ASSIGNMENT 4: JOCALF INTERPRETER

KYRYLO CHERNYSHOV (KC875), ROQUE SOTO (RAS656)

1. Summary

There was no single most challenging part; rather, a few tiny things were challenging in their own way. For example, we had some trouble with implementing recursive functions, negative integers, and external functions. Some interesting design decisions that we made include making the Int expression type a string, and encapsulating the env and state types away from the implementation of eval_expr.

2. Design, implementation and testing

2.1. **Types.** Our most important data types were the env and state types, which were associative arrays mapping identifiers/locations to values, the value type, a variant type that described every possible thing an expression could evaluate to (e.g. int, bool, closure, etc), and the expr type, a variant type that was every possible type of expression, as defined in the JoCalf reference.

2.2. **Helper functions.** We wrote helper functions that implemented each of the external functions as described in the reference; these were of type value -> value, and were bound to their identifiers in initial_env. We also wrote helper functions to manipulate state and environment variables. These functions abstracted away the specific structure of the data types, and provided an interface through which we could interact with the state and environment without caring about how they are implemented. We feel that this decision helped a lot in making the code easier to write. We also wrote helper functions to convert to primitive values (to_prim) and to strings (to_string), as defined in the reference. We made heavy use of library functions like fst/snd, and List.map.

2.3. **Implementation.** We basically implemented the interpreter feature-by-feature, going pretty much in the order that everything was listed in the expression definition in the reference. Some features we implemented together, generally the more complex ones, and the simpler ones we implemented on our own and merged.

2.4. **Testing.** Most of the testing was done in the REPL that had been provided to us. Before we got the basic pieces working, we did a bit of testing in utop, which was quite annoying. We also wrote a whole bunch of sanity check tests in test.ml. The pipeline was to implement each feature and then test it a bit in the REPL to get rid of the major bugs and convince ourselves that it was probably correct, and then at the end use all of the features together in more complex test cases, both in the REPL and in test.ml. An example of a corner case that only came up later was the handling of min_int, which failed. To solve this, we changed the definition of the Int value type to Int of string, and used string_of_int to convert it later, in eval.ml.

3. Division of labour

As mentioned, for the more complex features like function application and binop, we worked together, and for simpler ones like try-catch-finally and definitions we worked separately. Working together mostly involved one person writing 80% of the code, and the other person writing the rest and helping with debugging. My partner and I fixed more of each others' bugs than of our own.

4. Known problems

None, as far as we know.

5. Comments

We enjoyed this assignment even more than A4. We found it a lot easier, and a lot more fun. We probably spent a few days of a few hours a day working on this, which wasn't too bad. The hardest parts were probably externals, binops on objects, and function application. One thing we would change is throwing Exception: ``Unbound variable x'' when there was an unbound identifier, i.e. listing the identifier name. It's quite an easy feature to implement, and we thought it would have been fun to do.