CMSC 22610 Winter 2009	Implementation of Computer Languages - I	Project 4 February 26, 2009

LangF VM Code Generator

Due: March 20, 2009

1 Introduction

The fourth project is to implement a simple code generator for LangF, which takes a high-level intermediate representation and produces an object file for interpretation by a virtual machine. The virtual machine, described below, is a stack-based interpreter. Code generation consists of two phases. The first phase translates the high-level intermediate representation (the Core IR introduced in the third project and discussed in class) to a lower-level intermediate representation in which data representations and variable locations have been determined. The second phase translates this low-level intermediate representation to bytecode instructions.

2 Representation&Location Intermediate Representation (*RepLoc IR*)

The first phase (provided in the project seed code) translates the high-level intermediate representation (the Core IR) into a lower-level intermediate representation (the RepLoc IR). This intermediate representation makes a number of implementation decisions explicit. The main differences from the higher-level intermediate representations (such as the AST and the Core IR) are that anonymous functions have been eliminated, types have been erased, data representations have been determined, and variable locations have been determined. In this section, we describe some aspects of the RepLoc IR. Nonetheless, you will need to study and understand the `structure RepLocIR : REPLOC_IR` module provided in the project seed code.

2.1 Anonymous Function Elimination

In LangF, one can introduce a function either anonymously, via the **`fn`** $Param_1 \cdots Param_n$ **`=>`** Exp expression syntax, or named, via the **`fun`** $varid_1\ Param_{1,1} \cdots Param_{1,m_1}$ **`:`** $Type_1$ **`=`** $Exp_1 \quad \cdots$ **`and`** $varid_n\ Param_{n,1} \cdots Param_{n,m_n}$ **`:`** $Type_n$ **`=`** Exp_n declaration syntax. The conversion from the AST to the Core IR makes some simplifications, one being that all functions take exactly one parameter. Thus, in the Core IR, one can introduce a function either anonymously, via expression syntax analogous to **`fn`** $Param$ **`=>`** Exp, or named, via declaration syntax analogous to **`fun`** $varid_1\ Param_1$ **`:`** $Type_1$ **`=`** $Exp_1 \quad \cdots$ **`and`** $varid_n\ Param_n$ **`:`** $Type_n$ **`=`** Exp_n.

The conversion from the Core IR to the RepLoc IR makes a further simplification: all functions are introduced via the declaration syntax. This is easily accomplished — we convert any anonymous function **`fn`** $Param$ **`=>`** Exp expression to the following expression:

$$\textbf{\texttt{let fun}}\ varid_f\ Param\ \textbf{\texttt{:}}\ Type_{Exp}\ \textbf{\texttt{=}}\ Exp\ \textbf{\texttt{in}}\ varid_f\ \textbf{\texttt{end}}$$

where $varid_f$ is a fresh variable identifier and $Type_{Exp}$ is the type of the expression Exp.

2.2 Type Erasure

As discussed in class, LangF can be interpreted and/or executed without tracking how type variables are instantiated. Recall the evaluation rules for type-function introduction and application:

$$\frac{}{\mathrm{E} \vdash \textbf{\texttt{fn [}}\alpha\textbf{\texttt{] =>}}\ Exp \ \Downarrow\ \mathrm{Clos}(\mathrm{E}, \textbf{\texttt{[}}\alpha\textbf{\texttt{]}}, Exp)} \qquad \frac{\mathrm{E} \vdash Exp_f \ \Downarrow\ \mathrm{Closure}(\mathrm{E}', \textbf{\texttt{[}}\alpha\textbf{\texttt{]}}, Exp) \quad \mathrm{E}' \vdash Exp \ \Downarrow\ Val}{\mathrm{E} \vdash Exp_f\ \textbf{\texttt{[}}Type_a\textbf{\texttt{]}} \ \Downarrow\ Val}$$

Notice that the argument type Type_a does not affect the evaluation of the type-function body *Exp*; however, also notice that the type-function body is not evaluated until the type-function is applied to a type. Similarly, recall that a **`datatype`** declaration has no effect on the environment used for subsequent evaluation:

$$\frac{}{\mathrm{E} \vdash \texttt{datatype} \ldots \Downarrow \{\}}$$

Thus, the conversion from the Core IR to the RepLoc IR discards all type information: type arguments in data-constructor expressions, **`datatype`** declarations, type arguments in data-constructor patterns, type annotations in variable parameters, variable patterns, and **`val`** and **`fun`** declarations. Type-variable parameters and type arguments in apply expressions are also eliminated, but, as noted above, the type-function body should not be evaluated until the type-function is applied to a type. Rather than being completely eliminated, type-variable parameters and type arguments in apply expressions are replaced by a dummy variable parameter and a dummy argument. That is, we convert any (Core IR) function declaration *varid* **`[`**α**`]`** **`:`** *Type* **`=`** *Exp* to the following (RepLoc IR) function declaration:

$$\mathit{varid}\ \texttt{(}\mathit{varid}_u\texttt{)}\ \texttt{=}\ \mathit{Exp}$$

where varid_u is a fresh variable identifier (and type annotations have been discarded) and we convert any (Core IR) application *Exp* **`[`***Type***`]`** to the following (RepLoc IR) application:

$$\mathit{Exp}\ \texttt{Unit}$$

where `Unit` is a nullary data constructor.

2.3 Data Representations

All LangF values are represented by a single 32-bit machine word. In many cases, this word is a pointer to a heap-allocated object, but it might also be an immediate value. Values that are represented as pointers (to heap-allocated objects) are called *boxed* values, while values that are represented as immediate integers are called *unboxed* values. As explained below, the values of a **`datatype`** may be both boxed and unboxed, in which case we describe the corresponding type-constructor type as having a *mixed* represntation. Since type variables can be instantiated to any type, they have a mixed representation. The `Integer` type is represented as an immediate integer, while the `String` type is represented as a pointer (to a heap-allocated string). Function types are also represented as pointers (to heap-allocated records) and are described in Section 4. Data constructors and **`datatype`** type-constructor types are the interesting case.

The conversion from the Core IR to the RepLoc IR chooses a very simple data representation for data constructors and **`datatype`** type-constructor types. Consider a **`datatype`** type constructor *tyconid* with the following dacon declarations:

$$\mathit{daconid}_0\ \texttt{\{}\ \mathit{Type}_{0,0}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_{0,k_0}\ \texttt{\}}\ \texttt{|}\ \cdots\ \texttt{|}\ \mathit{daconid}_n\ \texttt{\{}\ \mathit{Type}_{n,0}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Type}_{n,k_n}\ \texttt{\}}$$

Then the representation of dacon_i is determined by its number of arguments. If dacon_i is nullary (*i.e.*, $k_i = 0$), then dacon_i is represented by the immediate integer i; hence, it is an unboxed value. (This corresponds to `RepLoc.DaConRep.UnboxedTag i` in the project seed code.) If dacon_i is non-nullary (*i.e.*, $k_i > 0$), then dacon_i **`{`** $v_{i,0}$ **`,`** $\ldots$ **`,`** v_{i,k_i} **`}`** is represented by a heap-allocated record $\langle i, v_{i,0}, \cdots, v_{i,k_i} \rangle$ where the first element is the immediate integer i and the subsequent elements are the values $v_{i,0}, \cdots, v_{i,k_i}$; hence, it is a boxed value. (This corresponds to `RepLoc.DaConRep.TaggedBox i` in the project seed code.) If the **`datatype`** type constructor *tyconid* is comprised entirely of nullary data constructors, then it will have an unboxed representation. If the **`datatype`** type constructor *tyconid* is comprised entirely of non-nullary data constructors, then it will have a boxed representation. Otherwise, it will have a mixed representation. Applying this convention to the following datatypes:

```
datatype Unit = Unit
datatype Bool = False | True
datatype Option ['a] = None | Some {'a}
datatype List ['a] = Nil | Some {'a, List ['a]}
datatype Pair ['a, 'b] = Pair {'a, 'b}
```

results in the following representations:

Type constructor	Representation	Data constructor	Representation	DaConRep
`Unit`	unboxed	`Unit`	0	`UnboxedTag 0`
`Bool`	unboxed	`False`	0	`UnboxedTag 0`
		`True`	1	`UnboxedTag 1`
`Option`	mixed	`None`	0	`UnboxedTag 0`
		`Some {`v`}`	$\langle 1, v \rangle$	`TaggedBox 1`
`List`	mixed	`Nil`	0	`UnboxedTag 0`
		`Cons {`v_h`,`v_t`}`	$\langle 1, v_h, v_t \rangle$	`TaggedBox 1`
`Pair`	boxed	`Pair {`v_a`,`v_b`}`	$\langle 0, v_a, v_b \rangle$	`TaggedBox 0`

This convention is general-purpose, but has some inefficiencies in certain specific cases. We discuss a more efficient convention in Section 7.

