Computer Science 320 (Fall, 2009)
Concepts of Programming Languages

Assignment 8: Compilation Using Monads

Out: Tuesday, December 1, 2009
Due: Friday, December 11, 2009

In this assignment, you will write a compiler for a subset of Mini-Haskell. You will make changes only to the `Compile` module.

This assignment assumes you have completed the problems in in **Assignment 5**, **Assignment 6**, and **Assignment 7**. You may retrieve the new version of the source code, though only three modules are modified or new: `Machine.hs` defines the machine code to which you will be compiling Mini-Haskell expressions, `Main.hs` now has an additional function for testing the compiler, and `Compile.hs` is the skeleton code for this assignment.

The machine code instructions of the machine language to which you will be compiling Mini-Haskell code are listed in the `Instruction` data type definition found inside the `Machine` module. The appendix page of this assignment specifies the behavior of each instruction. Labels are program locations (as in C) and valid memory locations are represented as positive integers.

```
type Label = String
type MemLoc = Int
```

You are not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from the lecture notes and textbook. You may **not** use any other materials without explicit citation.

Problem 1. (20 pts)

The compiler will be implemented as a function `comp` that takes a Mini-Haskell expression and returns both a sequence of machine instructions and the memory location at which a result can be found once these instructions are executed. Thus, naively, its type might be:

```
comp ::  Exp -> (MemLoc, [Instruction])
```

However, this function must maintain an environment mapping variable names to memory locations. It must also maintain a list of fresh variables and fresh memory locations as it transforms an expression into a list of machine instructions. The `Env` module can be used for the environment, and the following two infinite lists can be used for fresh variables and memory locations:

```
memlocs :: [MemLoc]
memlocs = locs 0 where locs n = n:locs (n+1)

labels :: [Label]
labels = labs 0 where labs n = ("L"++show n):labs (n+1)
```

To operate, the `comp` function must take all of these components as arguments, and return these components (potentially modified) in its result. Thus, its actual type must be substantially more complicated:

```
comp :: Exp -> (Env MemLoc, [MemLoc], [Label])
            -> ((Env MemLoc, [MemLoc], [Label]), (MemLoc, [Instruction])
```

We can summarize the type of this function by introducing the monad `Fresh`:

```
data Fresh a =
  Fresh ((Env MemLoc, [MemLoc], [Label]) -> ((Env MemLoc, [MemLoc], [Label]), a))
```

Given any type `a`, a value of type `Fresh a` is a computation that cannot begin until the environment and fresh lists are supplied. Furthermore, upon terminating, this computation will return the environment and fresh lists, possibly modified to reflect any changes that occurred during the computation. Once we introduce this monad, the type for `comp` becomes manageable again.

```
comp ::  Exp -> Fresh (MemLoc, [Instruction])
```

(a) In the `Compile` module, add an `instance` declaration that makes the higher-order type `Fresh` a member of the `Monad` class. The `Monad` module has already been imported for you at the beginning of the `Compile` module.

(b) Define a value `freshMemLoc::Fresh MemLoc`. This value represents a computation that, when given an environment and two fresh lists, returns these along with a memory location from the fresh memory locations list. Be certain that the fresh memory location returned by the computation does not appear in the fresh memory locations list returned with it.

(c) Define a value `freshLabel::Fresh Label`. This value represents a computation that, when given an environment and two fresh lists, returns these along with a label from the fresh label list.

(d) Define a function `putVar::String -> MemLoc -> Fresh ()`. This function should update the environment maintained by the monad with a new entry that relates the specified variable name (a string) to the specified memory location. Since no result needs to be returned when an update occurs, the computation should return the environment and two lists along with a unit value, `()`.

(e) Define a function `getVar::String -> Fresh MemLoc`. This function should retrieve the memory location from the environment that is associated with the specified variable name and return it along with the environment and two lists. Remember that the environment will always contain the memory location corresponding to the variable because we compile only well-typed expressions in which all variables are bound.

Problem 2. (30 pts)

The function `comp::Exp -> Fresh (MemLoc, [Instruction])` can now be defined using Haskell `do` notation. In this problem, you will implement the cases that handle Mini-Haskell expressions consisting of integer constants, boolean constants, `if` expressions, and applications of arithmetic operators. **Remember that you can assume that expressions are well-typed.**

(a) Add a case that handles integer constants. If an expression is an integer, the compiler should find a fresh memory location. It should then generate code that sets the value of that location to the integer, and returns that memory location along with the generated machine code.

(b) Add a case that handles boolean constants. Boolean constants will be represented in memory using integer values. Any non-zero integer value can be used to represent `True`. Zero must be used to represent `False`. In this case, the compiler should behave as in part (a): it should obtain a fresh memory location, generate code that sets that location to the appropriate integer value, and return both.

(c) Add a case that handles conditional expressions. The compiler must reserve a fresh memory location, this time for the result of the entire computation represented by the expression. Then, it is necessary to compile each subexpression, which will generate three sequences of machine code. The three sequences represent the condition, the "true" block, and the "false" block.

