

- | | |
|------------------------------------|--------------------------------------|
| 1. How to use the manual system | 20. Algebraic types |
| 2. About the name "Miranda" | 21. Abstract types |
| 3. About this release | 22. Placeholder types |
| 4. The Miranda command interpreter | 23. The special function <u>show</u> |
| 5. Brief summary of main commands | 24. Formal syntax of Miranda scripts |
| 6. List of remaining commands | 25. Comments on syntax |
| 7. Expressions | 26. Miranda lexical syntax |
| 8. Operators | 27. The library mechanism |
| 9. Operator sections | 28. The standard environment |
| 10. Identifiers | 29. Literate scripts |
| 11. Literals | 30. Some hints on Miranda style |
| 12. Tokenisation and layout | 31. UNIX/Miranda system interface |
| 13. Iterative expressions | 32. --> CHANGES <<-- |
| 14. Scripts, overview | 33. Licensing information |
| 15. Definitions | 34. Bug reports |
| 16. Pattern matching | 35. Notice |
| 17. Compiler directives | |
| 18. Basic type structure | 99. Create a printout of the manual |
| 19. Type synonyms | 100. An Overview of Miranda (paper) |

printout of online manual pages
manual pages last revised 16 Apr 1991

```

-----
| Copyright 1990, Research Software Limited.  Holders of a Miranda(tm) |
| release two software license are permitted to copy this document, or |
| any portion of it, as necessary for licensed use of the software, |
| provided this copyright notice and statement of permission are |
| included, and provided that such copies are for licensee's internal |
| use only and not for distribution outside the license site. |
-----

```

.....

1 How to use the Miranda on-line reference manual:

.....

The manual is menu driven, and is a separate subsystem that can be invoked from the Miranda command interpreter. To access the manual from Miranda, type

```
    /man
```

followed by return. To access the manual directly from UNIX, without entering Miranda first, type the line

```
    mira -man
```

as a UNIX command.

On entering the manual system a contents page is displayed listing numbered sections. In response to the question at the bottom of the page you should type the number of the section you wish to see (and, as always, your response should be entered by pressing the return key). The contents of that section are then displayed to you. When you are ready to leave the manual, press return, instead of a section number, in response to the question at the foot of the contents page - typing q (for "quit") will have the same effect.

If the section is more than one page long, you should press the space bar when you have finished reading a page, in order to see the next page (or press return to see one more line). At the end of the section, you may press return to go back to the contents page.

The manual is organised in short sections, each at most two or three pages (i.e. screenfulls) long. Where a section would otherwise be too long to fit in with this philosophy, the manual is organised recursively and will display a subsidiary contents page, with a list of numbered subsections, which you may read, as before, by typing the appropriate number. To return to the main menu from a submenu, press the return key, instead of a selection number, in response to the question at the bottom of the page.

The manual is intended to provide online documentation of the Miranda programming language and the command system in which it is embedded. It is not intended to be a tutorial on functional programming. It does not provide information about the UNIX operating system, for which separate documentation is available.

Summary of manual behaviour:

Whenever the manual system prompts the user for input - by asking for "next selection" - the complete repertoire of possible responses is:

```
q                quit the manual
<return>        exit one level of menu structure
                  (i.e. if at foot of section, return to
menu
                  if at submenu, return to master
menu
                  if at main menu, quit the manual)
<number>        display section from current contents page
.                (dot) same again, i.e. redisplay last requested
section
+                display next section in numerical order
-                display previous section
!command        temporary escape to UNIX, executes shell command
!!              repeat last shell command
```

Summary of the behaviour of 'more':

Individual sections of the manual are displayed to you using the UNIX program 'more' (unless you have an environment variable VIEWER set to something else(*)). The 'more' program has a large number of features, most of which are not relevant here (see UNIX manual page for full information). The 'more' program has its own prompt - the responses you can give to 'more' usually include

```
[SPACE]          display next screenful
[return]        display one more line
q                (quit) cease showing me this file
'                (single quote) go back to beginning of file being displayed
```

b (back) scroll backwards by one screenful

(*) See the section on Environment Variables under UNIX/Miranda system interface.

.....

2 About the name 'Miranda'

.....

The word 'Miranda' is not an acronym. It is a proper name (like 'ADA').

"Miranda (f). From the Latin meaning 'to be admired'. This name was first used by Shakespeare for the heroine of 'The Tempest', a young girl blessed with many admirable qualities. Like other unusual Shakespearean names it has been used quite frequently in the 20th century."

- Collins Dictionary of First Names,
William Collins and Son Ltd, London, 1967

"O wonder! How many goodly creatures are there here! O brave new world."

- 'The Tempest' by William Shakespeare
(from a speech by Miranda, Act 5, Scene 1)

Important Note. When used as the name of a functional programming system, 'Miranda' is a trademark (tm) of Research Software Ltd.

Note that only the first letter of Miranda is upper case - it should never be written all in capitals.

.....

3 About this release of Miranda(October 1989)

.....

This is release 2 of the Miranda system. It has the following main additions, compared with release 1

the library directive %free (for parametrised scripts)
unbounded precision integers
local polymorphism

Both compiler and run time system are also significantly faster (overall improvement about 25%, but much greater in some specific cases) and all reported bugs have been fixed. Many thanks to the people who sent in bug reports from release one. Miranda is now running at over three hundred sites and is becoming accepted as one of the standard vehicles for functional programming.

The opportunity has also been taken to tidy up the language definition by removing two features that with the benefit of hindsight now seem questionable - namely 'laws' and strictness annotations on algebraic data types. The compiler still accepts these (for a transitional period) but they now attract an 'obsolete feature' warning and will cease to be supported at the next release. One other minor change has

been made, for compatibility with Bird and Wadler - introduction of an (optional) keyword 'if' before guards.

All the differences between release 1 and release 2 of Miranda are listed in the CHANGES section of the manual. If you are new user you can safely ignore the CHANGES section.

```
-----  
| Users with existing scripts, prepared under release one, should read |  
| part 1 of the ``CHANGES'' section carefully to see if they will need |  
| to make any alterations to their scripts to get them to compile |  
| under the new release. |  
-----
```

The online manual pages are primarily intended to document the system at the level required by an expert user (meaning someone who already knows quite a lot about programming and programming languages in general, and has some previous experience of functional programming in particular.) There is a certain amount of tutorial material, but if you are a beginner to functional programming you may find parts of the manual hard to follow, and will need to seek help elsewhere.

The following paper gives a useful summary of the features of Miranda:
D. A. Turner "An Overview of Miranda", SIGPLAN Notices, December 1986.
A copy of this paper is included in the manual pages.

The following introductory text book is recommended for use with Miranda. It is a tutorial text aimed at undergraduates learning a functional language as their first programming language, with numerous well chosen examples and exercises. It is equally suitable for a graduate course on functional programming. The notation used in the book is closely based on Miranda(*):-

"An Introduction to Functional Programming" by Richard Bird and Philip Wadler, published by Prentice Hall International, March 1988.
ISBN 13-484189-1 (Cloth)
ISBN 13-484197-2 (Paper)

All the standard functions listed in appendix B of Bird and Wadler are present in the Miranda standard environment with the same names and the same definitions. (Note in particular that the Miranda function 'foldl' is now the same as that in Bird and Wadler, removing what used to be a source of confusion.)

```
-----  
(* ) Apart from some minor typographical considerations, the notation used in Bird and Wadler is a subset of Miranda. The main typographical points to note are (i) the book uses letters of the greek alphabet for type variables - in Miranda type variables are written * ** *** ...; (ii) the book uses a mathematical font for the arithmetic and logical operators - Miranda, for obvious reasons, uses the ascii character set - see section 8 of this manual for a list of operators used in Miranda.
```

.....

4 The Miranda command interpreter

.....

The Miranda system is invoked from unix by the command

```
mira [file]
```

where `file' (an optional parameter) is the pathname of a file containing a Miranda script (i.e. a set of definitions). If no file is given a default filename `script.m' is assumed. The parameter of the mira command (or script.m by default) becomes your current script, during the ensuing Miranda session.

Note that a file containing a Miranda script must have a name ending in `.m' and the `mira' command will add this extension automatically if absent. That is if you say `mira thing', it will be interpreted as a reference to the file `thing.m'. It is a UNIX convention (which often turns out to be useful) that files containing program sources should have names indicating which language they are written in.

The set of names in scope at any time are those of the current script, together with the names of the standard environment, which are always in scope.

The prompt `Miranda' indicates that you are talking to the Miranda command interpreter. This activity is called a Miranda ``session''. Each command should be typed on a single line, following the prompt, and is entered by hitting return. Any command not beginning with one of the special characters `/', `?', or `!' is assumed to be an expression to be evaluated. The following commands are available during a session.

exp

Any Miranda expression typed on a line by itself is evaluated, and the value is printed on the screen. If the value is of type [char] it is printed literally, otherwise the special function `_ s_ h_ o_ w' is applied to

it to convert it to printable form. Example

```
Miranda sum[1..100]
```

```
5050 (response)
```

There is a special symbol \$\$ which is always set to the last expression evaluated. So after the above command \$\$ will have the value 5050, and this can be used in the next expression - e.g. `\$\$/2' will produce the response 2525 (and the value of `\$\$' is now 2525).

exp &> pathname

A background process is set up to evaluate exp, and the resulting output (including error messages, if any) sent to the designated file.

exp &>> pathname

As above, except that the output is appended to the designated file, instead of replacing its previous contents.

exp::

Print the type of the expression (instead of the value). Useful for investigating the type structure of a script.

?

Lists all identifiers currently in scope, grouped by file of origin, starting with the standard environment.

?identifier(s)

Gives more information about any identifier defined in the current environment (namely its type and the name of the file in which it is defined). This command will also accept a list of identifiers, separated by spaces.

!command

Execute any UNIX shell command.

!!

Repeat last shell command.

Note that the character '%' can be used in any Miranda session command, including a '! ' command, as an abbreviation for the pathname of the current script. So for example

!wc %

does a word count on the current script. (If for some reason you need to include a literal % character in a command, you can escape it with a preceding backslash.)

All the remaining commands begin with `/' . Each of the following commands can be abbreviated to its first letter.

/edit (also /e)

Edit the current script. Calls up the screen editor `vi'. (This is the default, but another editor can be substituted - see manual section on UNIX/Miranda system interface). On quitting the editor, if changes have been made to any relevant source file, the Miranda system automatically recompiles the current script before returning you to the Miranda command interpreter. This check is performed after all /edit commands and also after each shell escape.

[An alternative mode of operation, possible on a windowing system, is to keep an editor window and a Miranda session window both open, rather than using the /e command. In this case you should set an environment variable RECHECKMIRA (to any non-empty string) - this will cause the Miranda system to check for source file updates before every interaction, instead of only after /e and ! commands.]

/edit pathname (also /e pathname)

Edit arbitrary script. Note that the pathname should end in `.m' and that this will be added if missing.

Note by the way that (if you are using a Berkeley UNIX, or a derivative) Miranda understands the cshell's ~ convention in pathnames. That is ~/file means file in your home directory, and ~jack/file means file in jack's home directory. This doesn't work under system 5 (sorry).

/file (also /f)

Print the name of file containing the current script.

/file pathname (also /f pathname)

Change to new current script. Equivalent to quitting the Miranda system and reinvoking it with a new sourcefile. Like /e, /f adds ".m"

to the end of the filename if you omit it.

Important special case - reselecting the current script, eg by saying
/f %

forces the current script to be RECOMPILED - this can be useful if script has errors and you wish to see the error messages again.

/help (also /h)
Display summary of main available commands.

/man (also /m)
Enter online manual system.

/quit (also /q)
Quit the Miranda system. Typing the end of file character (control-D) also has this effect.

Finally note that \$- and \$+ are allowed as notations for the standard input in Miranda expressions. The standard input as a list of characters is denoted by `\$\$-'. As a simple example, evaluating the expression

reverse \$-
causes everything typed at the keyboard upto the next control-D to be echoed backwards.

The notation `\$\$+' also denotes the standard input, but as a sequence of Miranda expressions (one per line), and returns their values. For example

sum \$+
reads a sequence of numeric expressions from the standard input, and returns the sum of their values. See the manual section on reading with interpretation (under UNIX/Miranda system interface) for further information.

.....

5 SUMMARY OF MAIN AVAILABLE COMMANDS:

.....

exp	evaluate a Miranda expression
exp &> file	send value of expression to file (background process)
exp &>> file	append value of expression to file (background process)
exp::	print type of exp
?	list all identifiers currently in scope
?identifier(s)	give more information about identifier(s)
!command	execute any UNIX shell command
!!	repeat last shell command
/edit	edit current script (default editor=vi)
/edit filename	edit filename
/file	display filename of current script
/file filename	change to new current script
/help	display this command summary
/man	ENTER ONLINE REFERENCE MANUAL (menu)

driven)
 /quit quit the Miranda system
 Abbreviations:
 each "/" command can be abbreviated to its first letter - /e /f /h /m /q
 % is shorthand for the name of the current script (in commands)
 Special case - note that `/f %' forces recompilation of current script
 \$\$ is shorthand for the last expression evaluated (in expressions)
 For a list of auxiliary commands say /aux

.....

6 LIST OF REMAINING COMMANDS:

.....

/aux	print this list of auxiliary commands
/cd [dirname]	change directory (defaults to home directory)
/count (/nocount)	switch on (off) statistics following each evaluation
/dic	report size of dictionary for storing names etc
/editor	report name of resident editor (used by /e command)
/editor PROG	change resident editor to PROG
/find id(s)	like `?ids' but look under original names of aliases
/gc (/nogc)	switch on (off) garbage collection reports
/heap	report size of heap
/heap SIZE	set heap to SIZE cells
/hush (/nohush)	switch off (on) prompts and other unnecessary feedback
/list (/nolist)	(dis)enable listing of script to screen when compiling
/miralib	report absolute pathname of the directory miralib
/settings	print current settings of flags
/strictif	enforce `if' in guards (/nostrictif to cancel)
/version	print version information
??identifier	open source file at definition of identifier
...	lines beginning in ` ' are ignored (comment facility)

.....

7 Expressions

.....

In Miranda an expression denotes a value. Expressions occur in two contexts, but have the same(*) syntax in both situations. First, the basic mode of operation of the Miranda command interpreter is that it evaluates expressions typed at the terminal (these are called `command-level expressions'). Second, expressions are an important syntactic component of scripts (because scripts are collections of definitions, and the right hand sides of definitions are composed of expressions).

Expressions typed at the terminal are volatile - only by being made part of a script can an expression be saved for future use.

An expression is either simple, or a function application, or an operator expression, or an operator.

A simple expression is one of the following:

Identifier, e.g. `x` (see separate manual entry)

Literal, e.g. `136` (see separate manual entry)

An operator section (see separate manual entry)

The keyword 'show' or 'readvals' (see separate manual entries)

A list, such as `[1,5,7,9]` or a 'dotdot' list or a list comprehension (see separate manual entry on iterative expressions).

A tuple, such as `(True,"green",37)`. Tuples differ from lists in that they can have components of mixed type. They are always enclosed in round parentheses. The empty tuple, which has no components, is written `()`. Otherwise, a tuple must have at least two components - there is no concept of a one-tuple. Tuples cannot be subscripted, but their components can be extracted by pattern matching. Since there is no concept of a one-tuple, the use of parentheses for grouping does not conflict with their use for tuple formation.

Any expression enclosed in parentheses is a simple expression.

Function application is denoted by juxtaposition, and is left associative, so e.g.

f a b

denotes a curried function application of two arguments. (So f is really a function of one argument whose result is another function - thus `f a b` is actually parsed as `(f a) b` - the advantage of this arrangement is that `f a` has a meaning in its own right, it is a partially applied version of f.)

Operator expressions

e.g. `5*x-17`

There are a variety of infix and prefix operators, of various binding powers (see manual entry on operators). Note that function application is more binding than any operator.

An operator on its own can be used as the name of the corresponding function, as shown in the following examples

arith2_ops = [+,-,*,/,div,mod,^]

sum = foldr (+) 0

both of which are legal definitions. Note that when an operator is passed as a parameter it needs to be parenthesised, to force the correct parse. An ambiguity arises in the case of `-` which has two meanings as an operator (infix and prefix) - the convention is that `-` occurring alone always refers to the infix (i.e. dyadic) case. The name `neg` is provided for the unary minus function, as part of the standard environment.

A formal syntax for expressions can be found in the manual section called 'Formal syntax of Miranda scripts'.

(*) Note: There is one exception to the rule that command level

expressions have the same syntax as expressions inside scripts - the notation `\$\$', meaning the last expression evaluated, is allowed only in command level expressions.

.....

8 Operators and their binding powers

.....

Here is a list of all prefix and infix operators, in order of increasing binding power. Operators given on the same line are of the same binding power. Prefix operators are identified as such in the comments - all others are infix.

operator	comments
: ++ --	right
associative \ associative	
&	
associative	
~	
prefix	
> >= = ~= <= <	continued relations
allowed, eg 0<x<=10	
+ -	
left associative	
-	
prefix	
* / _ d _ i _ v _ m _ o _	
d	left associative
^	
right associative	
.	
associative	
#	
prefix	
!	
left associative	
\$identifier \$IDENTIFIER	right associative

Brief explanation of each operator:

```

:          prefix an element to a list, type *->[*]->[*]
++ --    list concatenation, list subtraction, both of type [*]->[*]->[*]
          A formal definition of list subtraction is given below.
\  
&      logical `or', `and', both of type bool->bool->bool
~       logical negation, type bool->bool
> >= = ~= <= <
          comparison operators, all of type *->*->bool
          Note that there is an ordering defined on every (non-
          function)type. In the case of numbers, characters and
          strings the order is as you would expect, on other types it
          as an arbitrary but reproducible ordering. Equality on
          structured data is a test for isomorphism. (i.e. in LISP
  
```

terms it is "EQUAL" not "EQ"). It is an error to test functions for equality or order.

+ - plus, minus, type num->num->num

- unary minus, type num->num

Note that in Miranda unary minus binds less tightly than the multiplication and division operators. This is the usual algebraic convention, but is different from PASCAL.

* / div mod

times, divide, integer divide, integer remainder, all of type num->num->num

`/' can be applied to integers or fractional numbers, and always gives a fractional result, so eg 6/2 is 3.0 div and mod can only be applied to integers and give integer results, eg 7 div 2 is 3, 7 mod 2 is 1.

div and mod obey the following laws, for a b any integers with $b \neq 0$

(i) $b * (a \text{ div } b) + a \text{ mod } b = a$

(ii) if $b > 0$ then $0 \leq a \text{ mod } b < b$

if $b < 0$ then $b < a \text{ mod } b \leq 0$

^ `to the power of', type num->num->num

. function composition, type (**->***)->(*->**)->*->***

length of list, type [*]->num

! list subscripting, type [*]->num->*

note that the first element of a non-empty list x is x!0 and the last element is x!(#x-1)

\$identifier \$IDENTIFIER

do-it-yourself infix, `a \$f b' is equivalent in all contexts to `f a b'. Also works for constructors of two or more arguments.

Note on list subtraction

Here is a formal definition of the `--' operator in Miranda. It is defined using an auxiliary function `remove' which removes the first occurrence (if any) of a given item from a list.

```

x -- [] = x
x -- (b:y) = (remove b x) -- y
remove b [] = []
remove b (a:x) = x,           if a=b
                  = a:remove b x, otherwise

```

.....

9 Sections

.....

An infix operator can be partially applied, by supplying it with only one of its operands, the result being a function of one argument. Such expressions are always parenthesised, to avoid ambiguity, and are called `sections'. Two different kinds of sections (called presections and postsections) are possible since either the first or the second operand can be supplied.

An example of a presection is

(1/)

which denotes the reciprocal function. An example of a postsection is

(/3)

which gives a concise notation for the 'divide by three' function. Note that both of these examples are functions of type (num->num). With one exception (see below) sections can be formed using any infix operator. Further examples are (0:) which is a function for prefixing lists of numbers with zero, and (^2) which is the square function.

Sections may be regarded as the analogue of currying for infix operators. They are a minor syntactic convenience, and do not really add any power to the language, since any function denoted in this way could have been introduced by explicit definition. For the first two examples given above we could have written, say

```
reciprocal y = 1/y
divide_by_three x = x/3
```

and then used the function names, although this would have been somewhat more verbose.

To summarise the conventions for infixes, taking infix division as an example, note that the following expressions are all equivalent.

```
a / b
(/) a b
(a /) b
(/ b) a
```

The usual rules about operator precedence (see manual section on operators) apply to sections. For example we can write (a*b+) but not (a+b*), because '*' is more binding than '+'. The latter example should have been written ((a+b)*). As always when writing complicated expressions, if there is any possibility of ambiguity it is better to put in extra parentheses.

Special case

It is not possible to form a postsection in infix minus, because of a conflict of meaning with unary minus. For example:

(-b)

is a parenthesised occurrence of negative b, not a section. As a way round this there is a function 'subtract' in the standard environment with the definition:- subtract x y = y - x. This is a normal curried function, so we can write (subtract b) to get the function that subtracts b from things.

Presections in infix minus, such as (a-), cause no ambiguity. There are no problems with infix plus because Miranda does not have a unary plus operator.

Acknowledgement:

The idea of sections is due to Richard Bird (of Oxford University Programming Research Group) and David Wile (of USC Information Sciences Institute).

.....

10 Identifiers

.....

An identifier consists of a letter followed by zero or more additional

characters which may be letters digits or occurrences of `_` or `'` (underscore and single quote) Examples:

```
x yellow p1ld y' GROSS_INCOME
```

Note that both upper and lower case letters are allowed, and they are treated as different, so `x` and `X` are not the same identifier. There is no restriction on the length of identifiers, and all the characters are significant in deciding whether two identifiers are the same. Identifiers fall into two different classes (called in the formal syntax ``identifier'` and ``IDENTIFIER'`) depending on whether their initial letter is upper or lower case.

