Functional Programming Hackathon: Exercises

Hackathon theme:
Your company has subcontracted the development of a banking software module, which has just been delivered. Unfortunately, the subcontractor has just filed for bankruptcy and is not responding to your calls. It appears that some specifications were not properly followed, which may make this module incompatible with existing software, and render its maintenance difficult.
To make things worse, you were not given a complete list of potential errors but it appears that some variable identifiers may be incorrect, that multiple versions of the same functions coexist in the code without being documented and that the might also be some bugs in the software.

You will receive throughout the day a list of potential issues: however, the tight schedule and the uncoordinated process for problem notification would make manual modification of the source code error prone and too risky.
The only way forward is to analyse the source code automatically, and potentially correct it by the same mechanism. This process, which involves the creation of code analysing the source code, is known as *code walking*. As you are dealing with Scheme code, you can take advantage of *homoiconicity* to design your code walking system.

Hackathon Exercises

0. Acquire the source code from the file "hackathon.rkt". You will need to define a file path to access this file with the file system (see preparatory lecture, and instructions in the file "FileSystemHackathon.rkt") **[5 marks]**

1. Within the source code, the function names are the symbols immediately following a `define` instruction: (define (function var1 var2 …) […]). Write a function / set of functions that collects all function names from the source code. Only do this for the first-level functions (if a function is defined inside another function, there is no need to collect its name). *Hint: there are 14 first-levels functions in the source code file*. **[10 marks]**

2. The source code contains several functions with the same name, as these were various versions of the same function that have been left in the code. Identify these functions names manually from the previous results. For each name used multiple times, write a function / set of functions that will extract from the source code all functions definitions with that name. Assign the result to a variable `<name>-versions` (for

instance if there are several versions of a `transfer-funds` function, that variable would be `transfer-funds-versions`. **[15 marks]**

3. Write a function / set of functions that will extract all functions and variable names from the source code. You will need to collect all symbols first, then calculate the difference from a list of Scheme reserved keywords. The relevant Scheme reserved keywords are: {`define, if, begin, set!, let, cond, eq?, else, error, lambda, list, car, set-car!, apply, true, false, >=, >, +, -`}. **[15 marks]**

4. One of the developers has used the wrong variable name for `amount`. Write a function / set of functions that will automatically fix this by replacing every occurrence of '`amount`' by the correct variable symbol '`$-amount`' in the entire source code. **[15 marks]**

5. Write a function / set of functions that will identify in the source code all functions that do *not* perform arithmetic calculations on the account balance. **[20 marks]**

6. Back to the list of functions that all have the same name (exercise 2). Imagine some structural properties that would determine which of these versions is the most recent. Structural properties are complexity measures of the function code as a 'tree of symbols' (could be length, number of variables, depth, or combinations of these). Having chosen that structural property, write a function / set of functions that extracts the most recent version out of that list of multiple versions. **[20 marks]**

7. One of the function definitions has misspelled one variable name in the function definition only, the 'first line': `(define (function variable) […])`, but not in the body of the function. Write a function / set of functions that could diagnose that problem automatically (again, only for top-level functions). *Hint: you may need to access the variable via a non-trivial combination of `car` and `cdr`*. This only needs to work for functions of *one* variable, so you will only check the first variable occurring after the function name, even if the function has more than one variable. **[20 marks]**

NOTE: maximum total points, as per the above marking scheme, amount to 120 – the marks will however be capped to 100.