Introduction to Functional Programming using Miranda

Ananda Amatya & Mike Poppleton
MIS
Coventry University

8 April 1996

Contents

Preface

This is a Tutorial Introduction to Functional Programming using the Miranda Language. As there already exist quite a few books, this volume addresses the needs of the less able students. We draw heavily from the existing literature as regards examples and exercises. However, a consistent notation is followed.

Chapter 1 deals with the basic concepts of Functional Programming, and introduces Miranda Basic Types.

Chapter 2 treats the Structured Types, Lists and Tuples, and then goes on to discuss the link between pattern matching definitions and inductive proofs, and the question of efficiency of algorithms.

Chapter 3 deals with Type Synonyms and Constructed Types in Miranda (also called Algebraic Types). Some simple examples of such Concrete Types are given, followed by an end of Chapter Exercise.

Chapter 4 treats the Abstract Types in Miranda, starting with an outside view of Set Type, followed by the Signature and Representation of sets. A few exercises form the end of Chapter exercises.

Chapter 5 is an approach to Formal Specification using Miranda Abstract Types. Executable specifictions are illustrated by considering a simple application. The end of chapter exercises test the implementation of functions used.

Chapters 6 and 7 extend the Abstract Types in Miranda to Maps and their use in an extended specification for the problem used in Chapter 5.

Finally, in chapter 8, an animation package as a practical illustration of executable specifications is described. Further details on it appear in Appendix A.

The book grew out of our lecture notes to second year degree students. The main requirement was to cater for students with experience in Imperative Programming but with little mathematical maturity.

Thanks are due to Dr. Nick Godwin who contributed towards the introduction and establishment of such a course in Coventry University.

The book by Bird and Wadler [BW88] has influenced our understanding of Functional Programming, and we would like to express our appreciation of it.

The Animation Package grew out of Student Projects in Coventry University and uses the Thompson Model for interactive functional programs [Tho90].

The answers to the end of chapter exercises are available in a separate booklet.

Chapter 1

Introduction

1.1 A functional view of computation

1.1.1 Paradigm

- Model
- Framework
- Mental picture of how to think about something
- Here, how to structure a computation.

1.1.2 The Imperative, Or Procedural Paradigm

- Detail of how a given computation is to proceed
- Conceptually close to the computer hardware
- How to drive that hardware to perform the required computation
- Sequences of instructions are organised into procedures. These instructions address and alter contents of memory locations, and control their own sequence of operations (sequence, selection, and iteration).

```
- Imperative style - example - PASCAL:

        a := 100 ;          * alter memory
        b := d+e ;          * alter memory + sequence
        if a < b then       * selection
        begin
                ....
        end
```

```
else
begin
        ....
end
while b >= 17 do  * iteration
begin
        ....
end
        ....
```

- This corresponds to the von Neumann computer architecture:
  - Modifiable data store,
  - Stored instructions,
  - Run-time instruction sequence determination.

Examples

FORTRAN, Ada, C, Modula-2.

Advantage

Procedural languages are close to the hardware, hence easy to understand.

Disadvantage

- Programs are very long.
- The central algorithm (i.e. the method for solving the problem) needs to include a lot of detail of how to implement the method on the von Neumann computer (i.e. storage and manipulation of values, and control of execution flow).

1.1.3 Aside - Recall that

- *specification* is the process of defining precisely *what* tasks the user requires the proposed software system to perform. The specification should be concise, complete, and correct (this is no mean task!). The specification is not usually executable: it is usually stated in textual, or other notational form.
- *implementation* is the process of producing the programs to perform the tasks.

1.1.4 The Declarative paradigm

- This style of programming states what output the computation is to produce, for each possible input. In this sense a declarative program is very like a specification (see above); there is some discussion in the literature as to whether functional programs are actually executable specifications. The style concentrates on the algorithms for problem solution, and avoids the unnecessary detail of how values are to be stored and control of instruction sequence.
- Declarative languages can be split into three categories: *Functional*, *Relational* (or *Logical*), and *Specification* languages.
- The functional paradigm specifies computation by means of mathematical functions. Examples: Miranda, Haskell, Hope.
- The logic, or relational programming paradigm is based on mathematical relations. Examples: Prolog, Poplog.

```
***     The functional style example:
***     Function Definitions in Miranda:

> square x = x * x

> mymin x y = x,        if x <= y
>           = y,        if x > y

**      Another comparative example:
**      Imperative style - PASCAL:

SumSq := 0 ;
i := N ;
while i > 0 do
begin
        SumSq := SumSq + i*i ;
        i := i-1 ;
end

**      Functional style - Miranda: (don't worry about how it works for now!)

> sum_sq 0 = 0
> sum_sq n = n*n + sum_sq (n-1)
```

1.2 Getting started - Miranda scripts and definitions

- Miranda programming will be done on System K/H.
- Note "&" denotes the Unix prompt

```
& mira

Miranda
```

- You are now talking to the Miranda interpreter.
- See lab exercises.
- "Expressions" of arbitrary complexity can be evaluated at the prompt.

1.2.1 Example:

Here are four user-entered expressions to be evaluated by the Miranda command-line interpreter, and the results:

```
Miranda 2+3
5
Miranda 6-(3+4.2/2)
0.9
Miranda pi
3.14159265359
Miranda (max 5 10) * 2
20
Miranda square 6
36
Miranda
```

1.2.2 Standard Environment

- Miranda has a "standard", or "built-in environment", which defines all the arithmetic operators, and a number of useful functions (e.g. 'max', above) and constants (e.g. 'pi', above).
- Miranda can evaluate expressions as long as these expressions refer to definitions known to the Miranda interpreter.
- These known definitions are:
  - definitions in the "standard environment" - see the Miranda online manual for details.
  - function or constant definitions in the current user SCRIPT.

- A 'script' is simply a file containing a sequence of function and constant definitions for use by Miranda. The script contains the "program" you write in Miranda. The UNIX file containing these lecture notes is itself a Miranda script: all lines starting '>' denote Miranda definitions (e.g. 'square'). Try the above examples in the lab to satisfy yourselves.

IMPORTANT NOTE

There are TWO ways of stating definitions in a Miranda script:

1. If you want a lot of text in your script that Miranda should ignore (such as in these notes!) you want a "literate" style of script (which is what this file is). The *first nonblank character* of the file should be a '>'; this ensures that *only* lines starting with a '>' in position 1 are treated as Miranda definitions. You will first use this script file in the lab.

2. If most of your text is Miranda definitions, use a standard script (i.e. *don't* have a '>' as first character of the file). You then do *not* prefix Miranda text lines with ' >'. You must, however, then prefix non-Miranda text lines with the comment symbol '||'.

IMPORTANT REMINDER

There are *two* modes of interacting with Miranda:

1. Interpretive, where you are inputting expressions for evaluation at the Miranda prompt. This is *all* you can do in interpretive mode.

2. Programming, where you are editing your script. It's in the *script* that you do your 'programming', in the form of structured function definitions.

1.3 Values, Types, Expressions

1.3.1 Types

Values have types

num: 0 3 1.414 -6
bool: True False
char: 'a' 'Q' '6'

- Use '::' to find type of values.

Examples:

```
Miranda 2::
num
Miranda 'a'::
char
Miranda True::
bool
Miranda [1,2]::
[num]
```

1.3.2 Expressions

Denotes unique values.

Examples:

4
$1 + 5/2$
square 2
lesser 3 4
square (*lesser* 3 4)

All these are expressions.

1.3.3 Programming

Programming in Miranda is defining functions and putting them in a script file.

Editing

Editting in Miranda is creating and/or altering a script file.

Execution

When Miranda is started, some standard functions are already in the environment. This is why we can use it as a calculator. On submitting a script to the Miranda compiler, assuming no errors, the functions defined in the script are added to this environment. This is available for use during the ensuing Miranda Session.

Examples:

An example Miranda session:

```
Miranda 2+5/2
4
Miranda /f 1.3.m
```

1.3.4 Evaluation

(Often referred to as Reduction.)

Examples:

```
Miranda square 2
4
Miranda
```

The expression square 2 is evaluated giving 4. This is simplification to "simplest" form. We often say that square 2, reduced to its Normal Form is 4.

1.3.5 Literate Style

- Opposite of Standard script style.
- Everything is comment except those lines starting with '>'.
- First char of file must be '>'
- Text starting without '>' is regarde as comment.
- Only lines starting with '>' are executed.

- Must be preceded and followed by a blank line.

Examples:

```
> || This is literate script
```

```
> cube x = x*x*x
```

End of script.

1.3.6 Standard Style

- Each line preceded by '||' is comment
- Lines without '||' at the start form Miranda program

Examples:

```
|| Standard style:

>    square x = x*x  || function to find square of a number
>    ten = 10        || defines a constant variable 10
```

1.3.7 Function

Another example of a function:

```
>|| function giving the sum of squares of two numbers

>    sumsq x y = square x + square y
```

Function application forms the main part of executing Miranda scripts.

1.3.8 Constant

Constant Function:

```
>   three = 3
```

1.3.9 Value

The result of evaluating an expression.

Examples:

The value of *square* 3 is 9.
The value of $3 > 4$ is $False$.

Values are Miranda Language defined objects which cannot be simplified any further.

Examples:

4, 4.0, 'c', True, [1,2]

1.3.10 Operator

Functions written in infix notation, the operator is placed between the two arguments.

Examples:

$3 + 7$, here $+$ is a binary operator, takes two num and gives num
-6, here $-$ is a unary operator, takes one num and gives one num

1.3.11 Prefix

Normally functions use prefix notation, i.e., arguments follow the function name, e.g., *sumsq* 2 3.

1.3.12 Basic Type

num: Basic type name for numbers (integer or real).

bool: Basic type for logical values True or False.

char: Basic type for character values 'a','Z', etc..

1.3.13 Operator Precedence

In order of decreasing precedence, highest first:

Function application
^ (power)
/ dev mod
+ -

1.3.14 Associativity Of Operators

Left-associative

Binary operator '-' is Left-associative:

- 1 - 2 - 3 - 4 is same as ((1 - 2) - 3) - 4 (=-8).
- Not Right-associative which would give 1 - (2 - (3 - 4)) (=-2).

Right-associative

- Binary Operator '^' is Right-associative:
  - 2^2^3 is same as 2^(2^3) (= 2^8 = 256).
  - Not Left-associative which would give (2^2)^3 (= 4^3 = 64).

Associative

- If both left- and right- associative(referred to simply as associative).
- Binary Operator '+' is associative:
  - 1 + 2 + 3 + 4 is same as ((1 + 2) + 3) + 4 (=10).
  - And also same as 1 + (2 + (3 + 4)) (=10).

Similarly '*' is Associative but '/' is left-associative.

1.3.15 Script

Another name for a program. Consists of a list of definitions.

Example script:

```
> square x = x * x || This defines the function square
> || which takes a value as one argument
> || and returns as result the value
> || of multiplying the argument by itself.

> lesser x y  = x, if x<=y || These two lines define the function lesser
>     = y, if x>y   || which takes two values as arguments
> || and returns as the result the value
> || of the smaller of the two arguments.
```

1.4 More on Function defintion

1.4.1 Arguments: Parameters or Variables

Number of values required to be input to the function being evaluated.

Examples:

$>$ $volbox\ a\ b\ c \quad = \quad a * b * c$
has three parameters a,b,c. When evaluating, three values need to be supplied for a, b and c:

```
Miranda volbux 1 2 3
6
Miranda
```

1.4.2 Evaluation

Function/constant defintions behave like mathematical equations. Equational reasoning can be applied.

Examples:

$>$ $x = 2$
$>$ $twice\ y = 2 * y$
$>$ $result = twice\ x$
The evaluation of this amount to substituting 2 for x and 2*y for twice y. This correctly gives 4.

1.4.3 Guard Expression

Some times function definition gives different possible expressions on the right hand side of the '=' sign, depending on the values of the arguments.

Examples:

$>$ $lesser\ x\ y \quad = x, \quad if\ x \leq y$
$>$ $\qquad\qquad\quad = y, \quad otherwise$

$>$ $le\ x\ y \quad = True, \quad if\ x \leq y$
$>$ $\qquad\qquad\quad = False, \quad otherwise$

Notes:

- The conditions on the right (if.., otherwise) are called guards, and these are called guarded definitions.
- Layout rules require that the ='s are aligned vertically between the various parts of the guarded function definition.
- Each part of the function definition gives a partial function, and all guards must be given to give the total function.
- The guards must be mutually exclusive, and complete.

1.4.4 Local Definitions

Some times local definitions can be used to simplify the defintion of a function:

Examples:

$$
\begin{array}{lll}
> & areaT\ a\ b\ c & = sqrt((a+b+c)/2 * ((a+b+c)/2 - a) * \\
> & & ((a+b+c)/2 - b) * ((a+b+c)/2 - c))
\end{array}
$$

is better written using local definition as

$$
\begin{array}{lll}
> & areaT\ a\ b\ c & = sqrt\ (s * (s-a) * (s-b) * (s-c)) \\
> & & where \\
> & & s = (a+b+c)/2
\end{array}
$$

The local definition uses 'where'. Layout rules determine the expression after where.

1.5 Boolean Type

1.5.1 Binary Boolean Operators for Comparison

Comparison of num-valued expressions using binary boolean operators:

= > >= < <= ~= (*BW uses* /=)

```
Miranda 7=9
False
Miranda 2^2^3 = 256
True
Miranda 3 < 2^2 + 1
True
Miranda 2 ~= 2
False
```

Notes:

Binary boolean operator takes two num args and gives a bool result:

num → *num* → *bool*

1.5.2 Logical operators

& (and)

/ (or)
~ (not)

- Logical operators & and ∨ are binary. They take 2 bool args and give a bool result:

  bool → *bool* → *bool*

```
Miranda 3<7 & 17>18
False
Miranda 2=2 \/ 4>7
True
```

- Logical operator ~ is unary. It takes one bool arg and gives a bool result: *bool* → *bool*

```
Miranda ~2=2
False
Miranda ~3>4
True
```

Examples:

Function definitions with boolean operators:

$$\begin{array}{lll} > & test1\ a\ b & = \quad a > b\ \&\ a < 2 * b \\ > & twogtthree & = \quad 2 > 3 \end{array}$$

Notes:

Predicates use boolean operators

1.5.3 Expression evaluation on Miranda System

- Miranda expression goes through 2 stages:
  - Syntax checking:

$$\begin{array}{rl} 2\ +\ 3\ \leq\ 7 & ok \\ 5\ +\ 6\ *\ <\ C & no, \end{array}$$

    $*$ expects a number on both sides of it.
  - Type Checking:

$$\begin{array}{rl} 2\ +\ 3\ \leq\ 7 & ok \\ 17\ >\ `a' & no, \end{array}$$

    17 and $`a'$ have different types.
- Values and expressions have types:
  - Values:

$$\begin{array}{ll} True & (bool) \\ `a' & (char) \\ 2 & (num) \end{array}$$

  - Expressions:

$$\begin{array}{ll} 2 > 3 & (bool) \\ True \vee False & (bool) \\ 4 > 3 & (bool) \\ 2 * 3 + 4 & (num) \end{array}$$

- Functions also have types:
  - If function f takes an argument of type A and gives result of type B, then f has the type $A \rightarrow B$. In Miranda, this is written as

$$\begin{array}{lcl} > & f & :: A \rightarrow B \\ > & square & :: num \rightarrow num \\ > & square\ x & = x * x \end{array}$$

  - If a function f takes two args A and B and returns a result C,

$$\begin{array}{lcl} > & f & :: A \rightarrow B \rightarrow C \\ > & test & :: num \rightarrow num \rightarrow bool \\ > & test\ a\ b & = a > b \& a < 2 * b \end{array}$$

- Since operators are prefixed form of functions, their types are defined in the same way:

$$\begin{array}{lcl} (+) & :: & num \rightarrow num \rightarrow num \\ (\vee) & :: & bool \rightarrow bool \rightarrow bool \\ (\leq) & :: & num \rightarrow num \rightarrow bool \end{array}$$

  Operators with parentheses have the prefix form as functions:

```
Miranda 2<=3
True
Miranda (<=) 2 3
True
```

Notes:

- Miranda has strong typing.
- It infers the types of functions.
- No need to specify the type of a function.
- However, specifying the type of a function helps documentation. Provides checks for type errors in function definitions.

1.6 The character type

- Third of the basic types (*num*, *bool*, *char*)
- Char values:

  ```
  'a' 'C' '@' '#' '$' '\n' '\t'
  ```

- The single quotes (') are necessary.
- The characters are ordered (ASCII).
- Can compare using boolean operators:

```
Miranda 'c' < 'g'
True
Miranda 'N' < 'b'
True
```

1.6.1 Strings

- Miranda defines Strings using *char*s.
- Sequence of characters in double quotes ("):

  "fred"
  "123"

- Lexicographical comparison using boolean operators is allowed:

```
Miranda "fred" < "joe"
True
Miranda "fred" < "123"
```

Notes:

Basic types (*num*,*char*,*bool*) can be compared using boolean comparison operators.

- Boolean comparison operators can be used with *num*s:

  $2 < 3$ gives *bool* result $True$
  i.e., $(<) :: num \to num \to bool$

- They can be used also with *char*s:

  $'b' <' a'$ gives *bool* result $False$
  i.e., $(<) :: char \to char \to bool$

- They can even be used with *bool*s

$True < False$ gives $bool$ result $False$
i.e., $(<) :: bool \rightarrow bool \rightarrow bool$

- And also with strings:

  "$fred$" < "joe" gives $bool$ result $True$
  i.e., $(<) :: [char] \rightarrow [char] \rightarrow bool$

  Here $[char]$ means String.

Polymorphism

- This means that the type of $(<)$ is variable, depending on its arguments. Such operators and functions are said to be *polymorphic.*
- In order to show that the arguments of $(<)$ can be of different types, the type of $(<)$ is written as:

  $(<) :: * \rightarrow * \rightarrow bool$

  (BW uses $(<) :: a \rightarrow a \rightarrow bool$)
- This is of course, also true of all the other boolean comparison operators.

1.6.2 Type Inference

1.
   $>$ $square\ x\ =\ x\ *\ x$
   Since $*$ is defined only for num values, x must be of type num. Hence $square$ takes a num and gives a num:

   $square :: num \rightarrow num$

2.
   $>$ $lesser\ x\ y\ \ =\ \ x,\ \ if\ x\ \leq\ y$
   $>$ $\qquad\qquad\ =\ \ y,\ \ otherwise$
   Since $\leq$ is defined for any type so long both the values on its left and right are of the same type, both x and y must be of the same type, so

   $$x\ \ ::\ \ *$$

   and,

   $$y\ \ ::\ \ *$$

   Also,

   $$lesser\ \ ::\ \ * \rightarrow * \rightarrow *$$

1.7 Evaluation

1.7.1 Referential Transparency

- Within a given context, an expression always has the same value.
- Using pure reasoning, better shorter and more efficient programs may be found.
- Without it (referential transparency) such improvements are not possible.
- Side effects (assignments) violate referential transparency.

Equational Reasoning

- Evaluation of expressions is Equational.
- Equations are used to replace 'equals by equals' to simplify an expression to its Normal form.
- This process of evaluation of an expression is also called Reduction.
- Another name for replacing 'equals by equals' is Rewriting, and the equations are called Rewrite rules.

Examples:

- Definitions of functions:

$$\begin{aligned} square\ x &= x * x \qquad (square.0) \\ double\ x &= x + x \qquad (double.0) \end{aligned}$$

- Evaluation of the expression square 3:

$$\begin{aligned} & square \\ \Rightarrow \quad & (square.0) \\ & 3 * 3 \\ \Rightarrow \quad & (*,\ simple\ arithmetic\ rule) \\ & 9 \end{aligned}$$

- The expression is evaluated by pattern matching of it against the definitions (Rewrite Rules).

Examples:

Evaluation of the expression $square((double\ 3) + 2)$:

$$\begin{aligned} & square((double\ 3) + 2) \\ \Rightarrow \quad & (square.0) \\ & ((double\ 3) + 2) * ((double\ 3) + 2) \end{aligned}$$

$$
\begin{array}{ll}
\Rightarrow & (double.0) \\
 & ((3+3)+2) * ((double\ 3)+2) \\
\Rightarrow & (+) \\
 & (6+2) * ((double\ 3)+2) \\
\Rightarrow & (double.0) \\
 & (6+2) * ((3+3)+2) \\
\Rightarrow & (+) \\
 & (6+2) * (6+2) \\
\Rightarrow & (+) \\
 & 8 * (6+2) \\
\Rightarrow & (+) \\
 & 8 * 8 \\
\Rightarrow & (*) \\
 & 64
\end{array}
$$

Notes:

An 8 step reduction.

Different reduction orders are possible:

$$
\begin{array}{ll}
 & square((double\ 3)+2) \\
\Rightarrow & (double.0) \\
 & square((3+3)+2) \\
\Rightarrow & (+) \\
 & square(6+2) \\
\Rightarrow & (+) \\
 & square\ 8 \\
\Rightarrow & (square.0) \\
 & 8 * 8 \\
\Rightarrow & (*) \\
 & 64
\end{array}
$$

- Only 5 reductions.
- More efficient.

Normal Order Reduction

Outermost, leftmost first

Also called Lazy Evaluation

1.8 End Of Chapter Exercises

1. Define a function which, using *distt* or *dist* from the lecture notes, returns triangle area, given the 3 vertices as 2-tuples, or pairs. You need to know that,

   if a, b, and c are the lengths of the 3 sides, then
   using $s = (a+b+c)/2$, we have
   $squared-area = s*(s-a)*(s-b)*(s-c)$

2. Define and test a function *age* for computing a person's age in years on a given date, given the date of birth. Two arguments: the first contains the name and date of birth, the second the given date. Example evaluations of this function:

   $age\ (\text{"}Joe\text{"},(15,3,1965))\ (2,4,1992)$ $\quad answer:\ 27$
   $age\ joe\ today$ $\quad answer:\ 27$

   What is the type of *age*? Enter your type expression into your script to see if Miranda agrees. The second example above assumes that *joe* and *today* are constants that have been defined in the script, e.g. by:

   $$\begin{aligned} joe &= (\text{"}Joe\text{"},(15,3,1965)) \\ today &= (2,4,1992) \end{aligned}$$

   Notice how the constant looks like a *nullary function*, i.e. a function with no arguments.
   Further development (jumping ahead by making use of the show function and list concatenation):

   - Alter your function to yield a more user-friendly string value such as "Joe is 27", rather than a basic num value such as 27.
   - USING THE ONLINE MANUAL, and B&W 2.3.2, use library string formatting functions, format your output in tabular fashion, e.g.

Born on	on date	Joe is
15/3/1965	2/4/1992	27

3. The two solutions (roots) of the quadratic equation are obtained as follows:

   $$\begin{aligned} a*x^2 + b*x + c &= 0 \\ solutions &= (-b \pm sqrt(b^2 - 4a*c))/(2*a) \end{aligned}$$

   Define and test a Miranda function which accepts the three coefficients a, b , c as arguments, and yields as result a pair (2- tuple) whose components are the two roots. Make sure that your function copes suitably with the case where two real roots do not exist.
   Optional extra: Can you make your solution more user friendly, e.g. to output result as a pair if two roots exist, and a suitable message if not ? You will probably need to use the *error* built-in function.

4. The following series tends in the limit to $pi/4$:
   $1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - .. + .. - ...$
   Define and test a function which, given an argument n, will sum n terms of the series and multiply by 4, to yield an approximation for pi.

Optional Extra: Investigate the accuracy and time efficiency of your function, by comparing it with Miranda's built-in *pi* constant, and by giving the Miranda command /count (which will track various execution statistics).

5. (OPTIONAL brain-teaser:) The following function definition, to yield the n'th Fibonacci number, is terribly inefficient. Can you devise a more efficient definition? (Compare the relative efficiency using /count).

$$\begin{aligned} fib\ 0 &= 0 \\ fib\ 1 &= 1 \\ fib\ n &= fib\ (n-1)\ +\ fib\ (n-2) \end{aligned}$$

6. Give the types for the following functions defined before (work them out *before* using Miranda to confirm!):

   mult, *multt*, *add*, *addt*, *succ*, *pred*

7. Give types for the following function definitions:

$$\begin{aligned} one\ x &= 1 \\ applytwice\ f\ x &= f\ (f\ x) \\ condapply\ p\ f\ g\ x &= f\ x,\ if\ p\ x \\ &= g\ x,\ otherwise \end{aligned}$$

8. ([BW88]B&W 1.4.2) Give examples of functions with the following types:

$$\begin{aligned} (num &\to num) \to num \\ num &\to (num \to num) \\ (num &\to num) \to (num \to num) \end{aligned}$$

   Which brackets are redundant ?

9. ([BW88]B&W 2.5.1) Define versions of the function ($\wedge$) and ($\vee$) using patterns for the second argument. Define versions which use patterns for both arguments. Draw up a table showing the values of AND and OR for each version.

10. Define two functions:

    - to return the greatest common denominator of two numbers
    - to determine whether a number is prime or not

11. What is the type of $\sharp$ defined in the text?
    What is the type of the following:
    > $makepair\ x\ y\ =\ (x, y)$
    Define *filter* without using list comprehension. What is its type?
    Now define *filter* using pattern-matching. Is this a better definition ?

12. The function *zip* takes a pair of lists and returns a list of pairs of corresponding elements:

$$\begin{aligned} zip\ ::\ ([*], [**]) &\to [(*, **)] \\ zip\ ((a : xs), (b : ys)) &= (a, b) : zip\ (xs, ys) \end{aligned}$$

Using *zip*, define and test a scalar product function *sp*, which returns the sum of the products of respective elements of two list arguments, i.e. (informally):

$$spxsys \quad = \quad x1 * y1 + x2 * y2 + ...$$

Using *zip*, define and test a function *myzip*4 which converts a 4- tuple of lists into a list of 4-tuples. This exercise is not asking you to copy the Miranda standard environment *zip*4 definition!

13. Suppose a list xs of integers contains an equal number of odd and even numbers. Define and test a function $riffle$ so that $(riffle\ xs)$ is some rearrangement of xs such that even and odd numbers alternate.

14. ([Hol91]Holyer 8.1) What is wrong with each of the following, assuming they are to be directly evaluated? Correct, and evaluate by hand, and confirm using Miranda:

$10\ /\ 7\ div\ 3$	$[2+3]*4$
$code\ x$	$letter\ "x"$
$(1,2)\ ++\ (3,4)$	$\sharp\ 'abcd'$
$max2\ (1,2)$	$tl\ 1 : [2..5]$

15. ([Hol91]Holyer 2.8.3) Using *hd* and a list comprehension with one generator and one filter, find the first power of 2 greater than one million.

16. 16 ([Hol91]Holyer 2.8.5) Use a list comprehension with two generators and one filter to produce the list
$[(1,1),\ (1,2),\ ...(5,5)]$
of the fifteen pairs of integers between 1 and 5 for which the first number is less than or equal to the second. Then find a second comprehension to do the same thing using just two generators and no filter.

17. ([BW88]B&W 2.8.1) Declare the types for the following function definitions:

$$\begin{aligned} const\ x\ y &= x \\ subst\ f\ g\ x &= f\ x\ (g\ x) \end{aligned}$$

Check, using Miranda.

18. ([BW88]B&W 3.5.1) Consider the function all which takes a predicate p and a list xs and returns $True$ if all elements of xs satisfy p, and $False$ otherwise. Give a formal definition of all which uses $foldr$.

