Memorial
University of Newfoundland

Technical Report #2000–03

COMPILING AND USING THE hugs INTERPRETER

by

D.C. Craig

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

December 2000

Technical Report #2000–03

COMPILING AND USING THE hugs INTERPRETER

by

D.C. Craig

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

December 2000

Abstract

This paper presents an overview on how to compile and use the `hugs` implementation of the Haskell interpreter on the Computer Science machines at MUN. Instructions are also provided on how to load and interpret user-defined scripts. Three sample scripts are also provided which demonstrate some of the capabilities of the language. These examples also highlight some of the differences between Haskell and Miranda. Note that this technical report makes references to Technical Report 2000–02. Therefore having access to both reports will be helpful.

1 The `hugs` Interpreter

This section provides information regarding compiling and using the `hugs` interpreter. These instructions apply only to the February 2000 release of `hugs`, and may or may not be applicable to subsequent releases.

1.1 Supported Platforms

`hugs` was compiled for two different operating systems and architectures, namely Digital Unix V3.2G (Rev. 62) running on a DEC-Alpha processor (`garfield`) and Linux version 2.2.14 running on an Intel Pentium II processor (`liz`). The latter machine was chosen so that informal benchmarks comparing the relative performance of `hugs` with Miranda could be performed (see Technical Report 2000–02).

The `hugs` interpreter used for this report was obtained from the URL

`http://www.haskell.org/hugs/`

which contains links to frequently asked questions, documentation and a downloading page for the interpreter. Information on configuring and installing the program are presented below.

Although `hugs` was compiled for only two different platforms as part of this research, there should be no difficulty building it for other Unix operating systems should the need arise. Indeed, `hugs` has also been ported to other non-Unix operating systems as well, such as Microsoft Windows. However, these ports have not been evaluated as part of this research.

1.2 Compiling the Interpreter

The `hugs` distribution comes with a `configure` script which makes compiling and installing the interpreter very easy on a variety of Unix-like platforms. For example, to compile and install `hugs` in your home directory, give the following commands (this assumes that you have successfully downloaded and unarchived the distribution):

```
cd hugs98/src/unix
./configure --prefix=$HOME
cd ..
make install
```

The `configure` script identifies the peculiarities of the architecture and operating system upon which it is run. It generates a `Makefile` which is tuned towards your specific platform. The `--prefix` option specifies the path where the final binary and related libraries are to be installed. By default, the interpreter and the libraries will be installed in appropriate subdirectories of `/usr/local`. For this report, `hugs` was installed in `/users/cs/grad/donald/pub/haskell`. Another useful configure option is `--enable-timer` which causes the interpreter to display timings as functions are evaluated. More options regarding the configuration can be generated by typing `./configure --help`.

For more details related to the compilation and installation of the interpreter, consult the `Install` file which comes with the distribution.

1.3 Using the Interpreter

To run `hugs` on either `garfield` or `liz`, simply give the command:

```
~donald/pub/haskell/bin/hugs
```

This is a custom script (it's not actually included with the `hugs` distribution) which will run the appropriate binary for the platform on which it is run.[1] Upon running this script, the following banner and the `hugs` prompt will be displayed as follows:

```
--   -- --  --  ----   ---      -----------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-1999
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: February 2000  _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/users/cs/grad/donald/pub/haskell/share/hugs/lib/Prelude.hs":

Hugs session for:
/users/cs/grad/donald/pub/haskell/share/hugs/lib/Prelude.hs
Type :? for help
Prelude>
```

Note that the banner uses escape sequences to display its startup progress as it parses and processing the `Prelude` file. When using the `script` program to capture the output of a `hugs` session, these escape sequences can result in a lot of garbage being sent to the `script` output file. This can be mitigated by specifying the `+q` option to the hugs interpreter.

```
~donald/pub/haskell/bin/hugs +q
```

This option "turns on" (by virtue of the `+`) quiet mode which prevents the interpreter from explicitly showing progress during its processing of scripts.