2.4 Variable Locations

During the execution of a LangF program, the same variable may denote multiple different values, depending upon "where" the program is in its execution. Consider the following LangF program:

```
val zero = 0
val one = 1
val two = 2
fun fib (n : Integer) : Integer =
  case n <= one of
     True => one
   | False =>
       let
         val a = fib (n - one)
         val b = fib (n - two)
       in
         a + b
       end
  end
; fib (10 + zero)
```

Clearly, the variables `n`, `a`, and `b` take on different values at different times — for the execution of the function application `fib 10`, `n` is `10`, `a` is `55`, and `b` is `34`, while for the execution of the function application `fib 9` (executed as part of the execution of `fib 10`), `n` is `9`, `a` is `34`, and `b` is `21`.

For the tree interpretation of LangF programs developed in class, we used separate environments to keep track of the value of a variable. Saving an environment (in the function-introduction rule), replacing an environment (in the function-application rule), and extending an environment (in the function-application rule, the match-rule rule, and the let-expression rule) ensured that the proper value for a variable is available when the variable is evaluated.

Maintaining a dynamic environment is simple to implement (see, for example, the Core IR interpreter in the project seed code at `langfc-src/core-ir/interpret.sml`), but has some inefficiencies that can be eliminated when targetting a virtual machine (or a physical machine). In essence, we decide (once and for all) "where"

to find the value of a variable when at a particular program point. The "where" is called the variable's *location*. In the RepLoc IR, we distinguish four kinds of locations:

- Param: The variable is the current function's parameter.
- Local(i): The variable is the ith local variable of the function. Every application of the function will have its own copy of the local variables. Local variables are variables that are bound within the body of the function.
- Global(i): The variable is the ith global variable of the function. Every application of the function shares the same global variables, which correspond to the free variables (or *environment*) of the function. Global variables are variables that are bound outside the function.
- Self(f): The variable is the function named f from the function's group of mutually recursive functions.

In the LangF program above, within the body of the `fib` function, the variables have the following locations:

Variable	Location
`n`	Param
`a`	Local(0)
`b`	Local(1)
`one`	Global(0)
`two`	Global(1)
`fib`	Self

Similarly, in the program itself (that is, outside the body of the `fib` function), the variables have the following locations:

Variable	Location
`zero`	Local(0)
`one`	Local(1)
`two`	Local(2)
`fib`	Local(3)

The conversion from the Core IR to the RepLoc IR chooses a location for every variable. The location of a variable may vary depending upon where the variable is being accessed from (for example, the variables `one`, `two`, and `fib` are Local outside the body of the `fib` function, while they are Global and Self inside the body of the `fib` function).

For a group of mutually-recursive functions, the conversion from the Core IR to the RepLoc IR also records the list of variable locations (as accessed outside the functions) that are accessed as Global within the functions. These variables correspond to the free variables (or *environment*) of the functions. (See Section 4 for more details.) Finally, for each function and for the program itself, the conversion from the Core IR to the RepLoc IR records the maximum number of local variables.

The choice of variable locations used by the conversion from the Core IR to the RepLoc IR is general-purpose, but has some inefficiencies in certain specific cases. We discuss more efficient use of local variables in Section 7.

2.5 Example

To illustrate the RepLoc IR, Figure 1 gives the RepLoc IR program for the LangF program above.[1]

The `#(4)` at line 1 indicates that the program itself has four local variables (corresponding to `zero`, `one`, `two`, and `fib`). Lines 3–5 correspond to the **`val`** declarations of `zero`, `one`, and `two` in the LangF program.

Lines 6–18 correspond to the **`fun`** `fib (n: Integer) : Integer = ...` declaration in the LangF program. The `$(Local(1), Local(2))` at line 6 indicates that the first and second local variables (corresponding to `one` and `two`) should be saved (see Section 4) in order to be accessed as `Global(0)` and `Global(1)`

[1] You can save the RepLoc IR for a LangF program by executing the compiler with the command `./bin/langfc -Ckeep-convert-to-reploc=true` *`file`*`.lgf`.

```
1  #(4)
2  let
3     val Local(0) = 0
4     val Local(1) = 1
5     val Local(2) = 2
6     fun $(Local(1), Local(2))
7     and Local(3) =
8       fib__000 #(2) =>
9          (case !Lte (Param, Global(0)) of
10             True@UnboxedTag(1) => Global(0)
11           | False@UnboxedTag(0) =>
12                let
13                   val Local(0) = Self(fib__000) (!Sub (Param, Global(0)))
14                   val Local(1) = Self(fib__000) (!Sub (Param, Global(1)))
15                in
16                   !Add (Local(0), Local(1))
17                end
18          end)
19 in
20    Local(3) (!Add (10, Local(0)))
21 end
```

Figure 1: Sample RepLoc IR for Fibonacci program

within the body of the function. Line 7 indicates that the function will be accessed as `Local(3)` in the program itself, while Line 8 indicates that the function will be accessed as `Self(fib__000)` within the body of the function and that the function has two local variables (corresponding to `a` and `b`). Lines 9–18 correspond to the body of the function.

Note that the data constructors in the patterns at lines 10 and 11 have unboxed representations. The function argument `n` is accessed as `Param` at lines 9, 13, and 15, while the free variables `one` and `two` are accessed as `Global(0)` and `Global(1)` at lines 9, 10, 13, and 14. Lines 13 and 14 correspond to the **`val`** declarations of `a` and `b` in the LangF program; note that the recursive use of the function name within the function body is accessed as `Self(fib__000)`.

Finally, line 20 corresponds to the `fib (10 - zero)` expression in the LangF program. Outside the body of the `fib` is accessed as `Local(3)`.

3 Virtual Machine

In this section, we describe the virtual machine (VM) to be targetted by the code generator. The virtual machine is a stand-alone program that takes an object file and executes it. An object file consists of a sequence of bytecode instructions, a literal table that contains string constants, and a C function table that contains the names of runtime-system functions (that are used to implement services such as I/O).

3.1 Values

The VM supports four types of values: 31-bit tagged integers, 32-bit pointers to heap-allocated records of values or heap-allocated strings, 32-bit pointers to stack positions, and 32-bit pointers to bytecode instructions.

v	::=	i	tagged integer
	|	p	pointer to heap-allocated object
	|	$spos$	pointer to stack position
	|	$caddr$	pointer to bytecode instruction

An integer value i is represented by $2i+1$ in the VM (hence, is represented by a 32-bit word); this tagging is required to distinguish integers from pointers for the garbage collector. The VM takes care of tagging/untagging integers; the only impact of this representation on the VM code generator is that integer constants must be in the range -2^{30} to $2^{30}-1$, a syntactic restriction that is enforced by the type checker.

We use 32-bit word addresses for pointers to heap-allocated objects and pointers to stack positions, but use 8-bit byte address for pointers to bytecode instructions.

3.2 Configurations

The virtual machine can be described by the various components of internal state that are referenced and updated as it executes. There are three components of immutable state (loaded from the object file) and six components of mutable state. We collect these components together into a *configuration*:

$$\text{VM} \quad ::= \quad \langle \text{LitTbl}; \text{CFunTbl}; \text{Code}; ; \text{Heap}; \text{Stack}; \text{SP}; \text{FP}; \text{EP}; \text{PC} \rangle$$

The immutable state, loaded from the object file, is comprised of the literal table, the C function table, and the sequence of bytecode instructions.

