November 2023

Conditions for Assignment Completion

Group Formation: Groups of two (2) students enrolled in the same practical class. Exceptionally, and only if necessary, groups of three may be accepted.

Group and Topic Choice: The groups are chosen in the activity available for this purpose on Moodle. Groups are already created, named *Txx_Gyy*, where *Txx* represents the practical class (from T01 to T14), and *Gyy* represents the group within the class (from group G01 to G14). Groups from each practical class should organize themselves into these groups. Registration is mandatory for all students taking the assignment.

Deadlines: The assignment (source code and readme file) must be delivered by 8:00 AM on January 2, 2024, in the activity to be made available for this purpose on Moodle, with demonstrations carried out during the week of January 2-5, 2024. The demonstrations should be scheduled with the teacher of each practical class.

Assessment Weight: This assignment counts for 25% of the final grade. The evaluation focuses on implemented features, the quality of the code and respective comments, the readme file, and participation in the assignment presentation. Grades may differ between group members.

Languages and Tools: The implemented assignment must run under GHCi version 9.0 or later. Assignments will be subjected to a battery of tests, so you must adhere to the stated file and function naming conventions (details at the end of this document and also in the submission activity). The inability to test the developed code will result in penalties in the evaluation.

Assignment Description

1. Consider a low-level machine with configurations of the form (c, e, s) where c is a list of instructions (or code) to be executed, e is the evaluation stack, and s is the storage. We use the evaluation stack to evaluate arithmetic (composed of integer numbers only, which can positive or negative) and boolean expressions.

   The instructions of the machine are: **push-n, add, mult, sub, true, false, eq, le, and, neg, fetch-x, store-x, noop, branch(c1, c2)** and **loop(c1, c2)**.

   There are several basic arithmetic and boolean operations:

   - **add**, **mult** and **sub** add, subtract and multiply the top two integer values of the stack, respectively, and push the result onto the top of the stack. In particular, **sub** subtracts the topmost element of the stack with the second topmost element.

- **eq** and **le** compare the top two values of the stack for equality and inequality, respectively, and push a boolean with the comparison result onto the top of the stack. **eq** can compare both integers and booleans, while **le** only works for integers. In particular, **le** determines whether the topmost stack element is less or equal to the second topmost element.

In addition to the usual arithmetic and boolean operations, there are six instructions that modify the evaluation stack:

- **push-n** pushes a constant value n onto the stack; **true** and **false** push the constants *tt* and *ff*, respectively, onto the stack.
- **fetch-x** pushes the value bound to x onto the stack, whereas **store-x** pops the topmost element of the stack and updates the storage so that the popped value is bound to x.
- **branch(c1, c2)** will also change the flow of control: if the top of the stack is the value *tt* (that is, some boolean expression has been evaluated to true), then the stack is popped and *c1* is to be executed next. Otherwise, if the top element of the stack is *ff*, then it will be popped and *c2* will be executed next.
- **noop** is a dummy instruction that returns the input stack and store.

There are two instructions that change the flow of control:

- **branch(c1, c2)** will be used to implement the conditional: as described above, it will choose the code component *c1* or *c2* depending on the current value on top of the stack. If the top of the stack is not a truth value, the machine will halt as there is no next configuration (since the meaning of branch(⋯,⋯) is not defined in that case).
- A looping construct such as a while loop can be implemented using the instruction **loop(c1, c2)**. The semantics of this instruction is defined by rewriting it to a combination of other constructs, including the branch instruction and itself. For example, $loop(c1, c2)$ may be transformed into $c1 \mathbin{+\!\!+} [branch([c2, loop(c1, c2)], [noop])]$.[1]

Example 1 Assuming that the initial storage s has the pair $(x, 3)$ (meaning that variable x has value 3), running step by step the program $[push - 1, fetch - x, add, store - x]$ with an empty stack and storage s, represented by the triple

$$([push - 1, fetch - x, add, store - x], [], [(x, 3)])$$

gives (step by step): $([fetch{-}x, add, store{-}x], [1], [(x, 3)])$, $([add, store{-}x], [3, 1], [(x, 3)])$, $([store - x], [4], [(x, 3)])$ and finally the output $([], [], [(x, 4)])$

