BABEL User's Guide

1 The BABEL command interpreter

After the installation of the system by executing the Unix file 'create' in a directory xdir together with babel.tar.Z you have to include the following directory in your path:

```
xdir/system/bin
```

Additionally the environment variable 'BABEL' has to be set with the path above (`setenv BABEL xdir/system`). Furthermore you can find some examples in the directory `xdir/system/examples`.

The BABEL system is invoked from unix by the command

`babel` [*file*]

where *file* (an optional parameter) is the pathname of a file containing a BABEL script (i.e. a set of definitions). If no file is given a default filename `script.bab` is assumed. The parameter of the `babel` command (or `script.bab` by default) becomes your current script, during the subsequent BABEL session.

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

The set of names in scope at any time are those of the current script.

The prompt `[file] BABEL >` indicates that you are talking to the BABEL command interpreter and that your current script is `file`. This activity is called a *BABEL session*. Each command should be typed on a single line, following the prompt, and is entered by hitting return. Any command not beginning with one of the special characters `/`, or `!` is assumed to be an expression to be evaluated. The following commands are available during a session:

exp

Any BABEL expression typed on a line by itself is evaluated. The result and the bindings of the Variables in the expression after its evaluation are printed on the screen. If there is more than one solution the system offers the option to compute them.

Example:

```
[dame.bab] BABEL > dame(8,X)
Goal:
Parsing ...
Type checking ...
Code generation ...
Running ...
Result:  true
Bindings:
X/ [4,2,7,3,6,8,5,1]
do you want more solutions?  (y/n) y
Result:  true
Bindings:
X/ [5,2,4,7,3,8,6,1]
do you want more solutions?  (y/n) n

[dame.bab] BABEL >
```

`!` *command*

Execute any UNIX shell command.

`/e`

Edit the current script. Invokes the screen editor `vi`. On leaving the editor, the BABEL system automatically recompiles the current script before returning to the BABEL command interpreter.

`/e` *file*

Edit a script which can be found in the current directory. Note that this time the entire filename has to be given.

`/f` *file*

Change to new current script. Equivalent to quitting the BABEL system and reinvoking it with a new sourcefile. Unlike `/e`, `/f` adds `.bab` to the end of the filename if you omit it.

`/g`

Display the grammar of BABEL.

`/h`

Display summary of all available commands.

`/l`

List all BABEL scripts in the current directory.

`/lazy`

Change the evaluation strategy to lazy. Note that the default strategy is eager and that the file has to be recompiled when the state is changed.

`/eager`

Change the evaluation strategy to eager. This is also the default state.

`/[no]expanding`

Rather than generating abstract machine code for the emulator, the compiler provides an expansion into C-code. With the standard settings, the gcc-compiler is chosen. Usually the emulated version is 2 times slower than the expanded program.

`/q`

Quit the BABEL system.

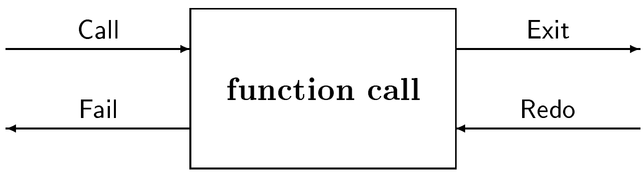
2 Summary of main available commands

exp	evaluate a BABEL expression
`!`*command*	execute any UNIX shell command
`/e`	edit current script (editor=vi)
`/e` *filename*	edit *filename*
`/eager`	change to eager evaluation strategy
`/f` *filename*	change to new current script
`/g`	display the grammar of BABEL
`/h`	display the command summary
`/l`	list the BABEL scripts in the current directory
`/lazy`	change to lazy evaluation strategy
`/[non]strict`	[do not] infer strictness information for lazy strategy
`/[no]expanding`	[do not] generate expanded C-code
`/q`	quit the BABEL system
`/{graph,trail,stack,datastack}size [`*n*`]`	display [change] the size of the data structures

3 The Debugger

3.1 The debugging model

The BABEL debugger is based on Byrd's debugging model. This model has been used as the standard source-level debugger in many Prolog systems. The idea is that during the computation every function call is associated with a box of the following kind:



The box is created whenever a function is called. Hence each box represents the evaluation of a function call and therefore different calls of the same function result in different boxes. For the further understanding it is important to note that a box can contain other boxes. If the right hand side of a rule contains further function calls their associated boxes are created inside the box which represent the application of this rule. You can think of this recursive box structure as a tree with the leftmost innermost function call of the goal as its root (if the evaluation strategy is lazy it is the leftmost outermost function call).