LitTbl	::=	$\{n \mapsto$ **"***string***"**$, \cdots\}$	literal table
CFunTbl	::=	$\{n \mapsto$ *cfun*$, \cdots\}$	C function table
Code	::=	$\texttt{instr}_0 \cdots \texttt{instr}_{n-1}$	sequence of bytecode instructions

The mutable state is comprised of a heap of heap-allocated objects (records of values and strings), a stack of values, and four special registers. The stack pointer (SP) points to the top stack element (the stack grows towards lower addresses, so pushing an element on the stack decreases the stack pointer while popping an element from the stack increases the stack pointer). The frame pointer (FP) points to the base of the current stack-frame and is used to access the function argument and local variables. The environment pointer (EP) points to the current environment and is used to access global variables. The program counter (PC) points to the next bytecode instruction to execute.

Heap	::=	$\{p \mapsto \text{HeapObj}, \ldots\}$	heap of heap-allocated objects
HeapObj	::=	$\langle v_0, \ldots, v_{n-1} \rangle$	record of n values
	|	**"***string***"**	string
Stack	::=	$v \cdots v$	stack of values (rightmost value is the top stack element)
SP	::=	*spos*	stack pointer; always points to top stack element
FP	::=	*spos*	frame pointer
EP	::=	p	environment pointer
PC	::=	*caddr*	program counter

3.3 Instructions

We specify the semantics of instructions using the following notation:

$$\langle \text{H}; \cdots \overset{\text{FP}}{\downarrow} \cdots v_n \cdots \overset{\text{SP}}{\downarrow} v_0; \text{SP}; \text{FP}; \text{EP}; \text{PC} \rangle \Rightarrow \langle \text{H}'; \cdots \overset{\text{FP}'}{\downarrow} \cdots v'_m \cdots \overset{\text{SP}'}{\downarrow} v'_0; \text{SP}'; \text{FP}'; \text{EP}'; \text{PC}' \rangle$$
$$\text{Code}[\text{PC}] = \texttt{instr}$$

which means that the virtual machine transitions from one configuration to another when the (original) program counter points to the instruction **instr**; the transition takes a stack with $v_n \cdots v_0$ at the top and maps it to a stack with $v'_m \cdots v'_0$ at the top, leaves all other stack elements unchanged, and (possibly) changes the heap, the stack pointer, the frame pointer, the environment pointer, and the program counter. We omit the immutable state, since it never changes during the execution of the virtual machine. When necessary, we use the notation $\overset{\text{SP}}{\downarrow}$ and $\overset{\text{FP}}{\downarrow}$ to indicate the elements in the stack to which the stack pointer and frame pointer are pointing (in the above, SP is pointing to v_0).

In the following, we increment (resp. decrement) the stack pointer by 1 when popping (resp. pushing) a single value. Technically, since virtual machine values are 32-bit quantities, the increment (resp. decrement) should be by 4. Similarly, in the following, we increment the program counter by 1 when transfering control to the next instruction in the sequence of bytecode instructions. Technically, since different bytecode instructions have different encoding lengths, the increment should be by the length of the encoded bytecode instruction.

The instructions are organized by kind in the following description.

Arithmetic instructions

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 + i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{add}$
Pops the top two stack elements (which must be integers), adds them, and pushes the result (modulo 2^{30}).

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 - i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{sub}$
Pops the top two stack elements (which must be integers), subtracts them, and pushes the result (modulo 2^{30}).

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 \times i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{mul}$
Pops the top two stack elements (which must be integers), multiplies them, and pushes the result (modulo 2^{30}).

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1/i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{div}$
Pops the top two stack elements (which must be integers), divides them, and pushes the result (modulo 2^{30}). The result is undefined if i_2 equals 0.

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 \mod i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{mod}$
Pops the top two stack elements (which must be integers), divides them, and pushes the remainder (modulo 2^{30}). The result is undefined if i_2 equals 0.

$\langle \mathrm{H}; \cdots\ i; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (-i); \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{neg}$
Pops the top stack element (which must be an integer), negates it, and pushes the result (modulo 2^{30}).

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 = i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{equ}$
Pops the top two stack elements (which must be integers), compares them, and pushes 1 if i_1 equals i_2 and pushes 0 otherwise.

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 < i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{less}$
Pops the top two stack elements (which must be integers), compares them, and pushes 1 if i_1 is less than i_2 andpushes 0 otherwise.

$\langle \mathrm{H}; \cdots\ i_1\ i_2; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (i_1 \leq i_2); \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{lesseq}$
Pops the top two stack elements (which must be integers), compares them, and pushes 1 if i_1 is less than or equal to i_2 and pushes 0 otherwise.

$\langle \mathrm{H}; \cdots\ v; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (v = 0); \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{not}$
Pops the top stack element, compares it to zero, and pushes the 1 if v equals 0 and pushes 0 otherwise.

Stack instructions

$\langle \mathrm{H}; \cdots\ ; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ i; \mathrm{SP}-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{int(}i\texttt{)}$
Pushes the integer i.

$\langle \mathrm{H}; \cdots\ ; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H} \oplus \{p \mapsto \texttt{"}\mathit{string}_n\texttt{"}\}; \cdots\ p; \mathrm{SP}-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{literal(}n\texttt{)},\ p \notin \mathrm{dom}(\mathrm{H}),\ \mathit{LitTbl}[n] = \texttt{"}\mathit{string}_n\texttt{"}$
Heap allocates a string, initialized from the nth string literal, and pushes a pointer to the newly allocated object.

$\langle \mathrm{H}; \cdots ; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ \mathit{caddr}; \mathrm{SP} - 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{label(}l\texttt{)}, \ \mathrm{CAddrOf}(l) = \mathit{caddr}$

Pushes the code address named by the label l. Note that in the encoding of this instruction, the code address is specified as an offset from the program counter (PC).

$\langle \mathrm{H}; \cdots \ v_1 \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_0 \ v_1; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{swap}$

Swaps the top two stack stack elements.

$\langle \mathrm{H}; \cdots \ v_n \ v_{n-1} \ \cdots \ v_1 \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_0 \ v_{n-1} \ \cdots \ v_1 \ v_n; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{swap(}n\texttt{)}$

Swaps the top stack element with the nth element from the top of the stack. Note, the **`swap`** instruction is equivalent to **`swap(`**1**`)`**, but with a shorter instruction encoding.

$\langle \mathrm{H}; \cdots \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_0 \ v_0; \mathrm{SP} - 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{dup}$

Duplicates the top stack element.

$\langle \mathrm{H}; \cdots \ v_n \ v_{n-1} \ \cdots \ v_1 \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_n \ v_{n-1} \ \cdots \ v_1 \ v_0 \ v_n; \mathrm{SP} - 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{push(}n\texttt{)}$

Pushes the nth element from the top of the stack. Note, the **`dup`** instruction is equivalent to **`push(`**0**`)`**, but with a shorter instruction encoding.

$\langle \mathrm{H}; \cdots \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots ; \mathrm{SP} + 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{pop}$

Pops the top stack element.

$\langle \mathrm{H}; \cdots \ v_n \ v_{n-1} \ \cdots \ v_1 \ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots ; \mathrm{SP} + n; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{pop(}n\texttt{)}$

Pops the top n stack elements. Note, the **`pop`** instruction is equivalent to **`pop(`**0**`)`**, but with a shorter instruction encoding.

Heap instructions

$\langle \mathrm{H}; \cdots \ v_0 \ \cdots \ v_{n-1}; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H} \oplus \{p \mapsto \langle v_0, \ldots, v_{n-1} \rangle\}; \cdots \ p; \mathrm{SP} + n - 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{alloc(}n\texttt{)}, \ p \notin \mathrm{dom}(\mathrm{H})$

Heap allocates a record of n elements, initialized from the top n stack elements, and pushes a pointer to the newly allocated object.

