CMSC 22610 **Implementation** **Project 4**
Winter 2007 **of** **February 22**
Computer Languages

MinML bytecode generation

Due: March 11 at 10pm

1 Introduction

The final part of the project is to implement a code generator for MinML. The target for this code generator is a stack-based interpreter, which is described below. Your implementation will consist of two phases: the first phase will translate the AST into a simplified representation of the AST in which variable locations will be determined. The second phase will translate the simplified AST to bytecode instructions.

2 Data representations

All MinML values are represented by a single machine word. In many cases, this word is a pointer to heap-allocated storage, but it might also be an immediate value. We refer to values that are represented as pointers as *boxed* values, while values that are represented as immediate integers are *unboxed*. As we explain below, the values of a datatype may be both boxed and unboxed, in which case we describe the type as having a *mixed* representation. Since type variables can be instantiated to any type, they have a mixed representation. The `unit` and `int` types map directly onto immediate integers (`()` is represented as `0`) and strings are represented as pointers. Function types are also represented as pointers and are described below in Section 4. Datatype representations are the interesting case. Let t be a datatype with n nullary constructors ($C_1, \ldots, C_n$) and k data-constructor functions F_1 **`of`** $\tau_1, \ldots, F_k$ **`of`** τ_k. Then the following table gives the representation of the various constructors based on the number of constructors and the representations of the τ's.

n	k	C_i	$F_j(v)$	t's representation
> 0	0	i	n.a.	unboxed
0	1	n.a.	v	τ_1's representation
> 0	1	i	v if τ_j is boxed.	mixed
> 0	1	i	$\langle v \rangle$ if τ_j is unboxed or mixed.	mixed
≥ 0	> 1	i	$\langle j, v \rangle$	mixed

In this chart, i means the immediate value i and $\langle \cdots \rangle$ means heap-allocated tuple. Applying this algorithm to the the builtin datatypes we get:

$$\begin{array}{rcl} \texttt{false} & \rightarrow & 0 \\ \texttt{true} & \rightarrow & 1 \\ \texttt{nil} & \rightarrow & 0 \\ a\texttt{::}b & \rightarrow & \langle a, b \rangle \end{array}$$

3 The MinML virtual machine

In this section, we describe the MinML 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.

3.1 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).

3.2 Registers

The MinML 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.

3.3 Instructions

We define the semantics of the instructions using the following notation

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

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 $v = 0$; otherwise it pushes 0.

$\cdots\, v$ **boxed** $\Longrightarrow \cdots b$
pops v and pushes 1 if v is boxed; otherwise it 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.

$\cdots \langle v_0, \ldots, v_{n-1} \rangle$ **explode** $\Longrightarrow \cdots v_0 \cdots v_{n-1}$
pops a tuple off the stack and pushes its elements.

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

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. Note that a function's argument will be at offset 2, while the local variables start at offset -1.

$\cdots v$ **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 $+ n$.

$\cdots addr$ **call** $\Longrightarrow \cdots pc$
pop the destination address ($addr$), push the current PC value (which will be the address of the next instruction), and transfers control to $addr$.

$\cdots$ **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.

$\cdots$ *arg pc fp* $\cdots$ v **ret** $\Longrightarrow \cdots v$

pops the result v, resets the stack pointer to the frame-pointer; pops the saved FP into the FP register, pops the return PC, pos the argument `arg`, pushes the result v, and then jumps to the return address.

$\cdots$ *arg pc fp* $\cdots$ v *addr* **tailcall** $\Longrightarrow \cdots v$ *pc*

pops the code address $addr$, the argument v, and the current frame off the stack (like **ret**), stores v, and then transfers control to $addr$. Unlike the **call** instruction, this instruction does not push the return PC.

$\cdots v_1 \cdots v_m$ **ccall(**n**)** $\Longrightarrow \cdots v$

Calls the nth C function. The C function will pop its arguments (v_i) from the stack and push its result.

Miscellaneous instructions

$\cdots$ **nop** $\Longrightarrow \cdots$

no operation.

$\cdots$ **halt** $\Longrightarrow \cdots$

halts the program.

