

Miranda draw(*labelled*(*scatter* "+" [11, 19, 33, 17] [3.2, 12, 7, 8]))

should be as shown in fig. A3.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

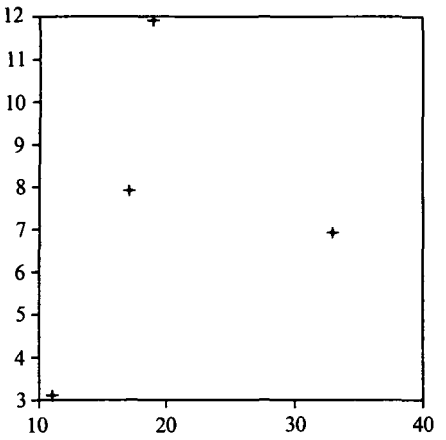

Fig. A3

And the result of

Miranda draw (*labelled*(*histogram* [2, 4, 3, 1]))

should be as shown in fig. A4.

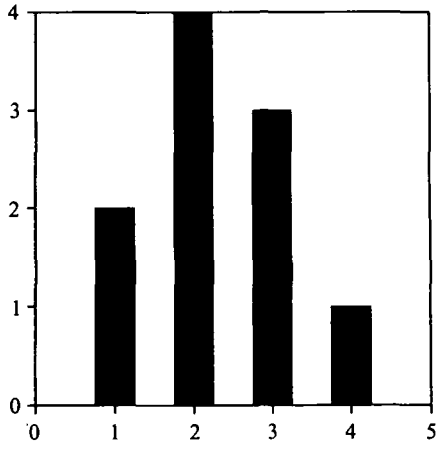

Fig. A4

Note that your *labelled* function should pick the appropriate scale for the axes of the graph based on the coordinates in the picture.

Appendix B: Tutorial exercises

The following is a selection of the tutorial exercises, each one from a different week. The complete set can be found in the lecture notes (Lambert, 1990):

1 Here is a specification for a function that calculates the definite integral of a function $f(x)$

$$integral\ f\ a\ b = \int_a^b f(x)\,dx.$$

What is the type of *integral*? (Hint: what are the types of f, a, b?)

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

2 Write down a list comprehension for a list containing all the values of type *char* which are letters of the alphabet. (The function *letter* that we saw last week determines if a *char* is alphabetic.) ([*decode* $i \mid i \leftarrow [0 \,..\, 127]$] gives all possible values of type *char*.)

3 The standard function *takewhile* is given a boolean function p and a list xs and takes values from the start of xs as long as p is *True* (see page 56 of the text).

Miranda takewhile even [4, 6, 1, 3, 2, 4]
[4, 6]

Write a definition for the function *takewhile*, using *take*, *position* and *map*.

4 Prove by induction that the length of the concatenation of two lists is the same as the sum of the individual lengths; in symbols

$$\#(xs +\!\!+ ys) = \#xs + \#ys$$

5 There are many other ways of sorting a list than insertion sort. Another method is called selection sort. The first element of the sorted list is the minimum of the list. The rest of the sorted list is found by sorting the list with the minimum element removed. Using recursion, write a function *selsort* that uses selection sort to sort a list.

6 Another fractal besides the snowflake curve is the Sierpinski triangle. The rule for the Sierpinski triangle is shown in fig. B1. This says that the triangle is drawn

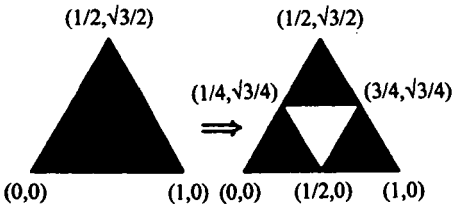

Fig. B1

by making three smaller copies of itself and placing them as indicated. Write a function *triangle* of type *num*→*picture* that draws a Sierpinski triangle to a given numerical level.

7 What is the order of $T_{ztp2\,xs\,ys}(m,n)$, if m is the length of xs and n is the length of ys?

Appendix C: Lab exercises

C.1 Lab 1

There are two parts to the first lab. In the first part, the students learn how to use the Display Manager to manipulate windows and edit files. When they invoke the lab, an edit window is created on a file containing instructions on how to read it, and edit it. In the second part they learn the basic shell commands for listing directories, copying and renaming files, and saving to and restoring from floppy discs.

