Ginger - A Simple Functional Language

Mike Joy

Department of Computer Science,
University of Warwick,
COVENTRY,
CV4 7AL,
UK.

Phone: +44 1203 523368

Fax: +44 1203 525714

Email: M.S.Joy@dcs.warwick.ac.uk

Last revision: 12 December 1992

ABSTRACT

Ginger is a lazy functional language with simple syntax and semantics, heavily sugared lambda-calculus spiced with primitive data types and operators. *Ginger* is designed to run on a parallel machine, and operators to control parallelism are included. Primitives for a novel "divide-and-conquer" style list processing model are also included. This document is the reference manual for the language.

Contents

1. Introduction

Many functional languages have been created, for a variety of purposes ranging from low-level codes such as FLIC [12] and GCODE [9] to user-friendly languages with substantial environments such as Miranda™ [14] and Haskell [8] . In order to facilitate work at Warwick and Birmingham Universities into parallel graph reduction machines it was necessary to choose a high level language with which to work.

Initially FLIC was used, but this language proved cumbersome, and was unsuited to teaching environments. None of the other mainstream languages was felt suitable either, principally because they were too big for the purpose. It was necessary to be able to introduce new constructs and data types into the language with ease, and these experimental features would fit awkwardly, if at all, with a known and already tightly-defined language.

We therefore decided to create a new language with the following features:

- Simplicity - syntax (and semantics) should be straightforward and easy to understand.
- Flexibility - it should be possible to add new data types and new operators swiftly.

As a result, *ginger* was created. Essentially heavily sugared lambda-calculus, the main constructs which constitute the *core* of the language are:

- Data types - integer, floating point number, character, list, boolean.
- Local definitions - both *expression where definitions* and *let definitions in expression* constructs.
- Functions - an inbuilt set of several dozen "useful" functions together with a library of standard functions which can be optionally included. All functions are by default Curried (that is, prefix notation).
- Infix operators - the principal arithmetic, boolean and list operators; each also has a prefix form.
- Lambda expressions - can be coded in directly, if desired.
- Parallel evaluation - constructs for explicity apportioning sections of program to remote processors are provided.
- I/O - input from files allowed (a file considered a list of characters), but output to the standard output stream only.
- UNIX® environment.

No attempt is made to provide the sophisticated facilities available in a large language such as Miranda or Haskell, and in particular no further development of I/O is envisaged. However, the language will develop with time, according to the requirements of its users. This document describes the *current* state of the language.

It is not the purpose of this document to give the reader a tutorial introduction to functional languages - we cite [2,4,10,11] for this purpose. Nor do we intend to discuss the λ-calculus or combinatory logic from first principles - texts such as [5,6] are widely available. We assume the reader can program in a high-level lazy functional language such as Miranda or Haskell, for which [7] and [3] respectively are good introductory texts, and is familiar with "Turner combinators" [13] (`S`, `K`, `I`, `B`, `C`, `S′`, `B′`, `C′`, `Y`).

Implementation of *ginger* has been performed in ISO C and some conventions at the lexical level follow those for C. Reference will therefore from time to time be made to the ANSI standard.

2. Data Types

The following data types are accepted by *ginger* as core types. For the formal specifications, see the BNF below.

2.1. Integer

This type (denoted by `Integer`) is implemented as `long int` in C, and the lexical syntax also follows that for C. For example, the following are valid integers:

```
42
-4535634
0
```

2.2. Real

This type (denoted by `Real`) is implemented as `double` in C, and the lexical syntax also follows that for C. For example, the following are valid reals:

```
0.0
2.3
-5.67
2.3E99
6.7e-45
2.
4.e+33
```

The following are *not* syntactically correct:

```
0.0.0
+2.3
.67
2.3E
6.7e-
```

2.3. Character

This type (denoted by `Char`) is implemented as `char` in C, and the lexical syntax also follows that for C. For example, the following are valid characters:

```
'X'
'\a'
'\xa2'
'\\'
'"'
```

2.4. Boolean

This this enumerated type (denoted by `Boolean`) has values denoted by the keywords `True` and `False`.

2.5. List

A list is an ordered sequence of zero or more objects. An empty list is denoted by `[]`, a finite list by (for example)

```
[a1,a2,a3,a4,a5]
```

and more generally by using the list constructor function `:` (see below). In addition, the following syntactical constructs are available:

`[a1..]`	*the infinite list of integers commencing* `a1`
`[a1..an]`	*the list of* n *integers from* `a1` *to* `an` *inclusive*
`[a1,a2..]`	*the infinite list of integers commencing* `a1` *with step of* `a2-a1`
`[a1,a2..an]`	*the list of integers from* `a1` *to* `an` *inclusive with step of* `a2-a1`

Table 1: "dotdot" notation for lists

2.6. Strings

The syntax of a string is the same as that for C, for example:

```
"hello"
"this string has a\ttab in it and a \abell"
"this string has a hex escape at the end\xa3"
```

However, a string is implemented as a list of characters.

