COMP 321: Design of programming languages
Homework 6: Interpretation for a fragment of C

---

1. Written problems

There are no written problems for this assignment.

2. Coding problems (20 points)

For this assignment you will write an interpreter for the fragment of C for which you have already written a type-checker. Here is an informal description of execution/evaluation for this language.

Programs. Every program must define a function named `main` that has no parameters and returns an `int`. A program is executed by invoking `main`.

Statements.

- A declaration statement makes no "user-visible" changes to the environment, but we know from class that some changes have to be made.
- An initialization statement τ `x =` e must evaluate e in the current environment to get a value v, and then bind v to x in the current environment.
- An expression statement e ; is executed by evaluating e. The value of e is discarded.
- A return statement `return` e must terminate execution of the current function and return control to the caller.
- A do-while statement `do` s `while` `(`e`)` ; is executed as follows. s is executed, then e is evaluated. If e evaluates to `true`, the process is repeated; otherwise execution of the statement is complete.
- A while statement `while` `(`e`)` s is executed as follows. Evaluate e. If e evaluates to `true`, execute s and repeat. Otherwise, execution of the statement is complete.
- A for statement `for(`τ `x =` e_0; e_1; e_2`)` s is executed as follows. Initialize `x` to the value of e_0. Now repeat the following process. If e_1 evaluates to `false`, execution of the statement is complete. Otherwise, execute s, evaluate e_2, and repeat.
- An if statement `if` `(`e`)` s is executed as follows. Evaluate e. If e evaluates to `false`, execution of the statement is complete. Otherwise execute s.
- An if-else statement `if` `(`e`)` s_0 `else` s_1 is executed as follows. Evaluate e. If e evaluates to `true`, execute s_0. Otherwise execute s_1.
- A block statement $\{ss\}$ is executed by pushing a new environment onto the stack; executing ss; and then popping the environment from the stack.

Expressions. The evaluation of expressions is mostly obvious. Here are a few notes on those that are less-obvious.

- The value of a variable is the value that is bound to it in the closest enclosing block (including the current block).
- Evaluating `x++` or `++x` has a side-effect of incrementing `x` by 1. The value of `x++` is the value of `x` before incrementing. The value of `++x` is the value of `x` after incrementing. `x−−` and `−−x` are similar, except `x` is decremented by 1.
- There are no `<<` or `>>` expressions in this language.

- An assignment expression `x = `e evaluates e to get a value v and then binds v to `x` in the current environment.
- A conditional expression e ? e_0 : e_1 is evaluated as follows. If e evaluates to `true`, then the value of the conditional expression is the value of e_0 and e_1 is not evaluated. Otherwise the value of the conditional expression is e_1 and e_0 is not evaluated.

The interpreter. You must implement a structure named `Interp` that implements the evaluation and execution functions. You will need to implement at least the following:

- A type `value` to represent values.
- A type `env` that represents a map from identifiers to values (or more likely a stack of such maps). But see below for help with that.
- `exec` : `AnnAst.program -> int`. `exec` p executes the program p (by invoking its `main` function) and returns the value that is returned by `main`.
- `evalNoEnv` : `AnnAst.exp -> value`. `eval` e is the value to which e evaluates under the empty environment. Of course, you will need to implement a more general `eval` function to which `evalNoEnv` delegates.
- `valueToString` : `value -> string`. `valueToString`(v) returns a string representation of v. This function need only be implemented for values of base type (`int`, `double`, etc.). It is used by the driver program to print the result of evaluating an expression.

I have provided `Frame` and `Env` structures for you. `Frame` defines a type `'a frame`, which represents maps from identifiers to values of type `'a`. `Env` defines a type `'a env`, which represents an environment—i.e., a stack of values of type `'a frame`. You may use these structures for your frames and environments if you like. Be sure to read the documentation carefully.

Although the text indicates that statement (sequence) execution just returns an `env`, as we discussed in class, this is not sufficient for handling `return` statements and function call expressions. My suggestion in class is to have the function that executes a sequence of statements return a `value*env`, and use the `value` part as a flag to know whether or not a `return` statement had been executed. This is not ideal, but sufficient for us; you are welcome to implement alternative solutions to dealing with the issue of handling `return` statements.

