

Tokenizer

For this part of the assignment, you'll build a tokenizer for Scheme, in C. (You must implement the tokenizer from scratch – don't use Lex or any similar tool, even if you happen to know what it is and how to use it.)

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 [03-tokenizer.zip](#) file, and extract the zip file to your ProgrammingLanguages folder that you've been working in.

2. Assignment details

You should fill in `tokenizer.c` to implement the functions specified in `tokenizer.h`. More generally, your tokenizer should read Scheme code from a file and return a linked list of tokens accompanied by the type of each token. You must include the following token types:

```
boolean, integer, double, string, symbol, open, close, quote
```

(The last three are for `(`, `)`, and `'`.)

The output format is, per line, `token:type` .

On comments: You should strip out comments as you tokenize, so that anything after a `;` on a line is ignored.

Make sure to be super careful if you have created your test file with an editor on Microsoft Windows. Linux and Mac use an ascii `\n` at the end of a line, but Windows uses a `\r\n` at the end of a line. All of our tests run on Linux so you don't need to worry about this unless you are creating your own test files with Windows. If you are, you need to either just create your test files on Linux instead, or make sure that your tokenizer code can additionally handle `\r\n` at the ends of a line. (Which file endings do you get if you create the files within WSL, or while VSCode is connected to Docker? I'm not sure.)

For example, here's the input file `t04.scm` that I supplied in the exemplary tests:

```
(+ x2 ( + ( quote x ) "foo;; still a string" 323) ;; now it's a comment!
```

On the above, your program should output:

```
(:open
+:symbol
x2:symbol
(:open
+:symbol
(:open
quote:symbol
x:symbol
):close
"foo;; still a string":string
323:integer
):close
```

Your tokenizer should handle bad input gracefully; your program should never crash with a segfault, bus error, or the like. If the input code is untokenizable, print an error message. You may print something generic such as “Syntax error: untokenizeable”, or you can provide more detailed information in the error message depending on what the problem is. Whatever you print, it should start with the string text “Syntax error” so that it will pass the automated tests. (Upper or lower case doesn’t matter.) After encountering an error, your program should exit after printing the error - don’t read any more input. This is why we wrote the function `textit`. Also feel free to follow your error message by printing Scheme code around the error to be helpful if you can, though this optional.

3. Syntax details


Scheme is a complex language. In the interest of sanity, here is a simplified syntax for numbers that I expect your tokenizer to handle (adapted from [Dybvig’s book](#)):

```
<number>  -> <sign> <ureal> | <ureal>
<sign>     -> + | -
<ureal>    -> <uinteger> | <udecimal>
<uinteger> -> <digit>+
<udecimal> -> . <digit>+ | <digit>+ . <digit>*
<digit>    -> 0 | 1 | ... | 9
```

Note that `*` is shorthand for zero or more repetitions of the symbol, and `+` is shorthand for one or more repetitions of the symbol. Tip: if you want to convert strings to numbers, you can use the functions `strtol` and `strtod` in the standard library.

Similarly, you should recognize symbols (identifiers) with the following syntax (again adapted from [Dybvig’s book](#)):

```
<identifier> -> <initial> <subsequent>* | + | -  
<initial>      -> <letter> | ! | $ | % | & | * | / | : | < | = | > | ? | ~ | _  
<subsequent> -> <initial> | <digit> | . | + | -  
<letter>      -> a | b | ... | z | A | B | ... | Z  
<digit>       -> 0 | 1 | ... | 9
```



This is a little inconsistent with the actual behavior of Scheme, but it simplifies things up a bit (at the cost of some minor incompatibilities).

Symbols are case-sensitive.

You may also find [the syntax section of Dybvig's book](#) to be helpful. The dialect described may not be completely consistent with the above, but it's readable in English. The BNF that I have provided above is considered to be the correct specification for this assignment. Try to match the behavior of the Guile whenever you're in doubt.

Scheme provides many different ways of enclosing lists (parentheses, square brackets, etc.) We will only write code that uses parentheses for purposes of enclosing lists. None of our test code will use square brackets. You can write your tokenizer and later parts of the project to only work on parentheses. That said, if you wish your project to also work on square brackets, it isn't that hard of an extension. If you think you want to go that way, you'll need to make sure that you tokenize parentheses and brackets differently.

Again, we are going to identify the single quote (`'`) as a special token all of its own. It should be stored in a `SchemeVal` as a `QUOTE_TYPE`, and displayed by your tokenizer accordingly. Don't get this confused with the word `quote`, which is tokenized as a regular symbol.

4. Some additional hints

I suppose it is theoretically possible to code this assignment up so that it produces precisely the right output, but without actually storing the data in a list. Don't do that; perhaps that might be easier to get results on this portion of the project, but it will leave you in a useless position to continue for the next portion.

