Adaptable Software: How to build a modular monadic extensible compiler using The State Monad and pseudoconstructors over monadic values

by Dan Popa
"Vasile Alecsandri" Univ. of Bacău

Abstract: The lazy evaluation mechanism included by the Haskell Language and the State Monad are used to build a modular plugin based compiler for a DSL called Simple. This is helping programmers to avoid the backpatching procedure, so producing a clear, modular simplified, monadic code generator.

Keywords and phrases: *the state monad, code layouts, modular trees and pseudoconstructors over monadic values, AST's*

Introduction

Three particular difficult tasks from the compiler building procedure are approached in this paper. The code generator we are writing here is intended to produce didactic code for the virtual machine described in [2].
The three problems which are subjects of our investigation and implementation in Haskell are:

1. The modular building of a compiler for a domain small language (DSL) which should be simple enough to be presented to the students from the faculties of Informatics or Computer Science. The task of language building is usually considered difficult by students, so the need of modular, clear, simplified implementations.
2. To prove that pseudoconstructors over monadic values defined and used by us in [4] are usable for adaptable modular compiler construction.
3. The usage of the famous State Monad – which is described in [3] by Paul Hudak and colleagues as being an important piece of software for Haskell programmers.

Due to the lazy evaluation system of the Haskell language and it's interaction with the mutually recursive procedure of the code generator and the backpatching procedure wich is frequently involved in usual (C based) compilers, the problem of building code generators for compilers in Haskell captured our interest. We intend to show that *backpatching* procedure becomes obsolete and should not be implemented in Haskell, at all. The proof will be constructive, by effectively building a compiler.
Also, due to the process of development by successive release of versions, which is implied by the languages construiction, the modular compilers are attracting researchers. It is a nice goal to build a compiler by simply putting together some Lego-like pieces or modules and just compiling it in a single executable binary.
The code generator will be shown in the next paragraphs, being incrementally build,

and, we have to say, in a manner inspired by the book [2]. The code generator is suitable to be used in the backend of The Simple Language, or other imperative languages.

Prerequisite

The reader should be familiar with the Simple's Code Generator from [2] (Chap.7) and the idea of backpatching one of the challenges being to avoid the writing of the backpatch procedure by using the graph-reduction mechanism in Haskell. It will demonstrate the posibilities of the lazy-evaluation mechanism of the Haskell implementation. As a result, the adresses wich are usually backpatched will be computed *by need* and at the right moment, by the Haskell program.
Also, the reader should be familiar with the assembly languages as those one produced by this pretty-printing function (quoted below).
The pretty printing procedure we are using here is accepting a term having the form of wto embraced pairs: ((a,l), b) , where a is the length of the code, l is the list of instructions and b is the base address in the code segment.
The pretty-printing function used by us is:

```
---- PRETTY PRINTER ----
prettyprint ((a,l),b)
  = "\n" ++ "Length of the code:" ++
     show a ++ myprintl 0 l                    – 0, as base address can be replaced by b
```

The myprint function is used to print the list of instructions, and is having the following definition.

```
myprintl nr []
  = "\n" ++ show nr
myprintl nr ((Instr a b) : l)
  =  "\n" ++ show nr ++ "\t" ++
     a ++ show b ++ myprintl (nr+1) l
```

It is using a basic (tail) recursion pattern to increment and transmitt the current address, noted as *nr*. As a result, such printing procedure will produce listings like the following (example).

```
{-- *MCOMP> mainA0

Length of the code:9
0       LD_INT 10
1       LD_INT 20
2       GT 0
3       JZ 7
```

```
4       LD_INT 45
5       STORE 120
6       JP 9
7       LD_INT 50
8       STORE 120
9*MCOMP>
--}
```

The instructions (mnemonics and operands) of the assembly language used here are described, are implemented using a Haskell *data declaration*:

```
----- INSTRUCTIONS ----
data  Instr a = Instr String a
                    deriving (Show,Read,Eq)
```

Each one of the elements of this type is having a name (for example “LD_INT”) and an argument, a. The type of a is left free. The instructions can be shown, read and compared for equality.