19. (a) ([BW88]B&W 3.5.2) Which, if any, of the following equations are true?

$$\begin{aligned} foldl\ (-)\ x\ xs &= x - sum\ xs \\ foldr\ (-)\ x\ xs &= x - sum\ xs \end{aligned}$$

(b) ([Hol91]Holyer 3.4.5) Define a version cat of the standard function concat using foldl rather than $foldr$. Verify that they have the same effect on finite lists by trying out an example using a list of numbers and another using a list of strings.

20. (a) ([Hol91]Holyer 3.4.4) Write a function capitalise which converts the first letter of a lower case word to upper case using the standard functions code and decode. You do not need to know what codes the letters have, only that the lower case ones have consecutive code numbers, as do the

upper case ones.
(b) ([Hol91]Holyer 3.4.6) Define a function join which joins two words together with a space in between. Use join, capitalise and foldr1 to define a function sentence which takes a list of words such as ["*the*", "*cat*", "*sat*","*on*", "*the*", "*mat*"] and produce a sentence such as "*The cat sat on the mat.*" by joining the words, capitalising the first word, and adding a full stop.

21. ([BW88]B&W 3.5.4) Consider the following definition of function *insert*:

$$insert\ x\ xs \quad = \quad takewhile\ (\leq x)\ xs\ +\!+\ [x]\ +\!+\ dropwhile\ (\leq x)\ xs$$

Show that if xs is a list in non-decreasing order, then so is $(insert\ x\ xs)$. Using insert, define a function *isort* for sorting a list into non-decreasing order.

22. ([Dav92]Davie 3.13.15) Write a function which converts an integer number from a given base to a number in base 10.

23. Refer to the definitions of the $\sharp$ and *reverse* functions using *foldl* in the text.
(a) Verify that

$$oneplus(x\ plusone(y\ z)) \quad = \quad plusone(oneplus(x\ y)\ z)$$

and

$$oneplus\ x\ 0 \quad = \quad plusone\ 0\ x$$

where

$$oneplus\ x\ y \quad = \quad 1\ +\ y$$

and

$$plusone\ x\ y \quad = \quad x\ +\ 1$$

(b) Veify that

$$postfix(x\ prefix(ys\ z)) \quad = \quad prefix(postfix(x\ ys)\ z)$$

and

$$postfix\ x\ [\,] \quad = \quad prefix\ [\,]\ x$$

where

$$postfix\ x\ xs \quad = \quad xs\ +\!+\ [x]$$

and

$$prefix\ xs\ x \quad = \quad [x]\ +\!+\ xs$$

24. (OPTIONAL : Lazy infinite lists) Write a definition that prints an infinite list of 1s
Write a program that prints the infinite text

"1 *sheep*, 2 *sheep*, 3 *sheep*, ..."

as an aid to insomniacs.

25. ([BW88]B&W5.1.1) Using the recursive definitions of addition and multiplication of natural numbers given in the text; prove all or some of the following familiar properties of arithmetic:

$$\begin{array}{lcll} 0 + n & = & n = n + 0 & (+ \textit{ has identity } 0) \\ 1 * n & = & n = n + 1 & (* \textit{ has identity } 1) \\ k + (m+n) & = & (k+m) + n & (+ \textit{ associative}) \\ m + n & = & n + m & (+ \textit{ commutation}) \\ k * (m*n) & = & (k*m)*n & (* \textit{ associative}) \\ k * (m+n) & = & (k*m) + (k*n) & (+ \textit{ distribution through } *) \\ m * n & = & n * m & (* \textit{ commutation}) \end{array}$$

26. ([BW88]B&W5.1.2) Prove that

$$\begin{array}{rcl} Fn+1 * Fn-1 - (Fn)^2 & = & (-1)^n \\ Fn+m & = & Fn * Fm+1 + Fn-1 * Fm \end{array}$$

for all natural numbers $n \geq 1$ and $m \geq 0$, where Fm is the mth Fibonacci number.

27. ([BW88]B&W5.1.3) The binomial coefficient, $binom\ n\ k$ denotes the number of ways of choosing k objects from a collection of n objects.

a) Give a recursive definition of *binom*.

b) Prove that if $k > n$ then $binom\ n\ k = 0$.

c) Rewrite the equation

$$sum\_over\_k\_from\_0\_to\_n\ (binom\ n\ k) \quad = \quad 2^n$$

in functional notation.
Prove that the equation is true for all natural numbers.

28. ([BW88]B&W5.3.1) Give a recursive definition of index operation $(xs!i)$.

29. ([BW88]B&W5.3.2) Give a recursive definition of *takewhile* and *dropwhile*.

30. ([BW88]B&W5.3.3) Prove the law

$$\begin{array}{rcl} init(xs\ +\!+\ [x]) & = & xs \\ last(xs\ +\!+\ [x]) & = & x \\ xs & = & init\ xs\ +\!+\ [last\ xs] \end{array}$$

for every x and every (non-empty) finite list xs.

31. ([BW88]B&W5.3.4) Prove the laws

$$\begin{array}{lcl} take\ m\ (drop\ n\ xs) & = & drop\ n\ (take\ (m\ +\ n)\ xs) \\ drop\ m\ (take\ n\ xs) & = & drop\ (m\ +\ n)\ xs \end{array}$$

for every natural number m and n and every finite list xs.

32. ([BW88]B&W5.3.5) Prove the laws:

$$\begin{aligned} map\ (f.g)\ xs &= map\ f\ (map\ g\ xs) \\ map\ f\ (concat\ xss) &= concat\ (map\ (map\ f)\ xss) \end{aligned}$$

for any functions f and g, and every finite list xss.

33. ([BW88]B&W5.3.6) Prove the law:

$$takewhile\ p\ xs\ +\!+\ dropwhile\ p\ xs \quad = \quad xs$$

for every total predicate p, and finite list xs.

34. ([BW88]B&W5.4.1) Prove that

$$(xs\ +\!+\ ys)\ -\!-\ xs \quad = \quad ys$$

for every finite list xs, ys.

35. ([BW88]B&W5.4.2) Prove that

$$reverse\ (xs\ +\!+\ ys) \quad = \quad reverse\ ys\ +\!+\ reverse\ xs$$

for every finite list xs and ys.

36. ([BW88]B&W6.1.1) Use the $T-$ and O-notations to give computation times for the following functions: hd, $last$, $(\sharp)$, fib, and $fastfib$.

37. ([BW88]B&W6.1.2) If

$$g(n) \quad = \quad O(n^2)\ -O(n^2)$$

may we conclude that
$g(n) = 0$?
What should the right-hand side of the equation be?

38. ([BW88]B&W6.2.1) Give innermost, outermost, and outermost graph reduction sequences for each of the following terms:
$cube(cube\ 3)$
$map(1+)\ (map\ (2*)\ [1,2,3])$
$hd([1,2,3]\ +\!+\ loop)$
Count the number of reduction steps in each sequence (if it terminates).

39. ([BW88]B&W6.2.2) Give the outermost reduction sequences for each of the following terms:
$zip\ (map\ sqr\ [1..3],\ map\ sqr\ [4..6])$
$take\ (1+1)(drop(3-1)[1..4])$
$take\ (42\ -6*7)(map\ sqr[1234567..7654321])$
Indicate all outermost radixes that are not reduced because of the restrictions imposed by pattern matching.

Chapter 2

Structured Types

2.1 Tuple

- Similar to Modula-2, Pascal Records
- A tuple is a collection of a specific number of values, of specific types, in specific order:

Examples:

$(6, -1)$	(num, num)
$(3.1e3, '\$', True)$	$(num, char, bool)$
$(3, (-5.2, False), "fred")$	$(num, (num, bool), [char])$

Note:

The tuple type (a, b) corresponds to the Cartesian Product $A \times B$ from set theory,

where a represents a value from SET A
and b represents a value from SET B

Examples:

Area of a triangle in cartesian geometry:

$A(xa, ya)$, $B(xb, yb)$, $C(xc, yc)$ are the three vertices of the triangle.

Need a function to find the distance between two points (xa, ya), (xb, yb)

$$dist\ (xa, ya)\ (xb, yb)$$

```
                =  sqrt(xdelta * xdelta + ydelta * ydelta)
                   where
                   xdelta = xb - xa
                   ydelta = yb - ya
```

This may be rewritten as

```
> dist pa pb  =  sqrt(xdelta * xdelta + ydelta * ydelta)
>              where
>              (xa, ya) = pa
>              (xb, yb) = pb
>              xdelta = xb - xa
>              ydelta = yb - ya
```

The given tuples are identified as the two input parameters.

Area:

```
> area pa pb pc  =  sqrt(s * (s - a) * (s - b) * (s - c))
>                   where
>                   a = dist pb pc
>                   b = dist pc pa
>                   c = dist pa pb
>                   s = (a + b + c)/2
```

2.2 More on functions ($\rightarrow$)

Examples:

- The function

  $$multt\ (x, y)\ =\ x * y$$

  has type:

  $$multt :: (num, num)\ \rightarrow num$$

- Compare with

  $$mult\ x\ y\ =\ x * y$$

  which has type:

  $$mult :: num \rightarrow num \rightarrow num$$

- In interpreting

  $$mult\ x\ y,$$

  Miranda follows a convention:

  - Function application associates to the left. This means

    $$mult\ x\ y$$

    is interpreted as:

    $$(mult\ x)\ y$$

- The result of applying $mult$ to x is a function which is then applied to y
- Examples:

  $$mult\ 3\ 4$$

  is interpreted as

  $$(mult\ 3)\ 4$$

  which applies

  $$(mult\ 3)$$

  to

  $$4$$

  to give

  $$12.$$

- Examples:

  $$mult\ 3\ 7$$

is interpreted as

$(mult\ 3)\ 7$

which applies

$(mult\ 3)$

to

7

to give

21.

- Thus the result of applying

  $mult$

  to

  3

  is a function

  $(mult\ 3)$

  which takes one *num* argument and gives a *num* result.

- Type of the function $(mult\ 3)$

  $(mult\ 3) :: num \to num$

- So *mult* takes a num argument and gives as result a function of type $num \to num$

- This is normally written as

  $mult :: num \to num \to num$

- '$\to$' is right-associative, i.e,

  $num \to num \to num$

  means

  $num \to (num \to num)$

- Functions are curried, i.e.,

  $f\ x\ y\ z$ means $((f\ x)\ y)\ z$

  $f :: * \to (* \to (* \to *))$

- The result of applying f to the first argument is to give a function which applies to the second argument giving another function which is then applied to the third argument.

- This is known as partial parametrization.

Examples:

```
>    add       ::  num → (num → num)
>    add x y   =   x + y
>    succ      ::  num → num
>    succ      =   add 1
>    pred      ::  num → num
>    pred      =   add (−1)
```

2.3 Recursion and Pattern-matching

```
> sum_sq 0 = 0
> sum_sq n = n*n + sum_sq (n-1)
>
> test0    = sum_sq 0
> test5    = sum_sq 5
```

Notes:

- This works for 0,1,2,3,..., i.e., all the natural numbers.
- Pattern evaluated in sequence in Miranda.
- If the order is reversed, does not work:

```
>|| sum_sq2 n = n*n + sum_sq2 (n-1)
>|| sum_sq2 0 = 0 ||produces unreachable case error
```

- Strictly speaking the above definition is in error, as the case 0 is given by both equations.
- This definition depends on the order of evaluation.
- The only reason it works in the first case is due to Miranda's sequential evaluation.
- However, the following always works for natural numbers:

```
> sum_sq3 (n+1) = (n+1)*(n+1) + sum_sq3 n
> sum_sq3 0     = 0
>
> test30        = sum_sq3 0
> test35        = sum_sq3 5
```

- and also this

```
> sum_sq4 0     = 0
> sum_sq4 (n+1) = (n+1)*(n+1) + sum_sq4 n
>
> test40        = sum_sq4 0
> test45        = sum_sq4 5
```

- A bit like iteration; this example terminates on evaluation of *first* pattern.
- A safer definition (less elegant) using guards

```
> sum_sq2 n = 0,                   if n=0
>           = n*n + sum_sq2 (n-1), if n>0
>           = error"Argument must be non-negative integer", otherwise
```

2.3.1 More examples:

- Fibonacci:

```
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

- Works, but changing the order of these equations will cause problem.
- However, order does not matter, if the third equation is replaced by

```
>||fib (n+2) = fib (n+1) + fib n
```

- The following does not work:

```
>||fib2 n = fib2 (n-1) + fib2 (n-2)
>||fib2 0 = 0
>||fib2 1 = 1
```

- All three below should work:

```
> fib3 (n+2) = fib3 (n+1) + fib3 n
> fib3 0     = 0
> fib3 1     = 1

> fib4 0     = 0
> fib4 (n+2) = fib4 (n+1) + fib4 n
> fib4 1     = 1

> fib5 1     = 1
> fib5 (n+2) = fib5 (n+1) + fib5 n
> fib5 0     = 0
```

- The following definition of count is correct:

```
> count 0     = 0
> count 1     = 1
> count (n+2) = 2
```

- Defined for all natural number arguments, and there is a unique definition for each argument value.
- count defined for mutually exclusive (disjoint) and complete (exhaustive) patterns.

2.4 Lists

- Linearly ordered collection of values.
- Identical to the set-theoretic concept of a sequence.
- Elements referred to by their ordinal positions in the list (1st, 2nd, 10th, etc.).
- All elements of a list must have the same type.
- Bounded (finite) or unbounded (infinite) list.

2.4.1 Examples:

```
[213,23,4,8]                              [num]
"abcDEF"                                  [char]  (String)
[(75,'a'),(60,'b'),(50,'c'),(40,'d')]     [(num,char)]
["23","qwerty","Manchester United"]       [[char]]
[(+),(*),mult']                           [num->num->num]
[]                                        [*]
```

Notes:

```
(+)     prefixed form of operator +
[]      Empty list, which can have any list type
```

2.4.2 Abbreviation for arithmetic series

```
[a..b]          [a,a+1,a+2,..,b-1,b]    (b>a)   [num]
[a..b]          [a,a-1,a-2,..,b+1,b]    (b<a)   [num]
[a,b..c]        [a,a+b,a+2*b,..a+k*d]
                where d=b-a and a+k*d <= c <= a+(k+1)*d

[1,3..10]       [1,3,5,7,9]
```

2.4.3 Infinite list

```
[1..]
```

2.5 Basic List operators

2.5.1 head

```
head::[*]->*
head x:xs = x
```

- Head – not defined for a null list.
- Returns the 1st element.

Examples:

```
head [3,4,1,2] => 3
```

2.5.2 tail

```
tail::[*]->[*]
tail x:xs = xs
```

- Tail not defined for an empty list.
- Returns the list without the 1st element of the input list.

Example:

```
tail [3,4,1,2] => [4,1,2]
```

2.5.3 init

```
init::[*]->[*]
init x:[] = []
init x:xs = x:(init xs)
```

Example:

```
init [3,4,1,2] => [3,4,1]
```

- Returns list with all but the last element.
- Not defined for null list

2.5.4 last

```
last::[*]->*
last x:[] = x
last x:xs = last xs
```

Example:

```
last [3,4,1,2] => 2
```

- Returns last element of the list.
- Not defined for null list

2.5.5 cons

- cons (construct) inserts an element at the front of a list.
- Infix form for it is ':'.

```
':'::*->[*]->[*]
```

Example:

```
8:[3,4,1,2] => [8,3,4,1,2]
P:"iggy" => "Piggy"
```

- Any list may be written using ':' and '[ ]' in its recursive constructed form:

```
[3,4,1,2] = 3:4:1:2:[]
```

- ':' is right-associateive, i.e.,

```
3:4:1:2:[] = 3:(4:(1:(2:[])))
```

2.5.6 ++

- Concatenation joins two lists. Infix form for this operator is '++'
- Both operands must be of the same type

```
'++'::[*]->[*]->[*]
```

Example:

```
[3,4,1,2] ++ [3,7] => [3,4,1,2,3,7]
```

Exercise:

Type of '#'?

```
'#'::[*]->num
```

Type of 'map' ?

```
map:: *->**->[*]->[**]
```

Type of makepair x y = (x,y)?

```
makepair::*->**->(*,**)
```

Type of fst(a,b) = a?

```
fst::(*,**)->*
```

2.6 List Comprehension

A notation for describing lists which is borrowed from set comprehension in Mathematics.

2.6.1 Notation:

$[< expression > | < qualifier >; ...; < qualifier >]$
$qualifier$: generator or filter

- There can be arbitrary number of qualifiers, e.g.,
  - generator
  - generator;filter
  - generator;generator

Example:

```
[n*n | n <- [1..5]]
=> [1,4,9,16,25]
[n*n | n <- [1..10];n mod 2 = 1]
=> [1,9,25,49,81]
[(a,b) | a <- [1..3]; b <- [1..2]]
=> [(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```

- Function definition using list comprehension:

```
divisors n = [d | d <- [1..n];n mod d = 0]
sum_sq n   = sum [m*m | m <- [1..n]]
spaces n   = [' '|j <- [1..n]]
```

- Function to return the greatest common denominator of two numbers:

```
gcd a b = max [d | d <- divisors a;b mod d = 0]
```

- Test

```
gcd 12 9:
a=12, b=9,
divisors 12,
d = 1,2,3,4,6,12,
9 mod d = 0 =>1,3 => max 3
```

```
gcd 21 14:
a=21, b=14,
divisors 21,
d = 1,3,7,21,
14 mod d = 0 =>1,7 => max 7
```

- Function to determine whether a number is prime or not:

```
is_prime n = (divisors n = [1,n])
```

2.7 Functions as Values

- Functions are first-class citizens.
- They can be used as values.

Example:

- List whose elements are functions:

```
> funlist = [tl, init]
> funlist:: [[*]->[*]]
```

as,

```
tl::[*]->[*]
init::[*]->[*]
```

2.7.1 Function with function arguments:

1. $>$ $map\ sq\ [1,2,4,5]\ =>\ [1,4,16,25]$
   - arguments:
     - sq (function),
     - $[1,2,4,5]$ (list)
2. $>$ $map\ isUpper\ "1Xa\$Q"\ \Rightarrow\ [False, True, False, False, True]$
   - argument:
     - $isUpper$ (function),
     - $"1Xa\$Q"$ (list)

Notes:

- map applies its function argument to each element of its list argument:
  $map :: (* \to **) \to [*] \to [**]$
- $\to$ right-associative.
- () appears only in type signature.

2.7.2 Higher-order Function

A function which takes a function as argument and/or returns a function as result is called a higher-ordered function.

map is a higher-ordered function.

2.7.3 Function composition

```
f.g x = f(g x)
  (.)::(**->***)->(*->**)->(*->***)
```

Function compositon (.) takes a function of type ($* \rightarrow *$) on its right and a function of type ($** \rightarrow ***$) on its left, and returns a functio of type ($* \rightarrow ***$).

```
> apply f x = f x
> apply :: (*->**) -> * -> **

> condp p x y i = x, if p i
>               = y, otherwise
> condp:: (*->bool) -> ** -> ** -> * -> **
```

2.8 More list operators and functions

2.8.1 *filter*

- *filter* as an example higher-order function over lists.

```
> filter p xs = [x | x <- xs; p x]
```

```
e.g.,
```

```
filter odd [1..6] => [1,3,5]
```

```
filter upper "QwertyKbD" => "QKD"
```

```
> filter::(*->bool)->[*]->[*]
```

2.8.2 *map*

- A definition for *map*: comprehension definition.

```
> map f xs = [],                      if xs=[]
>          = f (hd xs):map f (tl xs), otherwise
```

2.8.3 More functions for lists

```
> take n xs -- returns a list with 1st n elements of the list xs
> drop n xs -- returns a list with els of xs left after removing 1st n els
```

Example:

```
take 3 "the quick"  => "the"
drop 4 (map sq [1..6]) => [25,36]
```

Law:

- The following statement is always true for a list (A Law):

  $take\ n\ xs\ +\!+\ drop\ n\ xs\ =\ xs$

- Such statements (*laws*) are very useful.

takewhile:

```
> takewhile p xs
```

- Returns a list whose elements are all the leading elements of xs satisfying p.
- The first element that does not satisfy the predicate p and all subsequent elements are discarded.

dropwhile:

```
> dropwhile p xs
```

- All the leading elements of xs that satisfy p are discarded.
- Returns a list containing the 1st element of xs not satisfying p, and all subsequent elements of xs.

Example:

```
takewhile isDigit "321abc45" => "321"
(assume isDigit predicate exists)
dropwhile isDigit "321abc45" => "abc45"
```

Aother Law:

- Again a *law* containing takewhile and dropwhile always holds for a list:
  $takewhile\ p\ xs\ ++\ dropwhile\ p\ xs\ =\ xs$

2.9 Programming Example

Conversion table from Metric to Imperial heights:

3 arguments:

Bottom (of the range)

Top (of the range)

Step (Interval width for successive heights to be tabulated)

```
|| Basic conversion function from m to (f,i):

>    mtoimp m = (f,i)
>               where
>               totali   = m/0.0254
>               integeri = entier totali
>               f        = integeri div 12
>               i        = integeri mod 12 + totali - integeri


|| Function to produce a list of (m,(f,i)) conversion
|| pairs, giving successive rows of the conversion table.

>    mimpairs::num->num->num->[(num,(num,num))]
>    mimpairs b t s = [(m, mtoimp m) | m<- [b,b+s..t]]

|| Function to lay out output (First attempt -- not very tidy!):

>    mimptab b t s = lay (map show (minpairs b t s))

|| Some support functions for better layout:

>    showpair (m,i) = ljustify 10 (showfloat 2 m) ++ showimp i
>    showimp (f,i) = ljustify 6 (shownum f) ++ rjustify 5 (showfloat 2 i)


|| Final function to output table (using these support functions)
|| Example call at Miranda prompt:
||      mimptable 1.0 3.0 0.1
|| Produces conversion table for height 1.0 to 3.0 m
|| in increments of 0.1 m.

>    mimptable b t s = "\n" ++
>                      cjustify 22 "Metric/Imperial" ++
>                      "\n" ++
>                      cjustify 22 "Table" ++
```

```
>                       "\n"++
>                       "\n(m)=   (ft) + (ins)\n" ++
>                       lay (map showpair (mimppairs b t s))
```

Notes:

```
lay :: [[char]] -> [xhar]
```

This function concatenates a list of strings, terminating each string with a new line character, to facilitate display on an output device, e.g.,

```
lay ["str1","str2","str3"] => "str1\nstr2\nstr3"
```

which will be output as

```
str1
str2
str3
```

The inverse function, for inputting data, is lines:

```
lines:: [char] -> [char]
```

lines would be used in conjunction with read.
read takes one string argument, the name of a file in the UNIX system, and returns all the text from the file as a string:

```
read:: [char] -> [char]
```

$- is used instead of filename for keybd input.

2.10 The Fold Operators

Fold operators are general operators which can convert lists into non-list values as well as lists, depending on the arguments.

Two main fold operators: $foldr$ and $foldl$, are defined to deal with two different reduction orders, e.g., reductions starting from the right or from the left respectively.

2.10.1 Examples:

$$foldl\ (+)\ 0\ [1,2,3,4] \Rightarrow (((0+1)+2)+3)+4 \Rightarrow 10$$
$$foldr\ (+)\ 0\ [1,2,3,4] \Rightarrow 1+(2+(3+(4+0))) \Rightarrow 10$$

In addition, two more fold operators are defined as variants of these fold operators that apply to non-null lists: $foldr1$ and $foldl1$.

2.10.2 Examples:

$$foldl1\ max2\ [1,2,3,4] \Rightarrow ((1\ `max2`\ 2)\ `max2`\ 3)\ `max2`\ 4 \Rightarrow 4$$
$$foldr1\ max2\ [1,2,3,4] \Rightarrow 1\ `max2`\ (2\ `maxs`\ (3\ `max2`\ 4)) \Rightarrow 4$$

No initial value argument is taken by $foldl1$ and $foldr1$.
Not defined for null list.
A fifth fold function, $foldl'$ also exists as a variant of $foldl$ which uses the concept of *strictness* to allow reduction orders that incorporate space efficiency considerations.

2.10.3 Example:

$$\begin{aligned} sum\ [1..4] & \\ & \Rightarrow \quad foldl'\ (+)\ 0\ [1..4] \\ & \Rightarrow \quad strict(foldl'\ (+))\ (0+1)\ [2,3,4] \\ & \Rightarrow \quad foldl'\ (+)\ 1\ [2,3,4] \\ & \Rightarrow \\ & .. \\ & \Rightarrow \quad 10 \end{aligned}$$

strict means (non-lazy), the $(0+1)$ is evaluated (does not always follow the rule outermost reduction first.)

2.11 *foldr*

foldr reduces a list from right to left.
An informal definition of *foldr* using function f, value a and list $[x_1, x_2, ..., x_n]$:

$$foldr\ f\ a\ [x_1, x_2, ..., x_n] = f\ x_1\ (f\ x_2\ (...(f\ x_n\ a)...))$$

An alternative definition using instead of function f, the operator $\oplus$ (with its prefix form $(\oplus)$)would be:

$$foldr\ (\oplus)\ a\ [x_1, x_2, ..., x_n] = x_1\ \oplus\ (x_2\ \oplus\ (...(x_n\ \oplus\ a)...))$$

In particular, this means:

$$\begin{aligned}
foldr\ (\oplus)\ a\ [\,] &= a \\
foldr\ (\oplus)\ a\ [x_1] &= x_1\ \oplus\ a \\
foldr\ (\oplus)\ a\ [x_1,\ x_2] &= x_1\ \oplus\ (x_2\ \oplus\ a) \\
foldr\ (\oplus)\ a\ [x_1,\ x_2,\ x_3] &= x_1\ \oplus\ (x_2\ \oplus\ (x_3\ \oplus\ a)) \\
&\ldots
\end{aligned}$$

Example ($\oplus$ = +):

$$\begin{aligned}
& foldr\ (+)\ 0\ [1..4] \\
\Rightarrow\ & 1\ +\ (2\ +\ (3\ +\ 4\ +\ 0)) \\
\Rightarrow\ & 10
\end{aligned}$$

(i.e., sum of 1, 2, 3, 4)

Example ($\oplus$ = *):

$$\begin{aligned}
& foldr\ (*)\ 1\ [1..4] \\
\Rightarrow\ & 1\ *\ (2\ *\ (\ 3\ *\ (4\ *\ 1))) \\
\Rightarrow\ & 24
\end{aligned}$$

(i.e., prouct of 1, 2, 3, 4)

Example ($\oplus$ = ++):

$$\begin{aligned}
& foldr\ (++)\ [\,]\ [\text{"25"},\ \text{"}December\text{"},\ \text{"1994"}] \\
\Rightarrow\ & \text{"25"}\ ++\ (\text{"}December\text{"}\ ++\ (\text{"1994"}\ ++[\,])) \\
\Rightarrow\ & \text{"25 }December\text{ 1994"}
\end{aligned}$$

(i.e., concatenate "25", "December", "1994")

Notes:

- brackets group from the right
- the operator is applied from the right
- fold right

2.11.1 Type of $foldr$

$$foldr :: (* \rightarrow ** \rightarrow **) \rightarrow ** \rightarrow [*] \rightarrow **$$

type of list element: $*$
type of a: $**$
type of result of application of $\oplus$: $**$
type of $\oplus$: $(* \rightarrow ** \rightarrow **)$

The type of a $(**)$ may be the same as the type of the list element $(*)$ as in the following examples:

```
sum    = foldr (+) 0    ||add a list of nums
product= foldr (*) 1    ||multiply a list of nums
concat = foldr (++)[ ]  ||concat a list of lists
and    = foldr (&) True ||conjoin a list of bools
or     = foldr (\/)False||disjoin a list of bools
```

As $+$, $*$, $\&$, $\bigvee$ and $++$ are associative operators and for each of the above, identity element a are defined
sum (identity 0): $x + 0 = x = 0 + x$
product (identity 1): $x * 1 = x = 1 * x$
concat (identity [ ]): $xs ++ [\,] = xs = [\,] ++ xs$
and (identity $True$): $x \,\&\, True = x = True \,\&\, x$
or (identity $False$): $x \bigvee False = x = False \bigvee x$

In the above cases, as $\oplus$ is associative and a is the identity element, the following simpler definitions without brackets apply:

$$\begin{aligned} foldr\ (\oplus)\ a\ [\,] &= a \\ foldr\ (\oplus)\ a\ [x_1, x_2, ..., x_n] &= x_1 \oplus x_2 \oplus ... \oplus x_n \end{aligned}$$

This will not work if the operator is not associative as in the case of $(-)$.