Identifiers are used for three different purposes in Miranda - (i) as variables, i.e. names for values (note that the names of functions are also considered to be variables), (ii) as typenames, such as ``bool'` and ``char'`, and (iii) as constructors (see section on algebraic types). The names of constructors must begin with an upper case letter, while variables and typenames must begin with a lower case letter.

Reserved words - the following are special symbols of the Miranda language.

abstype div if mod otherwise readvals show type where with
(10)

These are often shown as underlined (or bold) in published documents, but in programs they are typed as ordinary lower case words (which means that these words cannot be used as identifiers).

Predefined identifiers

The following identifiers are normally predefined, and thus always in scope. They constitute the ``standard environment'`. They are defined in the script `"stdenv.m"`, contained in the directory `/usr/lib/miralib`.

(a) predefined typenames

```
bool char num sys_message
```

(b) predefined constructors

```
False True :: bool
```

```
Appendfile Closefile Exit Stderr Stdout System Tofile :: sys_message
```

(c) predefined variables

```
abs and arctan cjustify code concat const converse cos decode digit  
drop dropwhile e entier error exp filemode filter foldl foldl1 foldr  
foldr1 force fst getenv hd hugenum id index init integer iterate  
last lay layn letter limit lines ljustify log log10 map map2 max  
max2 member merge min min2 mkset neg numval or pi postfix product  
read rep repeat reverse rjustify scan seq showfloat shownum  
showscaled sin snd sort spaces sqrt subtract sum system take  
takewhile tinynum tl transpose undef until zip2 zip3 zip4 zip5 zip6  
zip (88)
```

See manual entry ``Standard environment'` for more detailed information.

[NB There ought to be a way of suppressing or modifying the inclusion of the standard environment, in case you want to use one of these identifiers for another purpose - a mechanism for doing this will be

provided in later releases.]

.....

11 Literals

.....

There are three types of literal constant - numerals, character constants, and string constants.

Numerals are written in the following style

12 5237563 24.6 4.5e13 0.63e-6

A numeral containing either a decimal point or a scale factor (`. ' or `e') is fractional, and is stored internally as double precision floating point (accuracy approximately 17 decimal digits). Integers are held in a different internal representation, and have unbounded precision.

The two kinds of number, integer and fractional, are both of type `num', as far as the type-checker is concerned, and can be freely mixed in calculations. There is automatic conversion from integer to fractional when required, but not in the opposite direction. To convert from fractional to integer, use `entier' (see standard environment).

Negative numbers are denoted by applying the prefix `- ' operator to a numeral, thus:

-12 -4.5e13

but note that the notation -12 is an expression, not a literal, so if you wish to apply a function to it, you must write f (-12), not f -12, which would be read as an attempt to subtract 12 from f.

Character constants are written using single quotes, thus

'a' '0' '\n'

String constants are written using double quotes, thus

"hello dolly" "" "\n\n\n"

Escape conventions in character and string constants are as in `C', using the backslash character.

\n newline
\t tab
\f formfeed
\r carriage return
\b backspace
\ the backslash character itself
\ ' single quote
\ " double quote
\ddd up to three decimal digits, arbitrary ascii

character code

Note that literal newlines are not allowed inside character or string quotes, unless escaped by backslash, in which case the newline is ignored. Thus

"hello \
dolly"

means the same as "hello dolly".

These conventions are exactly as in `C`, except that we use decimal rather than octal for numeric specification of an arbitrary ascii character code (the use of octal numbers in computing now seems a curious anachronism which there is no need to perpetuate).

.....

12 Tokenisation and layout

.....

A Miranda script or expression is regarded as being composed of tokens, separated by layout.

A token is one of the following - an identifier, a literal, a type variable, or a delimiter. Identifiers and literals each have their own manual section. A type variable is a sequence of one or more stars, thus * ** *** etc. (see basic type structure). Delimiters are the miscellaneous symbols, such as operators, parentheses, and keywords. A formal definition of the syntax of tokens, including a list of all the delimiters is given under `Miranda lexical syntax`.

RULES ABOUT LAYOUT

Layout consists of white space characters (spaces, tabs, newlines and formfeeds), and comments. A comment consists of a pair of adjacent vertical bars, together with all the text to the right of the bars on the same line. Thus

```
|| this is a comment
```

Layout is not permitted inside tokens (except in char and string constants, where it is significant) but may be inserted freely between tokens to make scripts more readable. Layout is ignored by the compiler except in two respects:

1) At least one space (or other layout characters) must be present between two tokens that would otherwise form an instance of a single larger token. For example in

```
f 19 'b'
```

we have a function, f, applied to a number and a character, but if we were to omit the two intervening spaces, the compiler would read this as a single six-character identifier, because both digits and single-quotes are legal characters in an identifier. (Where it is not required to force the correct tokenisation, or because of the offside rule, see below, the presence of layout between tokens is optional.)

2) Certain syntactic objects (roughly, the right hand sides of declarations -- for an exact account see those entities followed by a `(;)` in the formal syntax) obey Landin's _ o_ f_ f_ s_ i_ d_ e _ r_ u_ l_ e [Landin 1966].

This requires that every token of the object lie either directly below or to the right of its first token. A token which breaks this rule is said to be `offside' with respect to that object and terminates its parse. For example in

```
x = 2 < a
y = f q
```

the 'y' is offside with respect to the right hand side of the definition of 'x' (because it is to the left of the initial '2'). In such a case

the trailing semicolon may be omitted from the right hand side of the equation for x.

It is because of the offside rule that Miranda scripts do not normally contain explicit semicolons as terminators for definitions. The same rule enables the compiler to determine the scopes of nested where's by looking at their indentation levels. For example in

```
f x = g y z
      where
      y = (x+1)*(x-1)
      z = p x (q y)
g r = groo (r+1)
```

it is the offside rule which makes it clear that the definition of 'g' is not local to the right hand side of the definition of 'f', but those of 'y' and 'z' are.

It is always possible to terminate a right hand side by an EXPLICIT semicolon, instead of relying on the offside rule. For example the above script could be written all in one line, as

```
f x = g y z where y = (x+1)*(x-1); z = p x (q y);; g r = groo (r+1);
```

Notice that we need TWO semicolons after the definition of z - the first terminates the rhs of the definition of 'z', and the second terminates the larger rhs of which it is a part, namely that of the definition of 'f'. If we put only one semicolon at this point, the definition of 'g' would be local to that of 'f'.

This example should convince the reader that code using layout information to show the block structure is much more readable, and this is the normal practise.

[Reference P.J. Landin "The Next 700 Programming Languages", CACM vol 9 pp157-165 (March 1966).]

Note that an additional comment convention applies in scripts whose first character is a '>'. See separate manual entry on 'literate scripts'.

.....

13 Iterative expressions

.....

1. Dotdot expression
2. List comprehensions
3. Diagonalising list comprehensions

.....

13/1 Dotdot notation

.....

The following abbreviations are provided for denoting lists, of type [num], whose members form a finite or infinite arithmetic series. Let 'a', 'b', 'c' stand for arbitrary numeric expressions.

```

[a..b]          list of numbers from a to b
inclusive, interval = 1
[a..]          infinite list starting at a and increasing by 1
[a,b..c]       arithmetic series, first member a, second
member b,
               last member not greater than c (if
b-a non-negative)
               or not less than c (if b-a negative).
[a,b..]        infinite series starting at a, interval = (b-a)

```

So the notation [1..10] has as value the list [1,2,3,4,5,6,7,8,9,10]. Here are some more examples

```

nats = [0..]
evens = [0,2..]
odds_less_than_100 = [1,3..99]
neg_odds = [-1,-3..]
tenths = [1.0,1.1 .. 2.0]

```

.....

13/2 List comprehensions

.....

```
[exp | qualifiers]
```

List of all 'exp' such that 'qualifiers'. If there are two or more qualifiers they are separated by semicolons. Each qualifier is either a generator, of which the allowed forms are

```

pattern-list <- exp
(first form)

```

```

pattern <- exp, exp ..
(second form)

```

or else a filter, which is a boolean expression restricting the range of the variables introduced by preceding generators. The variables introduced on the left of each '<-' are local to the list comprehension.

Some examples

```

sqs = [ n*n | n<-[1..] ]
factors n = [ r | r<-[1..n div 2]; n mod r = 0 ]
knights_moves [i,j] = [ [i+a,j+b] | a,b<-[-2..2];
a^2+b^2=5 ]

```

Notice that a list of variables on the lhs of a '<-' is shorthand for multiple generators, e.g. 'i,j<-thing' expands to 'i<-thing; j<-thing'.

The variables introduced by the generators come into scope from left to right, so later generators can make use of variables introduced by earlier ones. An example of this is shown by the following definition of a function for generating all the permutations of a given list.

```
perms [] = [[]]
perms x = [ a:p | a<-x; p<-perms(x--[a]) ]
```

The second form of generator allows the construction of lists from arbitrary recurrence relations, thus

```
x <- a, f x ..
```

causes x to assume in turn the values `a', `f a', `f(f a)', etc.

An example of its use is in the following definition of the fibonacci series

```
fibs = [ a | (a,b) <- (1,1), (b,a+b) .. ]
```

Another example is given by the following expression which lists the powers of two

```
[ n | n <- 1, 2*n .. ]
```

The order of enumeration of a list comprehension with multiple generators is like that of nested for-loops, with the rightmost generator as the innermost loop. For example the value of the comprehension [f x y | x<-[1..4]; y<-[1..4]] is

```
[ f 1 1, f 1 2, f 1 3, f 1 4, f 2 1, f 2 2, f 2 3, f 2 4,
  f 3 1, f 3 2, f 3 3, f 3 4, f 4 1, f 4 2, f 4 3, f 4 4 ]
```

As a consequence of this order of enumeration of multiple generators, if any generator other than the first (leftmost) is infinite, some combinations of values will never be reached in the enumeration. To overcome this a second, `_d_i_a_g_o_n_a_l_i_s_i_n_g`, form of list comprehension is provided (see separate manual section).

Note that list comprehensions do NOT remove duplicates from the result list. To remove duplicates from a list, apply the standard function `mkset'.

```
.....
```

13/3 Diagonalising list comprehensions

```
.....
```

```
[ exp // qualifiers ]
```

Syntax and scope rules exactly as for standard list comprehensions, the only difference being the use of `//' in place of the vertical bar. The order of enumeration of the generators is such that it is guaranteed that every possible combination of values will be reached eventually. The diagonalisation algorithm used is "fair" in the sense that it gives equal priority to all of the generators.

For example the value of [f x y//x<-[1..4]; y<-[1..4]] is

```
[ f 1 1, f 1 2, f 2 1, f 1 3, f 2 2, f 3 1, f 1 4, f 2 3,
  f 3 2, f 4 1, f 2 4, f 3 3, f 4 2, f 3 4, f 4 3, f 4 4 ]
```

The algorithm used is "Cantor diagonalisation" - imagine the possible combinations of values from the two generators laid out in a (perhaps infinite) rectangular array, and traverse each diagonal in turn starting from the origin. The appropriate higher-dimensional analogue of this algorithm is used for the case of a list comprehension with three or more generators.

As an example of an enumeration that could not be defined at all using a standard list comprehension, because of the presence of several infinite generators, here is a definition of the list of all pythagorean triangles (right-angled triangles with integer sides)

```
pyths = [(a,b,c)//a,b,c<-[1..];a^2+b^2=c^2]
```

In the case that there is only one generator, the use of `//` instead of `|` makes no difference to the meaning of the list comprehension.

.....

14 Scripts

.....

In Miranda the script is the persistent entity that is saved from session to session (i.e. it plays the role of what is called a program in conventional languages). Associated with a Miranda session at any given time is a single current script, identified by a UNIX pathname ending in `.m`.

A script is a collection of declarations, establishing an environment in which you wish to evaluate expressions. The order of the declarations in a script is not significant - for example there is no requirement that an identifier be defined before it is used.

An identifier may not have more than one top-level binding in a given script.

Here are the kinds of declaration that can occur in a script:

1) a definition (of a function, data structure etc. - see manual entry `definitions' for more details). Example

```
fac n = product[1..n]
```

2) a specification of the type of one or more identifiers, of the form
var-list :: <type>

Example

```
fac :: num->num
```

See 'Basic type structure' for an account of possible types. Note that these type specifications are normally optional, since the compiler is able to deduce them from the definitions of the corresponding identifiers. It is however possible to introduce an identifier by means of a type specification only, without giving it a defining equation (such identifiers are said to be `specified but not defined' and are useful in program development). A special case of this is the introduction of an otherwise undefined typename - see separate manual entry on `placeholder types'.

3) the definition of a user defined type - these are of three kinds, synonyms, algebraic types, and abstract types (see separate manual entry on each).

4) a library directive (%export, %include or %free) these are used specify the interfaces between separately compiled scripts - see separate manual entry on the library mechanism.

There is a manual entry giving the formal syntax of Miranda scripts.

Note

A directory called `ex' (short for `examples') containing a collection of example scripts is supplied with the Miranda system, and will be found under the `miralib' directory (usually kept at /usr/lib/miralib - the Miranda session command `/miralib' will tell you where it is on your system).

A convention which the Miranda system consistently understands (in Miranda session commands, library directives etc.) is that a pathname enclosed in <angle_brackets>, instead of "string_quotes" is relative to the miralib directory. In particular note that the Miranda session command

```
/cd <ex>
```

will change your current directory to be "....miralib/ex". You can then say, e.g.

```
!ls
```

to see what's in there. In fact there is a README file, so a good thing to say next would be

```
!vi README
```

```
.....
```

15 Definitions

```
.....
```

The purpose of a definition is to give a value to one or more variables. There are two kinds of definition, `scalar' and `conformal'. A scalar definition gives a value to a single variable, and consists of one or more consecutive equations of the form

```
fnform = rhs
```

where a `fnform' consists of the name being defined followed by zero or more formal parameters. Here are three examples of scalar definitions, of `answer', `sqdiff' and `equal' respectively

```
answer = 42
sqdiff a b = a^2 - b^2
equal a a = True
equal a b = False
```

When a scalar definition consists of more than one equation, the order of the equations can be significant, as the last example shows. (Notice that `equals' as defined here is a function of two arguments with the same action as the built in `=' operator of boolean expressions.)

A conformal definition gives values to several variables simultaneously

and is an equation of the form
pattern = rhs

An example of this kind of definition is
(x,y,z) = ploggle

For this to make sense, the value of `ploggle' must of course be a 3-tuple. More information about the pattern matching aspect of definitions is given in the manual section of that name.

Both inform and pattern equations share a common notion of `right hand side'

Right hand sides

The simplest form of rhs is just an expression (as in all the equations above). It is also possible to give several alternative expressions, distinguished by guards. A guard consists of the word `if' followed by a boolean expression. An example of a right hand side with several alternatives is given by the following definition of the greatest common divisor function, using Euclid's algorithm

$$\begin{aligned} \text{gcd } a \text{ b} &= \text{gcd } (a-b) \text{ b}, && \underline{\text{if}} \text{ } a > b \\ &= \text{gcd } a \text{ (b-a)}, && \underline{\text{if}} \text{ } a < b \\ &= a, && \underline{\text{if}} \text{ } a = b \end{aligned}$$

Note that the guards are written on the right, following a comma. The layout is significant (because the offside rule is used to resolve any ambiguities in the parse).

The last guard can be written `otherwise', to indicate that this is the case which applies if all the other guards are false. For example the well known rule for recognising a leap year can be written:

$$\begin{aligned} \text{leap } y &= y \underline{\text{div}} \text{ } 400 = 0, && \underline{\text{if}} \text{ } y \underline{\text{mod}} \text{ } 100 = 0 \\ &= y \underline{\text{div}} \text{ } 4 = 0, && \underline{\text{otherwise}} \end{aligned}$$

The otherwise may here be regarded as standing for if y mod 100 ~= 0. In the general case it abbreviates the conjunction of the negation of all the previous guards, and may be used to avoid writing out a long boolean expression.

Although it is better style to write guards that are mutually exclusive, this is not something the compiler can enforce - in the general case the alternative selected is the first (in the order they are written) whose guard evaluates to True.

Note that for compatability with earlier versions of Miranda the keyword `if' is optional - but its use is strongly encouraged, for readability. There is a Miranda command `/strictif' which causes the compiler to treat missing `if' as a syntax error (say `/nostrictif' to undo).

Block structure

A right hand side can be qualified by a _ w_ h_ e_ r_ e clause. This is written

after the last alternative. The bindings introduced by the where govern the whole rhs, including the guards. Example

$$\text{foo } x = p + q, \underline{\text{if}} \text{ } p < q$$

```

= p - q, if p>=q
      where
      p = x^2 + 1
      q = 3*x^3 - 5

```

Notice that the whole where clause must be indented, to show that it is part of the rhs. Following a where can be any number of definitions, and the syntax of such local definitions is exactly the same as that for top level definitions (including therefore, recursively, the possibility that they may contain nested where 's).

It is not permitted to have locally defined types, however. New typenames can be introduced only at top level.

.....

16 Pattern Matching

.....

The notion of 'pattern' plays an important role in Miranda. There are three contexts in which patterns can be used - in function definitions, in conformal definitions, and on the left of the '<-' in list comprehensions. We first explain the general rules for pattern formation, which are the same in all three contexts.

Patterns are built from variables and constants, using constructors. Here are some simple examples.

```

x
3
(x,y,3)

```

The first pattern is just the variable x, the second is the constant 3, the last example is built from two variables and a constant, using the (,,) constructor for 3-tuples. The components of a structured pattern can themselves be arbitrary patterns, permitting nested structures of any depth.

A pattern can also contain repeated variables, e.g. '(x,1,x)'. A pattern containing repeated variables matches a value only when the parts of the value corresponding to occurrences of the same variable are equal.

The constructors which can be used in a pattern include those of tuple formation '(...)', list formation '[...]', and the constructors of any user defined Algebraic Type (see separate manual section). In addition there are special facilities for matching on lists and natural numbers, as follows.

(Lists) The ':' operator can be used in patterns, so for example the following three patterns are all equivalent (and will match any 2-list).

```

a:b:[]
a:[b]
[a,b]

```

Note that ':' is right associative (see manual section on Operators).

(Natural numbers) It is permitted to write patterns of the form ``p+k'` where `p` is a pattern and `k` is a literal integer constant. This construction will succeed in matching a value `n`, if and only if `n` is an integer $\geq k$, and in this case `p` is bound to `(n-k)`. Example, ``y+1'` matches any positive integer, and ``y'` gets bound to the integer-minus-one.

Note that the automatic coercion from integer to floating point, which takes place in expression evaluation, does not occur in pattern matching. An integer pattern such as ``3'` or ``n+1'` will not match any floating point number. It is not permitted to write patterns containing floating point constants.

Case analysis

The main use of pattern matching in Miranda is in the left hand side of function definitions. In the simplest case a pattern is used simply to provide the right hand side of the function definition with names for subcomponents of a data structure. For example, functions for accessing the elements of a 2-tuple may be defined,

```
fst_of_two (a,b) = a
snd_of_two (a,b) = b
```

More generally a function can be defined by giving a series of equations, in which the use of different patterns on the left expresses case analysis on the argument(s). Some simple examples

```
factorial 0 = 1
factorial(n+1) = (n+1)*factorial n

reverse [] = []
reverse (a:x) = reverse x ++ [a]

last [a] = a
last (a:x) = last x, if x~=[]
last [] = error "last of empty list"
```

Many more examples can be found in the definition of the standard environment (see separate manual section). Note that pattern matching can be combined with the use of guards (see last example above). Patterns in a case analysis do not have to be mutually exclusive (although as a matter of programming style that is good practice) - the rule is that cases are tried in order from top to bottom, and the first equation which ``matches'` is used.

Conformal definitions

Apart from the simple case where the pattern is a single variable, the construction

```
pattern = rhs
```

is called a ``conformal definition'`. If the value of the right hand side matches the structure of the given pattern, the variables on the left are bound to the corresponding components of the value. Example

```
[a,b,3] = [1,2,3]
```

defines `a` and `b` to have the values 1 and 2 respectively. If the match fails anywhere, all the variables on the left are `_ u_ n_ d_ e_ f_`

i_n_e_d. For
example, within the scope of the definition
 (x,x) = (1,2)

the value of x is neither 1 nor 2, but u_n_d_e_f_i_n_e_d (i.e.
an error message
will result if you try to access the value of x in any way).

As a constraint to prevent "nonsense" definitions, it is a rule that the
pattern on the left hand side of a conformal definition must contain at
least one variable. So e.g. `1 = 2' is not a syntactically valid
definition.

Patterns on the left of generators

In a list comprehension (see separate manual entry) the bound entity on
the left hand side of an `<-` symbol can be any pattern. We give two
simple examples - in both examples we assume x is a list of 2-tuples.

To denote a similar list but with the elements of each tuple swapped
over we can write

```
[(b,a)|(a,b)<-x]
```

To extract from x all second elements of a 2-tuple whose first member is
17, we can write

```
[ b |(17,b)<-x]
```

Irrefutable patterns (*)

(Technical note, for people interested in denotational semantics)

DEFINITION:- an algebraic type having only one constructor and for which
that constructor is non-nullary (ie has at least one field) is called a
product type. The constructor of a product type is called a 'product
constructor'.

Each type of n-tuple (n~>=0) is also defined to be a product type. In
fact it should be clear that any user defined product type is isomorphic
to a tuple type. Example, if we define

```
wibney ::= WIB num bool
```

then the type wibney is isomorphic to the tuple type (num,bool).