There are many different ways of reading input files. I found it most useful to use the functions `fgetc` and occasionally `ungetc`. Look those up.

The heart of your code will be your `tokenize` function, which reads the file and returns a list of `SchemeVal`s. Here's a rough sketch of what that function might look like:

```
SchemeVal *tokenize() {
    int charRead; // int, not char, see fgetc docs
    SchemeVal *list = makeNull();
    charRead = fgetc(stdin);

    while (charRead != EOF) {

        if (charRead == .....) {
            ...
        } else if (charRead == .....) {
            ...
        } else {
            ...
        }
        charRead = fgetc(stdin);
    }

    SchemeVal *revList = reverse(list);
    return revList;
}
```

We will guarantee that no token will be longer than 300 characters long. That includes strings. Your code doesn't need to handle arbitrarily long tokens.

We will guarantee that no line will be longer than 300 characters long. Your code doesn't need to handle arbitrarily long lines.

FAQ: Should your tokenizer catch errors with mismatched parentheses, i.e.

`)(a b)`? No. That's the parser's job.

5. Preparing for celebrations of knowledge

5.1. Coding exercises

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:

- Square brackets: `[]`. Add `openbracket` and `closebracket` token types, and tokenize as well.
- Dot: `.`. A dot by itself is useful for handling dotted pairs (which may be an optional addition.) A `dot` token type should be identified if and only if there is a dot with whitespace on both sides of it. A dot at the beginning or end of file cannot be a `dot` token.

5.2. Non-coding exercises

The celebration of knowledge might also ask you to be able to identify what sorts of errors your tokenizer can catch, vs. what it might not, given a particular BNF. Think about changing the BNF provided in the assignment, and then identifying what sorts of input should be tokenized. Also think through what sorts of programming errors should not be caught by the tokenizer.

6. The files you start with

After you clone your repository, you should be able to see that you get the following starting files:

- `schemeval.h`: essentially the same header we have been using, but some new types are added in
- `linkedList.h`: the same header file, more or less, from the last assignment
- `talloc.h`: the same header file, more or less, from the last assignment
- `tokenizer.h`: the header file for the functions you'll need to write for this assignment. You'll also create a number of helper functions for that, but those don't need to appear in the header since they will be "private".
- `main.c`: Short main function that drives the whole program.
- `justfile`: contains instructions for the command `just build`, which will compile and test your code
- `test-files-m/` and `test-files-e/`: a directory of Scheme programs as test inputs, and expected outputs
- `test-m` and `test-e`: the usual testing scripts that you run
- `tester.py`: a helper script to handle automated testing.
- `lib`: a subdirectory of binaries, see below

At this point, you have a choice regarding how to proceed for `linkedList.c` and `talloc.c`. If you want to continue to build your own interpreter that is truly yours, you can copy in these files from the last project, and move forward. Alternatively, if you would prefer to use my versions instead, you can do so. In the `lib` directory you'll see `linkedList.o` and `talloc.o` files, as well as `.h` files to go with them. These `.o` files are compiled binary versions of my `linkedList.c` and `talloc.c`. If you would prefer to use them instead of your own, you can replace them in `justfile`. Specifically, in `justfile`, change the first line as indicated. I provide these binaries so that people who have trouble with earlier parts of the project don't get slowed down, but I heavily encourage you to use your own code if it works for two reasons:

- You'll feel a much stronger sense of ownership for the project if you do.
- You won't be able to trace bugs back to the linked list / talloc source code if you need to. You'll sometimes have to make do with cryptic errors from my code that say "linked list insertion error" or something like that. My code works as specified, but it will produce bad and sometimes difficult-to-understand output if it gets bad input.

To compile your code, issue the command `just build` at the command prompt. (This will give an error if you haven't yet created the .c files you need, or if you haven't changed `justfile` to refer to the binaries if you need them.)

To run your code, first create a file with some of your own Scheme code in it, and give it a filename (e.g., `myprog.scm`). Then issue this command at a prompt:

```
./interpreter < myprog.scm
```

Your tokenizer code should read data from "stdin." In that case, the above command will redirect the text from the `myprog.scm` file into your program.

To run your code through valgrind, you can similarly type

```
valgrind ./interpreter < myprog.scm
```

One challenge you may find is that when your output gets long, it gets hard to read before it scrolls all the way off the top of your terminal window. Piping your output through the UNIX `less` program helps. It will pause the output at the bottom of the screen, waiting for you press the space bar to continue, or q if you want to quit. You can do this with out without valgrind:

```
./interpreter < myprog.scm |& less
```



```
valgrind ./interpreter < myprog.scm |& less
```

7. Testing your code

Run the tests with `python test-m` and `python test-e`, as usual, which will automatically compile all of your code and run the tests.

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.

8. 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.

9. What to submit

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

Good luck, 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.