Example 2 Running step by step the program $[loop([true], [noop])]$ with an empty stack and storage s, represented by the triple

$$([loop([true], [noop])], [], s)$$

[1] $+\!\!+$ is list concatenation

gives (step by step):

- $([loop([true],[noop])],[],s)$
- $([true, branch([noop, loop([true],[noop])],[noop])],[],s)$
- $([branch([noop, loop(true,[noop])],[noop])],[tt],s)$
- $([noop, loop([true],[noop])],[],s)$
- $([loop([true],[noop])],[],s)$
- ...

and the unfolding of the loop-instruction is repeated. This is an example of an infinite looping computation sequence.

Example 3 Running a wrong configuration should return an error message. For example running the program [*Push 1, Push 2, And*] should raise an exception with the string: "Run-time error" (calling function *error "Run-time error"*).

In your program use the following data declaration in Haskell to represent instructions in the previous machine (*Inst*), alongside with a type synonym for a sequence of instructions (*Code*). The auxiliary .hs file provided in Moodle already contains these declarations to be use in the assignment.

```
data Inst =
  Push Integer | Add | Mult | Sub |
  Tru | Fals | Equ | Le | And | Neg |
  Fetch String | Store String | Noop |
  Branch Code Code | Loop Code Code
  deriving Show
type Code = [Inst]
```

(a) Define a new type to represent the machine's stack. The type must be named *Stack*.

(b) Define a new type to represent the machine's state. The type must be named *State*.

(c) Implement the *createEmptyStack* function which returns an empty machine's stack.

$$createEmptyStack :: Stack$$

(d) Implement the *createEmptyState* function which returns an empty machine's state.

$$createEmptyState :: State$$

(e) Implement the *stack2Str* function which converts a stack given as input to a string. The string represents the stack as an ordered list of values, separated by commas and without spaces, with the leftmost value representing the top of the stack.

$$stack2Str :: Stack \rightarrow String$$

For instance, after executing the code $[push-42, true, false]$, the string representing the stack is: *False,True,42*.

(f) Implement the *state2Str* function which converts a machine state given as input to a string. The string represents the state as an list of pairs variable-value, separated by commas and without spaces, with the pairs ordered in alphabetical order of the variable name. Each variable-value pair is represented without spaces and using an "=".

$$state2Str :: State \rightarrow String$$

For instance, after executing the code $[false, push - 3, true, store - var, store - a, store-someVar]$, the string representing the state is: *a=3,someVar=False,var=True*.

(g) Write an interpreter for programs in the same machine, which given a list of instructions (type defined as *Code*, i.e. *type Code = [Inst]*), a stack (type defined as *Stack*) and that is initially empty, and a storage (type defined as *State*), runs the list of instructions returning as ouput an empty code list, a stack and the output values in the storage. This evaluation function must be declared as:

$$run :: (Code, Stack, State) \rightarrow (Code, Stack, State)$$

2. Now consider a small imperative programming language with arithmetic and boolean expressions, and statements consisting of assignments of the form $x := a$, sequence of statements (instr1 ; instr2), if then else statements, and while loops.

   One can now define a translation (a compiler) from this language into lists of instructions in the previous machine.

   - Arithmetic and boolean expressions will be evaluated on the evaluation stack of the machine and the code to be generated must effect this. Thus, the code generated for binary expressions consists of the code for the right argument followed by that for the left argument and, finally, the appropriate instruction for the operator. In this way, it is ensured that the arguments appear on the evaluation stack in the order required by the instructions.

     Example 3 The compilation of $x + 1$ is $[push - 1, fetch - x, add]$.

     The code generated for an arithmetic expression must ensure that the value of the expression is on top of the evaluation stack when it has been computed.

   - The code generated for $x := a$ appends the code for the arithmetic expression a with the instruction $store - x$. This instruction assigns x the appropriate value and additionally pops the stack.
   - For a sequence of two statements, we just concatenate the two instruction sequences.