2.7. Undefined

This type (denoted by `Undefined`) is *bottom*, and is returned by all functions with an undefined result. It is not usually of use to a programmer except to force an error condition explicitly (via the predefined functions `error` or `undef`).

3. Syntax

For a detailed description of the syntax, see the BNF in the appendix below. This section is a brief overview of the syntax.

A *ginger* program is a sequence of *definitions* and *expressions*, each terminated by a semicolon. The definitions specify the user-defined names, and are of the format

name argument1 argument2 ... argumentN = expression

A name can have zero or more formal arguments, which must themselves be names (pattern-matching is NOT allowed in *ginger*).

There should normally be *one* top-level expression in a *ginger* program, which is understood to mean the expression which it is expected to evaluate. It is expected that this restriction will eventually be relaxed when the user-interface is sufficiently developed!

An *expression* is one of the following:

- a name
- a datum (of type `Real`, `Integer`, `Char`, `Boolean`, or a list)
- (expression)
- expression expression
- \name expression
- expression infix-operator expression
- expression **`where`** definitions **`endwhere`**
- **`let`** definitions **`in`** expression **`endlet`**
- **`if`** expression **`then`** expression **`else`** expression **`endif`**

where definitions is a sequence of semicolon-separated definitions in the same format as above.

When using a λ-calculus expression in the context of *ginger*, a backslash (\) denotes the λ symbol, and the dot following the bound variable will be omitted. Thus the λ-expression

λx.X

is denoted by by the *ginger* expression

```
\x X
```

Also, note that the **where** construct binds more tightly than \e; thus

```
(\f f where f = 99 endwhere) 88;
```

evaluates to 99.

Example: Factorial

The standard factorial function is given to illustrate the syntax. This *ginger* program defines the name `factorial` and then evaluates `factorial 10`.

```
factorial n =
    if n <= 1 then 1
    else n * factorial (n - 1) endif;
factorial 10;
```

4. Inbuilt Identifiers

The following (Curried) functions are predefined and *cannot* be redefined by the user. Each unary operator with the exception of `id` is strict in its argument, and all *n*-ary operators ($n > 1$) are non-strict in all arguments unless explicitly stated otherwise. Any function will return `Undefined` when presented with an argument of incorrect type. Operators which are enclosed in parentheses also have an infix form (see below), namely the string with the parentheses removed.

In the following, in the style of SML and Miranda, the notation `::` means "has type", `[Integer]` is the type "List of Integer", `*` is any type. The type `Number` refers to *either* `Integer` *or* `Real` - functions requiring an argument of such type will convert from `Integer` to `Real` if required.

`(!)` `:: [*]->Integer->*`

takes a list and an integer *n* as argument, returning the *n*th element of that list (the head of the list is element 0). If *n* is negative or greater than the length of the list less 1, `Undefined` is returned. Strict in both arguments.

`#` `:: [*]->Integer`

takes a list and returns its length.

`(%)` `:: Integer->Integer->Integer`

is the standard *modulus* function, functionally equivalent to the C operator `%`. If the second argument is zero, `Undefined` is returned. Strict in both arguments.

`(&)` `:: Boolean->Boolean->Boolean`

is the logical conjunction operator. Strict in first argument, it is defined by:

```
x & y = if x then y else False endif;
```

`(*)` `:: Number->Number->Number`

is multiplication; if both arguments are integers, an `Integer` is returned, otherwise a `Real`. Strict in both arguments.

`(+)` `:: Number->Number->Number`

is addition; if both arguments are integers, an `Integer` is returned, otherwise a `Real`. Strict in both arguments.

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

is *append*; this is defined by

```
x ++ y = if isnil x then y
    else hd x : (tl x ++ y) endif;
```

Strict in its first argument only.

`(-)` `:: Number->Number->Number`

is subtraction; if both arguments are integers, an `Integer` is returned, otherwise a `Real`. Strict in both arguments. Note that the symbol "`-`" is parsed at the *lexical* level, and thus

```
f -3
```

refers to `f` applied to the integer `-3`, whereas

```
f - 3
```

means the same as `(-) f 3`. For unary minus, use `neg`.

`(/)` `:: Number->Number->Number`

is division; if both arguments are integers, an `Integer` is returned, otherwise a `Real`. If the second argument is zero, `Undefined` is returned. Strict in both arguments.

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

is list construction (`CONS` of LISP). Note that `(:)` is *not* strict in its second argument, thus

```
hd (1 : 2)
```

will return `1`, not `Undefined`.

`(<)` `:: *->*->Boolean`

returns `True` if its first argument is less than the second. Strict in both arguments, which must be either both `Number` or both `Char`.

`(<=)` `:: Number->Number->Boolean`

returns `True` if its first argument is less than or equal to the second. Strict in both arguments, which must be either both `Number` or both `Char`.