Your intepreter must catch certain errors that the type-checker does not:

- If p is a program without a `main` defintiion, then `exec` p must raise `NoMainError`.
- The execution of a function with return type $\tau \neq$ `void` must finish by executing a `return` statement; if this fails to happen, then `exec` must raise `NoReturnError`. Note that if the function does have a `return` statement, then the type-checker guarantees that the expression returned by it is of the correct type; the problem here is that the type-checker does not ensure that there is a `return` statement.
- If an identifier is used in an expression before it is assigned a value, then `exec` must raise `UninitializedError`.
- `exec` may raise `RuntimeTypeError` for errors that do not fit into these categories (there may not be any such errors).

The test programs invoke functions such as `readInt` and `printInt` (and maybe `readDouble` and `printDouble`, etc.) to interact with the user. Metalanguage implementations of these functions are provided in the `IOBase` structure. You must hard-code into your expression evaluation function the following:

- Evaluation of the `readInt()` function delegates to `IoBase.readInt`(). `IoBase.readInt`() returns an (ML) `int` value that is the integer that was "read" (see below).

- Evaluation of `printInt(`e`)` consists of evaluating e to a value v that somehow encapsulates an (ML) <u>`int`</u> value n, and then invoking `IOBase.printInt(n)`.

What is going on here is that `IoBase` defines the various `readXXX` and `printXXX` functions to read from and write to user-specified streams. By default, `readXXX` reads from standard input and `printXXX` writes to standard output, and this is what the driver program I supply for you uses. So if you use the driver program to execute a program, when the execution calls `readXXX`, you (the user) must type an appropriate type value at the terminal, and when the execution calls `printXXX`(v), the value v will be printed to the terminal. However, this behavior changes for the testing code. There, the `readXXX` functions read from a specified file and `printXXX` functions write to a specified file. You need never set up `IoBase`; the driver and testing code do that work. You need only call the `readXXX` and `printXXX` functions appropriately in your interpreter.

Strategy. The structure of your interpreter should look a lot like the structure of your typing engine, though not quite identical. Start by getting expression evaluation under control for variable and function-free expressions. After that, you will need to jump directly to programs and declarations/initializations, because that is the only way to test programs or even expressions that involve variables and functions.

3. Code distribution and submission

I have provided code for the front-end of the implementation (i.e., everything through the type-checker) in the code distribution. You are responsible only for writing the `Interp` structure.

I have provided both a driver program and test suite for your code. The driver program (structure `Driver` in `driver.sml`) is similar to the driver programs in previous assignments. However, the invocation syntax has changed. See the documentation in `driver.sml` for details. The `make` target is `driver`.

The test suite attempts to execute a number of files, which are located in the `good` and `bad` subdirectories of `testfiles/programs`. There are really three files for each test program, which will always be of the form `f.cc`, `f.cc.input`, and `f.cc.output`. `f.cc` will be a program. `f.cc.input` consists of a sequence of lines, which represent the input to the program in `f.cc` when it calls `readXXX`. `f.cc.output` is the expected output of `f.cc` when `f.cc.input` is used as the input. `f.cc` is tested by running it using input from `f.cc.input` and writing the output of `printXXX` calls to a file named `f.cc.results`, and then comparing `f.cc.output` to `f.cc.results`. Some programs do not require any input; such programs still require a `.input` file, which can be empty. The test suite does not directly test expression evaluation (i.e., `Interp.evalNoEnv`), but there will be many test programs that do nothing but initialize some variables and then print the value of an expression. The `bad` directory is broken into subdirectories corresponding to exceptions. While files in those directories are tested, output will be written to a `.results` file, but that file is ignored in the test (though it may be helpful to you in debugging your code). As usual, you should add more files to both the `good` and `bad` subdirectories, and when you do so, they will automatically be part of your test suite.

As a point of comparison, my implementation of `Interp` has about 380 lines of code (including comments and blank lines).

You will submit only `interp.sml` (your interpreter implementation).