CSCI 5161: Introduction to Compilers Spring 2023, University of Minnesota Homework 2

Posted: Feb 2, 2023
Due: Feb 15, 2023

Note: The actual code that you have to write in this assignment is very little. However, there is a manual to read and a few new things to get used to. This can become daunting unless you get started right away. Spending time early getting used to the setup will also be useful for the assignments that follow: the next one will involve a similar structure but will be a bit more complicated and after that you will need to start writing real code for which you will have to get into a mode of consulting documentation for the ML library functions and such.

The General Objective in This Homework

In this homework, we will build a scanner for the *Tiger* language described in the Appendix of the text. In the process we will learn how to use ML-Lex.

The Specific Task

Do the program problem at the end of Chapter 2 in the text, i.e. starting at page 31; at the end of this, you should have a lexical analyser for Tiger programs. The text also describes what has to be turned in. The starting code for this problem is to be found in the directory hw2 that is also available in tarred and gzipped form here.

You will find several Tiger programs on which you can run your lexical analyzer in the directory testcases that is available in tarred and gzipped form here. For the moment just use the tests in the subdirectories syntax-and-sem-analysis and comments-and-strings. The examples in the other directories will become meaningful at later stages of compilation.

There are a few further comments that you might find helpful in doing this assignment:

- Once you have filled out the tiger.lex file, you may wonder how to run the system. In my setup, I start ML up and then do the following:

```
- CM.make "sources.cm";
- Parse.parse "foo.tig";
```

  If everything works right, I will get a printout of the sequence of tokens in the file named "foo.tig."

- The above function from the Parse structure that is defined in the file driver.sml prints the tokens to the terminal. I have included in that structure another function called parseandprint that allows this token stream to be "redirected" to a file instead. For example, if I type the following

```
- Parse.parseandprint "foo.tig" "foo.tig.out";
```

  at the ML prompt, then the token stream will be saved to the file foo.tig.out instead.

- You may be a bit more circumspect about going the whole nine yards before you have checked if your lexer description is free of errors. In this case, you would want to build the lexer first using the command (in the terminal)

```
ml-lex tiger.lex
```

  After you have crossed this bridge, you may then try to build the system using CM.make; the errors you get at this point should all be limited to ML errors that are detected when trying to compile tiger.lex.sml. (As noted in class, the make function from the CM structure will automatically take care of running ml-lex on the lexer file and then including the ML code file that is produced in the build.)

- Going to the other extreme, you may want to run all the tests provided on your program and check it for correctness against some kind of standard. I have provided a suitable scaffold for doing this in the subdirectory testing of the hw2 directory. Look at the README file out there to understand how to use this to automate the testing. This directory is also available in a tarred and gzipped form.

- Something that you may wonder about is the error message structure and, specifically, about the role of the value identifier `linePos`. The main "magic" in this code is the way line numbers are associated with each token. One way to do this is to maintain information about the line number and the character position within that line directly with *each* token. However, this is expensive. What is done instead is that *only* the character position is maintained with each token. Complementing this information, the character position (from the beginning of the file) for each line and also the highest line number are maintained. The first of these is stored in the reference identifier `linePos` of the *ErrorMsg* structure and the second is maintained in another reference identifier `lineNum`. Now, when an error occurs at a particular token, a backwards search is performed in `linePos` till a character position preceding that of the token is found. As we go backwards, we keep subtracting from the highest line number. When we stop, this value indicates the needed line number. Take a look at the method `ErrorMsg.error` to see how exactly these values are used. The bottomline here is that we pay a cost only when errors actually occur and avoid work when these are absent.

- You may need to know the ASCII codes for special characters. A table of such codes can be found here. I had also mentioned in class a special way of denoting such characters. Essentially, the non-printable characters with codes in the range 0 through 31 are denoted by `^c`, where `c` is a printable character in the range 64 through 95; to get back to the original code from this 'denotation' simply subtract 64 from the code of `c`.

- One thing that may confound you during testing ML programs is the display of nested structures: the ML pretty printer uses ellipsis that is useless in viewing big structures; this may be more meaningful later but perhaps you have encountered this issue already. Fortunately, you can overcome this by setting some internal reference variables that control display. In particular, execute the following two assignments in your ML session before you try to display any big structure all of whole details you want to see:

```
Control.Print.printDepth := 100;
Control.Print.printLength := 100;
```

  I have chosen 100 as a "big enough" number here; you can go bigger if you want (and if you can really assimilate such large quantities of information).

Last updated on Feb 2, 2023 by ngopalan atsign umn dot edu and micha576 atsign umn dot edu.