4 Implementing functions

MinML supports higher-order functions, which requires representing functions as heap-allocated values. For example, consider the function

```
fun add x = let fun f y = (x + y) in f end
```

The `add` function has the type

```
val add : int -> int -> int
```

When applied to an integer argument n, it returns a function that will add the integer n to its argument. The representation of the result of `add` must include the value of n. In general, the representation of a function will include the free variables of the function stored in a heap-allocated tuple that we call the function's *environment*. In this example, the environment has a single element (the value of `x`). The representation of a function must also contain the address of the function's code. We could store this address in the environment tuple, but for reasons we explain below, we instead represent a function as a pair of its code address and a pointer to its environment; we call this pair the *closure* of the function. Thus, the generated code for `add` would look like the following:

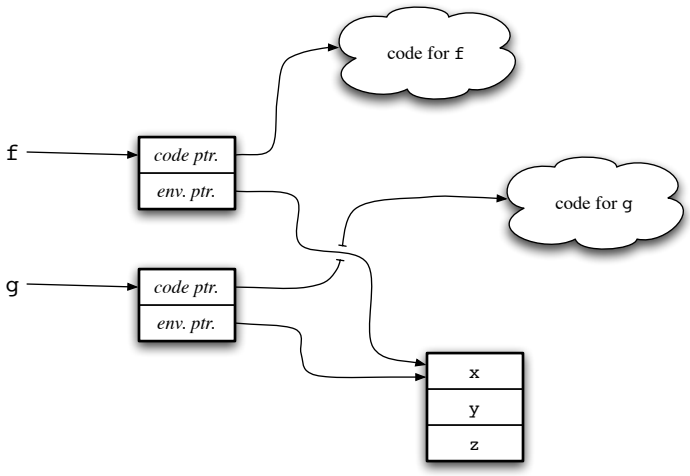

Figure 1: Representation of mutually-recursive functions

Label	Instruction	Comment
`add:`	**`entry`**`(0)`	
	`label``(f)`	
	`loadlocal``(2)`	push `x`
	`alloc``(1)`	allocate environment
	`alloc``(2)`	allocate (code ptr., env. ptr) pair
	`ret`	
`f:`	**`entry`**`(0)`	
	`loadglobal`(0)	push global variable `x`
	`loadlocal``(2)`	push `y`
	`add`	
	`ret`	

Things are a bit more complicated with mutually recursive functions. The approach that we take is to share a common environment between the functions, which is why we use the two-level representation of functions. For example, consider

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

In this case, the environment of `f` and `g` will have three values: `x`, `y`, and `z`. Figure 1 gives a pictorial representation of the representation of `f` and `g`. Note that in this case, since `f` and `g` share the same EP, the calls between the functions can be direct.

4.1 Calling conventions

An MinML function application "e_1 e_2" 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

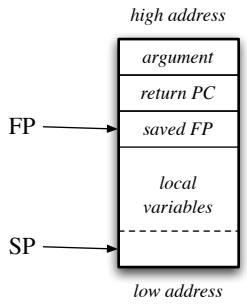

Figure 2: Stack-frame layout

the stack, which is then exploded into its code-pointer/environment-pointer parts. 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. This protocol is realized by the following sequence of instructions:

evaluate e_1	
evaluate e_2	
`swap`	swap the argument and function values
`explode`	pop the closure and push the environment and code pointers
`popep`	load the EP with the function's environment pointer
`call`	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 point to the top of the stack, and then allocates space for local variables. Figure 2 illustrates the layout of a function's stack frame after the entry protocol has been executed.

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 local variable space, restores the caller's frame pointer, pops the function's argument, and transfers control to the return address with the result on the top of the stack.

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, and restore the caller's EP.

There are two important variations on this protocol. The first is when a function calls itself or a mutually-recursive function. In that case, the EP already holds the environment pointer 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.

4.2 An example

To illustrate the VM, consider the following MinML program:

```
fun fact n = if (n <= 0) then 1 else n * fact(n-1);
fact 5
```

The `fact` 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 program is given in Figure 3 (we do not include the basis initialization code). In this code, we are saving and restoring the EP across the call to `fact` by pushing it on the stack before the call and poping it afterwards.

5 Intializing the Basis

To provide a richer programming model, we expand the basis from Project 3 with some additional operations.[1] Figure 4 gives the signature of the basis library. These functions have the same closure representation as any other function, but their implementation requires hand-crafted bytecodes.