The first monad we had used

The pseudoconstructors over monadic values introduced by us in [4] and used in [5]. They was defined to work with any monad, but due to the need of storing the current address as a state, we begun by using *the state monad* as presented in [3], (section 9.3.) having Int as the type of the states. The addresses in the machine code segment are, as usually, short integers, Int is, consequently, a good choice. So we had defined:

```
type S = Int
```

and had included the state monad by defining the state-capsules of state transformations as in:

```
---- THE STATE MONAD ----
data  SM a = SM (S -> (a,S))
```

The SM datatype defined above is declared as instance of the state monad being supplied the common operators as in section 9.3 of [3] .

```
instance Monad SM where
  SM c1 >>= fc2 = SM (\st0 -> let (r,st1) = c1  st0
                                  SM c2  = fc2 r
                              in c2 st1 )
  return k    = SM (\st -> (k,st))
```

Also, some basic monadic actions was defined:

```
readSM :: SM S
readSM = SM (\s -> (s,s))
```
The above readSM action reads the current address from the state monad.

```
updateSM :: (S -> S) -> SM S
updateSM f = SM (\s -> (s, f s))
```

Despite the recommendation of updating the state we had widely used a WriteSM action which – inside the monadic capsule - is acting as identity on it's first operand.

```
writeSM :: S -> SM S
writeSM a  = SM (\s -> (s,a))
```

```
allocSM :: S -> SM S
allocSM l = SM (\s -> (s, s+l))
```

```
runSM :: S -> SM a -> (a,S)
runSM s0 (SM c) = c s0
```

The runSM function is used, as usually, to trigger the computation in the state monad.

The compiler

As a consequence of the separation of the three components (or sets of components):

- the (state) monad
- the pretty printer
- and, last but not least, the *modular monadic entry-pointless compilation rules* we had defined,

the whole compiler can now be written in a single,short, line:

```
compile :: (Show t, Show t1) => SM (t, [Instr t1]) -> IO ()
compile arb = putStr . prettyprint $ runSM 0000 arb
-- arb being the modular monadic syntax tree, built using pseudoconstructors over monadic values, instead of regular data constructors
```

Some practical remarks:

The 0000 used above is the start address of the compilation process, so being the first address where the code will be generated. Of course, it can be changed or added as parameter of the compiler itself. Let's see how the compilation works and how the *modular monadic entry-pointless compilation rules of compilation was defined.*

Constant's compilation

Compiling the constants has the simple effect of emitting a “LD_INT” code followed by the value of the constant. So the module of the compiler is:

```
constant nr
  = do {        a0 <- readSM;
                let a1 = a0 +1
                in do {
                        writeSM a1;
                        return (1, [Instr "LD_INT " nr] )
                        }
        }
```

The current (free) address is recovered from the state and incremented as a result of code generation, then, stored back in the state of the monad. As a result we can define a main IO action as:

```
mainA1 = compile  (constant 10)
```

Running the program is producing he following output:

```
*MCOMP> mainA1
Length of the code:1
0       LD_INT 10
1
*MCOMP>
```

Variable's compilation

The compilation of the variables needs to use a symbol table, a mechanism able of identifying the location of each variable in the heap or the data segment. For our example, the following (dummy) function will be enough,

```
        symtable x  = 000 + ord x
```

because we intend to use 1 letter identifiers for variables and generate a short code, for tests only. For a long code, the 000 should be replaced by the base address in the used data-segment. The reader should feel free to change the code, here.
Receiving the address of a variable from the symtable function, the compilation of a variable – as it is used in expressions – is modularly described as:

```
variable s
  = do {        a0 <- readSM;
                let a1 = a0 +1
                    adr = symtable (s)
```

```
                in do {
                        writeSM a1;
                        return (1, [Instr "LD_VAR " adr] )
                        }
        }
```