A pattern composed only of product-constructors and identifiers, and
containing no repeated identifiers, is said to be "irrefutable". For
example `WIB p q', `(x,y,z)' and `(a,(b,c))' are irrefutable patterns.
We show what this means by an example. Suppose we define f, by

```
f :: (num,num,bool) -> [char]  
f (x,y,z) = "bingo"
```

As a result of the constraints of strong typing, f can only be applied
to objects of type (num,num,bool) and given any actual parameter of that
type, the above equation for f MUST match.

Interestingly, this works even if the actual parameter is an expression
which does not terminate, or contains an error. (For example try typing

```
f undef
```

and you will get "bingo", not an error message.)

This is because of a decision about the denotational semantics of algebraic types in Miranda - namely that product types (as defined above) correspond to the domain construction DIRECT PRODUCT (as opposed to lifted product). This means that the bottom element of a type such as (num,num,bool) behaves indistinguishably from (bottom,bottom,bottom).

Note that singleton types such as the empty tuple type `()`, or say,
it ::= IT

are not product types under the above definition, and therefore patterns containing sui-generis constants such as () or IT are not irrefutable. This corresponds to a semantic decision that we do NOT wish to identify objects such as () or IT with the bottom element of their type.

For a more detailed discussion of the semantics of Miranda see the formal language definition (in preparation).

(* A useful discussion of the semantics of pattern-matching, including the issue of irrefutable patterns, can be found in (chapter 4 of) the following

S. L. Peyton-Jones ``The Implementation of Functional Programming Languages'', Prentice Hall International, March 1987.
ISBN 0-13-453333-X

.....

17 Compiler directives

.....

Certain keywords, beginning with `%`, modify the action of the compiler when present in a script. These are called `compiler directives'. The directives currently available are as follows.

`%list %nolist`

If the `/list` feature is enabled (switched on and off by /list, /nolist at command level) the compiler echos the source to the user's terminal during compilation of a Miranda script. The directives %list, %nolist may be placed in a file to give more detailed control over this behaviour. If the compiler is in an echoing state then encountering `%nolist` causes it to cease echoing from that point onwards in the source, until the next occurrence of '%list' or the end of the source file in which the directive occurs, whichever is the sooner. These directives may occur anywhere in a script and have no effect on the semantics (i.e. they are just like comments, apart from having a side-effect on the lex analyser).

If the `/list` feature of the compiler is not enabled these directives are ignored. Since the default state of the compiler is now `/nolist` these directives are of marginal value and retained only for historical reasons.

`%insert`

A directive of the form
`%insert "pathname"`

may occur anywhere in a Miranda script, and is textually replaced by the

contents of the file "pathname" during lexical analysis. The pathname must be given as a literal string, enclosed in double quotes. (Most uses of this directive are now better handled by %include, see below).

If the %insert directive is textually indented in the file in which it occurs, the whole of the inserted text will be treated as being indented by the same amount as the initial `%` of the directive.

Insert directives may be invoked recursively, to a depth limit imposed by the operating system, typically about 16, which should be enough for any reasonable purpose. Note that the pathnames are resolved statically, not dynamically, so that the meaning of an %insert directive is computed relative to the file in which it occurs, NOT relative to the directory from which the compiler was invoked.

The use of static rather than dynamic pathname resolution is a departure from normal UNIX conventions (both the `C` compiler and the UNIX shell resolve pathnames dynamically) but is much more convenient in practice.

Note that if the subject of an %insert directive is a complete Miranda script it is always better to use %include (see manual section on the library mechanism), since this avoids needless recompilation of the definitions of the subsidiary script. The use of %include also imposes a hierarchical scope discipline, and is more likely to lead to well structured code.

A point to beware of when using %insert is that unlike %include, it does NOT add a missing `.m` extension to its pathname argument automatically. This is because the argument file may contain an arbitrary piece of text (e.g. an expression or a signature) and need not be a complete Miranda script, so it would be inappropriate to insist that it is pathname end in `.m` in all cases.

```
%include %export %free
```

These directives control the identifier bindings between separately compiled scripts. See separate manual entry on `the library mechanism` for details.

```
:::::::::::::::::::::::
```

18 Basic type structure and notation for types

```
:::::::::::::::::::::::
```

The Miranda programming language is strongly typed - that is each expression and each variable has a type that can be deduced by a static analysis of the program text.

Primitive types

```
num bool char
```

Values of type `num` include both integer and floating point numbers, e.g.

```
23      0      -17     1.26e11
```

They are stored using different internal representations, but can be freely mixed in calculations, and are both of type `num` for type checking purposes. There is automatic conversion from integer to

floating point when required (but not in the opposite direction - use `entier', see standard environment). Floating point numbers are held to double precision, integers to unbounded precision.

The values of type `bool' are the two truth values:
True False

The values of type `char' are ascii characters, e.g.
'a' '%' '\t'

List types

[t] is the type of lists whose elements are of type `t'

Thus [num] is the type of lists of numbers such as [1,2,3,4,5]

[[num]] is the type of lists of lists of numbers, such as [[1,2],[3,4]]

[char] are lists of characters - this is also the type of string constants, so e.g. ['h','e','l','l','o'] and "hello" are interchangeable objects of this type.

Tuple types

(t1,...,tn) is the type of a tuple with elements of type `t1' to `tn'

Example - the value (1,True,"red") is of type (num,bool,[char])

The type of the empty tuple, `()', is also written `()'.

Notice that tuples are distinguished from lists by being enclosed in parentheses, instead of square brackets.

There is no concept of a 1-tuple, in Miranda, so the use of parentheses to enclose subexpressions, as in say a*(b+c), does not conflict with their use for tuple formation.

Function types

t1->t2 is the type of a function with argument type `t1' and result type `t2'

The '->' is right associative, so e.g. `num->num->num' is the type of a curried function of two numeric arguments.

In addition to the built-in types described above, user defined types may be introduced - these are of three kinds, synonym types, algebraic types and abstract types - see separate manual entry for each.

Implicit typing

In Miranda the types of identifiers do NOT normally need to be declared explicitly - the compiler is able to infer the type of identifiers from their defining equations. For example if you write

```
plural x = x ++ "s"
```

the compiler will DEDUCE that `plural' is of type [char]->[char]. It is however permitted to include explicit type declarations in the script if desired, e.g. you could write (anywhere in the same script)

```
plural :: [char]->[char]
```

and the compiler will check this for consistency with the defining equation (the symbol `::' means `is of type'). More generally the type declared may be an instance (see below) of the type implied by the definition - in this case the effect of the declaration is to restrict the type of the identifier to be less general than it would otherwise have been.

Note that only top-level identifiers may be the subject of type declarations, and that the type of an identifier may be declared at most once in a given script.

Polymorphism

The final feature of the type system is that it permits polymorphic types. There is an alphabet of generic type variables, written

```
*      **      ***      etc.
```

each of which stands for an arbitrary type. We give a simple example - the identity function, which may be defined

```
id x = x
```

is attributed the type `*->*'. This means that `id' has many types - `num->num', `char->char', `[[bool]]->[[bool]]' and so on - each of these is an instance of its most general type, `*->*'.

Another simple example of polymorphism is the function `map' (see standard environment) which applies a function to every element of a list. For example `map integer [1,1.5,2]' is [True,False,True]. The type of map is

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

The most polymorphic possible object is `undef', the identifier which stands for the undefined, or error value (undef is defined in the standard environment). Since every type has an undefined value, the correct type specification for undef is

```
undef :: *
```

Many of the functions in the standard environment have polymorphic types - the text of the standard environment (see separate manual entry) is therefore a useful source of examples.

.....

19 Type synonym declarations

.....

These permit the user to introduce a new name for an already existing type, e.g.

```
string == [char]
```

type synonyms are entirely transparent to the typechecker (it best to think of them as being just macros). For obvious reasons, recursive type synonyms are not permitted.

It is also possible to introduce a synonym for a type forming operator, by introducing generic type variable as parameters of the definition, e.g.

```
invt * ** == (*->**)->(**->*)
```

So within a script containing the above two '=' definitions, the type 'invt num string' will be shorthand for
(num->[char])->([char]->num)

.....

20 Algebraic type definitions

.....

The simplest method of introducing a new data type into a Miranda script is by means of an algebraic type definition. This enables the user to introduce a new concrete data type with specified constructors. A simple example would be

```
tree ::= Nilt | Node num tree tree
```

The '=' sign (pronounced 'comprises') is always used to introduce an algebraic data type. This definition introduces three new identifiers - 'tree', a typename - 'Nilt' a nullary constructor of type tree (i.e. an atomic tree) - and 'Node', a constructor of type num->tree->tree->tree. Now we can define a particular tree, using the constructors Nilt and Node, for example

```
t = Node 3 Nilt Nilt
```

It is not necessary to have names for the selector functions because the constructors can be used in pattern matching. For example a function for counting the number of nodes in a tree could be written

```
size Nilt = 0  
size (Node a x y) = 1 + size x + size y
```

Note that the names of constructors must begin with an upper case letter (and conversely, any identifier beginning with an upper case letter is assumed to be a constructor).

An algebraic type can have any number (≥ 1) of constructors and each constructor can have any number (≥ 0) fields, of specified types. The number of fields taken by a constructor is called its 'arity'. A constructor of arity zero is said to be atomic. Algebraic types are a very general idea and include a number of special cases that in other languages require separate constructions.

One interesting case that all of the constructors can be atomic, giving us what is called in PASCAL a 'scalar enumeration type'. Example

```
day ::= Mon|Tue|Wed|Thu|Fri|Sat|Sun
```

The union of two types can also be represented as an algebraic data type - for example here is a union of num and bool.

```
boolnum ::= Left bool | Right num
```

Notice that this is a 'labelled union type' (the other kind of union type, in which the parts of the union are not distinguished by tagging information, is not permitted in Miranda).

An algebraic typename can take parameters, thus introducing a family of types. This is done by using generic type variables as formal

parameters of the `::=` definition. To modify our definition of `tree` to allow trees with different types of labels at the nodes (instead of all `num` as above) we would write

```
tree * ::= Nilt | Node * (tree *) (tree *)
```

Now we have many different tree types - `tree num`, `tree bool`, `tree([char]->[char])`, and so on. The constructors `Node` and `Nilt` are both polymorphic, with types `tree *` and `*->tree *->tree *->tree *` respectively.

Notice that in Miranda objects of a recursive user defined type are not restricted to being finite. For example we can define the following infinite tree of type `tree num`

```
bigtree = Node 1 bigtree bigtree
```

Obsolete Language Features

Algebraic data types in Miranda originally ((*see Turner 85) supported two additional features, which have now been dropped from the definition of the language. These are laws (equations on constructors written using `=>`) and strictness (written using `!`) on fields in `::=` definitions.

The release-two compiler continues to support these features, so former users of Miranda release one do not lose working programs at the change to release two. However, they will eventually cease to be supported by the Miranda compiler (sorry!). An `obsolete feature` warning given at each compilation of a script using them. A methodology for translating out these features is given in the CHANGES section of this manual.

Laws and strictness annotations are no longer part of the Miranda language, and should not be taught to new users.

(* D. A. Turner ``Miranda: A Non-Strict Functional Language with Polymorphic Types'', Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Springer Lecture Notes in Computer Science, vol. 201, pp 1-16).

.....

21 Abstract type definitions

.....

These enable a new data type to be defined by data type abstraction from an existing type. We give the classic example, that of defining `stack` as an abstract data type (here based on lists)

```
abstype stack *  
with empty::stack *  
  push::*->stack *->stack *  
  isempty::stack *->bool  
  top::stack *->*  
  pop::stack *->stack *
```

```
stack * == [*]  
empty = []
```

```
push a x = a:x
isempty x = x=[]
top (a:x) = a
pop (a:x) = x
```

The information given after 'with' is called the signature of the abstract type - the definitions of the identifiers in the signature are called the 'implementation equations' (these are the six equations given above). Outside of the implementation equations the information that stacks are implemented as lists is not available - [] and empty for example are incomparable objects of two different and unrelated types ([*] and stack * respectively). Only inside the implementation equations are the abstract objects treated as being equivalent to their representations.

The implementation equations do not have to appear immediately after the corresponding abstype declaration - they can occur anywhere in the script. For readability, however, it is strongly recommended that the implementation equations appear immediately after the abstype declaration.

Note that in Miranda there is no runtime cost associated with administering an abstract data type. The responsibility for enforcing the distinction between stacks and lists, for example, is discharged entirely at compile time (by the type checker). The runtime representation of a stack does not require any extra bits to distinguish it from a list. As a result the Miranda programmer can freely use abstract data types to structure his programs without incurring any loss of efficiency by doing so.

Notice that the mechanism used to introduce abstract data types in Miranda does not depend on the hiding of identifiers, and in this respect differs from the traditional approach. A fuller discussion of the Miranda abstype mechanism can be found in [*Turner 85].

(* D. A. Turner ``Miranda: A Non-Strict Functional Language with Polymorphic Types'', Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Springer Lecture Notes in Computer Science, vol. 201, pp 1-16).

The print representation of abstract objects
("advanced feature" - omit on first reading)

Values belonging to an abstract type are not in general printable. If the value of a command-level expression is of such a type it will normally print simply as

```
<abstract ob>
```

This is because the special function _ s_ h_ o_ w (which is actually a family of functions, see elsewhere) has no general method for converting such objects to a printable form. It is possible to extend the definition of show to include the ability to print members of an abstract type, using

some appropriate format. The convention for doing this is to include in the definition of the abstract type a function with the name `showfoo' (where `foo' is the name of the abstract type involved).

We illustrate how this is done taking `stack' as the example. Suppose we decide we wish stacks to print - using a syntax such that the output could be read back in (e.g. by readvals - see elsewhere) to generate the same stack.

```
empty is to print as "empty"
push 1 empty is to print as "(push 1 empty)"
and so on.
```

Note that we decide to fully parenthesise the output for safety - since we do not know the larger context in which our stack output may be embedded.

Because `stack' is a polymorphic abstraction, showstack will need to take as a parameter the appropriate show function for the element type (which is num in the above examples, but could have been any type). We add to the signature of stack the following function.

```
showstack :: (*->[char])->stack *->[char]
```

To obtain the output format illustrated above, an appropriate definition of showstack would be,

```
showstack f [] = "empty"
showstack f (a:x) = "(push " ++ f a ++ " " ++ showstack f x ++ ")"
```

If this definition is included in the script, stacks become printable, using the specified format. The effect is to extend the behaviour of the special built-in function show to handle stacks, and all data structures built using stacks (such as list of tree of stacks, stack of stacks and so on).

The general rule is as follows. Let `foo' be an abstract type name. To make foo's printable, we need to define a `showfoo' thus:

```
if foo is a simple type (not polymorphic)
showfoo :: foo ->[char]

if foo is polymorphic in one type variable (foo *)
showfoo :: (*->[char]) -> foo * -> [char]

if foo is polymorphic in two type variables (foo * **)
showfoo :: (*->[char]) -> (**->[char]) -> foo * ** ->
```

```
[char]
```

and so on. Note that the show function must be declared in the signature of the abstract type, and that the name of the function is significant - if we change its name from `showfoo' to `gobbledegook', its definition will cease to have any effect on the behaviour of show. It also needs to have the right type, and if it does not, again its presence will have no effect on the behaviour of show (in this case the compiler prints a warning message).

[Note on library directives: If you %export an abstract type, foo say, to another file, it is not necessary to %export the showfoo function explicitly to preserve the correct printing behaviour - if an abstract type comes into existence with a show function in its signature the compiler will `remember' how to print objects of the type even in scopes where the show function has no name.]

.....

22 Place holder types

.....

A placeholder type has no values belonging to it (apart from the undefined value, undef, which is a member of every type). Placeholder types are useful in program development. Placeholder types are declared as follows:

```
widget :: type
```

declares `widget' to be a `placeholder type'. This enables us to give type specifications involving widget, for example

```
f :: num->widget->widget  
g :: widget->widget
```

The sole purpose of this setup is (as the name suggests) to `hold a place' during program development. At a later stage the above declaration of widget as a placeholder type will be removed in favour of some specific type definition for widget (using :=, abstype or ==).

Placeholder typenames can have any arity, eg we could have said:

```
widget * ** :: type
```

This creates a family of placeholder types, such as `widget num bool' and so on. They are all `empty' (meaning they have no values except undef). The general form of this kind of specification is

```
tform-list :: type
```

where `tform' consists of a typename followed by zero or more generic type variables (and it is permitted to declare several placeholder types simultaneously, separated by commas, whence `tform-list').

The philosophy behind allowing placeholder types is that when developing a complex piece of software it is often helpful to declare the types of a group of functions and data structures, and have this type information checked for consistency, before taking representation decisions, such as whether a given type is going to be abstract or algebraic etc.

A placeholder type may be considered as equivalent to an algebraic type with no constructors.

.....

23 The use of `show for converting objects to their print representations

.....

The need often arises to convert an arbitrary Miranda value to its

printable representation as a string. For numbers the function ``shownum'` (of type `num->[char]`) can be used for this purpose. To be able to do this conversion for any type of object would seemingly require an infinite number of functions, one for each type. As a solution to this problem Miranda provides a special keyword, ``show'`.

For any object `x`

`show x`

is a string containing the printable representation of `x`. For example, if `x` is a number the above expression is equivalent to ``shownum x'`. In the general case, however, `x` could be a structured object of arbitrary complexity. Note that `show` is a reserved word, not an identifier.

In fact ``show'` behaves under most circumstances as though it was the name of a function, of type `*->[char]`. For example it can be passed as a parameter, so that say,

`map _s_h_o_w [a,b,c,d,e]`

is a legal expression of type `[[char]]`.

There are three important restrictions on the universality of `show`.

(i) You cannot ``show'` functions in any useful sense. (That would be a violation of referential transparency.) The result of applying `show` to any kind of function is just the string `"<function>"`.

(ii) You cannot ``show'` an abstract object unless an appropriate show-function was included when the type was defined (see manual section on Abstract types). The result of applying `show` to such an object is by default just the string `"<abstract ob>"`.

(iii) When it occurs in a script the context in which `show` is used must be such as to determine its type monomorphically. An example:

`my_show x = "hello\n"++ show x++"goodbye\n"`

In the absence of any other type information, the compiler will infer that `my_show` has type `*->[char]`, and that `x` is of type ``*'`. The use of `show` is therefore polymorphic, and will be rejected as illegal.

If however we intend that `my_show` will be used only on objects of type ``tree'`, say, and we add to the script the declaration ``my_show::tree->[char]'`, then the above use of `show` becomes monomorphic, and will be accepted.

The essence of restriction (iii) is that `show` is not a true polymorphic function of type `*->[char]`, but rather a family of monomorphic functions with the types `T->[char]` for each possible monotype `T`. The context must be sufficient for the compiler to determine which member of the family is required.

(For technical reasons this restriction applies only in scripts. In command-level expressions `show` behaves as if it were a genuine polymorphic function.)

::::::::::::::::::::::::::

24 Syntax of Miranda scripts and expressions

.....

```
script:= decl*                                rhs:= simple_rhs(;)
                                             cases

decl:= def                                     simple_rhs:= exp whdefs?
      tdef                                     cases:= alt(;) = cases
      spec                                     lastcase(;)
      libdir

def:= fnform = rhs                            alt:= exp , _ i_ f? exp
      pat = rhs

tdef:= tform == type(;)                       lastcase:= lastalt whdefs?
      tform ::= constructs(;)                lastalt:= exp , if? exp
      abstype tform-list with sig(;)        exp , otherwise

spec:= var-list :: type(;)                    whdefs:= where
      tform-list :: type(;)

def def*

sig:= spec spec*

constructs:= construct | constructs
            construct

construct:= constructor argtype*
            type $constructor type
            ( construct ) argtype*

type:= argtype
      typename argtype*
      type -> type
      type $typename type

argtype:= typename
          typevar
          ( type-list? )
          [ type-list ]

tform:= typename typevar*
        typevar $typename typevar

fnform:= var formal*
         pat $var pat
         ( fnform ) formal*

pat:= formal
     -numeral
     constructor formal*
     pat : pat
     pat + nat
     pat $constructor pat
     ( pat ) formal*

formal:= var

simple_rhs:= simple_rhs(;)
           cases

simple_rhs:= exp whdefs?

cases:= alt(;) = cases
       lastcase(;)

alt:= exp , _ i_ f? exp

lastcase:= lastalt whdefs?

lastalt:= exp , if? exp
          exp , otherwise

whdefs:= where

exp:= e1
     prefix1
     infix

e1:= simple simple*
    prefix e1
    e1 infix e1

simple:= var
       constructor
       literal
       readvals
       show
       ( infix1 e1 )
       ( e1 infix )
       ( exp-list? )
       [ exp-list? ]
       [ exp .. exp? ]
       [ exp , exp .. exp? ]
       [ exp | qualifs ]
       [ exp // qualifs ]

qualifs:= qualifier ; qualifs
          qualifier

qualifier:= exp
            generator

generator:= pat-list <- exp
           pat <- exp , exp ..

var:= identifier

constructor:= IDENTIFIER
```

```

        constructor                typename:= identifier
        literall
        ( pat-list? )
        [ pat-list? ]
        ::::::::::::::::::::::::::::

```

25 Comments on the syntax for Miranda scripts

```

        ::::::::::::::::::::::::::::

```

The syntax equations given express the syntax of Miranda scripts insofar as this can be done by a context free grammar. It therefore does not attempt to express the scope rules, nor the requirement that a script be well-typed, both of which are context sensitive restrictions on the syntax given here. The formal definition of Miranda [in preparation] will deal with these matters and also give a denotational semantics for the language.

Nevertheless, if the syntax is read in conjunction with the informal description of the language (see other manual sections and referenced papers) it should be found fairly informative, especially if the reader has some previous exposure to this style of language.