`(==)` `:: *->**->Boolean`

returns `True` if its first argument is equal to the second. The arguments can be of any type, including `List`. Strict in both arguments. If either argument is `Undefined` then `Undefined` is returned. If the arguments are of different types, `False` is returned. The two arguments *must* be of the same type:

```
0 == 0.0
```

will return `False`.

`(>)` `:: *->*->Boolean`

returns `True` if its first argument is greater than the second. Strict in both arguments, which must be either both `Number` or both `Char`.

`(>=)` `:: *->*->Boolean`

returns `True` if its first argument is greater than or equal to the second. Strict in both arguments, which must be either both `Number` or both `Char`.

`(^)` :: `Number->Number->Number`

is the *power* function; if both arguments are integers, an `Integer` is returned, otherwise a `Real`. If the first argument is negative, and the second one `Real`, then `Undefined` is returned. Strict in both arguments.

`(|)` :: `Boolean->Boolean->Boolean`

is logical (inclusive) *or*. Strict in first argument, it is defined by:

```
x | y = if x then True else y endif;
```

`~` :: `Boolean->Boolean`

is logical *not*.

`(~=)` :: `*->**->Boolean`

returns `True` if its first argument is not equal to the second. The arguments can be of any type, including `List`. If either argument is `Undefined` then `Undefined` is returned. If the two arguments are of different types, `True` is returned. Strict in both arguments.

`abs` :: `Number->Number`

returns the absolute value of its first argument, this is defined by

```
abs x = if x >= 0 then x else neg x endif;
```

`acos` :: `Number->Real`

returns the inverse cosine of its argument (in radians), `Undefined` if its argument is has absolute value greater than 1.

`and` :: `[Boolean]->Boolean`

takes a list of booleans and returns their logical conjunction. Defined by

```
and x =
    if isnil x then True
    else hd x & and (tl x)
    endif;
```

`asin` :: `Number->Real`

returns the inverse sine of its argument (in radians), `Undefined` if its argument is has absolute value greater than 1.

`atan` :: `Number->Real`

returns the inverse tangent of its argument (in radians).

`code` :: `Char->Integer`

returns the ASCII code denoting its character argument.

`compose` :: `(*->**)->(***->*)->***->**`

is equivalent to the combinator *B*, this is defined by

```
compose f g x = f (g x);
```

The infix form for compose is "`.`", thus

```
(f . g) x = f (g x)
```

`concat` :: `[[*]]->[*]`

takes a list of lists and concatenates them. Defined by

```
concat x =
    if isnil x then []
    else hd x ++ concat (tl x)
    endif;
```

`const` :: `*->**->*`

returns its first argument and discards its second; equivalent to the combinator *K*, this is defined by

```
const x y = x;
```

`converse` :: `(*->**->***)->**->*->***`

is equivalent to the combinator *C*, this is defined by

```
converse f x y = f y x;
```

`cos` :: `Number->Real`

returns the cosine of its argument (in radians).

`decode` :: `Integer->Char`

returns the character its argument (considered as an ASCII code) represents, `Undefined` if that argument is negative or greater than 255.

`distribute` :: `(*->**->***)->**->*->***`

is equivalent to the combinator *S*, this is defined by

```
distribute f g x = f x (g x);
```

`drop` :: `Integer->[*]->[*]`

takes a number and a list as arguments, and returns that list less that number of its initial elements. It is defined by:

```
drop n x =
    if n == 0 then x
    else drop (n - 1) (tl x)
    endif;
```

`dropwhile` :: `(*->Boolean)->[*]->[*]`

removes from its list argument its initial elements such that its first argument applied to them returns `True`, and is defined by

```
dropwhile f x =
    if isnil x then []
    elsif f (hd x) then dropwhile f (tl x)
    else x
    endif;
```

`error` :: `[Char]->Undefined`

always returns `Undefined`. Its first argument is discarded, but as a side-effect, a *ginger* implementation may (for example) print out the string on *stderr*. It is strict in the first argument.

`exp` :: `Number->Real`

returns *e* (*2.718 ...*) to the power of its argument.

`filter` :: `(*->Boolean)->[*]->[*]` takes a function `f` and a list `lis`, and returns the list of elements `elt` of `lis` such that `(f elt)` is `True`. It is defined by:

```
filter f x =
    if isnil x then []
    elsif f (hd x) then hd x : filter f (tl x)
    else filter f (tl x)
    endif;
```

`floor` :: `Number->Integer`

returns the largest integer less than or equal to its argument.

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

is strict in its third argument and is defined by

```
foldl op r x =
    if isnil x then r
    else foldl op (strict op r (hd x)) (tl x)
    endif;
```

`foldl1` :: `(*->*->*)->[*]->*`

is defined by:

```
foldl1 op x = foldl op (hd x) (tl x)
```

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

is strict in its third argument and is defined by

```
foldr op r x =
    if isnil x then r
    else op (hd x) (foldr op r (tl x))
    endif;
```

`foldr1` :: `(*->*->*)->[*]->*`

is defined by:

```
foldr1 op x =
    if isnil (tl x) then hd x
    else op (hd x) (foldr1 op (tl x))
    endif;
```

`force` :: `*->*`

takes an argument and forces all components of its argument to be fully evaluated.

