CMSC 22610 **Computer Language Implementation** **Handout 6**
Winter 2005 **March 11, 2005**

Runtime functions

1 Introduction

This handout provides further information about the interface between MLR programs and the runtime system. 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.[1] C functions are specified by an index into the C function table.

2 Runtime functions

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.

$\cdots\ \mathit{fid}\ \mathit{str}$ `ccall("MLR_print")` $\Longrightarrow \cdots$
prints the string to the output file specified by fid. The ID 0 is used to specify output to the standard output.

$\cdots\ \mathit{fid}\ \mathit{str}$ `ccall("MLR_printLn")` $\Longrightarrow \cdots$
prints the string to the output file specified by fid followed by a newline. The ID 0 is used to specify output to the standard output.

$\cdots\ \mathit{fid}$ `ccall("MLR_readLn")` $\Longrightarrow \cdots\ s$
reads a line of input from the input file specified by fid and pushes it on the stack. The ID 0 is used to specify input from the standard output.

$\cdots\ \mathit{str}$ `ccall("MLR_openIn")` $\Longrightarrow \cdots\ \mathit{fid}$
opens the named file for input and pushes its file ID on the stack.

$\cdots\ \mathit{str}$ `ccall("MLR_openOut")` $\Longrightarrow \cdots\ \mathit{fid}$
opens the named file for output and pushes its file ID on the stack.

$\cdots\ \mathit{str}$ `ccall("MLR_length")` $\Longrightarrow \cdots\ n$
pops the string str and pushes its length.

$\cdots\ \mathit{lst}$ `ccall("MLR_concat")` $\Longrightarrow \cdots\ \mathit{str}$
pops a list of strings and pushes their concatenation.

[1] 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.

$\cdots\ str\ i$ **ccall(**"MLR_sub"**)** $\Longrightarrow \cdots\ chr$

pops a string and an integer index and pushes the integer code of the character at the given position.

$\cdots\ str\ i\ n$ **ccall(**"MLR_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.

$\cdots\ i$ **ccall(**"MLR_intToString"**)** $\Longrightarrow \cdots\ str$

pops an integer and pushes its string representation.

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

3 Wrapping C functions

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

```
printLn:
        entry(0)
        int(0)
        loadlocal(2)
        ccall("MLR_printLn")
        ret
```

Note that the value itself is a closure and will have to be allocated on the heap:

```
        label(printLn)
        alloc(1)
```

For functions like `String.sub`, which take more than one argument, you will have to build intermediate closures:

```
sub:
        entry(0)
        label(sub.inner)
        loadlocal(2)
        alloc(2)
        storelocal(2)
        ret
sub.inner:
        entry(0)
        loadglobal(1)
        loadlocal(2)
        ccall("MLR_sub")
        storelocal(2)
        ret
```

Here, the `sub` function creates a closure with its argument (the index). The `sub.inner` function gets the first argument to `MLR_sub` from its environment and the second from its own argument.