$\langle \mathrm{H}; \cdots \ p; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_0 \ \cdots \ v_{n-1}; \mathrm{SP} + 1 - n; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{explode}, \ \mathrm{HP}[p] = \langle v_0, \cdots, v_{n-1} \rangle$

Pops a pointer to a heap-allocated record and pushes the record elements.

$\langle \mathrm{H}; \cdots \ p; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_i; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{select(}i\texttt{)}, \ \mathrm{H}[p] = \langle v_0, \cdots, v_{n-1} \rangle$

Pops a pointer to a heap-allocated record and pushes the ith record element. *Hint:* Use this instruction to implement pattern matching on a data constructor with a `TaggedBox` representation.

$\langle \mathrm{H}; \cdots \ p \ i; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots \ v_i; \mathrm{SP} + 1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{index}, \ \mathrm{H}[p] = \langle v_0, \cdots, v_{n-1} \rangle$

Pops an index i and a pointer to a heap-allocated record and pushes the ith record element. *NOTE:* This instruction should not be needed to complete the project (with or without extra credit).

$\langle \mathrm{H}; \cdots \ p \ i \ v_i'; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H} \oplus \{p \mapsto \langle v_0, \cdots, v_i', \cdots, v_{n-1} \rangle\}; \cdots ; \mathrm{SP} + 3; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} + 1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{update}, \ \mathrm{H}[p] = \langle v_0, \cdots, v_i, \cdots v_{n-1} \rangle$

Pops a value, an index i, and a pointer to a heap-allocated record and updates the ith record element with the value. *NOTE:* This instruction should not be needed to complete the project (with or without extra credit).

$\langle \mathrm{H}; \cdots\ v; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ (v \in \mathrm{dom}(\mathrm{H})); \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{boxed}$

Pops a value and pushes 1 if the value is a pointer to a heap-allocated object and pushes 0 otherwise. *Hint:* Use this instruction to implement pattern matching on a **`datatype`** type constructor with a mixed representation.

Environment-pointer instructions

$\langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ \mathrm{EP}; \mathrm{SP}-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{pushep}$

Pushes the environment pointer (EP).

$\langle \mathrm{H}; \cdots\ v; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots; \mathrm{SP}+1; \mathrm{FP}; v; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{popep}$

Pops the top stack element and stores it in the environment pointer.

$\langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ v; \mathrm{SP}-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{loadglobal(}i\texttt{)},\ \mathrm{H}[\mathrm{EP}] = \langle v_0, \cdots, v_{n-1}\rangle$

Interprets the environment pointer (EP) as a pointer to a heap-allocated record and pushes the ith record element.

Frame-pointer instructions

$\langle \mathrm{H}; \cdots\ v_2\ v_1\ \overset{\mathrm{FP}}{\downarrow}\ v_0\ v_{-1}\ v_{-2}\ \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_1\ \overset{\mathrm{FP}}{\downarrow}\ v_0\ v_{-1}\ \cdots\ v_i; \mathrm{SP}-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{loadlocal(}i\texttt{)}$

Fetches the value at the word addressed by $\mathrm{FP}+i$ and pushes the value. Note that a function's argument will be at offset 2, the function's return address will be at offset 0, and the local variables will start at offset -1.

$\langle \mathrm{H}; \cdots\ v_2\ v_1\ \overset{\mathrm{FP}}{\downarrow}\ v_0\ v_{-1}\ v_{-2}\ \cdots\ v'_i; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_{i+1}\ v'_i\ v_{i-1}\ \cdots; \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{storelocal(}i\texttt{)}$

Pops a value and stores it at the word addressed by $\mathrm{FP}+i$.

$\langle \mathrm{H}; \cdots\ \overset{\mathrm{SP}}{\downarrow}\ v_0; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_0\ \overset{\mathrm{SP}-1}{\downarrow}\ \mathrm{FP}\ w_0\ \cdots\ w_{n-1}; \mathrm{SP}-1-n; \mathrm{SP}-1; \mathrm{EP}; \mathrm{PC}\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{entry(}n\texttt{)}$

Pushes the frame pointer (FP), sets the frame pointer to the updated stack pointer ($\mathrm{SP}-1$, in terms of the original stack pointer SP), and pushes n uninitialized values.

Control-flow instructions

$\langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathit{caddr}\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{jmp(}l\texttt{)},\ \mathrm{CAddrOf}(l) = \mathit{caddr}$

Transfers control to the code address named by the label l. Note that in the encoding of this instruction, the code address is specified as an offset from the program counter (PC)

$\langle \mathrm{H}; \cdots\ v; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots; \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathit{caddr}\rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{jmpif(}l\texttt{)},\ v = 1,\ \mathrm{CAddrOf}(l) = \mathit{caddr}$

$\langle \mathrm{H}; \cdots\ v; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots; \mathrm{SP}+1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{jmpif(}l\texttt{)},\ v \neq 1$

Pops the top stack element, compares it to one, and transfers control to the code address named by the label l if v equals 1 and transfers control to $\mathrm{PC}+1$ otherwise.

$\langle \mathrm{H}; \cdots\ v_a\ \mathit{caddr}; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}\rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_a\ (\mathrm{PC}+1); \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathit{caddr}\rangle \qquad \mathrm{Code}[\mathrm{PC}] = \texttt{call}$

Pops the to-be-called function's code address, pushes the address of the next instruction (PC+1), and transfers control to the popped code address.

$\langle \mathrm{H}; \cdots\ v_a\ caddr_{ret} \overset{\mathrm{FP}}{\downarrow} spos_{fp}\ \cdots\ v_r; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_r; \mathrm{FP}+2; spos_{fp}; \mathrm{EP}; caddr_{ret} \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{ret}$

Returns from a function. Pops the result (v_r), sets the stack pointer to the frame pointer, pops the saved frame pointer ($spos_{fp}$) and sets the frame pointer to the saved frame pointer, pops the return code address ($caddr_{ret}$), pops the function argument (v_a), pushes the result, and transfers control to the return code address.

$\langle \mathrm{H}; \cdots\ v\ caddr_{ret} \overset{\mathrm{FP}}{\downarrow} spos_{fp}\ \cdots\ v_a\ caddr; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots\ v_a\ caddr_{ret}; \mathrm{FP}+1; spos_{fp}; \mathrm{EP}; caddr \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{tailcall}$

Tail calls a function. Pops the the to-be-called function's code address ($caddr$) and the to-be-called function's argument (v_a), pops the current frame (like the **ret** instruction), stores the argument, and transfers control to the popped code address. *NOTE:* This instruction should not be needed to complete the project, unless undertaking the extra credit portion of Section 7.3.

Miscellaneous instructions

$\langle \mathrm{H}; \cdots\ v_1\ \cdots\ v_m; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}'; \cdots\ v; \mathrm{SP}+m-1; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1 \rangle$
$\mathrm{Code}[\mathrm{PC}] = \texttt{ccall(}n\texttt{)},\ CFunTbl[n] = cfun_n,\ \mathrm{H}; cfun_n(v_1, \cdots, v_m) \Rightarrow \mathrm{H}'; v$

Calls the nth C function. The C function will pop its m arguments from the stack and push its result. The C function may allocate in the heap and may perform I/O.

$\langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC}+1 \rangle$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{nop}$

Performs no operation other than incrementing the program counter.

$\langle \mathrm{H}; \cdots; \mathrm{SP}; \mathrm{FP}; \mathrm{EP}; \mathrm{PC} \rangle \Rightarrow \square$ $\quad \mathrm{Code}[\mathrm{PC}] = \texttt{halt}$

Halts the virtual machine.

3.4 Runtime functions

The virtual machine provides the **ccall** instruction to invoke C functions. C functions take their arguments from the stack and return their result on the stack. The **ccall** instruction specifies the C function by an index into the C function table.

The virtual machine provides the following runtime system functions.

$\mathrm{H}; \texttt{"VM\_Argc"}(v) \Rightarrow \mathrm{H}; i$

Ignores its argument and returns the number of arguments (including the name of the object file) passed to the virtual machine.

