CS 322 Homework 1 – due 1:30p.m., Tuesday, Apr. 24, 2012

Interpreting fab Programs

Working individually or in teams of two, write an interpreter for (a subset of) the **fab** language, as described in the *fab Language Reference Manual* (available from the course web page).

To ease the implementation task, make the following simplifying assumptions and clarifications:

- By default, *exclude* anything to do with real numbers; if a program involving reals is given to the intepreter, the interpreter is permitted to fail in arbitrary ways. Implementing reals properly is left for extra credit (see below).
- The Manual fails to indicate the range of the **fab** `integer` type; it is intended to be a 32-bit signed value, i.e., in the range [-2147483648,2147483647].
- In implementing the arithmetic operations on integers, assume that they behave the same way as the corresponding Java operators. (This isn't an entirely trivial assumption; see the Java documentation for the division and remainder operations to learn why.)
- In implementing the `read` statement, assume that the integer literals being read will be in the format expected by the Java `Integer.parseInt` method. If the input being read is not of the correct form, the interpreter should halt execution with an error.

The interpreter should take a single command line argument specifying the `.fab` file to be interpreted. It should use the existing front-end code (see below) to perform lexical analysis, parsing, and checking. The interpreter itself can assume that the AST being interpreted has successfully passed the checker. The interpreter should execute the user program, reading any user input from standard input and writing any user output to standard output.

If the user program terminates normally, the interpreter should print the message "`Interpreter done`" to standard error. In addition, the interpreter should terminate with an appropriate error message to standard error if any of the following conditions occur: division by zero, failure to return a value from a function procedure, invalid user input (including end of file) to a `read` statement, array subscript out of bounds, dereferencing a `nil` record pointer. The interpreter may also terminate with an uncaught exception `java.lang.OutOfMemoryError` or `java.lang.StackOverflowError`. (Note that the maximum memory and stack sizes can be controlled using flags to the `java` command, but you should normally leave these alone.) Otherwise, the interpreter should never fail on any typechecked program.

A working interpreter is available in the jar file `interp.jar`; it can be run on file `foo.fab` by typing

```
java -classpath frontend.jar:interp.jar InterpDriver foo.fab
```

Implementation

The file `frontend.jar` contains a complete **fab** front-end, which parses and type-checks `.fab` files and produces an AST data structure. File `Ast.java` documents the AST. The front-end can be executed (up through type-checking) on a file `foo.fab` by typing

```
java -classpath frontend.jar CheckDriver foo.fab
```

It is recommended that you use this front-end implementation as the basis for your interpreter, since that is how we will test your submission. If you wish to use your own front-end modules from CS321, see the section below for important notes.

In any case, your intepreter implementation *must* be defined by a class `Interp` that compiles and links with the `Ast` and `InterpDriver` classes provided here; you must not change these classes. Thus, your `Interp` class must implement the method

```
static void interp(Ast.Program p) throws Interp.InterpError
```

which returns normally if the program executes successfully and otherwise throws an appropriate exception, where `InterpError` is a subclass of `Ast.Exception` that you define in the `Interp` class. Also, you will want to use the `Visitor` classes and `accept` methods in `Ast` to traverse declarations, statements, expressions, etc. Your interpreter class should be placed in a separate file `Interp.java`, which can be compiled using

```
javac -classpath .:frontend.jar Interp.java
```

There is a skeletal implementation in file `Interp0.java`, which is capable of handling very simple user programs such as:

```
{
   var x : integer := 2;
   write ("2 + 2 = ", x + x)
}
```

Feel free to build your interpreter by extending this skeletal version.

A number of short **fab** programs suitable for testing your interpreter are on the web page. You'll need to generate a much more thorough suite of test programs; if you wish to share these with the rest of the class, you can attach them to a message sent to the class mailing list.

Changes from CS321

The lexical analyzer and parser contained in `frontend.jar` are essentially identical to those provided in CS321 last term; this means that you can use your own versions of the lexer and parser without modification (provided you have obeyed the existing interfaces for building Ast's).

The Ast and type checker in `frontend.jar` have been significantly modified from those provided in CS321:

- The AST defines a set of `Visitor` interfaces and `accept` methods for the abstract classes.
- In order that this (new) `Ast` class can be used without further changes or additions, the checker has been reimplemented in a separate class `Check`, which uses visitors. This change is straightforward, but tedious.
- All uses and definitions of names carry a unique integer identifier, allowing multiple entities with the same name to be easily distinguished; this affects `VarDec`, `FuncDec`, `VarLvalue`, and `Param`.
- Each `RecordExp` and `RecordDerefLvalue` is tagged with the `RecordTypeDec` for the record type it handles.
- Each `FuncDec` now records the function's set of free identifiers, i.e. identifiers used in the body of the function that are not parameters or local variables of that function nor defined at top level. This will prove useful later in code generation.
- Top-level definitions of the built-in type names (`boolean_t`, `integer_t`, etc.), and functions to test types against these (`is_boolean_type`, `is_integer_type`, etc.) are now available in `Ast`.

If you want to use your own AST and checker code, you'll need to make similar modifications to it.

Extra Credit

For (substantial) extra credit, implement real numbers, including literals, `read` and `write`, arithmetic operations, and coercions from integers to reals where needed. You should implement reals normal IEEE double-precision floats (i.e., as Java `doubles`). Normally, the best way to do this in a statically-typed language like `fab` would be to extend the AST to distinguish between real and integer operators and include explicit integer-to-real coercion operators, which could be inserted during type checking. But for this assignment, you need to leave the AST and checker unchanged, which means that you must select the correct operations *dynamically*.

Note that the reference implementation does *not* support reals at all, so you will be on your own to interpret the Language Reference Manual to decide what "correct" behavior means. (Of course, this doesn't mean that all interpretations are reasonable!)

Submitting the Program

Place your `Interp` class in the single file `Interp.java`, and mail it to `cs322-01@cs.pdx.edu`. You must mail this file as a plain text *attachment*; the contents of the message itself don't matter. Your subject line should include the word "`HW1`". We should then be able to compile your code by creating a fresh directory, saving your attachment, copying in the provided `frontend.jar` and typing

```
   javac -classpath .:frontend.jar Interp.java
```

If we also copy in the provided file `InterpDriver.class`, we should then be able to execute your interpreter on file `foo.fab` by typing

```
   java -classpath .:frontend.jar InterpDriver foo.fab
```

Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! You may lose points if you fail to submit your program in the correct way.

If you are working in a team of two, only one team member should submit a solution, which must have the names of both team members in a comment at the top; the other team member should send mail identifying themselves as a team member.