CS235 Languages and Automata
Prof. Lyn Turbak
Wellesley College

Handout # 9
October 26, 2007
Revised October 31, 2007

Problem Set 4

Due: 11:59 pm on Monday, November 5

This is the final version of PS4, which includes the Lexer problem (Problem 6)

Required Reading: Stoughton, Sections 3.13–3.14; Appel Chapters 1–2; Kitty Reference Manual (Handout #10)

Strongly Suggested Reading: Sipser, Section 1.4

Submission:

You should turn in a hardcopy submission packet by slipping it under Lyn's office door by 11:59pm on the due date. This packet should include: (1) your written solutions for Problems 2, 3, 4, and 5; (2) your final version of the files `Kitty.lex` and `LexDefs.sml` from the `ps4` directory; (3) any new Kitty programs you wrote for testing your lexer in Problem 6; and (4) the requested transcripts in Problems 1 and 6.

You should also submit a softcopy (consisting of your final `ps4` directory) to the drop directory `~cs235/drop/ps4/`*username*, where *username* is your username. To do this, execute the following commands in Linux:

```
cd /students/username/cs235
cp -R ps4 ~cs235/drop/ps4/username/
```

Problem 1: Fun With `egrep` In this problem, you use `egrep` with some nontrivial regular expressions to search for lines with interesting properties in various files. `egrep` is the version of `grep` using extended regular expression syntax, as described in lecture. For full documentation on `egrep`, use `man egrep` in a shell or `M-x man egrep` in Emacs.

For each of the following problems, you should submit a transcript that includes the `egrep` command you executed along with its result.

a. : English Words The directory `~cs235/public_html/wordlists` contains word list files for English from the SCOWL word list database. In this problem, you should only use those files in this database that contain the substring `words`. For example:

```
[fturbak@puma ~] cd ~cs235/public_html/wordlists/

[fturbak@puma wordlists] egrep "hackers" *words*
english-words.20:hackers
english-words.50:bushwhackers
english-words.60:whackers
english-words.80:illywhackers
```

i. Find all the words that contain eight occurrences of the same letter.

ii. Suppose we give the name *duple* to two consecutive occurrences of the same character. For example, "keenness" contains three duples (`ee`, `nn`, and `ss`), two of which are consecutive. Find all the words that contain four conecutive duples.

iii. Suppose we give the name *vowelies* to the five vowels (`a`, `e`, `i`, `o`, `u`) and `y`. Find all the words that contain exactly one occurrence of each of the six vowelies in alphabetic order.

b. : Seuss The directory `~cs235/public_html/seuss` contains files with the text of two Dr. Seuss books: *Cat in the Hat* and *Green Eggs and Ham*.

```
[fturbak@puma seuss] grep "box" *.txt
cat-in-hat.txt:Came back in with a box.
cat-in-hat.txt:A big red wood box.
cat-in-hat.txt:"In this box are two things
cat-in-hat.txt:Then, out of the box
cat-in-hat.txt:in the box with the hook.
green-eggs-and-ham.txt:in a box?
green-eggs-and-ham.txt:Not in a box.
green-eggs-and-ham.txt:I do not like them in a box.
green-eggs-and-ham.txt:I would not, could not, in a box.
green-eggs-and-ham.txt:Not in a house. Not in a box.
green-eggs-and-ham.txt:I do not like them in a box.
green-eggs-and-ham.txt:So I will eat them in a box.
```

i. Find all the lines in the files that contain `that`, `hat`, or `eat`, where capitalization of letters doesn't matter. Your pattern should take advantage of the fact that all the words end in `at` and `hat` is contained in `that`. (Note: the patterns `\<` and `\>` are used to indicate the beginning and ending of a word.)

ii. Find all the lines in the files that contain occurrences of the words `box` or `fox` that are not preceded by the substrings `Not` or `not`.

iii. Find all the lines in the files that contain at least two occurrences each of two different words, where (1) all occurences of a word must have the same capitalization and (2) both second occurrences must follow both first occurrences (in either order). E.g., an example from *Green Eggs and Ham* is `Not on a train! Not in a tree!` (with words `Not` and `a`), and an example from *Cat in the Hat* is `Bump! Thump! Thump! Bump!` (with words `Bump` and `Thump`).