Key to abbreviations in syntax:-

```

> alt=alternative
decl=declaration    def=definition
> el=operator_expression    exp=expression
fnform=function_form
> libdir=library_directive    pat=pattern
qualifs=qualifiers
> rhs=right_hand_side
sig=signature        spec=specification
> tdef=type_definition
tform=typeform        var=variable
> whdefs=where_defs

```

Conventions

We use a variant of BNF, in which non-terminals are represented by lower case words, `:=` is used as the production symbol, and alternative productions are written on successive lines. (These departures from convention are adopted because `::=` and `|` are concrete symbols of the language.)

For any non-terminal x,

```

x*      means zero or more occurrences of x
x?      means the presence of x is optional
x-list  means one or more x's (separated by commas if >1)
x(;)    means that x is followed by an optional semicolon and
is subject to the off side rule (see section on Layout), so that every
token of x must lie below or to the right of the first. Provided the
layout makes it clear where x terminates, the trailing semicolon may be
omitted.

```

Notes

The syntax of the library directives (denoted by the non-terminal

``libdir'`) is given in a separate manual entry.`

Ambiguities in the syntax for ``type'` and ``construct'` are resolved by noting that ``->'` is less binding than ``$typename'` or ``$constructor'` and that all three are right associative.

In connection with the productions for `argtype`, note that type such as ``[num,bool]'` is an abbreviation for ``[(num,bool)]'` and represents the type of a list of tuples - the Miranda system itself never uses this abbreviation when printing a type, but accepts it in user scripts. (Use of this abbreviation is not recommended - it will probably be removed from the syntax at the next release.)

Ambiguities in the syntax for ``fnform'` and ``pat'` are resolved by taking into account the relative binding powers of the infix operators involved. Specifically that ``:'` is right associative and less binding than ``+'`, which is left associative, and that `$constructor`, `$var` have a higher binding power than either of these, and are right associative.

The productions given for ``cases'` correctly describe the concrete syntax of these entities, including the way the offside rule is applied to them. This concrete syntax is in one sense misleading, however. Namely in that if a ``lastcase'` with a trailing ``wheredefs'` is preceded by a series of alternatives, the scope of the names introduced by the where IS THE WHOLE ``cases'` IN WHICH IT OCCURS, AND NOT JUST THE ``lastcase'`.

Note that for compatibility with earlier versions of Miranda the use of the keyword ``if'` is optional.

The ambiguities in the syntax given for ``el'` are resolved by taking into account the relative binding powers of the operators (see manual section on Operators).

The syntax of `identifier`, `IDENTIFIER`, `literal`, `literal1`, `numeral`, `nat`, `infix`, `infix1`, `prefix`, `prefix1`, and `typevar` are given under Lexical Syntax (see next manual section).

::::::::::::::::::::::::::

26 Miranda lexical syntax

::::::::::::::::::::::::::

In this section square brackets are used to enclose a set of literal characters, using lex-style conventions, so eg `[a-z]` means a lower case letter. As usual `*` and `?` are used to mean zero-or-more, and zero-or-one, occurrences of the preceding entity. Parentheses are used for grouping, and subtraction of one syntactic entity from another means set difference. We also revert to using ``|'` for alternatives, as in standard BNF.

```
script:= (token | layout)*
```

```
layout:= nl | tab | formfeed | space | comment
```

```
comment:= vertical_bar vertical_bar (any - nl)* nl
```

```

token:= identifier | IDENTIFIER | literal | typevar | delimiter
identifier:= ([a-z] [a-zA-Z0-9_']* ) - delimiter
IDENTIFIER:= [A-Z] [a-zA-Z0-9_']*
literal:= numeral | charconst | stringconst
literal1:= literal - float
numeral:= nat | float
nat:= [0-9] [0-9]*
float:= [0-9]* [.] nat epart? | nat epart
epart:= [e] [+|-]? nat
charconst:= ['] (visible-[\]|escape) [']
stringconst:= ["] (visible-[\"]|escape)* ["]
escape:= [\] ([ntfrb\']|nl|ascii_code)
typevar:= [*][*]*
delimiter:= - | prefix1 | infix1 | other
infix1:= ++ | -- | : | \ / | & | ~ | > | >= | = | ~= | <= | < | + |
        * | / | div | mod | ^ | . | ! | $identifier | $IDENTIFIER
infix:= infix1 | -
prefix1:= ~ | #
prefix:= prefix1 | -
other:= abstype | if | otherwise | readvals | show | type | where |
        with | %export | %free | %include | %insert | %list |
%nolist |
        = | == | ::= | :: | => | vertical_bar | // | -> | ; | , |
( |
        ) | [ | ] | { | } | <- | .. | $$ | $- | $+ | $*
vertical_bar:= |

```

Notes

visible means any non-control character, including space (but not including eg newline), nl means literal newline, and ascii_code is a nat in the range 0..255 (maximum length 3 digits).

Notice that the syntax of `numeral' does not include negative numbers. Negative constants, such as -3 or -5.05e-17 are parsed by Miranda as applications of the prefix operator '-' to a positive numeral. This has no semantic significance.

Omission - the definition of `layout' does not include the additional

comment rules for LITERATE SCRIPTS (see separate manual section).

.....

27 The Miranda Library Mechanism

.....

1. Syntax of library directives
2. The `%include` directive (basic information)
3. About library directives (more detailed information)
4. The `%free` directive and parametrised scripts
5. Separate compilation and `.x` files

.....

27/1 Syntax of library directives

.....

```
libdir:= %include env(;)           parts:= part part*
         %export parts(;)
         %free{ sig }
env:= fileid binder? aliases?
binder:= { binding binding* }
binding:= var = exp(;)
         tform == type(;)
aliases:= alias alias*
alias:= identifier/identifier
        IDENTIFIER/IDENTIFIER
        -identifier
```

```
fileid:= "pathname"
        <pathname>
```

Note For the definition of ``sig'` (=signature), ``var'`, ``exp'`, ``tform'` and ``type'` see the main manual page on formal syntax of Miranda. For the definition of ``identifier'` and ``IDENTIFIER'` see lexical syntax.

.....

27/2 The `%include` directive basic information

.....

Suppose the file "mylib.m" contains some Miranda declarations (which could be any kind of legal Miranda declaration, eg function definitions, type definitions etc). To make these in scope in another script, the latter will contain the directive

```
    %include "mylib"
```

This can come anywhere in the script (there is for example no requirement that `%include` directives come at the front) but must be at top level - you may not place a `%include` directive inside a `where` clause. The subject of a `%include` directive may itself contain `%include` directives, and so on (to any reasonable depth).

By default, the names ``exported'` from an included script are all those defined in it at top level, but not those of subsidiary `%include`'s.

This can be modified by placing a ``%export'` directive in the included script. For a discussion of this and other ways of modifying the effect of `%include` see the manual section giving a detailed account of the library directives.

If the filename in a `%include` directive is enclosed in angle brackets instead of string quotes, this is understood to be a pathname relative to the `miralib` directory. So for example putting in your script

```
%include <ex/matrix>
```

brings into scope the definitions exported from the `matrix` package in the `Miranda examples` directory ``ex'`. (*See note below.)

Finally, note that that the relationship between includor and includee is unsymmetrical. If file A `%include's` file B, then the declarations of B are visible to A, but not vice versa.

There is a simpler (purely textual) directive

```
%insert "file"
```

which causes the contents of "file" to be substituted for the `%insert` directive during lexical analysis. This can occur anywhere in a `Miranda` script. See manual section on compiler directives.

(* Note to system administrators: an empty directory ``local'` is provided under the `miralib` directory, in which you can place additional libraries which you wish to make available to all `Miranda` users at your site.

::::::::::::::::::::::::::::

27/3 Explanation of library derictives

::::::::::::::::::::::::::::

The three directives `%include` `%export` `%free` constitute the `Miranda` library mechanism, which controls the sharing of identifiers between separately compiled scripts. The `%free` directive is covered in a separate manual entry and will not be discussed further here.

`%include "file"`

The presence of this directive, anywhere in a `Miranda` script, causes all the identifiers exported from the `Miranda` script "file" to be in scope in the containing script. Note that "file" should be the name of a `Miranda` source file (by convention these all have names ending in ``.m'`).

The following conventions apply to all filenames in library directives:

- 1) A fileid can be an arbitrary UNIX pathname
- 2) If the fileid given does not end in ``.m'` this is added.
- 3) If the fileid is surrounded by angle brackets instead of string quotes it is assumed to be a pathname relative to the ``miralib'` directory, otherwise it is taken to be relative to the directory of the script in which the directive occurs. (Examples, "pig" means "pig.m", `<stdenv>` means `"/usr/lib/miralib/stdenv.m"`)

In addition (if you are using Berkeley UNIX or a derivative) the ```~'`

convention of the cshell may be used to abbreviate home directories. That is '~' abbreviates your own home directory, and ~jack abbreviates jack's home directory, at the front of any pathname.

The file mentioned in a %include directive must contain a CORRECT, CLOSED Miranda script. (That is it must have no syntax or type errors, and no undefined names.) An attempt to %include a file which violates these conditions will be rejected by the compiler as a syntax error in the script containing the %include statement.

It is also illegal to %include a script which causes nameclashes, either against top-level identifiers of the including script or those of other %include directives in the script.

The effect of an %include directive can be modified by following it with one or more aliases (which are used to remove nameclashes between identifiers exported from the included script and those of the current script or of other %include files). There are two forms of alias, 'new/old' which renames and '-old' which suppresses an identifier altogether.

For example suppose we wish to include the file "mike" but it contains two identifiers, f and g say, which clash with top-level bindings of these identifiers in the current script. We wish to use the "mike" definition of 'f', but his 'g' is of no interest. The following directive could be used.

```
%include "mike" -g mike_f/f
```

Any other identifiers exported from "mike", not specifically mentioned in the aliases, will be accepted unchanged.

It is permitted to alias typenames, and constructors (say 'NEW/OLD') but typenames and constructors cannot be suppressed by a '-name' alias. Note that if you alias one or more of the constructors of an algebraic data type the behaviour of 'show' on objects of that type will be modified in the appropriate way.

A file which has been %included may itself contain %include directives, and so on, to any reasonable depth. A (directly or indirectly) cyclic %include is not permitted, however.

The '?identifier' command can be used to find the ultimate place of definition of an imported identifier. When aliasing has taken place the '?identifier' command will give both the current and the original name of an aliased identifier. If your installed editor is 'vi' the '??identifier' command will open the appropriate source file at the definition of the identifier. (There is also a command '/find identifier' which is like '?identifier' but will recognise an alias under its original name.)

Every script behaves as though it contained the directive
%include <stdenv>

It is therefore illegal to %include the standard environment explicitly, as this will lead to huge number of name clashes (because it is now present twice). As a consequence there is currently no way of aliasing

or suppressing the names in the standard environment. (This will be fixed in the future by providing a directive for suppressing the implicit inclusion of <stdenv>.)

%export parts

Any (correct, closed) Miranda script can be %included in another script (there is no notion of a "module" as something with a different syntax from an ordinary script). By default the names exported from a script are all those defined in it, at top level, but none of those acquired by a %include of another file. This behaviour can be modified (either to export more or to export less than the default) by placing a %export directive in the script, specifying a list of `parts' to be exported to an including file.

The presence of a %export directive in a script has no effect on its behaviour when it is the current script of a Miranda session - it is effective only when the script is %included in another. A script may contain at most one %export directive. This can be anywhere in the script, but to avoid nasty surprises it is advisable to place it near the top.

Each `part' listed in the export directive must be one of the following:

- identifier (variable or typename)
- fileid (in quotes, conventions as described above)
- the symbol `+'
- identifier

Notice that constructors need not (and cannot) be listed explicitly in an %export directive - if you export an algebraic typename, its constructors are AUTOMATICALLY exported along with it. The consequence of this is that you cannot use %export to create an abstract data type, by "hiding information" about how an algebraic type was formed. If you want to create an abstract data type you must use the abstype mechanism - see separate manual entry.

If a fileid is present in the export list, this must be the name of a file which is %included in the exporting script, and the effect is that all the bindings acquired by that %include statement (as modified by aliases if present) are re-exported. Allowing fileid's in the export list is merely a piece of shorthand, which can be used to avoid writing out long lists of names.

The symbol `+' is allowed in an export list as an abbreviation for all the top-level identifiers defined in the current script.

The default %export directive (i.e. that which is assumed if no %export statement is present) is therefore

%export +

This will export all the top-level identifiers of the current script, but not those acquired by %include directives.

Finally, the notation `-identifier' is allowed in an export list to indicate that this identifier NOT to be exported. This is useful if you have used a fileid or the symbol `+' to abbreviate a list of names, and wish to subtract some of these names from the final export list.

An example - the following export statement exports all the top-level identifiers of the current script, except 'flooby'.

```
%export + -flooby
```

The order of appearance of the items in an export list is of no significance, and repetitions are ignored. A negative occurrence of an identifier overrides any number of positive occurrences.

It is possible to find out what names are exported from a given Miranda script (or scripts) by calling, from UNIX, the command 'mira -exports files' (the extension '.m' will be added to each file name if missing). This will list (to stdout) for each file the exported names together with their types.

Some examples

(i) There are two scripts, called "liba.m" and "libb.m" say, containing useful definitions. For convenience we wish to combine them into a single larger library called say, "libc.m". The following text inside the file "libc.m" will accomplish this.

```
%export "liba" "libb"  
%include "liba"  
%include "libb"
```

You will notice that when "libc.m" is compiled, this does NOT cause recompilation of "liba.m" and "libb.m" (see section on separate compilation - the compiler is able to create an object code file for "libc.m", called "libc.x", by combining "liba.x" and "libb.x" in an appropriate way). This economy in recompilation effort is one reason why %include is a better mechanism than the simpler textual directive %insert (see section on compiler directives).

We could if we wished add some definitions to "libc.m" - if the corresponding names are added to the %export statement these bindings will then be exported along with those of the two sublibraries. Of course if we don't add the names of the locally defined objects to the %export directive they will be 'private definitions' of "libc.m", not visible to includors.

Recall that if no %export is directive is present, then ONLY the immediate definitions (if any) of "libc.m" will be exported. So a script which contains only %include directives and no %export cannot be usefully %included in another script (it is however perfectly acceptable as a current script).

(ii) [More technical - omit on first reading]

Our second group of examples is chosen to bring out some issues which arise when exporting types between scripts. Suppose we have the following script, called "trees.m"

```
tree * ::= NILT | NODE * (tree *) (tree *)  
reflect :: tree *->tree *  
reflect NILT = NILT  
reflect (NODE a x y) = NODE a(reflect y)(reflect x)
```

(a) If in another script we write `'%include "trees"'`, the following bindings will be imported - `tree NILT NODE reflect`. Now suppose we modify the "trees.m" by placing in it the following directive - `'%export reflect'`. When the modified "trees.m" script is included in another, we will get the following error message from the compilation of the including script:

```
MISSING TYPENAME
the following type is needed but has no name in this
scope:
'tree' of file "trees.m", needed by: reflect;
typecheck cannot proceed - compilation abandoned
```

Explanation - it is illegal to export an identifier to a place where its type, or any part of its type, is unknown. In this situation we call reflect a 'type-orphan' - it has lost one of its parents!

(b) Readoption of a type-orphan (a more subtle example). Assuming the "trees.m" script in its original form as above, we construct the following file "treelib.m"

```
%export size
%include "trees"
size :: tree *->num
size NILT = 0
size (NODE a x y) = 1+size x+size y
```

If we `%include` the above script in another as it stands, we will of course get a missing typename diagnostic for 'size' - consider however the following script

```
%include "treelib"
%include "trees"
... (etc)
```

Everything is now ok, because a name for size's missing parent is imported through another route (the second `%include` statement). The Miranda compiler recognises the 'tree' type imported by the second `%include` as being the same one as that referred to inside "treelib.m", because it originates (albeit via different routes) from the SAME SOURCEFILE. A 'tree' type imported from a different original sourcefile, even if it had the same constructor names with the same field types, would be recognised as a different type.

[Note: the Miranda compiler is always able to recognise when the same source file is inherited via different routes, including in cases involving files with multiple pathnames due to the presence of (hard or sybolic) links.]

[Further note: the lexical directive `%insert` (see compiler directives) should be regarded as making a textual copy of the material from the inserted file into the file containing the `%insert` directive - if the text of a type definition (in `::=` or `abstype`) is copied in this way, the compiler will regard the `%insert` as having created a new type in each such case, not identical with that in the inserted file.]

(c) Last example (typeclash). Finally note that that it is illegal for the same original type to be imported twice into the same scope even under different names. Consider the following script

```
    %include "trees" shrub/tree Leaf/NILT
Fork/NODE -reflect
    %include "trees"
... (etc)
```

The first `%include` taken on its own is perfectly ok - we have imported the ``tree'` type, and renamed everything in it by using aliases. However we have also inherited the ``tree'` type under its original name, via the second `%include`. The compiler will reject the script with the following message:

```
TYPECLASH - the following type is multiply named:
'tree' of file "trees.m", as: shrub,tree;
typecheck cannot proceed - compilation abandoned
```

The rule that a type can have at most one name in a given scope applies to both algebraic types and abstract types (it does not apply to synonym types, because these are not ``real'` types but mere macro's - you can have any number of synonyms for ``tree'` in scope at the same time - as long as the underlying ``real'` type has a unique name).

Typeclashes are illegal in Miranda in order to preserve the following two principles. (i) In any given scope, each possible type must have a unique canonical form (obtained by expanding out synonyms, and renaming generic type variables in a standard way). (ii) Each object of a ``printable type'` must have, in any given scope, a unique external representation (ruling out multiply named constructors). The first principle is necessary to the functioning of the typechecker, the second is demanded by the requirement that the function ``show'` be referentially transparent.

.....

27/4 The `%free` directive and parametrised scripts

.....

It is permitted to construct a script containing definitions which are dependent on information which will be supplied only when the script is made the subject of a `%include` directive. Such a script is said to be parametrised. This is indicated by the presence in the script of a directive of the form

```
%free { signature }
```

where ``signature'` is a list of specifications of the identifiers for which bindings will be provided at `%include` time. A script may contain at most one `%free` directive, which must therefore give all the identifiers on which the script is parametrised. The `%free` directive may appear anywhere in the script, but for clarity it is recommended that you place it at or near the top.

For example a script (called "matrices" say) defining the notion of matrix sum and matrix product, for matrices of as-yet-unspecified element type, could be written as follows:-

```

%export matmult matadd

%free { elem :: type
        zero :: elem
        mult, add :: elem->elem->elem
      }

matrix == [[elem]]

matadd :: matrix->matrix->matrix
matadd xx yy = [[add a b|(a,b)<-zip2 x y|(x,y)<-zip2 xx
yy]

matmult :: matrix->matrix->matrix
matmult xx yy = outerprod innerprod xx (transpose yy)
innerprod x y = sum [mult a b|(a,b)<-zip2 x y]
                where
                sum = foldr add zero
outerprod f xx yy = [[f x y|y<-yy|x<-xx]

```

Note that the identifiers declared under %free may denote types as well as values. When we write a %include directive for the above script we must provide bindings for all of its free identifiers. The bindings are given in braces following the pathname (and before the aliases, if any). Thus:-

```

%include "matrices" {elem==num; zero=0; mult=*; add=+; }

```

In the scope of the script containing the above directive the identifiers ``matmult'` and ``addmult'` will be available at type `[[num]]->[[num]]->[[num]]` and will behave as if their definitions had been written using 0, (+), (*) in place of the identifiers zero, add, mult.

The order in which the bindings are given is immaterial (it need not be the order in which the identifiers occurred in the %free directive) but a binding must be given for each free identifier of the %included script. Note that the binding for a type is given using ``=='` and for a value using ``='`. If the types of all the bindings (taken together) are not consistent with the information given in the free directive of the %included script, or if any required binding is missing, the compiler will reject the %include directive as incorrect.

The main advantage of a parametrised script is that different bindings may be given for its free identifiers on different occasions. For example the same script "matrices" may be invoked with different bindings to provide a definition of matrix addition and multiplication over matrices with elements of type bool. Thus:-

```

%include "matrices" {elem==bool;zero=False; mult=&; add=\/;}

```

It is even possible to `%include` the same parametrised script twice in the same scope (presumably with different bindings for the free identifiers) but in this case it will be necessary to alias apart the two sets of exported identifiers to avoid a nameclash. So we might add ``b_matadd/matadd b_matmult/matmult'` to the above directive if it were being used in the same script as the previous one.

Miscellaneous points

By default the identifiers declared `%free` in a parametrised script are not exported from the script. As always this can be overridden by explicitly listing them in an `%export` directive.

Free typenames of non-zero arity are declared in the following style.

```
    %free { stack * :: type
           table * ** :: type
           ...
        }
```

The corresponding bindings could be as follows

```
    %include... {stack * == [*]; table * ** == [(*,**)]}; ... }
```

When a parametrised script exports a locally created typename (other than a synonym type), each instantiation of the script by a `%include` is deemed to create a NEW type (this is relevant to deciding whether or not two types are the same for the purpose of readopting a type orphan, see previous manual section). This is because the compiler assumes that an abstract or algebraic type defined in a parametrised script will in general have an internal structure that depends on the free identifiers.

Finally note that the bindings for the free identifiers of a parametrised script must always be given EXPLICITLY. For example suppose we wish to `%include` the file "matrices" in a script already containing a type called ``elem'` over which we intend to do matrix multiplication. We must write

```
    %include "matrices" {elem==elem; etc. }
```

The binding ``elem==elem'` is not redundant, nor is it cyclic, because the two ``elem's` involved refer to two different scopes (on the left of the binding, that of the includee, and on the right that of the script containing the directive).

.....

27/5 Separate compilation and '.x' files

.....