$\mathrm{H}; \texttt{"VM\_Arg"}(i) \Rightarrow \mathrm{H} \oplus \{p \mapsto \texttt{"}arg_i\texttt{"}\}; p$ $\quad p \notin \mathrm{dom}(\mathrm{H})$

Heap allocates a string, initialized to the ith argument passed to the virtual machine, and returns a pointer to the newly allocated object. The 0th argument is the name of the object file.

$\mathrm{H}; \texttt{"VM\_Print"}(fid, p) \Rightarrow \mathrm{H}; 0$ $\quad p \in \mathrm{dom}(\mathrm{H}),\ \mathrm{H}[p] = \texttt{"}string\texttt{"}$

Prints the string pointed to by p to the file specified by *fid* and return the `Unit` value (represented by the tagged integer value 0). Use 0 for the standard output.

$\mathrm{H}; \texttt{"VM\_Size"}(p) \Rightarrow \mathrm{H}; n$ $\quad p \in \mathrm{dom}(\mathrm{H}),\ \mathrm{H}[p] = \texttt{"}string\texttt{"},\ |string| = n$

Returns the size of the string pointed to by p.

$\mathrm{H}; \texttt{"VM\_Concat"}(p_1, p_2) \Rightarrow \mathrm{H} \oplus \{p \mapsto \texttt{"}string_1 string_2\texttt{"}\}; p$
$p_1 \in \mathrm{dom}(\mathrm{H}),\ \mathrm{H}[p_1] = \texttt{"}string_1\texttt{"},\ p_2 \in \mathrm{dom}(\mathrm{H}),\ \mathrm{H}[p_2] = \texttt{"}string_2\texttt{"},\ p \notin \mathrm{dom}(\mathrm{H})$

Heap allocates a string, initialized to the concatenation of the strings pointed to by p_1 and p_2, and returns a pointer to the newly allocated object.

H; **"VM_Sub"**$(p, i) \Rightarrow \mathrm{H}; c$

$$p \in \mathrm{dom}(\mathrm{H}),\ \mathrm{H}[p] = \texttt{"}string\texttt{"},\ \mathrm{string}[i] = c$$

Returns the integer code of the character in the string pointed to by p at the index i.

If any of these functions encounters an error (*e.g.*, index out of bounds), then the virtual machine halts (with an error message).

4 Implementing Functions

LangF supports higher-order functions, which requires representing functions as heap-allocated objects. For example, consider the function:

```
val add : Integer -> Integer -> Integer =
  fn (x: Integer) => fn (y: Integer) =>
    x + y
```

When applied to an integer argument i, the `add` function returns a function that will add the integer i to its argument. The representation of the result of `add` must include the value of i. In general, the representation of a function will include the free variables of the function stored in a heap-allocated record, called the function's *environment*. In this example, the environment of the function `add` has no elements and the environment of the function returned by `add` has a single element (the value of the argument `x`). The representation of a function must also contain the address of the function's code. We could store this address in the environment record, but for reasons that we explain below, we instead represent a function as a two-element record of its code address and a pointer to its environment. This record is the *closure* representation of the function. Thus, the generated code for `add` might look something like the following:

Label	Instruction	Comment
`add:`	**`entry(`**0**`)`**	
	`loadlocal(`2**`)`**	push argument variable `x`
	`alloc(`1**`)`**	allocate environment
	`label(``f`**`)`**	push code address `f`
	`swap`	swap environment pointer and code pointer
	`alloc(`2**`)`**	allocate closure (code ptr, env. ptr)
	`ret`	return closure
`f:`	**`entry(`**0**`)`**	
	`loadglobal(`0**`)`**	push global variable `x` from environment
	`loadlocal(`2**`)`**	push argument variable `y`
	`add`	
	`ret`	return sum

Things are only a bit more complicated with mutually recursive functions. All functions in a group of mutually recursive functions share the same common environment, which will include all of the free variables of all of the functions (except the variables denoting the functions themselves). For example, consider the following pair of mutually recursive functions:

```
fun f (a: Integer) : Integer = if a < 0 then 1 else g (a + x)
and g (b: Integer) : Integer = f(b * y + z)
```

In this case, the environment of `f` and `g` will have three values: `x`, `y`, and `z`.

4.1 Calling Convention

A RepLoc IR function application "$e_1\ e_2$" is implemented using a four part protocol. (Remember, all LangF function applications of the form *Exp* **[***Type***]** have been replaced by function applications of the form *Exp* `Unit` in the conversion to the RepLoc IR.)

1. The caller pushes its own environment pointer (in order to restore it after the called function returns) using the **pushep** instruction and then evaluates the function and argument expressions from left to right, leaving the results on the stack. Then a **swap** instruction is used to move the function closure to the top of the stack, which is then exploded into its code-pointer and environment-pointer components. The environment pointer is loaded into the EP register using the **popep** instruction. Then the function is called using the **call** instruction, which has the effect of pushing the address of the following instruction on the stack. This protocol is realized by the following sequence of instructions:

Instruction	Stack	Comment
pushep	$\cdots\ ep_{caller}$	push caller's environment pointer
evaluate e_1	$\cdots\ ep_{caller}\ \langle f, ep_{callee}\rangle$	evaluate function (to a ptr to a code-ptr/env-ptr record)
evaluate e_2	$\cdots\ ep_{caller}\ \langle f, ep_{callee}\rangle\ arg$	evaluate argument
swap	$\cdots\ ep_{caller}\ arg\ \langle f, ep_{callee}\rangle$	swap the closure and argument values
explode	$\cdots\ ep_{caller}\ arg\ f\ ep_{callee}$	pop the closure and push the code ptr and env ptr
popep	$\cdots\ ep_{caller}\ arg\ f$	pop the callee's environment pointer into EP
call	$\cdots\ ep_{caller}\ arg\ raddr_{caller}$	call the function

2. The first instruction in the function is an **entry** instruction, which pushes the caller's frame pointer, sets the new frame pointer to the top of the stack, and then allocates space for n local variables.

Instruction	Stack	Comment
	$\cdots\ ep_{caller}\ arg\ raddr_{caller}$	stack on function entry
entry(n**)**	$\cdots\ ep_{caller}\ arg\ raddr_{caller}\ \overset{\text{FP}}{\downarrow}\ fp_{caller}\ w_0 \cdots\ w_{n-1}$	initialize callee's stack frame

3. When the callee is finished and the return result is on the top of the stack, it executes a **ret** instruction, which pops the result, deallocates the space for local variables, restores the caller's frame pointer, pops the return address and the function argument, pushes the result, and transfers control to the return address.

Instruction	Stack	Comment
	$\cdots\ ep_{caller}\ arg\ raddr_{caller}\ \overset{\text{FP}}{\downarrow}\ fp_{caller}\ w_0 \cdots\ w_{n-1}\ res$	stack on function exit
ret	$\cdots\ ep_{caller}\ res$	return to caller

4. When control is returned to the address following the **call** instruction in the caller, a **swap** instruction is used to move the caller's environment pointer to the top of the stack and a **popep** instruction is used load it into the EP register.

Instruction	Stack	Comment
	$\cdots\ ep_{caller}\ res$	stack on function return
swap	$\cdots\ res\ ep_{caller}$	swap the caller's environment pointer and the function result
popep	$\cdots\ res$	pop the caller's environment pointer into EP

This protocol is general-purpose, but has some inefficiencies in certain specific cases. We discuss variations on this protocol in Section 7.

4.2 Example

To illustrate the implementation of functions, consider the following LangF program:

```
val zero = 0
val one = 1
fun fact (n: Integer): Integer =
  if n <= zero then one else n * (fact (n - one))
; fact 5
```

The `fact` function has two variables in its environment (`zero` and `one`), one argument (`n`), and no local variables. The argument `n` will be located at offset 2 from the frame pointer, while `zero` and `one` will be located at offset 0 and offset 1 from the environment pointer. The program has three local variables (`zero`, `one`, and `fact`); `zero` will be located at offset -1 from the frame pointer, `one` will be located at offset -2 from the frame pointer, `fact` will be located at offset -3 from the frame pointer. One possible sequence of bytecode instructions for this program is given in Figure 2.