`getenv` :: `[Char]->[Char]`

returns the value assigned under UNIX to its first argument (considered as the name of a UNIX environment variable); this functional does *not* preserve referential transparency, since the values of variables may be changed between different invocations of a *ginger* program.

`hd` `:: [*]->*`

returns the first element of its argument (`Undefined` if its argument is null)..

`hugenum` `:: Real`

is the largest `Real` number representable - the value of `DBL_MAX` in C.

`id` `:: *->*`

is the identity function.

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

returns its list argument less the last element, and is defined by:

```
init x =
    if isnil (tl x) then []
    else hd x : init (tl x)
    endif;
```

`isdigit` `:: Char->Boolean`

returns `True` if its argument represents a digit (`’0’` ... `’9’`).

`isletter` `:: Char->Boolean`

returns `True` if its argument represents a letter (`’A’` ... `’Z’`, `’a’` ... `’z’`).

`isnil` `:: [*]->Boolean`

returns `True` if its argument is an empty list.

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

is defined by:

```
iterate f x = x : iterate f (f x);
```

`last` `:: [*]->*`

returns the last element of its list argument, and is defined by:

```
last x =
    if isnil (tl x) then hd x
    else last (tl x)
    endif;
```

`log` `:: Number->Real`

returns the logarithm (base *e*) of its argument, `Undefined` if its argument is negative.

`log10` `:: Number->Real`

returns the logarithm (base 10) of its argument, `Undefined` if its argument is negative.

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

is defined by

```
map f x =
    if isnil x then []
    else f (hd x) : map f (tl x)
    endif;
```

`max` :: `[*]->*`

takes a list of expressions and returns the one of maximum value.

`max2` :: `*->*->*`

returns the maximum of its two arguments.

`maxint` :: `Integer`

is the largest `Integer` number representable - the value of `LONG_MAX` in C.

`min` :: `[*]->*`

takes a list of expressions and returns the one of minimum value.

`min2` :: `*->*->*`

returns the minimum of its two arguments.

`minint` :: `Integer`

is the smallest `Integer` number representable - the value of `LONG_MIN` in C.

`neg` :: `Number->Number`

returns the value of its argument with sign reversed.

`or` :: `[Boolean]->Boolean`

takes a list of boolean expressions and returns their logical disjunction. Defined by:

```
or x =
    if isnil x then False
    else hd x | or (tl x)
    endif;
```

`product` :: `[Number]->Number`

multiplies the elements of its list argument together.

```
product x =
    if isnil x then 1
    else hd x * product (tl x)
    endif;
```

`postfix` :: `*->[*]->[*]`

adds its first element to the end of the list which is its second, and is defined by:

```
postfix a x = x ++ [a];
```

`read` :: `[Char]->[Char]`

attempts to read a file whose name is its string argument. If that file cannot be opened, `Undefined` is returned, otherwise the list of characters which comprise that file (excluding the EOF character). Input from Standard Input is achieved by giving the empty string `""` as argument to `read`.

`rep` :: `Integer->*->[*]`

returns a list, of length its first argument, containing only instances of its second argument:

```
rep n x =
    if n == 0 then []
    else x : rep (n - 1) x
    endif;
```

`repeat` :: `*->[*]`

returns an infinite list containing instances of its argument:

```
repeat x = x : repeat x;
```

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

reverses its list argument.

`showfloat` :: `Integer->Number->[Char]`

takes an integer *n* and a number *x* and returns a string representing *x* displayed to *n* decimal places. `Undefined` if *n* is negative.

`shownum` :: `Number->[Char]`

returns a string representing its argument.

`showscaled` :: `Integer->Number->[Char]`

takes an integer *n* and a number *x* and returns a string representing *x* displayed to *n* decimal places in scientific notation (*m.mmmmme[+/-]mm*). `Undefined` if *n* is negative.

`sin` :: `Number->Real`

returns the sine of its argument (in radians).

`sqrt` :: `Number->Real`

returns the square root of its argument (`Undefined` if its argument is negative).

`strict` :: `*->*->*`

is defined by

```
strict f x = f x;
```

but is strict in its second argument.

`sum` :: `[Number]->Number`

takes a list and returns the sum of its elements. Defined by:

```
sum x =
    if isnil x then 0
    else hd x + sum (tl x)
    endif;
```

`system` :: [Char]->[]

always returns the null list, but as a side effect executes the UNIX command which is its first argument. *Use with care!* One use for `system` is for debugging - for instance to trace the function `f`, one might have:

```
fdebug f = strict (const f)
        (system "/bin/echo Function f invoked");
```

`take` :: Integer->[*]->[*]

returns a list which is the initial segment of its second argument, of length the first argument, defined by:

```
take n x =
   if n == 0 then []
   else hd x : take (n - 1) (tl x)
   endif;
```

`takewhile` :: (*->Boolean)->[*]->[*]

creates the initial segment of its list argument such that its first argument applied to them returns `True`, and is defined by

```
takewhile f x =
    if isnil x then []
    elsif f (hd x) then hd x : takewhile f (tl x)
    else []
    endif;
```

`tan` :: Number->Real

returns the tangent of its argument (in radians).

