Linked List

For this first part of the interpreter project, you'll create a linked list that you'll then use throughout the rest of the project.

This is a "pair" assignment, which means that if you are working on a team with someone else, you and your partner should do your best to engage in the pair programming model. At any point in time, one of you is "driving," i.e. actually using the keyboard and mouse. The other is actively engaged following along, preventing bugs, and providing ideas.

You should make sure that over the course of an assignment that you spend roughly the same amount of time each "driving." I will also ask you to turn in a form rating the work that your partner does. My recommendation is to take turns approximately every 15 minutes or so. Set a timer to help you remember.

1. Get started

Download this 01-linkedlist.zip file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. Values and lists

One of the first questions to arise will be "What is this a linked list of? Integers? Strings? More lists?" The catch is that you'll be storing lists created in Scheme, which can contain objects of a whole variety of types. You might think: "Ah! I'll have a superclass of some sort, with subclasses for Integers, Strings, etc." And that would be totally awesome, if you were implementing this list using an OOP (object-oriented programming) language. Oops.

In C, one way to handle this type of typing issue is to use `union` types. (See our online C reference that I assigned or other online tutorials.) These types allow you to have a single datatype that sometimes includes an `int`, sometimes a `char *`, etc.) Every such value should also include a tag telling you what kind of data you're storing.

```
typedef enum {INT_TYPE, DOUBLE_TYPE, STR_TYPE,...} valueType;

typedef struct SchemeVal {
    valueType type;
    union {
        int i;
        double d;
        char *s;
        ...
    };
} SchemeVal;

typedef struct Value Value;
```

The names `INT_TYPE`, `DOUBLE_TYPE`, etc., represent types of Values. The `Value` struct includes a field (`type`) containing one of these tags, as well as a union of various possible fields. For example, you could create a value containing the integer 5 as follows:

```
SchemeVal myInt;
myInt.type = INT_TYPE;
myInt.i = 5;
```

If you have a `SchemeVal` and want to determine its type, you might choose to use a `switch` statement:

```
switch (val.type) {
case INT_TYPE:
    // do something with val.i
    break;
case DOUBLE_TYPE:
    // do something with val.d
    break;
...
}
```

You will thus want to make some sort of linked list implementation where the nodes contain SchemeVals. There are many different ways that one can construct a linked list. The most common approach you have likely seen is one that consists of nodes, where each node is a struct containing two items: a pointer to a value, and a pointer to another node.

Don't do it this way for the assignment. There's a better way that will save you much pain later.

Because you will eventually be using your linked list to represent Scheme S-expressions, you will have a much easier time if your linked list actually resembles a Scheme linked list. Specifically, each node should be a "cons cell" with two pointers within, and it should not be strictly typed.

Here is an abbreviation of the technique that you will use:

```
typedef struct SchemeVal {
    valueType type; // type will also have a CONS_TYPE as an option
    union {
        int i;
        double d;
        /* .... */
        struct {
            struct SchemeVal *car;
            struct SchemeVal *cdr;
        };
    };
};
```

The “head” pointer for your linked list, whatever you choose to call it, should be of type `SchemeVal*`. It should be `EMPTY_TYPE` if the list is empty, or point to a `SchemeVal`. That `SchemeVal` should be one that holds a cons cell. The `car` of that cell should point to the first item in your linked list; the `cdr` should point to another `SchemeVal`. And so on. At the end of the linked list, the `cdr` of that `SchemeVal` should point to a `EMPTY_TYPE` `SchemeVal`.

And finally: if you insert tokens at the beginning of the list, which is the simplest way, your tokens will be represented in backwards order from what you typically want. One could handle this efficiently by writing code to add at the tail of the linked list instead of the head. Instead, we’ll make things simpler by writing an additional function to reverse a linked list. Is this less efficient? Yeah. This project is large enough; we won’t focus very much on efficiency, though you might think about tracking places to make it more efficient if you want to improve it at the end.

You can feel free to leverage any linked list code that we may or may not have written in class, though bear in mind that it might not fit this framework.

3. Some specifics

After you clone your repository, you should be able to see that you get the following starting files (there are others not listed here):

- `schemeval.h`: this defines the SchemeVal structure, described above
- `linkedlist.h`: this defines all of the functions that you will need to write
- `main.c`: this is a tester function that makes some nodes, puts them into a linked list, displays them, and cleans up memory afterwards
- `justfile`: contains instructions for the command `just`, which will compile and test your code
- `test-e` and `test-m`: usual
- `test_utilities.py`: helper utilities used by `test-e` and `test-m`

The missing file here is `linkedlist.c`, which you'll need to create yourself in order to implement everything in `linkedlist.h`. To compile your code, issue the command `just build` at the command prompt. This will follow the instructions in the `justfile` for building your project in order to produce an executable called `linkedlist`. At first, it won't build at all because your linkedlist.c file isn't there. Create the file and for now, create every function that you need with no code inside it so that you can get everything to build. Once you have done that, you can begin implementing your functions, and testing appropriately.

The tester code creates an executable that you can run by typing `./linkedlist`, which takes parameters depending on whether or not it should run the exemplary tests. Once you've got it compiling, though, the easiest way to run the tests is to use `python test-m` and `python test-e`, as usual, which will automatically compile all of your code and run the `./linkedlist` executable for you.

Your code should have no memory errors when running on any input (correct or incorrect) using `valgrind`. The testing scripts will automatically run `valgrind` on your code, and show you if there are memory errors.

4. Preparing for celebrations of knowledge

The celebration of knowledge will 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: you've already implemented some Scheme functions: `car`, `cdr`, and `cons`. In the same spirit, add new functions titled `append`, and `list` that are analogs to the functions of the same name in Scheme.

5. Grading

You will receive a grade of 3 if:

- Your code passes the M tests.
- Your program is written to work in general, and not to only work for the specific tests that we have provided.

You will receive a grade of 4 if you satisfy the above requirements for a 3, and …

- Your code passes the E tests.
- You have a comment before each function describing what the function does (its input/output behavior, not a play-by-play of "first checks… then makes a recursive call…").
- Your code is well-structured and not more complex than needed.

6. What to submit

I've included a short script named `zipitup`, which will create a zipfile of everything in your `linkedlist` folder. Type `bash zipitup` at the prompt, and it should create a zip file that you can then just submit. If you are working on a team, only one of you should submit it on Gradescope, and make sure to use the Gradescope ability to add a team member to add the partner.

Good luck, and have fun!

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