The Miranda compiler compiles to an intermediate code, based on combinators. When a Miranda expressions are evaluated in the course of a session this code is executed by an interpreter.

Since compilation is a complex process (involving lexical analysis, parsing, type checking and code-generation, as well as a number of other

minor steps) it is undesirable that the results of compiling a script should just be "thrown away" at the end of a session. To avoid unnecessary acts of recompilation the Miranda system maintains an object-code file in association with each source file containing a Miranda script.

For each source file, called say `script.m', the Miranda system will create an object code file, called `script.x'. No action is required by the user to keep these files up-to-date, since this is taken care of automatically by the Miranda system. The .x files are never referred to directly by the Miranda user, and you should not try to edit them (they contain binary data).

You may however safely remove any .x file (if for example you don't wish it to use up filespace) since this will at worst cause the Miranda compiler to do some extra work later to recreate it.

If you select a script as the current script of a Miranda session, and it has an up-to-date .x file, this will be loaded instead, avoiding recompilation. If the .x file does not exist, or any relevant source file has been modified since the .x file was created, the script will be recompiled (and a side effect of your having selected this source file as the current script will be to bring into existence an up-to-date .x file for it).

[Inductive definition - source file B is `relevant' to source file A iff file A %inserts or %includes B or any file to which B is relevant. For a discussion of `%include' and the other library directives see manual sections on `The Library Mechanism'.]

Note that compiling a script containing %include statements will have the side effect of triggering subsidiary compilation processes for any relevant source files which have been modified since their corresponding .x file was created. Users familiar with the UNIX program `make' will recognise this process as essentially the same as that which happens when a `makefile' is obeyed. In the case of Miranda however, the `make' process is fully automated by being built into the compiler.

More advanced information

If you want to check that a given Miranda script has an up-to-date object code file without entering a Miranda session, this can be accomplished from UNIX by calling mira with a special flag, thus

```
mira -make script.m
```

will force the existence of an up-to-date `script.x', performing all (and only) those compilations which are necessary. Any number of source files can be given after the -make flag (and as usual if a `.m' extension is omitted it will be added automatically).

Example:- to make sure every Miranda source file in your current directory has an up-to-date object code file, say `mira -make *.m'.

Applying mira -make to a `.x' file is equivalent to applying it to the corresponding `.m' file. So another way to make sure everything in your current directory is up-to-date is to say `mira -make *.x'. This has

the advantage that it will also remove any `.x` files whose `.m` files no longer exist.

In the best UNIX tradition `mira -make` does its work silently unless something is wrong. If the source files are all correct closed scripts with up-to-date `.x` files, `mira -make` says nothing at all. If recompilations are necessary it informs you which source files are being compiled, and, as a last step, the names of any scripts which contain errors or undefined names are listed, to `stdout`.

The exit status of a `'mira -make'` (relevant if you are a shell programmer, or wish to include a `'mira -make'` command in a makefile for a larger setup) is as follows. If (AFTER any necessary recompilations have been performed) all the source files have up-to-date `.x` files, and do not contain any syntax errors, type errors, or undefined names (these facts are recorded in `.x` files) the exit status will be zero (=ok), otherwise it will be 1.

It is possible to find out what names are exported from one or more Miranda scripts without entering a Miranda session by using the command

```
mira -exports files
```

(as always the `.m` extension is added automatically to each filename, if missing). This command first calls `'mira -make'` on each file, to make sure everything is uptodate, and then lists to standard output the exported names together with their types (one per line). If more than one file is specified each group of names will be preceded by the name of the file to which they appertain.

Note that the export profile of a script includes information about its free identifiers (if any).

It is also possible to find out the names of all files on which a given set of Miranda scripts depend, via `%include` and `%insert` statements, by using the command

```
mira -sources files
```

This lists to standard output, one per line, the names of all relevant source files. The standard environment, `<stdenv>`, is always omitted from the list.

Effect of `'mv'` and `'rm'`

A `.x` file records (inter alia) the names of all relevant source files relative to the directory in which it is stored, together with their `'date and time last modified'`. Note that the UNIX command `'mv'` does not alter the time-last-modified of the file being moved. So it is possible when moving a miranda source file (or a group of interdependant source files) from one directory to another to save `mira` the bother of recompiling them, simply by moving all the relevant `.x` files into the new directory along with the sources. (This doesn't work however if you change the name of any of the source files during the move.)

[Note that `'tar'` has the same property, so the up-to-date-ness of Miranda `.x` files is preserved across a tape dump.]

If you use `'rm'` to remove a Miranda source file, the next time you invoke `mira` with the (now non-existent) file as its current script, it will promptly remove the corresponding `.x` file. The logic of this is

as follows:- `.x` files must be kept up-to-date with their sources, and the way to make a `.x` file up-to-date with a non-existent source is to make it too non-existent. As a consequence it is not possible to send someone a Miranda object code file without the corresponding source (mira will delete it as soon as they try to use it!).

From some points of view this last feature might be regarded as a bug - a way round it may be provided in a later release of the Miranda system.

.....

28 The Mirand Standar Environment

.....

(C) Research Software Limited 1989

We give here, in alphabetical order, a brief explanation of all the identifiers in the Miranda standard environment, each followed by its definition (except in a few cases where the definition cannot conveniently be given in Miranda). The lines marked with a `>` in column one are formal program text, the other lines in the file are comment. Note that a number of the functions given here are defined internally (for speed) even though their definitions could have been given in Miranda - in these cases the Miranda definition is given as a comment. This is the standard environment of Miranda release two.

Note that this environment includes all the functions of Appendix B in Bird and Wadler `"An Introduction to Functional Programming"`.

`abs` takes the absolute value of a number - e.g. `abs (-3)` is 3, `abs 3.5` is 3.5

```
> abs::num->num
> abs x = -x, if x<0
>       = x, otherwise
```

`and` applied to a list of truthvalues, takes their logical conjunction.

```
> and::[bool]->bool
> and = foldr (&) True
```

`arctan` is the trigonometric function, inverse tangent. It returns a result in the range $-\pi/2$ to $\pi/2$. See also `sin`, `cos`.

```
> arctan::num->num ||defined internally
```

`bool` is the type comprising the two truthvalues.

```
bool ::= False | True ||primitive to Miranda
```

`char` is the type comprising ascii characters (e.g. `'a'`, `'\n'`).

```
char::type ||primitive to Miranda
```

`cjustify` applied to a number and a string, centre justifies the string in a field of the specified width. See also `ljustify`, `rjustify`, `spaces`.

```
> cjustify::num->[char]->[char]
> cjustify n s = spaces lmargin+++spaces rmargin
>           where
>           margin = n - # s
>           lmargin = margin div 2
>           rmargin = margin - lmargin
```

`code' applied to a character returns the integer which is its ascii code. E.g. code 'a' is 97. See also `decode'.

```
> code::char->num  ||defined internally
```

`concat' applied to a list of lists, joins them all together into a single list with `+'. E.g. concat [[1,2],[],[3,4]] is [1,2,3,4].

```
> concat::[[*]]->[*]
> concat = foldr (++) []
```

`const' is a combinator for creating constant-valued functions. E.g. (const 3) is the function that always returns 3.

```
> const::*->**->*
> const x y = x
```

`converse' is a combinator for inverting the order of arguments of a two-argument function.

```
> converse::(*->**->***)->**->*->***
> converse f a b = f b a
```

`cos' is the trigonometric cosine function, argument in radians.

```
> cos::num->num  ||defined internally
```

`decode' applied to a number between 0 and 255 returns the corresponding value of type char.

```
> decode::num->char  ||defined internally
```

`digit' is a predicate on characters. True if the character is a digit. See also `letter'.

```
> digit::char->bool
> digit x = '0'<=x<='9'
```

`drop' applied to a number and a list returns the list with that many elements removed from the front. If the list has less than the required number of elements, `drop' returns []. Example

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

See also `take'.

```
> drop::num->[*]->[*]  ||defined internally, as below
```

```
drop (n+1) (a:x) = drop n x
drop n x = x, if integer n
           = error "drop applied to fractional number", otherwise
```

`dropwhile' applied to a predicate and a list, removes elements from the front of the list while the predicate is satisfied. Example:

```
dropwhile digit "123gone" has value "gone"
```

See also `takewhile'.

```
> dropwhile::(*->bool)->[*]->[*]
> dropwhile f [] = []
> dropwhile f (a:x) = dropwhile f x, if f a
>                   = a:x, otherwise
```

`e' is a transcendental number, the base of natural logarithms.

```
> e::num
> e = exp 1
```

`entier' when applied to a number returns its integer part, meaning the largest integer not exceeding it. E.g. entier 1.0 is 1, entier 3.5 is 3, entier (-3.5) is -4. Notice that for Miranda the number `1' and the number `1.0' are different values - for example they yield different results under the `integer' test. However `1=1.0' yields True, because of the automatic conversion from integer to float.

```
> entier::num->num ||defined internally
```

A useful fact about `entier', which relates it to the operators div and mod, is that the following law holds for any integers a, b with b \neq 0 and a/b within the range for which integers can be represented exactly as fractional numbers

```
a div b = entier (a/b)
```

`error' applied to a string creates an error value with the associated message. Error values are all equivalent to the undefined value - any attempt to access the value causes the program to terminate and print the string as a diagnostic.

```
> error::[char]->* ||defined internally
```

`exp' is the exponential function on real numbers. See also `log'.

```
> exp::num->num ||defined internally
```

`filemode' applied to a string representing the pathname of a UNIX file, returns a string of length four giving the access permissions of the current process to the file. The permissions are encoded as (in this order) "drwx", any permission not granted is replaced by a '-' character. If there is no file at pathname p, filemode p returns the empty string. Example

```
member (filemode f) 'w'
```

tests f for write permission. See also `getenv', `read', `system'.

```
> filemode::[char]->[char] ||defined internally
```

`filter' applied to a predicate and a list, returns a list containing only those elements that satisfy the predicate. Example

```
filter (>5) [3,7,2,8,1,17] has value [7,8,17]
```

```
> filter::(*->bool)->[*]->[*]
> filter f x = [a | a<-x; f a]
```

`foldl' folds up a list, using a given binary operator and a given start value, in a left associative way. Example:

```
foldl op r [a,b,c] = ((r $op a) $op b) $op c)
```

But note that in order to run in constant space, foldl forces `op' to evaluate its first parameter. See the definitions of `product', `reverse', `sum' for examples of its use. See also `foldr'.

```
> foldl::(*->**->*)->*->[**]->* ||defined internally, as below
```

```
foldl op r [] = r
foldl op r (a:x) = strict (foldl op) (op r a) x
                    where
                        strict f x = seq x (f x)
```

WARNING - this definition of foldl differs from that in older versions of Miranda. The one given here is the same as that in Bird and Wadler. The old definition had the two args of `op' reversed. That is:-

```
old_foldl op r = new_foldl (converse op) r
```

the function `converse' has been added to the standard environment.

`foldl1' folds left over non-empty lists. See the definitions of `max', `min' for examples of its use.

```
> foldl1::(*->*->*)->[*]->* ||defined internally, as below
```

```
foldl1 op (a:x) = foldl op a x
foldl1 op [] = error "foldl1 applied to []"
```

`foldr' folds up a list, using a given binary operator and a given start value, in a right associative way. Example:

```
foldr op r [a,b,c] = a $op (b $op (c $op r))
```

See the definitions of `and', `concat', `or', for examples of its use.

```
> foldr::(*->**->**->[*]->**) ||defined internally, as below
```

```
foldr op r [] = r
foldr op r (a:x) = op a (foldr op r x)
```

`foldr1' folds right over non-empty lists.

```
> foldr1::(*->*->*)->[*]->*
> foldr1 op [a] = a
> foldr1 op (a:b:x) = op a (foldr1 op (b:x))
> foldr1 op [] = error "foldr1 applied to []"
```

`force' applied to any data structure, returns it, but forces a check that every part of the structure is defined. Example

```
hd(force x)
```

returns the hd of x, but fully evaluates x first (so x must be finite). See also `seq'. Notice in particular the idiom `seq (force a) b' which returns `b' but only after fully evaluating `a'.

```
> force::*->* ||defined internally
```

`fst' returns the first component of a pair. See also `snd'.

```
> fst::(*,**) -> *
> fst (a,b) = a
```

`getenv' looks up a string in the user's UNIX environment. Example

```
getenv "HOME"
```

returns the pathname of your home directory. [If you want to see what else is in your UNIX environment, say `printenv' as a UNIX command.]

```
> getenv::[char]->[char] ||defined internally
```

`hd' applied to a non empty list, returns its first element. It is an error to apply `hd' to the empty list, []. See also `tl'.

```
> hd::[*]->*
> hd (a:x) = a
> hd [] = error "hd []"
```

`hugenum' is the largest fractional number that can exist in this implementation (should be around 1e308 for IEEE standard 64 bit floating point, will be around 1e38 for the VAX). See also `tinynum'.

```
> hugenum::num ||defined internally
```

`id' is the identity function - applied to any object it returns it.

```
> id::*->*
> id x = x
```

`index' applied to a (finite or infinite) list, returns a list of its legal subscript values, in ascending order. E.g. index "hippopotamus" is [0,1,2,3,4,5,6,7,8,9,10,11].

```
> index::[*]->[num]
> index x = f 0 x
>           where
>           f n [] = []
>           f n (a:x) = n:f(n+1)x
```

`init' is dual to `tl', it returns a list without its last component. Example

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

See also `last'. [Note, by the `dual' of a list processing function we mean the function which does the same job in a world where all lists have been reversed.]

```
> init::[*]->[*]
> init (a:x) = [], if x=[]
>           = a:init x, otherwise
> init [] = error "init []"
```

`integer' is a predicate on numbers. True if and only if the number is not fractional.

```
> integer::num->bool ||defined internally
```

`iterate' - iterate f x returns the infinite list [x, f x, f(f x), ...]
Example, iterate (2*) 1 yields a list of the powers of 2.

```
> iterate::(*->*)->*->[*]  
> iterate f x = x:iterate f(f x)
```

`last' applied to a non empty list returns its last element. This function is the dual of `hd'. Note that for any non-empty list x
(init x ++ [last x]) = x

```
> last::[*]->*    ||defined internally, as below
```

```
last x = x!(#x-1)
```

`lay' applied to a list of strings, joins them together after appending a newline character to each string. Example

```
lay ["hello","world"] = "hello\nworld\n"
```

Used to format output thus,

```
lay(map show x)
```

as a top level expression, causes the elements of the list x to be printed one per line. See also `layn', `lines'.

```
> lay::[[char]]->[char]  
> lay [] = []  
> lay (a:x) = a++"\n"++lay x
```

`layn' is similar to `lay', but produces output with numbered lines.

```
> layn::[[char]]->[char]  
> layn x = f 1 x  
>         where  
>         f n [] = []  
>         f n (a:x) = rjustify 4 (show n) ++") "++a++"\n"++f (n+1) x
```

`letter' is a predicate on characters. True if the character is a letter.

```
> letter::char->bool  
> letter c = 'a'<=c<='z' \/ 'A'<=c<='Z'
```

`limit' applied to a list of values, returns the first value which is the same as its successor. Useful in testing for convergence. For example the following Miranda expression computes the square root of 2 by the Newton-Raphson method

```
limit [x | x<-1, 0.5*(x + 2/x).. ]
```

```
> limit::[*]->*  
> limit (a:b:x) = a, if a=b  
>                = limit (b:x), otherwise  
> limit other = error "incorrect use of limit"
```

`lines' applied to a list of characters containing newlines, returns a list of lists, by breaking the original into lines. The newline characters are removed from the result. Example, `lines' applied to

```
"hello world\nit's me,\neric\n"
```

returns ["hello world","it's me","eric"]. Note that `lines' treats newline as a terminator, not a separator (although it will tolerate a

missing '\n' on the last line).

```
> lines::[char]->[[char]]
> lines [] = []
> lines (a:x) = []:lines x, if a=='\n'
>             = (a:x1):xrest, otherwise
>             where
>             (x1:xrest) = lines x, if x~=[]
>                       = []:[], otherwise
>                       ||this handles missing '\n' on last line
```

Note that the inverse of `lines' is the function `lay', in that applying `lay' to the output of `lines' will restore the original string (except that a final newline will be added, if missing in the original string).

`ljustify' applied to a number and a string, left justifies the string in a field of the specified width.

```
> ljustify::num->[char]->[char]
> ljustify n s = s++spaces(n - # s)
```

`log' applied to a number returns its natural logarithm (i.e. logarithm to the base `e'). It is the inverse of the exponential function, `exp'. See also log10. Note that the log functions use a different algorithm when applied to integer arguments (rather than just converting to float first) so it is possible to take log, or log10, of very large integers.

```
> log::num->num      ||defined internally
```

`log10' applied to a number returns its logarithm to the base 10.

```
> log10::num->num    ||defined internally
```

`map' applied to a function and a list returns a copy of the list in which the given function has been applied to every element.

```
> map::(*->**) -> [*] -> [**]
> map f x = [f a | a<-x]
```

`map2' is similar to `map', but takes a function of two arguments, and maps it along two argument lists. We could also define `map3', `map4' etc., but they are much less often needed.

```
> map2::(*->**->***)->[*]->[**]->[***]
> map2 f x y = [f a b | (a,b)<-zip2 x y]
```

Note: the Bird and Wadler function `zipwith' is just an uncurried version of `map2', that is `zipwith f (x,y)' means `map2 f x y'.

`max' applied to a list returns the largest element under the built in ordering of `>'. Examples

```
max [1,2,12,-6,5] = 12
max "hippopotamus" = 'u'
```

See also `min', `sort'.

```
> max::[*]->*
> max = foldl1 max2
```

`max2' applied to two values of the same type returns the larger under the built in ordering of '>'. See also `min2'.

```
> max2::*->*->*
> max2 a b = a, if a>=b
>           = b, otherwise
```

`member' applied to a list and a value returns True or False as the value is or not present in the list.

```
> member::[*]->*->bool
> member x a = or (map (=a) x)
```

`merge' applied to two sorted lists merges them to produce a single sorted result. Used to define `sort', see later.

```
> merge :: [*]->[*]->[*] ||defined internally, as below
```

```
merge [] y = y
merge (a:x) [] = a:x
merge (a:x) (b:y) = a:merge x (b:y), if a<=b
                   = b:merge (a:x) y, otherwise
```

`min' applied to a list returns its least member under '<'.

```
> min::[*]->*
> min = foldl1 min2
```

`min2' applied to two values of the same type returns the smaller under the built in ordering of '<'.

```
> min2::*->*->*
> min2 a b = b, if a>b
>           = a, otherwise
```

`mkset' applied to a list returns a copy of the list from which any duplicated elements have been removed. A list without duplications can be used to represent a set, whence the name. Works even on infinite list, but (beware) takes a time quadratic in the number of elements processed.

```
> mkset::[*]->[*]
> mkset [] = []
> mkset (a:x) = a:filter (~=a) (mkset x)
```

`neg' is a function of one numeric argument, with the same action as the unary '-' operator.

```
> neg::num->num
> neg x = -x
```

`num' is the type comprising both integer and fractional numbers (such as 42, -12.73e8).

```
num::type ||primitive to Miranda
```

`numval' converts a numeric string to the corresponding number - can cope with sign, decimal point and scale factor (uses same rules as Miranda compiler). Strings containing inappropriate characters cause an error (exception - leading white space is harmless).

```
> numval::[char]->num ||defined internally
```

`or' applied to a list of truthvalues, takes their logical disjunction.

```
> or::[bool]->bool
> or = foldr (\/) False
```

`pi' is the well known real number (the ratio of the circumference of a circle to its diameter).

```
> pi::num
> pi = 4*arctan 1
```

`postfix' takes an element and a list and adds the element to the end of the list. This is the dual of the prefix operator, `:'.

```
> postfix::*->[*]->[*]
> postfix a x = x ++ [a]
```

`product' applied to list of numbers returns their product. See also `sum'.

```
> product::[num]->num
> product = foldl (*) 1
```

`read' returns the contents of file with a given pathname. Provides an interface to the UNIX filing system. If the file is empty `read' returns [], but if the file does not exist, or lacks read permission, `read' causes an error. See also `filemode', `getenv'.

```
> read::[char]->[char] ||defined internally
```

`rep' applied to a number and a value, returns a list containing the specified number of instances of the value. (The name is short for `replicate'.) Example

```
rep 6 'o' = "oooooooo"
```

See also `repeat'.

```
> rep::num->*->[*]
> rep n x = take n (repeat x)
```

`repeat' applied to a value returns an infinite list, all of whose elements are the given value.

```
> repeat::*->[*]
> repeat x = xs
> where xs = x:xs
```

`reverse' applied to any finite list returns a list of the same elements in reverse order.

```
> reverse::[*]->[*]
```

```
> reverse = foldl (converse(:)) []
```

`rjustify' applied to a number and a string, right justifies the string in a field of the specified width.

```
> rjustify::num->[char]->[char]
> rjustify n s = spaces(n - # s)++s
```

`scan op r' applies `foldl op r' to every initial segment of a list. For example `scan (+) 0 x' computes running sums.

```
> scan::(*->**->*)->*->[**]->[*]
> scan op = g
>     where
>     g r = (r:). rest
>           where
>           rest [] = []
>           rest (a:x) = g (op r a) x
```

There is another way to explain `scan', which makes it clearer why it is useful. Let s_0 be the initial state of an automaton, and $f::state \rightarrow input \rightarrow state$, its state transition function - then `scan f s_0 ' is a function that takes a list of inputs for the automaton and returns the resulting list of states, starting with s_0 .

`seq' applied to two values, returns the second but checks that the first value is not completely undefined. Sometimes needed, e.g. to ensure correct synchronisation in interactive programs.

```
> seq::*->**->** ||defined internally
```

show is a keyword denoting a family of operators for converting values of different types to their print representations. See manual section on `show' for more details.