Problem 2: A Regular Language (Adapted from Sipser 1.48)

Let $L_0 = \{w \mid w \in \{0,1\}^*$ and w contains an equal number of occurrences of `01` and `10`$\}$. So `101` $\in L_0$ because it contains a single `01` and a singe `10`, but `1010` $\notin L_0$ because it contains two `10`s but only one `01`. Show that L_0 is a regular language.

Problem 3: Nonregular Languages (Adapted from Sipser Problem 1.46)

Prove that each of the following languages is not regular. You may use the pumping lemma and the closure of the class of regular languages under complement, union, and intersection.

a. $L_1 = \{0^n 1^m 2^{m+n} \mid m, n \in Nat\}$

b. $L_2 = \{w \mid w \in \{0,1\}^*$ and w is not of the form $0^n 1^n$ for some $n \in Nat\}$

c. $L_3 = \{w \mid w \in \{0,1\}^*$ and $w = w^R\}$

d. $L_4 = \{w \mid w \in \{0,1\}^*$ and w contains twice as many `0`s as `1`s.$\}$

e. $L_5 = \{w \mid w \in \{0,1\}^*$ and $|w| = n^2$ for some $n \in Nat$.$\}$

f. $L_6 = \{w \mid w \in \{($`(`$,$`)`$\}^*$ and w is a balanced sequence of parentheses$\}$. (A sequence of parentheses is balanced when each open paren is followed later by a matching close paren, and such open-close pairs are properly nested.)

Problem 4: Regular and Nonregular Languages (Adapted from Sipser Problem 1.49)

a. Show that the following language is regular:

$$L_7 = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at least } k \texttt{ 1}s, \text{ for } k \geq 1\}$$

b. Show that the following language is not regular:

$$L_8 = \{1^k y \mid y \in \{0,1\}^* \text{ and } y \text{ contains at most } k \texttt{ 1}s, \text{ for } k \geq 1\}$$

Problem 5: A Buggy Proof

We know that `0*1*` specifies a regular language. Find the bug in he following "proof" that `0*1*` is not a regular language:

> Towards a contradiction, assume that `0*1*` describes a regular language. By the Pumping Lemma, there is a pumping length p for this language. For simplicity, assume p is even. Choose the string $s = \texttt{0}^{p/2}\texttt{1}^{p/2}$. By the Pumping Lemma, $s = xyz$ where $x = \texttt{0}^{(p/2)-1}$, $y =$ `01`, and $z = \texttt{1}^{(p/2)-1}$. Pumping y yields the string $\texttt{0}^{(p/2)-1}\texttt{01011}^{(p/2)-1}$, which is not in `0*1*`. Therefore, the assumption that `0*1*` is regular must be wrong.

Problem 6: Constructing a Lexer for Kitty

In this problem you will implement and test a lexical analyzer (a.k.a. lexer, scanner) for Kitty, a simple programming language described in the *Kitty Reference Manual* (Handout #10). The reference manual describes all aspects of the language. For this assignment, you should focus on the lexical conventions described in Section 2.1 of the manual.

You will create your lexical analyzer using the ML-Lex lexical analyzer generator. ML-Lex is briefly described in Section 2.5 of Appel. A more detailed description may be found in the on-line ML-Lex manual at `http://smlnj.cs.uchicago.edu/doc/ML-Lex/manual.html`.

