The hbc compiler is a compiler for Haskell 1.2. It is based on the lmlc compiler and most of what is said about lmlc in the LML user's manual is true for hbc as well.

Haskell B. is an extended version of Haskell. This section describes the differences. Most of the extension are turned *off* by default, but can be used with the "-no-pedantic" flag. The section is organised into subsections in the same way as the Haskell report [Hud92].

Constructors do not have to begin with a upper case letter or a ":". Variables do not have to begin with a lower case letter.

- Gaps and & are allowed in Char constants.
- ISO8859-1 letters are allowed in identifiers. It's time to stop the english-speaking from forcing the rest of the world to adopt their alphabet. (This is the choice in Eiffel and Modula-3.) (I'm sorry that it's only western European languages that are readily useable, but ISO8859-1 seems to be the only widely accepted standard.)
- "`(,)`" is allowed and is the (curried) pairing function.
- Module identifiers may be strings, allowing the exact filename of an imported interface to be given. The module name given by import and the one in the interface need not agree, the one in the interface is taken as the module part of the original name.

- If the compiled files ends in ".lhs" it is assumed to be in literate style, i.e. only lines begining with "`>`" are assumed to be code, the rest are comments.
- The string "`{-#`" starts annotations. Unknown or badly placed annotations may cause an error.

The parsing of operators with that are neither left- not right-assocative is somewhat different. They may occur in an expression with associative operators of the same precedence without an error.

- A constructor can be declared to be strict in a subpart by putting `{-#STRICT#-}` after the type. E.g.

  ```
  data Length = MkLength Int {-#STRICT#-}
  ```

  gives you a type that is truly isomorphic to Int (and no runtime penalty at all).

- The restrictions on instance declarations have been relaxed. Previously they could only have the form

  `instance K => C (T` a_1 `...` a_n`)`

  where a_1 ... a_n are type vars. Now they can have the form

  `instance K => C (T` t_1 `...` t_n`)`

  where t_1 ... t_n are arbitrary type expressions. If more than one instance declaration can be used to resolve an overloading the following rule is used:

  > If there is a single most specific declaration (in the usual type specialisation ordering) then this is used, otherwise an ambiguity error is reported.

  This means that you can make declarations like

  ```
  instance Eq [Char] where ...
  ```

  and make a special purpose fast comparison routine for strings and get this whenever strings are compared. So far this has not been used anywhere in the Prelude, but a faster version of Ratio Integer (coded in C) will probably be included in the next version.

  This is a dubious feature and should not be used carelessly.

- The type system has been extended with existential types as described in [Läu92]. This feature can be turned on separately with the "-fexist-type" flag.

  All type variables that are free in a type definition are considered to be existentially quantified just before the constructor in which they appear. Example

```
data T = C a (a->Int)
```

Here `a` is free (since it does not appear after `T`) and thus existentially quantified. When constructing values `C` is polymorphic as usual.

```
hetl = [C 4 (+1), C [1,2] length, C 3.14 toInt]
```

This list has typing `hetl :: [T]`, but is really heterogenous since the elements have different types. When a constructor with a existentially quantified type is used in pattern matching[1] it behaves differently. The actual type of the quantified variable is not allowed to escape outside the expression tied to the pattern matching.

The following function is allowed

```
f c =
    case c of
       C v f -> f v
```

since it has typing `f :: T->Int`, but not this

```
g c =
    case c of
       C x y -> x
```

since the actual type of `x` is not know.

Existentially quantified variables can also be constrained by giving a context before a constructor. The following defines a type where the argument to the constructor has to have a type belonging to class `Text`.

```
data TText = (Text a) => T a
```

This can be used as follows:

```
l ::  [TText]
l = [T True, T 1, T (atan 1), T [True]]
s = map show l
```

and the value of `s` will be `["True","1","7.8539816339744828e-1","[True]"]`.

The automatic derivation of instance declarations does not work for types containing existentially quantified variables.

- On renaming the fixity of the original name is not given to the new one.

[1] Only in case in the current implementation.

- Tuples are only supported up to a bounded size.
- The class `Num` has a new method `fromInt` to convert from an `Int` to the type.
- The class `Integral` has the new method `toInt` to convert from the type to an `Int`.
- The class `Text` has the new method `showsType` which is similar to `shows`, but shows the type instead. The default method for `showsPrec` shows the type enclosed in "`<< >>`".

- There are several new I/O requests:

`Sleep x`	Suspends execution of the program for `x::Double` seconds.
`ChangeDirectory s`	Change current directory to `s::String`.
`GetTime`	Get the current time. Returns a `Response` of the new kind `Dbl Double` giving the time in seconds since 1970.
`DeleteDirectory s`	Delete the directory `s::String`.
`System s`	Executes `s::String` as a shell command. Returns `Success` or `OtherError errorcode`.
`ReadDirectory s`	Reads the directory s and returns it as a list of strings, one for each file in the directory.
`GetCpuTime`	Returns the amount of CPU time the program has consumed in seconds as a `Dbl Double`.