At this point, the interpreter may be used in much the same was as Miranda. For example:

```
Prelude> 1 + 1
2
Prelude> map (*2) [1..5]
[2,4,6,8,10]
Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]
```

[1] Naturally, the system administrators could install `hugs` in a more appropriate place so that the full pathname of the interpreter would not have to be specified to actually run it.

Various commands can be given to the `hugs` interpreter using a colon (`:`) as the first character in the command, as demonstrated by the `:type` command above. The colon used in this case serves a similar purpose as the slash character (`/`) in Miranda. For example, to quit the Miranda's interpreter, one would type `/quit`; to quit the `hugs` interpreter, one would type `:quit`.[2] Other commands are described in the next section.

1.4 Creating, Loading and Editing Scripts

Like Miranda, Haskell script files are typically written using an separate ASCII text editor then loaded by the interpreter for parsing, type-checking and execution. After writing a script file, the `hugs` interpreter can be started using the filename as a command line argument and the `hugs` interpreter will automatically load and compile the file, checking for type consistency.

Note that by convention, Haskell scripts end in `.hs`, but any extension (or none at all) is acceptable. Also note that unlike Miranda, no intermediate files are produced by the interpreter. Therefore, for example, `hugs` does not create `.x` files like Miranda does.

While in `hugs` one may use the `:load` *filename* command to tell the Haskell interpreter to process the specified filename. This has the same effect as specifying the filename on the command line and is analogous to the `/file` *filename* command in the Miranda interpreter.

Using the `:edit` command will bring up the file in the editor. The editor can be configured by setting the `EDITOR` environment variable. For example, under `ksh`, this can be done as follows:

```
export EDITOR="/bin/vi +%d %s"
```

Doing this will result in the `hugs` interpreter using the `vi` editor. In the event of errors in the script, the `:edit` command will bring up the file with the cursor on the line that contained the error. Note that other programs may also use the `EDITOR` environment variable, but may require only the actual editor executable be defined and not the actual format of the command line. If this is the case, then the `EDITOR` environment variable will have to be reset for other applications. Alternatively, the editor can be set while in the `hugs` interpreter by giving the following command:

```
Prelude> :set -E"/bin/vi +%d %s"
```

Haskell, unlike Miranda, does not have on online reference or manual built into the interpreter. With Miranda, one could give the command `/man` and get a fairly comprehensive reference manual describing many of the available features. Unfortunately, the `hugs` interpreter does not support such a comprehensive manual. Instead, the interpreter provides the `:?` command which only gives a brief overview of all the commands that are supported by the interpreter:

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
```

[2] Ctrl-D can also be used in both interpreters to terminate.

```
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names defined in <modules>
:find <name>        edit module containing definition of name
:!command           shell escape
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Prelude>
```

The hugs interpreter supports a variety of less commonly used options which can be used to configure the interpreter or to fine tune its performance, as can be seen by the output below. Note that many of these options can be set using command line arguments when starting the hugs interpreter.

```
Prelude> :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s    Print no. reductions/cells after eval
t    Print type after evaluation
f    Terminate evaluation on first error
g    Print no. cells recovered after gc
l    Literate modules as default
e    Warn about errors in literate modules
.    Print dots to show progress
q    Print nothing to show progress
w    Always show which modules are loaded
k    Show kind errors in full
u    Use "show" to display results
i    Chase imports while loading modules

OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Hugs)
pstr Set prompt string to str
rstr Set repeat last expression string to str
Pstr Set search path for modules to str
Estr Use editor setting given by str
cnum Set constraint cutoff limit
Fstr Set preprocessor filter to str

Current settings: +fewui -stgl.qk -h250000 -p"%s> " -r$$ -c40
Search path     : -P{Hugs}/lib:{Hugs}/lib/hugs:{Hugs}/lib/exts
```

