Assignment 9

CSCI 3136: Principles of Programming Languages

Due Apr 16, 2011

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

In this assignment, you will augment your TOY compiler to provide the following extensions and optimizations:

Coroutines (5 marks). Your first task is to add coroutines (defined using the `cofun` keyword or as a coroutine lambda expression `[| ... |]`) to the TOY language. There are two parts to this.

You call a coroutine the same way you call a function: using `LOC` or the built-in function `call`, `?` or `??`. The code you generate for a coroutine is different from a function though. A coroutine operates in a different *context* (call stack and current address) from its caller. This context needs to be set up when the coroutine is called and needs to be destroyed before it returns. Thus, where every normal function starts with an `MFR` instruction, every coroutine starts with an `MCX` (make context) instruction followed by an `MFR` instruction. `MCX` creates a new context whose parent context is the context of the caller. `RET` at the end of the coroutine automatically returns control to the context of the caller and destroys the coroutine's context. The binary code of `MCX` is `0x1E`. This instruction has no arguments. (I will upload an extended virtual machine that supports this instruction.)

The second part is to support the transfer of data and control between the caller and the coroutine. Control is transferred to the parent context of the current context using the built-in `suspend` function. Calling this from a coroutine transfers control to the parent context and pushes a closure for continuing the coroutine on the parent context's data stack. The parent context can call the built-in function `resume` to resume the coroutine. This function expects a coroutine closure on the top of the current context's data stack, pops it off the stack, and transfers control to the coroutine.

The transfer of data between a coroutine and its parent context can only be performed from within the coroutine. The built-in function >> pops a data item off the parent context's data stack and pushes it on the coroutine's data stack. The built-in function << transfers one data item in the opposite direction.

These new built-in functions have numbers 67 (`suspend`), 68 (`resume`), 69 (>>), and 70 (<<). (The `stack` built-in I added as debugging support has number 66.)

Eliminate unneeded frames (5 marks). Frames are needed only to store local variables. Thus, if a function has no local variables, it needs no frame and you can avoid generating an `MFR` instruction at the beginning of a functions without local variables. Implement this optimization. Note that this means that the number of hops along the static chain given as argument to `LOA`, `LOC`, and `LOJ` instructions also needs to be adjusted. (Figuring out the number of hops needed is part of this part of the assignment. It's a fairly simple adjustment.)

Tail recursion (5 marks). Make your functions tail-recursive: Whatever the last function or coroutine call in a function is, jump to that function instead of calling it. For user-defined functions, use `LOJ` instead of `LOC`. For built-in functions, use the instruction `JBI` (Jump builtin) instead of `CBI`. I will add this instruction to the extended virtual machine with binary code `0x32`. Just as `CBI`, it takes a one-byte argument: the built-in function to jump to.

Something to note (even though it has no impact on what your compiler needs to do) is that the built-in functions `call`, `?`, and `??` automatically jump to the function or coroutine closure they are given as argument, instead of calling it. Thus, no changes to the call stack are made in this case either.