Interpreter, Primitives

In the last assignment, you implemented `lambda`, which let you create functions within Scheme. Scheme also has primitive functions, which are "built-in" functions that are not written in Scheme themselves. You'll implement primitive functions for this assignment.

This assignment is to be done individually. You can talk to other people in the class, me (Dave), and any of the course staff (graders, lab assistants, teaching assistants, prefects) for ideas and to gain assistance. You can help each other debug programs, if you wish. The code that you must be your own, however, and you are not permitted to share your code with others, not even your partner from other portions of the course. See the course syllabus for more details or just ask me if I can clarify.

1. Get started

Download this 08-primitives.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. Functionality

Here are the particular primitive functions you need to implement: `+`, `null?`, `car`, `cdr`, `cons`, and `map`.

Primitives are functions not implemented in Scheme; you'll implement them as C functions that get called by your interpreter directly. More details follow in the section below on primitive functions. A few comments on these specific primitives:

- `+` should be able to handle any number of integer or real arguments (if 0 arguments, return 0). If any of the arguments are reals, return a real; else return an integer.

- `null?`, `car`, and `cdr` should take one argument each, but you should have error checking to make sure that's the case. Similarly, you need to check to make sure that the argument is appropriately typed.
- `cons` should take two arguments, but you should have error checking to make sure that's the case. You'll also need to modify the code you have that handles output, because `cons` allows you to create non-list pairs. You'll need to add functionality to use a dot to separate such pairs. This portion of the Dybvig Scheme book does a decent job at describing how such output should work. (Note that the `display` function in my linkedlist binaries won't work on pairs such as this one. If you're using my binaries and using the linkedlist `display` function, you'll need to write your own instead.)
- `map` should take two arguments as usual.

3. Primitive functions

There's one bit of trickiness that will come up: you're going to want to have both functions-as-closures and functions-as-primitives. Here's how to do this. In `schemeval.h`, there is now a `PRIMITIVE_TYPE`. Since this is a function to a pointer in C, here's how the additional portion appears:

```
typedef struct SchemeVal {
    objectType type;
    union {
        ...
        struct SchemeVal *(*pf)(struct SchemeVal *);
    };
} SchemeVal;
```

To show how I'm using primitive functions, here's relevant code from scattered spots in my implementation for an exponential function that you're not writing unless you want to do some optional additional work:

```
SchemeVal *primitiveExp(SchemeVal *args) {
   // check that args has length one and car(args) is numerical
   // assume that car(args) is of type double, should check that as well
   SchemeVal *result = talloc(sizeof(SchemeVal));
   result->type = DOUBLE_TYPE;
   result->d = exp(car(args)->d);
   return result;
}

void bind(char *name, SchemeVal *(*function)(SchemeVal *), Frame *frame) {
    // Add primitive functions to top-level bindings list
    SchemeVal *value = talloc(sizeof(SchemeVal));
    value->type = PRIMITIVE_TYPE;
    value->pf = function;
    frame->bindings = ....
    ....
}

void interpret(SchemeVal *tree) {

    ...

    // Make top-level bindings list
    Frame *frame = talloc(sizeof(Frame));
    frame->bindings = makeEmpty();
    frame->parent = NULL;


    bind("+", primitiveAdd, frame);
    bind("exp", primitiveExp, frame);
    ...
}
```

4. Your repository and your files

Your repository will follow the same structure that it has for the previous assignments. There should be no changes at all in the files that I provide; you'll then need to add in the files from your previous version, and/or switch over to using my binaries. But note that the binaries I provide only go through part 4 of the project; I am not providing a working if/let interpreter. (The project gets too intermingled at this point for me to be able to supply complete partial work.)

Building and testing your code will work precisely the same as on the previous assignment.

5. What to submit

Follow the instructions for the last assignment: submitting should be the same.

Have fun interpreting!

6. Preparing for celebrations of knowledge

The celebration of knowledge may ask you to write on paper C code to solve problems similar to the above. It might have you write the code from scratch; it might have you fill in some blanks in partially written code; or it might use some other method for having you demonstrate that you can write code. To prepare, you should practice writing C code on paper that can solve any of the above functions, or others like it.

If you're looking for more examples to practice:

- Implement the following additional primitives and/or special forms: `list`, `append`, and `equal?`. `equal?` only needs to work for numbers, strings, and symbols; it does not need to perform any sort of deep equality test on a nested structure.

This assignment was originally created by David Liben-Nowell and has since been updated by Dave Musicant, Jed Yang, and Laura Effinger-Dean.