- When generating code for the conditional, the code for the boolean expression will ensure that a truth value will be placed on top of the evaluation stack, and the branch instruction will then inspect (and pop) that value and select the appropriate piece of code.
- Finally, the code for the while statement uses the loop-instruction.

Example 4 The compilation of the factorial program

$$y := 1; \; while \; \neg(x = 1) \; do \; (y := y * x; x := x - 1)$$

outputs the code:

```
[push-1,store-y,loop([push-1,fetch-x,eq,neg],
            [fetch-x,fetch-y,mult,store-y,push-1,fetch-x,sub,store-x)])]
```

(a) Define three datas in Haskell to represent expressions and statements of this imperative language:

- *Aexp* for arithmetic expressions
- *Bexp* for boolean expressions
- *Stm* for statements

(b) Define a compiler from a program in this small imperative language into a list of machine instructions (as defined in part 1). The main compiler is function:

- $compile :: [Stm] \rightarrow Code$

that must use the two mandatory auxiliary functions which compile arithmetic and boolean expressions, respectively:

- $compA :: Aexp \rightarrow Code$
- $compB :: Bexp \rightarrow Code$

(c) Define a parser which transforms an imperative program represented as a string into its corresponding representation in the *Stm* data (a list of statements *Stm*). The parser[2] function must have the following signature:

- $parse :: String \rightarrow [Stm]$

The string representing the program has the following syntactic constraints:

- All statements end with a semicolon (*;*).
- The string may contain spaces between each token (a *token* is basically a sequence of characters that are treated as a unit as it cannot be further broken down. Examples of tokens are the keywords (*while, if, then, else*, etc.), variables, operators (+, - , *), delimiters/punctuators like the semicolon(;) or brackets, etc. integers and boolean values are also tokens).

[2]You may either implement the parser directly in Haskell or study and use the parser library *Parsec*.

- Variables begin with a lowercase letter (assume that no variable name can contain a reserved keyword as a substring. For instance, *anotherVar* is an invalid variable name as it contains the keyword *not*).
- Operator precedence in arithmetic expressions is the usual: multiplications are performed before additions and subtractions. Additions and subtractions have the same level of precedence and are executed from left to right (i.e. they are left-associative). Multiplications are also left-associative.
- Parentheses may be used to add priority to an operation. For instance, *1+2*3 = 7* but *(1+2)*3 = 9*.
- In boolean expressions, two instances of the same operator are also computed from left to right. The order of precedence for the operations is (with the first one being executed first): integer inequality ($\leq$), integer equality ($==$), logical negation (*not*), boolean equality ($=$), logical conjunction (*and*). For instance, *not True and 2 $\leq$ 5 = 3 == 4* is equivalent to *(not True) and ((2 $\leq$ 5) = (3 == 4))*.

Hint: define an auxiliary function *lexer :: String → [String]* that splits the string into a list of words (*tokens*). Example:

*lexer "23 + 4 * 421" = ["23","+","4","*","421"]*

and from this list of tokens build the corresponding data, i.e.

parse = buildData . lexer

For various examples of the language, please see the .hs file provided in Moodle.

Submission and Evaluation

Each group must submit a README file in pdf format together with the source code. The submission has to be made on the Moodle platform, in a ZIP file named

$$PFL\_TP2\_TXX\_GYY.ZIP$$

where TXX_GYY is the group name (TXX specifies the practical class and GYY the group number).

The ZIP file should contain the README file and a folder named *src*, which should include the properly commented Haskell source code in a *main.hs* file.

There should be no additional folders in the zip file, i.e., both the readme.pdf file and src folder should be at the root of the archive.

The readme file should include the identification of the group (group name, student number, and full name of each member of the group), as well as an indication of the contribution (in percentages, adding up to 100%) of each member of the group to the assignment. It should also describe the strategy used for solving both parts of the assignment, detailing the decisions made when defining the *data* and functions defined in the program.