So, running this simple action:

```
        mainB1 = compile  (variable 'a')
```

we have got this output:

```
MCOMP> mainB1
Length of the code:1
0       LD_VAR 97
1
```

The code of the character will modify the offset of the variable in the data segment:

```
        mainB2 = compile  (variable 'A')
```

Running this action in the IO monad we have got:

```
MCOMP> mainB2
Length of the code:1
0       LD_VAR 65
1
```

Data declarations

If the declarations are compiled in the manner used in [2][Anthony A Aabey - Compiler Construction Using Flex and Bison] and the code produced by a high level declaration is just a “DATA” instruction followed by the number of variables to be allocated on the stack. Then, in this case, the following module is usable:

```
-- Compiling declarations
-- nr  – the number of successive mono-typed variables
-- being n variables n locations are allocated, counting from 0 to n-1

datas n
  = do {        a0 <- readSM;
                let a1 = a0 + 1
                    arg = n -1
                in do {
                        writeSM a1;
                        return (1, [Instr "DATA" arg] )
                        }
        }
```

For example, the action of compiling the declarations of ten identical variables will

be compiled as seen below:

```
mainC1= compile  (datas 10)
Length of the code:1
0       DATA 9
1
MCOMP>
```

Remark: our dummy *symtable* function is correlated with the use of an amount of space, which is reserved for all possible variables, not only for few. Of course, this module can be replaced with an other one, if needed.

Compiling skip

Inspiring by the previous rule, we can compile *skip* by producing no code at all. In this case the first attempt is:

```
-- Compiling skip according with the previous model
skip ::  SM (Int,[Instr Int])
skip
  = do {        a0 <- readSM;
                let a1 = a0 + 0  -- conforming with the template
                in do {
                        writeSM a1;
                        return (0, [] )
                        }
        }
```

This code, which is matching the previous template, can be simplified till we get:

```
skip ::  SM (Int,[Instr Int])
skip  = return (0, [] )
```

Defining the action of compiling skip, a 0 length code is produced by running:

```
mainD1 = compile skip

Length of the code:0
0
```

Remark: The type signature SM (Int,[Instr Int]) is required by the empty list, because it's type can not be computed by the type system.

Compiling I/O statements: the read command

In [2][Anthony A Aabey - Compiler Construction Using Flex and Bison], the reads are treated in a simplified manner based on the idea that the virtual machine is including a special instruction, a special mnemonic, usable for the read of a single variable. So is simple to generate the code when reading a variable is needed:

```
-- Compiling I/O  operations: the  read statement
-- note that read is just used, so we will note our module as “readv”


readv s                     -- because the word “read” is in use
  = do {        a0 <- readSM;
                let     a1 = a0 +1
                        adr = symtable (s)
                in do {
                        writeSM a1;
                        return (1, [Instr "IN_INT " adr] )
                        }
        }
```

Now it's time to define the I/O action which prints the result of a read's compilation:

```
        mainD2 = compile (readv 'x')
```

Compiling it, the output is:

```
Length of the code:1
0       IN_INT 120
1
*MCOMP>
```

Compiling I/O statements: the write <exp> command

According to [2] compiling a write means to compile the expression which is following the write and adding a special instruction of the virtual machine. This is because of the presence of such an instruction, “OUT_INT” in the description of the virtual machine used there. In practice, different sort of codes or calls to the API of the operating system should be generated instead of this single instruction. (Note the unused operand, 0.) The module of the compiler will be:

```
-- Compiling writings looks close to the compilation of assignments
write exp
  = do {        a0 <-readSM;
                (l1,cod1) <- exp;
                let     a1 = a0 + l1
                        a2 = a1 + 1
                in do {writeSM a2;
                        return (l1 + 1, concat      [cod1,
                                                     [Instr "OUT_INT " 0] ] )
                        }
        }
```

