Interpreter 8: 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.

1 Pair programming structure

This is a "pair" assignment, which means that if you are working on a team with someone else, you and your partner should engage in the pair programming model. You should be physically together, sharing a computer and both looking at the same screen (I recommend using the Olin labs as it's easy to see the big screens together!). Review the instructions on Moodle about pair programming.

You should make sure that over the course of an assignment that you spend roughly the same amount of time each "driving." My recommendation is to take turns approximately every 15 minutes or so. Set a timer to help you remember. I will also ask you to turn in a form about your partnership when we switch partners and at the end of the term.

For the interpreter project, you are permitted to do some debugging separate from your partner. If you do this, you should be sure that you're each contributing similar amounts: it's not fair, either in terms of time or in terms of learning, if one person does all the debugging. Make sure that you share what you found with your partner if you do decide to do some debugging separately, and make sure you're both okay with splitting up that debugging before you dive in yourself.

If pair programming in real-time just doesn't work for you and your partner, then you will need to amicably split up and work individually. If you choose this option, you **must communicate that to me [Anna]**. Please do so as soon as possible, and I will then enable you to submit separately on Moodle. If you split, you will typically not be assigned a new partner.

If things are not going well with working with your partner or you find yourselves tempted to split up the work, please talk to me. While doing some debugging separately is fine, you should be doing the initial design and coding together - if you or your partner cannot explain how part of your interpreter works, then you have not followed the pair programming policies for this class; you'll be asked to fill out a survey about whether you and your patner followed the policies at the end of the course.

1.1 Get started

You'll download the folder with the starter code from the link below. Like for homeworks 1 and 2, unzip the folder and move it to where you'd like (I recommend the COURSES drive, and you'll **need** it to be there if you're in the Olin labs or if this is a homework using C).

Them you'll do the activity in VS Code: open a new window in VS Code (in WSL for windows users - see Homework 1), and drag the unzipped folder with the starter code onto the new window (or use Open Folder). Create a new terminal in the window.

If when running tests you run into a permissions error, you can fix this by running the following code in the terminal:

```
chmod a+x test-e test-m
```

This tells the computer that all users should be permitted to execute (run) the files `test-e` and `test-m`.

Click here to download the starter files.

2 Preparation: function pointers in C

In this assignment, you'll be dealing with pointers to functions in C. As it turns out, you can pass functions to other functions in C, just like you can in C!

I've provided you with an example of this in `functionpassing.c`. Click on the link to the code, and save the file. Then, run the code, and see how triple and square are used. The `fn` parameter in `transform` is a pointer to a function. Specifically, it's a pointer to a function that returns an `int`, and takes an `int` as a parameter. Note that the syntax around `(*fn)` is important: the `fn` is the name of the parameter (we could have named it anything we wanted!) and the parentheses are grouping the `*` with the name, indicating that this is a pointer. If we'd wanted to indicate we were passing a function that itself returned a pointer to something, our syntax would look like this:

```
int *(*fn)(int)
```

This says: `fn` is a pointer to a function that takes a single `int` as a parameter and returns type `int *`.

Before starting this assignment, I encourage you try each of the following modifications to my starter code:

- Make a new function `combine` that takes a pointer to a function with a `double` return type and two `int` parameters. Similar to transform, `combine` should additionally take in two values, call the input function on these values, and then print the result. Write a function with the appropriate signature that combines the two parameters in some way, and call `combine` appropriately in `main` to see the results.
- Function pointers in C are a little weird compared to other pointers. Specifically, functions will automatically handle dereferencing and getting an address. Try removing the `&` when the function is passed. You should see it works the same way (and if you print the pointer, you'll see the same address regardless of whether `&` is used). Then, try removing the dereference on the function call. Again, your code still works!

If you get stuck on the first task, `functionpassingsolution.c` has my example solution.

3 Functionality

Moving on to the interpreter portion of this assignment, you'll be implementing: `+`, `null?`, `car`, `cdr`, and `cons`.

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 our linkedlist binaries won't work on pairs such as this one. If you're using our binaries and using the linkedlist `display` function, you'll need to write your own instead.)

4 Sample tests

```
$ cat test-in-01.scm
(define length
  (lambda (L)
    (if (null? L)
        0
        (+ 1 (length (cdr L))))))

(length (quote ()))
(length (quote (4 5 6)))
$ ./interpreter < test-in-01.scm
0
3

$ cat test-in-02.scm
(define append
  (lambda (L1 L2)
    (if (null? L1)
        L2
        (cons (car L1) (append (cdr L1) L2)))))

(append (quote (4 5)) (quote (6 7)))

(define reverse-list
  (lambda (L)
    (if (null? L)
        L
        (append (reverse-list (cdr L)) (cons (car L) (quote ()))))))

(reverse-list (quote ()))
(reverse-list (quote (1 2 3 4)))
(reverse-list (quote (("computer" "science") "is" "awesome")))
$ ./interpreter < test-in-02.scm
(4 5 6 7)
()
```

```
(4 3 2 1)
("awesome" "is" ("computer" "science"))
```