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

C.2 Lab 2

In this lab, students correct the following script (which contains many errors)

Let us define a function that finds the minimum of two numbers.

```
minimum x y  =  x,  if x <= y
                y,  otherwise
```

This is a definition of a function that doesn't do anything useful but shows how you use a where clause.

```
f x y  =  minimum x y,  if x>10
       =  x-a,          otherwise
where a  =  square y+1
```

This is an attempt to write a definition of a function that adds one to its argument.

```
increment    ::  num->num
increment x  =   x
                 where x = x+1
```

The definition above doesn't make sense! How can x possibly equal $x+1$? That would mean that $0 = 1$!!

The lab instructions explain what the error messages from Miranda mean and how to correct the errors one after the other.

When they have a working program they use *give* for the first time to hand it in.

C.3 Lab 3

What you must do in this laboratory is write a function that given a date in the format (day, month, year) gives back the day of the week of that date. You should call your function *dayofweek*. For example

Miranda dayofweek (17, 3, 89)
Friday
Miranda dayofweek (25, 12, 88)
Sunday

(Your function only has to work for dates in the 20th century.)

Here, free of charge, is a type declaration for *dayofweek*

```
date       ==  (num,num,num)
dayofweek  ::  date->[char]
```

You should call the file in which you put your function definitions *date.m*

% *miranda date.m*

will ask whether to create it for you. Type *y* and press RETURN.

By the way, you should always make sure that your Miranda scripts start with the

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

characters >||. The *miranda* command does this for you automatically. If you delete these characters bad things will happen.

Remember, you must have a comment explaining what every function does and how it does it, and you must declare the type of every function. If you do not do this you will lose marks.

How to do the lab exercise

First, write a function called *leap* that determines if a year is a leap year.

Miranda leap 89
False
Miranda leap 88
True

A year is a leap year if the remainder on dividing it by 4 is zero. In Miranda *a* **mod** *b* is the remainder when *a* is divided by *b*.

Test your function *leap* to make sure it works properly. You should try both examples above and a few more.

Whenever you write a function, always test it thoroughly to make sure it does what you want before you use it in another function.

I write the next two functions that we need; they use lists, something that we have not covered yet.

mlengths takes a year and returns the number of days in each month of that year, e.g.

Miranda mlengths 88
[31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

```
mlengths     ::  num→[num]
mlengths yr  =   [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
                 where feb  =  29, if leap yr
                            =  28, otherwise
```

The function *dayno* is given a date in the form (*day*, *month*, *year*). It returns the number of days since Sunday 30th of December, 1900

```
dayno            ::  date→num
dayno (d, m, y)  =   d+sum (take (m−1) (mlengths y))+
                     (y−1)*365+ (y−1) div 4+1
```

Where on earth does this formula come from? Well, *d* is the number of days since the start of the month; *sum* (*take* (*m*−1) (*mlengths y*)) is the number of days between the start of the month and the start of the year; (*y*−1)*365+(*y*−1) **div** 4 is the number of days between the start of the year and the start of the century; (The (*y*−1) **div** 4 part allows for leap years), e.g.

Miranda dayno (1, 1, 01)
2
Miranda dayno (11, 8, 89)
32366

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

Why don't you use the MARK, COPY and PASTE keys to copy the above two definitions into *date.m*. Do not forget to include the comments. I suggest you experiment with dayno to see what it does.

Notice how *dayno* uses the function *mlengths* and *mlengths* uses the function that you wrote *leap*. When you write a Miranda program this is the way you usually work building up a complicated answer by putting together some simple functions.

Now you need to write a function called *dayname* which given the number of a day from 0 to 6 returns a string containing the name of the day

Miranda dayname 0
Sunday
Miranda dayname 6
Saturday

Once you have *dayname* working and tested it thoroughly you can use *dayno* and *dayname* to write the *dayofweek* function.

The shell command *cal* prints a calendar for a given year. Try

% *cal* 1989

You might find this useful to check your answers.

Note: your function does not have to work for dates before 1st January 1901.

Hint
Do not change the definition of *dayno* or *mlength*.

C.4 Lab 4

This is lab 4. In this lab you will learn how to use list comprehensions and a few functions that use lists. You might find it helpful to read the first few pages of chapter 3 of Bird & Wadler.

First of all, here is the definition of a function that determines the divisors of a given number (taken from page 51 of the text book).

```
divisors    ::  num→[num]
divisors n  =   [i | i←[1..n]; n mod i  =  0]
```

Why don't you use the MARK, COPY and PASTE keys to copy the above definition (including the comments describing it) into your script window. Then check to make sure that it works correctly with things like:

Miranda divisors 60
Miranda divisors 64
Miranda divisors 101

The standard function *sum* finds the sum of all the numbers in a list. Try

Miranda sum [2, 2]
Miranda sum [1..10]
Miranda sum (*divisors* 101)

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

(You don't have to define sum – it is already defined for you.)