Example ($\oplus = -$):

$$foldr\ (-)\ 0\ [1..4]$$

$$\Rightarrow \quad 1 - (2 - (3 - (4 - 0)))$$
$$\Rightarrow \quad -2$$

(i.e., 1 - 2 + 3 - 4 and NOT 1 - 2 - 3 - 4)

2.11.2 Definition of (#) using *foldr*

Anther case where the operator ($\oplus$) is not associative is the following definition of length of a list using *foldr*:

```
(#)=foldr oneplus 0
    where oneplus x n = 1 + n
```

The function *oneplus* is not associative:

$$\begin{aligned} oneplus\ x\ (oneplus\ y\ z) &= oneplus\ x\ (1\ +\ z) = 2\ +\ z \\ oneplus\ (oneplus\ x\ y)\ z &= oneplus\ (1\ +\ y)\ z = 1\ +\ z \end{aligned}$$

Not the same.

Example (#):

$$\begin{aligned} & foldr\ oneplus\ 0\ [1..3] \\ \Rightarrow \quad & (foldr) \\ & oneplus\ 1\ (oneplus\ 2\ (oneplus\ 3\ 0)) \\ \Rightarrow \quad & (oneplus) \\ & oneplus\ 1\ (oneplus\ 2\ (1+0)) \\ \Rightarrow \quad & (+) \\ & oneplus\ 1\ (oneplus\ 2\ 1) \\ \Rightarrow \quad & (oneplus) \\ & oneplus\ 1\ (1+1) \\ \Rightarrow \quad & (+) \\ & oneplus\ 1\ 2 \\ \Rightarrow \quad & (oneplus) \\ & 1+2 \\ \Rightarrow \quad & (+) \\ & 3 \end{aligned}$$

Example (#):

$$
\begin{array}{ll}
 & foldr\ oneplus\ 0\ ['a','b','c'] \\
\Rightarrow & (foldr) \\
 & oneplus\ 'a'\ (oneplus\ 'b'\ (oneplus\ 'c'\ 0)) \\
\Rightarrow & (oneplus) \\
 & oneplus\ 'a'\ (oneplus\ 'b'\ (1+0)) \\
\Rightarrow & (+) \\
 & oneplus\ 'a'\ (oneplus\ 'b'\ 1) \\
\Rightarrow & (oneplus) \\
 & oneplus\ 'a'\ (1+1) \\
\Rightarrow & (+) \\
 & oneplus\ 'a'\ 2 \\
\Rightarrow & (oneplus) \\
 & 1+2 \\
\Rightarrow & (+) \\
 & 3
\end{array}
$$

Type of *oneplus*:

$$oneplus\ ::\ *\ \rightarrow\ num \rightarrow\ num$$

oneplus ignores the value of the first argument, it simply counts it.

2.11.3 A function to *reverse* a List

$$
\begin{array}{lll}
reverse & = & foldr\ postfix\ [\,] \\
 & & where\ postfix\ x\ xs\ =\ xs\ +\!+\ [x]
\end{array}
$$

Example:

$$
\begin{array}{ll}
 & reverse\ "TOM" \\
\Rightarrow & (reverse) \\
 & foldr\ postfix\ [\,]\ "TOM" \\
\Rightarrow & (string) \\
 & foldr\ postfix\ [\,]\ ['T','O',\ 'M'] \\
\Rightarrow & (foldr)
\end{array}
$$

$$
\begin{array}{ll}
 & postfix\ 'T'(postfix\ 'O'(postfix\ 'M'\ [\,]))\\
\Rightarrow & (postfix)\\
 & postfix\ 'T'(postfix\ 'O'([\,]++['M']))\\
\Rightarrow & (++)\\
 & postfix\ 'T'(postfix\ 'O'\ ['M'])\\
\Rightarrow & (postfix)\\
 & postfix\ 'T'\ (['M']\ ++\ ['O'])\\
\Rightarrow & (++)\\
 & postfix\ 'T'\ (['M','O'])\\
\Rightarrow & (postfix)\\
 & ['M','O']\ ++\ ['T']\\
\Rightarrow & (++)\\
 & ['M','O','T']\\
\Rightarrow & (string)\\
 & "MOT"
\end{array}
$$

This definition of reverse appends successively the elements starting from the rightmost element of the list to the null list.

2.11.4 Definition of $Takewhile$ using $foldr$

$$
\begin{array}{lllll}
takewhile\ p & = & foldr\ (\oplus)\ [\,] & & (takewhile.1)\\
 & where & & & \\
 & x\ \oplus\ xs & = [x]\ ++\ xs, & if\ p\ x & (takewhile.2)\\
 & & = [\,], & otherwise & (takewhile.3)
\end{array}
$$

Example ($takewhile$):

$$
\begin{array}{ll}
 & takewhile\ (<'\ r')\ "bird"\\
\Rightarrow & (takewhile.1)\\
 & foldr\ (\oplus)\ [\,]\ "bird"\\
\Rightarrow & (foldr)\\
 & 'b'\ \oplus\ ('i'\ \oplus\ ('r'\ \oplus\ ('d'\ \oplus\ [\,])))\\
\Rightarrow & (takewhile.2)\\
 & 'b'\ \oplus\ ('i'\ \oplus\ ('r'\ \oplus\ (['d']\ ++\ [\,])))\\
\Rightarrow & (++.1)\\
 & 'b'\ \oplus\ ('i'\ \oplus\ ('r'\ \oplus\ ['d']))\\
\Rightarrow & (takewhile.3)\\
 & 'b'\ \oplus\ ('i'\ \oplus\ [\,])\\
\Rightarrow & (takewhile.2)
\end{array}
$$

$$
\begin{array}{ll}
 & 'b' \ \oplus \ (['i'] \ ++ \ [\,]) \\
\Rightarrow & \ (++.1) \\
 & 'b' \ \oplus \ ['i'] \\
\Rightarrow & \ (takewhile.2) \\
 & ['b'] \ ++ \ ['i'] \\
\Rightarrow & \ (++.2) \\
 & '['b',' i'] \\
\Rightarrow & \ (string) \\
 & "bi"
\end{array}
$$

Type of $(\oplus)$

$$(\oplus) :: * \ \rightarrow \ [*] \ \rightarrow \ [*]$$

2.11.5 Formal definition $foldr$

A formal definition of $foldr$ using recursive pattern matching is:

```
>  foldr f a [ ]  = a                    (foldr.0)
>  foldr f a x:xs = f x (foldr f a xs)   (foldr.1)
```

Example:

$$
\begin{array}{ll}
 & foldr \ (+) \ 0 \ [1,2,3,4] \\
\Rightarrow & (foldr.1) \\
 & (+) \ 1 \ (foldr \ (+) \ 0 \ [2,3,4]) \\
\Rightarrow & (foldr.1) \\
 & (+) \ 1 \ ((+) \ 2 \ (foldr \ (+) \ 0 \ [3,4])) \\
\Rightarrow & (foldr.1) \\
 & (+) \ 1 \ ((+) \ 2 \ ((+) \ 3 \ (foldr \ (+) \ 0 \ [4]))) \\
\Rightarrow & (foldr.1) \\
 & (+) \ 1 \ ((+) \ 2 \ ((+) \ 3 \ ((+) \ 4 \ (foldr \ (+) \ 0 \ [\,])))) \ \ (\longleftarrow) \\
\Rightarrow & (foldr.0) \\
 & (+) \ 1 \ ((+) \ 2 \ ((+) \ 3 \ ((+) \ 4 \ 0))) \\
\Rightarrow & (+) \\
 & (+) \ 1 \ ((+) \ 2 \ ((+) \ 3 \ 4))
\end{array}
$$

$\Rightarrow$ (+)
(+) 1 ((+) 2 7)
$\Rightarrow$ (+)
(+) 1 9
$\Rightarrow$ (+)
10

9 reduction steps.
The maximum space required ($\longleftarrow$) is much larger than the average.

2.12 *foldl*

foldl reduces a list from left to right.

An informal definition of *foldl* using the operator $\oplus$, list $[x_1, x_2, ..., x_n]$, and an inital value a is:

$$foldl\ (\oplus)\ a\ [x_1, x_2, ..., x_n] = (...((a\ \oplus\ x_1)\ \oplus\ x_2)...)\ \oplus\ x_n$$

This means:

$$\begin{aligned}
foldl\ (\oplus)\ a\ [\,] &= a \\
foldl\ (\oplus)\ a\ [x_1] &= a\ \oplus\ x_1 \\
foldl\ (\oplus)\ a\ [x1, x2] &= (a\ \oplus\ x_1)\ \oplus\ x_2 \\
foldl\ (\oplus)\ a\ [x_1, x_2, x_3] &= ((a\ \oplus\ x_1)\ \oplus\ x_2)\ \oplus\ x_3 \\
&\ldots
\end{aligned}$$

Notes:

Brackets group to the left.

2.12.1 Type of *foldl*

$$foldl :: (**\ \rightarrow\ *\ \rightarrow\ **)\ \rightarrow\ **\ \rightarrow\ [*]\ \rightarrow\ **$$

type of a : $**$
type of list element: $*$
type of result of applying $\oplus$: $**$
type of $\oplus$: $(**\ \rightarrow\ *\ \rightarrow\ **)$

When $\oplus$ is associative $*$ and $**$ of same type, then *foldl* and *foldr* have the same type.

Example ($\oplus\ =\ +$):

$$\begin{aligned}
& foldl\ (+)\ 0\ [1..4] \\
\Rightarrow\ & (((0\ +\ 1)\ +\ 2)\ +\ 3)\ +\ 4 \\
\Rightarrow\ & 10
\end{aligned}$$

Same as *foldr* (+) 0 [1..4].

Similarly *foldl* $(*)$ 1 [1..4] reduces to the same as *foldr* $(*)$ 1 [1..4].

On the other hand,

$foldl\ (-)\ 0\ [1..4]\ \Rightarrow\ (((0-1)-2)-3)-4\ \Rightarrow\ -10$
which is NOT the same as $foldr\ (-)\ 0\ [1..4]\ (=\ -2)$.

Even when they reduce to the same, $foldl$ is more efficient (i.e., faster or requiring less space or both) for some cases and $foldr$ in others.

For example, $foldl$ (rather its *strict* variant $foldl'$) is more efficient in *sum* and *product*, while $foldr$ is more efficient in *concat* and *and*.

Example *pack*:

$$\begin{aligned} pack\ xs &= foldl\ (\oplus)\ 0\ xs \\ &\quad where\ n\ \oplus\ x\ =\ 10\ *\ n\ +\ x \end{aligned}$$

This means:

$$\begin{aligned} pack\ [x_{n-1}, x_{n-2}, ..., x_0] &= (x_0\ *\ 10^0) + (x_1\ *\ 10^1)\ + \\ &\quad (x_2\ *\ 10^2) + ... + (x_{n-1}\ *\ 10^(n-1)) \end{aligned}$$

$$\begin{aligned} & pack\ 1234 \\ \Rightarrow\ & (((0\ \oplus\ 1)\ \oplus\ 2)\ \oplus\ 3)\ \oplus\ 4 \\ \Rightarrow\ & (((10*0+1)\ \oplus\ 2)\ \oplus\ 3)\ \oplus\ 4 \\ \Rightarrow\ & ((1\ \oplus\ 2)\ \oplus\ 3)\ \oplus\ 4 \\ \Rightarrow\ & ((10*1+2)\ \oplus\ 3)\ \oplus\ 4 \\ \Rightarrow\ & (10^2*1+10^1*2+3)\ \oplus\ 4 \\ \Rightarrow\ & 10^3*1+10^2*2+10^1*3+4 \end{aligned}$$

2.12.2 Definition of (#) using $foldl$

```
(#) = foldl plusone 0
      where plusone n x = n+1
```

Example (# using $foldl$**):**

$$\begin{aligned} & foldl\ plusone\ 0\ [1..3] \\ \Rightarrow\ & (foldl) \\ & plusone\ (plusone\ (plusone\ 0\ 1)\ 2)\ 3 \\ \Rightarrow\ & (plusone) \\ & plusone\ (plusone\ (0+1)\ 2)\ 3 \\ \Rightarrow\ & (+) \\ & plusone\ (plusone\ 1\ 2)\ 3 \end{aligned}$$

$\Rightarrow$ $(plusone)$

$plusone\ (1+1)\ 3$

$\Rightarrow$ $(+)$

$plusone\ 2\ 3$

$\Rightarrow$ $(plusone)$

$(2+1)$

$\Rightarrow$ $(+)$

3

2.12.3 Definition of *reverse* using *foldl*

```
reverse =   foldl prefix []
            where prefix xs x = [x] ++ xs
```

Example (*reverse* a list using *foldl*):

$reverse['a','b','c']$

$\Rightarrow$ $(reverse)$

$foldl\ prefix\ [\,]\ ['a','b','c']$

$\Rightarrow$ $(foldl)$

$prefix\ (prefix\ (prefix\ [\,]\ 'a')\ 'b')\ 'c'$

$\Rightarrow$ $(prefix)$

$prefix\ (prefix\ (['a']\ ++\ [\,])\ 'b')\ 'c'$

$\Rightarrow$ $(++)$

$prefix\ (prefix\ ['a']\ 'b')\ 'c'$

$\Rightarrow$ $(prefix)$

$prefix\ (['b']\ ++\ ['a'])\ 'c'$

$\Rightarrow$ $(++)$

$prefix\ ['b','a']\ 'c'$

$\Rightarrow$ $(prefix)$

$['c']\ ++\ ['b','a']$

$\Rightarrow$ $(++)$

$['c','b','a']$

These are equivalent to the corresponding definitions using *foldr*, but are more efficient.

2.13 *foldl1 & foldr1*

$foldl1$ and $foldr1$ take only two arguments, the function to be applied to the sucessive elements of the list, and the list. There is no a, the argument giving the initial or stating value.

2.13.1 $foldl1$

$foldl1$ is a variant of $foldl$ with the following informal properties:

$$foldl1\ (\oplus)\ [x_1, x_2, ..x_n] = (..((x_1\ \oplus\ x_2)\ \oplus\ x_3)..)\ \oplus\ x_n$$

More specifically,

$$\begin{aligned} foldl1\ (\oplus)\ [x_1] &= x_1 \\ foldl1\ (\oplus)\ [x_1, x_2] &= x_1\ \oplus\ x_2 \\ foldl1\ (\oplus)\ [x_1, x_2, x_3] &= (x_1\ \oplus\ x_2)\ \oplus\ x_3 \\ &\ldots \end{aligned}$$

It is undefined for an empty list, and is related to $foldl$ as follows:

$$foldl1\ (\oplus)\ xs\ =\ foldl\ (\oplus)\ (hd\ xs)(tl\ xs)$$

Example (maximum element of a list):

The function $maxel$ to find the maximum element of a non-empty list is defined by,

$$\begin{aligned} maxel\ &= foldl1\ (max) \\ &\quad where \\ &\quad x\ max\ y\ =\ x, \quad if\ x > y \\ &\qquad\quad =\ y,\ otherwise \end{aligned}$$

Here max is a binary operator that finds the maximum of its two arguments.

$$\begin{aligned} & maxel\ ['a','n','n','e'] \\ \Rightarrow\ & foldl1\ (max)\ ['a','n','n','e'] \\ \Rightarrow\ & (('a'\ max\ 'n')\ max\ 'n')\ max\ 'e' \\ \Rightarrow\ & ('n'\ max\ 'n')\ max\ 'e' \\ \Rightarrow\ & 'n'\ max\ 'e' \\ \Rightarrow\ & 'n' \end{aligned}$$

2.13.2 $foldr1$

Similarly $foldr1$ is defined as,

$$foldr1\ (\oplus)\ [x_1, x_2, ..x_n] = (x_1\ \oplus\ (x_2\ \oplus\ ..(x_{n-1}\ \oplus\ x_n)..))$$

More specifically,

$$\begin{aligned} foldr1\ (\oplus)\ [x_1] &= x_1 \\ foldr1\ (\oplus)\ [x_1, x_2] &= x_1\ \oplus\ x_2 \\ foldr1\ (\oplus)\ [x_1, x_2, x_3] &= x_1\ \oplus\ (x_2\ \oplus\ x_3) \\ &\ldots \end{aligned}$$

It is also undefined for an empty list, and is related to $foldr$ as follows:

$$foldr1\ (\oplus)\ xs\ =\ foldr\ (\oplus)\ (last\ xs)(init\ xs)$$

Example (minimum element of a list):

The function *minel* to find the maximum element of a non-empty list is defined by,

$$\begin{aligned} minel\ &= foldr1\ (min) \\ &\quad where \\ &\quad x\ min\ y\ =\ x, \quad if\ x < y \\ &\qquad\quad =\ y,\ otherwise \end{aligned}$$

Here *min* is a binary operator that finds the minimum of its two arguments:

$$\begin{aligned} & minel\ ['j','o','h','n'] \\ \Rightarrow\ & foldr1\ (min)\ ['j','o','h','n'] \\ \Rightarrow\ & 'j'\ min\ ('o'\ min\ ('h'\ min\ 'n')) \\ \Rightarrow\ & 'j'\ min\ ('o'\ min\ 'h') \\ \Rightarrow\ & 'j'\ min\ 'h' \\ \Rightarrow\ & 'o' \end{aligned}$$

Both have the type:

$$foldl1,\ foldr1 :: (*\ \rightarrow\ *\ \rightarrow\ *)\ \rightarrow\ [*]\ \rightarrow\ *$$

2.14 strict $foldl - foldl'$

A recursive pattern matching definition of the strict version $foldl'$ of $foldl$ is:

```
>   foldl' f a []     = a                    || (foldl'.0)
>   foldl' f a x:xs   = foldl' f (f a x) xs  || (foldl'.1)
```

Example: $foldl'\ (+)\ 0$

$foldl'\ (+)\ 0\ [1,2,3]$

$\Rightarrow$ $(foldl'.1)$
$foldl'\ (+)\ ((+)\ 0\ 1)\ [2,3]$

$\Rightarrow$ $(+)$
$foldl'\ (+)\ 1\ [2,3]$

$\Rightarrow$ $(foldl'.1)$
$foldl'\ (+)\ ((+)\ 1\ 2)\ [3]$

$\Rightarrow$ $(+)\ (innermost\ reductions\ -\ strict)$
$foldl'\ (+)\ 3\ [3]$

$\Rightarrow$ $(foldl'.1)$
$foldl'\ (+)\ ((+)\ 3\ 3)\ [\,]$

$\Rightarrow$ $(+)\ (innermost\ reductions\ -\ strict)$
$foldl'\ (+)\ 6\ [\,]$

$\Rightarrow$ $(foldl'.0)$
6

7 reduction steps.

The maximum space required remains virtually the same throughout the reduction, $(O(1))$. Space efficiency is improved, by choosing a mixture of outer and inner reuctions.

Formal definition of $foldr$ using recursive pattern matching is:

```
>   foldr f a []     = a                    || (foldr.0)
>   foldr f a x:xs   = f x (foldr f a xs)   || (foldr.1)
```

Example:$foldr\ (+)\ 0$

$foldr\ (+)\ 0\ [1,2,3]$

$\Rightarrow$ $(foldr.1)$
$(+)\ 1\ (foldr\ (+)\ 0\ [2,3])$

$\Rightarrow$ $(foldr.1)$
$(+)\ 1\ ((+)\ 2\ (foldr\ (+)\ 0\ [3]))$

$\Rightarrow$ $(foldr.1)$

$(+)\ 1\ ((+)\ 2\ ((+)\ 3\ (foldr\ (+)\ 0\ [\,])))\ \ (<--)$

$\Rightarrow$ $(foldr.0)$

$(+)\ 1\ ((+)\ 2\ ((+)\ 3\ 0))$

$\Rightarrow$ $(+)$

$(+)\ 1\ ((+)\ 2\ 3)$

$\Rightarrow$ $(+)$

$(+)\ 1\ 5$

$\Rightarrow$ $(+)$

6

7 reduction steps.

The maximum space required `<--` is much larger than in the previous case using $foldl'$, $O(n)$.

2.15 Pattern Matching on Lists

"cons" operator is written as ':' in pattern matching:

```
>   lengthp            :: [*] → num
>   lengthp [ ]        =  0                      ||(lengthp.1)
>   lengthp (a : xs)   =  1 + lengthp xs         ||(lengthp.2)

>   sump               :: [num] → num
>   sump [ ]           = 0                       ||(sump.1)
>   sump (a : xs)      =  a + sump xs            ||(sump.2)

>   mapp               :: (* → **) → [*] → [**]
>   mapp f [ ]         =  [ ]                    ||(mapp.1)
>   mapp f (x : xs)    =  f x : mapp f xs        ||(mapp.2)
```

2.15.1 Using guards (Non-pattern matching)

```
>   lengthg xs   =  0,                    if xs = [ ]
>                =  1 + lengthg.tl xs,    otherwise

>   sumg xs      =  0,                    if xs = [ ]
>                =  hd xs + sumg.tl xs,   otherwise

>   mapg f xs    =  [],                   if xs = []
>                =  f.hd xs : map f (tl xs),   otherwise
```

(uses function head (*hd*) and tail (*tl*) and function composition '.')

2.15.2 More complex pattern-matching definitions

```
>   mylimit :: [num] → num
>   mylimit (a : b : xs)     =  a,                    if abs(a − b) < 0.000001
>                            =  mylimit (b : xs),     otherwise
>   mylimit other            =  error "..."

>   error :: String → *
```

2.16 Infinite lists

Miranda provides infinitely long lists.

Try evaluating the expression

[1..]

It produces every integer, starting with 1.

How can the machine hold an infinite object?

Only LAZY machines (such as Miranda) can do so. They do it by not attempting to evaluate anything they do not need to. So as long as you only need to look at a finite part of an infinite object, everything will be all right.

Consider the following,

```
> ints = 1 : map (1+) ints      ||(ints.1)
```

This is the infinitely long list of integers. In fact, '*ints* = [1..]'.

Let us see how a lazy machine evaluate '*take*2*ints*'. It finds the *outermost* expression it can match:

```
    take 2 ints
⇒     (ints.1)
    take 2 (1 : map (1+) ints)
⇒     (+ patterns)
    take (1 + 1) (1 : map (1+) ints)
⇒     (take.3)
    1 : take 1 (map (1+) ints)
⇒     (ints.1)
    1 : take 1 (map (1+) (1 : map (1+) ints))
⇒     (map.2)
    1 : take 1 ((1+) 1 : map (1+) ints)
⇒     (+)
    1 : take 1 (2 : map (1+) ints)
⇒     (+ patterns)
    1 : take (0 + 1O) (2 : map (1+) ints)
⇒     (take.3)
    1 : 2 : take 0 (map (1+) ints)
⇒     (take.1)
    1 : 2 : [ ]
```

$$
\begin{array}{ll}
\Rightarrow & (\mathit{pretty\ printing}) \\
 & [1,\ 2]
\end{array}
$$

Conversely, an EAGER implementation evaluates the *innermost * expression it can match.

$$
\begin{array}{ll}
 & \mathit{take}\ 2\ \mathit{ints} \\
\Rightarrow & (\mathit{ints}.1) \\
 & \mathit{take}\ 2\ (1\ :\ \mathit{map}\ (1+)\ \mathit{ints}) \\
\Rightarrow & (\mathit{ints}.1) \\
 & \mathit{take}\ 2\ (1\ :\ \mathit{map}\ (1+)\ (1\ :\ \mathit{map}\ (1+)\ \mathit{ints})) \\
\Rightarrow &
\end{array}
$$

...and so on for ever, only ever using '$\mathit{ints}.1$'...

The ability to have potentially infinite values allows us to write some very elegant and simple programs, such as 'ints'.

2.17 Recursion and Induction

Function defintions using recursion allow inductive proofs of useful Laws. Main areas of application: Natural Numbers and Lists.

- Natural numbers:

  Every natural number is either 0 or else has the form $(n + 1)$ (i.e., 1, or 2, or 3, ...) for some n (= 0, or 1, or 2, ...).

- Lists:

  Every list is either empty ([ ]), or has the form $x : xs$ (i.e., one element list, or two elements list, or three elements list, ...) for some x and xs.

2.18 Natural Numbers

2.18.1 Example: Exponentiation

$x \wedge n$, x to power n, n natural number

Recursive definition:

$$\begin{array}{lll} > & x \wedge 0 & = 1 \quad || (\wedge.1) \\ > & x \wedge (n+1) & = x * (x \wedge n) \quad || (\wedge.2) \end{array}$$

Note:

Pattern matching. n - natural number (0,1,2,...)

Non-pattern matching:

$$\begin{array}{lll} > & x \wedge n & = 1, \quad if\ n = 0 \\ > & & = x * x \wedge (n-1), \quad if\ n > 0 \end{array}$$

Note:

- Pattern matching definition: easy to use equational reasoning.
- Ideal for proofs using mathematical induction.
- Easy to check that the definition covers every case:
  $x \wedge n$ defined for all natural numbers n:
  $for\ n = 0$ $(\wedge.1)$
  $for\ n > 0$ $(\wedge.2)$

Reduction Example:

$$\begin{array}{ll} & 2 \wedge 3 \\ \Rightarrow & (\wedge.2,\ n = 2) \\ & 2 * (2 \wedge 2) \\ \Rightarrow & (\wedge.2,\ n = 1) \\ & 2 * (2 * (2 \wedge 1)) \\ \Rightarrow & (\wedge.2,\ n = 0) \\ & 2 * (2 * (2 * (2 \wedge 0))) \\ \Rightarrow & (\wedge.1) \end{array}$$

$$
\begin{array}{ll}
 & 2 * (2 * (2 * 1))) \\
\Rightarrow & \quad (arith *) \\
 & 2 * (2 * 2)) \\
\Rightarrow & \quad (arith *) \\
 & 2 * 4 \\
\Rightarrow & \quad (arith *) \\
 & 8
\end{array}
$$

Here, 2 ^ 3 matches (^.2) for $n = 2$.

For every natural number power, n, x ^ n matches with EITHER the equation (^.1) when $n = 0$, OR the equation (^.2) when $n = 1$, or 2,... but NEVER both.

- *Exhaustive:* 0 and $n + 1$ cover all natural numbers.
- *Disjoint:* No natural number matches more than one pattern.
- *Order:* Order of equations immaterial.

Recursive definition:

x ^ 0 has a value given by the equation (^.1)
and
x ^ $(n + 1)$ defined in terms of x ^ n by equation (^.2)

Proof by mathematical Induction:

Inductive proof of a proposition $P(n)$ for all natural numbers n, consists of two steps:
case 0 : *Proof that* $P(0)$ *holds*; *and*
case $(n + 1)$: *Proof that if* $P(n)$ *holds then so does* $P(n + 1)$.

