CMSC 22610	**Implementation**	**Project 4**
Winter 2005	**of**	**March 1**
	Computer Languages	

MLR interpretation

Due: March 17

1 Introduction

The final part of the project is to implement a code generator for MLR. The target for this code generator is a stack-based interpreter, which is described below.

2 The MLR virtual machine

In this section, we describe the MLR virtual machine (VM). The virtual machine is a stand-alone program that takes an executable file and runs it. An VM executable consists of a code sequence, a literal table that contains string literals, and a C function table that contains runtime system functions used to implement services such as I/O.

2.1 Values

The VM supports three types of values: 31-bit tagged integers, 32-bit pointers to heap-allocated records of values, and 32-bit pointers to strings. A integer value n is represented by $2n + 1$ in the VM (this tagging is required for the garbage collector). The VM takes care of tagging/untagging, so the only impact of this representation on your code generator is that integer literals must be in the range -2^{30} to $2^{30} - 1$. We use word address for values (but byte addressing for instructions).

2.2 Registers

The MLR VM has four special registers: the stack pointer (SP), which points to the current top of the stack; the frame pointer (FP), which points to the base of the current stack frame and is used to access local variables; the environment pointer (EP), which points to the current closure object and is used to access global variables; and the program counter (PC), which points to the next instruction to execute.

2.3 Instructions

We define the semantics of the instructions using the following notation

$$\cdots \alpha \quad \texttt{instr} \quad \Longrightarrow \cdots \beta$$

which means that the instruction **`instr`** takes a stack configuration with α on the top and maps it to a stack with β on the top. The instructions are organized by kind in the following description.

Arithmetic instructions

$\cdots\, i_1\, i_2$ **`add`** $\Longrightarrow \cdots (i_1 + i_2)$
pops the top two integers, adds them and pushes the result.

$\cdots\, i_1\, i_2$ **`sub`** $\Longrightarrow \cdots (i_1 - i_2)$
pops the top two integers, subtracts them and pushes the result.

$\cdots\, i_1\, i_2$ **`mul`** $\Longrightarrow \cdots (i_1 \times i_2)$
pops the top two integers, multiplies them and pushes the result.

$\cdots\, i_1\, i_2$ **`div`** $\Longrightarrow \cdots (i_1/i_2)$
pops the top two integers, divides them, and pushes the result. The result is undefined if i_2 is zero.

$\cdots\, i_1\, i_2$ **`mod`** $\Longrightarrow \cdots i_1 \bmod i_2$
pops the top two integers, divides them, and pushes the remainder. The result is undefined if i_2 is zero.

$\cdots\, i$ **`neg`** $\Longrightarrow \cdots -i$
pops the integer on the top of the stack and pushes its negation.

$\cdots\, v_1\, v_2$ **`equ`** $\Longrightarrow \cdots b$
pops and compares the two values on top of the stack. If they are equal, then it pushes 1, otherwise it pushes 0. Note that if the values are pointers, then the comparison pushes true if the pointers are equal

$\cdots\, v_1\, v_2$ **`less`** $\Longrightarrow \cdots b$
pops and compares the two integers on top of the stack. If $v_1 < v_2$, then it pushes 1, otherwise it pushes 0.

$\cdots\, v_1\, v_2$ **`lesseq`** $\Longrightarrow \cdots b$
pops and compares the two integers on top of the stack. If $v_1 \leq v_2$, then it pushes 1, otherwise it pushes 0.

$\cdots\, v$ **`not`** $\Longrightarrow \cdots b$
pops v and pushes 1, if $v = 0$, and otherwise pushes 0.

Heap instructions

$\cdots\, v_0 \cdots v_{n-1}$ **`alloc(`**n**`)`** $\Longrightarrow \cdots \langle v_0, \ldots, v_{n-1} \rangle$
allocates an n element record, which is initialized from the top n stack values.

$\cdots \langle v_0, \ldots, v_{n-1} \rangle$ **`select(`**i**`)`** $\Longrightarrow \cdots v_i$
pops a record off the stack and pushes the record's ith component.

$\cdots \langle v_1, \ldots, v_{n-1} \rangle\, i$ **`index`** $\Longrightarrow \cdots v_i$
pops a record off the stack and pushes the record's ith component.

$\cdots \langle v_0, \ldots, v_{n-1} \rangle\, i\, v$ **`update`** $\Longrightarrow \cdots$
replaces the ith field of the record with the value v.

Stack instructions

$\cdots$ **int(**n**)** $\Longrightarrow \cdots n$
pushes the integer n onto the stack.

$\cdots$ **literal(**i**)** $\Longrightarrow \cdots s_i$
pushes a reference to the ith string literal (s_i) onto the stack.

$\cdots$ **label(**l**)** $\Longrightarrow \cdots addr$
pushes the code address named by the label. Note that in the encoding of this instruction, the code address is specified as an offset from the **label** instruction.

$\cdots\, v_1\, v_2$ **swap** $\Longrightarrow \cdots v_2\, v_1$
swaps the top two stack elements.

$\cdots\, v_0\, v_1 \cdots v_{n-1}\, v_n$ **swap(**n**)** $\Longrightarrow \cdots v_n\, v_1 \cdots v_{n-1}\, v_0$
swaps the top stack element with the n'th from the top. All other stack elements are unchanged.

$\cdots\, v_n \cdots v_0$ **push(**n**)** $\Longrightarrow \cdots v_n \cdots v_0\, v_n$
pushes the nth element from the top of the stack.

$\cdots\, v$ **pop** $\Longrightarrow \cdots$
pops and discards the top stack element.