After defining an action which compile a write and prints the result,

```
        mainE1 =compile (write (variable 'x'))
```

we have got this output:

```
Length of the code:2
0       LD_VAR 120
1       OUT_INT 0
2
```

Anticipating a bit, after the definition of the compilers modules which are responsible of arithmetic operations we can define:

```
        mainE2 = compile (write (plus (constant  10) (constant 20)) )
```

And get this output:

```
*MCOMP> mainE2

Length of the code:4
0       LD_INT 10
1       LD_INT 20
2       ADD 0
3       OUT_INT 0
4
```

Assignment's compilation

In [2] the layout of the code being generated when an assignment is compiled is composed by the code of the expression from the left side of the assignment followed by an instruction which is executed in order to store the result from the main (Accumulator) register. This last instruction is having the storage address (which is the location of the variable) as operand. The address of the variable is provided – in this case – by the *symtable* function. The module of the compiler is:

```
-- Compiling assignments
attr s exp
  = do {        a0 <-readSM;
                (l1,cod1) <- exp;
                let     a1 = a0 + l1
                        a2 = a1 + 1
                        adr = symtable s
                in do {writeSM a2;
                        return (l1 + 1, concat      [cod1,
                                                     [Instr "STORE " adr] ] )
                        }
        }
```

In order to use more elaborate expressions, binary (see the Anexa) and unary operators have to be used. For example, compiling the addition of two expressions means to compile the first one, then, compile the second one. After both codes, a supplementary instruction is added: ADD, which is the addition made by the virtual machine. Here is the addition's module of the compiler:

```
-- compilarea sumelor
plus exp1 exp2
  = do {        a0 <-readSM;
                (l1,cod1) <- exp1;
                writeSM (a0+l1);
                (l2,cod2) <- exp2;
                let   a3 = a0 + l1 + l2 + 1
                in do {writeSM a3;
                        return (l1+l2+1, concat  [cod1,
                                                  cod2,
                                                  [Instr "ADD " 0 ] ] )
                        }
        }
```

Now we can define an action like:

```
        mainAS = compile (attr 'x' (plus (variable 'x') (constant 1)))
```

And we get the following code as result of the compilation.

```
Length of the code:4
0       LD_VAR 120        – – get the value of the variable
1       LD_INT 1          – – the constant;  1
2       ADD 0             – – compute x + 1
3       STORE 120                 – – store it at the location of 'x'
4
*MCOMP>
```

By adding all binary operators, module by module, the compiler becomes able to compile complex expressions. (Of course, in the hypothesis that the parser will provide the correct tree for the expression, as a result of syntax analysis.)

Compiling alternatives / conditionals

Compiling the *if* command is probably made by the largest module of the compiler. (Maybe the try - catch – throw module could be bigger.) The compilation of a regular if … then … else … command will produce a piece of code composed by three little sequences (one for the expression, two for the branches) and two jumps.
In order to assure the correct base address for the code generation process made by the three imbricated uses of other three modules of the compiler, the new base address for each code should be computed and stored in the state monad every time before the start of each generator. (See before each writeSM action.)

```
iif  cond s1 s2
  = do{a0 <-readSM;
        (l1,cod1) <- cond;
        writeSM (a0+l1+1);
        (l2,cod2) <- s1;
        writeSM  (a0 + l1 + 1 + l2 + 1) ;
```

```
        (l3,cod3) <- s2;
        writeSM  (a0 + l1 + 1 + l2 + 1 +l3);
        return (l1+1+l2+1+l3 , concat [cod1,
                                       [Instr "JZ " (a0 + l1 + 1 + l2 + 1)  ],
                                       cod2,
                                       [Instr "JP " (a0 + l1 + 1 + l2 + 1 + l3) ],
                                       cod3 ] )
      }
```

Notations: l1,l2,l3 are the lengths of the three sequences of code. The following expressions are computing the base address for each sequence of code:

```
(a0+l1+1) ;
(a0 + l1 + 1 + l2 + 1) ;
(a0 + l1 + 1 + l2 + 1 +l3);
```

This is particularly important if the sequences are containing conditionals or loops, because the destination addresses appearing in that code should be correctly computed.