Proof by Induction of a Law for exponents:

To prove
x ^ $(m + n) = (x$ ^ $m) * (x$ ^ $n)$
for all x and all natural numbers m, n.

Proof by induction on m.

case 0:

$$x \text{ ^ } (0 + n)$$

$$
\begin{array}{ll}
\Rightarrow & (+) \\
& x \wedge n \\
\Rightarrow & (*) \\
& 1 * (x \wedge n) \\
\Rightarrow & (\wedge.1) \\
& (x^0) * (x^n)
\end{array}
$$

This establishes the theorem for $m = 0$.

case (m + 1):

Assume for some arbitrary m,
$x \wedge (m+n) \;=\; (x \wedge m) * (x \wedge n)$ (HYP)

$$
\begin{array}{ll}
& x \wedge ((m+1)+n) \\
\Rightarrow & (+) \\
& x \wedge ((m+n)+1) \\
\Rightarrow & (\wedge.2) \\
& x \;*\; (x \wedge (m+n)) \\
\Rightarrow & (HYP) \\
& x \;*\; (x \wedge m) * (x \wedge n) \\
\Rightarrow & (\wedge.2) \\
& (x \wedge (m+1)) * (x \wedge n)
\end{array}
$$

This establishes that the result holds for $m + 1$, if it holds for m.

2.18.2 Another Example: Fibonacci

Fibonacci numbers, $fib\ n$:

$$
\begin{array}{lllll}
> & fib\ 0 & = & & (fib.1) \\
> & fib\ 1 & = & & (fib.2) \\
> & fib\ (n+2) & = & fib\ n \;+\; fib\ (n+1) & (fib.3)
\end{array}
$$

Pattern matching definition of $fib\ n$ for n natural ($n = 0, 1, 2, ...$).
$fib\ 0$ matched by first equation
$fib\ 1$ matched by second equation
$fib\ 2$, $fib\ 3$, .. matched by third equation.
For any n the value of $fib\ n$ is defined by exactly one of the above equations.

Using algebra one can show (there are general methods of solving such second order recurrence relations):

$$
\begin{array}{lll}
fib\ k & = & (f_1\hat{}\ k\ -\ f_2\hat{}\ k)/(sqrt\ 5) \\
 & & where \\
 & & f_1 = (1 + sqrt\ 5)/2 \\
 & & f_2 = (1 - sqrt\ 5)/2
\end{array}
$$

(Exercise: Test it in Miranda.)

Inductive proof of this result:

This invovles showing equivalence of this result to the pattern matching definition of fib:

case 0:
First show that it is true for case for $fib\ 0$:

$$
\begin{array}{lll}
RHS & & \\
 & \Rightarrow & \\
 & & (f_1\hat{}\ 0\ -\ f_2\hat{}\ 0)/(sqrt\ 5) \\
 & \Rightarrow & (\hat{}.1) \\
 & & (1-1)/(sqrt5) \\
 & \Rightarrow & (algebra :-) \\
 & & 0 \\
 & \Rightarrow & \\
 & & LHS
\end{array}
$$

case 1:
Next show that it is true for $fib\ 1$:

$$
\begin{array}{lll}
RHS & & \\
 & \Rightarrow & \\
 & & (f_1^1\ -\ f_2^1)/(sqrt\ 5) \\
 & \Rightarrow & (\wedge.2) \\
 & & (f_1\ -\ f_2)/(sqrt\ 5) \\
 & \Rightarrow & (algebra) \\
 & & ((1 + sqrt\ 5)/2\ -\ (1 - sqrt\ 5)/2)/(sqrt\ 5) \\
 & \Rightarrow & (algebra) \\
 & & 1
\end{array}
$$

$\Rightarrow$

LHS

case k+2:

Now show that it is true for $fib\ k$ for any $k\ >=\ 2$:

For this, we show that if the result holds for $fib\ k$ and $fib\ k+1$ then it holds for $fib\ k+2$:

Assume true for k and k+1 (hypotheses).

$$\begin{aligned} fib\ k &= (f_1^k\ -\ f_2^k)/(sqrt\ 5) \\ & \quad where \\ & \quad f_1 = (1 + sqrt\ 5)/2 \\ & \quad f_2 = (1 - sqrt\ 5)/2 \end{aligned}$$

$$\begin{aligned} fib\ (k+1) &= (f_1^(k+1)\ -\ f_2^(k+1))/(sqrt\ 5) \\ & \quad where \\ & \quad f_1\ =\ (1 + sqrt\ 5)/2 \\ & \quad f_2\ =\ (1 - sqrt\ 5)/2 \end{aligned}$$

$$\begin{aligned} & fib\ k\ +\ fib\ (k+1) \\ & \quad = (f_1^k\ -\ f_2^k\ +\ f_1^{k+1}\ -\ f_2^{k+1}))/(sqrt\ 5) \\ & \quad = (f_1^k * (1 + f_1)\ -\ f_2^k * (1 + f_2))/(sqrt\ 5) \\ & \quad = (f_1^{k+2}\ -\ f_2^{k+})))/(sqrt\ 5) \\ & \quad = fib\ (k+2) \end{aligned}$$

where we have used

$$\begin{aligned} & f_1^2 \\ & \quad = (1 + sqrt\ 5)^2/4 \\ & \quad = (1 + 2 * sqrt\ 5 + 5)/4 \\ & \quad = (6 + 2 * sqrt\ 5)/4 \\ & \quad = (3 + sqrt\ 5)/2 \\ & \quad = 1\ +\ (1 + sqrt\ 5)/2 \\ & \quad = 1 + f_1 \end{aligned}$$

and

$$f_2^2$$

$$
\begin{aligned}
&= (1 - sqrt\ 5)^2/4 \\
&= (1 - 2 * sqrt\ 5 + 5)/4 \\
&= (6 - 2 * sqrt\ 5)/4 \\
&= (3 - sqrt\ 5)/2 \\
&= 1\ +\ (1 - sqrt\ 5)/2 \\
&= 1 + f_2
\end{aligned}
$$

Proved.

2.18.3 Some other Recursive definitions (+ and *):

$+$:

$$
\begin{aligned}
0\ +\ n &= n && (+.1) \\
(m+1)+n &= (m+n)+1 && (+.2)
\end{aligned}
$$

$*$:

$$
\begin{aligned}
0\ *\ n &= 0 && (*.1) \\
(m+1)*n &= (m*n)\ +\ n && (*.2)
\end{aligned}
$$

2.19 List: *zip*

2.19.1 *zip*

$zip :: ([*], [**]) \rightarrow [(*, **)]$

Recursive definition of *zip*:

$$
\begin{array}{lllll}
> & zip([\,], ys) & = [\,] & & || \ (zip.1) \\
> & zip(x : xs, [\,]) & = [\,] & & || \ (zip.2) \\
> & zip(x : xs, y : ys) & = (x, y) : zip(xs, ys) & & || \ (zip.3)
\end{array}
$$

Notes:

- Exhaustive:
  - First list empty covered by $(zip.1)$
  - Both empty covered by $(zip.1)$
  - Second empty, first non-empty covered by $(zip.2)$
  - Neither empty covered by $(zip.3)$
- Disjoint:
  - Only one equation true for every possible pair of lists.

A Law involving *zip*:

- The result of zip is another list with length minimum of the two list lengths. For evey finite list xs and ys:

$$
\begin{array}{lllll}
> & \sharp zip(xs, ys) = & min2\ (\sharp xs)\ (\sharp ys) & & \\
> & & where & & \\
> & & min2\ x\ y & = & x,\ if\ x < y \\
> & & & = & y,\ otherwise
\end{array}
$$

- Proof by induction on both xs and ys:

  $case\ [\,],\ ys$:

$$
\begin{array}{ll}
 & \sharp zip([\,], ys) \\
\Rightarrow & \quad (zip.1) \\
 & \sharp([\,]) \\
\Rightarrow & \quad (\sharp.1) \\
 & 0
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & (min2\ 0\ n\ =\ 0\ for\ all\ n) \\
& min2\ 0\ (\sharp ys)
\end{array}
$$

case $x : xs,\ [\,]$:

$$
\begin{array}{ll}
& \sharp zip(x : xs, [\,]) \\
\Rightarrow & (zip.2) \\
& \sharp([\,]) \\
\Rightarrow & (\sharp.1) \\
& 0 \\
\Rightarrow & (min2\ n\ 0\ =\ 0\ for\ all\ n) \\
& min2\ \sharp(x : xs)\ 0
\end{array}
$$

case $x : xs,\ y : ys$:

$$
\begin{array}{ll}
& \sharp zip(x : xs, y : ys) \\
\Rightarrow & (zip.3) \\
& \sharp((x, y) : zip(xs, ys)) \\
\Rightarrow & (\sharp.2) \\
& 1 + \sharp zip(xs, ys) \\
\Rightarrow & (hyp) \\
& 1 + (min2\ \sharp xs\ \sharp ys) \\
\Rightarrow & (+\ distribution\ through\ min2) \\
& min2\ (1 + \sharp xs)\ (1 + \sharp ys) \\
\Rightarrow & (\sharp.2) \\
& min2\ \sharp(x : xs)\ \sharp(y : ys)
\end{array}
$$

Proved.

Note:

- Wrong recursive definition of *zip*:

$$
\begin{array}{rcl}
zip([\,], ys) & = & [\,] \\
zip(xs, [\,]) & = & [\,] \\
zip(x : xs, y : ys) & = & (x, y) : zip(xs, ys)
\end{array}
$$

This is *wrong*, because both (zip.1) and (zip.2) give $zip([\,],[\,])$ as each of xs and ys can be [ ], even though both consistently, result in [ ].
It is required that every possible value of an argument can bind to only one equation in the pattern matching definition.

- However, this works in Miranda as the equations are evaluated top down.

2.20 List: *take* & *drop*

2.20.1 *take*

take n xs is the list made by taking the first n elements of the list xs. (this will be the whole of xs if $n >= \sharp xs$)

$take :: num \rightarrow [*] \rightarrow [*]$

Recursive definition of *take*:

$$
\begin{array}{lllll}
> & take\ 0\ xs & = [\,] & || & (take.1) \\
> & take\ (n+1)\ [\,] & = [\,] & || & (take.2) \\
> & take\ (n+1)\ (x:xs) & = x : take\ n\ xs & || & (take.3)
\end{array}
$$

Notes:

- Exhaustive:
  - *take* 0 xs: Only (*take*.1) deals with this case, take none from any list.
  - *take* $n+1$ [ ]: Only (*take*.2) deals with this case, take 1 or more from [ ].
  - *take* $n+1$ $(x:xs)$: Only (*take*.3) deals with this case, take one or more from nonempty list.
- Disjoint:
  - Only one equation true for every possible pair of num and list.

2.20.2 *drop*

drop n xs is the list left on removing the first n elements of the list xs. This will be [ ] if $n >= \sharp xs$.

$drop :: num \rightarrow [*] \rightarrow [*]$

Recursive definition of *drop*:

$$
\begin{array}{lllll}
> & drop\ 0\ xs & = xs & || & (drop.1) \\
> & drop\ (n+1)\ [\,] & = [\,] & || & (drop.2) \\
> & drop\ (n+1)\ (x:xs) & = drop\ n\ xs & || & (drop.3)
\end{array}
$$

Note:

- Exhaustive:
  - *drop* 0 xs: Only (drop.1) deals with this case, drop none from any list.

- $drop\ n + 1\ [\,]$: Only (drop.2) deals with this case, drop one or more from $[\,]$.
- $drop\ n + 1\ xs$: Only (drop.3) deals with this case, drop one or more from a nonempty list.

- Disjoint:
  - Only one equation true for every possible pair of lists.

A Law involving *take* & *drop*:

- To show

$$take\ n\ xs\ +\!+\ drop\ n\ xs\ =\ xs$$

  for every natural number n and finite list xs.

- Proof by induction on n and xs:
  case $0,\ [\,]$:

$$\begin{array}{ll} & take\ 0\ [\,]\ +\!+\ drop\ 0\ [\,] \\ \Rightarrow & \quad (take.1,\ drop.1) \\ & [\,]\ +\!+\ xs \\ \Rightarrow & \quad (+\!+.1) \\ & xs \end{array}$$

  case $(n+1),\ [\,]$:

$$\begin{array}{ll} & take\ (n+1)\ [\,]\ +\!+\ drop\ (n+1)\ [\,] \\ \Rightarrow & \quad (take.2, drop.2) \\ & [\,]\ +\!+\ [\,] \\ \Rightarrow & \quad (+\!+.1) \\ & [\,] \end{array}$$

  case $(n+1),\ (x : xs)$:

$$\begin{array}{ll} & take\ (n+1)\ (x:xs)\ +\!+\ drop\ (n+1)\ (x:xs) \\ \Rightarrow & \quad (take.3,\ drop.3) \\ & (x : take\ n\ xs)\ +\!+\ (drop\ n\ xs) \\ \Rightarrow & \quad (+\!+.2) \\ & x : (take\ n\ xs\ +\!+\ drop\ n\ xs) \\ \Rightarrow & \quad (hyp) \\ & x : xs \end{array}$$

Proved.

2.21 List: *hd* & *tl*

2.21.1 *hd*

hd selects the first element of the list.
$hd :: [*] \to *$

Definition of *hd*:

$> \quad hd\ (x : xs) = x \qquad (hd.1)$

Note:

- *hd* [ ] undefined.

tl

tl selects the list without the head.
$> \quad tl :: [*] \to [*]$

Definition of *tl*:

$> \quad tl\ (x : xs) = xs \qquad (tl.1)$

Note:

- *tl* [ ] undefined.

A law involving *hd* & *tl*:

- To show

$$[hd\ xs]\ ++\ tl\ xs = xs$$

  for every non-empty finite list xs.

- Proof by case analysis on xs.
  case $(x : xs)$:

$$[hd\ (x : xs)]\ ++\ tl\ (x : xs)$$

$$
\begin{array}{ll}
\Rightarrow & (hd.1, tl.1) \\
& [x] \ ++ \ xs \\
\Rightarrow & (by definition of [x]) \\
& (x : [\,]) \ ++ \ xs \\
\Rightarrow & (++.2) \\
& x : ([\,] \ ++ \ xs) \\
\Rightarrow & (++.1) \\
& x : xs
\end{array}
$$

Proved.

2.22 Lists: *init* & *last*

2.22.1 *init*

init gives a list of all the elements of the list apart from the last.
$init :: [*] \rightarrow [*]$

Definition of *init*:

$$\begin{array}{lll} > \quad init\ [x] & = [\,] & || \ (init.1) \\ > \quad init\ (x : x' : xs) & = x : init(x' : xs) & || \ (init.2) \end{array}$$

Notes:

- *init* [ ] not defined.
- (*init*.1) defines it for one element list.
- (*init*.2) defines it for two or more elements list.

last:

last gives the last element of the list.
$last :: [*] \rightarrow *$

Definition of *last*:

$$\begin{array}{lll} > \quad last\ [x] & = x & || \ (last.1) \\ > \quad last\ (x : x' : xs) & = last(x' : xs) & || \ (last.2) \end{array}$$

Notes:

- *last* [ ] not defined.
- (*last*.1) defines it for one element list.
- (*last*.2) defines it for two or more elements list.

A law involving *init* & *last*:

- To show

$$init\ xs \quad = \quad take\ (\sharp xs - 1)\ xs$$

for any finite non-empty list xs.

- Proof by induction on xs :

  case $[x]$:

$$
\begin{array}{ll}
 & \mathit{init}\ [x] \\
\Rightarrow & \quad (\mathit{init}.1) \\
 & [\,] \\
\Rightarrow & \quad (\mathit{take}.1) \\
 & \mathit{take}\ 0\ [x] \\
\Rightarrow & \quad (\sharp.1) \\
 & \mathit{take}\ (\sharp[\,])\ [x] \\
\Rightarrow & \quad (\sharp.2) \\
 & \mathit{take}\ (\sharp(x : [\,])\ -\ 1)\ [x] \\
\Rightarrow & \quad (x : [\,] = [x]) \\
 & \mathit{take}\ (\sharp[x]\ -\ 1)\ [x]
\end{array}
$$

  case $(x : x' : xs)$:

$$
\begin{array}{ll}
\mathit{init}\ (x : x' : xs) & \\
 & \Rightarrow \quad (\mathit{init}.2) \\
 & \quad x : \mathit{init}(x' : xs) \\
 & \Rightarrow \quad (\mathit{hyp}) \\
 & \quad x : \mathit{take}\ (\sharp\ (x' : xs)\ -1\ )\ (x' : xs) \\
 & \Rightarrow \quad (\sharp\ .2) \\
 & \quad x : \mathit{take}\ (\sharp xs)\ (x' : xs) \\
 & \Rightarrow \quad (\mathit{take}.3) \\
 & \quad \mathit{take}\ (\sharp\ xs\ +\ 1)\ (x : x' : xs) \\
 & \Rightarrow \quad (\sharp\ .2) \\
 & \quad \mathit{take}\ (\sharp\ (x' : xs))\ (x : x' : xs) \\
 & \Rightarrow \quad (\sharp\ .2) \\
 & \quad \mathit{take}\ (\sharp(x : x' : xs)\ -\ 1)\ (x : x' : xs)
\end{array}
$$

  Proved.

- Wrong to have the following definition:

$$
\begin{array}{lllll}
> & \mathit{init}\ [x] & = [\,] & || & (\mathit{init}.1) \\
> & \mathit{init}\ (x : xs) & = x : \mathit{init}\ xs & || & (\mathit{init}.2)
\end{array}
$$

  because xs can be [ ], and there are two ways of reducing $\mathit{init}\ [x]$, [ ] by the first equation and $x : \mathit{init}\ [\,]$ by the second.

- Similarly for *last*, wrong to define:

$$
\begin{array}{lllll}
> & \mathit{last}\ [x] & = x & || & (\mathit{last}.1) \\
> & \mathit{last}\ (x : xs) & = \mathit{last}\ xs & || & (\mathit{last}.2)
\end{array}
$$

because xs can be [ ], and there are two ways of reducing $last\ [x]$, x by the first equation and $last\ [\,]$ by the second.

2.23 Lists: *map* & *filter*

2.23.1 *map*

map applies a function to each element of a list.
$map :: (* \rightarrow **) \rightarrow [*] \rightarrow [**]$

Recursive definition of *map*:

$$
\begin{array}{lllll}
> & map\ f\ [\,] & = [\,] & & || (map.1) \\
> & map\ f(x:xs) & = f\ x\ : map\ f\ xs & & || (map.2)
\end{array}
$$

2.23.2 *filter*

filter removes elements of a list that do not satisfy a predicate.
$filter :: (* \rightarrow bool) \rightarrow [*] \rightarrow [*]$

Recursive definition of *filter*:

$$
\begin{array}{lllll}
> & filter\ p\ [\,] & = [\,] & & || (filter.1) \\
> & filter\ p\ (x:xs) & = x : filter\ p\ xs,\ if\ p\ x & & || (filter.2) \\
> & & = filter\ p\ xs, \quad otherwise & & || (filter.3)
\end{array}
$$

Example using *map* and *filter*:

$$
\begin{array}{ll}
 & filter\ even\ (map\ (\hat{}2)\ [1,2,3,4]) \\
\Rightarrow & \quad (map.2) \\
 & filter\ even\ (1 : map\ (\hat{}2)\ [2,3,4]) \\
\Rightarrow & \quad (filter.3) \\
 & filter\ even\ (map\ (\hat{}2)[2,3,4]) \\
\Rightarrow & \quad (map.2) \\
 & filter\ even\ (4 : map\ (\hat{}2)\ [3,4]) \\
\Rightarrow & \quad (filter.2) \\
 & 4 : filter\ even\ (map\ (\hat{}2)\ [3,4]) \\
\Rightarrow & \quad (map.2) \\
 & 4 : filter\ even\ (9 : map\ (\hat{}2)\ [4]) \\
\Rightarrow & \quad (filter.3) \\
 & 4 : filter\ even\ (map\ (\hat{}2)\ [4]) \\
\Rightarrow & \quad (map.2)
\end{array}
$$

$$
\begin{array}{ll}
 & 4 : filter\ even\ (16 : map\ (\hat{}2)\ [\,])\\
\Rightarrow & \quad (filter.2)\\
 & 4 : 16 : filter\ even\ (map\ (\hat{}2)\ [\,])\\
\Rightarrow & \quad (map.1)\\
 & 4 : 16 : filter\ even\ [\,]\\
\Rightarrow & \quad (filter.1)\\
 & 4 : 16 : [\,]\\
\Rightarrow & \\
 & [4, 16]
\end{array}
$$

A law involving map **&** $filter$**:**

- To show

$$filter\ p\ (map\ f\ xs)\ =\ map\ f\ (filter\ (p.f)\ xs)$$

  for any function f, *total* predicate p, and finite list xs.

  A predicate p is total if for every x the value of p is always $True$ or $False$ and never undetermined.

- Proof by induction on xs :

  case $[\,]$:

$$
\begin{array}{ll}
 & filter\ p\ (map\ f\ [\,])\\
\Rightarrow & \quad (map.1)\\
 & filter\ p\ [\,]\\
\Rightarrow & \quad (filter.1)\\
 & [\,]\\
\Rightarrow & \quad (map.1)\\
 & map\ f\ [\,]\\
\Rightarrow & \quad (filter.1)\\
 & map\ f\ (filter\ (p.f)\ [\,])
\end{array}
$$

  case $(x : xs),\ (p(f\ x))\ =\ True$:

$$
\begin{array}{ll}
 & filter\ p(map\ f(x : xs))\\
\Rightarrow & \quad (map.2)\\
 & filter\ p(\ f\ x\ : map\ f\ xs)\\
\Rightarrow & \quad (filter.2)\\
 & f\ x\ :\ filter\ p(map\ f\ xs)\\
\Rightarrow & \quad (hyp.)
\end{array}
$$

$$
\begin{array}{ll}
 & f\ x\ :\ map\ f(filter\ (p.f)\ xs) \\
\Rightarrow & \quad (map.2) \\
 & map\ f\ (x :\ filter\ (p.f)\ xs) \\
\Rightarrow & \quad (filter.2) \\
 & map\ f\ (filter\ (p.f)\ (x : xs))
\end{array}
$$

$case\ (x : xs),\ (p(f\ x))\ =\ False$:

$$
\begin{array}{ll}
 & filter\ p(map\ f(x : xs)) \\
\Rightarrow & \quad (map.2) \\
 & filter\ p(\ f\ x\ : map\ f\ xs) \\
\Rightarrow & \quad (filter.3) \\
 & filter\ p(map\ f\ xs) \\
\Rightarrow & \quad (hyp.) \\
 & map\ f(filter\ (p.f)\ xs) \\
\Rightarrow & \quad (filter.3) \\
 & map\ f\ (filter\ (p.f)\ (x : xs))
\end{array}
$$

Because p is total considering the two possibilities $True$ and $False$ for the value of $(p(f(x))$ is sufficient.

2.24 Lists: $--$

2.24.1 List Difference, $(--)$

The value of $xs \;--\; ys$ is the list that results when, for each element y in ys, the first occurrence (if any) of y is removed from xs.
$-- :: [*] \to [*] \to [*]$

Recursive definition of $(--)$:

$$
\begin{array}{lll}
> & xs \;--\; [\,] = xs & \| (--.1) \\
> & xs \;--\; (y : ys) = remove\ xs\ y \;--\; ys & \| (--.2) \\
> & remove\ [\,]\ y = [\,] & \| (remove.1) \\
> & remove\ (x : xs)\ y = xs, \quad if\ x = y & \| (remove.2) \\
> & \qquad\qquad\quad = x : remove\ xs\ y,\ otherwise & \| (remove.3)
\end{array}
$$

2.25 Lists: *reverse* (A Reverse Function)

reverse is a function to reverse the elements of a list, with type:
$reverse :: [*] \rightarrow [*]$

Definition of *reverse*:

$$\begin{array}{lll} > \quad reverse\ [\,] & = [\,] & \|\ (reverse.1) \\ > \quad reverse\ (x : xs) & = reverse\ xs\ +\!+\ [x] & \|\ (reverse.2) \end{array}$$

A law for *reverse*:

- To show

$$reverse(reverse\ xs) \quad = \quad xs \qquad (hyp)$$

  for every finite list xs.

- Proof by induction on xs:
  case $[\,]$:

$$\begin{array}{ll} & reverse(reverse\ [\,]) \\ \Rightarrow & \quad (reverse.1) \\ & reverse\ [\,] \\ \Rightarrow & \quad (reverse.1) \end{array}$$

  case $(x : xs)$:

$$\begin{array}{ll} & reverse(reverse\ (x : xs)) \\ \Rightarrow & \quad (reverse.2) \\ & reverse(reverse\ xs\ +\!+\ [x]) \\ \Rightarrow & \quad (auxrev.1,\ \textit{an auxiliary result proved below}) \\ & x : reverse(reverse\ xs) \\ \Rightarrow & \quad (hyp) \\ & x : xs \end{array}$$

  Proved.

- The auxiliary result:

$$reverse(xs\ +\!+\ [x]) \quad = \quad x : reverse\ xs \qquad (auxrev.1)$$

  for every x and every finite list xs.

– Proof by induction on xs :
$case$ [ ] :

$$\begin{array}{ll} & reverse([\,] \;{+}{+}\; [x]) \\ \Rightarrow & \quad ({+}{+}.1) \\ & reverse\ [x] \\ \Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\ & reverse\ (x : [\,]) \\ \Rightarrow & \quad (reverse.2) \\ & reverse\ [\,] \;{+}{+}\; [x] \\ \Rightarrow & \quad (reverse.1) \\ & [\,] \;{+}{+}\; [x] \\ \Rightarrow & \quad ({+}{+}.1) \\ & x : [\,] \\ \Rightarrow & \quad (reverse.1) \\ & x : reverse[\,] \end{array}$$

$case\ y : ys$:

$$\begin{array}{ll} & reverse(y : ys \;{+}{+}\; [x]) \\ \Rightarrow & \quad ({+}{+}.2) \\ & reverse(y : (ys \;{+}{+}\; [x])) \\ \Rightarrow & \quad (reverse.2) \\ & reverse\ (ys \;{+}{+}\; [x]) \;{+}{+}\; [y] \\ \Rightarrow & \quad (hyp) \\ & (x : reverse\ ys) \;{+}{+}\; [y] \\ \Rightarrow & \quad ({+}{+}.2) \\ & x : (reverse\ ys \;{+}{+}\; [y]) \\ \Rightarrow & \quad (reverse.2) \\ & x : reverse(y : ys) \end{array}$$

Proved.