`tinynum` :: Real

is the smallest `Real` number representable - the value of `DBL_MIN` in C.

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

returns the list which is its argument less the first element.

`undef` :: Undefined

is the *bottom* value.

`until` :: (*->Boolean)->(*->*)->*->*

is defined by:

```
until f g x =
    if f x then x
    else until f g (g x)
    endif;
```

5. Infix Operators

The following infix operators are implemented (see the previous section for their definitions). The following table lists them in order of increasing binding power together with their associativity.

Operators	*Associativity*
`: ++`	right associative
`|`	left associative
`&`	left associative
`< <= == > >= ~=`	not associative
`+ -`	left associative
`% * /`	left associative
`^`	right associative
`.`	left associative
`!`	left associative

Table 2: Infix operators

6. Parallelism Operators

The following operators are used where *ginger* simulates a parallel machine.

`leftchild` `:: Integer->Integer`

`leftchild n` returns the processor which is the left child of processor `n` in a binary tree network. If the real network is not a binary tree, then a binary tree network is mapped onto the actual network, and appropriate results are given as if the network were a binary tree.

`neighbours` `:: Integer->[Integer]`

returns a list of the processors which are the nearest neighbours of its argument.

`parallel` `:: Integer->*->*->*`

has the semantics

```
parallel n x y = y;
```

but on a parallel machine indicates that expression `x` should be offshipped to processor `n`. When the evaluation of `x` is complete, its value is left on processor `n`, and migrated to its parent processor upon demand from that processor.

`rightchild` `:: Integer->Integer`

`rightchild n` returns the processor which is the right child of processor `n` in a binary tree network, as for `leftchild`.

`whereis` `:: *->Integer`

returns the number of the processor on which the graph representing its argument is located.

In order for programs written for a parallel machine to evaluate on a non-parallel machine, default definitions for the above operators are provided, as follows:

```
leftchild n = 0;
neighbours n = [];
rightchild n = 0;
parallel n x y = y;
whereis x = 0;
```

7. Divide-and-Conquer

In [1] an alternative implementation strategy for lists based on the divide-and-conquer principle was proposed. *Ginger* implements these "divide-and-conquer lists". In this section we specify the primitives *ginger* provides. Note that these lists are enabled in place of ordinary lists only if the option "*-d*" is given to the command `ginger`.

`element` :: `[*]->*`
: returns the element of the singleton list `x`.

`fst` :: `[*]->*`
: returns the first element of `split x`.

`issingleton` :: `[*]->Boolean`
: returns `True` if its argument is a singleton list.

`reduce` :: `(*->*->*)->*->[*]->*`
: is the divide-and-conquer equivalent of `foldl` and `foldr`. It is defined by

```
reduce f z x =
    if isnil x then z
    elsif issingleton x then element x
    else f (reduce f z (fst x)) (reduce f z (snd x))
    endif;
```

`snd` :: `[*]->*`
: returns the second element of `split x`.

`singleton` :: `*->[*]`
: returns a list containing its one element as argument.

These functions all return `Undefined` if *ginger* is not running divide-and-conquer lists, for two reasons. First of all, `fst` and `snd` cannot be easily defined in terms of conventional lists, and secondly, we felt it prudent to attempt to prevent users from mixing programming methodologies.

Conversely, when *ginger* is running using divide-and-conquer lists, the following functions *cannot* be used:

```
dropwhile foldl foldl1 foldr foldr1 iterate repeat takewhile
```

8. Bibliography

References

1. T.H. Axford and M.S. Joy, "List Processing Primitives for Parallel Computation," *Computer Languages*, 19, 1, pp. 1-17 (1993).
2. R. Bird and P.L. Wadler, *Introduction to Functional Programming,* Prentice Hall, Hemel Hempstead (1988). ISBN 0-13-484197-2 (pbk).
3. A.J.T. Davie, *An Introduction to Functional Programming Using Haskell,* Cambridge University Press, Cambridge, UK (1992). ISBN 0-521-27724-8 (pbk).
4. H. Glaser, C. Hankin, and D. Till, *Principles of Functional Programming,* Prentice-Hall, Englewood Cliffs, NJ (1984). ISBN 0-13-709163-X (pbk).
5. J.R. Hindley, B. Lercher, and J.P. Seldin, *Introduction to Combinatory Logic,* Cambridge University Press (1972).
6. J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-Calculus,* Cambridge University Press, Cambridge, UK (1986). London Mathematical Society Student Texts 1. ISBN 0-521-31839-4 (pbk).
7. I. Holyer, *Functional Programming with Miranda,* Pitman (1991). ISBN 0-273-03453-7.
8. P.R. Hudak and P.L. Wadler, "Report on the Functional Programming Language Haskell," Research Report CSC/89/R5, Department of Computer Science, University of Glasgow, Glasgow, UK (1989).