- In the optional ReadChannels request there are now two special kind of channels: "`TIMEOUT:d`" where d is a floating point number. Every time the system starts checking for input on any of the given channels there will be a timeout of d seconds, after this the channel "`TIMEOUT:d`" will generate the character `'*'` (Why `'*'`?, you ask. The answer is the decimal ISO-8859-1 code for `'*'`.) The channel "`TICK:d`" is similar but it is guaranteed to generate a `'*'` every d seconds whether the system is reading or not. This normally what you want for real time games. There can be at most one "`TICK`" or "`TIMEOUT`" channel per ReadChannels request.
- The character `'\f'` is not handled as specified in the section on presentation. It should clear the screen (or make a form feed), but it is sent as it is to the standard output.

This section describes the use of the interactive system. The interactive system can load both source files and compiled files. Functions defined with the interactive system runs 10 – 50 times slower than compiled code; compiled code called from interactive use runs at full speed.

The user interface is very spartan; the system should be used from inside EMACS or a window system to get command editing etc.

The system is started by

```
hbi
```

and you will then be greeted by (something like)

```
Welcome to interactive Haskell B. version 0.998.4 I80386 1992 Aug 20!
Loading prelude...  960 values, 70 types found.
Type "help;" to get help.
>
```

WARNING! The interactive system is still somewhat untested and there may be many lurking bugs (e.g. interrupting programs sometimes kills the system). Tread carefully!

The interactive version of Haskell B uses dynamic loading of object files, since this is rather complicated for some machines it has not been implemented for all machines yet.

The first prompt is "> " and any subsequent prompts until a command is completed will be "`#` ". The following commands are available (note that they all end with semi-colon):

expression ;	Evaluate and print the expression (more on printing below).
`let` { *decl* } ;	Make the bindings in declarator and make them available in the rest of the session. No evaluation of the bound objects is performed. The usual offside rule applies here.
topdecl	Any top level declaration except *decl*.
`load` *string* ;	Load a file. The string specifies what should be loaded, it can have three forms: *.hs — load a .hs file. This must be an ordinary Haskell B module. *.o — load .o and .hi files. These files must be produced by compiling an ordinary Haskell B module with hbc. * — loads *.hs if it is newer than *.o, otherwise the *.o file.
`import` *ident* ;	Load a .o file using the $HBCINCPATH, used to load library stuff.
`source` *string* ;	Read commands from the given file.
`whatis` *ident* ;	Give information about the identifier *ident*.
`help` ;	Print some information about hbi.
`quit` ;	Quit the system.

The defaults for numeric types are `Integer` and `Double` instead of `Int` and `Double` in the compiler. (This is nicer when hbi is used as a calculator.)

It is possible to interrupt the evalution of an expression by hitting the interrupt key. If the program does not stop after the first at first try, then try again. The first only interrupts the program at certain well defined places and is less dangerous. The second stops the program anywhere except in garbage collection.

The system catches most errors and prints an appropriate message. Unfortunately some errors, like stack overflow, are uncaught and the system just dies.

Values of type `String` are printed as they are, all other values are printed by first applying `show` to it and then printing the resulting string.

During garbage collection the string " GC " will occur on the output.

The hbc command can be used to compile a module or link compiled module together.

The Haskell report does not specify how Haskell modules, files etc. should be treated by the compilation system. Hbc takes a very simple approach, each file should contain exactly one module and the name of the file should be the same as the module name with the extension ".hs" added. When such a file is compiled by hbc an interface file and an object file are produced. The interface file has the same name as the source file, but with the extension ".hi" instead of ".hs", the object file hase the ".o" extension.

When compiling an "`import`" declaration the compiler needs to consult the interface file for the imported module. To find the interface file the compiler looks for the module name extended with ".hi" in the *include path*. The include path is a colon separated list of directories that are searched in order for the imported files. The include path can be set in several ways. The default value is "`.:$LMLDIR/hbc_library`", where `$LMLDIR` is the value of the `LMLDIR` environment variable (which defaults to `/usr/local/lib/lmlc` if not set). If the environment variable `HBCINCPATH` is set this will be taken as the value of the include path instead. After this initial value of the include path has been established it can also be altered by the "-i" flag on the command line. Just "-i" flag will empty the include path. A "-i*path*" will prepend *path* to the include path.

The include path is also used during linking. All files named "lib.a" (or "lib_p.a" or "lib_ph.a" if the "-p" respective "-ph" flags have been used) in the directories in the include path will be given to the linker. This means that a directory in the include path should contain (at least) a number of interface files, and the corresponding object files in an archive called "lib.a".