Just as a little demonstration,in the beginning, let's compile a simple if, without interior loops or conditionals:

```
mainI1 =
        compile (iif  (gt (constant  10) (constant 20))
                      (attr 'x' (constant 45))
                      (attr 'x' (constant 50))  )
```

which is the syntax tree of a funny statement like: if (10 > 20) then x:=45 else x:=50.

```
Length of the code:9
0       LD_INT 10
1       LD_INT 20
2       GT 0
3       JZ 7
4       LD_INT 45
5       STORE 120
6       JP 9
7       LD_INT 50
8       STORE 120
9
*MCOMP>
```

Remark: This is unoptimized code, being produced by an unoptimized abstract syntax three, as the above. But it is not the task of the code generator to optimize such trees. A closed to the reality example may be:

```
mainI3 = compile (iif (gt (variable 'x')(constant 0))
                          (attr 'x' (constant 1))
                          (attr 'x' (constant 2 )) )
```

This AST is reprezenting a statement like: if (x>0) then x:=1 else x:=2

```
*MCOMP> mainI3
```

```
Length of the code:9
0       LD_VAR 120
1       LD_INT 0
2       GT 0
3       JZ 7
4       LD_INT 1
5       STORE 120
6       JP 9
7       LD_INT 2
8       STORE 120
9
*MCOMP>
```

Also, the 120 appearing in the code is the location of the 'x' variable as it was provided by the symbol table's function, *symtable*.

Sequence's compilation

In this demonstrative module of the modular monadic compiler only two intermediate addreses should be saved and stored in the state of the state monad. As a consequence, a simple let...in... Haskell expression can do the trick, and the module of the compiler is:

```
-- Compiling sequences of two statements. Longest sequences can also be
represented using a stair of such levels.
-- The intermediate address, the first one after the first sequence of code should be
saved in the state monad, in case that the second sequence is containing loops or
conditionals.

sequ s1 s2                    -- “seq” is used otherwise in Haskell
  =do {a0 <-readSM;
       (l1,cod1) <- s1;
       let a2 = a0 +l1
       in do
           { writeSM a2;    -- the begining of the second code should be stored in state
             a2 <-readSM;   -- a bit redundant
            (l2,cod2) <- s2;
             let   a4 = a2 +  l2
             in do {  writeSM a4;
                      return (l1+l2 , concat [cod1, cod2] )
                   }
           }
      }
```

As it is visible above, the use of let...in … expressions is an alternative to the use of those long expressions in the *writeSM* actions, but it did not shorten the code – in

fact new levels of do-notation are introduced (with all accompanying symbols). That is why the previous module dedicated to the compilation of conditionals had NOT used the let … in … expressions.
Now, let's compile a sequence of two statements by defining and firing an action:

```
mainW = compile (sequ (attr 'x' (constant 45))  (attr 'x' (constant 50))  )
```

Starting the above action we have got this code:

```
MCOMP> mainW

Length of the code:4
0       LD_INT 45
1       STORE 120
2       LD_INT 50
3       STORE 120
4
*MCOMP>
```

So, this module is also functional.