9. M.S. Joy and T.H. Axford, “GCODE: A Revised Standard Graphical Representation for Functional Programs,” *ACM SIGPLAN Notices,* 26, 1, pp. 133-139 (1991). Also Research Report 159, Department of Computer Science, University of Warwick, Coventry (1990) and Research Report CSR-90-9, School of Computer Science, University of Birmingham (1990).

10. G. Michaelson, *An Introdunction to Functional Programming through Lambda Calculus,* Addison-Wesley, Wokingham, Berks. (1989). ISBN 0-201-17812-5.

11. S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, London, UK (1987). ISBN 0-13-453325-9 (pbk).

12. S.L. Peyton Jones and M.S. Joy, “FLIC - a Functional Language Intermediate Code,” Research Report 148, Department of Computer Science, University of Warwick, Coventry, UK (1989). Revised 1990. Previous version appeared as Internal Note 2048, Department of Computer Science, University College London (1987).

13. D.A. Turner, “A New Implementation Technique for Applicative Languages,” *Software - Practice and Experience*, 9, pp. 31-49 (1979).

14. D.A. Turner, “Miranda: A Non-Strict Functional Language with Polymorphic Types” in *Functional Programming Languages and Computer Architecture,* ed. J.-P. Jouannaud, pp. 1-16, Springer-Verlag, Berlin, DE (1985). Lecture Notes in Computer Science 201; ISBN 3-540-15975-4; Proceedings of Conference at Nancy.

Appendix 1: Formal BNF

Ginger first of all passes its input through the C preprocessor *cpp* and the *ginger* input may consequently include *cpp* directives. These are distinguished by lines commencing with the character #; such lines are ignored by *ginger* . The BNF which follows assumes such lines have been filtered out.

```
script              ::=    <statement>
                     |     <statement> <script>

statement           ::=    <definition> ";" | <expression> ";"

definition          ::=    <identifiers> "=" <expression>

identifiers         ::=    <identifier> <identifiers>
                     |     <identifier>

definitions         ::=    <definition>
                     |     <definition> ";" <definitions>

expression          ::=    <application>
                     |     <abstraction>
                     |     "let" <definitions> "in" <expression> "endlet"
                     |     <expression> "where" <definitions> "endwhere"
                     |     "if" <expression> "then" <expression> <ifendexp>

ifendexp            ::=    "else" <expression> "endif"
                     |     "elsif" <expression> "then" <expression> <ifendexp>

simple              ::=    <real>
                     |     <int>
                     |     <bool>
                     |     <char>
                     |     <string>
                     |     <list>
                     |     <identifier>
                     |     <operator>
                     |     "(" <expression> ")"

int                 ::=    "-" <digitsequence> | <digitsequence>
real                ::=    <realA>
                     |     <realA> <exp> <digitsequence>
                     |     <realA> <exp> "+" <digitsequence>
                     |     <realA> <exp> "-" <digitsequence>
realA               ::=    <int> "."
                     |     <int> "." <digitsequence>
exp                 ::=    "e" | "E"

bool                ::=    "True" | "False"

char                ::=    "'" <singlechar> "'"
string              ::=    """ <stringterminator>

stringterminator    ::=    """
                     |     <singlestring> <stringterminator>
```

singlechar ::= <hexchar>
| <escapedchar>
| *any single character with the exception of single-quote*

singlestring ::= <hexchar>
| <escapedchar>
| *any single character with the exception of double-quote*

hexchar ::= "\" <hexinitiator> <hexdigit> <hexdigit>
hexinitiator ::= "x" | "X"
escapedchar ::= "\" *any character except <hexinitiator>*

application ::= <application> <infixoperator> <application>
| <application> <simple>
| <simple>

abstraction ::= "\" <identifier> <expression>

list ::= <nulllist>
| <finitelist>
| <dotdotlist>

nulllist ::= "[" "]"

finitelisttail ::= <expression> "]"
| <expression> "," <finitelisttail>
finitelist ::= "[" <finitelisttail>

dotdotlist ::= "[" <expression> ".." "]"
| "[" <expression> "," <expression> ".." "]"
| "[" <expression> ".." <expression> "]"
| "[" <expression> "," <expression> ".." <expression> "]"

identifier ::= <identifierinitiator>
| <identifier> <identifierchar>

digitsequence ::= <digit>
| <digit> <digitsequence>

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
hexdigit ::= <digit> | "A" | ... | "F" | "a" | ... | "f"
letter ::= "A" | ... | "Z" | "a" | ... | "z"
identifierinitiator ::= <letter> | "_"
identifierchar ::= <identifierinitiator> | <digit>

operator ::= *any predefined operator - see above*
infixoperator ::= *any predefined infix operator - see above*

comment ::= <commentstart> *anything except <commentend>* <commentend>
commentstart ::= "/*"
commentend ::= "*/"

Appendix 2: Running Ginger under UNIX

Ginger is called by the command `ginger`, and may be called with or without filename arguments; without filename arguments input is from the standard input stream, otherwise from the named files (the Standard Input may be named explicitly with the filename `-`). The input is then passed through the C preprocessor `cpp` before being parsed.

