Assignment 2: Writing a Feeny AST Interpreter

Patrick S. Li

August 31, 2015

1 Introduction

In this assignment, you will write your own implementation of Feeny as an abstract syntax tree (AST) interpreter. This implementation will serve as an operational definition of the behaviour of Feeny and later assignments will refer back to it. For this reason, the emphasis will be on simplicity and minimalism rather than on performance.

The operational semantics of Feeny are informally described in assignment 1, so please refer back to it when necessary.

2 Harness Infrastructure

A minimal framework is provided for you for lexing and parsing Feeny programs into in-memory data structures that represent the abstract syntax tree.

The function

```
void interpret (ScopeStmt* stmt)
```

in the file `src/interpreter.c` will be the entry point of your interpreter, and will be called with the parsed AST.

2.1 AST Data Structures

The data structures for representing the AST are defined in the file `src/ast.h`, and `src/ast.c` implements a pretty printer for viewing the AST as text. The data structure closely mirrors the description of Feeny's constructs in assignment 1, and you may also read the implementation of the pretty printer to see how to operate on this data structure.

The `print_exp` function prints out a Feeny expression. The `print_slotstmt` function prints out the slot definition for an object. The `print_scopestmt` function prints out a local or global Feeny statement.

2.2 Test Harness

The provided bash script `run_tests` will compile, parse, and interpret the test programs in the `test` directory. To run it, type:

```
./run_tests interpreter.c
```

at the terminal. For each test program, e.g. `cplx.feeny`, the output of the interpreter will be stored in `output/cplx.out`. The default implementation of `interpret` does nothing except print out the AST.

Please ensure that your implementation can be driven correctly by `run_tests`. Your assignment will be graded using this script.

Note that `sudoku2.feeny` is a long-running program that runs a Sudoku solver 200 times. During development you may want to comment out the appropriate line in `run_tests` to prevent it from running.

2.3 Compiling and Running Manually

To compile and run your implementation of the interpreter manually, you may also follow these steps. Compile your interpreter by typing:

```
gcc src/cfeeny.c src/utils.c src/ast.c src/interpreter.c
   -o cfeeny -Wno-int-to-void-pointer-cast
```

at the terminal. This will create the `cfeeny` executable.

Next, you have to run the supplied parser which will read in the source text for a Feeny program and dump it to a binary AST file.

```
./parser -i tests/cplx.feeny -oast output/cplx.ast
```

Finally, call the `cfeeny` executable with the binary AST file as its argument to start the interpreter.

```
./cfeeny output/cplx.ast
```

3 Structure of the Interpreter

You are free to structure your interpreter however you like as long as it correctly implements the semantics of Feeny. If you are having trouble, here is one possible structure and order of implementation.

3.1 Interpreter Objects

There are four different kinds of objects used by the interpreter, the Null object, Environment objects, Integers, and Arrays. The only operation that all objects need is support for querying its object type. Given a pointer to an object, we need to be able to determine what type of object it is.

```
int obj_type (Obj* o)
```

returns an integer that indicates what type of object `o` points to.

For this initial implementation, you may use `malloc` to allocate storage for the objects on the heap and not worry about calling `free` on them. The test programs do not run for long enough for the memory leak to be a problem. In a later exercise, we will write a garbage collector to automatically reclaim storage.

3.1.1 Null Objects

Null objects do not have to support any operation except for a way to create them.

```
NullObj* make_null_obj ()
```

3.1.2 Integer Objects

Integer objects are created from a 32-bit integer value, and also need to support the basic arithmetic and comparison operators.

```
IntObj* make_int_obj (int value)
IntObj* add (IntObj* x, IntObj* y)
IntObj* sub (IntObj* x, IntObj* y)
...
Obj* lt (IntObj* x, IntObj* y)
Obj* le (IntObj* x, IntObj* y)
...
```

3.1.3 Array Objects

Array objects are created given a length and an initial value, and needs to support retrieving its length, storing a value at a specific index, and retrieving a value at a specific index.

```
ArrayObj* make_array_obj (IntObj* length, Obj* init)
IntObj* array_length (ArrayObj* a)
NullObj* array_set (ArrayObj* a, IntObj* i, Obj* v)
Obj* array_get (ArrayObj* a, IntObj* i)
```

3.1.4 Environment Objects

Environment objects are created given a parent object.

```
EnvObj* make_env_obj (Obj* parent)
```

Environments must support two operations, adding a new binding from a name to an environment entry, and retrieving the associated environment entry for a given name.

```
void add_entry (EnvObj* env, char* name, Entry* entry)
Entry* get_entry (EnvObj* env, char* name)
```

There are two different types of environment entries that may be added to an environment.