The interactive system uses the same mechanism for its `import` command. But for this to work properly the directories in the include path should also contain the object files for each of the interface files, since hbi cannot (yet) use archives.

Module names that begin with "LML" are supposed to be used for code compiled with the LML compiler. The internal naming scheme is somewhat different for these modules. To mix Haskell and LML, you should compile the LML module with the "-fboth" flag to get Haskell interface files.

If the file "Main.hs" contains

```
main _ = [AppendChan stdout (hello "world")]
hello s = "hello, " ++ s
```

then it can be compiled with the command

```
hbc Main.hs -o hello
```

This gives an executable file called "hello".

4.3.1 Ordinary flags

-c	Separate compilation flag. Will produce a ".o" file only, without linking.
-D*name*	Define a symbol (this flag is sent to the preprocessor).
-I*name*	Search for include files in *name* (this flag is sent to the preprocessor).
-l*name*	Search for object files in archive *name* (this flag is sent to the linker).
-o *name*	Name the executable file *name*.
-S	Produce an assembly file only. This file has suffix ".s".
-T	Produce code to enable runtime tracing. For further information look a the section on runtime flags.
-p	Enable profiling with prof.
-pg	Enable profiling with gprof. The information from gprof is difficult to interpret since the order in which functions are called with lazy evaluation is rather strange.
-ph	Enable heap profiling. For further information look at the section on runtime flags.
-i*path*	Prepend path to the include path. If *path* is empty the include path is emptied.
-M	Enable preprocessing by the C preprocessor for Haskell.

4.3.2 Debug flags

-f*flag* "Advanced" compiler flags. These flags are normally never used, the description is here for information only. All these flags can be given with or without a loading "no-". The list below shows the non-default setting.

allow-redef	Allow redefinition of functions Prelude (Haskell only).
no-asimpl	Do not do "advanced" reorderings of arithmetic expressions.
auto-derive	Make derived instances automatically (Haskell only).
no-B-conv	Do not eliminate the use of the V stack.
both	Produce both an LML and a Haskell interface file.
no-C-scheme	Use a different approach in the C scheme.
caseopt	Transform strict expressions containing case expressions, this may cause code explosion.
no-chkind	
code	Produce assembly code.
conctype	Compile conctypes.
copy-indir	($\langle\nu,\ G\rangle$only).
no-copy-ret	($\langle\nu,\ G\rangle$only).
curry	Compile Haskell.
do-extra*N*	Allocate *N* extra slots for tail call ($\langle\nu,\ G\rangle$only).
no-elimdup	($\langle\nu,\ G\rangle$only).

no-eval-opt Do not use information on what variables that have already been evaluated.

no-evalupdunw Do not use runtime system code to return the value of a variable.

no-exist-type Do not allow existential types.

no-exist-escape-check Do not check if existential types escape their scope (dangerous).

no-export-check Do not check exports (Haskell only).

no-flat-names Do not use flat namespace (Haskell only).

fullname Print full names (include leading '_').

G-code Produce G-code.

no-G-opt Do not optimize the G-code.

no-gen-cmp Do not use generic comparison when possible (Haskell only).

no-gen-read Do not generate a readsPrec function in derived instances (Haskell only).

in-flic Read FLIC code.

indir Generate code that uses indirection nodes.

no-M-opt Do not optimize the M-code.

no-noeval Never add the NOEVAL annotation during analysis, instead re-evaluate each time used.

nu For the $\langle \nu, G \rangle$-machine.

out-flic Print the FLIC code for the program.

no-overload-restr Do not use the monomorphism restriction.

parallel Code for SPARKs etc ($\langle \nu, G \rangle$ only).

pedantic Compile Haskell instead of Haskell B.

no-prelude Do not import Prelude (Haskell only).

profile Emit code to do profiling.

profile-heap Emit code to do heap profiling.

relax More relaxed checking (Haskell only).

no-rev-vis Do not make module entities visible during interface checking (Haskell only).

semi-strict Compile code for strict evaluation that can be mixed with lazy code.

no-simpl Do not do any compile time reductions.

sparc8 Use the smul/sdiv instructions available on SPARC-8.

no-spark No sparks emitted ($\langle \nu, G \rangle$ only).

statistics Emit code that counts everything ($\langle \nu, G \rangle$ only).

stingy Emit code for stingy evaluation.

no-strcmp Do not generate special code for string comparison.

strict Compile code for strict evaluation.

no-strict-anal Do not do any strictness analysis.

no-subst Do not do compile time β-reductions.

test-eval Emit code that explicitely tests if a node is evaluated before do a call to evaluate it. This saves time but wastes code space.

no-toprec Do not make top level mutually recursive (Haskell only).

trace Produce code to enable runtime tracing.

no-type Do not do type checking.