2.25.1 Example using $reverse$

$$\begin{array}{ll} & reverse['a','b','c'] \\ \Rightarrow & \quad (reverse.2) \\ & reverse['b','c'] \;{+}{+}\; ['a'] \\ \Rightarrow & \quad (reverse.2) \\ & (reverse['c'] \;{+}{+}\; ['b']) \;{+}{+}\; ['a'] \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & (reverse.2) \\
 & ((reverse[\,] \mathbin{+\!+} ['c']) \mathbin{+\!+} ['b']) \mathbin{+\!+} ['a'] \\
\Rightarrow & (reverse.1) \\
 & (([\,] \mathbin{+\!+} ['c']) \mathbin{+\!+} ['b']) \mathbin{+\!+} ['a'] \\
\Rightarrow & (+\!+.1) \\
 & (['c'] \mathbin{+\!+} ['b']) \mathbin{+\!+} ['a'] \\
\Rightarrow & (list\ pretty\ printing,\ no\ reduction) \\
 & ('c' : [\,] \mathbin{+\!+} ['b']) \mathbin{+\!+} ['a'] \\
\Rightarrow & (+\!+.2) \\
 & ('c' : ([\,] \mathbin{+\!+} ['b'])) \mathbin{+\!+} ['a'] \\
\Rightarrow & (+\!+.1) \\
 & 'c' : ['b'] \mathbin{+\!+} ['a'] \\
\Rightarrow & (+\!+.2) \\
 & 'c' : (['b'] \mathbin{+\!+} ['a']) \\
\Rightarrow & (list\ pretty\ printing,\ no\ reduction) \\
 & 'c' : ('b' : [\,] \mathbin{+\!+} ['a']) \\
\Rightarrow & (+\!+.2) \\
 & 'c' : ('b' : ([\,] \mathbin{+\!+} ['a'])) \\
\Rightarrow & (+\!+.1) \\
 & 'c' : ('b' : ['a']) \\
\Rightarrow & (list\ pretty\ printing,\ no\ reduction) \\
 & ['c',' b',' a']
\end{array}
$$

Note:

- number of elements in the list 3.
- 3 reductions by $(revrse.2)$;
- followed by 1 reduction by $(reverse.1)$;
- followed by 1 reduction by $(+\!+.1)$ for $[\,] \mathbin{+\!+} ['c']$;
- followed by 1 reduction by $(+\!+.2)$, followed by 1 reduction by $(+\!+.1)$ for $['c'] \mathbin{+\!+} ['b']$;
- followed by 1 reduction by $(+\!+.2)$, followed by 1 reduction by $(+\!+.2)$, followed by 1 reduction by $(+\!+.1)$ for $['c',' b'] \mathbin{+\!+} ['a']$.

2.25.2 order of function *reverse*

- Number of reduction steps using ++:

$$
\begin{array}{ll}
 & [x_1, x_2, x_3, .., x_{n-1}] \ ++ \ [x_n] \\
\Rightarrow & \quad (list: \ pretty \ printing, \ no \ reduction) \\
 & x_1 : [x_2, x_3, .., x_{n-1}] \ ++ \ [x_n] \\
\Rightarrow & \quad (++.2) \\
 & x_1 : ([x_2, x_3, .., x_{n-1}] \ ++ \ [x_n]) \\
\Rightarrow & \quad (list: \ pretty \ printing, \ no \ reduction) \\
 & x_1 : (x_2 : [x_3, .., x_{n-1}] \ ++ \ [x_n]) \\
\Rightarrow & \quad (++.2) \\
 & x_1 : (x_2 : ([x_3, .., x_{n-1}] \ ++ \ [x_n])) \\
 & ... \\
\Rightarrow & \quad (...) \\
 & x_1 : (x_2 : (x_3 : (...x_{n-1} : ([\,] \ ++ \ [x_n])...))) \\
\Rightarrow & \quad (++.1) \\
 & x_1 : (x_2 : (x_3 : (...x_{n-1} : [x_n]...))) \\
\Rightarrow & \quad (list: \ pretty \ printing, \ no \ reduction) \\
 & [x_1, x_2, x_3, ..., x_n]
\end{array}
$$

  $n-1$ $(++.2)$ reductions followed by 1 $(++.1)$ reduction

- The above definition of reverse amounts to putting each element of the list individually in reverse order first and then concatenating them:

$$
\begin{array}{ll}
 & [x_1, x_2, x_3, ..., x_n] \\
\Rightarrow & \quad (n \ (reverse.2) \ followed \ by \ 1 \ (reverse.1)) \\
 & [x_n] \ ++ \ [x_{n-1}] \ ++ \ [x_{n-2}] \ ++ \ ... \ ++ \ [x_1].
\end{array}
$$

- This is then concatenated to form $[x_n, x_{n-1}, x_{n-2}, ...x_2, x_1]$ by using $(++.1)$ and $(++.2)$:

$$
\begin{array}{ll}
 & [x_n] \ ++ \ [x_{n-1}] \ ++ \ [x_{n-2}] \ ++ \ [x_{n-3}] \ ++ \ ... \ ++ \ [x_1] \\
\Rightarrow & \quad (1 \ \times \ (++.2), 1 \ \times \ (++.1)) \\
 & [x_n, x_{n-1}] \ ++ \ [x_{n-2}] \ ++ \ [x_{n-3}]... \ ++ \ [x_1] \\
\Rightarrow & \quad (2 \ \times \ (++.2), 1 \ \times \ (++.1)) \\
 & [x_n, x_{n-1}, x_{n-2}] \ ++ \ [x_{n-3}]... \ ++ \ [x_1] \\
\Rightarrow & \quad (3 \ \times \ (++.2), 1 \ \times \ (++.1)) \\
 & [x_n, x_{n-1}, x_{n-2}, x_{n-3}] \ ++ \ ... \ ++ \ [x_1] \\
 & ...
\end{array}
$$

  This will mean a total of n reductions by $(++.1)$ and $(1+2+..n-1 = (n-1)*n/2)$ reductions using $(++.2)$.

- Thus
  - In general for a list of length n:
  - n reductions by ($reverse.2$), followed by 1 reduction by ($reverse.1$), followed by $n + (n-1) * n/2 = n * (n+1)/2$ reductions using $(++)$.
  - Total number of reduction steps $= n + 1 + n * (n+1)/2 = 1 + 3 * n/2 + n^2/2$. For large n this is proportional to n^2.
  - We say the function $reverse$ is of order n^2 and is denoted using "Big O" as $O(n^2)$.

2.26 Lists: *rev* (A Fast Reverse Function)

Definition of *rev*:

$$
\begin{array}{lllll}
> & rev\ xs & = & shunt\ [\,]\ xs & \|\ (rev.1) \\
> & shunt\ ys\ [\,] & = & ys & \|\ (rev.2) \\
> & shunt\ ys\ (x : xs) & = & shunt\ (x : ys)\ xs & \|\ (rev.3)
\end{array}
$$

2.26.1 Example using *rev*

$$
\begin{array}{ll}
 & rev\ ['a','b','c'] \\
\Rightarrow & \quad (rev.1) \\
 & shunt\ [\,]\ ['a','b','c'] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ [\,]\ ('a' : ['b','c']) \\
\Rightarrow & \quad (rev.3) \\
 & shunt\ ('a' : [\,])\ ['b','c'] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ ['a']\ ['b','c'] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ ['a']\ ('b' : ['c']) \\
\Rightarrow & \quad (rev.3) \\
 & shunt\ ('b' : ['a'])\ ['c'] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ ['b','a']\ ['c'] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ ['b','a']\ ('c' : [\,]) \\
\Rightarrow & \quad (rev.3) \\
 & shunt\ ('c' : ['b','a'])\ [\,] \\
\Rightarrow & \quad (list\ pretty\ printing,\ no\ reduction) \\
 & shunt\ ['c','b','a']\ [\,] \\
\Rightarrow & \quad (rev.2) \\
 & ['c','b','a']
\end{array}
$$

Notes:

- number of elements in the list 3.
- 1 reduction by (*rev*.1);
- followed by 3 reductions by (*rev*.3);

- followed by 1 reduction by (rev.2).

2.26.2 Order of function rev

- This defintition of reverse amounts to shunting out (or more intuitively,"pouring out") the elements of the given list to an empty list:
  - If the given list is empty, then its reverse is also an empty list. This involves 2 reduction steps: first the use of (rev.1) with $xs = [\,]$, and then the use of (rev.2) with $ys = [\,]$.
  - If the given list is non-empty, then the *shunt* function removes the current head of the second list and makes it the head of the first list, using (rev.3). This 'shunting' operation is a single reduction step. This process is repeated until all the elements of the given list are 'shunted across', i.e., removed from the second list and put at the head of the first list. Finally, (rev.2) is used to give the reversed list. So for a list of length n, one reduction by (rev.1) to invoke *shunt*, n reductions by (rev.3) to 'empty' the given list & to 'build' the reversed list, and 1 reduction by (rev.2) to ouput the reversed list.
  - So altogether, $n + 2$ reduction steps are required to reverse a list of size n, $O(n)$.
- Note:
  - The function rev uses the function *shunt*, which does reversing and concatenation in one reduction step, giving it a $O(n)$ time complexity.
  - The function *reverse* does all the reversing first, and then concatenates one by one. The concatenation function (++) has a $O(n^2)$ behaviour. This makes *reverse* have a time complexity of $O(n^2)$.

2.27 $fastfib$ (A Fast Fibonacci Function)

2.27.1 Order of fib

- Fibonacci function, fib, as previously defined is:

$$\begin{array}{lllll} > & fib\ 0 & = 0 & & || \ (fib.1) \\ > & fib\ 1 & = 1 & & || \ (fib.2) \\ > & fib\ (n+2) & = fib\ n\ +\ fib\ (n+1) & & || \ (fib.3) \end{array}$$

- Number of reduction steps to evaluate $fib\ 0$ is 1, e.g.,

$$\begin{array}{ll} & fib\ 0 \\ \Rightarrow & \ (fib.1) \\ & 0 \end{array}$$

  We'll denote it as $T(0)$.

- Similarly, number of reduction steps to evaluate $fib\ 1$ is 1, e.g.,

$$\begin{array}{ll} & fib\ 1 \\ \Rightarrow & \ (fib.2) \\ & 0 \end{array}$$

  We'll denote it as $T(1)$.

- Also, denote, the number of reductions to evaluate $fib\ n$, $fib\ (n+1)$ and $fib\ (n+2)$ by $T(n)$, $T(n+1)$ and $T(n+2)$, respectively.

  Since, only 1 reduction step is needed to evaluate $fib\ (n+2)$, once $fib\ n$ and $fib\ (n+1)$ are known, i.e.,

$$\begin{array}{ll} & fib\ (n+2) \\ \Rightarrow & \ (fib.3) \\ & fib\ n\ +\ fib\ (n+1) \end{array}$$

  we can write:

$$T(n+2) \ = \ T(n+1)\ +\ T(n)\ +\ 1$$

  – This is a second order non-homogeneous recurrence relation, for which solutions can be found in a standard way. Its solution is:

$$T(n) \ = \ c_1 * ((1 - sqrt\ 5)/2)^n + c_2 * ((1 + sqrt\ 5)/2)^n - 1$$

  where the coefficients c_1 and c_2 are found using the initial values $T(0) = 1$ and $T(1) = 1$, to be

$$\begin{array}{lll} c_1 & = & (1\ -\ sqrt\ 5)/(sqrt\ 5) \\ c_2 & = & (1\ +\ sqrt\ 5)/(sqrt\ 5) \end{array}$$

  On substituting and simplifying the above solution becomes:

$$\begin{array}{lll} T(n) & = & -(2/sqrt\ 5) * ((1 - sqrt\ 5)/2)^{(n+1)} \\ & & +(2/sqrt\ 5) * ((1 + sqrt\ 5)/2)^{(n+1)} - 1 \end{array}$$

Using the value of

$$\begin{aligned} fib(n+1) &= ((1+sqrt\ 5)/2)^{(n+1)}/(sqrt\ 5) \\ &\quad -((1-sqrt\ 5)/2)^{(n+1)})/(sqrt\ 5) \end{aligned}$$

we can write

$$T(n) = 2 * fib(n+1) - 1$$

- This means that the number of reductions using the above definition of fib increases very rapidly as n increases. Since

$$(1+sqrt\ 5)/2 = 1.6180 \quad (mod > 1)$$

and

$$(1-sqrt\ 5)/2 = -0.6180 \quad (mod < 1)$$

for large n, $T(n)$ increases as $(1.6)^n$, i.e., exponentially with n. We say $T(n)$ is non-polynomially bound (NP). This is very inefficient.

- In the above we used the general result from the Mathematical Theory of Recurrence relations, without proof. However, it is possible to prove by induction the above result for $T(n)$, i.e.,

$$T(n) = 2 * fib(n+1) - 1$$

as follows:
$case\ n = 0:$

$$\begin{aligned} T(0) &= 2 * fib(1) - 1 \\ &= 2*1 - 1 \\ &= 1 \end{aligned}$$

which we have proved before.
$case\ n = 1:$

$$\begin{aligned} T(1) &= 2 * fib(2) - 1 \\ &= 2 * 1 - 1 \\ &= 1 \end{aligned}$$

which also we proved earlier.
We shall now make the inductive hypothesis that the solution holds for n and $n+1$,

$$T(n) = 2 * fib(n+1) - 1$$

and

$$T(n+1) = 2 * fib(n+2) - 1$$

We now use the recurrence relation for $T(n+2)$ above:

$$\begin{aligned} T(n+2) &= T(n+1) + T(n) + 1 \\ &= 2 * fib(n+2) - 1 + 2 * fib(n+1) - 1 + 1 \\ &= 2(fib(n+2) + fib(n+1)) - 1 \\ &= 2 * fib(n+3) - 1 \end{aligned}$$

Proved.

2.27.2 Definition of $fastfib$

We now define a faster version of the fibonacci function:

$$
\begin{array}{lll}
> \quad fastfib\ n & = fst(twofib\ n) & || (fastfib.1) \\
> \quad twofib\ 0 & = (0,1) & || (fastfib.2) \\
> \quad twofib\ (n+1) & = (b, a+b) & || (fastfib.3) \\
> & \quad where & \\
> & \quad (a,b) = twofibn &
\end{array}
$$

Equivalence between fib and $fastfib$:

- Let us first convince ourselves that this indeed gives the same number sequence as the function fib.

  In fact, $> \quad twofibn \quad = (fib\ n,\ fib(n+1)) \quad || (hyp)$

- This can be proved by induction as follows:
  $case\ n = 0$:

$$
\begin{array}{rcl}
twofib\ 0 & = & (fib\ 0,\ fib\ 1) \\
 & = & (0,\ 1)
\end{array}
$$

  which agrees with the definition $(fastfib.2)$.

  $case\ n = n+1$:

$$
\begin{array}{ll}
 & twofib(n+1) \\
\Rightarrow & \quad (twofib.3) \\
 & (b, a+b)\ where\ (a,b) = twofib\ n \\
\Rightarrow & \quad (hyp) \\
 & (b, a+b)\ where\ (a,b) = (fib\ n,\ fib(n+1)) \\
\Rightarrow & \\
 & (fib(n+1), fib\ n\ +\ fib(n+1)) \\
\Rightarrow & \quad (fib.3) \\
 & (fib\ (n+1),\ fib\ (n+2))
\end{array}
$$

  Proved.

Order of $fastfib$:

This definition is much faster:

$$
\begin{array}{ll}
 & fastfib\ n \\
\Rightarrow & \quad (twofib.1) \\
 & fst\ (twofib\ n)
\end{array}
$$

This is 1 reduction step.

$twofib\ n$ is reduced succexxively to $twofib\ (n-1)$, $twofib\ (n-2)$,... down to $twofib(0)$ in n reduction steps using $(twofib.3)$

$twofib(0)$ itself is reduced to $(0,1)$ using $(twofib.2)$.

Hence the total number of reduction steps for $fastfib\ n$ is $(n+2)$ which is $O(n)$. This is more efficient, i.e., the number of reductions required is much less and so the result is obtained much faster.

2.28 Efficiency

Efficiency of programs:

- How fast programs run, how expensive are they in memory?
- Modelling efficiency of programs (environmental factors)
- Complexity - comparing performance (time and space)

2.28.1 Example:Sum of Natural Numbers

$$
\begin{array}{lllll}
> & sumto\ 0 & = 0 & || & (sumto.1) \\
> & sumto\ n+1 & = (n+1)\ +\ sumto\ n & || & (sumto.2)
\end{array}
$$

Order:

$$
\begin{array}{ll}
 & sumto\ 0 \\
\Rightarrow & \quad (sumto.1) \\
 & 0
\end{array}
$$

Hence,

$$T(0) \quad = \quad 1$$

$$
\begin{array}{ll}
 & sumto\ (n+1) \\
\Rightarrow & \quad (sumto.2) \\
 & (n+1)\ +\ sumto\ n
\end{array}
$$

Hence,

$$T(n+1) \quad = \quad T(n)\ +\ 1$$

This is a first order recurrence relation, and has the solution:

$$T(n+1) \quad = \quad T(0)\ +\ n+1$$

This may be arrived at by backward recursion from $T(n)$ to $T(n-1)$, then to $T(n-2)$,... and eventually to $T(0)$. Alternatively, use inductive proof.
Hence,

$$T(n+1) = 1\ +\ n+1$$

i.e., $T(n)$ is $O(n)$.

2.28.2 Example:Factorial n

$$
\begin{array}{lll}
> & fact\ 0 & = 1 \qquad\qquad || (fact.1) \\
> & fact\ n+1 & = (n+1) * fact\ n \qquad || (fact.2)
\end{array}
$$

$$
\begin{aligned}
T(0) &= 1 \\
T(n+1) &= T(n) + 1
\end{aligned}
$$

This is same as for *sumto* above giving $T(n)$ to be $O(n)$.

2.28.3 Example: Binary string with no two 1's consecutive

$$
\begin{array}{lll}
> & count\ 0 & = 0 \qquad\qquad\qquad\qquad\qquad || (count.1) \\
> & count\ 1 & = 2 \qquad\qquad\qquad\qquad\qquad || (count.2) \\
> & count\ 2 & = 3 \qquad\qquad\qquad\qquad\qquad || (count.3) \\
> & count(n+3) & = count(n+1) + count(n+2) \qquad || (count.4)
\end{array}
$$

$$
\begin{aligned}
T(0) &= 1 \\
T(1) &= 1 \\
T(2) &= 1 \\
T(n+3) &= T(n+1) + T(n+2) + 1
\end{aligned}
$$

This is a non-homogeneous second order recurrence relation. Comparing it with the fib case, we find that it has the solution,

$$
T(n+1) = 2 * fib(n+1) - 1
$$

(and, $T(0) = 1$)

This means $T(n+1)$ has the form $(1.6^{(n+1)})$, i.e., $T(n)$ is $O(a^n)$. This means an exponentially rising number of reduction as n increases.

2.28.4 Example: Towers of Hanoi

If $T(n)$ is the number of steps required to move n discs from peg A to peg B, this is same as moving $n-1$ discs from peg A to peg C, $T(n-1)$, moving the last disc from A to B, 1 move and moving the $n-1$ discs from C to B, $T(n-1)$:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2 * T(n-1) + 1
\end{aligned}
$$

This has the solution,

$$
\begin{aligned}
T(n) &= 2^{(n-1)} * T(1) + 2^{(n-2)} + 2^{(n-3)} + ... + 1 \\
&= 2^{(n-1)} + 2^{(n-2)} + 2^{(n-3)} + ... + 1 \\
&= 2^n - 1
\end{aligned}
$$

(Check:

$$\begin{aligned} T(n) & \\ &= 2 * T(n-1) + 1 \\ &= 2(2 * T(n-2) + 1) + 1 \\ &= 2(2 * (2 * T(n-3) + 1) + 1) + 1 \\ &= ... \\ &= 1 + 2 + 2^2 + 2^3 + ... + 2^(n-1) * T(1) \end{aligned}$$

A more rigorous proof is by induction.)

Hence, the recursive algorithm for solving the Tower of Hanoi problem is of $O(a^n)$. This means that the number of steps increases exponentially with n, the number of discs - very inefficient:

1 disc requires 1 move
2 discs requires 3 moves
3 discs requires 7 moves
4 discs requires 15 moves
.....
10 discs requires 1023 moves!
.....

2.28.5 Example: Travelling Salesman

Shortest route from T0 to T1,T2,..,Tn (no repeats).

$T(n)$: number of routes from T0 to n other towns T1,T2,..,Tn,

$T(n-1)$: number of routes from T0 to $n-1$ other towns T1,T2,..,Tn-1

For each of the $T(n-1)$ routes there are n possible routes that can be formed with Tn.

Hence,

$$T(n) = n * T(n-1)$$

e.g.,

$$\begin{aligned} T(1) &= 1\ (T0T1) \\ T(2) &= 2\ (T0T1T2, T0T2T1) \\ T(3) &= 6\ (T0T1T2T3, T0T1T3T2, T0T3T1T2, T0T2T1T3, T0T2T3T1, T0T3T2T1) \end{aligned}$$

etc.

Hence,

$$\begin{aligned} T(n) &= 1 * 2 * 3 * 4.. * n \\ &= n! \end{aligned}$$

Again proof by induction.

This means that the number of possible routes in the Travelling Salesman problem increases as the factorial function, which grows very rapidly as the number of towns to be visited increases.

2.28.6 Need for Efficency Measures:

- Criteria for comparing algorithms:
  - Execution time
  - Memory requirement
- Complexity functions:
  - Comparison of different algorithms for the same task
  - Behaviour of complexity functions with change in the size of the task.

2.29 Asymptotic Behaviour

Example:

1. *reverse xs*: For a list xs of size n, number of steps needed to reduce *reverse xs* (i.e., to evaluate the expression *reverse xs*) is proportional to n^2.
2. *rev xs*: Number of steps needed to reduce *rev xs* is proportional to n.

2.29.1 Notation:T & O

$T_f(x)$:

1. Number of reduction steps to compute $f\ x$, i.e., the number of steps required to reduce $f\ x$ to canonical (normal) form.
2. It gives a measure of the time required to perform the computation.
3. The actual number of reduction steps depends on the model of computation used. We shall use Outermost Graph Reduction.

$O(\ )$:

1. 'O' stands for order of at most.
2. Examples: $T_{reverse}(xs) = O(n^2)$, and $T_{rev}(xs) = O(n)$ where n is the length of xs.
3. More precise meaning of O:
   If $g(n)$ is some function of n, then whenever we write:

   $$g(n) = O(h(n))$$

   this means that there exists some constant M such that

   $$|g(n)| \leq M|h(n)|$$

   for every positive n.
4. Does not define $g(n)$ precisely, but does say that $g(n)$ is bounded by a function that is proportional to $h(n)$.

   Here $|g(n)|$ is the absolute value of $g(n)$; normally $g(n)$ is positive as it represents the time resource used by the program with input of size n.

Example:

$$g(n) = a_0 + a_1 * n + a_2 * n^2 + ... + a_m * n^m$$

To show

$$g(n) \quad = \quad O(n^m)$$

A suitable M for this problem is

$$M \quad = \quad |a_0| + |a_1| + .. + |a_m|$$

For,

$$g(n) \quad = \quad (a_0/n^m + a_1/n^{(m-1)} + .. + a_m)n^m, for\ n\ >\ 0$$

$$\begin{aligned} |g(n)| \quad &\leq \quad (|a_0/n^m| + |a_1/n^{(m-1)}| + .. + |a_m|) * n^m \\ &\leq \quad (|a_0| + |a_1| + .. + |a_m|) * n^m \\ &\leq \quad M * n^m \\ &\quad\ where\ M = |a_0| + |a_1| + ... + |a_m| \\ &\quad\ and\ n > 0. \end{aligned}$$

Note:

if

$$f(n) \quad = \quad O(h(n))$$

&

$$g(n) \quad = \quad O(h(n)$$

it does NOT follow that $f(n)\ =\ g(n)$.

2.29.2 Example: *reverse* & *rev*

Thus there exist constants M_1 and M_2 such that for every list of size $n\ >\ 0$,

$$T_{reverse}(xs) \quad = \quad O(n^2)$$

$\Rightarrow$ no. of steps to compute $reverse\ xs\ \leq\ M_1 * n^2$ for $n\ >\ 0$
&

$$T_{rev}(xs) \quad = \quad O(n)$$

$\Rightarrow$ no. of steps to compute $rev\ xs \leq M_2 * n$ for $n\ >\ 0$

The relative sizes of M_1 and M_2 are unimportant, and for large n, the second program is the faster one.
For $M_1 = 2$, $M_2 = 20$

	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$	$n=9$	$n=10$	$n=11$	$n=12$	$n=13$
$M_1 * n^2$	2	8	18	32	50	72	98	128	162	200	242	288	338
$M_2 * n$	20	40	60	80	100	120	140	160	180	200	220	240	260

For longer lists the second program is better.

2.29.3 Asymptotic analysis

Order of magnitude:

1. *reverse xs*: $O(n^2)$, quadratic order of magnitude
   rev xs: $O(n)$, linear order of magnitude
2. Program with smaller order of magnitude runs faster under all implementations for sufficiently large inputs.
3. Asymptotic analysis yields implementation independent information.
4. More detailed information needed to compare two programs with the same order of magnitude.
5. For small data (n):
   Size of the constant of proportionality may be important. Actual number of reduction steps, and timing analysis in particular implementations may be required.

2.30 Models of Reduction

2.30.1 Meaning of number of reduction steps

Example: pyth

Definition:

```
>   pyth x y  = sqr x + sqr y     || (pyth.1)
>   sqr x     = x * x             || (sqr.1)
```

Evaluation of pyth 3 4:

$$
\begin{array}{ll}
 & pyth\ 3\ 4 \\
\Rightarrow & \quad (pyth.1) \\
 & sqr\ 3\ +\ sqr\ 4 \\
\Rightarrow & \quad (sqr.1) \\
 & (3\ *\ 3)\ +\ sqr\ 4 \\
\Rightarrow & \quad (*) \\
 & 9\ +\ sqr\ 4 \\
\Rightarrow & \quad (sqr) \\
 & 9\ +\ (4\ *\ 4) \\
\Rightarrow & \quad (*) \\
 & 9\ +\ 16 \\
\Rightarrow & \quad (+) \\
 & 25
\end{array}
$$

6 reduction steps.

Notes on reduction:

- Each reduction step replaces a subterm by an equivalent term.
- A subterm which can be reduced is referred to as a *redex* (reducible expression).
- In each reduction step a redex on the left side (e.g., $pyth\ 3\ 4$) is replaced by the right side (e.g., $sqr\ 3\ +\ sqr\ 4$).
- The right hand side may itself be a redex (like $sqr\ 3\ +\ sqr\ 4$, $3 * 3$) or a primitive value (like 9, 25).
- When no redexes are left, the reduction is complete.
- The number of reduction steps depends on the order in which the redexes are reduced.

Example:

Using the definition of fst:

$> \quad fst(x,y) \quad = x \qquad\qquad || \ (fst.1)$

Evaluation of $fst(sqr\ 4, sqr\ 2)$:

One possible reduction:

$$
\begin{array}{ll}
 & fst\ (sqr\ 4,\ sqr\ 2) \\
\Rightarrow & \quad (fst.1) \\
 & sqr\ 4 \\
\Rightarrow & \quad (sqr.1) \\
 & 4 * 4 \\
\Rightarrow & \quad (*) \\
 & 16
\end{array}
$$

3 reduction steps.

Another possible reduction:

$$
\begin{array}{ll}
 & fst\ (sqr\ 4, sqr\ 2) \\
\Rightarrow & \quad (sqr.1) \\
 & fst\ (4 * 4, sqr\ 2) \\
\Rightarrow & \quad (*) \\
 & fst\ (16, sqr\ 2) \\
\Rightarrow & \quad (sqr.1) \\
 & fst\ (16, 2 * 2) \\
\Rightarrow & \quad (*) \\
 & fst(16, 4) \\
\Rightarrow & \quad (fst.1) \\
 & 16
\end{array}
$$

5 reduction steps.

2.30.2 Reduction Policies

Two reduction policies:

1. Outermost reduction:
   - Pick the redex that is NOT CONTAINED IN any other redex (fst).
   - Reduce it (fst) using the equation for it (results in sqr).
   - Pick the next redex NOT CONTAINED IN any other redex (sqr).
   - Reduce it (sqr) using the equation for it (results in $*$).

- Pick the next redex NOT CONTAINED IN any other redex ($*$).
- Reduce it ($*$) using the equation for it (results in primitive, 16).