`shownum' applied to a number returns as a string a standard print representation for it. A special case of the operator `show'. Applied to fractional numbers `shownum' gives 12 decimal places (less any trailing zeros), using a format appropriate to the size of number. For more detailed control over number format see `showfloat', `showscaled'.

```
> shownum::num->[char] ||defined internally,
```

`showfloat p x' returns as a string the number x printed in floating point format, that is in the form "digits.digits", where the integer p (≥ 0) gives the number of digits after the decimal point.

```
> showfloat::num->num->[char] ||defined internally,
```

`showscaled p x' returns as a string the number x printed in scientific format, that is in the form "n.nnnnnne[+/-]nn", where the integer p (≥ 0) gives the number of digits required after the decimal point.

```
> showscaled::num->num->[char] ||defined internally,
```

`sin' is the trigonometric sine function, argument in radians.

```
> sin::num->num ||defined internally
```

`snd' returns the second component of a pair.

```
> snd::(*,**)->**
> snd (a,b) = b
```

`sort' applied to any finite list sorts the elements of the list into ascending order on the built in '<' relation. Note that you cannot sort a list of functions. Example

```
sort "hippopotamus" = "ahimooppstu"
```

The following definition uses merge-sort, which has $n \log n$ worst-case behaviour.

```
> sort::[*]->[*]
> sort x = x, if n<=1
>         = merge (sort(take n2 x)) (sort(drop n2 x)), otherwise
>         where
>           n = #x
>           n2 = n div 2
```

`spaces' applied to a number returns a list of that many spaces.

```
> spaces::num->[char]
> spaces n = rep n ' '
```

`sqrt' is the square root function on (integer or fractional) numbers. The result is always fractional.

```
> sqrt::num->num ||defined internally
```

`subtract' is a name for (converse) infix minus. Needed because you cannot form postsections in '-'. (See manual entry on `sections'.) Example

```
subtract 3
```

is the function that subtracts 3.

```
> subtract::num->num->num
> subtract x y = y - x
```

`sum' applied to list of numbers returns their sum.

```
> sum::[num]->num
> sum = foldl (+) 0
```

`sys_message' is an algebraic type containing a family of constructors used to control output to UNIX files. See manual section on Output to UNIX files for details.

```
> sys_message ::= Stdout [char] | Stderr [char] | Tofile [char] [char] |
>               Closefile [char] | Appendfile [char] | System [char] |
>               Exit num
```

`system' applied to a string causes the string to be executed as a UNIX shell command (by `sh'). The result returned is a 3-tuple, comprising the standard_output, error_output, and exit_status respectively, resulting from the execution of the UNIX command. See manual section on

Input from UNIX files etc for more details.

```
> system::[char]->([char],[char],num)  ||defined internally
```

`take' applied to a number and a list returns the specified number of elements from the front of the list. If the list has less than the required number of elements, `take' returns as many as it can get.

Examples

```
take 2 [1,2,3,4] = [1,2]
take 7 "girls" = "girls"
```

```
> take::num->[*]->[*]  ||defined internally, as below
```

```
take (n+1) (a:x) = a:(take n x)
take n x = [], if integer n
           = error "take applied to fractional number", otherwise
```

`takewhile' applied to a predicate and a list, takes elements from the front of the list while the predicate is satisfied. Example:

```
takewhile digit "123gone" has value "123"
```

```
> takewhile::(*->bool)->[*]->[*]
> takewhile f [] = []
> takewhile f (a:x) = a:takewhile f x, if f a
>                   = [], otherwise
```

`tinynum' is the smallest positive fractional number that can be distinguished from zero in this implementation (should be around 1e-324 for IEEE standard 64 bit floating point).

```
> tinynum::num  ||defined internally
```

`tl' applied to a non empty list returns the list without its first element. Example, tl "snow" is "now".

```
> tl::[*]->[*]
> tl (a:x) = x
> tl [] = error "tl []"
```

`transpose' applied to a list of lists, returns their transpose (in the sense of matrix transpose - rows and columns are interchanged). Example

```
transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]
```

The following definition is slightly more subtle than is at first sight necessary, in order to deal correctly with `upper triangular' matrices. Example

```
transpose [[1,2,3],[4,5],[6]] = [[1,4,6],[2,5],[3]]
```

```
> transpose :: [[*]]->[[*]]
> transpose x = [], if x'=[]
>              = map hd x':transpose(map tl x'), otherwise
>              where
>              x' = takewhile (~=[]) x
```

It might be thought that this function belongs in a specialised library of matrix handling functions, but it has been found useful as a general purpose list processing function, whence its inclusion in the standard environment.

`undef' is a name for the completely undefined value. Any attempt access it results in an error message. Note that `undef' belongs to every type.

```
> undef::*
> undef = error "undefined"
```

`until' applied to a predicate, a function and a value, returns the result of applying the function to the value the smallest number of times necessary to satisfy the predicate. Example

```
until (>1000) (2*) 1 = 1024
```

```
> until::(*->bool)->(*->*)->*->*
> until f g x = x, if f x
>                = until f g (g x), otherwise
```

`zip2' applied to two lists returns a list of pairs, formed by tupling together corresponding elements of the given lists. Example

```
zip2 [0..3] "type" = [(0,'t'),(1,'y'),(2,'p'),(3,'e')]
```

This function is often useful in list comprehensions, where it provides an idiom for accessing two generators simultaneously. For example the following expression returns the scalar product of x and y (x,y::[num])

```
sum [ a*b | (a,b) <- zip2 x y ]
```

```
> zip2::[*]->[**]->[(*,**)]  ||defined internally, as below
```

```
zip2 (a:x) (b:y) = (a,b):zip2 x y
zip2 x y = []
```

Note that if the lists being zipped are of different lengths, the length of the result is that of the shortest list (this holds for zip2 and all the following zip functions).

The function `zip3' is analogous but takes three lists and returns a list of 3-tuples. Similarly for `zip4', `zip5', `zip6' - zip functions above zip6 are not provided in the standard environment.

```
> zip3 (a:x) (b:y) (c:z) = (a,b,c):zip3 x y z
> zip3 x y z = []
> zip4 (a:w) (b:x) (c:y) (d:z) = (a,b,c,d):zip4 w x y z
> zip4 w x y z = []
> zip5 (a:v) (b:w) (c:x) (d:y) (e:z) = (a,b,c,d,e):zip5 v w x y z
> zip5 v w x y z = []
> zip6 (a:u)(b:v)(c:w)(d:x)(e:y)(f:z) = (a,b,c,d,e,f):zip6 u v w x y z
> zip6 u v w x y z = []
```

The following is included for compatibility with Bird and Wadler. The normal Miranda style is to use the curried form `zip2'.

```
> zip::([*],[**])->[(*,**)]
> zip (x,y) = zip2 x y
```

End of definitions of the standard environment

```
::::::::::::::::::::::::::
```

29 Literate scripts (an alternative comment convention)

.....

The standard comment convention for Miranda scripts is that anything rightwards from a pair of vertical bars to the end of a line is taken to be comment and ignored by the compiler, thus

```
||This is a comment
```

Everything else in the script is taken to be formal program text. An inverted style of commenting is also available in Miranda, permitting the construction of a "literate script" (the name is taken from Professor Donald Knuth's idea of "literate programming"). In a literate script EVERYTHING is assumed to be comment, except for lines marked with the formalising symbol '>' in column 1. For example the following lines

```
> fac 0 = 1
> fac (n+1) = (n+1)*fac n
```

would be taken as formal program text - and could be preceded and/or followed by some narrative explaining what the factorial function is and why we define it in this way.

To minimise the danger that you will accidentally omit the '>' from one line of your formal text without the compiler noticing that something is wrong, the following additional rule applies to Miranda literate scripts - whenever a group of lines of formal program text is preceded or followed by some lines of "narrative", the two types of text must be separated by at least one blank line. You will see that this has been done for the definition of factorial given above. (Definition - a "blank line" is one containing only white space.)

Within the formal sections of a literate script the standard comment convention still works. For example

```
> result = sum [fac n | n <- [1..50]] ||NB this is a large number!
```

The compiler takes a decision on which comment convention applies by looking at the first line of a script. If this has a '>' in column 1, then it is a literate script, otherwise the compiler assumes it is a conventional script. Typically the first line of a literate script will just be a comment, eg

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

In fact this manual section is a legal Miranda script, defining `fac' and `result' (see first line).

An alternative convention is based on the name of the file - if this ends in `.lit.m' then it is assumed to be a literate script, independently of the form of the first line. This makes it possible to have literate scripts which begin in `narrative' mode.

As an aid to maintaining good layout in literate scripts, a simple text formatting program, called `just' (short for justify), is supplied with the Miranda system. This leaves untouched the formal sections of the script and formats the narrative parts to specified width (default 72).

If you use ``vi'` to edit your scripts note that the ``just'` program can be called from within the editor by saying, e.g.

```
:1,$!just
```

[Warning - do NOT use ``just'` on non-literate scripts, it will make a mess of them!]

There is a UNIX manual page for ``just'` which gives details of its behaviour - to see the manual page from within Miranda say ``!man just'`. Note that ``just'` is a general purpose text formatting tool, and is not in any way Miranda-specific.

As an additional aid to the use of document preparation tools in conjunction with Miranda scripts, the Miranda compiler will recognise underlined keywords. This applies both to reserved words, such as ``div'` and ``mod'` and to directives such as ``%export'` (underlining of the initial ``%'` is optional). The style of underlining accepted is ``backspace-underline-character'` as generated by `nroff/troff`. It will also recognise the underlined symbols `>` and `<` as being equivalent to `>=`, `<=` respectively. This works in both literate scripts and scripts using the standard comment convention.

Note on %insert and lite rate scripts

An `%insert` directive in a literate script will be effective provided it occurs in a line beginning with ``>'` (otherwise it is a comment and will be ignored). The inserted text may itself use either the standard or the literate comment convention (decided by the name of the file and whether or not its first line begins with `'>'`).

Similarly, text `%inserted` into a non-literate script may use either the standard or the literate comment convention (again decided by the name of the file and the form of its first line).

Using LaTeX with Miranda literate scripts

Because of the `.lit.m'` convention it is possible for a file to be both a Miranda script and a LaTeX source file. In such a case the sections of formal Miranda text (recognised by the Miranda compiler by the ``>'` in column 1) will be surrounded by the LaTeX commands

```
\begin{verbatim}
```

```
\end{verbatim}
```

A similar arrangement can be made for `troff`.

This works fine - but you may wish to do even better, and have reserved words in the Miranda text underlined in your printed document. The program ``mtotex'` takes a `.m'` file containing an ordinary Miranda literate script, and creates a `.tex'` file of the same name, in which various LaTeX commands have been inserted, including to underline reserved words in program text. See UNIX manual page for ``mtotex'` for details. Note that the `.tex'` file created by `mtotex` is not itself a legal Miranda script.

Acknowledgements

The `'>'` inverse-comment convention (and the "blank line" rule) are due to Richard Bird and Philip Wadler of Oxford University Programming Research Group, and were first used in their language "Orwell". The ``mtotex'` program was supplied by John Cupitt of the University of Kent.

.....

30 Some guidelines on good programming style in Miranda

.....

Functional programming is still at an early stage of development and some heterogeneity of programming style is therefore inevitable (and desirable). Nevertheless a certain amount is known, and there is no need for every newcomer to functional programming to discover all the pitfalls by trial and error. We give here a series of suggested guidelines for good programming style in Miranda. The list is not meant to be exhaustive.

These rules are also not intended to be followed rigidly in all cases, regardless of conflicting considerations. That is why they are only suggestions for good style and not grammar rules.

Avoid the indiscriminate use of recursion

A Miranda script that consists of large number of functions which call each other in an apparently random fashion is no easier to understand than, say, a piece of FORTRAN code which is written as a rat's nest of GOTO statements. An excessive reliance on recursion (especially mutual recursion) can be an indication of a weak programming style. Some pointers:

Use list comprehensions, `'..'` lists, and library functions, in preference to ad-hoc recursion. For example it is probably clearer to define factorial by writing

```
fac n = product[1..n]
```

than to define it from first principles, as

```
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

and to define the cartesian product of two lists by a list comprehension, thus

```
cp x y = [(a,b)|a<-x;b<-y]
```

is certainly a lot clearer than the recursive definition,

```
cp (a:x) y = f y ++ cp x y
  where
    f (b:y) = (a,b): f y
    f [] = []

cp [] y = []
```

The standard environment contains a number of useful list processing functions (eg `map filter reverse foldr foldl`) with whose properties it is worth becoming familiar. They capture common patterns of recursion over lists, and can often be used to simplify your code, and reduce the reliance on 'ad-hoc' recursion. Programs using list comprehensions and standard functions are also likely to run faster (on the current implementation) than equivalent programs using ad-hoc recursion.

The standard environment is only a basic collection of useful general purpose functions. As you get used to programming in Miranda you will probably begin to discover other useful functions that express common

patterns of recursion (perhaps over data structures other than lists). It is a good practice to collect such functions in libraries (together with some explanations of their properties) so that you can reuse them, and share them with others. Not all of them will survive the test of time, but it cannot hurt to experiment.

To cause the definitions from such a library to be in scope in another script you would use a ``%include'` directive (see manual section on library directives).

Avoid unnecessary nesting of definitions

Scripts that get deeply nested in where-clauses are harder to understand, harder to reason about formally, harder to debug (because functions defined inside where's cannot be exercised separately) slower to compile, and generally more difficult to work with.

A well structured script will consist of a series of top-level definitions, each of which (if it carries a where-clause at all) has a fairly small number of local definitions. A third level of definition (where inside where) should be used only very occasionally. [And if you find yourself getting nested four and five levels deep in block structure you can be pretty sure that your program has gone badly out of control.]

A function should normally be placed inside a where clause only if it is logically necessary to do so (which will be the case when it has a free variable which is not in scope outside the where clause). If your script consists, of say six functions, one of which solves a problem and the other five of which are auxiliary to it, it is probably not a good style to put the five subsidiary functions inside a where clause of the main one. It is usually better to make all six top level definitions, with the important one written first, say.

There are several reasons for this. First that it makes the program easier to read, since it consists of six separate chunks of information rather than one big one. Second that the program is much easier to debug, because each of its functions can be exercised separately, on appropriate test data, within a Miranda session. Third that this program structure is more robust for future development - for example if we later wish to add a second ``main'` function that solves a different problem by using the same five auxiliary functions in another way, we can do so without having to restructure any existing code.

There is a temptation to use ``where'` to hide information that is not relevant at top-level. This may be misguided (especially if it leads to code with large and complex where-clauses). If you don't wish all of your functions or data structures to be "visible" from outside, the proper way to do this is to include a ``%export'` directive in the script.

Note also that (in the current implementation) functions defined inside a "where" clause cannot have their types explicitly specified. This is a further reason to avoid putting structure inside a where clause that does not logically have to be there.

Specify the types of top level identifiers

The Milner type discipline is an impressive advance in compiler technology. It is also a trap for the unwary. The fact that the

Miranda compiler will accept several hundred lines of code without a single type specification, and correctly infer the types of all the identifiers does NOT mean that it is sensible to write code with no type information. (Compare: compilers will also accept large programs with no comments in, but that doesn't make such programs sensible.)

For other than fairly small scripts it is good style to insert an explicit specification of the type of any top level identifier whose type is not immediately apparent from its definition. Type specifications look like this

```
ack::num->num->num
```

says that 'ack' is a function taking two numbers and returning a number. A type specification can occur anywhere in a script, either before or after the definition of the corresponding identifier, but common sense suggests that the best place for it is just before the corresponding definition.

If in doubt it is always better to put in a type specification than to leave it out. The compiler may not need this extra type information but human beings definitely do. The extra type information becomes particularly important when your code reaches the level of complexity at which you start to make type errors.

If your script contains a type error it is unreasonable to expect the compiler to correctly locate the real source of the error in the absence of explicit type declarations. A type error means different parts of your code are inconsistent with one another in their use of identifiers - if you have not given the compiler any information about the intended use of an identifier, you cannot expect it to know which of several conflicting uses are the 'wrong' ones. In such a case it can only tell you that something is wrong, and indicate the line on which it first deduced an inconsistency - which may be many lines later than the 'real' error. Explicit type declarations make it much more likely that the compiler will spot the 'real error' on the line where it actually occurs.

Code containing explicit type information is also incomparably easier for other people to read.

Use safe layout

This is a point to do with the operation of the offside rule. It is most easily explained by means of an example. Consider the following definition, here assumed to be part of a larger script

```
hippo = (rhino - swan)/piglet
      where
      piglet = 17
      rhino = 63
      swan = 29
```

Some time after writing this we carry out a global edit to expand 'hippo' to 'hippopotamus'. The definition now looks like this.

```
hippopotamus = (rhino - swan)/piglet
              where
              piglet = 17
              rhino = 63
```

swan = 29

the where-clause has become offside, and the definition will no longer compile. Worse, it is possible (with a little ingenuity) to construct examples of layout where changing the length of an identifier will move a definition from one level of scope to another, so that the script still compiles but now has a different meaning!!! Replacing an identifier by a shorter one can cause similar difficulties with layout.

The layout of the `hippo' definition was unsafe, because the level of indentation depended on the length of an identifier. There are several possible styles of `safe' layout. The basic rule to follow is:

Whenever a right hand side goes on for more than one line (because it consists of a set of guarded cases, or because it carries a where clause, or just because it is an expression too big to fit on one line), you should take a newline BEFORE starting the rhs, and indent by some standard amount (not depending on the width of the lhs).

There are two main styles of safe layout, depending on whether you take the newline before or after the `=' of the definition. Here are two possible safe layouts for the `hippo' definition

```
hippo =
  (rhino - swan)/piglet
  where
  piglet = 17
  rhino = 63
  swan = 29
```

```
hippo
  = (rhino - swan)/piglet
  where
  piglet = 17
  rhino = 63
  swan = 29
```

The reason that either style can be used is that the boundary, for offside purposes, of a right hand side, is set by the first symbol of the rhs itself, and not by the preceding `=' sign.

Both of these layouts have the property that the parse cannot be affected by edits which alter the lengths of one or more identifiers. Either of these layout styles also have the advantage that successive levels of indentation can move to the right by a fixed step - this makes code easier to read and lessens the danger that your layout will `fall off' the right hand edge of the screen (although if you follow the advice given earlier about avoiding deeply nested block structure this is in any case unlikely to be a problem).

It would be convenient if there was a program for reformatting Miranda scripts with a standard layout. Apart from ensuring that the layout was `safe' in the above sense, it might make it easier for people to read each other's code. A layout program of this kind may be provided in later releases of the system.

Acknowledgement: The `hippopotamus' example (and the problem of unsafe

layout) was first pointed out by Mark Longley of the University of Kent.

Write order independent code

When defining functions by pattern matching it is best (except in a few cases where it leads to real clumsiness of expression) to make sure the patterns are mutually exclusive, so it does not matter in what order the cases are written.

For the same reason it is better style to use sets of guards which are composed of mutually exclusive boolean expressions. The keyword ``otherwise'` sometimes helps to make this less painful.

By way of illustration of some of the issues here is a good definition of a function ``merge'` which combines two already sorted lists into a single sorted result, eliminating duplicates in the process

```
merge [] y = y
merge (a:x) [] = (a:x)
merge (a:x) (b:y)
  = a:merge x (b:y), if a<b
  = b:merge (a:x) y, if a>b
  = a:merge x y, if a=b
```

First note the use of mutually exclusive sets of patterns (it was tempting to write ``merge x [] = x'` as the second case, but the above is probably better style). Note also that we didn't use ``otherwise'` as the last guard here because it would have spoiled the symmetry of the three tests.

A related issue to these is that where a function is not everywhere defined on its argument type, it is good practice to insert an explicit error case. For example the definition given in the standard environment for ``hd'`, the function which extracts the first element of a list, is

```
hd (a:x) = a
hd [] = error "hd []"
```

Of course if a function is applied to an argument for which no equation has been given, the Miranda system will print an error message anyway, but one advantage of putting in an explicit call to ``error'` is that the programmer gets control of the error message. The other (and perhaps main) advantage is that for someone else reading the script, it explicitly documents the fact that a certain use of the function is considered an error.

Note by the way, that for reasons of upwards compatability with earlier versions of Miranda, the keyword ``if'` in guards is optional - but it is definitely better style to put it in. You can tell the compiler to treat missing ``if'` as a syntax error by using the command `/strictif (/nostrictif to undo)` in a Miranda session. Once set this flag is remembered for subsequent sessions.

Obsolete Language Features

Laws and strictness annotations (see manual section on algebraic data types) are no longer part of the Miranda language. They will continue to be supported by the compiler for a transitional period, but should not be taught to new users.

.....

31 UNIX/Miranda system interface information

.....

1. Input from UNIX files etc
2. Output to UNIX files etc
3. Reading with interpretation (``readvals'` and `$+`)
4. Using Miranda to build UNIX commands
5. How to change the default editor
6. How to alter sizes of workspaces
7. Flags, setup files etc
8. Environment variables used by Miranda

.....

31/1 Input from UNIX files etc.

.....

The following Miranda functions provide an interface to the UNIX file system from within Miranda expressions:

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

This takes a string valued argument, which it treats as a UNIX pathname, and returns the contents of the file or device of that name, also as a string (i.e. as a list of characters). There is no end-of-file character, the termination of the file is indicated simply by the end of the list of characters. The Miranda evaluation terminates with an error message if the file does not exist or the user does not have read permission for it.

A special case - the notation ``$-` denotes the contents of the standard input, as a list of characters. Note that multiple occurrences of ``$-` always denote a single shared input stream. So for example `($- ++ $-)` reads one lot of data from the terminal and duplicates it.