no-unrec	Do not remove recursion.
no-vector	Do not use vector apply nodes.
verbose	Be talkative.
zap-redex	Overwrite the top application node when entering a function. This plugs some memory leaks.
zero-fill	Fill frame slots with 0 ($\langle \nu, G \rangle$only).
all	Print the internal form after each pass.
errmap	Print the error message map.
input	Print the internal form of the syntax tree after it has been read.
pcurry	Print the curry transformation (Haskell only).
rename	Print after rename pass.
constr	Print after constructor transformation pass.
remmatch	Print after pattern matching transformation.
rem-sign	Print after signature removal (Haskell only).
rem-class	Print after class removal (Haskell only).
rem-lazy	Print after lazy pattern removal (Haskell only).
genderiv	Print after derived instance generation (Haskell only).
remzf	Print after list comprehension removal (Haskell only).
remdeep	Print after declarator transformation.
Ecnv	Print after conversion to another internal form.
addrestr	Print after adding type extraction restrictions.
ptype	Print the type.
apconv	Print after application transformation.
eqtrans	Print after comparison transformation.
simpl	Print after compile time reductions.
pstrict	Print after strictness analysis.
addclose	Print after closures have been added.
lambdalift	Print after lambda-lifting.
bconv	Print after V stack elimination.
addarity	Print after arity information has been added.
transformed	Print program after all program transformations are done.
Gcode-unopt	Print the unoptimized G-code.
gcode	Print the G-code.
mcode	Print the M-code.
debug	Print more information.
import-debug	Print the import list.
appearley	
subearley	Conctype.
g-live	
g-flow	
g-contents	
g-dup	
g-save	
g-temp	
g-join	
g-2-op	
g-rcg	

	g-spill	
	g-bblock	
	no-gcheck	
	spill-msg	$\langle\nu,\ G\rangle$register alllocation.
	test	Some extra messages.

-H*size*	Set the heapsize to *size* during compilation.
-t	Skip typchecking phase.
-v	Make the front end verbose.
-w	Make the compiler very talkative.
-X*flags*	Send the *flags* to the compiler process.
-Z	Do not prepend a "_" on identifier names. This makes it possible to compile functions for internal use in the runtime system.

4.4.1 Syntax analyser messages

NOT YET WRITTEN!!!

4.4.2 Compiler messages

Numbers of the kind x.y.z refers to sections in the Haskell report.

[1] Varying number of args to *id*
  4.4.2 All lhs for a function binding must have the same number of patterns.
[2] Different function names at *id*
  LML
[3] Bad function: *id*
  Should not happen
[4] Bad identifier in pedantic mode: *id*
  2.3 Haskell restricts the use of the first character in identifiers.
[5] Bad identifiers in pedantic mode: *id*...
  see [4]
[6] Illegal module name: *id*
  5.4 User-defined module names mut not begin with Prelude
[7] Bad deriving for: *id*...
  Appendix D The requested instances could not be derived
[8] Redefinition of imported entity: *id*...
  ? The module redefines something imported.
[9] Circular class structure
  4.3.1 The superclass relation must not be cyclic
[10] Extra definitions in class
  4.3.1 Only defaults for methods may be declared in a class
[11] Type variable not in all methods for class
  4.3.1 All method types must mention the class variable
[12] Constrained variable in class
  4.3.1 No method type must constrain the class variable

[13] Malformed class declaration
Should not happen
[14] Instance declaration contains extra bindings: *id...*
4.3.2 An instance declaration may only bind the class methods
[16] Unknown class in instance declaration
- The instance declaration is invalid because the class id is not a class
[17] Multiple type variables in instance declaration
4.3.2 (Pedantic only) The instance declaration must have at most one type variable
[18] Bad identifiers in pedantic mode: *id...*
see [4]
[19] Circular type synonyms.
4.2.2 Mutually recursive type synonyms not allowed
[20] Unknown id among *id...*
4.4.1 A type signature non-existing binding
[21] Multiple definition in binding: *id...*
- Multiply declared identifiers
[22] Undefined type *id*
- The id is not a type
[23] Bad type arity
- A type constructor was used with the wrong number of arguments
[24] Bad type synonym arity
4.2.2 A type synonym was used with the wrong number of arguments
[25] Undefined class *id*
- Undefined class in context
[26] Not a class *id*
- Non-class in context
[27] Unbound type var(s) *tid*
4.2.1 Data type definition contains type vars that are not in the lhs
[28] Duplicate type var(s) in
4.2.1 Non-linear lhs in a data type definition
[29] Non-classes in deriving: *id...*
- Non-class in deriving
[30] Synonym has bad type var:
4.2.2 Synonym has non-linear lhs or unbound var in rhs
[31] Bad synonym definition:
4.2.2 Rhs in synonym is not a type
[32] Bad type var
LML, conctype
[33] Undefined identifier *id*
- Undefined identifier in expression
[34] Bad conditional pattern (overlap?) near
?
[35] Constructor with too many arguments:
- (Really a type error)
[36] Constructor with too few arguments in pattern:
??? Constructor applications in patterns have to be saturated

[37] Pattern binding does not bind any variables:
- Not really an error, but unlikely to be correct

[38] Bad defaults:
4.3.4 Only numeric classes are allowed as defaults

[39] More than one default
4.3.4 A module must contain at most one default

[40] Bad restriction:
see [83]

[41] Bad synonym:
see [83]

[42] Bad instance:
see [83]

[43] Bad type def:
see [83]

[44] Multiple signatures in
4.4.1 It is illegal to give more than one type signature for one variable.