When a function is called a box is created and entered through the Call port. If the function call can sucessfully be evaluated the box is left through the Exit port, otherwise through the Fail port. If it necessary to find an alternative computation because of a failure in a subsequent function call (the associated box is left trough the Fail port) the box is entered again through the Redo port.

The four ports are the only visible points the computation process. At these points the debugger stops and outputs the ports together with the function and its arguments. But there are serveral possibilities to influence the number of stops and their position in the computation.

3.2 The debug modes

The BABEL debugger allows the using of three debug modes which are all well known from imperative programming languages.

1. *Trace mode*
   The trace mode is a single step mode. The debugger steps to the next port which state is ON for the current function. It stops there if the *wait mode* of the port is ON otherwise it just displays the current state of computing and goes on evaluating.

2. *Skip mode*
   In the skip mode the debugger skips the execution of the current function call. It is obviously that this make only sense at a Call or a Redo port because these are the ports *before* the skipped computation. The skip mode allows the user to omit unnecessary details of a subcomputation.

3. *Leap mode*
   The BABEL debugger is able to place up to five conditional breakpoints at a function. In the leap mode the debugger runs the computation without stopping until the next fulfilled breakpoint is reached. Conditional breakpoints allow the user to control his program on a higher level than this is possible with tracing or skipping.

Whenever the debugger stops and shows the current state of computation called *prompt* it is possible to change the debug mode. The debugger starts in the trace mode. So the first point where the debugger stops is the Call port of the box which is associated with the leftmost outermost or leftmost innermost function call (depending on the evaluation strategy) of the goal.

3.3 A simple debugging session

When you are in the BABEL command interpreter you first have to enter `/d` to switch the debug mode of BABEL to ON (you turn the debug mode OFF by simply enter `/d` again).

You start debugging as you did for computation without debugging. Consider the following simple BABEL program:

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

Suppose you want to compute $fib(4)$ (evaluation strategy is eager). Enter this goal and the debugger will answers with the prompt

```
%01 (1              ) call:  fib(4) ?
```

for the leftmost innermost function call of the goal. By simply hitting return you turn the trace mode ON and the debugger goes on running the program until the next port is reached. Pressing return eleven times will produce the following sequence of prompts:

```
%02 (1.1            ) call:    fib(3) ?
%03 (1.1.1          ) call:      fib(2) ?
%04 (1.1.1.1        ) call:        fib(1) ?
%04 (1.1.1.1        ) exit:        fib(1) = 1 ?
%05 (1.1.1.2        ) call:        fib(0) ?
%05 (1.1.1.2        ) exit:        fib(0) = 1 ?
%05 (1.1.1          ) exit:      fib(2) = 2 ?
%06 (1.1.2          ) call:      fib(1) ?
%06 (1.1.2          ) exit:      fib(1) = 1 ?
%06 (1.1            ) exit:      fib(3) = 3 ?
%07 (1.2            ) call:      fib(2) ? s
```

If you subsequently type `s` the computation of $fib(2)$ is skipped and the next prompt the debugger will show is:

```
>09 (1.2            ) exit:     fib(2) = 2 ?
```

Pressing return two times will complete the computation:

```
%09 (1              ) exit:     fib(4) = 5 ?
Result:  5

[fib.bab] BABEL >
```

Suppose you are only interested in the result of the function call *fib*(3) during the computation of *fib*(4). Enter again the goal and at the first prompt the debugger shows you type `fib<(3) & $exit` which places a conditional breakpoint at fib. The condition behind `<` is only fulfilled if the argument of fib is 3 and if the port is `Exit`. Pressing `l` starts the leap mode and the debugger goes on computing until this situation is reached. `n` turns the debugger OFF i.e. the evaluation goes on without stop any more. Here you see the complete debugging session:

```
[fib.bab] BABEL >fib(4)
Compiling ...
Type checking ...
Running ...
Loading debugging session ...
%01 (1              ) call:  fib(4) ? fib<(3) & $exit
===== conditional breakpoint '((3) & $exit)' placed on 'fib'

%01 (1              ) call:  fib(4) ? l
>06 (1.1            ) exit:    fib(3) = 3 ? n
Result:  5

[fib.bab] BABEL >
```

3.4 The prompt

Whenever the debugger is waiting for input is shows the *prompt*. It contains information about the current state of computing and it is always of the following kind:

$$\textit{Debug\_mode Number\_of\_conditions}\ (\textit{Box\_identifier})\ \textit{Port}\text{:}\ \textit{Space}\ (\textit{Argument}_1, \ldots, \textit{Argument}_n)$$

Debug_mode is a character indicating in which debug mode the current state of evaluation was reached. The following table shows all possibilities:

character	debug mode
%	trace mode
>	skip mode
*	leap mode

Number_of_conditions is the number of conditions placed on the current function.