- Variable entries, which contains a single value, and supports retrieving and changing this value.
- Code entries, which contains a list of arguments and a statement representing the code body, and supports operations for retrieving these quantities.

3.2 Evaluating Expressions

All expressions evaluate to an object. To perform the evaluation we require the environment representing the local context, as well as the environment representing the global context.

```
Obj* eval_exp (EnvObj* genv, EnvObj* env, Exp* exp)
```

3.3 Evaluating Statements

I separate the evaluation of statements into two procedures. There are two contexts under which statements are evaluated:

1. When we expect an object to be returned, as in the case of evaluating the body of a function, method, or an if branch.
2. When a statement has no return value or if we discard the return value, as in top level statements or the first statement in a sequence of statements.

These two circumstances are implemented by these procedures.

```
void exec_stmt (EnvObj* genv, EnvObj* env, ScopeStmt* s)
Obj* eval_stmt (EnvObj* genv, EnvObj* env, ScopeStmt* s)
```

3.4 Recommended Order of Implementation

I recommend implementing the basic operations for the interpreter objects, and testing each of them for correctness. Next, implement the print expression in order to easily monitor and debug the behaviour of a program. Finally implement the rest of the expressions one-by-one, saving the method call expression for last. For method calls, first implement the cases where the receiver object is an integer or array. At this point, you should be able to run programs that perform basic arithmetic and that do not make use of the object system. The last feature to implement are method calls for environments.

4 Report

Implement your AST interpreter and place it in the `src` directory, naming it `interpreter_xx.c`, where `xx` is replaced with your initials. Then ensure that it can be properly driven by the testing harness by typing:

```
./run_tests interpreter_xx.c
```

Include your implementation of Towers of Hanoi, Stacks, and More Towers of Hanoi from the previous exercise, and ensure that your interpreter works correctly for those as well.

1. (66 points) **Interpreter Statistics**: Create a table in your report that measures the following statistics for the test programs: `bsearch.feeny`, `inheritance.feeny`, `cplx.feeny`, `lists.feeny`, `vector.feeny`, `fibonacci.feeny`, `sudoku.feeny`, `sudoku2.feeny`, `hanoi.feeny`, `stacks.feeny`, `morehanoi.feeny`.

   (a) (1 point) Total amount of time (in milliseconds) for `interpret` to run and return.

   (b) (1 point) Total number of method calls.

   (c) (1 point) Total number of method calls where receiver is an integer.

   (d) (1 point) Total number of method calls where receiver is an array.

   (e) (1 point) Total number of method calls where receiver is an environment.

   (f) (1 point) Total amount of time (in milliseconds) spent looking up an entry in an environment object.

2. (17 points) **Control Flow**: Conspicuously, Feeny has very limited control flow operators. For example, there is no return, break, continue, or goto statements.
   (a) (6 points) Explain how you might extend your interpreter to support a return statement.
   (b) (5 points) Explain how our choice of implementation language (C) affects the implementation of the return statement.
   (c) (6 points) For a different language of your choosing, explain how the return statement would be implemented.
3. (17 points) **Object Cloning**: Suppose you wanted to write a function called `clone` that takes a single object as an argument, and returns a *copy* of the object.
   (a) (4 points) Using your judgement, write down a precise definition of what `clone` should do.
   (b) (4 points) Can `clone` be implemented as a pure Feeny function, or does it need to be specially supported by the interpreter?
   (c) (5 points) If `clone` can be implemented as a pure Feeny function, then give its implementation here. Otherwise, explain how the interpreter would be extended to support it.
   (d) (4 points) Explain what negative consequences adding support for cloning could have on the design of Feeny. What properties are lost?

5 Deliverables

Students may work in pairs or alone. Please submit your written answers to section 4 as *report_XX.pdf*, and your programs as *interpreter_XX.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign2_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny2]` in the subject header.