[45] Id(s) *id*... not in scope for signature
4.4.1 It is illegal to give a type signature for a variable bound in an outer scope.

[46] Identifiers without original names in interface files: *id*...

[47] Module does not import: *id*...

[48] Multiply defined: *id*...

[49] Multiple visible names: *id*...

[50] Not a type *id*

[51] Exported constructors for

[52] Exported methods for

[53] Export id *id* not defined

[54] Ambiguous overloading:

[55] Ambiguous overloading:

[56] Not an instance

[57] Ambiguous instances for

[58] Type restriction context not compatible with deduced:

[59] Bad restriction

[61] occurence

[62] Bad overloading in exports:

[63] unify1 (occurence)

[64] unify2 (occurence)

[65] Cannot unify types:

[66] unify1

[67] unify1

[68] unify arity
Cannot happen(?)

[70] Later pattern completely overlapped

[71] Mismatching constructor type

[72] Variable *id* leftmost in pattern

[73] Duplicate id in pattern

[74] Bad pattern

[75] No instance for superclass

[76] Instance context does not imply class context

[77] Too many superclass instances

[78] Bad type variables in class declaration:

[79] Type or class must be in this module:

[80] Bad type variables in instance declaration:

[81] Type synonym in instance declaration:

[83] context not implied

[84] Unused type var in context:
4.1.2 The context c must only contain type variables referenced in t.

[85] Export id *id* not defined
LML

[86] Bad export specifiers for *id*...

[87] Duplicates in export list *id*...

[88] Module *id* should export *id*...

[89] Bad import specifiers for *id*...

[90] Multiple instance declarations *inst*...

[91] Bad renaming: *id* to *id*...
1.4 The renaming would violate the namespace structure.
[93] Existential type escapes: *type*
- An existential type must not escape the scope of its introducing variable.
[94] Existential types not allowed
LML Show functions can only be made for non-existential types.

4.4.3 Linker messages

The linker may complain about undefined symbols. Usually this means that that you have forgotten to include some files during linking. It is usually possible to figure out the names of the functions by looking at the symbols the linker complains about.

If any of the symbols "_USE_none", "_USE_heap", or "_USE_time" are undefined it means that you have tried to mix files compiled with different profiling options.

An LML/Haskell program automatically decodes a number of flags:

-a	Print extra space after each printed item. Obsolete, use a show function instead.
-C	Produce a "core" file when a signal occurs.
-f	Prints a description of the flags; the program is not executed.
-H*size*	Set maximum heap size. Default is 4000000.
-h*size*	Set minimum heap size. Default is 200000.
-A*size*	Set pointer stack size. Default is 100000.
-V*size*	SPARC only! Set return stack size. Default is 50000.
-T	Enter the runtime tracer. This only works if some of the files were compiled with the "-T" flag. If this flag is used twice a trace is produced without any user interaction.
-gc-gen	Use a generational garbage collector. This garbage collector is an implementation of the garbage collector described in [App89].
-gc-baker	Another garbage collector. Does not work currently.
-gc-slide	Yet another garbage collector.
-u	Unbuffered output; default is buffered. Obsolete, use CUNBUFF instead.
-X	Debug mode (gives some additional messages).

- Marks the end of decoded arguments.

If the runtime system was compiled with dumping enabled there are some additional flags:

-d	Print a stack dump on error.
-G	Print a stack dump before and after each garbage collection.
-K*addr*	When GC reaches stack location *addr* the routine debstop is called.
-M*n*	The maximum number of dumped nodes is set to *n*.
-t*n*	The depth of dump is set to *n*.

If the runtime system was compiled with GC statistics enabled there are some additional flags:

-B	Sound the bell at the start of garbage collection.
-s	Produce a brief "STAT" file, containing garbage collection statistics.
-S	Produce a verbose "STAT" file.

The file "STAT" (when produced) will contain various (selfexplanatory?) statistics.

