TidBIT!

Programming Language

Purpose Develop a language that has a different internal processing from its user-facing representation.

This homework modifies the TidBIT! (This is de-Bruijn Indices, Toots!) language, which is a blending of the language we developed in class that supports first-class functions and BARF, which used an environment for its evaluation.

Exercises

Exercise 1 Download `tidbit.rkt` and be sure you understand how it works.

Exercise 2 Add missing tests, especially in regards to errors, to solidify your understanding.

Exercise 3 The most significant change you will be making is creating and then using a second language, called `CORE`, which will handle the actual evaluation of the language. Define `CORE`, which looks exactly like `TIDBIT`, but uses constructor names like `CNum`, `CAdd`, etc.

Exercise 4 `CORE` is implemented with de-Bruijn indices. In other words, symbol ids are not helpful here! Rename `CId` to `CIdx`, and be sure it uses the correct type for a de-Bruijn index, as opposed to a symbol. Then, remove the bound IDs from `CWith` and `CFun`, since de-Bruijn representations don't make use of these.

Exercise 5 Currently, `TIDBIT` is evaluated with an `ENV` that maps id's to values. However, `CORE` has no such id's, and uses indices instead. Thus, change the type of `ENV` to just be a list of `VAL`. Modify `VAL` to indicate it keeps track of `CORE` values, and that functions in the core language do not take argument names.

Exercise 6 Delete the `lookup` function, as it's no longer needed. Modify `eval` to process a `CORE`, making changes as needed. Comment out `run` and your tests for now.

Exercise 7 Now begins the process of translating a `TIDBIT` to a `CORE`. The most interesting part of this transformation is replacing an `Id` with a `CIdx`. To do this, one must know how many bindings "up" an id is bound by. Thus, create a helper type, `BINDING-DEPTH`, which is a function type from symbols to naturals. Define `empty-depth`, which always errors, and `binding-encountered`, which is a function that takes a `BINDING-DEPTH` and a symbol, indicating a new binding was just seen, and produces the appropriate `BINDING-DEPTH`. Test this function thoroughly! **Hint:** The error `empty-depth` throws is equivalent to what `lookup` threw!

Exercise 8 Define `preprocess`, which takes a `TIDBIT` and `BINDING-DEPTH` and produces the equivalent `CORE`.

Exercise 9 Comment `run` back in and change it as needed, and add all your old tests back. Note that while the implementation of the language has changed drastically, all of the old tests should still pass!

Exercise 10 Separating the language that users see and that is actually evaluated lets us modify each independently. While it's nice to provide a `with` form in the language, we do not need it in `CORE`. Remove the `CWith` constructor and case from `eval`, and then modify `preprocess` to convert a `With` to an equivalent `Fun` and `Call` (which will then need to be `preprocess`ed.

Exercise 11 On the flipside, we can make the language the users see more robust while not changing the core language! Modify the CFG, type, and parser for `TIDBIT` to allow for multi-argument (one or more) functions and multi-argument (one or more) function calls. Handle this in the preprocessor by currying these. Note that the translation can be done in two ways: either translate the input to a different `TIDBIT` and continue `preprocess`ing, as done in the previous change, or translate to the final `CORE`. One of these will usually be more convienient, and while your types may allow for empty lists, you know that (**hint:** because of your parser) these lists will never be empty.