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
2 Data Types
3 Syntax
4 Inbuilt Identifiers
5 Infix Operators
6 Parallel Operators
7 Divide-and-Conquer
8 Bibliography

Appendix 1 Formal BNF
Appendix 2 Running Ginger under UNIX
Appendix 3 Identifiers Defined in Header Files
Appendix 4 Predefined Operators - Quick Reference
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 explicitly 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 \( \lambda \)-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:
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 is an enumerated type (denoted by Boolean) with 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:
2.6. Strings

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

"hello"
"this string has a \t tab in it and a \a bell"
"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
\( \lambda x. X \)

is denoted by by the ginger expression

\( \ \backslash x \ X \)

Also, note that the where construct binds more tightly than \( \& \); thus

\( \ (\ & f \ f \ where \ \& = 99 \ endwhere) \ 88; \)
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.

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

4. Inbult 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", \([\text{Integer}]\) is the type "List of Integer", * is any type. The type Number refers to either \text{Integer} or \text{Real} - functions requiring an argument of such type will convert from \text{Integer} to \text{Real} if required.

\( (!) \) :: \( [*] \rightarrow \text{Integer} \rightarrow * \)
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.

\( \# \) :: \( [*] \rightarrow \text{Integer} \)
takes a list and returns its length.

\( (%) \) :: \( \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \)
is the standard modulus function, functionally equivalent to the C operator \( \% \). If the second argument is zero, Undefined is returned. Strict in both arguments.

\( (=) \) :: \( \text{Boolean} \rightarrow \text{Boolean} \rightarrow \text{Boolean} \)
is the logical conjunction operator. Strict in first argument, it is defined by:

\[
    x \ & \ y = \ if \ x \ then \ y \ else \ False \ endif;
\]

\( (*) \) :: \( \text{Number} \rightarrow \text{Number} \rightarrow \text{Number} \)
is multiplication; if both arguments are integers, an Integer is returned, otherwise a Real. Strict in both arguments.

\( (+) \) :: \( \text{Number} \rightarrow \text{Number} \rightarrow \text{Number} \)
is addition; if both arguments are integers, an Integer is returned, otherwise a Real. Strict in both arguments.
(+++) : [*]->[*]->[*]

is append; this is defined by

\[ x \ ++ \ y = \text{if } \text{isnil } x \ \text{then } y \]
\[ \quad \text{else } \text{hd } x : (\text{tl } x \ ++ \ y) \ \text{endif}; \]

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

\[ \text{hd} (1 : 2) \]

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:

\[ x \mid y = \text{if } x \text{ then } True \text{ else } y \text{ endif;} \]

(~) :: 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

\[ \text{abs } x = \text{if } x \geq 0 \text{ then } x \text{ else } \text{neg } x \text{ endif;} \]

acos :: Number->Real
returns the inverse cosine of its argument (in radians), Undefined if its argument has absolute value greater than 1.

and :: [Boolean]->Boolean
takes a list of booleans and returns their logical conjunction. Defined by

\[ \text{and } x = \text{if } \text{isnil } x \text{ then } True \text{ else } \text{hd } x \& \text{ and } (\text{tl } x) \text{ endif;} \]

asin :: Number->Real
returns the inverse sine of its argument (in radians), Undefined if its argument 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

\[ \text{compose } f \ g \ x = f \ (g \ x); \]

The infix form for compose is ".", thus

\[ (f . g) \ x = f \ (g \ x) \]
concat :: [[*]]->[*]
takes a list of lists and concatenates them. Defined by
\[
\text{concat } x =
\begin{align*}
&\text{if } \text{isnil } x \text{ then } [] \\
&\text{else } \text{hd } x \mathbin{\mathbin{++}} \text{concat} \ (\text{tl} \ x)
\end{align*}
\]
const :: *->[*]
returns its first argument and discards its second; equivalent to the combinator \(K\), this is defined by
\[
\text{const } x \ y = x;
\]
converse :: (*->[***])->[**]->*->***
is equivalent to the combinator \(C\), this is defined by
\[
\text{converse } f \ x \ y = f \ y \ x;
\]
cos :: Number->[Real]
returns the cosine of its argument (in radians).
de 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
\[
\text{distribute } f \ g \ x = f \ x \ (g \ x);
\]
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:
\[
\text{drop } n \ x =
\begin{align*}
&\text{if } n = 0 \text{ then } x \\
&\text{else } \text{drop} \ (n - 1) \ (\text{tl} \ x)
\end{align*}
\]
dropwhile :: (*->[Boolean])->[*]->[*]
removes from its list argument its initial elements such that its first argument applied to them returns True, and is defined by
\[
\text{dropwhile } f \ x =
\begin{align*}
&\text{if } \text{isnil } x \text{ then } [] \\
&\text{elsif } f \ (\text{hd} \ x) \text{ then } \text{dropwhile} \ f \ (\text{tl} \ x) \\
&\text{else } x
\end{align*}
\]
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 \( \text{lis} \), and returns the list of elements \( \text{elt} \) of \( \text{lis} \) such that \( (f \ \text{elt}) \) is True. It is defined by:

\[
\text{filter } f \ \text{x} = \\
\begin{cases}
\text{if isnil } \text{x} \text{ then } [] & \\
\text{elsif } f \ (\text{hd } \text{x}) \text{ then } \text{hd } \text{x} : \text{filter } f \ (\text{tl } \text{x}) & \\
\text{else } \text{filter } f \ (\text{tl } \text{x}) &
\end{cases}
\]