Box_identifier identifies the current box. At the beginning of debugging the debugger shows the long version (LONG) of each identifier. In this case an identifier is a sequence of numbers where each number stands for a box when passing through the box tree from its root to the current box. By entering the command `f` the *identifier mode* is turned to SHORT and *Box_identifier* is shown as an unique number.

Port is the port at which the evaluation stops.

Space are several blanks dependent on the current position in the box tree. They can be suppressed by entering `i`, which turns the *indent mode* ON/OFF.

$Argument_i$ are the arguments of the function call. Note that the arguments can differ for two prompts of the same box because a variable has been bound.

If the evaluation stops at a Exit port the prompt is extended by

= *Result*

where *Result* is the result of the associated function call.

3.5 Conditional Breakpoints

3.5.1 Placing breakpoints

Whenever the prompt is shown you can place breakpoints at functions by entering a command of the form

[*Funcs*] < [*Cond*]

where *Funcs* and *Cond* are optional. *Funcs* is a regular expression defining the functions at which a breakpoint with condition *Cond* should be placed. *Funcs* has the syntax

Funcs	⟶	*Funcs Funcs*	concatenation
	|	*Funcs*, *Funcs*	union
	|	(*Funcs*)	paranthesis
	|	*Funcs*@	any number of repetition
	|	*	any charcter string
	|	?	any single character
	|	*characters*	

where *characters* are the characters a function identifier can consists of in BABEL (a,...,z,A,...,Z,_). *Cond* is described by the grammar

Cond	⟶	([*Term*{ , *Term*}*])	complete list of arguments
	|	(*Cond*)	paranthesis
	|	*Cond* & *Cond*	and operator
	|	*Cond* \/ *Cond*	or operator
	|	~*Cond*	negation
	|	*Term* [~]= *Term*	test on equality
	|	`$call` \| `$redo` | `$exit` \| `$fail`	ports
Term	⟶	*Number*	an single integer
	|	?	*any* term
	|	!	a variable
	|	{ *address* }	code address
	|	*Constructor* [(*Term* { , *Term*}*)]	constructor application
	|	[*Term* | *Term*]	list
	|	[[*Term*{ , *Term*}*]]	list
	|	(*Function* { *Term*}*)	function application
	|	`$calls`	number of function calls
	|	`$arg1` \| `$arg2` | ...	arguments of function call

where *Constructor*, *Function* and *Number* are defined as in BABEL.

The debugger tries to place a breakpoint with condition *Cond* on every function that matches *Funcs*. If this is not possible for a particular function e.g. because *Cond* contains $arg3 although the arity of the function is 2 an error message is printed.

If *Cond* is omitted an unconditional breakpoint is placed on the function(s). This can also be achieved by using a conditional breakpoint of the form

$$\underbrace{\texttt{(?,?,?,...,?)}}_{n \text{ times}}$$

where n is the arity of the function. The default value for *Funcs* is the name of the current function.

3.5.2 Removing breakpoints

Whenever a breakpoint is placed it gets an unique number for the particular function between 1 and 5. The numbers of all conditions are shown by entering the commands `d` or `a`. `d` displays information about the current function whereas `a` displays information about all functions. They are necessary for removing particular conditions which can be done by typing a command of the form

$$[\,\textit{Funcs}\,]\ \texttt{>}\ [\,\textit{Numbers}\,]$$

where *Funcs* and *Numbers* are optional. *Funcs* specifies the functions and *Numbers* the numbers of the conditions. *Numbers* is also an regular expression whose syntax is nearly identical to the one for *Funcs* described above. The only difference of course are the allowed *Characters*. If *Numbers* is omitted the debugger removes all conditions e.g. the default value for *Numbers* is `*`.