2. Innermost reduction:

   - Pick the redex that does NOT CONTAIN any other redex (sqr, either of the two).
   - Reduce it (sqr) using the equation for it (results in $*$).
   - Pick the next redex that does NOT CONTAIN any other redex ($*$ or sqr, either will do, choose $*$, say).
   - Reduce it (whichever of $*$ or sqr was picked) using the equation for it (results in primitive, 16, if $*$ chosen).
   - Repeat (i.e., reduce) for the other redex that was not chosen first (sqr, results in $*$).
   - Now reduce the redex that does NOT CONTAIN any other redex using the equation for it($*$, resulting in 4).
   - Finally, reduce the resulting redex that does NOT CONTAIN any other redex (results in the primitive, 16).

2.31 Termination

Some reduction orders may fail to terminate.

Example:

$$
\begin{array}{lll}
> & answer & = fst(42, loop) \quad || \ (answer.1) \\
> & loop & = tl\ loop \quad || \ (loop.1)
\end{array}
$$

fst and tl are as defined before.

2.31.1 Innermost Reduction

$$
\begin{array}{ll}
 & answer \\
\Rightarrow & \quad (answer.1) \\
 & fst(42, loop) \\
\Rightarrow & \quad (loop.1) \\
 & fst(42, tl\ loop) \\
\Rightarrow & \quad (loop.1) \\
 & fst(42, tl\ (tl\ loop)) \\
\Rightarrow & \quad ....
\end{array}
$$

Does NOT terminate.

2.31.2 Outermost Reduction

$$
\begin{array}{ll}
 & answer \\
\Rightarrow & \quad (answer.1) \\
 & fst(42, loop) \\
\Rightarrow & \quad (fst.1) \\
 & 42
\end{array}
$$

2 reductions.

Notes:

- When both the reduction methods terminate, they give the same value (reduce to the same result).
- Whenever there is a reduction order which terminates, then the outermost reduction terminates.

2.31.3 Outer vs Inner Reduction

- Outermost reduction is usually referred to as Normal Order Reduction. This is because it results in a normal form (a canonical nonreducible value) whenever such a normal form exists.
- Outermost reduction is also sometimes referred to as Lazy evaluation, because it does not reduce a redex unless it is essential for evaluating the result.
- Inner reduction is usually referred to as Applicative order reduction, and sometimes as Eager evaluation.

2.31.4 Strict vs Non-strict Functions

- A strict function is undefined whenever its argument is undefined. For example:
  - Multiplication is strict in its first and second argument, since

    $$\bot * x \quad = \quad \bot$$

    and

    $$x * \bot \quad = \quad \bot$$

  - The tuple constructing function is not strict since

    $$(\bot, x) \quad \neq \quad \bot$$

    and

    $$(x, \bot) \quad \neq \quad \bot$$

- Outermost reduction is essential for evaluating non-strict functions.
- Unlike Miranda some functional languages allow only strict functions (SML). For such languages, innermost and outermost reductions are equivalent.

2.32 Graph Reduction

- Outermost reduction seems to require fewer (at most same number of) reduction steps than innermost reduction.
- On efficiency grounds outermost reduction seems to be preferable to innermost reduction.
- However, for the outermost and innermost reductions as defined above, it is possible to have examples where innermost reduction takes fewer reduction steps.

2.32.1 Example: $sqr(4+2)$

More outermost reduction steps than innermost reduction steps.

Innermost reduction:

$$
\begin{array}{ll}
 & sqr(4+2) \\
\Rightarrow & \quad (+) \\
 & sqr\ 6 \\
\Rightarrow & \quad (sqr.1) \\
 & 6*6 \\
\Rightarrow & \quad (*) \\
 & 36
\end{array}
$$

3 reduction steps.

Outermost reduction:

$$
\begin{array}{ll}
 & sqr\ (4+2) \\
\Rightarrow & \quad (sqr.1) \\
 & (4+2)*(4+2) \\
\Rightarrow & \quad (+) \\
 & 6*(4+2) \\
\Rightarrow & \quad (+) \\
 & 6*6 \\
\Rightarrow & \quad (*) \\
 & 36
\end{array}
$$

4 reduction steps.

Note: Time Complexity

- Reason obvious – the reduction of $4 + 2$ to 6 is duplicated.
- If all such duplicate reductions were to be counted only once, then we can ensure that outermost reduction never takes more reduction steps than the innermost reduction steps.
- Graph Reduction achieves this. Shared sub-terms are represented by graphs. Sharing subterms in a graph is achieved using where clause in the language implementation.
- With graph reduction, outermost reduction never requires more reduction steps than innermost reductions.
- Since, number of reduction steps gives a measure of time efficiency, outermost graph reduction is the fastest.

2.32.2 Space Complexity

- Size of a term or a graph is measured by the total number of arguments in it.
- Size of a graph gives Space efficiency measure, the size of the largest graph during reduction to the normal form being used for it.

Example:$sqr\ 3\ +\ sqr\ 4$

$sqr\ 3\ +\ sqr\ 4$
Each of the sqr take one argument, and $+$ takes two arguments.
So, this term has size 4.

Example:$sqr\ (4 + 3)$

$sqr\ (4 + 3) = (4 + 3) * (4 + 3)$
$+$ takes two argument and $*$ takes two argument. The two $+$'s are counted only once (graph reduction).
So, this graph has size 4.

2.33 Reduction & Pattern matching

2.33.1 Example:$zip(map\ sqr\ [\ ], loop)$

Using zip,

$> \quad zip([\,], ys) \quad = [\,] \quad || \ (zip.1)$

$> \quad zip(x : xs, [\,]) \quad = [\,] \quad || \ (zip.2)$

$> \quad zip(x : xs, y : ys) \quad = (x, y) : zip(xs, ys) \quad || \ (zip.3)$

map

$> \quad map\ f\ [\,] \quad = [\,] \quad || \ (map.1)$

$> \quad map\ f\ (x : xs) \quad = f\ x : map\ f\ xs \quad || \ (map.2)$

and $loop$

$> \quad loop \quad = tl\ loop \quad || \ (loop.1)$

$$
\begin{array}{ll}
 & zip(map\ sqr\ [\,], loop) \\
\Rightarrow & \quad (map.1) \\
 & zip([\,], loop) \\
\Rightarrow & \quad (zip.1) \\
 & [\,]
\end{array}
$$

This is outermost reduction. In order to reduce $zip(map\ sqr\ [\ ]$ had to be reduced, i.e, $map\ sqr\ [\ ]$ appears inside no other redex.
Alternatively, $loop$ which also appears no other redex could have been reduced:

$$
\begin{array}{ll}
 & zip(map\ sqr\ [\,], loop) \\
\Rightarrow & \quad (loop.1) \\
 & zip(map\ sqr\ [\,], tl\ loop) \\
\Rightarrow & \quad (loop.1) \\
 & zip(map\ sqr\ [\,],\ tl\ (tl\ loop) \\
 & ..
\end{array}
$$

This is also outermost reduction, since each $tl\ loop$ appears inside no other redex.
To decide which sub-term is to be reduced in pattern matching definition,

1. Check whether the result of reducing the term will be used by the pattern match. If so reduce it.
2. After its reduction, check whther the second term reduced is needed by the pattern match. If so reduce it.

In the above example, zip requires one of the terms to be in a form $x : xs$ or $[\,]$. As the second sub-term loop does not lead to either, the first sub-term was reduced.
Consider $zip(1 : map\ sqr\ [2], map\ sqr\ [4, 5, 6])$

The first sub-term is in the form $x : xs$ (as required by zip). In order for zip to work the second term should be in the form [ ] or $y : ys$. Hence, it needs to be reduced:

$$
\begin{array}{ll}
 & zip(1 : map\ sqr\ [2], map\ sqr\ [4,5,6]) \\
\Rightarrow & \quad (map.2) \\
 & zip(1 : map\ sqr\ [2], 16 : map\ sqr[5,6]) \\
\Rightarrow & \quad (zip.3) \\
 & (1,16) : zip(map\ sqr\ [2], map\ sqr[5,6]) \\
\Rightarrow & \quad (map.2) \\
 & (1,16) : zip(4 : [\,], map\ sqr[5,6]) \quad (needed\ for\ further\ zip) \\
\Rightarrow & \quad (map.2) \\
 & (1,16) : zip(4 : [\,], 25 : map\ sqr[6]) \\
\Rightarrow & \quad (zip.3) \\
 & (1,16) : (4,25) : zip([\,], map\ sqr\ [6]) \\
\Rightarrow & \quad (zip.1) \\
 & (1,16) : (4,25) : [\,]
\end{array}
$$

The second sub-term $map\ sqr\ [6]$ does not need to be reduced for the pattern match needed for zip's reduction.

2.34 Models of Implementation

Use outermost graph reduction.
If the reduction steps are:

$$
\begin{aligned}
& e_0 \\
\Rightarrow & \\
& e_1 \\
\Rightarrow & \\
& e_2 \\
\Rightarrow & \\
& .. \\
\Rightarrow & \\
& e_n
\end{aligned}
$$

where e_n is the normal form od eo.
Time taken is proportional to the number of reduction steps, n.
Space required is the size of the largest graph (maximum of sizes of e_0 to e_n).

2.35 End Of Chapter Exercises

1. Define a function which, using *distt* or *dist* from the lecture notes, returns triangle area, given the 3 vertices as 2-tuples, or pairs. You need to know that,

   if a, b, and c are the lengths of the 3 sides, then
   using $s = (a+b+c)/2$, we have
   $squared-area = s*(s-a)*(s-b)*(s-c)$

2. Define and test a function *age* for computing a person's age in years on a given date, given the date of birth. Two arguments: the first contains the name and date of birth, the second the given date. Example evaluations of this function:

   $age\ (\text{"}Joe\text{"}, (15,3,1965))\ (2,4,1992)$ $\quad answer:\ 27$
   $age\ joe\ today$ $\quad answer:\ 27$

   What is the type of *age*? Enter your type expression into your script to see if Miranda agrees. The second example above assumes that *joe* and *today* are constants that have been defined in the script, e.g. by:

   $$\begin{aligned} joe &= (\text{"}Joe\text{"}, (15,3,1965)) \\ today &= (2,4,1992) \end{aligned}$$

   Notice how the constant looks like a *nullary function*, i.e. a function with no arguments.
   Further development (jumping ahead by making use of the show function and list concatenation):

   - Alter your function to yield a more user-friendly string value such as "Joe is 27", rather than a basic num value such as 27.
   - USING THE ONLINE MANUAL, and B&W 2.3.2, use library string formatting functions, format your output in tabular fashion, e.g.

Born on	on date	Joe is
15/3/1965	2/4/1992	27

3. The two solutions (roots) of the quadratic equation are obtained as follows:

   $$\begin{aligned} a*x^2 + b*x + c &= 0 \\ solutions &= (-b \pm sqrt(b^2 - 4a*c))/(2*a) \end{aligned}$$

   Define and test a Miranda function which accepts the three coefficients a, b , c as arguments, and yields as result a pair (2- tuple) whose components are the two roots. Make sure that your function copes suitably with the case where two real roots do not exist.
   Optional extra: Can you make your solution more user friendly, e.g. to output result as a pair if two roots exist, and a suitable message if not ? You will probably need to use the *error* built-in function.

4. The following series tends in the limit to $pi/4$:
   $1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - .. + .. - ...$
   Define and test a function which, given an argument n, will sum n terms of the series and multiply by 4, to yield an approximation for pi.

Optional Extra: Investigate the accuracy and time efficiency of your function, by comparing it with Miranda's built-in *pi* constant, and by giving the Miranda command /count (which will track various execution statistics).

5. (OPTIONAL brain-teaser:) The following function definition, to yield the n'th Fibonacci number, is terribly inefficient. Can you devise a more efficient definition? (Compare the relative efficiency using /count).

$$\begin{aligned} fib\ 0 &= 0 \\ fib\ 1 &= 1 \\ fib\ n &= fib\ (n-1)\ +\ fib\ (n-2) \end{aligned}$$

6. Give the types for the following functions defined before (work them out *before* using Miranda to confirm!):

   mult, *multt*, *add*, *addt*, *succ*, *pred*

7. Give types for the following function definitions:

$$\begin{aligned} one\ x &= 1 \\ applytwice\ f\ x &= f\ (f\ x) \\ condapply\ p\ f\ g\ x &= f\ x,\ if\ p\ x \\ &= g\ x,\ otherwise \end{aligned}$$

8. ([BW88]B&W 1.4.2) Give examples of functions with the following types:

$$\begin{aligned} (num &\to num) \to num \\ num &\to (num \to num) \\ (num &\to num) \to (num \to num) \end{aligned}$$

   Which brackets are redundant ?

9. ([BW88]B&W 2.5.1) Define versions of the function ($\wedge$) and ($\vee$) using patterns for the second argument. Define versions which use patterns for both arguments. Draw up a table showing the values of AND and OR for each version.

10. Define two functions:

    - to return the greatest common denominator of two numbers
    - to determine whether a number is prime or not

11. What is the type of $\sharp$ defined in the text?
    What is the type of the following:
    $>$ $\quad makepair\ x\ y\ =\ (x, y)$
    Define *filter* without using list comprehension. What is its type?
    Now define *filter* using pattern-matching. Is this a better definition ?

12. The function *zip* takes a pair of lists and returns a list of pairs of corresponding elements:

$$\begin{aligned} zip\ ::\ ([*], [**]) &\to [(*, **)] \\ zip\ ((a : xs), (b : ys)) &= (a, b) : zip\ (xs, ys) \end{aligned}$$

Using *zip*, define and test a scalar product function *sp*, which returns the sum of the products of respective elements of two list arguments, i.e. (informally):

$$spxsys \quad = \quad x1 * y1 + x2 * y2 + ...$$

Using *zip*, define and test a function *myzip4* which converts a 4- tuple of lists into a list of 4-tuples. This exercise is not asking you to copy the Miranda standard environment *zip4* definition!

13. Suppose a list xs of integers contains an equal number of odd and even numbers. Define and test a function $riffle$ so that $(riffle\ xs)$ is some rearrangement of xs such that even and odd numbers alternate.

14. ([Hol91]Holyer 8.1) What is wrong with each of the following, assuming they are to be directly evaluated? Correct, and evaluate by hand, and confirm using Miranda:

$10\ /\ 7\ div\ 3$	$[2 + 3] * 4$
$code\ x$	$letter\ "x"$
$(1, 2)\ +\!+\ (3, 4)$	$\sharp\ 'abcd'$
$max2\ (1, 2)$	$tl\ 1 : [2..5]$

15. ([Hol91]Holyer 2.8.3) Using *hd* and a list comprehension with one generator and one filter, find the first power of 2 greater than one million.

16. 16 ([Hol91]Holyer 2.8.5) Use a list comprehension with two generators and one filter to produce the list
$[(1,1),\ (1,2),\ ...(5,5)]$
of the fifteen pairs of integers between 1 and 5 for which the first number is less than or equal to the second. Then find a second comprehension to do the same thing using just two generators and no filter.

17. ([BW88]B&W 2.8.1) Declare the types for the following function definitions:

$$\begin{aligned} const\ x\ y &= x \\ subst\ f\ g\ x &= f\ x\ (g\ x) \end{aligned}$$

Check, using Miranda.

18. ([BW88]B&W 3.5.1) Consider the function all which takes a predicate p and a list xs and returns $True$ if all elements of xs satisfy p, and $False$ otherwise. Give a formal definition of all which uses $foldr$.

19. (a) ([BW88]B&W 3.5.2) Which, if any, of the following equations are true?

$$\begin{aligned} foldl\ (-)\ x\ xs &= x\ -\ sum\ xs \\ foldr\ (-)\ x\ xs &= x\ -\ sum\ xs \end{aligned}$$

(b) ([Hol91]Holyer 3.4.5) Define a version cat of the standard function concat using foldl rather than $foldr$. Verify that they have the same effect on finite lists by trying out an example using a list of numbers and another using a list of strings.

20. (a) ([Hol91]Holyer 3.4.4) Write a function capitalise which converts the first letter of a lower case word to upper case using the standard functions code and decode. You do not need to know what codes the letters have, only that the lower case ones have consecutive code numbers, as do the

upper case ones.
(b) ([Hol91]Holyer 3.4.6) Define a function join which joins two words together with a space in between. Use join, capitalise and foldr1 to define a function sentence which takes a list of words such as ["*the*", "*cat*", "*sat*","*on*", "*the*", "*mat*"] and produce a sentence such as "*The cat sat on themat.*" by joining the words, capitalising the first word, and adding a full stop.

21. ([BW88]B&W 3.5.4) Consider the following definition of function *insert*:

$$insert\ x\ xs \quad = \quad takewhile\ (\leq x)\ xs\ +\!+\ [x]\ +\!+\ dropwhile\ (\leq x)\ xs$$

Show that if xs is a list in non-decreasing order, then so is $(insert\ x\ xs)$. Using insert, define a function *isort* for sorting a list into non-decreasing order.

22. ([Dav92]Davie 3.13.15) Write a function which converts an integer number from a given base to a number in base 10.

23. Refer to the definitions of the $\sharp$ and *reverse* functions using *foldl* in the text.
(a) Verify that

$$oneplus(x\ plusone(y\ z)) \quad = \quad plusone(oneplus(x\ y)\ z)$$

and

$$oneplus\ x\ 0 \quad = \quad plusone\ 0\ x$$

where

$$oneplus\ x\ y \quad = \quad 1\ +\ y$$

and

$$plusone\ x\ y \quad = \quad x\ +\ 1$$

(b) Veify that

$$postfix(x\ prefix(ys\ z)) \quad = \quad prefix(postfix(x\ ys)\ z)$$

and

$$postfix\ x\ [\,] \quad = \quad prefix\ [\,]\ x$$

where

$$postfix\ x\ xs \quad = \quad xs\ +\!+\ [x]$$

and

$$prefix\ xs\ x \quad = \quad [x]\ +\!+\ xs$$

24. (OPTIONAL : Lazy infinite lists) Write a definition that prints an infinite list of 1s
Write a program that prints the infinite text

"1 *sheep*, 2 *sheep*, 3 *sheep*, ..."

as an aid to insomniacs.

25. ([BW88]B&W5.1.1) Using the recursive definitions of addition and multiplication of natural numbers given in the text; prove all or some of the following familiar properties of arithmetic:

$$\begin{array}{lcll} 0 + n & = & n = n + 0 & (+ \textit{ has identity } 0) \\ 1 * n & = & n = n + 1 & (* \textit{ has identity } 1) \\ k + (m+n) & = & (k+m) + n & (+ \textit{ associative}) \\ m + n & = & n + m & (+ \textit{ commutation}) \\ k * (m*n) & = & (k*m)*n & (* \textit{ associative}) \\ k * (m+n) & = & (k*m) + (k*n) & (+ \textit{ distribution through } *) \\ m*n & = & n*m & (* \textit{ commutation}) \end{array}$$

26. ([BW88]B&W5.1.2) Prove that

$$\begin{array}{rcl} Fn+1 * Fn-1 - (Fn)^2 & = & (-1)^n \\ Fn+m & = & Fn * Fm+1 + Fn-1 * Fm \end{array}$$

for all natural numbers $n \geq 1$ and $m \geq 0$, where Fm is the mth Fibonacci number.

27. ([BW88]B&W5.1.3) The binomial coefficient, *binom n k* denotes the number of ways of choosing k objects from a collection of n objects.

a) Give a recursive definition of *binom*.

b) Prove that if $k > n$ then *binom n k* $= 0$.

c) Rewrite the equation

$$sum\_over\_k\_from\_0\_to\_n\ (binom\ n\ k) \quad = \quad 2^n$$

in functional notation.
Prove that the equation is true for all natural numbers.

28. ([BW88]B&W5.3.1) Give a recursive definition of index operation $(xs!i)$.

29. ([BW88]B&W5.3.2) Give a recursive definition of *takewhile* and *dropwhile*.

30. ([BW88]B&W5.3.3) Prove the law

$$\begin{array}{rcl} init(xs\ +\!+\ [x]) & = & xs \\ last(xs\ +\!+\ [x]) & = & x \\ xs & = & init\ xs\ +\!+\ [last\ xs] \end{array}$$

for every x and every (non-empty) finite list xs.

31. ([BW88]B&W5.3.4) Prove the laws

$$\begin{array}{rcl} take\ m\ (drop\ n\ xs) & = & drop\ n\ (take\ (m\ +\ n)\ xs) \\ drop\ m\ (take\ n\ xs) & = & drop\ (m\ +\ n)\ xs \end{array}$$

for every natural number m and n and every finite list xs.

32. ([BW88]B&W5.3.5) Prove the laws:

$$\begin{array}{rcl} map\ (f.g)\ xs & = & map\ f\ (map\ g\ xs) \\ map\ f\ (concat\ xss) & = & concat\ (map\ (map\ f)\ xss) \end{array}$$

for any functions f and g, and every finite list xss.

33. ([BW88]B&W5.3.6) Prove the law:

$$takewhile\ p\ xs\ +\!+\ dropwhile\ p\ xs \quad = \quad xs$$

for every total predicate p, and finite list xs.

34. ([BW88]B&W5.4.1) Prove that

$$(xs\ +\!+\ ys)\ -\!-\ xs \quad = \quad ys$$

for every finite list xs, ys.

35. ([BW88]B&W5.4.2) Prove that

$$reverse\ (xs\ +\!+\ ys) \quad = \quad reverse\ ys\ +\!+\ reverse\ xs$$

for every finite list xs and ys.

36. ([BW88]B&W6.1.1) Use the $T-$ and O-notations to give computation times for the following functions: hd, $last$, $(\sharp)$, fib, and $fastfib$.

37. ([BW88]B&W6.1.2) If

$$g(n) \quad = \quad O(n^2)\ -O(n^2)$$

may we conclude that
$g(n) = 0$?
What should the right-hand side of the equation be?

38. ([BW88]B&W6.2.1) Give innermost, outermost, and outermost graph reduction sequences for each of the following terms:
$cube(cube\ 3)$
$map(1+)\ (map\ (2*)\ [1,2,3])$
$hd([1,2,3]\ +\!+\ loop)$
Count the number of reduction steps in each sequence (if it terminates).

39. ([BW88]B&W6.2.2) Give the outermost reduction sequences for each of the following terms:
$zip\ (map\ sqr\ [1..3],\ map\ sqr\ [4..6])$
$take\ (1+1)(drop(3-1)[1..4])$
$take\ (42\ -6*7)(map\ sqr[1234567..7654321])$
Indicate all outermost radixes that are not reduced because of the restrictions imposed by pattern matching.

Chapter 3

Constructed Types

3.1 Type Synonyms

3.1.1 Example:*Point, Side & Area*

```
>   point       ==   (num, num)
>   geomarea    ==   num
>   length      ==   num
>   area        ::   point → point → point → geomarea
>   side        ::   point → point → length
>   side a b    =    sqrt((fsta − fstb)^2 + (snda − sndb)^2)
>   area a b c  =    sqrt(s ∗ (s − (sideab)) ∗ (s − (sidebc)) ∗ (s − (sideca)))
>                    where
>                    s = ((sideab) + (sidebc) + (sideca))/2
```

3.2 Enumerated Types

3.2.1 Example: $Direction\&Day$

$$
\begin{array}{llll}
> & direction & ::= & North|East|South|West \\
> & day & ::= & Mon|Tues|Weds|Thur|Fri|Sat|Sun \\
> & nextday & :: & day -> day \\
> & nextday\ Mon & = & Tues \\
> & nextday\ Tues & = & Weds \\
> & nextday\ Weds & = & Thur \\
> & nextday\ Thur & = & Fri \\
> & nextday\ Fri & = & Sat \\
> & nextday\ Sat & = & Sun \\
> & nextday\ Sun & = & Mon
\end{array}
$$

$$
\begin{array}{llll}
> & workday & :: & day \rightarrow bool \\
> & workday\ d & = & Mon \leq d \leq Fri
\end{array}
$$

3.3 Algebraic Types

3.3.1 Example:$Fahrt,\ Date,\ \&Perrec$

$$
\begin{array}{llll}
> & fahrtemp & ::= & Fahrt\ num \\
> & freezing & :: & fahrtemp \rightarrow bool \\
> & freezing\ (Fahrt\ n) & = & n < 32
\end{array}
$$

$$
\begin{array}{llll}
> & day & ::= & Date\ num\ num\ num \\
> & perrec & ::= & Perrec\ num\ string\ day
\end{array}
$$

3.3.2 Example: $Celc, Fahr\&Kelv$

$$temp ::= Celcnum|Fahrnum|Kelvnum$$

3.3.3 Example: File

$$
\begin{array}{llll}
> & file * & ::= & File\ [*] \\
> & perfile & == & file\ perrec
\end{array}
$$

3.4 *File & Pair*

3.4.1 Example: *File & Pair*

```
>  file *      ::=  File [*]
>  numfile     =    File [3, 2, 33, 0, -13]
>  stringfile  =    File ["My", "name", "is", "Fred"]

>  pair * **   ::=  Pair * **
>  pairnb      =    Pair 7 False
>  pairlns     =    Pair [8, 9, 17] "kid's ages"
```

3.4.2 Natural Numbers as Algebraic Types

3.4.3 Example: *speed, temp, nat, & radd*

```
>  speed              ::=  Slow | Average | Fast
>  temp               ::=  Fahr num | Celc num | Kelv num
>  nat                ::=  Zero | Succ nat
>  radd n Zero        =    n
>  radd n (Succ m)    =    Succ (radd n m)
```

3.4.4 List of Natural Numbers

3.4.5 Exercise: *numlist, stringlist & rtake*

```
>  list *                      ::=  Nil | Cons * (list *)
>  numlist                     ==   list num
>  stringlist                  ==   list [char]

>  rtake                       ::   nat -> list * -> list *
>  rtake Zero xs               =    Nil
>  rtake (Succ n) Nil          =    Nil
>  rtake (Succ n) (Cons e xs)  =    Cons e (rtake n xs)
```

3.4.6 Binary Tree

3.4.7 Exercise: *btree, build, squash & magic*

$$
\begin{array}{lll}
> \quad xbtree\ * & ::= & Null \mid Leaf\ * \mid Node\ (xbtree\ *)\ *\ (xbtree\ *) \\
 & & \\
> \quad build & :: & [*] \to xbtree\ * \\
> \quad build\ [] & = & Null \\
> \quad build\ [x] & = & Leaf\ x \\
> \quad build\ (x : xs) & = & Node\ left\ x\ right \\
> & & where \\
> & & left\ =\ build\ (filter\ (<= x)\ xs) \\
> & & right\ =\ build\ (filter\ (> x)\ xs) \\
 & & \\
> \quad squash & :: & xbtree\ * \to [*] \\
> \quad squash\ Null & = & [] \\
> \quad squash\ (Leaf\ x) & = & [x] \\
> \quad squash\ (Node\ left\ x\ right) & = & squash\ left + + [x] + + squash\ right \\
 & & \\
> \quad magic & = & squash.build
\end{array}
$$

3.5 End Of Chapter Exercises

1. Give an alternative definition of addition (to `radd`) for type *nat*.
2. Can you define subtraction for type *nat* as a model of the naturals? If so, how, and are there any problems? If not, why not?
3. ([BW88]B&W 8.3.2) Define multiplication as an operation on type *nat*.
4. Define subtraction for the following representation of the integers:

```
int ::= Pos nat | Neg nat
```

5. Define operations `rtakewhile` and `rconcat` (analogous to *takewhile* and ++ for type [*]) for type *list* *.
6. Define operations `rmap` and `rlength` (analogous to *map* and # for type [*]) for type *list* *.
7. Define a function `mirror` to form the mirror image of a tree of type *xbtree* *.
8. For type `btree *` in B&W define a function `maptree` which will apply a given function `(type * -> **)` to the data item `type *` contained in each leaf (*tip* according to B&W).
9. Using functions you have seen in lectures & the book, define a function which will take any numeric tree of type *xbtree* * and convert it into an *ordered tree*, i.e. one which can be "squashed" (using *squash*) into an ordered list.