```
Editor setting  : -E"vi +%d %s"
Preprocessor    : -F
Compatibility   : Haskell 98 (+98)
Prelude>
```

1.5 Notes on the Microsoft Windows Implementation of `hugs`

Should the Computer Science department wish to experiment with Haskell for teaching functional programming courses, many students may try to run `hugs` at home on Microsoft Windows machines. The port of Haskell to the Windows environment was not evaluated for this report and may consequently have nuances of its own which might be inconsistent with the Unix environment. The fragmented flavours of Microsoft Windows (for example, Windows 95, Windows 98, Windows ME, Windows NT and Windows 2000, not to mention the various "Second Editions" of some of these products) make it very difficult to evaluate the interpreter on all possible Microsoft environments. However, it is unlikely that the implementation of `hugs` on Microsoft Windows would employ API calls that are not available under all the different generations of the operating systems.

There is a document that comes with the `hugs` distribution that may be helpful to students wanting to run the interpreter on their home machines using Windows. This document can be found in the file:

```
hugs98/docs/windows-notes.txt
```

of the distribution. Students who wish to run it should probably be informed that they are on their own if they do and if they have problems that they can't resolve, then they should rely on the interpreter installed on the Computer Science machines.

2 Some Examples

This section shows the solutions to three Miranda assignments converted to Haskell. The assignments were given during the Computer Science 4719 course offered during the Intersession 2000 semester. The first assignment was to define addition, subtraction and multiplication functions for so-called `vnums` which are a tuple-string representation of a signed number (for example, `('-', "123")` represents the number –123). The second assignment used algebraic types to represent boolean expressions. Various functions were then defined to determine whether boolean expressions were tautologies and to convert boolean expressions to sum-of-products and to product-of-sum forms. Finally, the third assignment provides a flexible means of generating symbolic sequences.

2.1 Example #1

The script below is the official solution to Assignment #1 converted to Haskell.[3] The changes between the Miranda and Haskell solutions are highlighted with a grey background.

```
type Vnum = (Char, [Char])

sign (sx, xs) = sx
```

[3]Actually, a few minor changes were made. The original solution had a problem with the `addn` function passing strings of unequal length to the `addabs` function. This problem has been corrected in the solution below by introducing the function `pad_nums` and having `addabs` (and `subabs`) do the zero-padding of its arguments.

```
digits (sx, xs) = xs

numof ch = (ord ch) - (ord '0')
charof n = chr ((ord '0') + n)

repp 0 x = []
repp (n+1) x = x: repp n x

remzeros []       = []
remzeros ('0':xs) = remzeros xs
remzeros xs       = xs

extend xs n | n <= (length xs) = xs
            | n >  (length xs) = (repp (n - (length xs)) '0') ++ xs

negate' ('+',xs) = ('-',xs)
negate' ('-',xs) = ('+',xs)

sub xnum ynum = add xnum (negate' ynum)

add (sx, xs) (sy, ys) = add1 (sx, xs) (sy, ys)

add1 ('+', xs) ('+', ys) = ('+', (addabs xs ys))
add1 ('-', xs) ('-', ys) = ('-', (addabs xs ys))
add1 ('-', xs) ('+', ys) = add1 ('+', ys) ('-', xs)
add1 ('+', xs) ('-', ys)
                | equal xs ys   = ('+', ['0'])
                | greater xs ys = ('+', (subabs xs ys))
                | otherwise     = ('-', (subabs ys xs))

equal xs ys = ( (remzeros xs) == (remzeros ys) )

pad_nums xs ys = (nxs, nys)
                 where
                 n = max (length xs) (length ys)
                 nxs = extend xs n
                 nys = extend ys n

greater xs ys = nxs > nys
                where (nxs, nys) = pad_nums xs ys

addabs xs ys
        = remzeros (reverse (revadd (reverse nxs) (reverse nys) [] '0'))
                where (nxs, nys) = pad_nums xs ys