Compiling the While Loop

The layout of the code producing when a while loop is compiled is containing only one (imbricated) major sequence of code, the corp of the loop. So, only one let…in..expression is used in this module of the compiler:

```
while cond s1
  =do {a0 <-readSM;
       (l1,cod1) <- cond;
       writeSM (a0+l1+1);
       (l2,cod2) <- s1;
       let   a2 = a0 + l1  + 1 + l2+1
       in do {writeSM a2;
              return (l1+l2+2, concat  [cod1,
                                        [Instr "JZ " a2 ],
                                        cod2,
                                        [Instr "JP " a0]] )
             }
      }
```

In this moment we can define this action in order to produce the code starting for this dummy while loop.

```
mainW0 = compile
                (while (gt      (constant  10) (constant 20))
                       (attr 'x' (constant 45)))
```

And the generated code is:

```
Length of the code:7
0       LD_INT 10
```

```
1       LD_INT 20
2       GT 0
3       JZ 7
4       LD_INT 45
5       STORE 120
6       JP 0
7
MCOMP>
```
A more realistic example can be this one, a while loop decrementing it's counter:
```
        mainW1 = compile
                (while (gt (variable 'x') (constant 0))
                        (attr 'x' (minus (variable 'x') (constant 1)) ))
```
Producing:
```
Length of the code:9
0       LD_VAR 120
1       LD_INT 0
2       GT 0
3       JZ 9
4       LD_VAR 120
5       LD_INT 1
6       SUB 0
7       STORE 120
8       JP 0
9
MCOMP>
```

Compiling the do-while loop

An other kind of loop is the do-while loop, also looping “around” a block of code – only one. So an other module of the compiler can be written, defining an other pseudoconstructor over monadic values:

```
dowhile s1 cond
  = do { a0 <-readSM;
        (l1,cod1) <- s1;
        writeSM (a0+l1);
        (l2,cod2) <- cond;
        let   a2 = a0 + l1  + l2 + 1
        in do {writeSM a2;
                return (l1+l2+1, concat  [cod1,
                                          cod2,
                                          [Instr "JNZ " a0 ] ] )
              }
       }
```

The layout of the produced code is recognizable as argument of the monadic return operator, as usual in this paper. An action can be written, as ana example, too see how do-whiles are compiled by this module:

```
mainDW1 = compile
            (dowhile    (attr   'x' (constant 45))
                        (gt (constant  10) (constant 20) ) )
```

And here is the machine code produced by running this action:

```
*MCOMP> mainDW1

Length of the code:6
0       LD_INT 45
1       STORE 120
2       LD_INT 10
3       LD_INT 20
4       GT 0
5       JNZ 0
6
MCOMP>
```

And here is a bigger, more realistic example:

```
mainDW2 = compile
        (dowhile    (attr 'x' (minus (variable 'x') (constant 1)) )
                    (gt (variable 'x') (constant 0)) )
```

This tree build with pseudoconstructors is in fact the tree of a statement which looks similarly with: *do { x = x -1 ; } while x > 0 .* And here is the code:

```
Length of the code:8
0       LD_VAR 120
1       LD_INT 1
2       SUB 0
3       STORE 120
4       LD_VAR 120
5       LD_INT 0
6       GT 0
7       JNZ 0
8
*MCOMP>
```

Conclusion

At this point, a set of conclusions can be drawn, the important, in our opinion are:

1. The *backpatching* procedure can be successfully removed from the compiler, it's place being taken by the automatic graph-reduction procedure which is included in the lazy-evaluation mechanism of the Haskell language. The programmer should not worry about it, nowadays. As the reader had *noticed by absence*, the *backpatching* procedure is not necessary anymore when using

a language like Haskell for compiler construction.

2. The idea of using *data types a la carte as presented by Wouter Swierstra*, was not used by us, leading to a simple, clear, solution.
3. In fact, *pseudoconstructors over monadic values* are assuring us a simple and flexible environment. This is why his paper is not cited as reference, here. Modular monadic compilers, horizontally sliced in portions containing a modular monadic parser, a modular monadic typechecker , and this above described, modular code generators can be built (around pseudoconstructors over monadic values,) having modular trees built on pseudoconstructors over monadic values as internal representations of ASTs.
4. No explicit functors are used – even if implicit functors and categories - are involved in the compilation process. This makes a compiler design which is suitable for both students and programmers,too.
5. The compiler is modularly extensible, being a collections of descriptions which can be all placed in different modules, imported and compiled together with a small main program. Modules are reusable. The replacement of usual tree declarations with *pseudoconstructors over monadic values* eliminates the need of back-reediting the tree's declarations, and also eliminates the dependence between modules and that AST's declarations.
6. This design can be developed, still remaining based on this notions *the state monad, code layouts, modular trees and pseudoconstructors over monadic values*. More layouts of code can be found in classic books on compiler constructions, like [1]. Speed can be improved, using functional composition. This should be investigated. The code of the compiler and also some different but related approaches are available from [6].