$\cdots\, v_1 \cdots v_n$ **pop(**n**)** $\Longrightarrow \cdots$
pops and discards the top n stack elements.

$\cdots$ **loadlocal(**n**)** $\Longrightarrow \cdots v$
fetches the value (v) in the word addressed by FP $+\, n$ and pushes it on the stack.

$\cdots\, v$ **storelocal(**n**)** $\Longrightarrow \cdots$
pops v off the stack and stores it in the word addressed by FP $+\, n$.

$\cdots$ **loadglobal(**n**)** $\Longrightarrow \cdots v$
fetches the value (v) in the word addressed by EP $+\, n$ and pushes it on the stack.

$\cdots$ **pushep** $\Longrightarrow \cdots ep$
push the current contents of the EP on the stack.

$\cdots\, ep$ **popep** $\Longrightarrow \cdots$
pop a value from the stack and store it in the EP.

Control-flow instructions

$\cdots$ **jmp(**n**)** $\Longrightarrow \cdots$
transfer control to instruction PC $+\, n$.

$\cdots\, b$ **jmpif(**n**)** $\Longrightarrow \cdots$
pops b off the stack and if $b \neq 0$ it transfers control to instruction PC $+\, n$.

$\cdots$ **call** $\Longrightarrow \cdots pc$
pushes the current PC value (which will be the address of the next instruction) and transfers control to $addr$, where the current value of the EP is $\langle addr, \ldots \rangle$.

$\cdots$ **`entry(`**n**`)`** $\Longrightarrow \cdots fp\, w_1 \cdots w_n$
pushes the current value of the FP register and sets FP to SP. Then it allocates n uninitialized words on the stack.

$\cdots\ pc\, fp \cdots v$ **`ret`** $\Longrightarrow \cdots$
resets the stack pointer to the frame-pointer; pops the saved FP into the FP register, pops the return PC, and then jumps to the return address.

$\cdots\ pc\, fp \cdots v$ **`tailcall`** $\Longrightarrow \cdots pc$
pops the current frame off the stack (like **`ret`**) and then transfers control to $addr$, where the current value of the EP is $\langle addr, \ldots \rangle$. Unlike the **`call`** instruction, this instruction does not push the return PC.

$\cdots\ args$ **`ccall(`**n**`)`** $\Longrightarrow \cdots v$
Calls the nth C function. The C function will pop its arguments from the stack and push its result.

Miscellaneous instructions

$\cdots$ **`nop`** $\Longrightarrow \cdots$
no operation.

$\cdots$ **`halt`** $\Longrightarrow \cdots$
halts the program.

3 Calling conventions

An MLR function call is implemented using a four-part protocol:

1. The caller evaluates the function and argument expressions from left to right and pushes the results onto the stack. Then a **`swap`** instruction is used to get the function closure on top of the stack. The closure 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.

2. The first instruction in the function is an **`entry`** instruction, which pushes the frame-pointer, sets the new frame pointer to point to the top of the stack, and then allocates space for locals.

3. When the callee is finished and the return result is on the top of the stack, it stores the result at the location of the argument (immediately below the return PC) and then executes a **`ret`** instruction, which deallocates the local variable space, restores the frame pointer, and transfers control to the return address.

4. When control is returned to the address following the **`call`**, the caller must save the return result, which will be on top of the stack.

There are two important variations on this protocol. The first is when a function calls itself. In that case, the EP already holds the closure and does not have to be set. The second case is when the function call is a *tail call*, *i.e.*, the last action a function takes before returning. Tail calls are used to implement looping in functional languages The VM has a special **`tailcall`** operator that discards the caller's stack frame and does not push the return PC.

3.1 An example

To illustrate the VM, consider the following MLR implementation of the factorial function:

```
fun fact (n : int) : int in
  if (n <= 0) then 1 else n * fact(n-1)
```

This function has one argument (`n`) and no local variables. The argument `n` will be located at $+4$ from the frame pointer, while `i` is at -2 and `p` is at -4 from the frame pointer. The VM code for this function is given in Figure 1 (we assume that the code is located at address 100). Consider a

Address	Instruction	Comment
100:	**`entry`**`(0)`	
102:	**`loadlocal`**`(2)`	push `n`
104:	**`int`**`(0)`	
106:	**`lesseq`**	is `(n < 0)`?
107:	**`jmpif`**`(11)`	jump to `120`
109:	**`loadlocal`**`(2)`	push `n`
111:	**`loadlocal`**`(2)`	push `n`
113:	**`int`**`(1)`	
115:	**`sub`**	
116:	**`call`**	self-recursive call of `fact`
117:	**`mul`**	
118:	**`jmp(2)`**	jump to `122`
120:	**`int`**`(1)`	
122:	**`storelocal`**`(2)`	save result in argument position
124:	**`ret`**	

Figure 1: AVM code for factorial function

non-tail call of the factorial function

```
fact x
```

This will produce the following code sequence:

```
loadlocal(fact)
loadlocal(x)
swap
popep
call
```

Here we have assumed that `fact` and `x` are local variables and we have used their names to refer to their offsets.

3.2 Submission

Your CVS repository will be seeded with CVS modules for each of the projects. For this project, the CVS module is named `project-4` and contains the sample scanner, parser, and typechecker implementation. We will also provide a code generation API. We will collect the projects at 12 midnight on Thursday March 17th from the repositories, so make sure that you have committed your final version before then.

4 Document history

Mar. 1 Original version.

Mar. 2 Added missing **`update`** operation.

Mar. 3 Changed the semantics of **`ret`** and **`tailcall`**, and added the **`push`** instruction.

Mar. 11 Changed documentation of **`ccall`** instruction.