revadd [] [] sofar c = sofar ++ [c]
```

```
revadd (rx:rxs) (ry:rys) sofar c
           = revadd rxs rys (sofar ++ [newdig]) newc
             where
             newdig = charof (((numof rx) + (numof ry) + (numof c)) `mod` 10)
             newc = charof (((numof rx) + (numof ry) + (numof c)) `div` 10)

subabs xs ys
        = remzeros (reverse (revsub (reverse nxs) (reverse nys) [] '0'))
                where (nxs, nys) = pad_nums xs ys

revsub [] [] sofar c = sofar
revsub (rx:rxs) (ry:rys) sofar c
        =  revsub rxs rys (sofar ++ [newdig] ) newc
           where
           newdig = charof (subdigfun (numof rx) (numof ry) (numof c))
           newc = charof(subcarryfun (numof rx) (numof ry) (numof c))

subdigfun nrx nry nc | (nry + nc) <= nrx = nrx - (nry + nc)
                     | otherwise         = (10 + nrx) - (nry + nc)
subcarryfun nrx nry nc | (nry + nc) <= nrx = 0
                       | otherwise         = 1

mult_sign '+' '+' = '+'
mult_sign '+' '-' = '-'
mult_sign '-' '+' = '-'
mult_sign '-' '-' = '+'

addn 0 absnum = "0"
addn 1 absnum = absnum
addn (n + 1) absnum = addabs absnum (addn n absnum)

multabs mplier mcand
        = foldr mult_add "0" mlist
          where
          mlist = [addn (numof dig) mplier ++ zeros |
                             (dig, zeros) <- zip (reverse mcand) zlist]
          zlist = [repp k '0' | k <- [ 0.. ]]

mult_add xdigs ydigs
        =  addabs xdigs ydigs

mult vnum1 vnum2
        = (mult_sign (sign vnum1) (sign vnum2),
               (multabs (digits vnum1) (digits vnum2)))

test1 = add ('+',  "9")   ('+', "94")
```

```
97:  test2 = add ('+',  "")    ('-', "42")
98:  test3 = add ('-',  "13")  ('-', "78")
99:  test4 = sub ('+',  "682") ('+', "47")
100: test5 = sub ('-',  "17")  ('+', "7")
101: test6 = mult ('+', "11")  ('+', "50")
102: test7 = mult ('-', "6")   ('-', "13")
103: test8 = mult ('+', "")    ('+', "10")
```

Looking over the solution, it can be seen that the Haskell version is very similar to the Miranda version with only a few modest changes — most of which are syntactic in nature. Line 1 replaces the original `vnum == (char, [char])` type synonym with the Haskell equivalent `type Vnum = (Char, [Char])`. Note that the name of the synonym (and the `Char` types) are capitalized. Lines 6 and 7 replace the `decode` and `code` functions with `chr` and `ord`, respectively, to convert between integers and ASCII. Lines 16 and 17, 30–32 and 68–72 have been changed to use the Haskell guard syntax discussed in Technical Report 2000–02. In lines 19, 20 and 22, the name of the `negate` function was changed to `negate'` because `negate` is a function name already defined in the prelude. (Haskell's `negate` function simply returns the negative of a number.)

Line 34 changes the relational equality operator from `=` to `==`, as required by Haskell. Forgetting to use the correct relational equality operator is a common source of problems when writing Haskell scripts for the first time. In line 38, we note that Miranda's `#` operator, which determines the length of a list, has been replaced with the `length` function call in Haskell. Also note that Miranda `max2` function has been replaced with Haskell's `max` function. Both of these functions determine the maximum value of two supplied values. As an aside, in Miranda, the function `max` determines the maximum value in a list of values; in Haskell, this is done by the function `maximum`. Miranda's `max2` and Haskell's `max` function are defined similarly as are Miranda's `max` and Haskell's `maximum` function. To help clarify the differences between these function, see Table 1. There are also corresponding minimum-related functions.

Table 1: Comparisons of the *maximum*-related functions

Miranda function and type	Haskell Function and type	Purpose
`max2 :: *->*->*`	`max :: Ord a => a -> a -> a`	Returns the maximum of two values
`max :: [*]->*`	`maximum :: Ord a => [a] -> a`	Returns the maximum value in a list of values

Lines 54 and 55 employ the infix versions of the `mod` and `div` operators in Haskell. As described in Technical Report 2000–02, they are enclosed in backquotes. Finally, line 86 demonstrates a subtle, but important difference between the `zip` functions between Miranda and Haskell. In Miranda, the `zip` function is basically an uncurried form of its `zip2` function. Haskell, however, does not have an uncurried `zip` function; its `zip` function is curried and is essentially equivalent to Miranda's curried `zip2` function. Miranda's `zip2` and Haskell's `zip` have essentially the same type:

Miranda: `zip2 :: [*]->[**]->[(*,**)]`
Haskell: `zip :: [a] -> [b] -> [(a,b)]`

In the Miranda solution to the assignment, the uncurried application of the `zip` function: `zip (reverse mcand, zlist)` has been replaced with the (curried) `zip` function application in Haskell: `zip (reverse mcand) zlist`.

2.2 Example #2

The second solution presented below to Assignment #4, from the Intersession Semester of 2000 demonstrates the similarity between Haskell and Miranda when dealing with algebraic types. While the similarities are evident, there are some syntactic and semantic differences which are demonstrated by this example. Again, the differences are highlighted in the script below. (The comment delimiters, which are `--`, in Haskell, instead of `||`, have not been highlighted.)