A number of options are available.

- `-c` The program is compiled only, the intermediate code GCODE sent to the standard output.
- `-d` Divide-and-conquer lists are implemented in lieu of "ordinary" lists.
- `-D` *n* Debug facility. The number *n*, in binary, indicates the amount of debugging to be produced; if *n* is *0*, or is omitted, a list of options will be displayed and *ginger* will terminate.
- `-h` Gives a short ‘usage’ help message and terminates.
- `-l` Output is "sugared" to display its internal structure. For instance, the string `"hello"` would be output as `['h','e','l','l','o']`
- `-m` Messages are output about resource usage, etc.
- `-M` *n* Changes size of (virtual) memory to *n* (default 10000).
- `-S` *n* The size of stack is changed to *n* (default 20).
- `-#` *n* When evaluation simulates a parallel machine, *n* processors are utilised.

Note that *ginger* does not (yet) have sophisticated debugging facilities. If you are developing a non-trivial functional program you are advised to write it first in (say) Miranda or Haskell, and then translate it into *Ginger*

Appendix 3: Identifiers Defined in Header Files

Several header files are provided, containing useful functions which are sufficiently complex not to be inbuilt Each of these header files may be accessed by (for instance)

```
#include <stdlib.g>
```

at the start of the *ginger* program. For their specific definitions, consult the relevant file.

`<stdio.g>`

This file contains functions relevant to i/o; in particular, several text formatting functions are provided.

`cjustify` `:: Number->[Char]->[Char]`

returns a string, of length its first argument, containing its second argument centered within that string, and padded with blanks.

`lay` `:: [[Char]]->[Char]`

takes a list of strings, and concatenates them together separating them with newline characters.

`layn` `:: [[Char]]->[Char]`

is as `lay`, but each string in the argument is prepended with its "line number".

`lines` `:: [Char]->[[Char]]`

is the reverse of `lay`; it takes a string containing newline characters, and returns a list of the substrings separating those newlines.

`ljustify` `:: Number->[Char]->[Char]`

returns a string, of length its first argument, containing its second argument, and padded out at the end with blanks.

`rjustify` `:: Integer->[Char]->[Char]`

returns a string, of length its first argument, containing its second argument, and padded out at the front with blanks.

`spaces` `:: Integer->[Char]`

returns a string, containing only blanks, of length its argument.

`<stdlib.g>`

This file contains miscellaneous functions.

`index` `:: [*]->[Integer]`

when applied to a list of length *n* will return `[0,1,2..n-1]`

`limit` `:: [*]->*`

takes a list containing values which converge to a limit; that limit is returned.

`merge` `::  [*]->[*]->[*]`

takes two lists (assumed sorted) and returns their merge.

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

is such that `scan op r` applies `foldl op r` to every initial segment of a list, and is defined by:

```
scan op = g
   where
   g r x =
      if isnil x then [r]
      else r : g (op r (hd x)) (tl x)
      endif
   endwhere;
```

`sort` :: [*]->[*]

sorts its list argument.

`transpose` :: [[*]]->[[*]]

takes a list of lists of expressions, thought of as representing a 2-dimensional matrix, and returns its transpose.

Appendix 4: Predefined Functions - Quick Reference

General Functions

Functions marked ‡ cannot be used with divide-and-conquer lists

operator	Where defined	Type
#	*inbuilt*	[*]->Integer
(!)	*inbuilt*	[*]->Integer->*
(%)	*inbuilt*	Integer->Integer->Integer
(&)	*inbuilt*	Boolean->Boolean->Boolean
(*)	*inbuilt*	Number->Number->Number
(+)	*inbuilt*	Number->Number->Number
(++)	*inbuilt*	[*]->[*]->[*]
(-)	*inbuilt*	Number->Number->Number
(/)	*inbuilt*	Number->Number->Number
(:)	*inbuilt*	*->[*]->[*]
(<)	*inbuilt*	*->*->Boolean
(<=)	*inbuilt*	*->*->Boolean
(==)	*inbuilt*	*->**->Boolean
(>)	*inbuilt*	*->*->Boolean
(>=)	*inbuilt*	*->*->Boolean
(^)	*inbuilt*	Number->Number->Number
(|)	*inbuilt*	Boolean->Boolean->Boolean
(~=)	*inbuilt*	*->**->Boolean
~	*inbuilt*	Boolean->Boolean
abs	*inbuilt*	Number->Number
acos	*inbuilt*	Number->Real
and	*inbuilt*	[Boolean]->Boolean
asin	*inbuilt*	Number->Real
atan	*inbuilt*	Number->Real
cjustify	*<stdio.g>*	Number->[Char]->[Char]
code	*inbuilt*	Char->Integer
compose	*inbuilt*	(*->**)->(***->*)->***->**
concat	*inbuilt*	[[*]]->[*]
const	*inbuilt*	*->**->*
converse	*inbuilt*	(*->**->***)->**->*->***
cos	*inbuilt*	Number->Real
decode	*inbuilt*	Integer->Char
distribute	*inbuilt*	(*->**->***)->(*->**)->(*->***)
drop	*inbuilt*	Integer->[*]->[*]
dropwhile	*inbuilt ‡*	(*->Boolean)->[*]->[*]
error	*inbuilt*	[Char]->Undefined
exp	*inbuilt*	Number->Real
filter	*inbuilt*	(*->Boolean)->[*]->[*]
floor	*inbuilt*	Number->Integer
foldl	*inbuilt ‡*	(*->**->*)->*->[**]->*
foldl1	*inbuilt ‡*	(*->*->*)->[*]->*
foldr	*inbuilt ‡*	(*->**->**)->**->[*]->**
foldr1	*inbuilt ‡*	(*->*->*)->[*]->*
force	*inbuilt*	*->*
getenv	*inbuilt*	[Char]->[Char]
hd	*inbuilt*	[*]->*
hugenum	*inbuilt*	Real