5.1.1 Heap profiling (Authors: Colin Runciman and David Wakeling)

If the program has been compiled with the heap profiling turned of (the -ph flag to the compiler) it decodes the following flags:

-i*float*	set the sampling interval of the heap profiler to *float* seconds. Normally the profiler runs with an exponentially increasing profiling interval.
-g[{*g*,...}]	Give profiling information by *module group*. When the graph is sampled the space occupied by each node (in bytes) is charged to the particular group of modules that produced the node. In some cases, only certain groups of modules may be of interest, and these groups can be named in an optional restriction set following the -g flag.
-m[{*m*,...}]	Give profiling information by *module*. Similar to the -g flag. In this case though, the space occupied by each node is charged to the module that produced the node. Once again, only certain modules may be of interest, and those can be named in a restriction set.
-p[{*p*,...}]	Give profiling information by *producer*. In this case, the space occupied by each node is charged to the function that produced the node.
-c[{*c*,...}]	Give profiling information by *construction*. As for the -p flag. In this case the space occupied by each node is charged to the construction that it represents, with the function component being used for closures.
-t[{*t*,...}]	Give profiling information by *type*. In this case, the space occupied by each node is charged to the type of the node.

Two or more of the -g, -m, -p, and -c flags may be used together. In this case, the first flag specifies what kind of profile is to be produced, and the remainder are used to specify restrictions.

During reduction the graph is periodically sampled and the samples are written to a file whose name is that of the program, extended with a ".hp" suffix. This file can be converted to a PostScript file by the hp2ps program.

A notable feature is that there is absolutely no concept of scope. Profiles do not distinguish between nested functions with the same name, or between functions in different modules with the same name. The only way to make such distinctions is to copy and rename function bodies. Some names get lost during compilation, and obscure identifiers appear in their place. This happens most often in programs that make heavy use of higher-order functions; my apologies.

Examples:

Running the program "a.out -p" gives a producer profile.

"a.out -p -c{(.)}" gives a producer profile, in which the only producers of interest are those of "(.)" nodes.

"a.out -c -m{lex,parse,typecheck}" gives a construction profile, but only for constructions produced by the modules "lex", "parse" and "typecheck".

"a.out -c -m{lex,parse,typecheck} -p{tokeniser,syntax,tcheck}" gives a construction profile, but only for constructions produced by the modules "lex", "parse" and "typecheck", and only for the functions "tokeniser", "syntax" and "tcheck" within these modules.

5.1.2 Tracing

There is no debugger available that can handle programs produced by lmlc/hbc, but if programs are compiled with the "-T" flag there is a simple interactive tracer that can be used. The tracer is invoked by giving the "-T" flag to a *compiled* program. Unfortunately the tracer cannot be used together with the interactive system (yet).

The tracer has an interactive interface where the user can turn tracing of and off, run till a certain point etc.

The following commands are available:

`help`	Print a help message describing the commands.
`quit`	Quit the tracer and the program.
`leave`	Leave a recursive invokation of the tracer.
`next`	Trace (i.e. print messages) until the next function is entered.
`cont`	Run (i.e. do not print messages) the program to completion.
`rcont`	Trace the program to completion.
`exit`	Trace until the current function exits.
`rexit`	Run until the currentfunction exits.
`stop` *re*	Set breakpoints on all functions matching *re*.
`nostop` *re*	Remove breakpoints from all functions matching *re*.
`arg` *n*	Evaluate (to WHNF) and print argument number *n*.
`farg` *n*	Fully evaluate and print argument number *n*.
`on` *re*	Turn on tracing for functions matching *re*. *Re* may contain "*" which matches any number of characters.
`off` *re*	Turn off tracing for functions matching *re*. *Re* may contain "*" which matches any number of characters.
`where`	Show call stack.
`depth` *n*	Set print depth to *n*. Default value is 3.
`file y/n`	Turn on/off file (module) name printing.

Identifier may be prefixed by their file (module) name followed by a dot to make them unique. An empty command will repeat the previous command. Any kind of error or call to fail will cause the tracer to be entered.

The tracer prints the following messages:

`Enter` *expr*	A function is just about to be entered. The *expr* shows the function with its arguments.
`Return` *expr*	A function is just about to return with an evaluated expression.
`Return variable`	A function is just about to return with variable that might evaluate to a function.
`Jump (unknown)`	A function is just about to tail call another function that was not known at compile time (or had a number of arguments that did not correspond to its arity).
`Jump`	A function is just about to tail call a function known at compile time.

Each message is indented with a depth corresponding to the call stack depth. In the case of a tail called the function which is called can be seen from the following "Enter" message, provided that function has been compiled with the trace flag on. The enter and exit messages always come in pairs, even if tracing is turned off for a particular function between.

The tracer is not able to determine the type for all[2] constructed values. If it cannot then it uses the name CONn for the n:th constructor of a type.

Traced and non-traced modules can be mixed, but only calls to traced code can be observed. Each module in a program can be compiled for tracing, but then the trace flag can be omitted when linking the program. In this case the program will run with a moderate slowdown (it will take about 25% longer). If it is linked for tracing, but run without the "-T" flag it may run as much as 5 times slower.