```
-- **************************************************
-- Algebraic type for boolean expressions
-- **************************************************
data Bexp = T | F | Var Char |
            Andd Bexp Bexp |
            Orr Bexp Bexp |
            Implies Bexp Bexp |
            Iff Bexp Bexp |
            Nott Bexp deriving Eq

-- ********************************************
-- The function vars will extract the
-- names of variables from a boolean
-- expression
-- ********************************************
vars T = []
vars F = []
vars (Var x) = [x]
vars (Andd be1 be2) = (vars be1) ++ (vars be2)
vars (Orr be1 be2) = (vars be1) ++ (vars be2)
vars (Implies be1 be2) = (vars be1) ++ (vars be2)
vars (Iff be1 be2) = (vars be1) ++ (vars be2)
vars (Nott be) = vars be

-- ********************************************
-- The function envs will produce a
-- list of all possible environments
-- for some given list of variables
-- ********************************************
envs [] = [[]]
envs (x: xs) = [ (x,T): env | env <- (envs xs) ] ++
               [ (x,F): env | env <- (envs xs) ]

-- *********************************************
-- The function valof produces the value
-- associated with the variable x in some
-- given environment
-- *********************************************
valof x ((y, yval): env) | x == y    = yval
                         | otherwise = valof x env
```

```
-- ***********************************************
-- The function (eval e env) evaluates the
-- expression e in the environment env.
-- This will involve recursively evaluating
-- sub_expressions of e in the same environment.
-- ***********************************************
eval T env = T
eval F env = F
eval (Var x) env = valof x env
eval (Andd be1 be2) env = andfun (eval be1 env) (eval be2 env)
eval (Orr be1 be2) env = orfun (eval be1 env) (eval be2 env)
eval (Implies be1 be2) env = impliesfun (eval be1 env) (eval be2 env)
eval (Iff be1 be2) env = ifffun (eval be1 env) (eval be2 env)
eval (Nott be) env = nottfun (eval be env)
andfun x y = bfun ((x == T) && (y == T))
orfun x y = bfun ((x == T) || (y == T))
impliesfun T y = bfun (y == T)
impliesfun F y = T
ifffun x y = bfun (x == y)
nottfun T = F
nottfun F = T
bfun True = T
bfun False = F