3.5.3 The build in functions spy and nospy

BABEL has two build in functions spy and nospy. They offers the possibility to place and remove breakpoints automatically during the evaluation of a goal. spy and nospy can be used in the goal like

$$if\ (spy\ fib\ 2\ a(2))\ then\ fib(4)$$

or in the BABEL program like

$$\begin{array}{lcl} fib(0) & := & if\ (spy\ fib\ 2\ a(2))\ then\ 1. \\ fib(1) & := & if\ (nospy\ fib\ 2)\ then\ 1. \\ fib(X) & := & fib(X-1) + fib(X-2). \end{array}$$

spy:
The first argument specifies the function at which a breakpoint shall be placed and the second argument the unique number of the condition (if there is already a condition with this number the *old* one is deleteted). The number 0 causes the debugger to select a free number by itself. The last argument represents the condition. It consist of any constructor of the same arity as the function which is applied to any terms representing the *Terms* in a condition of the form

$$(\,[\,\textit{Term}\{\,,\textit{Term}\}^*\,]\,)$$

Hence a call of spy of the form

$$\textit{spy func 0 constructor}(\textit{term}_1, \ldots, \textit{term}_n)$$

corresponds to entering the command

$$func\texttt{<(}term_1\texttt{,}\ \ldots\texttt{,}term_n\texttt{)}$$

nospy:
The first argument of nospy defines the function from which a condition shall be removed. The second argument is a number specifying exactly which breakpoint is meant. Hence a call of nospy of the form

$$nospy\ func\ number$$

corresponds to entering the command

$$func > number$$

3.6 Port handling

For each function used in the program the debugger manage the state of the four ports and their wait modes. A box always adopts the behaviour of its associated function. This means e.g. if you turn the Call state of a particular function to OFF the Call port of all boxes associated with this function is ignored.

The wait modes of a particular function and a particular port influences debugging only if the state of this port is ON. In this case a wait mode which is ON causes the debugger to stop computation and to wait for input. If the wait mode is OFF only the prompt is shown and the debugger immediately goes on computing.

You can change the state of ports and wait modes by entering one of the following commands

command	meaning
[*Funcs*] + [*What*]	turn state of *What* to ON
[*Funcs*] - [*What*]	turn state of *What* to OFF
[*Funcs*] = [*What*]	set the state of *What* to ON and everyone else to OFF

The meaning of *Funcs* is like mentioned above. *What* is a regular expression describing the ports and wait modes which shall be altered. The syntax of *What* is similar to the one shown for *Funcs* except the *Characters*. The shortened form for ports and wait modes which are used in *What* are:

character	meaning
`c`	Call port
`r`	Redo port
`e`	Exit port
`f`	Fail port
`wc`	wait mode of Call port
`wr`	wait mode of Redo port
`we`	wait mode of Exit port
`wf`	wait mode of Fail port

Therefore allowed characters are `c`,`r`,`e`,`f`,`w` and examples for changing ports and wait modes are

`f*, map+wc,we,c,r` and `fib-w*`

If *What* is omitted the debugger uses the default value `c,r,e,f` for *What*.

At the beginning of a debugging session the state of ports and wait modes of all functions are initiated with ON. An execption are the two build in functions spy and nospy. Their ports are initiated with OFF.

3.7 Lazy evaluation

If the evaluation strategy is lazy the computation of an argument of a function call is suspended until it is really needed. Such an argument is represented by the address where the code for computing it starts. The debugger shows this kind of arguments by enclosing the code address with brackets, e.g. {321}.

When an argument must be evaluated a special function is called named `EvalArg`. Neither it is a build in function of BABEL nor it is a user defined function. Instead the debugger *causes* the evaluation of an suspended argument as to be a function call of `EvalArg` with the corresponding code address as the only argument. Thus it is possible to treat the computation of arguments as invocations i.e. it possible to place breakpoints and to change the state of ports and wait modes.

3.8 Commands

The BABEL debugger offers the following commands

`a`

Display the state of ports and wait modes for all functions.

`b`

The debugger starts backtracking.

`c`

Show the path from the root of the box tree to the current box by displaying box identifiers and their associated function calls.

`d`

Display the state of the ports and wait modes of the current function.

`f`

Change the identifier mode to LONG/SHORT.

`h` or `?`

Prints the file debug.hlp on the screen. This file contains information about all available commands.

`i`

Turn the indent mode ON/OFF.

`l`

Turn the leap mode ON e.g. the debugger runs the program without stopping until a fulfilled breakpoint is reached.

`n`

Close the debug session and go on computing the goal.

`q`

Close the debug session and stop computing the goal.

`r`

Transfers control back to the Call port of the current box. This command is partly available at Redo and Exit ports.

`s`

Turns the skip mode ON e.g the debugger skips the execution of the current function call. This command is only available at a Call or Redo port.

`t` or `RETURN`

Turns the trace mode ON e.g. the debugger steps to next port which is ON.

[*Funcs*] < [*Cond*]

Place a conditional breakpoint at all functions which matches *Funcs*.