id	*inbuilt*	`*->*`
index	*<stdlib.g>*	`[*]->[Integer]`
init	*inbuilt*	`[*]->[*]`
isdigit	*inbuilt*	`Char->Boolean`
isletter	*inbuilt*	`Char->Boolean`
isnil	*inbuilt*	`[*]->Boolean`
iterate	*inbuilt ‡*	`(*->*)->*->[*]`
last	*inbuilt*	`[*]->*`
lay	*<stdio.g>*	`[[Char]]->[Char]`
layn	*<stdio.g>*	`[[Char]]->[Char]`
limit	*<stdlib.g>*	`[*]->*`
lines	*<stdio.g>*	`[Char]->[[Char]]`
ljustify	*<stdio.g>*	`Number->[Char]->[Char]`
log	*inbuilt*	`Number->Real`
log10	*inbuilt*	`Number->Real`
map	*inbuilt*	`(*->**)->[*]->[**]`
max	*inbuilt*	`[*]->*`
max2	*inbuilt*	`*->*->*`
maxint	*inbuilt*	`Integer`
merge	*<stdlib.g>*	`[*]->[*]->[*]`
min	*inbuilt*	`[*]->*`
min2	*inbuilt*	`*->*->*`
minint	*inbuilt*	`Integer`
neg	*inbuilt*	`Number->Number`
or	*inbuilt*	`[Boolean]->Boolean`
postfix	*inbuilt*	`*->[*]->[*]`
product	*inbuilt*	`[Number]->Number`
read	*inbuilt*	`[Char]->[Char]`
rep	*inbuilt*	`Integer->*->[*]`
repeat	*inbuilt ‡*	`*->[*]`
reverse	*inbuilt*	`[*]->[*]`
rjustify	*<stdio.g>*	`Integer->[Char]->[Char]`
scan	*<stdlib.g>*	`(*->**->*)->*->[**]->[*]`
showfloat	*inbuilt*	`Integer->Number->[Char]`
shownum	*inbuilt*	`Number->[Char]`
showscaled	*inbuilt*	`Integer->Number->[Char]`
sin	*inbuilt*	`Number->Real`
sort	*<stdlib.g>*	`[*]->[*]`
spaces	*<stdio.g>*	`Integer->[Char]`
sqrt	*inbuilt*	`Number->Real`
strict	*inbuilt*	`*->*->*`
sum	*inbuilt*	`[Number]->Number`
system	*inbuilt*	`[Char]->[]`
take	*inbuilt*	`Integer->[*]->[*]`
takewhile	*inbuilt ‡*	`(*->Boolean)->[*]->[*]`
tan	*inbuilt*	`Number->Real`
tinynum	*inbuilt*	`Real`
tl	*inbuilt*	`[*]->[*]`
transpose	*<stdlib.g>*	`[[*]]->[[*]]`
undef	*inbuilt*	`Undefined`
until	*inbuilt*	`(*->Boolean)->(*->*)->*->*`

Divide-and-Conquer Functions

operator	Where defined	Type
`element`	*inbuilt*	`[*]->*`
`fst`	*inbuilt*	`[*]->*`
`issingleton`	*inbuilt*	`[*]->Boolean`
`reduce`	*inbuilt*	`(*->*->*)->*->[*]->*`
`singleton`	*inbuilt*	`*->[*]`
`snd`	*inbuilt*	`[*]->*`

Parallelism Operators

operator	Where defined	Type
`leftchild`	*inbuilt*	`Integer->Integer`
`neighbours`	*inbuilt*	`Integer->[Integer]`
`parallel`	*inbuilt*	`Integer->*->*->*`
`rightchild`	*inbuilt*	`Integer->Integer`
`whereis`	*inbuilt*	`*->Integer`