10. Test the set functions: define some example sets and try out the functions. Get a feeling for what more complex expressions do, such as (for example)

```
comp (union (intersect a b) ) (<10)
```

    where `a`, `b`, `c` are of type *set num*.

11. Test the laws discussed in section 3.2 for sets, by defining example constant sets. Suggest one or two laws of your own and test them.

12. Give an expression using function `comp` and an input list of strings, to return the set of strings in that input list containing at least one upper case letter. You will need to specially define a predicate function.

13. Give an expression using function `gen` and the input list `[1..10]`, to return the set of squares of even numbers between 1 and 10.

14. Give an expression using function `exists` to determine whether the set of names of your classmates contains one starting with the letter 'U'.

15. Define (an implementation function for) `forall`, for type `aset~ *`.

16. `a`, `b`, and `c` are three sets that intersect. Define a function using *aset* $*$ operations `intersect` and `setdiff` that returns all elements in set a that are NOT in BOTH `b` and `c`. (You will need to complete exercises 4.1 and 4.2 if you want to execute your function, but this is an option)

17. Experiment with type *stack* $*$ to satisfy yourself that

    - operations work as you would expect - operations only work when preconditions apply - the laws are satisfied by example stacks - the representation of the type is "encapsulated", e.g. what happens if (knowing the representation is '[*]'), you try to do

        5 : s where s is of type `stack num` ?

    - the comments in the lecture about `show` functions are justified:

18. The following is a constructed type for temperatures:

```
temp ::= Fahr num | Celc num
```

    You are required to make an example stack of `temp`'s. - Define an element display function, to display an element as Celcius. - Apply verb+showaset+ to your example stack, to show the stack in Celcius format. Use the UNIX script facility to record your results.

19. For the abstract type *stack* $*$, give the signature and definition of operation `pop`.

20. Prove law 2 for type *stack* $*$.

21. Prove law 4 for type *stack* $*$.

22. Give the signature of, and define, the precondition for `pop`.

23. Give signatures and definitions for operations `join` and `front` for type *queue* $*$.

24. Prove the two stated laws for type *queue* $*$.

Chapter 4

Abstract Types

4.1 Abstract Types from Outside: The Set

- Yet another mechanism for type construction: WHY?
- Recall: Comparison of Miranda 'built-in' / 'primitive' types with constructed 'representation' types. e.g., (Section 3.1.2, Section 3.1.3)
  - *nat* for (some of) *num*
  - *list* $*$ for $[*]$
- So, built-in types (e.g., $[*]$) come with built in structure, e.g.,

$$[\ ,\ ,\ ,\ ] \quad : \quad [x|x \leftarrow \ ...;\ ...]$$

We examined the possible internal structure (representation) using constructed types.

4.1.1 Type *aset* $*$

- Imagine a type *aset* $*$ which gives us sets of things of type * (cf. $[*]$)
- Sets vs Lists:
  - No duplication
  - No ordering
- So:
  - Not valid set: $\{5,1,1,3,3\}$
  - $\{1,3,5\} = \{5,1,3\} = \{5,3,1\}$
  - Valid list: $[5,1,1,3,3]$
  - $[5,1,1,3,3] \neq [1,3,5] \neq [5,1,3] \neq [5,3,1]$

- Some operation definitions for *aset**

```
>  abstype aset *
>  with
>||emptyset: give an empty set
>  emptyset :: aset *
>||mkaset: makes a list of elements into a set of those elements
>  mkaset :: [*] -> aset *
>||setaslist: useful operation to express a set as a list
>  setaslist :: aset * -> [*]
>||card: give cardinality of set (no. of elements)
>  card :: aset * -> num
>||belongs_to: is a value a member of a set?
>  belongs_to :: * -> aset * -> bool
>||subset_of: is a set a subset of another set?
>  subset_of :: aset * -> aset * -> bool
>||union: give the union of two sets
>  union :: aset * -> aset * -> aset *
>||showaset: set display operation
>  showaset :: (* -> [char]) -> aset * -> [char]
>||comp: gives a set as set comprehension using a single predicate
>  comp :: aset * -> (* -> bool) -> aset *
>||gen: gives a set using a filter and a generator function
>  gen :: aset * -> (* -> bool) -> (* -> **) -> aset **
```

4.1.2 Examples:

- `mkaset :: [*] -> aset *`

$$mkaset\ [1,2,3,2,1,4]\ \Rightarrow\ \{1,2,3,4\}$$

```
> numset :: aset num
> numset = mkaset [1, 2, 3, 1, 4]
```

$$mkaset\ [`a`, `z`, `\$`]\ \Rightarrow\ \{`a`, `z`, `\$`\}$$

```
> chset  :: aset char
> chset  = mkaset [`a`, `z`, `\$`]
```

$$mkaset\ [\text{“}fred'', \text{“}joe'', \text{“}sue'']\ \Rightarrow\ \{\text{“}fred'', \text{“}joe'', \text{“}sue''\}$$

```
> strset :: aset [char]
> strset = mkaset [``fred'', ``joe'', ``sue'']
```

- `card :: aset * -> num`

$$card\ (mkaset\ [1,2,3,1,4])\ \Rightarrow\ 4$$

$$card\ (mkaset\ [`a`, `z`, `\$`])\ \Rightarrow\ 3$$

$$card\ (mkaset\ [\text{“}fred\text{”}, \text{“}joe\text{”}, \text{“}sue\text{”}])\ \Rightarrow\ 3$$

- `belongs_to :: * -> aset * -> bool`

$$belongs\_to\ 5\ (mkaset\ [1,2,3,1,4])\ \Rightarrow\ False$$

$$belongs\_to\ `a`\ (mkaset\ [`a`, `z`, `\$`])\ \Rightarrow\ True$$

$$belongs\_to\ \text{“}jim\text{”}\ (mkaset\ [\text{“}fred\text{”}, \text{“}joe\text{”}, \text{“}sue\text{”}])\ \Rightarrow\ False$$

- `union :: aset * -> aset * -> aset *`

```
> set1 = mkaset [1,3,5,3,7,9]
```

$$set1\ \Rightarrow\ \{1,3,5,7,9\}$$

```
> set2 = mkaset [2,4,6,8,9]
```

$$set2\ \Rightarrow\ \{2,4,6,8,9\}$$

$$union\ set1\ set2\ \Rightarrow\ \{1,2,3,4,5,6,7,8,9\}$$

$$set1 \cup set2$$

- `comp :: aset * -> (* -> bool) -> aset *`

$$comp\ set2\ (>5)\ \Rightarrow\ \{6,8,9\}$$

$$\{s : S | p\ s\}$$

“filter for sets”

- `gen :: aset * -> (* -> bool) -> (* -> **) -> (aset **)`

$$gen\ set1\ (<= 5)\ (*100)\ \Rightarrow\ \{100,300,500\}$$

$$\{s : S | p\ s\ .\ g\ s\}$$

“generator for sets”

- `forall:: aset * -> (* ->bool) -> bool`

$$forall\ set1\ (>= 1)\ \Rightarrow\ True$$

$$forall\ set1\ (<6)\ \Rightarrow\ False$$

$$\forall\ s : set1.s < 6$$

- `exists :: aset * -> (* -> bool) -> bool`

$$exists\ set2\ (>8)\ \Rightarrow\ True$$

$$\exists s : set2.s > 8$$

- operator sections:

$$(>8),\ (<6)$$

4.1.3 Set Laws

Some LAWS for *aset* $*$: For any sets A, B of type *aset* $*$ and any element a of type $*$,

1. $$A \cup B = B \cup A$$

   i.e.,

   $$union\ A\ B\ =\ union\ B\ A$$

   Miranda Notation:

   ```
   > union setA setB $seteq union setB setA
   ```

2. $$a \in A = \{a\} \subset A$$

   i.e.,

   $$belongs\_to\ a\ A = subset\_of\ (mkaset\ [a])\ A$$

   Miranda Notation:

   ```
   > belongs_to a setA = subset_of (mkaset [a]) setA+
   ```

4.1.4 Examples:

```
> a = mkaset [2,3,4,3,5,6,6]
> b = mkaset [3,5,79,11,3,5]
> c = mkaset [8,6,4,2,6]

> testLaw1 = union a b $seteq union b a
> testLaw2 = union a (union b c) $seteq union (union a b) c
```

4.1.5 Miranda Sessions

```
Miranda (union a b) $seteq (union b a)
True
Miranda ((interesct a b) $union (setdiff a b)) $seteq a
True
Miranda belongs_to 5 a
True
Miranda subset_of (mkaset [5]) a
True

...etc.
```

4.1.6 Implementation of $aset\ *$ using Lists

```
> aset *  == [*]
>         emptyset            =  []
>         mkaset []           =  []
>         mkaset (x : xs)     =  mkaset xs, if member xs x
>                             =  x : mkaset xs, otherwise
>         setaslist set       =  set
>         card set            =  #set
>         belongsto x set     =  member set x
>         subsetof set1 set2  =  and [belongsto x set2 | x <- set1]
>         union set1 set2     =  mkaset (set1 ++ set2)
>         intersect set1 set2 =  mkaset [x|x <- set1; y <- set2; x = y]
>         setdiff set1 set2   =  mkaset (set1 -- set2)
>         comp set p          =  filter p set
>         gen set p g         =  map g (filter p set)
>         seteq set1 set2     =  subset_of set1 set2 & subset_of set2 set1
>         showaset f set      =  "{" ++ init(showmems f set) ++ "}", if set = emptyset
>                             =  "{}", otherwise
>                                where
>                                showmems f set = "", if set = emptyset
>                                = f (hd set) ++ "," ++ showmems f (tl set), otherwise
```

4.1.7 Some Constant Definitions

```
>       charset     =  mkaset['a', 'b', '5', '8', '$', 'S', 'W']
>       strset      =  mkaset ["qwerty","KBD","123","father","etc.","$%^&"]
>       digset      =  comp charset digit
>       lenset      =  gen strset hasUpper(#)
>       hasUpper str =  or (map isUpper str)
>       isUpper c   =  'A' <= c <= 'Z'
```

4.2 Abstract Types

- Composite types so far have been made up by grouping 'primitive' types into particular forms (or templates) for values they are to describe, e.g.,

```
> string == [char]
> atuple == (string,[num],string)
> atuplist * == [(num, *)]
> vehicle ::= Car string |
>       Truck string num num |
>       Bicycle
```

- Each of the types above on the left are determined by the form and the 'primitive' types on the right, i.e., by the kinds of values they represent. They are called Concrete types (Bird & Wadler).

- The operations satisfied by values of such types depend purely on the form of these definitions and on the operations satisfied by the primitive types they are made from.
- Such types are very unlike 'primitive' types (e.g., *num*), which have their forms HIDDEN, and are determined purely by the operations that the values undergo (e.g., *add num num*).
- Classifying values by the kinds of operations they usually undergo rather than by their form makes types more useful in describing solutions of real problems.
- This suggests a better way of defining another kind of composite types - Abstract types, by operational considerations rather than by the representation of values. They are better at modelling real (more abstract) objects.
- An example of abstract type is *aset* $*$. This models mathematical sets. Its operations are in the forefront having the properties of the mathematical sets. The actual representation is HIDDEN, just like in the case of primitive types like *num*. Another example is the *stack*.
- The operations satisfied by abstract types satisfy certain properties. For example, for num, idempotent, commutative, associative and distributive laws under its various operations, +, *, -, /. We have also seen laws satisfied by operations on sets.

4.2.1 Abstype Stack

The operations of interest are:

1. empty: return an empty stack
2. push: give a new stack formed by adding an element to an old stack.
3. top: give a copy of the top element of the stack
4. pop: give a new stack obtained after removing the top element of an old stack.
5. isempty: is the given stack empty?

These operations satisfy laws such as:

1. top(push x s) = x (... for any element x and any stack s)
2. pop(push x s) = s (... for any element x and any stack s)
3. isempty empty = True
4. isempty(push x s) = False (... for any element x and any stack s)

Law 1 simply restates LIFO (Last In First Out) property of a stack

Abstract types are constructed in Miranda in two stages:

1. Defining a type name with appropriate operation signatures. The syntax for this is:

```
abstype <typename> ---- <typename> may be polymorphic (e.g., stack *)
with
<operation type signatures>
```

This is the only part of the abstract type that is visible to the application. In other words, variables of this type may be operated on only by the operations that appear in its type definition.

2. Implementing the type giving a definition of the type, and definitions of the operations.

   This part is hidden from the application in which this type is used. There are usually many ways of implementing the abstract type. These are called different representations for the type. However different the implementation the operations will have the same effect. In theory, the application should not be aware of the implementation. In practice, however, this might be noticed by speed and memory requirement differences.

Miranda Abstract types are NOT truely abstract. In addition to operations in the type definition, certain other operations may be available in one representation, but not in another. For example, equality and ordering relations are available in representations using lists, but not in represerntions using functions.

Example: stack * (polymorphic stack)

```
>  abstype stack *
>  with
>|| signature expressions
>|| return an empty stack
> empty :: stack *
>|| return a stack after pushing an element to stack
> push :: * -> stack * -> stack *
>|| pre-condition for top (stack non-empty?)
> pre_top :: stack * -> bool
>|| return top element of stack
> top :: stack * -> *
>|| pre-condition for pop (stack non-empty?)
>|| pre_pop :: ??
>|| return stack after popping
>|| pop :: ??
>|| check stack is empty (stack empty?)
> isempty :: stack * -> bool
>|| display all stack elements
> showstack :: (* -> [char]) -> stack * -> [char]
```

This models stacks (of numbers, of characters, of plates, etc., hence polymorphic).

(There may be some problem with "offside rule". Start the implementation indented to the left of the operations list above.)

An implementation (representation) of the stack using lists is given below:

```
>|| hidden representation of type
> stack * == [*]
>|| implementation equations for operations
>|| return a null list (stack)
> empty = []
>|| return element consed list (stack)
> push a xs = a:xs
>|| is list non-empty? (stack)
>|| pre_top xs = ??
>|| copy head of list (stack)
> top (a:xs) = a
>|| is list non-empty? (stack)
>|| pre_pop xs = ??
>|| return head removed list (stack)
>|| pop (x:xs) = ??
>|| is list empty? (stack)
> isempty xs = xs=[]
>|| show all list elements (stack)
> showstack f []     = ""
> showstack f (x:xs) = ``(`` ++ f x ++ " " ++ showstack f (xs) ++ ``)''
```

4.2.2 Laws on stack function

$$Law\ 1: \ top\ (push\ x\ s)\ =\ x\ ...\ for\ any\ element\ x,\ any\ stack\ s$$

Proof :

$top\ (push\ x\ s)$
$=$ $(defn.\ of\ push : push.1)$
$top\ (x : s)$
$=$ $(defn.\ of\ top : top.1)$
x

$$Law\ 3: \ isempty\ empty\ =\ True$$

Proof :

$isempty\ empty$
$=$ $(defn.\ of\ empty : empty.1)$
$isempty\ []$
$=$ $(defn.\ of\ isempty : isempty.1)$
$[]\ =\ []$
$=$ $(apply\ '='\ \ operator := .1)$
$True$

4.2.3 Testing the Laws

$$
\begin{array}{lcl}
> numstack & = & push\ 3\ (push\ 7\ (push\ 6\ empty)) \\
> strstack & = & push\ "123"\ (push\ "father" \\
> & & (push\ "caught"\ (push\ "a\ flea"\ empty)))
\end{array}
$$

4.2.4 Another Law

$$pop :: \ stack * \rightarrow (*, stack *)$$

$$Law : \ pop(push\ x\ s) \ = \ s$$

$$> showstack\ (push\ 6(snd(pop\ s)))$$

4.2.5 Notes on Abstract Types:

1. Encapsulation of representation: dirty programming (using 'hd', 'tl', etc.)
2. Implementation independence
3. Display of values of abstract types: cannot be automatically shown. showstack function has two arguments: a 'show' function, taking a stack componennt of type '*' to (displayable) type [char], and the stack argument.. Define an abstype operation 'showx' gor an abstype 'x'.

   ```
   Mira> push 1 empty
   <abstract object>

   Mira>push 1 empty
   (push 1 empty)
   ```

   $$showstack\ show\ (push\ 1\ empty)$$

   will work.
4. Precondtion and Postcondition: What happens if you try to apply 'top' or 'pop' to an empty stack? ... Partial function So define PRECONDITION for any given operation. If

   $$op ::< some\ argument > \rightarrow < result >$$

   then its precondtion is a PREDICATE with the same signature, except for the result:

   $$pre\_op ::< some\ argument > \rightarrow bool$$

   So,

   $$pre\_top\ s = \tilde{}isempty\ s$$

   The Precondition: Usually seen as a predicate on both the arguments to, and the result of the operation. It tells us something about what the operation does.

5. Classifying operations of an abstract type:

   (a) CONSTRUCTOR.

   $$op :: things \to T$$

   $$op :: things \to (T, things)$$

   (b) ACCESSOR:

   $$op :: things \to T \to things$$

   (c) TRANSFORMER:

   $$op :: things \to T \to (T, things)$$

6. Abstract types don't need to be polymorphic!

4.2.6 Abstract Queue

```
> string == [char]

> abstype queue *
> with
>     start                    :: queue *
||   empty queue
>     join                     :: * -> queue * -> queue *
||   return queue after joining
>     pre_front                :: queue * -> bool
||   check queue non-empty
>     front                    :: queue * -> *
||   view front element
>     pre_reduce               :: queue * -> bool
||   check queue non-empty
>     reduce                   :: queue * -> queue *
||   return queue after serving
>     showqueue                :: (* -> string) -> queue * -> string

||hidden representation
> queue * == [*]

||implementation
> start              =  []
> join x xs          =  xs ++ [x]
> pre_front xs       =  xs ~= []
> front (x : xs)     =  x
> pre_reduce xs      =  xs ~= []
> reduce (x : xs)    =  xs
> showqueue f []     =  "[]"
> showqueue f (x:xs) =  f x ++ ":" ++ showqueue f xs
```

```
||another hidden representation
|| > queue *  == repqueue *

||implementation
|| > repqueue *                  ::=  Start | Join * (repqueue *)

|| > start                       =    Start
|| > join x Start                =    Join x Start
|| > join y (Join x q)           =    Join y (Join x q)
|| > pre_front q                 =    (q ~= Start)
|| > front Start                 =    error"Empty Queue"
|| > front (Join x Start)        =    x
|| > front (Join x q)            =    front q
|| > pre_reduce q                =    (q ~= Start)
|| > reduce (Join x Start)       =    Start
|| > reduce (Join z (Join y x))  =    Join z (reduce (Join y x))
|| > showqueue f q               =    "Empty", if (pre_reduce q)
|| >                             =    f (front q) ++ showqueue f (reduce q), otherwise
```

4.2.7 Four Laws on Queues:

```
Law1  =  front (join x start)        =  x
Law2  =  front (join y (join x q))   =  front (join x q)
Law3  =  reduce (join x start)       =  start
Law4  =  reduce (join y (join x q))  =  join y (reduce (join x q))
```

4.2.8 Tesing the Laws on Queues:

```
> testLaw1  =  front (join x start)        =  x
>                                             where
>                                             x = 1
> testLaw2  =  front (join y (join x q))   =  front (join x q)
>                                             where
>                                             x = 1
>                                             y = 2
>                                             q = join 5(join 6(join 7(join 8(join 9 start))))
> testLaw3  =  reduce (join x start)       =  start
>                                             where
>                                             x = 1
> testLaw4  =  reduce (join y (join x q))  =  join y (reduce (join x q))
>                                             where
>                                             x = 1
>                                             y = 2
>                                             q = join 5(join 6(join 7(join 8(join 9 start))))
```

4.3 The Set Abstype

- describe precisely the operations and properties we wish sets to have.
- define the operation signatures and properties (laws) before we choose a model.
- prove the model.
- polymorhic

4.3.1 ADT aset Signatures

$$
\begin{array}{lll}
> abstype\ aset\ * & & \\
> with & & \\
> & emptyset & :: \ aset\ * \\
> & mkaset & :: \ [*] \ \rightarrow \ aset\ * \\
> & setaslist & :: \ aset\ * \rightarrow [*] \\
> & card & :: \ aset\ * \rightarrow num \\
> & belongs\_to & :: \ * \rightarrow aset\ * \rightarrow bool \\
> & subset\_of & :: \ aset\ * \rightarrow aset\ * \rightarrow bool \\
> & union & :: \ aset\ * \rightarrow aset\ * \rightarrow aset\ * \\
> & seteq & :: \ aset\ * \rightarrow aset\ * \rightarrow bool \\
> & intersect & :: \ aset\ * \rightarrow aset\ * \rightarrow aset\ * \\
> & setdiff & :: \ aset\ * \rightarrow aset\ * \rightarrow aset\ * \\
> & comp & :: \ aset\ * \rightarrow (* \rightarrow bool) \rightarrow aset\ * \\
> & gen & :: \ aset\ * \rightarrow (* \rightarrow bool) \rightarrow (* \rightarrow \ **) \rightarrow aset\ ** \\
> & exists & :: \ aset\ * \ \rightarrow \ (* \ \rightarrow \ bool) \ \rightarrow \ bool \\
> & forall & :: \ aset\ * \ \rightarrow \ (* \ \rightarrow \ bool) \ \rightarrow \ bool \\
> & showaset & :: \ (* \rightarrow [char]) \rightarrow \ aset\ * \rightarrow [char]
\end{array}
$$

4.3.2 ADT aset Representation

$$> aset\ * \ == \ [*]$$

4.3.3 ADT aset Implementation

```
> emptyset          =  []
> mkaset []         =  []
> mkaset (x : xs)   =  mkaset xs,                                  if member xs x
>                   =  x : mkaset xs,                              otherwise
> setaslist set     =  set
> card set          =  #set
> belongs_to x set  =  member set x
> subset_of set1 set2 = and [ belongs_to x set2 | x ← set1 ]
> union set1 set2   =  mkaset (set1 ++ set2)
> seteq set1 set2   =  subset_of set1 set2                         subset_of set2 set1
> intersect set1 set2 = [x|x ← set1; belongs_to x set2]
> setdiff set1 set2 = [x|x ← set1; belongs_to x set2]
> comp set p        =  filter p set
> gen set p g       =  map g (filter p set)
> exists set p      =  or (map p set)
> forall set p      =  (comp set p = set)
> showaset f set >  =  "{" ++ init(showmems f set) ++ "}",          if set ~= emptyset
>                   =  "{}",                                       otherwise
>                      where
>                      showmems f set = "",                        if set ~= emptyset
>                      = f (hd set) ++"," ++ showmems f (tl set),  otherwise
```

4.3.4 Notes

```
> atestset    =  mkaset ["abc", "def", "ghi"]
> listset     =  mkaset [[1,2,3]. [4,5,6], [2,4,6], [1,3,5]]
> dlist ls =  =  "[" ++ concat (map shownum ls) ++ "]"
```

4.4 End Of Chapter Exercises

1. Give an alternative definition of addition (to `radd`) for type *nat*.

2. Can you define subtraction for type *nat* as a model of the naturals? If so, how, and are there any problems? If not, why not?

3. ([BW88]B&W 8.3.2) Define multiplication as an operation on type *nat*.

4. Define subtraction for the following representation of the integers:

```
int ::= Pos nat | Neg nat
```

5. Define operations `rtakewhile` and `rconcat` (analogous to *takewhile* and ++ for type [∗]) for type *list* ∗.

6. Define operations `rmap` and `rlength` (analogous to *map* and # for type [∗]) for type *list* ∗.

7. Define a function `mirror` to form the mirror image of a tree of type *xbtree* $*$.

8. For type `btree *` in B&W define a function `maptree` which will apply a given function `(type * -> **)` to the data item `type *` contained in each leaf (*tip* according to B&W).

9. Using functions you have seen in lectures & the book, define a function which will take any numeric tree of type *xbtree* $*$ and convert it into an *ordered tree*, i.e. one which can be "squashed" (using *squash*) into an ordered list.

Chapter 5

Specificatios

5.1 Introduction To Specification

5.1.1 Example: plane

Informal Specification

System to record the passengers on board a plane:

1. record identity of each person on board – name (assume unique)
2. initialise capacity of the plane – a fixed number (no. on board cannot exceed it)

Formal Specification

Some Auxiliary Type Synonyms:

To simplify matters introduce the following type synonyms:

```
string==[char]
person==string
```

State:

The state of the system at any time is described by:

- set of people on board at that time
- capacity of plane

This is represented in Miranda as a tuple:

```
(aset person, num)
```

using ADT *aset* $*$.

Type Synonym:

The type synonym *plane* will then give the state of the system:

```
plane == (aset person, num)
```

ADT plane

In order to encapsulate (hide) this structure we define an abstract data type *plane* whose implementation will be this tuple.

State Invariant

The ADT *plane* needs to satisfy the state invariant for this system. This invariant (or law) is:

- no. of people on board $\leq$ capcity of the plane
- the capacity is a non-negative whole number

In Miranda, this is expressed as a function, *p_inv*:

```
||signature (part)
p_inv :: plane -> bool
||implementation (part)
p_inv (onboard, capacity) = card onboard <= capacity &
                            natural capacity
```

Here, *natural* is a built-in (predicate) function:

```
natural :: num -> bool
natural x = integer x & x >= 0
```

natural itself uses the built-in predicate *integer*. *p_inv* is a predicate on values of type *plane*.

Initial State:

```
||signature (part)
p_empty :: num -> plane
||implementation (part)
p_empty n = (emptyset, n)
```

- p_empty is a CONSTRUCTOR for '*plane*'.
- It creates a *plane* with an initial state with specified capacity and empty set of persons.

Precondition for *p_empty*

In order for *plane* so constructed to satisfy the invariant, *p_inv*, *n* must be *natural*.This is because *p_empty* assumes that *num* is a natural number.

```
||signature (part)
pre_p_empty :: num ->bool
||implementation (part)
pre_p_empty n = natural n
```

Boarding operation on *plane*

board: takes a person and adds it to plane:

```
||signature (part)
board :: person -> plane -> plane
```

This amounts to adding a person to the set of people on board using set union:

```
||implementation (part)
board per (onboard, capacity) = (union onboard (mkaset [per]), capacity)
```

Precondition for *board*

```
||signature (part)
pre_board :: person -> plane -> bool
```

This precondition requires:

- Person should not already be in plane.

- No. on board must be less than capacity of plane.

```
||implementation (part)
pre_board per (onboard, capacity)
= ~ belongs_to per onboard & card onboard < capacity
```

Note:

Because, a set cannot contain duplicate elements, *board* operation will not aboard a person already on board (the cardinality is not affected either), i.e., *board* will succeed, but will not affect the state of the plane. So,

```
||alternative implementation
pre_board per (onboard, capacity) = card onboard < capacity
```

will do.

Reporting

Operation to determine whether a person is on board

```
||signature (part)
is_onboard :: person -> plane -> bool
||implementation (part)
is_onboard per (onboard, capacity) = belongs_to per onboard
```

Precondition for *is_onboard*

```
>|| pre_is_onboard:: person -> plane -> bool
>|| pre_is_onboard per pln = True
```

Note:

- *is_onboard* is a TOTAL operation.
- Can be applied to every person and plane, including the case when the person is null or plane is empty.

So no need for precondition, as *is_onboard* is always defined.