5 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 `value.h`, we'll use a `PRIMITIVE_TYPE`. Since this is a function to a pointer in C, here's how the additional portion appears:

```
typedef enum {INT_TYPE, ..., PRIMITIVE_TYPE, ...} valueType;

struct Value {
    valueType type;
    union {
        int i;

        ...

        // A primitive style function; just a pointer to it, with the right
        // signature (pf = my chosen variable for a primitive function)
        struct Value *(*pf)(struct Value *);
    }
}
```

To show how I'm using primitive functions, here are some relevant functions and snippets of 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:

```
 /*
  * Given two int or double values in args,
  * returns a value representing argument 1
  * raised to the power of argument 2.
  * Raises an error with incorrect args length
  * or types.
  */
Value *primitiveExp(Value *args) {
   //Code omitted
}

/*
 * Adds a binding between the given name
 * and the input function. Used to add
 * bindings for primitive funtions to the top-level
 * bindings list.
 */
void bind(char *name, Value *(*function)(Value *), Frame *frame) {
    //Code omitted
}

void interpret(Value *tree) {

    ...

    // Make top-level bindings list

    // Initialize frame (code omitted)

    //Bind primitive functions
    bind("exp", primitiveExp, frame);
    ...
}
```

Note that it really is important that you bind the functions and not just that you look for a primitive function name occurring as the first thing after parentheses. Primitive functions aren't special forms like `if` or `lambda`, and treating them in the same way where you look for them to appear directly after parentheses will lead to problems if these functions are passed as inputs to other functions or returned as outputs.

6 Capstone work

Work in this section is 100% optional, and not worth any points. Nonetheless, if you're looking for an extra challenge, these are fun additional exercises to try.

- 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.

7 Starter files and your code

The files follow the same structure that they have for the previous assignments. The only change in the files I'm providing is the change to add a primitive function in `value.h`, as described above. You'll then need to add in the files from your previous version, and/or switch over to using our 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. Note that the tokenizer assignment gives you instructions for how to run a single file (which can either be one you create or one of our tests). If your code is not passing the tests, I strongly encourage you to try running on one test (of your creation or of ours) and figure out what's going wrong there before moving on. You can run valgrind as well as use gdb; if you get stuck, reach out for help!

8 How to test and submit your work

8.1 Testing your work

Go back and look at the sections at the end of Scheme Intro 2 labeled "How to test your work". Everything there about M tests and E tests is just about the same as for this assignment, but note that you should run the tests remotely (on mantis) to double check that the tests run correctly. For C homeworks after the first two, the tests will also use Valgrind to check your code: this does not run properly on Macs, and more generally, sometimes has differences between versions. To be 100% sure that all the tests pass, make sure you run them on the remote server, following the instructions on Moodle.

8.2 Submitting your work

You'll submit your work on Moodle. First, create a `CollaborationsAndSources.txt` file in the same folder as your code. In that file, indicate in what ways (if any) you collaborated with other people on this assignment and . Indicate any people you talked about the assignment with besides me (Anna) and our prefect. Did you share strategies with anyone else? Talk about any annoying errors and get advice? These are fine things to do, and you should note them in the `CollaborationsAndSources.txt` file. Give the names of people you talked with and some description of your interactions. If you used any resources outside of our course materials, that is also something to note in `CollaborationsAndSources.txt`. If you didn't talk with anyone or use any outside sources, please note that explicitly in `CollaborationsAndSources.txt`. Look back at the Homework 1 instructions if you want more details on the `CollaborationsAndSources.txt` file.

After making `CollaborationsAndSources.txt` and finishing the assignment, zip up the folder with your code (presumably, this is just zipping up the original folder you download, now with changed and added files). Upload that zip to Moodle. Only one of you and your partner (if you have one) needs to upload the file to Moodle.

For grading, the graders will first run the M tests on your code on our department server; note that they will use our original versions of the tests, so you **should not** change them in order to make your code pass the tests - doing so will mean your code still won't pass the tests for the graders! If the tests pass, a visual inspection of your code shows that you have not hyper-tailored your code to pass the tests, and you have a `CollaborationsAndSources.txt` file, then you'll earn a grade of at least M for the assignment. Hyper-tailoring your code would be doing things like checking for the exact input from the test, meaning you haven't written your code in a way that would work for similar inputs.

If your code passes all of the M tests, then the graders will run the E tests (again, on our department server). If these tests all pass, and (as above) your code is not hyper-tailored to the tests and a `CollaborationsAndSources.txt` file is present, then you'll earn a grade of E for the assignment.

If your code does not pass all of the M tests, then you'll earn a grade of NA (Not Assessable) or SP (Some Progress) for the assignment. Not Assessable means that your submission does not have an attempt at one or more functions in the assignment. Some Progress means that your submission has some code (in the correct language) for all functions in the assignment, but that the code does not pass all of the M tests, is hyper-tailored to the tests, and/or is missing a `CollaborationsAndSources.txt` file. file.

Good luck, ask questions, and have fun!

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