A problem with understadning the trace messges is that they refer to the program after the extensive transformations performed by the compiler. An aid to understanding is to look at the program after all transformations; given the "-ftransformed" flag the compiler will show this.

5.1.3 Avoiding flag decoding

The automatic flag decoding in LML/Haskell programs can be avoided by linking the program with a small file compiled with the C compiler. This file can contain information about the minimum and maximum heap sizes. An sample file:

```
/*
** Compile this with your C compiler and link with an LML/Haskell program
** to inhibit flag decoding.
** The min and/or max heap size may be set here as well in that case.
*/

int nodecodeflags=1;
int minheap = 1000000;
int maxheap = 8000000;
```

[2]For the moment of almost none of them.

The current strategy for memory allocation is as follows: On startup a heap is allocated, the size of this never changes during execution. The size is twice (there are two heap halves) the heapsize given as argument, or a default of 4 megabytes.

Only part of this memory is used during exection to lower the working set of the program. How large part is determined after each garbage collection. The amount is used (i.e., available for allocation) is the amount that was copied when the collection occured multiplied by eight and 200 kilobytes added to this. In this way the working set is adapted to the amount of heap that is actually in use.

NOT WRITTEN YET (maybe it's impossible?)!!!

Hbc comes with a number of useful extra libraries. To get access to them an `import` of the approprate module is all that is needed.

This a list of the current modules. Any useful contributions to these are welcome!

`Either` binary sum data type

```
data Either a b = Left a | Right b
```

constructor Left typically used for errors, can be renamed to `Wrong` on import.

`Maybe` type for failure or success

```
data Maybe a = Nothing | Just a
thenM :: Maybe a -> (a -> Maybe b) -> Maybe b
```

apply a function that may fail

`Option` an alias for Maybe

```
data Option a = None | Some a
thenO :: Option a -> (a -> Option b) -> Option b
```

`IO` LML style I/O functions

```
openFile :: String -> Either String String
```

returns error message or contents of file

```
getFile :: String -> String
```

returns file contens or fails

```
getEnvi :: String -> Option String
```

maybe value of an environment variable

```
statFile :: String -> Either String [Int]
```

returns error message or (incomprehensible) file information

```
openDirectory :: String -> Either String [String]
```

returns error message or a list of all files in the directory

```
progArgs :: [String]
```

the arguments to the executing program

```
progName :: String
```

name of the executing program

```
data FileMode = ...
data FilePerm = ...
data FileStat = ...
decodeStat :: [Int] -> FileStat
```

decode information from statFile

`ListUtil` Various useful functions involving lists that are missing from the Prelude

```
assoc :: (Eq c) => (a -> b) -> b -> [(c, a)] -> c -> b
```

assoc f d l k looks for k in the association list l, if it is found f is applied to the value, otherwise d is returned

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

flattening map (LMLs concmap)

```
unfoldr :: (a -> (b, a)) -> (a -> Bool) -> a -> [b]
```

unfoldr f p x repeatedly applies f to x until (p x) holds. (f x) should give a list element and a new x

```
mapAccuml :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
```

mapAccuml f s l maps f over l, but also threads the state s through (LMLs mapstate)

```
union :: (Eq a) => [a] -> [a] -> [a]
```

unions of two lists

```
intersection :: (Eq a) => [a] -> [a] -> [a]
```

intersection of two lists

```
chopList :: ([a] -> (b, [a])) -> [a] -> [b]
```

LMLs choplist

```
assocDef :: (Eq a) => [(a, b)] -> b -> a -> b
```

LMLs assocdef

```
lookup :: (Eq a) => [(a, b)] -> a -> Option b
```

lookup l k looks for the key k in the association list l and returns an optional value

`Pretty` John Hughes's pretty printing library.

```
type Context = (Bool, Int, Int, Int)
type IText = Context -> [String]
text :: String -> IText
```

just text

```
(~.) :: IText -> IText -> IText
```

horizontal composition

```
(^.) :: IText -> IText -> IText
```

vertical composition

```
separate :: [IText] -> IText
```

separate by spaces

```
nest :: Int -> IText -> IText
```

indent

```
pretty :: Int -> Int -> IText -> String
```

format it

`QSort` Sort function using quicksort.

```
sortLe :: (a -> a -> Bool) -> [a] -> [a]
```
sort le l sorts l with le as less than predicate

```
sort :: (Ord a) => [a] -> [a]
```
sort l sorts l using the Ord class

`Random` Random numbers.

```
randomInts :: Int -> Int -> [Int]
```
given two seeds gives a list of random Int

```
randomDoubles :: Int -> Int -> [Double]
```
given two seeds gives a list of random Double

`RunDialogue` Test run programs of type Dialogue. Only a few Requests are implemented, unfortunately not ReadChannel.

```
run :: Dialogue -> String
```
just run the program, showing the output

```
runTrace :: Dialogue -> String
```
run the program, showing each Request and Response

`Termcap` Terminal independant screen handling. Programs using this library must be linked with "-ltermcap".

```
noColumns :: Int
```
number of columns on screen

```
noLines :: Int
```
number of lines on screen

```
clear :: String
```
string to clear the screen

```
moveTo :: Int -> Int -> String
```
moveTo x y gives the string to move to coordinate x y

`Trace` Simple tracing.

```
trace :: String -> a -> a
```
trace x y prints x and returns y

`UnsafeDirty` Don't use it!!

