UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Prof. R. Fateman

Fall, 2002

CS 164 Programming Assignment 2: Lexical Analysis for Tiger

Due: Friday, Sept 20, 2002, 11:59PM

Your second programming assignment is to familiarize yourself with the TIGER language syntax and semantics, and to write the lexical analyzer component of a TIGER system.

This is intended to be another easy assignment. Try not to make it harder than it really is: work with your partner, start immediately, and read the programs we give you. Especially if you can't understand that material, ask for help. In asking for help on a new program, first try to clarify to yourself what you want it to do and then how it goes wrong.

Read the software/tests/ files that end in `.tig` to see some examples of TIGER programs. Within the software directory on the class home page, or on `~cs164/software` is a complete implementation of TIGER as it happens, as an interpreter. There are two subdirectories: `sun` for Sun Solaris and `i86` for i86 architecture (on windows or linux). There are files with the same names in each directory, but different binary formats. There is a readable file called `tigrun.cl` in each directory that loads files in a certain order and sets up the TIGER interpreter in machine-code form. Later this semester you will see how to write the source for these pieces, yourself.

Components of the project that are used here include a working lexical analyzer, a parser, typechecker, and interpreter. It is entirely possible that there are some bugs, but they should only be minor.

The program `tigrun` reads source code from a file and prints to the display. This should really be all that you need to evaluate expressions and hence run small programs. For example, `(tigrun "~cs164/software/tests/hello.tig")` should access the file containing `print("hello world")` which does about what you would expect.

1. The intention of this first exercise is to help you become comfortable with TIGER, since you will eventually implement it! Write a program, undoubtedly based on the `merge.tig` example, to read in a list of positive or negative integers and to print the list in ascending order. Run it and show that it works as expected. Test it in various ways to see if there are conditions under which it breaks. We recommend a simple sorting algorithm. (*Hint:* There are simpler and faster sorts than "bubble sort". If you haven't heard this before: The only advantage of bubble sort is a cute name. You should know of at least 2 better $O(n^2)$ algorithms.)

2. For this part you will have to write, with our help, a lexical analyzer. You can see how your lexical analyzer should work by running ours. To run `lex` on a file, you can execute the program `fs` for "file scanner". The program `fs` repeatedly reads tokens from the named file. It takes an optional second argument as to what to do with the tokens. One possibility is to print the tokens as they are recognized. Another is to push them on a list and return the list (reversed) as the tokenized input. To run the lexical analyzer on a string, use the program `fstring`. You may need to type `(in-package :tiger)`, or for short at the top level of lisp, `:pac :tiger` for access to these programs internal to the package in which we put the internals to `tigrun`. The definitions of these programs are also given in `shortlex.cl`, which provides a skeleton for your program.

To write your program, first look at the listing of `source/shortlex.cl` and notice the pieces of code with comments marked with `*** EXERCISE ***`. These represent tasks for you to complete, with the requirement that you should mimic the behavior of our program. Fill them in, and run the whole version on test cases to make sure your program works the same as ours or better. As turns out to be usual in CS164, the tricky programming parts are generally those written to make sure the program does something reasonable on erroneous input. Any differences between your program output and ours should be explained. You should certainly run on `testlex.txt` in the `software/tests` directory, but you must devise some of your own tests as well, as noted in that file.

There are several debugging strategies you can pursue, including running pieces of our pre-written code interspersed with your own. In this case you will have to run within `(in-package :tiger)` and to make emacs happy, you must change the comment on the top line of `shortlex` where it says `cl-user` to `tiger` and uncomment the second line in your file so that it is set in the `:tiger` package. You can selectively trace programs that you've written or that are written in the compiled file. For example, you can trace our `collect-number`. You can then read in *your* definition and see if it operates the same. (Yes, the lisp `defun` program is smart enough to notice if something being redefined has been traced and will re-trace a re-definition automatically. And yes, you can trace compiled functions indistinguishably from interpreted functions, and yes, you can re-load files or individual functions into the same lisp.)

You may draw some inspiration from the "literature", in particular `pitmantoken.cl`, which we expect to mention in class – but cite it please, if you do so.

One part of the exercise is optional: TIGER does not have floating-point numbers as one of its primitive data types. This omission relieves the implementation of the need to consider many issues (conversions, type-checking, input and output, additional arithmetic operations, etc.). But realistically, most language processors permit floating-point numbers. At least for the lexical processing, we are going to allow floats. Write a version of `collect-number` to replace `collect-integer` and put it into your lex system so that you can handle numbers like 1.0E-3 as well as integers. It is actually easy to write this program; getting it *exactly* right is not so easy. Some of the tricky parts disappear in Lisp, since it allows you to compute with integers of any length.