```
filemode :: [char]->[char]
```

Takes a pathname and returns a string representing the access permissions of the current process to the file of that name. The string is empty if the file does not exist, otherwise it is of length four containing, in order, the following characters - 'd' if the file is a directory, 'r' if it is readable, 'w' if it is writeable, 'x' if it is executable. Each character not applicable is replaced by '-'. So for example "drwx" is the filemode of a directory with all access permissions, while "-rw-" is the filemode of a normal file with read and write but no execute permission.

```
getenv :: [char]->[char]
```

Looks up the string associated with a given name in the current UNIX environment (see man (2) getenv in the UNIX manual system). For example

```
getenv "HOME"
```

returns the name of the current home directory. Returns the empty string if the given name not present in the environment.

```
system :: [char]->([char],[char],num)
```

The effect of `system string` is that a UNIX process is forked off to execute `string` as a shell command (by `/bin/sh`). The result of the call to `system` is a triple containing the standard output, error output, and exit_status, respectively, resulting from the execution of the UNIX command. (The exit_status of a UNIX command is a number in the range 0..127, with a non-zero exit status usually indicating some kind of abnormal event.) Note that inspecting the exit_status will force the Miranda process to wait until the execution of the shell command has completed - otherwise the two processes proceed concurrently.

If the attempt to set up a shell process fails, `system` returns the result ([],errmess,-1), where errmess is an error message.

WARNING - the function `system` provides a very general interface to UNIX. Obviously, this can be abused to cause the evaluation of a Miranda expression to have side effects on the state of the filing system. It is not intended to be used in this way - `system` should be used only to obtain information about the state of the world. If you wish to change the state of the world, this should be done by placing a `System` message in your output list (see next manual section).

Since reading data from the terminal would constitute a side effect, the background process created by `system` comes into being with its standard input closed.

Implementation Restriction

`read`, `filemode`, `getenv`, and `system` all require their argument to be at most 1024 characters long.

Note on system reading functions and referential transparency

Although `read`, `filemode`, `getenv` do not have side effects, they are not referentially transparent because it cannot be guaranteed that an expression like

```
read "file"
```

will return the same result if evaluated twice. Some external event may have changed the state of the filing system in the meantime. Clearly the same problem applies to `system` - consider for example the expression

```
system "date"
```

which gets date-and-time as a string. Evaluating this twice in succession is unlikely to produce the same result.

Strictly speaking all calls to `read` and the other functions in this section ought to be evaluated with respect to the state-of-the-world as it existed before the evaluation of the given Miranda expression commenced. Otherwise referentially transparent behaviour cannot be guaranteed. Enforcing this would appear to require, among other things, taking a copy of the whole filing system before each Miranda command-level evaluation. For obvious reasons this is not implemented.

.....

31/2 Output to UNIX files etc.

.....

Since Miranda is a functional language, the evaluation of an expression cannot in itself cause a side effect on the state of the world. The side effects occur when the value of the expression is printed. The value of a command level expression is a list of `system messages', where the possible forms of message are shown by the following type declaration,

```
sys_message ::= Stdout [char] | Stderr [char] | Tofile [char] [char] |
              Closefile [char] | Appendfile [char] | System [char] |
              Exit num
```

The system `prints' such a list of messages by reading it in order from left to right, evaluating and obeying each message in turn as it is encountered. The effect of the various messages is as follows.

Stdout string

The list of characters `string' is transmitted to the standard output, which will normally be connected to the user's screen. So for example the effect of obeying

```
[Stdout "!!!"]
```

is that three exclamation marks appear on the screen.

Stderr string

The list of characters `string' is sent to the standard error output. [Explanation to those unfamiliar with UNIX stream philosophy: all normal UNIX processes come into existence with a standard input stream, and two output streams, called standard out and standard error respectively. Under normal circumstances standard error and standard out are both connected to the users screen, but in principle they could be connected to different places.]

Tofile fil string

The characters of the string are transmitted to the file or device whose UNIX pathname is given by `fil'. Successive `Tofile' messages to the same destination are appended together (i.e. the first such message causes the file to be opened for writing, and it remains open until the end of the whole message list). Note that opening a file for output destroys its previous contents (unless preceded by an `Appendfile' message, see below).

Closefile fil

The stream which has been opened to the file `fil' (presumably the subject of some previous `Tofile' messages) is closed. If `fil' was not in fact open this command has no effect (i.e. is harmless). All open-for-output streams are automatically closed at the end of a message-list evaluation, so it is only necessary to invoke `Closefile' explicitly if you wish to terminate output to given file during a message-list evaluation. (One reason why you might want to do this is so as not to have too many output files open at one time, since many UNIX systems place a limit on the number of streams which a process can have.)

Appendfile fil

If obeyed before any `Tofile' messages to destination `fil', causes the file to be opened in `append-mode', so its previous contents are added to, instead of being replaced.

System string

Causes `string' to be executed as a shell command (by `/bin/sh') at this point in time. Enables arbitrary UNIX commands to be invoked from within a Miranda output list. The shell process comes into being with its streams (standard input, standard output, standard error) inherited from the Miranda process.

Exit num

Causes the UNIX process evaluating the message list to terminate at this point with exit status `num' (an integer between 0 and 127). The remaining messages in the list (if any) are discarded. The exit status of a Miranda evaluation which terminates other than by a call to Exit will be 0 if it terminates successfully or 1 if it encounters a runtime error. The exit status is only relevant if you are using Miranda to implement a stand-alone UNIX command (see separate manual page about this).

[Explanation: the exit status of a UNIX command is a one byte quantity which is communicated back to the calling shell and can be tested by it - the usual convention is that 0 exit status means all ok, anything else means something was amiss. If you are not into shell programming you can safely ignore the whole issue.]

The default message wrapper

We have stated above that the value of a command level expression is expected to be of type `[sys_message]'. Otherwise it is converted to a list of messages by applying the default message wrapper,

```
wrap x = [Stdout x]
```

This requires that x be a string - if not, the operator `show' is applied to x to convert it into a printable representation.

This explains how the Miranda system is able to function in its standard `desk-calculator' mode. When you type, say `2+3', the compiler notices that this is not of type [sys_message] and silently converts it to

```
[Stdout (show(2+3))]
```

before obeying it.

Output redirection

A Miranda command of the form

```
exp &> pathname
```

causes a background process to be set up for the evaluation of `exp', with both the standard output and the standard error output of the process redirected to `pathname'. The expression is coerced to be of type [sys_message] using exactly the same rules as in the standard case. If the value of `exp' is an explicit list of messages, the destination of `Tofile' messages are unaffected by the global redirection (only messages which would otherwise have gone to the screen are sent to `pathname').

If two (blank separated) pathnames are given after the `&>', standard output is redirected to the first file and standard error to the second. Thus:

```
exp &> outfil errfil
```

If the '&>' is replaced by a '&>>', instead of overwriting the previous contents, the relevant output is appended to the end of the file. Thus:

```
exp &>> pathname(s)
```

As with the '&>' command, either one or two pathnames can be given, depending on whether you wish standard error to be merged with standard out, or separated from it.

Note that a background process created by a '&>' or '&>>' command has no standard input - if the expression contains '\$-', the latter will evaluate to '['.

Implementation Restrictions

Arguments representing pathnames (to Tofile, Appendfile, Closefile) are restricted to 1024 characters in length - pathnames longer than this cause an error message. The shell command supplied to System is also restricted to 1024 characters in length.

.....

31/3 Reading with interpretation ('readvals' and '\$+')

.....

There is a function readvals which takes a string representing a UNIX pathname, and returns a list of values found in the file of that name. The values may be represented by arbitrary Miranda expressions, written one per line. Blank lines, and Miranda style comments (| |...) are ignored. If the input file appears to be a teletype, readvals reacts to syntactically incorrect or wrongly typed data by prompting the user to repeat the line, and such bad values are omitted from the result list. If the input file does not appear to be a teletype, bad data causes readvals to abort with an error message.

Note that, similarly to show

- (i) readvals is a reserved word, not an identifier.
- (ii) the context in which it is used must be such as to determine its type monomorphically. Extra type specifications may be needed in the script to meet this condition.

Here is a simple example of how r_e_a_d_v_a_l_s might be used in a script

```
x :: [num]
x = readvals "data"
```

The file 'data' should contain expressions of type num (one per line).

The readvals function provides Miranda with a simple form of data

persistence - data can be written to a file (e.g. using 'show') and read back using readvals in a later session. Data objects saved in this way must of course be finite. Notice also that if you wish to save data containing functions, you will have to set up some special arrangement, since such data cannot be written out using 'show'.

Data of abstract type can be written to file using show and read back with readvals - provided that an appropriate show-function was included

in the signature of the abstract type (see manual section on abstract types).

Finally note that \$+ behaves exactly like an application of readvals to the name of the file to which the standard input is connected. For example

```
sum $+
```

read a sequence of numeric expressions from the keyboard (one per line) up to the next control-D, and then returns their sum.

```
.....
```

31/4 Using Miranda to build executable files

```
.....
```

FIRST METHOD (using a 'here document')

Create a file ('prog' say) containing the following

```
mira [script] <<!  
  exp1  
  exp2  
  .  
  .  
  .
```

Remember to make the file executable (by saying 'chmod +x prog'). Now when 'prog' is executed as a UNIX command, the result will be that the miranda expressions 'exp1', 'exp2' ... will be evaluated in the environment defined by 'script', and the results sent to the standard output. As usual, if 'script' is omitted, a default name 'script.m' is assumed. The text following the special redirection '<<!' is called a "here-document". The contents of the here-document are fed to the mira command in place of its standard input. (So anything you would type to the miranda system at top level can appear in the here document.)

Here-documents are a normal feature of UNIX, not something special to miranda. Miranda's only contribution to making this work smoothly is that it detects when its input is not coming from a terminal, and in this case suppresses prompts and other extraneous feedback. Note also that lines beginning '| |' are ignored by the Miranda command interpreter, which gives a way to include comments in the text of the here-document.

The program 'prog' might be invoked with one or more arguments, for example,

```
prog big 23
```

In the here-document, '\$1' can be used to denote the first argument, '\$2' the second and so on - in this case 'big' and '23' respectively will be textually substituted for these before the here-document is fed to mira as input. Arguments not present are replaced by empty text. Other replacements carried out on the text of the here-document are - '\$*' is replaced by all the arguments together, as a single piece of text, '\$#' is replaced by the number of arguments present ('2' in the case shown above), and '\$0' is replaced by the name of the program being executed (in this case 'prog'). If the here-document contains instances of '\$' which you don't want replaced by something (for example inside a

Miranda string), you have to escape them by preceding them with a backslash character.

The drawbacks of commands built in this way are two - (a) they have no way of taking information from the terminal during execution (because the here-document replaces the standard input) and (b) the method of access to command line arguments is clumsy. These problems are overcome by:

SECOND METHOD (using a 'magic string')

Create a file ('prog' say) containing a normal miranda script, but with the first two lines in the following form

```
#!/usr/local/mira -exp
<expression>
```

The remainder of the file can be any legal miranda script, including that it can be empty. Remember to give the file execute permission (by saying 'chmod +x prog'). Now when 'prog' is executed as a UNIX command, the result will be that the miranda expression '<expression>' is evaluated in the environment defined by the rest of the script, and the results sent to the standard output. Note that only a single expression is permitted, and it must occur entirely on the second line of the file. The form of the first line is rigid - the three initial characters must be '#', '!' and space, followed by the absolute pathname of the miranda interpreter. The flag "-exp" is necessary and no other flags are allowed here.

[Warning: if you omit the '-exp' flag you can get some very strange errors when you try to execute the file, e.g. a spurious complaint that the 'mira' program has been installed with the wrong permissions.]

A command of this form can take data from the terminal. The notation '\$-' can be used anywhere in the second and later lines of the file to denote the list of characters taken from the standard input. (That is '\$-' behaves like a Miranda identifier of type [char].)

The command can be invoked with arguments, eg

```
prog pig 23
```

and the notation '\$*' can be used in the script as a Miranda identifier of type [[char]] denoting the argument list, with the convention that the initial (zero'th) argument is the name of the command. So in this case the value of '\$*' would be

```
["prog","pig","23"]
```

Of course, if there are no arguments, '\$*' will be a singleton list containing just the command name.

Explanation

The line beginning '#!' is a standard UNIX incantation, called a 'magic string', indicating that the following pathname is to be used as a command interpreter with the name of the file in which it occurs as its argument (see under 'execve' in section 2 of the UNIX manual). The flag "-exp" advises the Miranda system to expect a special script, of the above form, containing an expression to be evaluated as its second line.

[Notes to system administrators:

(1) In some versions of UNIX 'execve' places a limit (typically 32 chars) on the total length of the 'magic string'.

(2) Because the UNIX `execve' program permits at most one flag in a magic string, it is not possible to give a `-lib' flag to mira in conjunction with a `-exp' flag. This is a possible source of difficulty if you keep the miralib directory at a non-standard place. One way round this is to set the environment variable MIRALIB, instead of using a `-lib' flag. See manual section on flags etc.]

Example

The following script is a Miranda version of the UNIX `cat' command - if it is invoked without arguments it simply copies its standard input to its standard output, otherwise it concatenates the contents of its argument files to the standard output.

```
#!/usr/local/mira -exp
output

output = [Stdout $-], _i_f t1 $* = []
        = [Stdout (concat(map read(tl $*)))], _i_f badargs=[]
        = [Stderr (concat(map errmsg badargs))], otherwise
badargs = [f|f<-tl $*;~member(filemode f)'r']
errmsg f = f++": cannot access\n"
```

See also the manual pages on input from UNIX files, output to UNIX files, for the explanation of `read', `filemode', and the constructors Stdout, Stderr etc.

The rule that Miranda source files must have names ending in ".m" does not apply to scripts in this special form (in keeping with the UNIX convention that the names of executable programs require no special extension).

Embarrassing Pause

A drawback of the way Miranda #! scripts are currently implemented is that such a script is recompiled each time it is used - if the script is large this can lead to an irritating delay each time the program is executed. This can be minimised by placing as many as possible of the definitions needed by the #! file in a separate script and %including them (see section on library directives). These definitions will then not be recompiled at each invocation, but retrieved from the corresponding object code file.

.....

31/5 How to change the default editor.

.....

The Miranda /e command (see manual page on command interpreter) invokes an editor. By default this is the standard UNIX screen editor "vi". If you don't like "vi", or would prefer to use another editor, this is easily arranged.

The Miranda command
/editor

reports the editor currently in use by the Miranda system. To change it, just say

```
/editor foo
```

where "foo" is the name of the editor you wish to use instead. Alternatively, when next invoking the miranda system from UNIX you can supply it with a flag requesting a specific editor by name, as follows:

```
mira -editor foo ...
```

In either case it is only necessary to do this once, since the Miranda system stores such information permanently. (It does this in a file called ".mirarc" in your home directory - you should not remove or tamper with this file).

More advanced information

The interface between the Miranda system and the editor is improved if there is a way of telling the editor to open a file with the cursor positioned at a specified line number. For example to make the editor `vi' open a file at line 13, the UNIX command is

```
vi +13 file
```

the Miranda system has built in knowledge of this, so if the installed editor is `vi' and the compiler has found a syntax error in the script, the Miranda `/e' command will open the script at the line containing the error.

To retain this ability when substituting another editor for `vi', you must supply the `/editor' command with the template of a UNIX command for invoking your editor at a given line number. In this template the line number is represented by the character `!' and the filename by the character `%'. For example the full template for `vi' would be supplied to Miranda in the following way

```
/editor vi +! %
```

If the `%' character does not occur in your template, Miranda will assume that the name of the file is to be added to the end of the command, as its final argument. So the template for `vi' could equally well be given as

```
/editor vi +!
```

As another example, the appropriate command for installing micro-emacs as the resident editor is

```
/editor ue -g!
```

If you install an editor without the capability to be opened at a specified line number (i.e. you cannot give a template for it containing the `!' character), the /e command loses its ability to "go to the right place" after an error, and the `??' command will be disabled (`??identifier' asks the Miranda system to open the relevant source file at the definition of the given identifier.)

The Miranda system will work perfectly well without either of these features, but there is a significant loss of power in the user interface. If you intend to use the Miranda system a lot, you should either contemplate learning an editor with an open-at-line-number feature (such as vi, or micro-emacs), or else lobbying the person who maintains your usual editor, if you have access to him, to enhance its interface to UNIX by adding an 'open at line number' flag to its command line.

If your installed editor lacks the 'open at line number' feature, you may find it convenient to have the script listed to the screen during compilation (this feature of the compiler can be switched on and off by the commands '/list', '/nolist'). As an assistance to naive users, the Miranda system turns on '/list' for you if the '/editor' command is used to install an editor without the 'open at line number' feature.

[Pathological case:

"What do I do if the name of my editor contains a '!' or a '%' character?" You escape it with a backslash, so if the name of your editor is 'oh!calcutta!', to install it in Miranda you must say
/editor oh!\!calcutta\!

end of pathological case]

.....

31/6 How to alter the sizes of workspaces

.....

The Miranda system uses two main internal workspaces called "heap" and "dic". If either overflows during a session, a self explanatory error message is given. The sizes of both areas may be changed by the user if required. Any change made is remembered thereafter and for subsequent sessions until countermanded.

The heap contains almost all the data structures created both by the Miranda compiler and the evaluation system. To compile and/or run very large scripts you may need a bigger heap. To find out (change) the current size of the heap say

/heap (or /heap newsize e.g. /heap 50000)

The heap size may also be altered by using a flag (see below). You should be aware that running Miranda processes with a very large heap may give you slower responses, and in a multi-user virtual memory environment is likely to be regarded as antisocial.

The dictionary is used to store identifiers and file names by the Miranda compiler. It is unlikely that you will need to change the size of the dictionary. The current size of the dictionary can be ascertained by the command

/dic

it cannot be changed dynamically, from within the Miranda system. To alter the dictionary size use a flag (see next para).

The sizes of either or both areas may be set by flags when invoking the miranda system. The following shows both possibilities

mira -dic 50000 -heap 100000 [script]

Note that the size of the heap is given in 'cells' (a cell is 9 bytes, currently) and the size of the dictionary is in bytes.

The most recent settings of the workspace sizes are stored in the file ".mirarc" in the users home directory, and automatically carried over to

the next miranda session.

::::::::::::::::::::::::::

31/7 Flags, setup files etc.

::::::::::::::::::::::::::

(This section may be of particular interest to installers and system administrators)

The full form of the `mira' command is

```
    mira [flags...] [script]
```

this command causes a Miranda session to be entered with the given file as current script. If no script is specified a default filename `script.m' is assumed. The specified file need not yet exist - in this case you will be starting a Miranda session with an empty current script.

Note that `.m' is the standard extension for Miranda language source files - the mira command always adds the `.m' extension, if missing, to any filename argument denoting a Miranda script.

The available flags are:

```
    -lib pathname
```

Tells mira to look for miralib (the directory containing libraries, manual pages etc.) at `pathname' instead of in the usual places, `/usr/lib/miralib' or `/usr/local/lib/miralib'. Can also change this by setting an environment variable, see below.

```
    -gc
```

Switches on a flag causing the garbage collector to print information each time a garbage collection takes place. This flag can also be switched on and off from within the miranda session by the commands `/gc', `/nogc'.

```
    -count
```

Switches on a flag causing statistics to be printed after each expression evaluation. This flag can also be switched on and off from within the miranda session by the commands `/count', `/nocount'.

```
    -list
```

```
    -nolist
```

Switches on (off) a flag causing Miranda scripts to be listed to the screen during compilation. This flag can also be switched on and off from within the miranda session by the commands `/list', `/nolist'.

```
    -strictif
```

```
    -nostrictif
```

Switches on (off) a flag causing the Miranda compiler to enforce the presence of the keyword `if' in guard syntax. By default the presence of the `if' is optional, for upwards compatibility with earlier versions. This switch is also available from within a miranda session by the commands `/strictif', `/nostrictif'.