5 Code Generation API

The code generation API is organized into three modules. The `CodeStream: CODE_STREAM` module implements code streams, which are an abstraction of the generated object file. The `Label: LABEL` module implements labels for naming code addresses. The `Instruction: INSTRUCTION` module implements an abstract type of virtual machine instructions.

5.1 Code Streams

The `CodeStream: CODE_STREAM` module provides a container to collect the instructions emitted by your code generator. The `CodeStream.new` function creates an empty code stream. The `CodeStream.emit` function saves an instruction at the end of the code stream. The `CodeStream.string` and `CodeStream.c_function` functions registers string literals and C functions and returns the corresponding index to be used with the **`literal`** and **`ccall`** instructions.

5.2 Labels

The `Label: LABEL` module provides an abstract type of label that is used to represent code locations. The `CodeStream.defineLabel` function associates a label with the current position in the code stream. The control-flow instructions take a label as an argument and there is an instruction for pushing the value of a label onto the stack, which is required to create closures.

5.3 Instructions

The `Instruction: INSTRUCTION` module provides an abstract type that represents virtual machine bytecode instructions. For those instructions that take arguments, the module provides constructor functions and for those instructions without arguments, the module provides abstract values.

6 Requirements

You should implement a virtual machine code generator for LangF. Your implementation should include (at least) the following module:

```
structure VMCodeGenerator : VMCODE_GENERATOR
```

Label	Instruction	Comment
`_main:`	**`entry(`**3**`)`**	initialize stack frame with 3 local variables
	`int(`0**`)`**	
	`storelocal(`-1**`)`**	store local variable `zero`
	`int(`1**`)`**	
	`storelocal(`-2**`)`**	store local variable `one`
	`loadlocal(`-1**`)`**	load local variable `zero`
	`loadlocal(`-2**`)`**	load local variable `one`
	`alloc(`2**`)`**	allocate `fact`'s environment
	`label(``fact`**`)`**	
	`swap`	
	`alloc(`2**`)`**	allocate `fact`'s closure
	`storelocal(`-3**`)`**	store local variable `fact`
	`nop`	begin function call `fact 5`
	`pushep`	push `_main`'s env ptr
	`loadlocal(`-3**`)`**	evaluate function; load local variable `fact`
	`int(`5**`)`**	evaluate argument
	`swap`	swap the closure and argument values
	`explode`	
	`popep`	set EP to callee's environment
	`call`	
	`swap`	swap `_main`'s env ptr and result
	`popep`	restore EP to `_main`'s env ptr
	`halt`	
`fact:`	**`entry(`**0**`)`**	initialize stack frame with 0 local variables
	`loadlocal(`2**`)`**	load function parameter `n`
	`loadglobal(`0**`)`**	load environment variable `zero`
	`lesseq`	compute `n <= zero`
	`jmpif(``L1`**`)`**	
	`loadlocal(`2**`)`**	load function parameter `n`
	`nop`	begin function call `fact (n - 1)`
	`pushep`	
	`label(``fact`**`)`**	evaluate function; load self variable `fact`
	`pushep`	evaluate function; load env ptr for self variable `fact`
	`alloc(`2**`)`**	evaluate function; allocate self variable `fact`'s closure
	`loadlocal(`2**`)`**	evaluate argument; load function parameter `n`
	`loadglobal(`1**`)`**	evaluate argument; load environment variable `one`
	`sub`	evaluate argument; compute `n - one`
	`swap`	swap the closure and argument values
	`explode`	
	`popep`	set EP to callee's environment
	`call`	
	`swap`	swap `fact`'s env ptr and result
	`popep`	restore EP to `fact`'s env ptr
	`mul`	compute `n * (fact (n - one))`
	`jmp(``L2`**`)`**	
`L1:`	**`loadglobal(`**1**`)`**	load environment variable `one`
`L2:`	**`ret`**	

Figure 2: Sample VM code for factorial program

The VMCODE_GENERATOR signature is as follows:

```
signature VMCODE_GENERATOR =
sig
   val codeGen : ErrorStream.t *
                 RepLocIR.Prog.t ->
                 CodeStream.t
end
```

The **structure** RepLocIR : REPLOC_IR and **structure** CodeStream : CODE_STREAM modules are provided in the seed code; the REPLOC_IR signature implementation is at langfc-src/reploc-ir/reploc-ir.sig; the RepLocIR structure implementation is at langfc-src/reploc-ir/reploc-ir.sml; the CODE_STREAM signature implementation is at langfc-src/code-stream/code-stream.sig; and the CodeStream structure implementation is at langfc-src/code-stream/code-stream.sml.

Note that your implementation of VMCodeGenerator.codeGen should call CodeStream.new to create a code stream in which to emit instructions and return this code stream. Your implementation *should not* call CodeStream.finish — writing the code stream to an object file is handled by the top-level driver (langfc-src/driver/main.sml).

6.1 Errors

None! A program which has been verified by the front-end (scanner, parser, and type checker) should compile and generate an object file without errors.

7 Extra Credit (10pts)

As noted throughout this document, various choices, conventions, and protocols are general-purpose, but have some inefficiencies in certain specific cases. For extra credit, you may attempt to address some of these inefficienies. If you do so, please modify the *cnetid*-proj/project4/README file to list which improvements you have undertaken and a few sentences about how you addressed the inefficiencies. It should be clear from reading the following sections that these are only a small number of the possible improvements one can make. Take CMSC22620 to learn more!!

Note: Make sure that you have a completely working virtual machine code generator before undertaking these improvments.

7.1 Improve Data Representations: DaCon Representations (2pts)

The data representation for data constructors given in Section 2.3 (and used by the RepLocIRConverter: REPLOC_IR_CONVERTER module provided in the project seed code) assigns a unique tag to every data constructor. However, some data constructors do not need a tag. Consider the Pair data constructor; it is the one and only data constructor for the Pair type constructor. Any pattern match on a value of Pair type must match the Pair data constructor, so there is no need to check the tag. Similarly, consider the Nil and Cons data constructors. Any pattern match on an unboxed value of List type must match the Nil data constructor while any pattern match on a boxed value of List type must match the Cons data constructor; again, there is no need to check the tag.

Consider a **datatype** type constructor *tyconid* with the following dacon declarations:

$$\begin{aligned} &\mathit{daconid}_0 \texttt{ | } \cdots \texttt{ | } \mathit{daconid}_{n-1} \\ &\texttt{ | } \mathit{daconid}'_0 \texttt{ \{ } \mathit{Type}_{0,0} \texttt{ , } \cdots \texttt{ , } \mathit{Type}_{0,k_0} \texttt{ \} | } \cdots \texttt{ | } \mathit{daconid}'_{m-1} \texttt{ \{ } \mathit{Type}_{m-1,0} \texttt{ , } \cdots \texttt{ , } \mathit{Type}_{m-1,k_{m-1}} \texttt{ \}} \end{aligned}$$

where we have separated the n nullary data constructors ($daconid_0, \cdots, daconid_{n-1}$) from the m non-nullary data constructors ($daconid'_0, \cdots, daconid'_{m-1}$). The following table gives a more efficient representation for the data constructors and the type constructor, based on the number of nullary and non-nullary data constructors, the number of arguments to non-nullary data constructors, and the representation of the arguments to non-nullary data constructors.

n	m	$dacon_i$		$dacon'_j$ {$v_0, \ldots, v_{k_j}$}			$tycon$
> 0	0	i	`UnboxedTag i`		n.a.	n.a.	unboxed
0	1	n.a.	n.a.	if $k_0 = 1$	v_0	`Transparent`	$Type_{0,0}$'s representation
0	1	n.a.	n.a.	if $k_0 > 1$	$\langle v_0, \cdots, v_{k_j} \rangle$	`Boxed`	boxed
> 0	1	i	`UnboxedTag i`	if $k_0 = 1$ and $Type_{0,0}$ boxed	v_0	`Transparent`	mixed
> 0	1	i	`UnboxedTag i`	if $k_0 > 1$ or $Type_{0,0}$ not boxed	$\langle v_0, \cdots, v_{k_j} \rangle$	`Boxed`	mixed
≥ 0	> 1	i	`UnboxedTag i`		$\langle j, v_0, \cdots, v_{k_j} \rangle$	`TaggedBox j`	mixed