The VM starts execution with the top of the stack pointing to a list of the command-line arguments and the PC pointing to the first instruction in the code stream. Furthermore, you must initialize the pervasive environment by creating closures for the predefined top-level functions. Thus, you should view a program

let *decls* **in** *exp*

as being in a context roughly like

```
let
val args = ...
fun print s = ...
fun fail s = ...
...
decls in exp
```

The creation of the `args` function requires popping the command-line arguments off the stack and putting them in the environment of the `args` function. At the end of the code for *exp*, your code generator should place a **`halt`** instruction.

The `fail` function should be implemented by printing its argument to the standard output and then halting.

5.1 Runtime functions

The VM provides the **`ccall`** instruction to invoke C functions. C functions expect their arguments on the stack and return their result on the stack.[2] C functions are specified by an index into the C function table.

The VM provides the following runtime system functions. We present them using the same convention that we used to present the semantics of the bytecode instruction set.

$\cdots$ *fid str* **`"MinML_print"`** $\Longrightarrow \cdots$

prints the string to the file specified by *fid*. Use `0` for the standard output.

[1] We omit the predefined operators, since they should be directly translated to bytecode instructions.

[2] The project handout states that "It is the responsibility of the caller to remove the arguments from the stack," but I have decided that it is easier to let the runtime functions pop their arguments.

Label	Instruction	Comment
`_main:`	**`entry`**`(2)`	
	`pushep`	
	`storelocal``(-1)`	save the EP in the stack frame
	`int``(0)`	`fact`'s empty environment
	`label``(fact)`	
	`alloc``(2)`	allocate `fact`'s closure
	`storelocal``(-2)`	store as local variable
	`loadlocal``(-2)`	push `fact`'s closure
	`int``(5)`	
	`swap`	
	`explode`	
	`popep`	set EP to callee's environment
	`call`	
	`pop`	discard result
	`loadlocal``(-1)`	restore caller's EP
	`halt`	
`fact:`	**`entry`**`(0)`	
	`loadlocal``(2)`	push `n`
	`int``(0)`	
	`lesseq`	is `(n < 0)`?
	`jmpif``(L1)`	
	`loadlocal``(2)`	push `n`
	`loadlocal``(2)`	push `n`
	`int``(1)`	
	`sub`	compute `n-1`
	`label``(fact)`	
	`call`	self-recursive call of `fact(n-1)`
	`mul`	compute `n*fact(n-1)`
	`jmp``(L2)`	
`L1:`	**`int`**`(1)`	
`L2:`	**`ret`**	

Figure 3: VM code for factorial program

```
datatype bool = false | true
datatype 'a list = nil | :: of ('a * 'a list)
type unit
type int
type string

val args : unit -> string list

val print : string -> unit

val fail : string -> 'a

val itos : int -> string

val size : string -> int
val sub : string * int -> int
val substring : string * int * int -> string
val concat : string list -> string
```

Figure 4: The MinML Basis

$\cdots$ *str* **"MinML_size"** $\Longrightarrow \cdots n$
pops the string *str* and pushes its length.

$\cdots$ *lst* **"MinML_concat"** $\Longrightarrow \cdots$ *str*
pops a list of strings and pushes their concatenation.

$\cdots$ *str* i **"MinML_sub"** $\Longrightarrow \cdots$ *chr*
pops a string and an integer index and pushes the integer code of the character at the given position.

$\cdots$ *str* i n **"MinML_substring"** $\Longrightarrow \cdots$ *str*
pops a string (*str*), integer index (i), and integer length (n), and pushes the substring of *str* that starts at position i and has n characters.

$\cdots$ i **"MinML_intToString"** $\Longrightarrow \cdots$ *str*
pops an integer and pushes its string representation.

$\cdots$ str_1 str_2 **""MinML_stringCmp"** $\Longrightarrow \cdots n$
pops two strings, compares them, and pushes -1 if str_1 is lexically less than str_2, 0 if str_1 is equal to str_2, or 1 if str_1 is lexically greater than str_2.

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

5.2 Wrapping C functions

As part of your bootstrap code, you will need to wrap calls to C functions inside MinML-style functions. For example, the value `print` names an MinML function that takes a single string argument and prints it to the standard output. The code for this function is as follows:

```
print:
        entry(0)
        int(0)
        loadlocal(2)
        ccall("MinML_print")
        ret
```

Note that the value itself is a closure and will have to be allocated on the heap and then stored as a local in the top-level function

```
        label(print)
        alloc(1)
        int(0)
        storelocal(print)
```

6 The code generation API

The code generation API is organized into three modules. The `Emit` module implements code streams, which are an abstraction of the generated output file; the `Labels` module implements labels for naming code locations, and the `Instructions` module implements an abstract type of VM instructions. Each of these modules is described below.