A number is called perfect if the sum of its divisors is equal to twice the number. For example, 6 is a perfect number, since the divisors of 6 are {1, 2, 3, 6} and $1+2+3+6 = 12 = 2 \times 6$.

Write a function called *perfect* that takes a number and returns *True* if it is perfect and *False* otherwise. Do not forget to declare the type of *perfect*. (And do not forget to test it either!!)

Now write a function called *perfects* that takes a number and uses a list comprehension to return a list of all the perfect numbers less than that number. Here is what my perfects function gave me:

Miranda perfects 30
[6, 28]

There is only one problem with the program you have written: It is too slow. Try

Miranda perfects 500

to see what I mean. Do not wait for the answer. Use the DM command CTRL-C to interrupt. Move the cursor into the miranda window and press CTRL-C. (You can use the CTRL-C command to interrupt any Miranda function or shell command that is taking too long or has gone haywire.)

The rest of the lab we will spend in making the *divisors* function more efficient. **Warning:** Students often worry too much about making their programs efficient. It is more important to be correct than efficient. The definition of *divisors* that we have right now is simple and obviously correct. The more efficient function is going to be more complicated and more likely to be wrong. We are only making it more efficient because it turns out to be too slow in practice. It would not matter how inefficient it was if it had turned out to be fast enough for our purposes.

The divisors function does a lot of unnecessary work – for example, when we calculate divisors 60 we check if numbers like 58 and 59 divide 60 – we should realise that they cannot possibly divide 60.

We make use of the following rule: the divisors of a number occur in pairs. For example, $60 = 1 \times 60 = 2 \times 30 = 3 \times 20 = 4 \times 15 = 5 \times 12 = 6 \times 10$. The smaller number of each pair must be less than or equal to the square root of the original number. (Why?) If we just look at numbers up to the square root of the number we will have half the divisors.

Write a function *sdivisors* that takes a number n and returns a list of all the divisors of n that are less than or equal to *sqrt n*. (Hint: Do not use a guard like $i \leq sqrt\ n$ – modify the generator.)

Miranda [1..*sqrt* 60]
[1, 2, 3, 4, 5, 6, 7]
Miranda sdivisors 60
[1, 2, 3, 4, 5, 6]

OK, we have got half the divisors – how do we get the other half? Well, if 5 is a

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

divisor of 60, so is 60 **div** 5. Write down a list comprehension that returns the other six divisors of 60.

Now define a function *divisors*2 that uses *sdivisors* to find half the divisors, a list comprehension to find the other hand, and the ++ operator to put them together. Try

*Miranda divisors*2 60

Hint: If I want to square all the numbers from 1 to 10 I cannot say *square* [1..10] since the square function will only square a single number, but [*square i* | *i*←[1..10]]

You probably give the divisors in a different order than *divisors*. This is OK – the specification for *divisors* did not say that the divisors had to be in ascending order, so there might be more than one correct solution. Now try

*Miranda divisors*2 36

Oops!! this is not right! You probably printed the divisor 6 twice. Numbers like 36 that are perfect squares are an exception to the rule that divisors always occur in pairs. You will have to modify your definition of *divisors*2 to allow for this case. (Hint: add a guard to the list comprehension.)

When you have *divisors*2 working to your satisfaction, modify the definition of *perfect* so that it uses *divisors*2 instead of *divisors*. Now you can find all the perfect numbers less than 500. (There are exactly 3 of them.)

C.5 Lab 5

The game of Mastermind (or Bulls and Cows) is played as follows. One player secretly picks a four letter word. The other player tries to guess it. If her guess is correct the game is over. Otherwise the first player tells her how many letters in her guess that were in the secret word and in the correct position (bulls) and how many were in the word but in the wrong position (cows). For example, if the secret word was 'meet' and the guess was 'mute' the answer would be '1 bull and 2 cows.'

Your task is this lab is to write a function called *score*, of type [*char*]→[*char*]→[*char*] which could be used as part of a program to play Mastermind. For example

Miranda score "*pull*" "*limp*"
0 *bulls and* 2 *cows.*
Miranda score "*pull*" "*gulp*"
2 *bulls and* 1 *cow.*
Miranda score "*cat*" "*dog*"
0 *bulls and* 0 *cows.*
Miranda score "*pull*" "*pull*"
Correct!

1 In all the preceding labs, I have told you exactly what functions you need to write in order to work out the answer. This time you are on you own. I found I needed to write three other functions beside score. (You might use a different number of functions.)

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

You need to identify some subproblems and write a specification for a function to solve each one. If you are not clear on what a function should do, you will find it difficult to write it!