Applying this convention to the datatypes from Section 2.3 results in the following representations:

Type constructor	**Representation**	**Data constructor**	**Representation**	`DaConRep`
`Unit`	unboxed	`Unit`	0	`Tagged 0`
`Bool`	unboxed	`False`	0	`Tagged 0`
		`True`	1	`Tagged 1`
`Option`	mixed	`None`	0	`Tagged 0`
		`Some {`v`}`	$\langle v \rangle$	`Boxed`
`List`	mixed	`Nil`	0	`Tagged 0`
		`Cons {`v_h`,`v_t`}`	$\langle v_h, v_t \rangle$	`Boxed`
`Pair`	boxed	`Pair {`v_a`,`v_b`}`	$\langle v_a, v_b \rangle$	`Boxed`

7.2 Improve Calling Convention: Self Application (2pts)

The calling convention for a function application given in Section 4.1 always evaluates the function expression to a closure (a pointer to a record of a code-pointer and an environment-pointer). However, if the function expression is a variable accessed as $\mathrm{Self}(f)$ (*i.e.*, it is a recursive call of the function or a call of another function in the same mutually-recursive group), then we already know that the code-pointer will be the code address of the label f and the environment pointer will be the same as the environment pointer of the caller function (hence, already in the EP register). Furthermore, there is no need to save and restore the EP register of the caller function.

Figure 3 gives an improved sequence of bytecode instructions for the `fact` function from Figure 2.

7.3 Improve Calling Convention: Tail Calls (2pts)

If a function application appears in *tail position* (*i.e.*, as the last action a function performs before returning), then, according to the calling convention of Section 4.1, the generated code will always look like the following:

Label	**Instruction**	**Comment**
	`pushep`	save caller's env ptr
	⋮	
	`call`	
	`swap`	swap caller's env ptr and result
	`popep`	restore EP to caller's env ptr
	`ret`	

(with some minor variations if the calling-convention improvement of Section 7.2 has been adopted). Note that the caller function never accesses its parameter or its local variables after the callee function returns; hence, the caller's stack frame wastes space during the execution of the call. The virtual machine has a special **`tailcall`** instruction that discards the caller's stack frame and does not push a return address.

Label	Instruction	Comment
`fact:`	**`entry(`**0**`)`**	initialize stack frame with 0 local variables
	`loadlocal(`2**`)`**	load function parameter `n`
	`loadglobal(`0**`)`**	load environment variable `zero`
	`lesseq`	compute `n <= zero`
	`jmpif(``L1`**`)`**	
	`loadlocal(`2**`)`**	load function parameter `n`
	`nop`	begin function call `fact (n - 1)`
	`loadlocal(`2**`)`**	evaluate argument; load function parameter `n`
	`loadglobal(`1**`)`**	evaluate argument; load environment variable `one`
	`sub`	evaluate argument; compute `n - one`
	`label(``fact`**`)`**	evaluate function; load self variable `fact`
	`call`	
	`mul`	compute `n * (fact (n - one))`
	`jmp(``L2`**`)`**	
`L1:`	**`loadglobal(`**1**`)`**	load environment variable `one`
`L2:`	**`ret`**	

Figure 3: Improved VM code for factorial program

Tail calls should be used to implement looping in functional languages like LangF. For example, consider the mutually-recursive even/odd functions:

```
1  fun even (x:Integer) : Bool =
2     case x < 0 of
3        True => even (~x)
4      | False => case x == 0 of
5                    True => True
6                  | False => odd (x - 1)
7                 end
8     end
9  and odd (x:Integer) : Bool =
10    case x < 0 of
11       True => odd (~x)
12     | False => case x == 0 of
13                   True => False
14                 | False => even (x - one)
15                end
16    end
```

Note that there are a tail calls at lines 3, 6, 11, and 14.

7.4 Improve Calling Convention: Caller Env Ptr (2pts)

The calling convention for a function application given in Section 4.1 always saves and restores the EP register of the caller function. However, the environment pointer of a function never changes during the execution of the function. Therefore it can be more efficient to save the EP register once into the caller function's stack frame, and then restore it from the stack frame. We can adapt the general-purpose protocol from Section 4.1 to reflect this more efficient protocol:

1. The caller *does not* push its own environment pointer; the rest of the first part of the protocol remains the same.

Instruction	Stack	Comment
evaluate e_1	$\cdots\ \langle f, ep_{callee}\rangle$	evaluate function (to a ptr to a code-ptr/env-ptr record)
evaluate e_2	$\cdots\ \langle f, ep_{callee}\rangle\ arg$	evaluate argument
swap	$\cdots\ arg\ \langle f, ep_{callee}\rangle$	swap the closure and argument values
explode	$\cdots\ arg\ f\ ep_{callee}$	pop the closure and push the code ptr and env ptr
popep	$\cdots\ arg\ f$	pop the callee's environment pointer into EP
call	$\cdots\ arg\ raddr_{caller}$	call the function

2. The first instruction in the function is an **entry** instruction, which pushes the caller's frame pointer, sets the new frame pointer to the top of the stack, and then allocates space for the function's environment pointer and n local variables. The function's environment pointer is saved to $\mathrm{FP}-1$. Note that the function's local variables will now start at offset -2 from the FP register.

Instruction	Stack	Comment
	$\cdots\ arg\ raddr_{caller}$	stack on function entry
entry($n+1$**)**	$\cdots\ arg\ raddr_{caller}\ \overset{\mathrm{FP}}{\downarrow}\ fp_{caller}\ w\ w_0 \cdots\ w_{n-1}$	initialize callee's stack frame
pushep	$\cdots\ arg\ raddr_{caller}\ \overset{\mathrm{FP}}{\downarrow}\ fp_{caller}\ w\ w_0 \cdots\ w_{n-1}\ ep_{callee}$	push callee's environment pointer
storelocal(-1**)**	$\cdots\ arg\ raddr_{caller}\ \overset{\mathrm{FP}}{\downarrow}\ fp_{caller}\ ep_{callee}\ w_0 \cdots\ w_{n-1}$	save callee's environment pointer to $\mathrm{FP}-1$

3. The third part of the protocol remains the same.

Instruction	Stack	Comment
	$\cdots\ arg\ raddr_{caller}\ \overset{\mathrm{FP}}{\downarrow}\ fp_{caller}\ ep_{callee}\ w_0 \cdots\ w_{n-1}\ res$	stack on function exit
ret	$\cdots\ res$	return to caller

4. When control is returned to the address following the **call** instruction in the caller, a **loadlocal(**-1**)** instruction is used to move the caller's environment pointer to the top of the stack and a **popep** instruction is used load it into the EP register.

Instruction	Stack	Comment
	$\cdots\ res$	stack on function return
loadlocal(-1**)**	$\cdots\ res$	fetch the caller's environment pointer
popep	$\cdots\ res$	pop the caller's environment pointer into EP

Note that this revised protocol removes one instruction from the first part and adds two instructions to the second part. However, the second part appears in the code stream once per function declaration, while the first part appears in the code stream once per function application. Hence, there is a net savings in code size if one applies this revised protocol to functions with bodies that include at least two function applications. There is also a savings in stack usage for nested function applications. Consider the expression `f (g (h z))`. Using the original calling-convention protocol, there will be three copies of the caller's environment pointer on the stack when calling `h`. Using this revised calling-convention protocol, there will be only one copy of the caller's environment pointer on the stack.

Although this revised calling-convention protocol reduces code size for functions with bodies that include at least two function applications, if the function only makes self applications or tail calls, then there is never a need to restore the function's environment pointer after a call (assuming the improved calling convention for self applications described in Section 7.2 and the improved calling convention for tail calls described in Section 7.3) and, hence, never a need to save it in the first place. For such functions, it is better to stick with the original calling-convention protocol.