`Miranda` Functions found in the Miranda[3] library.

`Printf` C printf style formatting. Handles same types as printf in C, but requires the arguments to be tagged. Useful for formatting of floating point values.

```
data UPrintf = UChar Char | UString String | UInt Int |
        UInteger Integer | UFloat Float | UDouble Double
```

[3] Miranda is a registered trade mark.

```
printf :: String -> [UPrintf] -> String
```
convert arguments in the list according to the formatting string

`Word` Bit manipulation

```
class Bits a where
    bitAnd :: a -> a -> a
```
bitwise and
```
    bitOr :: a -> a -> a
```
bitwise or
```
    bitXor :: a -> a -> a
```
bitwise xor
```
    bitCompl :: a -> a
```
bitwise negation
```
    bitRsh :: a -> Int -> a
```
bitwise right shift
```
    bitLsh :: a -> Int -> a
```
bitwise left shift
```
    bitSwap :: a -> a
```
swap word halves
```
    bit0 :: a
```
word with least significant bit set
```
    bitSize :: a -> Int
```
number of bits in a word
```
data Byte
```
8 bit quantity
```
data Short
```
16 bit quantity
```
data Word
```
32 bit quantity
```
instance Bits Byte, Bits Short, Bits Word
instance Eq Byte, Eq Short, Eq Word
instance Ord Byte, Ord Short, Ord Word
instance Text Byte, Text Short, Text Word
instance Num Byte, Num Short, Num Word
wordToShorts :: Word -> [Short]
```
convert a Word to two Short
```
wordToBytes :: Word -> [Byte]
```
convert a Word to four Byte
```
bytesToString :: [Byte] -> String
```
convert a list of Byte to a String (bit by bit)
```
wordToInt :: Word -> Int
```
convert a Word to Int
```
shortToInt :: Short -> Int
```
convert a Short to Int
```
byteToInt :: Byte -> Int
```
convert a Byte to Int

`Time` Manipulate time values (a Double with seconds since 1970).

```
-- year mon day hour min sec dec-sec weekday
data Time = Time Int Int Int Int Int Int Double Int
dblToTime :: Double -> Time
```
convert a Double to a Time
```
timeToDbl :: Time -> Double
```
convert a Time to a Double
```
timeToString :: Time -> String
```
convert a Time to a readable String

`Hash` Hashing functions.

```
class Hashable a where
    hash :: a -> Int
```
hash a value, return an Int
```
-- instances for all Prelude types
hashToMax :: (Hashable a) => Int -> a -> Int
```
hash into interval [0..x-1]

`IntegerMisc` Various functions on Integer implemented in C.

```
integerPowMod :: Integer -> Integer -> Integer -> Integer
```
integerPowMod x y z == x^y ‘mod‘ z
```
integerToString :: Int -> Integer -> String
```
converts the Integer to a string in the given base
```
integerGcd :: Integer -> Integer -> Integer
```
computes the GCD of two numbers

`NameSupply` Functions to generate unique names (Int).

```
type Name = Int
initialNameSupply :: NameSupply
```
the initial name supply (may be different every time the program is run)
```
splitNameSupply :: NameSupply -> (NameSupply,NameSupply)
```
split the namesupply into two
```
getName :: NameSupply -> Name
```
get the name associated with a name supply

There are a number of known bugs and discrepancies in hbc:

- The number of elements in a tuple is limited, the number of derived instances for is even more limited.

- The number of arguments to functions is limited.
- Non-derived instances of Binary do not work.
- Binary file operations may fail if a garbage collection occurs.
- Type restrictions of variables in pattern bindings are ignored.
- Operator precedences do not follow the operator on renaming.
- Extra parenthesis may be needed around the type in a type restriction on an expression.
- Extra parenthesis may be needed around types in data type definitions with infix constructors.

[App89] Andrew W. Appel. Simple generational garbage collector and fast allocation. *Software-Practice and Experience*, 19(2):171–183, 1989.

[Hud92] Paul Hudak et al. *Report on the Functional Programming Language Haskell*, March 1992. Version 1.2.

[Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, Department of Computer Science, New York University, New York City, USA, 1992.