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

\[
\text{foldl } \text{op } \text{r } \text{x} = \\
\begin{cases}
\text{if isnil } \text{x} \text{ then } \text{r} & \\
\text{else } \text{foldl } \text{op } \text{(strict op } \text{r } \text{(hd } \text{x}))) \ (\text{tl } \text{x}) &
\end{cases}
\]

foldl1 :: (*->*->*)->[*]->* is defined by:

\[
\text{foldl1 } \text{op } \text{x} = \text{foldl } \text{op } \text{(hd } \text{x}) \ (\text{tl } \text{x})
\]

foldr :: (*->**->**)->**->[*]->** is strict in its third argument and is defined by

\[
\text{foldr } \text{op } \text{r } \text{x} = \\
\begin{cases}
\text{if isnil } \text{x} \text{ then } \text{r} & \\
\text{else } \text{op } \text{(hd } \text{x}) \ (\text{foldr } \text{op } \text{r } \text{(tl } \text{x})) &
\end{cases}
\]

foldr1 :: (*->*->*)->[*]->* is defined by:

\[
\text{foldr1 } \text{op } \text{x} = \\
\begin{cases}
\text{if isnil } (\text{tl } \text{x}) \text{ then } \text{hd } \text{x} & \\
\text{else } \text{op } \text{(hd } \text{x}) \ (\text{foldr1 } \text{op } \text{(tl } \text{x})) &
\end{cases}
\]

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->[a]->[a]
returns a list, of length its first argument, containing only instances of its second argument:

\[
\text{rep } n \ x = \\
\text{if } n == 0 \ \text{then } [] \\
\text{else } x : \text{rep } (n - 1) \ x \\
\text{endif;}
\]

repeat :: *->[a]
returns an infinite list containing instances of its argument:

\[
\text{repeat } x = x : \text{repeat } x;
\]

reverse :: [a]->[a]
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 (\( m.mmmme[+/-]mm \)). 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 :: *->[a]->[a]
is defined by

\[
\text{strict } f \ x = f \ x;
\]
but is strict in its second argument.

sum :: [Number]->Number
takes a list and returns the sum of its elements. Defined by:

\[
\text{sum } x = \\
\text{if } \text{isnil } x \ \text{then } 0 \\
\text{else } \text{hd } x + \text{sum } \text{tl } x \\
\text{endif;}
\]
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:

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

take :: Integer->[a]->[a]
returns a list which is the initial segment of its second argument, of length the first argument, defined by:

```haskell```
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

```haskell```
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:

```haskell```
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.
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
  
  \[ \text{parallel} \ n \ x \ y = y; \]
  
  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.

\[
\text{element} \quad ::= \quad [*]->* \\
\text{return the element of the singleton list } x.
\]

\[
\text{fst} \quad ::= \quad [*]->* \\
\text{return the first element of } \text{split } x.
\]

\[
\text{issingleton} \quad ::= \quad [*]->\text{Boolean} \\
\text{return True if its argument is a singleton list.}
\]

\[
\text{reduce} \quad ::= \quad (*->*->*)->*->[*]->* \\
\text{is the divide-and-conquer equivalent of } \text{foldl and foldr}. \text{It is defined by}
\]

\[
\begin{align*}
\text{reduce } f \ z \ x &= \\
\text{if isnil } x \text{ then } z \\
\text{elsif issingleton } x \text{ then } \text{element } x \\
\text{else } f \left( \text{reduce } f \ z \ (\text{fst } x) \right) \left( \text{reduce } f \ z \ (\text{snd } x) \right) \\
\text{endif;}
\end{align*}
\]

\[
\text{snd} \quad ::= \quad [*]->* \\
\text{return the second element of } \text{split } x.
\]

\[
\text{singleton} \quad ::= \quad *->[*] \\
\text{return a list containing its one element as argument.}
\]

These functions all return Undefined if ginger is not running divide-and-conquer lists, for two reasons. First of all, \text{fst} and \text{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:

\[
dropwhile \text{foldl} \text{ foldl1 foldr foldr1 iterate repeat takewhile}
\]

8. Bibliography

References


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.