-- **********************************************
-- The function (is_tautology be) tests whether
-- or not the boolean expression "be" is
-- a tautology.
-- **********************************************
is_tautology be
        = and [ (eval be env) == T | env <- (envs (vars be)) ]

-- ***********************************************
-- Expressions for test cases
-- ***********************************************
be1 = Implies (Andd (Var 'x') (Var 'y')) (Var 'x')
be2 = Implies (Var 'x') (Var 'y')
be3 = Implies (Var 'x') (Orr (Var 'x') (Var 'y'))
be4 = (Orr (Var 'x') (Nott (Var 'x')))
be5 = (Andd (Var 'x') (Nott (Var 'x')))
be6 = Nott (Andd (Orr (Var 'x') (Var 'y')) (Var 'z'))
be7 = (Orr (Andd (Var 'x') (Var 'y')) (Var 'z'))
be8 = (Andd (Orr (Var 'w') (Var 'x')) (Orr (Var 'y') (Var 'z')))
be9 = (Nott (Orr (Andd (Var 'x') (Var 'y')) (Var 'z')))
envlist = envs "xy"
```

```
-- *************************************************
-- The function (push_nots be) pushes down the not's
-- in the boolean expression be
-- *************************************************
push_nots (Andd be1 be2)
        = (Andd (push_nots be1) (push_nots be2))
push_nots (Orr be1 be2)
        = (Orr (push_nots be1) (push_nots be2))
push_nots (Nott (Nott be))
        = push_nots be
push_nots (Nott (Andd be1 be2))
        = (Orr (push_nots (Nott be1))
               (push_nots (Nott be2)))
push_nots (Nott (Orr be1 be2))
        = (Andd (push_nots (Nott be1))
                (push_nots (Nott be2)))
push_nots be = be

-- ************************************************
-- The function disty implements the distributivity
-- rules needed to obtain the pos_form of a
-- boolean expression
-- ************************************************
disty (Andd be1 be2)
        = (Andd (disty be1) (disty be2))
disty (Orr be1 (Andd be2 be3))
        = (Andd (disty (Orr be1 be2))
                (disty (Orr be1 be3)))
disty (Orr (Andd be1 be2) be3)
        = (Andd (disty (Orr be1 be3))
                (disty (Orr be2 be3)))
disty be = be

-- *****************************************************
-- The function disty1 implements the distributivity
-- rules needed to obtain the sop form of a
-- boolean expression
-- ****************************************************
disty1 (Orr be1 be2)
        = (Orr (disty1 be1) (disty1 be2))
disty1 (Andd be1 (Orr be2 be3))
        = (Orr (disty1 (Andd be1 be2))
               (disty1 (Andd be1 be3)))
disty1 (Andd (Orr be1 be2) be3)
        = (Orr (disty1 (Andd be1 be3))
               (disty1 (Andd be2 be3)))
disty1 be = be
```

```
136: sop_form be = disty1 (push_nots be)
137: pos_form be = disty (push_nots be)
138:
139: -- ***************************************
140: -- (show_bexp be) converts "be" into a
141: -- list of characters, suitable for printing
142: -- ****************************************
143: show_bexp (Var x) = [x]
144: show_bexp (Andd be1 be2) = "(" ++ (show_bexp be1) ++ " /\\ " ++
145:                                  (show_bexp be2) ++ ")"
146: show_bexp (Orr be1 be2) = "(" ++ (show_bexp be1) ++ " \\/ " ++
147:                                 (show_bexp be2) ++ ")"
148: show_bexp (Implies be1 be2) = "(" ++ (show_bexp be1) ++ " -> " ++
149:                                     (show_bexp be2) ++ ")"
150: show_bexp (Iff be1 be2) = "(" ++ (show_bexp be1) ++ " <-> " ++
151:                                 (show_bexp be2) ++ ")"
152: show_bexp (Nott (Var x)) = "~" ++ [x]
153: show_bexp (Nott be) = "~(" ++ (show_bexp be) ++ ")"
154:
155: test_exp be = (show_bexp be) ++ "\n" ++
156:               (show (is_tautology be)) ++ "\n\n"
157:
158: taut_tests bes = putStr ( concat (map test_exp bes) )
159:
160: pos_test be = " boolean expression " ++ (show_bexp be) ++ "\n" ++
161:               " pos_form " ++ (show_bexp (pos_form be)) ++ "\n\n"
162: pos_tests bes = putStr ( concat (map pos_test bes) )
163:
164: sop_test be = " boolean expression " ++ (show_bexp be) ++ "\n" ++
165:               " sop_form " ++ (show_bexp (sop_form be)) ++ "\n\n"
166: sop_tests bes = putStr ( concat (map sop_test bes) )
```

As with type synonyms from the previous example, the algebraic type for the boolean expression is defined differently. Instead of using the syntax:

```
bexp ::= T | F | Var char | ...
```

the type is defined as follows:

```
data Bexp = T | F | Var Char | ...
```

Again, note that as with type synonyms, the name of the algebraic type must be capitalized `Bexp` and this capitalization, of course, is preserved throughout the definition of the (recursive) algebraic type. One crucial difference between the two algebraic type definitions is the addition of the phrase `deriving Eq` at the end of the Haskell definition. This is necessary in order to make the boolean expression algebraic type a member of Haskell's equality (`Eq`) type class. This is necessary because the script subsequently performs equality tests on boolean expressions (see, for example, lines 56–60). Without making the boolean expression type an instance of the `Eq` class, `hugs` generates the following error, whenever some function, `<func>`, attempts to compare two boolean expressions for equality:

```
Instance of Eq Bexp required for definition of <func>
```

If we were creating a algebraic type which could be ordered (and subsequently compared using `<` and `>`, for example) then we would also have to make the algebraic type an instance of the `Ord` type class. This could be done by appending `deriving (Eq, Ord)` at the end of the algebraic type declaration. As an aside, this is very similar to how the `Bool` type is defined in the prelude file.

Lines 39 and 40 convert the Miranda syntax for guards into the appropriate guard syntax for Haskell and lines 56–58, 60 and 72 use the appropriate Haskell equality operator `==`. Also note that the logical-`AND` and logical-`OR` operators have been changed to conform with Haskell's syntax, on lines 56 and 57.

The only remaining differences occur with string handling. Note that on lines 158, 162 and 166, the function `putStr` is applied to the results of the `concat` functions. (The `concat` functions, in this context, produce strings). The `putStr` function causes all the escape sequences within the strings to be displayed correctly. For example, `\\` will output a single `\` and `\n` will display a newline, as expected. Unlike Miranda, the `hugs` implementation of Haskell does not tolerate invalid escape sequences. For example, `\/` will produce an error, because no such escape sequence exists. The `/` character can simply be output as is. On lines 144 and 146 of the original solution to the assignment, the `/` character was escaped with a backslash. This results in an error when `hugs` attempts to process it. Miranda is more forgiving of invalid escape sequences.

2.3 Example #3

The final example listed below displays symbolic sequences, for example, `x^1/2 + x^2/4 + x^3/8 + x^4/16 + x^5/32`. As with the previous example, the script makes liberal use of algebraic types to achieve this. Again, note that we change the syntax for the `Expn` algebraic type to conform with Haskell. Apart from the syntactic differences, there are two important type differences between the definitions of the algebraic types. The original Miranda algebraic type defined `Num` as a constructor. However, because Haskell uses `Num` as a type class, the type constructor used by the Haskell script (on Line 5) must use a different name. The constructor `Number` was chosen instead. This change is reflected in the rest of the script too.

Secondly, when defining the `Const` constructor in Miranda, the `num` type was used as a parameter. At first, it may appear that we can use Haskell's `Num` as an equivalent to Miranda's `num`. However, this is not the case because `Num` in Haskell is a type class. Instead, a type instance is required for the constructor parameter. Therefore, we have chosen to use the `Integer` type instance (which is an instance of the `Num` type class) on Line 12 of the listing. Note that this change requires that the eval function on Line 50 use the integer division (`` `div` ``) operator instead of the Float/Double division operator (`/`). Had the original Float/Double division operator been left in place, then Haskell would generate the confusing message `Instance of Fractional Integer required for definition of eval` when processing the `eval` group of functions. Apart from these minor changes, the Miranda and Haskell scripts are identical.