The files that you need for this problem are in the `ps4` directory, which you can retrieve by performing `cvs update -d`. This directory contains the following files:

1. `Token.sml` specifies the structure of Kitty tokens and some operations on tokens. You should study this file.
2. `Scanner.sml` provides utility functions for using a scanner created by ML-Lex. We studied most of these functions in lecture.
3. `LexDefs.sml` provides some functions that are helpful in writing your Kitty lexer.
4. `Kitty.lex` is a skeleton of the Kitty lexer description file that you will flesh out.
5. `load-kitty-scanner.sml` is a file for loading the Kitty lexer.

The `ps4` directory also contains a `test` subdirectory populated with some sample Kitty programs. You may wish to add your own test programs to the `test` subdirectory.

Your task is to flesh out the ML-Lex lexical analyzer specification file `Kitty.lex` that describes the structure of Kitty tokens. Review the lecture slides on this and/or study the ML-Lex documentation to understand how a `.lex` file is structured.

Once you have define `Kitty.lex`, you can generate a lexical analyzer by invoking the following in a Unix shell:

```
ml-lex Kitty.lex
```

If successful, this will create a file `Kitty.lex.sml` that is an automatically generated SMLNJ program for scanning tokens according to the specification file.

To load the generated scanner, execute the following in the `*sml*` buffer:

```
use "load-kitty-scanner.sml"
```

For example, suppose that `test/count.kty` contains a Kitty program. You can get a list of the tokens in the file via

```
Scanner.fileToTokens("test/count.kty");
```

and print out the tokens (one per line) by invoking

```
Scanner.printTokensInFile("test/count.kty");
```

Notes

- The code section of `Kitty.lex` opens the module in `LexDefs.sml`, which itself opens the module in `Tokens.sml`. This means you can use the values in these modules without qualification. If you wish to add any SML code for use in your lecture, define it in `LexDefs.sml`.
- Any time you modify your `Kitty.lex` file, you need to first remake `Kitty.lex.sml` (by invoking `ml-lex Kitty.lex` in a shell) and reload the scanner by invoking `use "load-kitty-scanner.sml"` in the `*sml*` buffer.
- Every token has components that indicate the leftmost and rightmost positions of the characters from which it was constructed. In this implementation, each position is a pair of integers: the first represents the line number, and the second represents a character on that line. We will refer to a pair of such positions as the **extent** of the token.

  Non-value-bearing tokens are created by invoking the name of the terminal as a constructor on two position values that indicate the leftmost and rightmost positions of the token. Value bearing tokens are created by by invoking the name of the terminal as a constructor on three arguments: the value followed by the leftmost and rightmost position values. Here are some examples of constructor invocations that create tokens:

  > `ELSE((3,4),(3,7))` creates an `ELSE` token covering characters 4 through 7 on line 3.
  >
  > `INTLIT(301, (23,17), (23,19))` creates an integer literal token with value 301 constructed from characters 17 through 19 on line 23.

  Most tokens are on a single line. The one exception is string literal tokens, which may span several lines.
- The `LexDefs.sml` file contains several functions that construct appropriate tokens using the `yytext` and `yypos` information available in the lexer. You should be able to create all Kitty tokens by calling one of these functions.
- Any SML errors in your `Kitty.lex` file will not be caught until the invocation of

  ```
  use "load-kitty-scanner.sml"
  ```

  in the `*sml*` buffer. You will need to fix such errors in your `Kitty.lex` file, which necessitates generating the scanner again before you can see if your fixes worked. (This is why the code definitions have been moved to a separate file `LexDefs.sml`. This allows you to test the validity of your SML function definitions in `LexDefs.sml` (via `use "LexDefs.sml"` without having to recompile your lexer.)

- The `.lex` file is divided into three sections. You can only use ML-style comments in (1) the topmost section and (2) within the parenthesized right-hand-sides of productions in the third section. As far as I can tell, the second and third sections do not support any other commenting conventions. (There is no documentation about comments in the ML-Lex manual.)
- Test your scanner on a range of Kitty files that exercise all possible token types. Include in your submission the test Kitty programs and transcripts of the scanner acting on these.