```plaintext
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> "" | ":" <singlestring> string terminator
string ::= "#" | <singlestring> <string terminator>
```
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> "]"

finitelist ::= "["] <finitelisttail>

dotdotlist ::= "[" <expression> "," <dotdotlisttail>

dotdotlist ::= "["] <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 \texttt{cpp} before being parsed.

A number of options are available.

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

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

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*

<table>
<thead>
<tr>
<th>operator</th>
<th>Where defined</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>inbuilt</td>
<td>[*]-&gt;Integer</td>
</tr>
<tr>
<td>(!)</td>
<td>inbuilt</td>
<td>[<em>]-&gt;Integer-&gt;</em></td>
</tr>
<tr>
<td>(%)</td>
<td>inbuilt</td>
<td>Integer-&gt;Integer-&gt;Integer</td>
</tr>
<tr>
<td>(@)</td>
<td>inbuilt</td>
<td>Boolean-&gt;Boolean-&gt;Boolean</td>
</tr>
<tr>
<td>(*)</td>
<td>inbuilt</td>
<td>Number-&gt;Number-&gt;Number</td>
</tr>
<tr>
<td>(+)</td>
<td>inbuilt</td>
<td>Number-&gt;Number-&gt;Number</td>
</tr>
<tr>
<td>(++)</td>
<td>inbuilt</td>
<td>[<em>]-&gt;[</em>]-&gt;[*]</td>
</tr>
<tr>
<td>(-)</td>
<td>inbuilt</td>
<td>Number-&gt;Number-&gt;Number</td>
</tr>
<tr>
<td>(/)</td>
<td>inbuilt</td>
<td>Number-&gt;Number-&gt;Number</td>
</tr>
<tr>
<td>(:)</td>
<td>inbuilt</td>
<td><em>-&gt;[</em>]-&gt;[*]</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>inbuilt</td>
<td><em>-&gt;</em>-&gt;Boolean</td>
</tr>
<tr>
<td>(&lt;=)</td>
<td>inbuilt</td>
<td><em>-&gt;</em>-&gt;Boolean</td>
</tr>
<tr>
<td>(==)</td>
<td>inbuilt</td>
<td>*-&gt;**-&gt;Boolean</td>
</tr>
<tr>
<td>(&gt;)</td>
<td>inbuilt</td>
<td><em>-&gt;</em>-&gt;Boolean</td>
</tr>
<tr>
<td>(&gt;=)</td>
<td>inbuilt</td>
<td><em>-&gt;</em>-&gt;Boolean</td>
</tr>
<tr>
<td>()</td>
<td>inbuilt</td>
<td>Number-&gt;Number-&gt;Number</td>
</tr>
<tr>
<td>()</td>
<td>inbuilt</td>
<td>Boolean-&gt;Boolean-&gt;Boolean</td>
</tr>
<tr>
<td>(=)</td>
<td>inbuilt</td>
<td>*-&gt;**-&gt;Boolean</td>
</tr>
<tr>
<td>(-)</td>
<td>inbuilt</td>
<td>Boolean-&gt;Boolean</td>
</tr>
<tr>
<td>abs</td>
<td>inbuilt</td>
<td>Number-&gt;Number</td>
</tr>
<tr>
<td>acos</td>
<td>inbuilt</td>
<td>Number-&gt;Real</td>
</tr>
<tr>
<td>asin</td>
<td>inbuilt</td>
<td>Number-&gt;Real</td>
</tr>
<tr>
<td>atan</td>
<td>inbuilt</td>
<td>Number-&gt;Real</td>
</tr>
<tr>
<td>cjustify</td>
<td>&lt;stdio.g&gt;</td>
<td>Number-&gt;[Char]-&gt;[Char]</td>
</tr>
<tr>
<td>code</td>
<td>inbuilt</td>
<td>Char-&gt;Integer</td>
</tr>
<tr>
<td>compose</td>
<td>inbuilt</td>
<td>(<em>-&gt;<strong>)-&gt;(</strong>-&gt;</em>)-&gt;*<strong>-&gt;</strong></td>
</tr>
<tr>
<td>concat</td>
<td>inbuilt</td>
<td>[[<em>]]-&gt;[</em>]</td>
</tr>
<tr>
<td>const</td>
<td>inbuilt</td>
<td><em>-&gt;**-&gt;</em></td>
</tr>
<tr>
<td>converse</td>
<td>inbuilt</td>
<td>(<em>-&gt;<strong>-&gt;</strong>)-&gt;**-&gt;</em>-&gt;***</td>
</tr>
<tr>
<td>cos</td>
<td>inbuilt</td>
<td>Number-&gt;Real</td>
</tr>
<tr>
<td>decode</td>
<td>inbuilt</td>
<td>Integer-&gt;Char</td>
</tr>