References

[1] Alfred V. Aho, Monica S. Lam, Rav Sethi, Jeffrey D. Ulman, Compilers Principles Techniques And Tools Second Edition , Addison Wesley, Pearson Education, 2007

[2] Anthony A Aabey - Compiler Construction Using Flex and Bison, Walla Walla College, Version of February 25, 2004

[3] Paul Hudak, Joseph H Fasel (et.co.) A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Also available as Research Report YALEU/DCS/RR-901, Yale University, Department of Computer Science, April 1992.

[4] Dan Popa – Direct modular evaluation of expressions using the monads and type classes in Haskell, STUDII ŞI CERCETĂRI ŞTIINŢIFICE Seria: MATEMATICĂ , UNIVERSITATEA DIN BACĂU, Nr. 18 (2008), pag. 233 – 248

[5] Dan Popa - Adaptable Software – Modular extensible monadic evaluator and typechecker based on pseudoconstructors, ARA Congress: ARA35 – SCIENCE & ART IN THE INFORMATICS' ERA, 6-10 Jully 2011, "POLITEHNICA" University of Timisoara

[6] Dan Popa - Modular_Monadic_Compilers_for_Programming_Languages http://www.haskell.org/haskellwiki/Modular_Monadic_Compilers_for_Programming_Languages

Appendix A
The module of the compiler which is responible of the compilation of the “>” comparation operator is presented below, in case the reader needs it.

```
gt exp1 exp2
  = do{a0 <-readSM;
       (l1,cod1) <- exp1;
       writeSM (a0+l1);
       (l2,cod2) <- exp2;
       let   a3 = a0 + l1 + l2 + 1
       in do {  writeSM a3;
                return (l1+l2+1, concat   [cod1,
                                           cod2,
                                           [Instr "GT " 0 ] ] )
            }
```

Of course, other modules of the modular compiler, responsible for other binary operators are similarly written.

Appendics B
Also some programs from the end of section 7 of [1] was compiled as part of our experiments. Here is such a piece of code, represented using pseudoconstructors over monadic values:

```
main4 = putStr . prettyprint $ runSM 0
   (program (datas 2)
          (sequ (readv 'n')
           (sequ (iif (lt (variable 'n') (constant 10) )
                      (attr 'x' (constant 1))
                      (skip)
                 )
                 (while (lt (variable 'n')  (constant 10))
                        (sequ (attr 'x' (mult (constant 5)(variable 'x'))
                              )
                              (attr 'n' (plus (variable 'n') (constant 1))
                              )
                        )
                 )
           )
          )
   )
```

Despite some different mnemonics (jumps are writtenby us in Z80's notations) and the different symtable() function used – which allocates different addresses for

variables - the produced code have the same layout.

```
*MCOMP> main4

Length of the code:22
0       DATA 1
1       IN_INT 110
2       LD_VAR 110
3       LD_INT 10
4       LT 0
5       JZ 9
6       LD_INT 1
7       STORE 120
8       JP 9
9       LD_VAR 110
10      LD_INT 10
11      LT 0
12      JZ 22
13      LD_INT 5
14      LD_VAR 120
15      MULT 0
16      STORE 120
17      LD_VAR 110
18      LD_INT 1
19      ADD 0
20      STORE 110
21      JP 9
22*MCOMP>
```

The original example from [2] can be found in Figure 7.2, page 35 in the 2004 edition, and compared with the above code.