2 Your program should work with words of any length.

3 I found the −− (list difference) operator useful. We have not discussed this in class. *xs* −− *ys* means the list *xs* with each value in *ys* removed. Try

Miranda "*pull*" −− "*limp*"
Miranda # ("*pull*" −− "*limp*")
Miranda "*pull*" −− "*play*"
Miranda "*cat*" −− "*dog*"
Miranda [1, 2, 3, 4] −− [4, 1, 3, 2]

to get a feel for what it does. I also used *zip*2, #, *show* and ++

4 Your program will be tested by another program, so try to make your answer look exactly like mine (the same punctuation/spelling/etc).

5 Yes, that line above says 1 *bull and* 2 *cows* **not** 1 *bulls and* 2 *cows*. Get the rest of your program working before you worry about this.

6 If the word and guess are not the same length you should use the standard function *error* to print an error message.

7 Working out the number of bulls is easiest so you might want to start with that.

8 This lab is almost the same as exercise 3.3.12 from the text book.

9 To test your program, you can include some definitions that let you score function to play mastermind. Add this line to your program (note the semicolon)

%*include* ⟨*cs*711/*master_mind*⟩ {*score* = *score*;}

Then you can do

Miranda play 632

(or use any other number) to play the game.

C.6 Lab 6

In this lab you will write some recursive functions. All the standard functions such as #, *map*, *sum*, and so on, are defined recursively. In this lab you are not allowed to use any of them (or any list comprehensions), but must write your own versions. Have a look at section 5.3 of the text if you get stuck!

1 Write a function *mysum* that computes the sum of all the numbers in a list.

Miranda mysum [3, 2, 7]
12

(This one should be very easy – we did it in class.)

2 Write a function *average* that computes the average of the elements in a list.

Miranda average [3, 2, 7]
4.0

https://doi.org/10.1017/S0956796800000575 Published online by Cambridge University Press

(Remember, you are not allowed to use # – you must write your own function to calculate the length of a list.)

3 Write a function *mapc* which takes a function and a list and applies the function to the list, the tail of the list, the tail of the tail of the list. That is

$$mapc\ f\ xs \quad = \quad [f\ xs, f\ (tl\ xs), f\ (tl\ (tl\ xs)), \ldots]$$

Miranda mapc id [9, 4, 2]
[[9, 4, 2], [4, 2], [2]]
Miranda mapc sum [6, 1, 8, 9]
[24, 18, 17, 9]
Miranda mapc (*take* 3) [1..5]
[[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5], [5]]

Hint: have a look at the way *map* is defined.

4 Write a function *moving* that takes a number n and a list and computes the n-element moving average of the list. This is the average of the first n elements of the list, then the average of elements 2 to $n+1$, then the average of elements 3 to $n+2$ and so on.

Miranda moving 3 [6, 7, 3, 4, 4]
[5.3333333333, 4.6666666667, 3.6666666667]
Miranda moving 3 [6, 2]
[]

Hint: Would this work?

$$moving\ n \quad = \quad mapc\ (average \cdot take\ n)$$

Remember, in this lab you are not allowed to use any standard functions. If you feel you need to use a function such as *take*, you will have to write your own version and call it *mytake*.

Acknowledgements

We wish to thank Phil Wadler for comments on an earlier version of this paper. We also wish to thank the referees for helpful comments.

References

Bird, R. and Wadler, P. 1988. *Introduction to functional programming*. Prentice-Hall.

Carrington, D. A., Robinson, K. A. and Whale G. 1984. Give: A system for collecting and testing student assignments. In *Proc. 7th Australian Computer Science Conf.*

Elenbogen, B. S. and O'Kennon, M. R. 1988. Teaching recursion using fractals. In Prolog. *SIGCSE Bulletin*, **20** (1): 263–266, Feb.

ISO. 1985. Information processing systems – computer graphics – graphical kernel system (GKS) functional description. ISO 7942.

Lambert, T. 1990. *6.711 Computing 1A Programming Strand: Lecture Notes*. Department of Computer Science, University of New South Wales.

Liss, I. B. and McMillan, T. C. 1987. Fractals with turtle graphics: A CS2 programming exercise for introducing recursion. *SIGCSE Bulletin*, **19** (1): 141–146, Feb.

Olszewski, J. 1992. Communicating Processes in Pure Functional Language. In *Australian Computer Science Communications*, **14** (1): 615–630.

Papert, S. 1980. *Mindstorms: Children, Computers and Powerful Ideas*. Harvester.

Piotrowski, J. A. 1989. Abstract machines in Miranda. *SIGCSE Bulletin*, **21** (3), Sept.

Whale, G. 1991. Paperless assignment marking. In *Proc. 14th Australian Computer Science Conf.*