<tr>
<td>distribute</td>
<td>inbuilt</td>
<td>(<em>-&gt;<strong>-&gt;</strong>)-&gt;(</em>-&gt;<em>)-&gt;(</em>-&gt;***)</td>
</tr>
<tr>
<td>drop</td>
<td>inbuilt</td>
<td>Integer-&gt;[<em>]-&gt;[</em>]</td>
</tr>
<tr>
<td>dropwhile</td>
<td>inbuilt ‡</td>
<td>(<em>-&gt;Boolean)-&gt;[</em>]-&gt;[*]</td>
</tr>
<tr>
<td>error</td>
<td>inbuilt</td>
<td>[Char]-&gt;Undefined</td>
</tr>
<tr>
<td>exp</td>
<td>inbuilt</td>
<td>Number-&gt;Real</td>
</tr>
<tr>
<td>filter</td>
<td>inbuilt</td>
<td>(<em>-&gt;Boolean)-&gt;[</em>]-&gt;[*]</td>
</tr>
<tr>
<td>floor</td>
<td>inbuilt</td>
<td>Number-&gt;Integer</td>
</tr>
<tr>
<td>foldl</td>
<td>inbuilt ‡</td>
<td>(<em>-&gt;**-&gt;</em>)-&gt;*-&gt;**</td>
</tr>
<tr>
<td>foldl1</td>
<td>inbuilt ‡</td>
<td>(<em>-&gt;</em>-&gt;<em>)-&gt;[</em>]-&gt;*</td>
</tr>
<tr>
<td>foldr</td>
<td>inbuilt ‡</td>
<td>(<em>-&gt;**-&gt;</em>)-&gt;<strong>-&gt;[*]-&gt;</strong></td>
</tr>
<tr>
<td>foldr1</td>
<td>inbuilt ‡</td>
<td>(<em>-&gt;</em>-&gt;<em>)-&gt;[</em>]-&gt;*</td>
</tr>
<tr>
<td>force</td>
<td>inbuilt</td>
<td><em>-&gt;</em></td>
</tr>
<tr>
<td>getenv</td>
<td>inbuilt</td>
<td>[Char]-&gt;[Char]</td>
</tr>
<tr>
<td>hd</td>
<td>inbuilt</td>
<td>[<em>]-&gt;</em></td>
</tr>
<tr>
<td>hugenum</td>
<td>inbuilt</td>
<td>Real</td>
</tr>
</tbody>
</table>
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>  [*]->[*]
neg  inbuilt  Negative->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>  [*]->[*]
neg  inbuilt  Negative->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>  [*]->[*]
neg  inbuilt  Negative->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>  [*]->[*]
neg  inbuilt  Negative->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>  [*]->[*]
neg  inbuilt  Negative->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>  [*]->[*]
neg  inbuilt  Negative->Number
or  inbuilt  [Boolean]->Boolean
postfix  inbuilt  *->[*]->[*]
product  inbuilt  [Number]->Number
read  inbuilt  [Char]->[Char]
rep  inbuilt  Integer->*->[*]
repeat  inbuilt  (*->[*])-*->[*]
reverse  inbuilt  [*]->[*]
rjusti
Divide-and-Conquer Functions

<table>
<thead>
<tr>
<th>operator</th>
<th>Where defined</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>inbuilt</td>
<td>[*] -&gt; *</td>
</tr>
<tr>
<td>fst</td>
<td>inbuilt</td>
<td>[*] -&gt; *</td>
</tr>
<tr>
<td>issingleton</td>
<td>inbuilt</td>
<td>[*] -&gt; Boolean</td>
</tr>
<tr>
<td>reduce</td>
<td>inbuilt</td>
<td>(* -&gt; * -&gt; <em>) -&gt; * -&gt; [</em>] -&gt; *</td>
</tr>
<tr>
<td>singleton</td>
<td>inbuilt</td>
<td>* -&gt; [*]</td>
</tr>
<tr>
<td>snd</td>
<td>inbuilt</td>
<td>[*] -&gt; *</td>
</tr>
</tbody>
</table>

Parallelism Operators

<table>
<thead>
<tr>
<th>operator</th>
<th>Where defined</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftchild</td>
<td>inbuilt</td>
<td>Integer -&gt; Integer</td>
</tr>
<tr>
<td>neighbours</td>
<td>inbuilt</td>
<td>Integer -&gt; [Integer]</td>
</tr>
<tr>
<td>parallel</td>
<td>inbuilt</td>
<td>Integer -&gt; * -&gt; * -&gt; *</td>
</tr>
<tr>
<td>rightchild</td>
<td>inbuilt</td>
<td>Integer -&gt; Integer</td>
</tr>
<tr>
<td>whereis</td>
<td>inbuilt</td>
<td>* -&gt; Integer</td>
</tr>
</tbody>
</table>