[*Funcs*] > [*Numbers*]

Remove the conditions those numbers matches *Numbers* from the functions which matches *Funcs*.

[*Funcs*] + [*What*]

Turn ON the ports and wait modes which matches *What* of the functions which matches *Funcs*.

[*Funcs*] - [*What*]

Turn OFF the ports and wait modes which matches *What* of the functions which matches *Funcs* OFF.

[*Funcs*] = [*What*]

Turn the ports and wait modes which matches *What* ON and all others else OFF.

4 The Module System

BABEL offers the programmer a module system. Each BABEL file can be considered as module providing other BABEL files objects for use. These objects can be functions, predicates, constructors or datatypes. The providing file is the exporting file and the using one the importing file.

For the purpose of a defined interface moduls must declare which of its objects are exported and therefore can be used by other moduls. On the other hand a module which wants to use objects of another module must explicitely import those objects from there.

4.1 Qualified Identifiers

Within a module all objects are represented unequivocally by their identifiers. In some cases it may happen that the nonambiguity of identifiers is lost through the module system. A module can import two objects with the same name from two different moduls. Another possibility for losing nonambiguity is to import an object which has the same name like a local defined object. These objects may of course have different meanings, hence they must be distinguished.

Two objects with the same name *object_name* can be distiguished in BABEL by *qualified identifiers* which consists of the object name and the object source:

$$object\_source \,.\, object\_name$$

The object source is the module (Unix filename without extension ".bab" and with optional path) where the desired object is defined or from where it is imported.

Suppose a module `std` defines the function `length`. `length` is imported by a second module `test` which additionally defines a datatype `length`. So there are two objects with name `length` in `test`. To use the *datatype* `length` in a type specification you have to type `test.length`. For example

```
fun get_length:  [A] -> test.length.
```

If you want to use the *function* `length` in a function call you have to write `std.length`. For example

```
f := std.length [4,3,2,1].
```

If *object_source* is omitted and a local defined object with name *object_name* exists this one is accessed. Otherwise (no local defined object) the first imported object with name *object_name* is accessed. If there is no known object with this name in the module *object_name* is treated as a constructor.

Hence it is sufficient in the example above to write

```
fun get_length:  [A] -> length.
```

There are some places where qualified identifiers doesn't make sense and therefore where they are forbidden:

- in left hand sides of function or predicate definitions
- in left hand sides of type specifications
- in import declarations

4.2 Exporting Objects

A module which wants to provide some of its objects to other moduls has to declare them as to be exported. This is achieved by *export declarations* of the form

$$\texttt{\%export}\ object_1\,,object_2\,,\ldots,object_n\,.$$

`%export` is the keyword of an export declaration. The $object_i$ are the names of the exported functions, predicates, constructors or datatypes.

Export declarations can be given everywhere in a BABEL program. Each one is treated like an export declaration at the end of the BABEL program i.e. it is not necessary to define a function before exporting it.

The exported objects can be local defined objects or they are imported from other moduls. So it is allowed to use qualified identifiers for $object_i$ to distiguish between objects with the same name but different sources.

The *same* object can occur in two or more export declarations. But is is not allowed to export two objects with the same name but different sources. For example consider the program

```
%import length from "A".
%import length from "B".

length [] := 0.
length [H|T] := 1 + (length T).
```

and the export declarations

```
%export A.length, A.length.   /* allowed */
%export length, A.length.     /* not allowed */
%export length.               /* allowed */
%export A.length, B.length.   /* not allowed */
```

4.3 Importing Objects

A module which wants to use objects defined in other moduls must import them by *import declarations* of the form

%import $object_1, object_2, \ldots, object_n$ from "*module*".

`%import` is the keyword of an import declaration. The $object_i$ are the names of the imported objects and *module* is the name (complete Unix filename) of the module from where the objects should be imported.

Like export declarations import declarations can be made everywhere in a BABEL program. Each one is treated like an import declaration at the beginning of the BABEL program. Thus it is possible to use an object which will be defined later in the program by an import declaration or a local definition.

It is only possible to import objects from *module* which are declared as exported in *module*. Importing objects with the same name from two or more different moduls is supported. Subsequently, they are distinguished by qualified identifiers.

4.4 Including Files

A possibility to integrate BABEL source into a BABEL program is to use an *include statement* of the form

%include "*filename*".

An include statement in a BABEL program is treated by the compiler as if the source found in *filename* stands instead of the statement in the BABEL program. A loop generated by include statements is automatically detected by the compiler.