```
1:  -- ******************************************
2:  --   Program to generate symbolic sequences
3:  -- ******************************************
4:
5:  data Expn = Var | Number |
6:    Divide Expn Expn |
7:    Mult Expn Expn |
```

```
   Plus Expn Expn |
   Minus Expn Expn |
   Power Expn Expn |
   Fact Expn |
   Const Integer |
   Eval Expn

-- ********************************************
--   xlate will convert an Expn to printable form
--   given n and v, which are to be substituted
--   for occurences of Number and Var
-- ********************************************

xlate :: Expn -> Integer -> Char -> [Char]
xlate Number n v = (show n)
xlate Var n v = [v]
xlate (Divide e1 e2) n v = (xlate e1 n v) ++ "/" ++ (xlate e2 n v)
xlate (Mult e1 e2) n v = (xlate e1 n v) ++ "*" ++ (xlate e2 n v)
xlate (Plus e1 e2) n v = (xlate e1 n v) ++ "+" ++ (xlate e2 n v)
xlate (Minus e1 e2) n v = (xlate e1 n v) ++ "-" ++ (xlate e2 n v)
xlate (Power e1 e2) n v = (xlate e1 n v) ++ "^" ++ (xlate e2 n v)
xlate (Fact e) n v = (xlate e n v) ++ "!"
xlate (Const m) n v = show m
xlate (Eval e) n v = show (eval e n)

-- ********************************************
--  descriptors for test cases
-- ********************************************

d1 = (Power Var Number)
d2 = (Divide (Power Var Number) (Fact Number))
d3 = (Divide Number (Power Var Number))
d4 = (Mult Number (Power Var Number))
d5 = (Divide (Power Var Number) (Eval (Power (Const 2) Number)))

-- ********************************************
--   eval evaluates an Expn which should
--   only contain constants and operators
-- ********************************************

eval (Const m) n = m
eval Number n = n
eval (Divide e1 e2) n = (eval e1 n) `div` (eval e2 n)
eval (Mult e1 e2) n = (eval e1 n) * (eval e2 n)
eval (Power e1 e2) n = (eval e1 n) ^ (eval e2 n)
eval (Plus e1 e2) n = (eval e1 n) + (eval e2 n)
eval (Minus e1 e2) n = (eval e1 n) - (eval e2 n)
```

```
eval (Fact e) n = fact (eval e n)

fact 0 = 1
fact (n+1) = n * (fact n)

-- *******************************************
--  Functions for generating sequences
--  of expressions
-- *******************************************

exseq ex ns v = [xlate ex n v | n <- ns]

plusses [ex] = ex
plusses (ex:exs) = ex ++ " + " ++ (plusses exs)

pms [ex] = ex
pms [ex1, ex2] = ex1 ++ " + " ++ ex2
pms (ex1 : ex2 : ex3 : exs) = ex1 ++ " + " ++ ex2 ++ " - " ++ (pms (ex3 : exs))
```

3 The hugs License

The license under which hugs (February 2000 version) is distributed is presented below. The license if very liberal and allows redistribution with relatively few restrictions.

```
The Hugs 98 system is Copyright (c) Mark P Jones, Alastair Reid, the
Yale Haskell Group, and the Oregon Graduate Institute of Science and
Technology, 1994-1999, All rights reserved, and is distributed as free
software under the following license.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

- Neither name of the copyright holders nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND THE CONTRIBUTORS
```