```
    -hush
```

-nohush

The miranda system decides whether or not to give prompts and other feedback by testing its standard input with `isatty'. If the standard input does not appear to be a terminal it assumes that prompts would be inappropriate, otherwise it gives them. In either case this behaviour can be overridden by an explicit flag ("-hush" for silence, "-nohush" for prompts etc). This switch is also available from within a miranda session by the commands `/hush', `/nohush'.

-dic SIZE

Causes the dictionary (used by the compiler to store identifiers etc.) to be set up with SIZE bytes instead of the default 24kb.

-heap SIZE

Causes the heap to be set up with SIZE cells instead of the default (currently 20k). This can also be done from within the miranda session by the command `/heap SIZE'. A cell currently occupies 9 bytes.

-editor name

Causes the resident editor (initially `vi', unless the environment variable EDITOR was set to something else) to be `name' instead. This can also be done from within the miranda session by the command `/editor name'.

SPECIAL CALLS

In addition the following special calls to `mira' are available which do not enter a Miranda session but accomplish another purpose, as described below.

mira -man

To enter the miranda online manual system directly from the UNIX shell.

mira -exp

Special call permitting the use of miranda script as a stand-alone UNIX command. See separate manual page for details.

mira -make [sourcefiles]

Forces a check that all the miranda source files listed have up-to-date object code files, triggering compilation processes if necessary (see manual section on separate compilation).

mira -exports [sourcefiles]

Sends to stdout a list of the identifiers exported (see manual section on library directives) from each of the given miranda source files, together with their types (may also force recompilation if needed).

mira -sources [sourcefiles]

Sends to stdout a list of all the Miranda source files on which the given source files directly or indirectly depend (via %include or %insert statements), excluding the standard environment <stdenv>.

mira -version

Prints version information. This information can also be obtained from within a Miranda session by the command `/version'.

SETUP FILES

The current settings of `dic`, `heap` and `editor` are saved in the file ``.mirarc'` in the users home directory, and are thereby carried over to the next miranda session. The settings of the compiler flags which control whether or not source is listed to the screen during compilation (toggled by the commands `/list/nolist` during a Miranda session), and whether or not the keyword ``if'` in guards is enforced (toggled by the commands `/strictif /nostrictif` during a Miranda session) are also saved in the users ``.mirarc'` file.

The default settings of these entities, which will be picked up by new users, executing ``mira'` for the first time, are `dic 24000` (bytes), `heap 20000` (9-byte cells), `editor _v_i`, `nolist`, `nostrictif`. The current settings can be interrogated, during a Miranda session by the command ``/settings'`.

The defaults can be changed, on a system wide basis, by moving a copy of a ``.mirarc'` file containing the desired settings into the ``miralib'` directory (normally found at `/usr/lib/miralib`). The user's local `.mirarc` file, once created in his home directory by the first call to `mira`, will override the global one, however.

The behaviour of the ``mira'` program is also sensitive to the settings of certain environment variables - see separate manual entry about this.

OUTPUT BUFFERING

Output from the Miranda system to the user's terminal should not be line buffered, or some things will not work as they are intended. There is no problem about input being line buffered, however.

::::::::::::::::::::::::::

31/8 Environment variables used by Miranda

::::::::::::::::::::::::::

(This section may be of particular interest to installers and system administrators)

The behaviour of the ``mira'` program is sensitive to the settings of certain environment variables.

An alternative location for the `miralib` directory may be specified by setting the environment variable `"MIRALIB"`. An explicit `-lib` flag, if present, overrides this.

The first time it is called (i.e. if no `.mirarc` file is present, either in the home directory or in `miralib`) the miranda system picks up the name of the resident editor from the environment variable `EDITOR` - if this is not set it assumes ``vi'`.

If the environment variable `RECHECKMIRA` is set (to any non-empty string) the Miranda system rechecks to see if any relevant source files have been updated, and performs any necessary recompilation, before every interaction with the user - this is the appropriate behaviour if an (eg emacs) editor window is being kept open permanently during the Miranda session. If this environment variable is not set, the check is

performed only after `e' commands and `!' escapes.

To decide what shell to use in `!' escapes, mira looks in the environment variable SHELL (this will normally contain the name of the user's login shell). If no SHELL is entered in the environment, /bin/sh is assumed.

For displaying pages of the manual mira uses the program entered in the environment as VIEWER - if this variable is not set it uses either `/usr/ucb/more -d' (Berkeley UNIX) or `/usr/bin/pg -e' (system 5).

The manual reading system is also sensitive to the settings of two other environment variables, MENUVIEWER and RETURNTO MENU. If you set VIEWER to something else, you may also need to set the environment variable RETURNTO MENU. More information about this may be obtained by typing ??? to the "next selection" prompt of the manual system.

The manual system clears the user's screen before displaying a contents page and before each invocation of VIEWER - it finds out how to do this by looking in the termcap (Berkeley UNIX) or terminfo (system 5) database. If there is no appropriate TERM CAP/TERMINFO entry in the environment a warning message is printed each time you enter the manual system.

How to set an environment variable in your UNIX shell:
(Reminder/tutorial information)

Example, setting the environment variable VIEWER to /usr/local/view

- (i) if you use the Bourne shell (usual on system 5)
say at the UNIX command level (i.e. before calling Miranda)

```
VIEWER=/usr/local/view; export VIEWER
```

to make it permanent add this line to your .profile

- (ii) if you use the C shell (usual on Berkeley UNIX)
say at the UNIX command level (i.e. before calling Miranda)

```
setenv VIEWER /usr/local/view
```

to undo the above say `unsetenv VIEWER',
to make it permanent add the setenv line to your .login

::::::::::::::::::::::::::

32 CHANGES

::::::::::::::::::::::::::

This section catalogues the main changes to the Miranda system between release one (of April 1987) and this release, which is release two (of August 1989). All relevant sections of the manual have been revised to take account of these changes.

1. Non-upwards compatible changes <-- PLEASE READ!!
2. New language features

3. New system features
4. How to rewrite scripts containing laws and strictness annotations
5. Release history

.....

32/1 Non upwards compatible changes (between release one and release two)

.....

Release two has a number of significant new features - see next two sections - but most release one scripts should recompile and run with none or only trivial changes needed. The small number of non-upwards compatible changes are listed here.

1) Algebraic data types with laws (equations for constructors in `=>') and with strictness annotations (`!' written after some of the fields) are no longer officially a part of Miranda. Language simplification. For a transitional period (ie at least until release three) the compiler will still compile these correctly, but an `obsolete feature' warning is now given. A separate manual section sketches you how you can rewrite your scripts to remove these features, if you wish to do so.

2) For compatibility with Bird and Wadler(*) an (optional) keyword `if' has been introduced after the guard comma. Thus

```
abs x = x, if x>=0
      = -x, otherwise
```

Since the `if' is optional, the change is upwards compatible - except in one respect, NAMELY THAT `if' HAS BECOME A RESERVED WORD (sorry!). If you were using `if' as the name of a conditional function in your scripts, we suggest you change its name to `cond', or `test'.

3) Changes to the standard environment.

i) The definition of `foldl' has been changed to be the same as that in Bird and Wadler, removing a serious cause of confusion. Uses of the old foldl will be picked up by the compiler as a type error. The change needed is trivial - the operator supplied to foldl now takes its two arguments in the opposite order. The quickest way to fix this is to replace each occurrence of `foldl op ...' by `foldl (converse op) ...'

ii) The following new names have been added to the standard environment.

```
converse foldl1 foldr1 fst hugenum log10 map2 max2 merge
min2
scan showfloat showscaled snd system tinynum zip
```

See standard environment for documentation. If you are have defined one of these identifiers at top-level in a script the compiler will report a nameclash. There is also a new reserved word, `readvals' (see `New language features').

4) A missing `, otherwise' now generates a syntax error, rather than a warning.

5) Miranda now stores integers in a different internal representation from floating point numbers. Both kinds of number are both still of type `num' for typechecking purposes, and there is an automatic runtime coercion from integer to floating point when required. This makes the

change upwards compatible in almost all cases. However, `div`, `mod` and `!` (subscripting) now give a runtime error if applied to floating point arguments: `fractional number where integer expected`.

If you get unexpected errors of this kind, the most likely cause is writing `/` (fractional division) where you should have `div` (integer division). Example: `8/4` is 2.0, whereas `8 div 4` is 2. Since there is now a distinction between 2.0 and 2, you can no longer use `/` where `div` was intended.

6) The default number printing function, `shownum`, now prints fractional numbers to 12 decimal places, but with trailing zeroes suppressed, instead of a fixed 6 places. (Since Miranda uses double precision real arithmetic, it seems misleading to show only six places. You can always say `showfloat 6` if you want the old behaviour.)

Numerous minor bugs have also been fixed, including all the bugs reported by user sites. These are not listed here, since we assume that failure to preserve bugs will not be regarded as a violation of upwards compatibility (!)

Note that a quick way to recompile all the Miranda scripts in the current directory is to say

```
mira -make *.m
```

as a UNIX command. Recompiling your scripts will be sufficient to pick up all instances of problems 1) to 4) above, as syntax or type errors. Problem 5), if present, will show up as a runtime error.

(* Richard Bird and Philip Wadler ``An Introduction to Functional Programming'' Prentice Hall International, March 1988.

.....

32/2 New Language Features

.....

1) The %free directive, supporting parametrised scripts, has been implemented. See new manual page in the manual section on the library mechanism.

2) Local definitions can now be polymorphic, and used at more than one type in the body of the where clause. That is, there is now no difference between local definitions and top-level ones, as regards their right to use polymorphism. This change is completely upwards compatible, but removes a long standing anomaly in the type system, which several people have complained about. As a side effect of the changes to the compiler made to bring this about, the compilation of large where clauses is now more efficient (i.e. takes less time, and produces better code, than before).

3) Integers are now held in a distinct internal representation from floating point numbers - integers can be of unbounded size and are not subject to rounding error. The internal representation of floating point numbers is double precision (circa 17 decimal digits of accuracy).

A numeric literal is integer if it contains neither a decimal point nor a scale factor, otherwise it is floating point. Both kinds of number are still of a single type, ``num'` as far as the typechecker is concerned, and there is an automatic run-time conversion from integer to floating point when required.

The change is upwards compatible except in the following two respects. (i) It is now a (run-time) error to use a floating point number for subscripting a list (`x!n`), or as an argument to the integer division operations ``div'`, ``mod'`. (ii) The fractional division operation ``/'` now always produces a floating point result.

In connection with the improvements to Miranda's arithmetic the following have been added to the standard environment

`hugenum tinynum log10 showfloat showscaled`

See comments in standard environment for explanation.

4) The definition of irrefutable pattern has been modified, to treat user-defined single constructor types homogenously with tuple types. (So any type with only one constructor is now a product type.) See paragraphs on irrefutable patterns at end of section 16 in the manual for a description of Miranda's amended policy on product types.

5) There is an additional convention for literate scripts - that if the name of the file containing a Miranda script ends in ``.lit.m'`, it must contain a literate script. This makes it possible to start a literate script in ``narrative mode'` - so it could be eg a LaTeX source file.

6) The `%insert` directive now works in literate scripts as well as in scripts using the standard comment convention. The `%inserted` text need not use the same comment convention as the containing script.

7) The additional notation ``-identifier'` is now allowed in an `%export` list. It has the effect of subtracting the identifier from the list of things exported (see manual section on library directives).

8) The interface to UNIX has been made more powerful by the addition of two new functions - ``system'` and ``readvals'`. These are documented in the manual section on UNIX/Miranda system interface.

The function ``system'` takes a string and returns the result of executing it as a UNIX shell command. See manual section on Input from UNIX files etc for details.

The function ``readvals'` which takes a string denoting a UNIX file, and reads from the file a list of Miranda values. Note that for technical reasons ``readvals'` is a reserved word, not an identifier. Special case - the notation ``$+'` reads values from the standard input. See manual page on ``readvals'` and ``$+'` for details.

9) The notation ``$-` for the standard input is now permitted in scripts as well as in command level expressions. (Likewise ``$+`).

.....

.....

The compiler's treatment of undefined names has been modified - they are now reported with line numbers, as encountered during the typecheck, instead of merely being listed at the end of compilation. This is in response to the observation that many undefined names are actually spelling mistakes. In order to avoid an excessive number of warnings, if there are multiple occurrences of the same undefined name, only the first is reported.

The compiler now reports definitions included under a `where' and not used, directly or indirectly, in the body of the where, as redundant (this is a warning, not an error).

Two new commands are allowed in a Miranda session

```
/strictif
/strictif
/nostrictif
```

To make the compiler treat `if' after the guard comma as compulsory (/strictif), or optional.

The following additional flags are now accepted by the `mira' program

```
-list
-nolist
-sources
-strictif
-nostrictif
-version
```

See under `Flags etc' in UNIX/Miranda system interface.

`mira -exports' now accepts multiple file names. Example, the shell command `mira -exports *.m' gives the export profiles of all the Miranda scripts in the current directory.

The `!' escape in mira (and in the manual system) now shows more intelligence about which shell to use. If your login shell is `csh' (normal for users of Berkeley UNIX) it uses that rather than `sh'.

All Miranda commands involving filenames now use C-shell conventions in interpreting filenames beginning with `~'. This also works for library directives (%insert, %include).

[Note: the `~' convention does not work in Miranda systems running under system 5.]

The convention (in library directives) that filenames surrounded by angle brackets are relative to the miralib directory now also works in Miranda commands. Specifically `/e', `/f' and `/cd' now understand this convention. Example the command

```
/f <ex/primes>
```

in a Miranda session selects for the current script the file `primes.m' from the directory of examples `ex', stored under the miralib directory.

The exit status of a Miranda process (other than one which terminates with an explicit call to Exit) is now 1 if the process terminates with a runtime error, 0 otherwise. This is a change of policy, to make it more convenient to use programs written in Miranda in shell scripts and

makefiles. (Formerly exit status was always 0, unless Exit called.)

A bug in the lex analyser causing it to reject with an OFFSIDE error a Miranda script whose last line ends in EOF rather than a newline character, has been fixed. This was a nuisance for people whose installed editor is emacs, which does not automatically add a newline to the last line of a file.

One other change has been made for the benefit of people using emacs. If the environment variable RECHECKMIRA is set, the mira program checks to see if any relevant source file has been updated before every interaction with the user, instead of only after /edit commands. This is useful if an editor window is being kept open permanently during a Miranda session.

There is a new manual section on environment variables (under UNIX/Miranda system interface).

.....

32/4 How to rewrite scripts containing laws and strictness annotations

.....

It has been decided to remove the following features from the definition of the Miranda language

algebraic types with laws (=>)

algebraic types with field annotations for strictness (!)

These features have been found to be of rather marginal value, and they introduce significant and undesirable complications into the semantics.

For the time being they are still supported by the compiler, so no-one loses a working script, but they are no longer officially part of Miranda and their use now attracts an 'obsolete feature' warning at each compilation. At some stage in the future (probably at the next release) they will cease to be supported by the compiler.

Laws

Here is a method for translating Miranda code using laws into equivalent code not using this feature. We take an example from the Nancy paper(*), of self-ordering lists, for illustration

```
-----  
|  
|  olist ::= Onil | Ocons num olist  
|  
|  Ocons a (Ocons b x) => Ocons b (Ocons a x), if a>b  
|  
-----
```

For each constructor which has laws associated with it (in this case Ocons) we introduce a new function name ('ocons' say). Now we make two changes to the script

- 1) Throughout the script (including the rhs of the laws) replace all

right-hand-side occurrences of the lawful constructors by the associated function names. Only the 'left-hand-side' uses of the constructor, i.e. in pattern matching, are left alone.

2) Turn each law into a function definition, by replacing the outermost occurrence of the constructor on the lhs of the law by the associated function name, and replacing each '=' by '='. We must also add a last case to the function definition, stating that it is equal to a call of its associated constructor on the same arguments if no earlier case applies.

The definition of olist now looks like this

```
-----  
|  
| olist ::= Onil | Ocons num olist  
|  
| ocons a (Ocons b x) = ocons b (ocons a x), if a>b  
| ocons a x = Ocons a x  
|  
|-----
```

and throughout the rest of the code any occurrences of 'Ocons' other than in pattern matching is likewise replaced by the function 'ocons'. This translation must preserve the behaviour of the script (because it corresponds exactly to the way in which laws are implemented).

[A comment on this translation:-

Notice that the fact that objects of type 'olist' are ordered now depends on a voluntary discipline by the programmer - that he always builds his olists by calling the function ocons, and never by using the raw constructor Ocons. So the original script using laws had a security about it that the translation no longer expresses - if we come back later and add new code, we might 'forget' the discipline and accidentally build a non-ordered olist, by calling Ocons directly.

The proper solution to this is to make olist into an abstract data type (although this will involve a more radical rewrite of your script). In fact the main reason for dropping lawful types from Miranda is the observation that the abstype declaration is a cleaner and more general mechanism for introducing unfree types.

For example here is a possible definition of ordered lists as an abstract type

```
abstype olist  
with onil :: olist  
    ocons::num->olist->olist  
    ohd::olist->num  
    ot1::olist->olist  
    oempty::olist->bool
```

see below

```
olist == [num] ||constraint: the list is kept ordered,
```

```
ocons a (ocons b x) = b:ocons a x, a>b  
ocons a x = a:x, otherwise  
ohd = hd
```

```
otl = tl
oempty = ([])
```

In the rest of the script we can manipulate olists ONLY by calling the functions declared in the signature of the abstype (that is, the information following the `with'). It is therefore logically impossible to create a non-ordered olist. One thing has been lost - in the rest of the script we can no longer do pattern matching on olists - but that is a small price to pay for security.

end of comment]

Strictness annotations

The semantic effect of a strictness annotation (!) on a field in an algebraic type definition is to make certain expressions evaluate to BOTTOM (error or non-termination) that would otherwise have had a value. It therefore follows that removing all the strictness annotations must be semantically harmless - it cannot introduce an error into a working program.

A secondary purpose of strictness annotations was to gain some control over space behaviour by forcing some constructors to be call-by-value in some of their arguments. However, if you are interested in modifying the space behaviour of your programs by changing the order of evaluation, you can always use `force' and `seq' to do this explicitly.

For example, if you wish data structure x to be fully evaluated before being passed to function f, you can write

```
seq (force x) (f x)
```

(*) D. A. Turner: ``Miranda: A Non-Strict Functional Language with Polymorphic Types'', Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Springer Lecture Notes in Computer Science, vol. 201, pp 1-16).

.....

32/5 History of Miranda releases

.....

November 1985 - pre-release version (version 0.292 for VAX, 0.293 for SUN)

April 1987 - release one (version 1.009 or 1.016 or 1.019)

October 1989 - release two (versions 2.0xx for various machines)

Notes:-

A free upgrade to release one was offered to all sites running the Miranda prerelease. There should be no remaining copies of this in use.

The release number of a Miranda version is the integer part of the

version number - e.g. version 1.016 is a release one system. Small increments to the decimal part of a version number represent recompilations for different machines, and are of no significance - i.e. if you have version 2.006 and someone else has version 2.008 this does not mean that your version is out of date.

.....

33 Licensing Information

.....

The Miranda system is a licensed product of Research Software Limited, and may be used only on computers for which a currently valid license has been obtained. Two kinds of license are available:

Educational licenses, permitting the use of the software in teaching, and in not-for-profit research carried out by teaching institutions and funded from their own internal resources.

Commercial licenses, needed for all other purposes (including University research funded by external grants).

The Miranda system runs on a variety of computers under the UNIX operating system, including (at the time of writing) VAX, Microvax, DECstation 3100, SUN 3, SUN 4, Apollo, HP 9000 series, ORION 1/05, MIPS. Ports to some other UNIX systems are planned in the near future.

Further information about the Miranda system, including the computer systems for which it is available, and current license fees, may be obtained by mail, electronic mail, or telephone, from:

Research Software Limited
23, St Augustines Road,
Canterbury
Kent CT1 1XP
ENGLAND

telephone (24 hours) +44 227 471844
email: mira-request@ukc.ac.uk
uucp: mcvax!ukc!mira-request

(telephone callers from within the UK should dial `0' and not `44')

License holders of earlier versions of Miranda can obtain release two for a small upgrade fee - contact Research Software for details.

Future developments

The following are planned (no dates as yet)

- (i) A version of Miranda for the Macintosh.
- (ii) Native code versions of Miranda, giving much greater speed than the current interpretive system.

Existing license holders will be notified when either of these become available.

.....

34 Bug Reports

.....

If you find a bug, please report it to Research Software.

INTERNET: mira-bugs@ukc.ac.uk

USENET: mcvox!ukc!mira-bugs

First check the list below, however, in case it is already known. All bug reports will be gratefully received. Suggestions for improvements to the language or system, or their documentation, are also welcome. When sending a bug report, please remember to state the version number of your Miranda system and the type of machine that it is running on.

Note on error messages:-

The error messages from the Miranda system are mostly self explanatory. Note however that any message of the form "impossible event ..." indicates that a serious, and unexpected, internal error has occurred. Messages of this form should always be reported to Research Software as bugs. Any segmentation faults or core dumps should also be reported, as these too should not occur.

List of known bugs, deficiencies, and unimplemented features

There ought to be a directive which can be placed in a script to suppress or modify the automatic inclusion of the standard environment - this will be attended to in the next release.

It is not currently permitted to give a type specification for an identifier defined locally, as part of a where clause. That is (see formal syntax) `spec' is not allowed after where. There is no strong reason for this restriction and it will be lifted in later releases.

When abstract objects are tested for equality or order (under <, <= etc) the result is that obtained by applying the operation to the underlying representation type. In some cases this will be what you want, but in others it will be wrong - there ought to be a way of controlling this.

The standard input cannot be accessed both as a list of characters (with \$-) and as a list of values (with \$+) at the same time. If \$+ is in use, an occurrence of \$- will evaluate to [], and vice versa. This is not referentially transparent - uses of \$+ and \$- ought to share data.

Scripts with multiple occurrences of \$+ also behave opaquely - each occurrence of \$+ reads from the terminal independently, and they do not share data with each other. (Multiple occurrences of \$+ in a single command-level expression behave transparently, however.)

There is a subtle issue concerning `show' and %export. If you define a function which, internally, uses `show' on an object of algebraic type, and then %export that function to another scope, the format used by the `show' depends on the status of the algebraic type in the NEW scope. Thus if the type has been aliased the new constructor names will be used, and if the algebraic type is no longer in scope, it will show as "<unprintable>" (this latter case can arise if an abstract type based on

the algebraic type is exported, and one of the signature identifiers invokes `show' on the base type). Although this behaviour is defensible, it might be better for each use of `show' at algebraic type to be frozen to use the constructor names in the scope where it occurs. This will probably be fixed in a later release. [An analogous issue arises with `readvals'.]

Implementation restrictions not mentioned elsewhere in the manual:- A shell command called from mira using `!' is limited to 1024 characters in length after any implied expansions (eg of `%') have been performed. The same restriction applies to the result of expanding out a `/e' command. The pathnames of Miranda scripts are limited to 200 characters in length, including the `.m' extension. The name of the editor supplied for the `/e' command, and the absolute pathnames of the miralib directory, and of the user's home directory, are also each limited to 200 characters in length.

.....

35 Notice

.....

This manual contains a description of a commercial software product, the Miranda programming system, available under license from Research Software Limited. The information supplied in this manual is believed to be accurate but is given without warranty. Research Software Limited reserve the right to alter the specifications of this software without notice. The information contained in this manual is supplied for the use of license holders of the Miranda system and their students and staff, and may not be published without the prior written permission of Research Software Limited.