Packaging

Using the ADT *plane*:

```
%include ``plane.m''
u_board::person->plane->plane
u_board per pln
  = board per pln, if pre_board per pln
  = error``Cannot do ... person already aboard or plane already full'', otherwise
```

Note:

- plane.m already includes aset.m.
- Only plane operations are visible, not set operations.

Testing

Use in-script definitions such as :

```
> eplan = p_empty 4
> p1 = board ``fred'' epln
> p2 = board ``joe'' p1
> p3 = board ``mary'' p2
....
```

and capture as UNIX script files the Miranda session (animation):

```
Miranda pre_board ``fred'' p2
False
Miranda is_onboard ``susan'' p3
False
....
```

5.1.2 Another Example: Plane Reservation/Confirmation System

Reservation

- *reserve*: record reservation
- *pre_reserve*: record must not be in the reservation or confirmation lists

This forms the basis of your next course-work.

Confirmation

- *confirm*: remove from reservation list and add to confirmation list
- *pre_confirm*: must be in resevation list and NOT in confirmation list

Here, *rplane* is an ADT which encapsulates set of reserved persons, set of confirmed persons and the capacity of the plane.

ADT *rplane* (incomplete)

```
> abstype rplane
> with
>    rp_inv :: rplane -> bool
>    rp_reserve::??
>    rp_pre_reserve::??
>    rp_confirm:: person -> rplane -> rplane
>    rp_pre_confirm:: person -> rplane -> bool

> rplane ==(aset person, aset person, num)

> rp_inv (res,con,n) = card (union res con) <= n &
                       intersect res con $setequal emptyset &
                       natural n

> rp_confirm r (res, con, n) = (comp res p, union con (mkaset [r],n)
                               where
                               p elt = elt ~= r
> rp_pre_confirm r (res, con, n) = belongs_to r res & ~belongs_to r con
```

Testing *rplane*

To test the abstype '*rplane*' it will be useful to define

```
> erplane = rp_empty 4
> rrp1 = rp_reserve ``brian'' erplane
> rrp2 = rp_reserve ``mary'' rrp1
> rrp3 = rp_confirm ``mary'' rrp2
etc.
```

5.1.3 Conclusion

- SPECIFYING: Required system behaviour is precisely specified

- ANIMATING: The behaviour of the system is animated, with a view to improving the specification. Prototyping.

SPECIFICATION

Specification phase involved:

1. Identifying user requirements,
2. Specifying the objects (types) required to build the system,
3. specifying any invariant properties of the system,
4. specifying the initial "empty" state of the system,
5. For each operation required:
   (a) specifying the type of the operation,
   (b) specifying the precondition of the operation.

Specification vs Animation vs Implementation

- Specification = WHAT the system is and does.
- Animation = WHAT the specification looks like.
- Implementation = HOW the system will do it.

Specifying Big Systems

- Objects of many types, hierarchies: e.g., different booking classes in the plane example (first, club, economy); booking system for all planes, routes, sources, desinations, etc.
- MODULARITY helps break up the system into manageable, logically related components.
- Miranda abstype is a helpful modularisation construct. e.g.,

  ```
  airline == (aset plane, aset person,...)
  ```

  using ADT plane to build ADT airline, just like ADT set was used to build ADT plane.

Postconditions

- In the examples an operation (function) definition was regarded as its own postcondition.
- Such a definition imposes certain relationships (or constraints) between the input arguments and the output result of the operation.
- The postcondition is essentially a predicate on the objects (types) involved.

- For big system, it is not always possible to arrive at operation definitions. Usually a lot of development work is needed.
- In such cases it is easier to arrive at the postcondition as a predicate. The predicate gives the constraints on the input arguments and the output result of the operation.

5.2 End Of Chapter Exercises

1. Complete the signatures and definitions for functions `intersect` and `setdiff` of abstype *aset* $*$.
2. Experiment with *aset* $*$:
   (a) Create some sets (of differing types);
   (b) Try out the operations, including `showaset`;
   (c) Can you apply a binary set operation (e.g. `subset\_of`,`union`) to two sets of different types ?
   (d) State some laws about sets and test them out (see whether they hold) for your created sets.
3. Are any operation preconditions for type *aset* $*$ are other than $True$? If so, say why, and define them. If not, suggest why.
4. Define the operation (i.e. type, pre- and postconditions) of disembarkation for type *plane*. Define the corresponding user-oriented operation.
5. Define an operation to report the number of passengers on a plane.
6. Define and package *plane* as an abstype. Implement and test one or more user-oriented operations.

Chapter 6

Maps

6.1 The Map Abstype

6.1.1 Introduction: Plane Booking System Revisited

- Need to book (i.e., associate) seats to passengers.
  - No such association was made in the simple plane example considered so far.
  - Only passenger identifiers (names) and the capacity of the plane were used.
- Relevant type synonyms as before (but including *seat*):

$$\begin{aligned} string &== [char] \\ person &== string \\ seat &== string \end{aligned}$$

- A simple model of the assignment of seats to passengers could be given as a tuple:

$$seating == (seat, person)$$

  - However, this allows too much freedom:
    - For each person there could be any number of seats
      (i.e., for each possible *second* value of *person* in the pair, there could be all possible choices for the *first* value of *seat*).
    - For each set there could be any number of persons
      (i.e., for each possible *first* value of *seat* in the pair, there could be all possible choices for the *second* value of *person*).
  - Mathematically, the tuple pair (*seat*, *person*) is a Cartesian Product of the types (sets) *seat* and *person*, i.e.,

$$seat \times person$$

  - Clearly, this does not model the real world:
    * Whereas, it is OK to have more than one seat assigned to a person,
    * it is NOT ok for a seat to be allocated to several persons.

- What is needed is a MAP type where:
  - A *seat* is "mapped" to a corresponding *person*.
  - This is done by defining a Miranda *abstype amap* $*$ $**$ which maps a type $*$ to a type $**$.
  - A first element of type *seat* must be associated with at most one second element of type *person*.
  - The *seat* forms a set of seats, and the *person* forms the range of the "mapping" of 'seats' to 'persons'.

- So,

  $$amap\ seat\ person$$

  will represent the seat booking system.

6.1.2 ADT *amap* $*$ $**$

ADT *amap* $*$ $**$ Signatures

```
> %include "aset.m"
> abstype amap * **
> with
>|| give the empty map
>   emptymap :: amap * **
>|| make a map from a list of pairs
>   mkamap :: [(*,**)] -> amap * **


>|| give domain of map (set of first components of pairs)
>   dom :: amap * ** -> aset *
>|| give range of map (set of second components of pairs)
>|| ran :: ??
>   ran :: amap * ** -> aset **


>|| domain restrict map: remove all maplets apart from
>|| those with first components in specified set
>|| drestrict :: ??
>   drestrict :: aset * -> amap * ** -> amap * **
>|| domain-delete map: remove all maplets with first
>|| components in specified set
>   ddelete :: aset * -> amap * ** -> amap * **


>|| range-restrict map: remove all maplets apart from
```

```
>|| those with second component in specified set
>   rrestrict :: aset ** -> amap * ** -> amap * **
>|| range-delete map: remove all maplets with second
>|| componenets in specified set
>|| rdelete :: ??
>   rdelete :: aset ** -> amap * ** -> amap * **


>|| overwrite map1 with map2: return map2 plus any maplets
>|| from map1 with first components not in dom map2
>   overwrite :: amap * ** -> amap * ** -> amap * **
>|| form union of two maps
>   mapunion :: amap * ** -> amap * ** -> amap * **
>|| apply map to value: given value (type *), return
>|| second component (type **) of corresponding maplet
>   apply :: amap * ** -> * -> **


>|| convert map to set of pairs
>   mapasset :: amap * ** -> aset (*,**)
>|| convert set of pairs to map
>   setasmap :: aset (*,**) -> amap * **
>|| show map contents, using given display functions
>|| for domain and range types
>   showamap :: (*->[char])->(**->[char])-> amap * **->[char]
```

ADT *amap* $*$ $**$ **Representation**

```
> amap * **  ==  [(*,**)]
```

ADT *amap* $*$ $**$ **Implementaion**

```
> emptymap = []
>|| mkamap : throw away any duplicates in domain
> mkamap [] = []
> mkamap ((x,y):rest) = mkamap rest, if member (map fst rest) x
>                     = (x,y):mkamap rest, otherwise
> dom amap = mkaset(map fst amap)
>|| ran ?? = ??
> ran amap = mkaset(map snd amap)


>|| drestrict ?? = ??
> drestrict set amap = [(x,y) | (x,y) <- amap ; belongs_to x set]
> ddelete set amap = [(x,y) | (x,y) <- amap ; ~belongs_to x set]
> rrestrict set amap = [(x,y) | (x,y) <- amap ; belongs_to y set ]
>|| rdelete ?? = ??
```

```
> rdelete set amap = [(x,y) | (x,y) <- amap; ~belongs_to y set]


> overwrite map1 map2 = mapunion map2 (ddelete (dom map2) map1)
>|| overwrite: another, more model-dependent, definition:
>|| overwrite map1 map2 = map2 ++ (ddelete (dom map2) map1)
> mapunion map1 map2 = setasmap (union (mapasset map1) (mapasset map2))
> apply m e = hd [y | (x,y) <- m ; x=e ]


> mapasset m = mkaset m
> setasmap = mkamap.setaslist
> showamap f g map = showaset showmaplet (mkaset map)
>                    where
>                    showmaplet (x,y) = f x ++" |-> "++ g y
```

IMPORTANT NOTES:

- Some of these operations will have non-'True' preconditions.
- Experiment with the operations.
- LAWS (PROPERTIES) for this type:
  - Is *mapunion map*1 *map*2 always/ever/not = *mapunion map*2 *map*1?
  - Is *overwrite map*1 *map*2 always/ever/not = *overwrite map*2 *map*1?

6.2 End Of Chapter Exercises

1. Complete the signatures and definitions for remaining operations of abstype *amap* $*$ $**$, i.e.

   `ran, drestrict, rdelete`

2. Experiment with *amap* $*$ $**$:
   (a) Create some maps (of differing types);
   (b) Try out the operations, including `showamap`;
   (c) State some laws about maps and test them out (see whether they hold) for your created maps.
3. Are any operation preconditions for type *amap* $*$ $**$ are other than $True$? If so, say why, and define them. If not, suggest why. What is strange or wrong about operation `mkamap` ?
4. State and try out some laws for types *amap* and *aset*.
5. How would you test two sets for equality? Two maps ?

Chapter 7

Plane Specification

7.1 plane: Specification using Map

7.1.1 ADT's and Type Synonyms to be used in *plane* ADT

First include the definitions of abstypes *aset* $*$ and *amap* $* \ **$:

```
> %include "aset.m"
> %include "amap.m"
```

and the defintions of type synonyms:

```
> string == [char]
> person == string
> seat   == string
```

7.1.2 Simplifying Assumptions on the System

ADT *plane*

Use the following type (*plane*) to specify a seat booking system (ignoring capacity of the plane, for the time being, for simplicity):

amap seat person

Invariants on the ADT *plane*

For simplicity assume no invariants.

ADT *plane* Signature

The abstype plane is defined with the following signature:

```
> abstype plane
> with
>||  create an initial state: empty plane
>    p_empty::plane
>||  give the owner of a given seat in the plane;
>||  'owner' is a partial function;
>||  can be applied only if seat is booked (i.e., precondition true).
>    owner::seat -> plane -> person
>||  precondition for owner: returns true if seat booked, and 'owner'
>||  can be applied; otherwise (returns false) owner cannot be applied
>    pre_owner::seat -> plane -> bool
>||  associate given seat to given persion, and add this to the
>||  plane booking; can not be applied, if seat already booked;
>||  must satisfy the precondition that seat is not booked.
>    book :: seat -> person -> plane -> plane
>||  precondition for book: returns true if seat unbooked, so that
>||  book can be applied, otherwise book cannot be applied.
>    pre_book :: seat -> person -> plane -> bool
>||  remove the booking by deleting the association between given seat
>||  and person from 'plane', can not be applied if seat not booked.
>    cancel:: seat -> person -> plane -> plane
>||  pre-condition for cancel: returns true if seat booked, so cancel
>||  can be applied, otherwise cancel cannot be applied.
>    pre_cancel::seat -> person -> plane -> bool
>||  change existing seat booking; overwrite the 'plane';
>||  cannot be applied, if seat has not been booked already;
>||  precondition: seat must be booked;
>||  if 'change' is to mean that overwrite, if seat booked, and book
>||  the seat for the person any way, even if the seat is not booked
>||  then pre-condition is weaker, i.e., always true, so not needed.
>||  'change' can be applied always: total function.
>    change:: seat -> person -> plane -> plane
>||  precondition for change: returns true, if seat booked,
>||  and 'change' can be applied, otherwise not;
>    pre_change :: seat -> person -> plane -> bool
>||  report change: change made to booked seat cancels previous
>||  booking of it; shows as a tuple, the new plane and the person
>||  who lost the seat, so that appropriate actions are be taken;
>||  no pre-condition needed as report can always be produced.
>    rchange :: seat -> person -> plane -> (plane, person)
>||  output seats booked by a person; can be always applied;
>||  no pre-condition needed: always true
```

```
>    s_booked :: person -> plane -> aset seat
>||  show the bookings: the set of all bookings (all seats and
>||  their associated persons); no pre-condition: always true
>    showplane :: plane -> string
```

ADT plane Representation

```
plane == amap seat person
```

ADT plane Implementation

```
> p_empty = emptymap
>||  creates an empty map, using map op. emptymap
>||  an initial state with no association of seat & person
> owner s p = apply p s
>||  gives the owner of seat s in plane p; using map op. 'apply'
>||  pre-condition: seat s of plane p must be booked.
> pre_owner s p = belongs_to s (dom p)
>||  returns true if s is booked: s in the domain of map, p;
>||  returns false, if s not booked: not in the domain
> book st per plane = mapunion plane (mkamap [(st,per)])
>||  form a maplet: (seat,person) pair, and add it to the
>||  'plane' map, by forming the 'mapunion' of the two maps
> pre_book st per plane = ~ belongs_to st (dom plane)
>||  st must not be booked; uses set op. belongs_to as (dom plane)
>||  is a set;true if st not in the domain of map 'plane'
> cancel st per plane = ddelete (mkaset [st]) plane
>||  removes maplet with first component st, i.e., {(st, pr)}
>||  from the map 'plane', using map op. 'ddelete'
> pre_cancel st per plane = belongs_to (st, per) (mapasset plane)
>||  (st, per) must be in the 'plane' map, map converted to
>||  set before set op. 'belongs_to' is used.
> change st per plane = overwrite plane (mkamap [(st,per)])
>||  map op. 'overwrite' replaces in the map 'plane', the maplet
>||  associated with st by the maplet:(st, per) pair
> pre_change st per plane = belongs_to st (dom plane)
>||  seat st must have been booked (does not use per)
>||  true if st is in the domain of the map 'plane'
> rchange st per plane = (change st per plane, apply plane st)
>||  map op. 'apply' gives the person in 'plane' who lost the
>||  seat 'st'; 'change' assigns seat 'st' to person 'pr'
> s_booked per plane = dom (rrestrict (mkaset [per]) plane)
>||  map op. 'rrestrict' gives the set of all maplet pairs of
>||  'plane' with second element 'per', and then map op. 'dom'
>||  gives all the seats associated with it
```

```
> showplane pln = showamap show show pln
>||  map op. showamap is used along with the built-in 'show' for
>||  both first and second elements of the maplet pairs of 'pln'
```

7.1.3 Testing ADT *plane*

Some test examples:

```
> p1 = book "S1" "fred" p_empty
> p2 = book "S2" "joe" p1
> pre_pduff = pre_book "S2" "brian" p2
> p3 = book "S3" "fred" p2
> p3can = cancel "S1" "fred" p3
> report = s_booked "fred" p3
> p3chg = change "S1" "george" p3
> p3chgr = rchange "S1" "george" p3
```

7.1.4 An injective map

- Assume for type *plane* that a person may book at most ONE seat.
- The map model supporting this is INJECTIVE, or "one-to-one".
- This can be seen from various points of view, e.g.
  - each range element is associated with at most ONE domain element
  - there are as many range elements as domain elements, i.e.,

$$p\_inv\ plane\ =\ card\ (ran\ plane)\ =\ card\ dom\ plane$$

- The 'change' operation is unchanged:

```
>|| change:: seat -> person -> plane -> plane
>|| change st per plane = overwrite plane (mkamap [(st,per)])
```

- But the precondition is now:

```
>|| pre_change :: seat -> person -> plane -> bool
>|| pre_change st per plane = belongs_to st (dom plane) &
>||                           ~ belongs_to per (ran plane)
```

7.1.5 A more realistic example

- Planes come in models with different seat numbers and ranges.
- To model this, *plane* type should include a set of possible seat numbers on the plane.

- Capacity of the plane is catered for: it's impossible to book a non-existent seat.
- A better plane:

```
> betterplane == (aset seat, amap seat person)
```

- Invariant: Any booked seat must exist on the plane

```
> p_inv:: betterplane -> bool
> p_inv (seats, bookings) = subset_of (dom bookings) seats
```

- Initial State: a possibe definition:

```
>bp_empty:: betterplane
>bp_empty = (emptyset, emptymap)
```

- this needs operations for "install seats" and "remove seats".
- Simpler to define seats already fixed in the initial state:

```
> bp_empty:: aset seat -> betterplane
> bp_empty seats = (seats, emptymap)
```

- Example operation: book a (specific) seat to a passenger:

  Set of sets is unchanged, and the new booking is "map-union-ed" onto the existing map of bookings.

```
> bpbook :: seat -> person -> betterplane -> betterplane
> bpbook st per (seats, bookings)
>    = (seats, mapunion bookings (mkamap [(st,per)]))
> pre_bpbook :: seat -> person -> betterplane -> bool
> pre_bpbook st per (seats, bookings)
>    = belongs_to st seats & || seat exists
>      ~ belongs_to st (dom bookings) || seat not yet booked
```

- Finally: need we check that?

```
> card (mapasset bookings) < card seats
```

7.2 End Of Chapter Exercises

1. Define an operation (i.e. type, pre- and postconditions) for type `plane`, to report any free (unbooked) seats.
2. Assume that no person is allowed to book more than two seats on a plane. What would the state invariant look like? Redefine the `book` operation precondition for type *plane* under this assumption.
3. For type *betterplane*, make the assumption in Q.3. State what the state invariant would look like, and define the `change` operation.

Chapter 8

Animation Tool Kit

8.1 Prototyping

[Hen86]

8.2 Executable Specification

[Gla95]

8.3 User Interface & HCI

[Tho90]

8.4 I/O

[Gor94]

Chapter 9

Conclusion

9.1 Haskell

[Tho96]

9.2 Parallelism

[Run95]

9.3 Strong Miranda

[Tur95]

9.4 Foundation

[Hue90]

Bibliography

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Dav92] A. J. T. Davie. *An Introduction to Functional Programming with Haskell System.* Cambridge University, 1992.

[Gla95] P. Glaser, H. & Henderson. Functional programming and software engineering. *Unknown*, 1995.

[Gor94] A.J. Gordon. *Functional Programming and Input/Output.* British Computer Society Distinguished Dissertation in Computer Science. Cambridge. Cambridge University Press, 1994.

[Hen86] P. Henderson. Functional programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engingeering*, SE-12(2), 1986.

[Hol91] I. Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[Hue90] G. Huet, editor. *Logical Foundations of Functional Programming.* Addison Wesley, 1990.

[Run95] D. Runciman, C. & Wakeling. *Applications of Functional Programming.* UCL Press, 1995.

[Tho90] S. Thompson. Interactive functional programs: A method and a formal semantics. In D. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, pages 249–286. University of Texas at Ausin, Addison-Wesley Publishing Company, 1990.

[Tho96] S. Thompson. *Haskell - The Craft of Functional Programming.* Addison-Wesley, 1996.

[Tur95] D. Turner. Elementary strong functional programming. *Unpublished*, 1995.

Appendix A

A Guide to the ADT Animator Toolkit

A.1 Linking to ADT Animator Toolkit scripts

Prepare for a ADT animator development by creating a directory, making it the working directory and linking to the animator programs:

1. Create a directory for the ADT animation (e.g., *STACK*):

   ```
   $ mkdir STACK
   ```

2. Move into this directory:

   ```
   $ cd STACK
   ```

3. Make symbolic links to the toolkit driver and animator programs in my area:

   ```
   $ ln -s /usrpk/cs/staff/csx135/205CS/ANIMATOR/*.m .
   ```

The directory STACK will now contain links to the following files of the Toolkit: (use `ls` to check)

```
animator.m     driver.m       linker.m       string_handling.m
ansi.m         interact.m     menu.m         types.m
des.m          io.m           menued.m       utils.m
desed.m        linem          screen.m
```

A.2 Creating ADT and Interface scripts

Create a ADT miranda file and an interface file, and ensure that there are no errors (types, etc.):

A.2.1 ADT

Create a ADT miranda file, with no errors (types, etc.):

1. Create a ADT miranda script file (e.g., *stack.m*) using an editor (e.g., *emacs*):

   ```
   $ emacs stack.m
   ```

2. Use Miranda to ensure that there are no errors in this script:

```
$ mira
miranda /f stack.m
```

The completed `stack.m` is:

```
|| stack ADT -- A. Amatya & M. Poppleton -- 2/8/95
abstype stack * with
|| signature expressions
    empty :: stack *
    push :: * -> stack * -> stack *
    pre_top :: stack * -> bool
    st_top :: stack * -> *
    pre_pop :: stack * -> bool
    pop :: stack * -> stack *
    showstack :: (* -> [char]) -> stack * -> [char]
|| hidden representation
stack * == [*]
|| implementatioin equations
    empty = []
    push a xs = a:xs
    pre_top xs = (xs ~= [])
    st_top (x:xs) = x
    pre_pop xs = (xs ~= [])
    pop (x:xs) = xs
    showstack f xs = f (st_top xs) ++ showstack f (pop xs), if pre_top xs
           = "Empty", otherwise
```

A.2.2 Interface

Create an interface (configuration) miranda file, with no errors (type, etc.):

1. Create an interface miranda script file (e.g., *int – stack.m*):

```
$ emacs int-stack.m
```

2. Again, use Miranda to ensure that there are no errors in it:

```
$ mira
miranda /f int-stack.m
```

The completed `int-stack.m` files is:

```
||This is a toolkit interface file for the Abstract Data Type Stack
||The ADT file related is stack.m

%include "stack.m"
%export "stack.m" statetype init_state call_new call_push call_pop call_top call_show_stack
```

```
statetype == stack num
init_state = empty

call_new :: statetype->[[char]]->(statetype,[[char]])
call_new st [answer]
    = (newst,out), if ((answer = "Y") \/ (answer = "y"))
    = (st,outerr), otherwise
      where
      newst = empty
      out = [line]
      outerr = [errline]
      line = "A new stack has been created."
      errline = "No new stack has been created"

call_push :: statetype -> [[char]] -> (statetype,[[char]])
call_push st [arg]
    = (newst,out)
      where
      x = numval arg
      newst = push x st
      out = [line1]
      line1 = arg ++ " has been pushed into the stack"

call_top :: statetype -> [[char]] -> (statetype, [[char]])
call_top st [arg]
    = (st,out), if pre_top st
    = error "Can't take the top of an empty stack", otherwise
      where
      out = [line1]
      line1 = "The top of the stack is " ++ show tp
      tp = st_top st


call_pop :: statetype -> [[char]] -> (statetype, [[char]])
call_pop st [arg]
    = (newst, out), if pre_pop st
    = error "Can't pop an empty stack", otherwise
      where
      newst = pop st
      out = [line1]
      line1 = "The popped stack is " ++ showstack show newst


call_show_stack :: statetype -> [[char]] -> (statetype,[[char]])
call_show_stack st [answer]
    = (st,out), if ((answer = "Y") \/ (answer = "y"))
    = (st,outerr), otherwise
      where
      out = [line1]++[otherLines]
```

```
line1 = "Elements of the Stack:\n"
otherLines = showstack show st
outerr = ["No output"]
```

A.3 Menu and Data Screen Design

Use Miranda to compile the toolkit driver program for creating menu and data entry screens and to link them together. As this is a large program you will need to increase the heap size to over 300,000 cells:

1. Increase heap size:

   ```
   miranda /heap 400000
   ```

2. Compile the toolkit driver (*driver.m*):

   ```
   miranda /f driver.m
   ```

3. Now execute the interactive function (`toolkit`) generated in store by it:

   ```
   miranda toolkit
   ```

A.3.1 Menu and Data Entry Screens

1. Use the menu generation option to create menu screens, and save them (using ^S option, and supplying a file name, $< filename >$ to the prompt).

   a) Give it the same name as the ADT (e.g. *stack* – without extension *.m*).

   b) The menu screens information will be saved in a file $PROJ- < filename > .m$ (e.g., $PROJ-$ *stack.m*).

   c) If an earlier version of this file already exists, it is best to remove it first (the update functions have bugs!).

2. Use the data entry screen generation option to create data entry screens, and save them under the same name as the menu screens (i.e., using ^S option, and supplying the same file name $< filename >$ to the prompt, as in the menu case above.

   a) Again, do not include *.m* extension.

   b) You will also be prompted to provide names for each of the data entry screens as you save them. Make these names meaningful, preferably using the ADT function names (e.g., data entry screen for *push* as *push*).

   c) The data entry screens information will be saved in the same file $PROJ- < filename > .m$, as above (e.g., $PROJ - stack.m$).

3. It is best to do both these operations in one session, even though it is possible to add the data entry screens information to an already existing $PROJ- < filename > .m$. In that case you need to ensure that no data entry screens information from previous session is already contained in it. Again the update functions have bugs!

4. By now the $PROJ- < filename > .m$ will contain both menu and data entry screen information.

A.3.2 Linking

1. The third option of toolkit is used for linking these screens.

   a) On selecting this option you will be prompted to supply the name of the project file. This is just the file where the information on menus and data entry screens were saved, $PROJ- < filename > .m$ (e.g., $PROJ - stack.m$). Give just the $< filename >$ (without the prefixed $PROJ-$ and the extension $.m$). In our example, it will be just *stack*.

   b) The toolkit then shows all the menu and data entry screen information numbered in the order they were created. You will be prompted to supply the name of the ADT file. Supply the ADT file name, again without extension $.m$ (*stack*, in our example).

   c) Now, follow the prompts to supply the information on linkage and the associated interface functions. These must be the same as the ones given in the interface file created earlier (e.g., $int - stack.m$ for our STACK example). For example, for our example the function associated with the data entry screen (called *push* above) will be *call_push* (defined in the $int - stack.m$ file).

2. There is no need to save the file of linkages, this is done automatically, when all the required information as requested by the prompts have been supplied. The file is saved with a prefix $link-$ to the project file, i.e., $link - PROJ- < filename > .m$ ($link - PROJ - stack.m$, for our example).

3. When all the associated functions have been supplied, use ^X to exit from toolkit.

A.4 Animation

1. Using Miranda compile the animator (*animator.m*):

   ```
   miranda /f animator.m
   ```

2. Execute the interactive animator function (*animator*) produced in store:

   ```
   miranda animator
   ```

3. The top level menu will appear. Use it to have an animation session. For our example, stack, we could create a new stack, push numbers, view top element, pop the top element or show the whole stack.

4. Exit the animator by ^X. Alas, there is a (yet another?) bug! The terminal does not return to normal mode!! Give the command to return to echo and canonical mode (! is shell escape for Miranda):

   ```
   miranda !stty echo icanon
   ```