Even for a function with a body that includes at least two function applications (that are neither self applications or tail calls), there may be control-flow paths on which there are no function applications. Consider the following LangF function:

```
fun foo (x: Integer) : Integer =
  case x < 0 of
     True => 0
   | False => let
                 val a = f x
                 val b = g (x + a)
                 val c = h (a * b)
              in
                 a + b + c
              end
  end
```

If `foo` is applied to a negative number, then it wasn't necessary to save the environment pointer at the beginning of the function. It would be more efficient (in terms of the number of virtual machine instructions executed) to only save the environment pointer in the stack frame when taking the `False` branch. On the other hand, consider the following LangF function:

```
datatype T = A | B {Integer, Integer}
fun foo (x: T) : Integer =
  case T of
     A => 0
   | B {y, z} => let
                   val a = if y < 0 then 0 else f z
                   val b = if z < 0 then 0 else g y
                   val c = if y == z then 1 else h (a * b)
                 in
                    a + b + c
                 end
  end
```

For this function, when should the environment pointer be saved to the stack frame? to minimize code size? to minimize instructions executed?

Finally, if a function has an empty environment (*i.e.*,, it has no variables accessed as Global), then its environment pointer is never needed and there is no need to save and/or restore the environment pointer, regardless of the number or kinds of applications in the body of the function.

7.5 Improve Variable Locations: Reuse Local Variables (2pts)

The algorithm for deciding variable locations used by the `RepLocIRConverter: REPLOC_IR_CONVERTER` module provided in the project seed code chooses a local variable slot for any variable bound within the body of a function and reserves that local variable slot for the entire lexical scope of the variable. Thus, a variable may consume a local variable slot even though it will never be used again. This is inefficient, because it unnecessarily increases the size of the function's stack frame.

For example, consider the following LangF function:

```
fun foo (x: Integer) : Integer =
  case x < 0 of
     True => let
               val a = g x x
               val b = g a a
               val c = g b b
               val d = g c c
               val e = g d d
             in
               e + x
             end
   | False => let
                val n = h x x
                val o = h n n
                val p = h o o
                val q = h p p
                val r = h q q
              in
                r + x
              end
  end
```

The RepLoc IR function (as produced by the Core IR to RepLoc IR conversion implemented in the project seed code) for this LangF function is as follows:

```
fun $(Local(0), Local(1))
and Local(2) =
  foo__004 #(5) =>
     (case !Lt (Param, 0) of
         True@UnboxedTag(1) =>
            let
               val Local(0) = (Global(0) Param) Param
               val Local(1) = (Global(0) Local(0)) Local(0)
               val Local(2) = (Global(0) Local(1)) Local(1)
               val Local(3) = (Global(0) Local(2)) Local(2)
               val Local(4) = (Global(0) Local(3)) Local(3)
            in
               !Add (Local(4), Param)
            end
       | False@UnboxedTag(0) =>
            let
               val Local(0) = (Global(1) Param) Param
               val Local(1) = (Global(1) Local(0)) Local(0)
               val Local(2) = (Global(1) Local(1)) Local(1)
               val Local(3) = (Global(1) Local(2)) Local(2)
               val Local(4) = (Global(1) Local(3)) Local(3)
            in
               !Add (Local(4), Param)
            end
      end)
```

This RepLoc IR function has five local variables. Note that the RepLoc IR function uses the same local variables in the `True` match rule and the `False` match rule. On the other hand, note that the RepLoc IR function does not reuse local variables within a match rule.

Since each **`val`** bound variable is only used to compute the next **`val`** bound variable or the function result, a single local variable would suffice. That is, the LangF function above could be converted to the following RepLoc IR

function:

```
fun $(Local(0), Local(1))
and Local(2) =
  foo__004 #(1) =>
     (case !Lt (Param, 0) of
         True@UnboxedTag(1) =>
            let
               val Local(0) = (Global(0) Param) Param
               val Local(0) = (Global(0) Local(0)) Local(0)
               val Local(0) = (Global(0) Local(0)) Local(0)
               val Local(0) = (Global(0) Local(0)) Local(0)
               val Local(0) = (Global(0) Local(0)) Local(0)
            in
               !Add (Local(0), Param)
            end
       | False@UnboxedTag(0) =>
            let
               val Local(0) = (Global(1) Param) Param
               val Local(0) = (Global(1) Local(0)) Local(0)
               val Local(0) = (Global(1) Local(0)) Local(0)
               val Local(0) = (Global(1) Local(0)) Local(0)
               val Local(0) = (Global(1) Local(0)) Local(0)
            in
               !Add (Local(0), Param)
            end
      end)
```

(Note that local variables in a RepLoc IR function are like variables in C — they can be assigned to more than once.)

If a variable bound in the body of a function is not free in an sub-expression in the body of the function, then it would appear that the local variable slot reserved for the bound variable may be reused in the sub-expression. However, consider the following LangF function:

```
fun foo (x: Integer) : Integer =
  let
    val a = g x x
  in
    (let val b = h x x in g b b end)
    + (h a a)
  end
```

Although `a` is not free in the sub-expression **`let val`** `b = h x x` **`in`** `g b b` **`end`**, `b` cannot reuse the local variable slot reserved for `a`. If `b` were to use the same local variable slot as `a`, then the evaluation of `h a a` would actually apply `h` to `b`'s value.

8 GForge and Submission

Sources for Project 4 have been (or will shortly be) committed to your repository in the `project4` sub-directory. You will need to *update* your local copy, by running the command:

```
svn update
```

from the *`cnetid`*`-proj` directory.

We will collect projects from the SVN repositories at 10pm on Friday, March 20; make sure that you have committed your final version before then. To do so, run the command:

```
svn commit
```

from the *cnetid*-proj directory.

9 Hints

- Start early!
- Study the interfaces. You will need to be familiar with the types and operations in the REPLOC_IR, CODE_STREAM, INSTRUCTION, and LABEL signatures.
- Avoid the temptation of premature optimization. That is, first implement a very simple code generator. Such a code generator might unecessarily push and pop some values, compute values in the "wrong" order (for example, in Figure 2 there is a **swap** before the allocation of fact's closure that could be avoided by putting the **label(**fact**)** instruction before loading values for and allocating the environment), or jump more often than necessary (again, in Figure 2 the **jmp(**L2**)** instruction could be replaced by a **ret** instruction).
- To complete the assignment, you should only need to make changes to the *cnetid*-proj/project4/langfc-src/vmcode-generator/vmcode-generator.sml file. If you undertake the extra credit portions dealing with data representations (Section 7.1) and variable locations (Section 7.5), then you should only additionally need to make changes to the *cnetid*-proj/project4/langfc-src/reploc-ir/convert.sml file.
- Executing the compiler (from the *cnetid*-proj/project4 directory) with the command

  ./bin/langfc *file*.lgf

  will produce a *file*.bin file that can be interpreted by the virtual machine. Executing the virtual machine (from the *cnetid*-proj/project4 directory) with the command

  ./bin/vm *file*.bin $arg_1 \cdots arg_n$

  will execute the program.
- Executing the compiler (from the *cnetid*-proj/project4 directory) with the command

  ./bin/langfc -Cinterpret-core=true -Cargs-interpret-core=$arg_1,\cdots,arg_n$ *file*.lgf

  will interpret the Core IR representation of the program during the compilation. Use this control to check the expected behavior of a LangF program.
- Executing the compiler (from the *cnetid*-proj/project4 directory) with the command

  ./bin/langfc -Ckeep-convert-to-reploc=true -Ckeep-vmcode-generate=true *file*.lgf

  will produce a *file*.convert-to-reploc.reploc file that contains the Representation&Location intermediate representation (RepLoc IR) of the program (the input to the vm code generator) and a *file*.vmcode-generate.vmcode file that contains a (readable) text version of the object file returned by the code generator. Use these controls and their outputs to check that your code generartor is working as expected.

Document history

March 1, 2009 Original version