It is then necessary to combine these appropriately by inserting jumps, conditional jumps, and labels. The sequence computing the condition must occur first. Then, if the memory location holding the result of the condition has a "true" value, the "true" block must be allowed to execute, and the second block should be skipped. Likewise, if the memory location holding the result of the condition has a "false" value, the code for the "true" block should be skipped, and the code for the "false" block should not. (Hint: Insert unique labels before and after each block, then decide where to place jumps or conditional jumps. Be careful to insert jumps before labels.)

Don't forget to copy the final result of the entire computation into the memory location you reserved at the very beginning. You may need to do this in multiple locations, as either block could be executed, and each block stores its result in a different location.

(d) Add two cases that handle expressions for addition and multiplication of the form:

```
(App (App (Op Plus) e1) e2)
```

Both subexpressions must first be compiled into sequences. Then, the sequences should be concatenated into a single sequence that ends with the appropriate machine instruction (either `Add` or `Mul`). You will again need to find a fresh memory location for the final result.

You should now be able to test your compiler on simple expressions such as:

```
if False then (if True then 1 else 2) else 3
```

One possible sequence of instructions that could be generated for the code above is reproduced below. **Note that other valid solutions exist.**

```
Set 1 0
CJump 1 "L0"
Set 3 1
CJump 3 "L2"
Set 4 1
CopyFromTo 4 2
Jump "L3"
Label "L2"
Set 5 2
CopyFromTo 5 2
Label "L3"
CopyFromTo 2 0
Jump "L1"
Label "L0"
Set 6 3
CopyFromTo 6 0
Label "L1"
```

Problem 3. (15 pts)

In this problem, you will implement the cases for `comp` that handle Mini-Haskell expressions consisting of `let` bindings and variables. As usual, you may assume that each `let` binding has only a single variable. You may extend your code to handle tuples as part of your solution to **Problem #5**.

(a) Add a case that handles variables. You will need to retrieve the memory location associated with the variable from the environment maintained by the monad.

(b) Add a case that handles `let` bindings. Both expressions must be compiled. Before the body is compiled, the environment must be updated so that it contains the memory location corresponding to the bound variable.

Problem 4. (35 pts)

In this problem, you will implement the cases for `comp` that handle Mini-Haskell expressions consisting of applications of the boolean and list operators.

(a) Add two cases that compile expressions for boolean conjunctions and disjunctions, i.e.

```
(App (App (Op And) e1) e2)
```

Recall that any non-zero integer represents "true", and zero represents "false".

(b) Add a case that compiles an application of the boolean negation operator: `App (Op Not) e`. There is a variety of ways to do this using the available machine instructions.

(c) Add two cases that compile the empty list expression `Nil`, and the application of the `(:)` operator (`(App (App (Op Cons) e1) e2)`). You may represent an empty list in memory using the integer value `0`. In order to handle `(:)`, you will need to obtain two fresh *adjacent* memory locations. You will then need to generate machine code that stores the *address* (that is, memory location) of the head of the list in the first location, and the address of the tail of the list in the second location. The best strategy is to return the memory location of a single new integer, and let this integer be the first of these two locations (this is analogous to returning a pointer to a list node).

(d) Add two cases that compile the `head` and `tail` operators. (Hint: You will need to employ the `DerefAndCopy` instruction.)

Unfortunately, only the value at the memory location containing the result can be displayed when running compiled code, as memory has no structure that could be used to determine whether a memory location holds a pointer or a value. Thus, you'll need to test code involving lists by using the `head` operator.

Problem 5. (*30 extra credit pts)

Extend your solutions to compile tuples. You will need to modify the manner in which you compile `let` bindings. You will also need to devise a method for representing tuples in memory. One approach is to use a technique similar to the one employed for lists: store the addresses of the components of a tuple in sequence. You do not need to check that the number of elements in a tuple is consistent, because the code being compiled is well-typed.

Problem 6. (*20 extra credit pts)

Devise a way to handle partially applied operators. There are no restrictions on solutions. Compiled expressions are well-typed, so it should be possible to devise a consistent representation.

A Semantics of Machine Code Instructions

We briefly define the behavior of the instructions in the target machine language.

```
type Label = String
type MemLoc = Int
data Instruction = Set MemLoc Int
                 | CopyFromTo MemLoc MemLoc
                 | DerefAndCopy MemLoc MemLoc
                 | Add MemLoc MemLoc MemLoc
                 | Mul MemLoc MemLoc MemLoc
                 | Label Label
                 | Jump Label
                 | CJump MemLoc Label
                 | Pass
```

The behaviors specified below are defined in the `Machine` module.

- `Set` stores the integer value supplied as its second argument in the memory location specified by the first argument;
- `CopyFromTo` copies the contents of the first memory location into the second memory location;
- `DerefAndCopy` obtains the address contained in the first memory location, then finds the value at this new address, and stores this value in the second memory location (thus, it derefences the pointer in the first memory location, then stores the value in the second memory location);
- `Add` takes the values in the first two memory locations, adds them, and stores the result in the third memory location;
- `Mul` takes the values in the first two memory locations, multiplies them, and stores the result in the third memory location;
- `Label` has no observable effects, but is used when executing `Jump` and `CJump` instructions;
- `Jump` automatically shifts control to the location specified by the supplied label;
- `CJump` checks the supplied memory location, and jumps to the specified label only if the value at the memory location is 0; otherwise, control is passed to the next instruction in the sequence;
- `Pass` has no observable effects.