6.1 Code streams

A code stream provides a container to collect the instructions emitted by your code generator. You create a code Once code generation is complete, you invoke the `finish` operation which does an assembly pass and then writes the binary object file to disk. The `Emit` module also provides hooks for registering string literals and C functions.

6.2 Labels

The `Labels` module defines an abstract type of label that is used to represent code locations. The `Emit` structure provides the `defineLabel` function for associating a label with the current position in the code stream, and the control-flow instructions take labels as arguments. There is also an instruction for pushing the value of a label on the stack, which is required to create closures (see Section 4).

6.3 Instructions

The `Instructions` module provides an abstract type that represents VM instructions. For those instructions that take arguments, it provides constructor functions and for those without arguments, it provides abstract values.

6.4 Instruction encodings

Most instructions in the VM are either one, two, or three bytes long.[3] The first byte is consists of a two-bit length field (bits 6 and 7), and a six-bit opcode field (bits 0-5). The length field encodes the number of extra instruction bytes (*i.e.*, zero for one-byte instructions, one for two-byte instructions, and two for three-byte instructions). In the case of the two and three byte instructions, the extra bytes contain immediate data (*e.g.*, the offset of a `load` instruction), which is stored in 2's complement big-endian format.[4] Figure 5 gives a list of the instructions and their lengths; note that some instructions have both one and two or two and three-byte forms. The actual opcodes for the VM instructions are given in the `opcode.sml` file, which is part of the sample code.

7 Submission

We will create new projects on the gforge server with a sample implementation of the typechecker and the code generation API. We will collect the projects at 10pm on Sunday March 11th from the repositories, so make sure that you have committed your final version before then.

8 Document history

Feb. 22 Original version.

Feb. 26 Simplify and clarify datatype representations. Expand and correct discussion of C functions; add `MinML_stringCmp` to runtime functions. Remove unused `index` instruction from the instruction set.

March 2 Correct description of `tailcall` instruction.

March 8 Fixed error in data-representation table.

[3] The one exception if the **`int`** instruction, which has a five byte form.

[4] The term "big-endian" means that the most significant byte comes first. For example, the number 513 is represented as the byte sequence $2, 1$.

Instruction	Length	Comment
add, **sub**, **mul**, **div**, **mod**, **neq**, **equ**, **less**, **lesseq**, **not**, **boxed**	1	
alloc(n)	2	if $0 \leq n < 256$
alloc(n)	3	if $256 \leq n < 2^16$
select(i)	2	if $0 \leq i < 256$
select(i)	3	if $256 \leq i < 2^16$
explode	1	
int(n)	2	if $-128 \leq n < 128$
int(n)	3	if $n < -128$ or $128 \leq n$
int(n)	5	if $n < -2^15$ or $2^15 \leq n$
literal(n)	2	if $-128 \leq n < 128$
literal(n)	3	if $n < -128$ or $128 \leq n$
label(n)	2	if $-128 \leq n < 128$
label(n)	3	if $n < -128$ or $128 \leq n$
swap	1	
swap(n)	2	$0 \leq n < 256$
push(n)	2	$0 \leq n < 256$
pop	1	
pop(n)	2	$0 \leq n < 256$
loadlocal(n)	2	$-128 \leq n < 128$
loadlocal(n)	3	if $n < -128$ or $128 \leq n$
storelocal(n)	2	$-128 \leq n < 128$
storelocal(n)	3	if $n < -128$ or $128 \leq n$
loadglobal(n)	2	$n < 256$
loadglobal(n)	3	if $256 \leq n < 2^16$
pushep, **popep**	1	
jmp(n)	2	if $-128 \leq n < 128$
jmp(n)	3	if $n < -128$ or $128 \leq n$
jmpif(n)	2	if $-128 \leq n < 128$
jmpif(n)	3	if $n < -128$ or $128 \leq n$
call(n)	2	if $-128 \leq n < 128$
call(n)	3	if $n < -128$ or $128 \leq n$
entry(n)	2	$0 \leq n < 256$
entry(n)	3	if $256 \leq n < 2^16$
ret, **tailcall**	1	
ccall(i)	2	$0 \leq i < 256$
nop, **halt**	1	

Figure 5: VM instruction lengths