ITB001 Problem Solving and Programming

Semester 1, 2006

Text Processing Assignment
Part 4: Interactive Procedures

Due: Friday, 2nd June 2006

Weight: 10%

Assignment overview

The major assignment in ITB001 involves developing a computer program to perform various text processing tasks. Text processing is an important application of Information Technology and the four parts of this assignment will introduce you to a variety of challenges in the area.

Part 1 is an introductory exercise involving statistical analysis of a given text message to determine its size in bits, bytes and (computer) words, and how much it would cost to transmit it. Completing this part of the assignment will give you practice at writing program code, and will briefly introduce the way text is represented in a computer.

Part 2 involves a more sophisticated form of statistical analysis and encryption of a given text message. The analysis procedure counts the number of distinct English words in the message. The encryption procedure allows you to encipher your message with a particular key so that it can be read only by someone with a corresponding decryption key. Completing this part of the assignment will give you practice at writing and documenting much larger programs, and will give you insight into the challenges involved in interpreting and manipulating text in a computer program.

Part 3 requires you to develop a program for translating text from one language to another. This is done by creating a dictionary of words and their synonyms in another language, and by then using this to replace those words in the text that are found in the dictionary. Completing this part of the assignment will give you practice at designing and using a data abstraction, and will illustrate the difficulty of automatic language translation.

Part 4 combines the features developed in the preceding parts and introduces an interactive spellchecker. In particular, it requires you to construct a user interface

that makes it easy to apply various operations to text messages. Completing this part will give you practice at reusing previously-developed solutions and at developing interactive programs, and introduces the basic ideas underlying the important 'object-oriented' programming paradigm.

Parts 1 to 3 can all be completed independently. Part 4 makes use of your solutions to the previous three parts. In Parts 2 to 4 of the assignment you need to not only produce a working program, but you must also write a document explaining your solution.

Overview of Part 4

In Parts 1 to 3 of this assignment we have developed several procedures for doing useful text-processing jobs, including analysing the cost of sending text messages, counting the number of words in a message, encrypting a message, and translating a message from one language to another. In this part of the assignment we will put all these features together to produce a single program for manipulating text messages.

This part of the assignment comprises three distinct tasks.

Task 4a requires you to define a procedure for an interactive spellchecker.

Task 4b requires you to write a procedure for constructing and manipulating text message 'objects'.

Task 4c requires you to clearly document your solution to Task 4a, especially how you tested the interactive program.

Reusing your previous solutions

Importantly, Tasks 4a and 4b require you to reuse the procedures you developed in Parts 1 to 3 of the assignment. The procedures you will need are as follows.

- From Part 1:
  - `text-cost`: for getting statistics about a text message in Task 4b
- From Part 2:
  - `word-count`: for telling us how many words there are in a message in Task 4b
  - `encipher`: for both encrypting and decrypting messages in Task 4b
  - `decryption-key`: for decrypting messages in Task 4b
- From Part 3:
  - `lookup`: for checking to see if a given word is in a dictionary in Task 4a
  - `string->tokens` and `tokens->string`: for disassembling and reassembling messages in Task 4a
  - `translation`: for translating text messages in Task 4b
  - You will also need some well-formed dictionaries, as would normally be created using procedures `make-dict` and `add-to-dict`, in Tasks 4a and 4b

Your solution to Tasks 4a and 4b should assume the existence of the procedures listed above. To accommodate this, your solution file will load your previous solutions using the following Scheme commands:

```
(load "ITB001-1.scm")
(load "ITB001-2.scm")
(load "ITB001-3.scm")
```

These commands are already in the template file for this part of the assignment. When developing your solution for Part 4, make sure that you have the three files listed above in the same folder (directory) as your file `ITB001-4.scm`. These files must contain definitions for all the procedures listed above.

However, when we test your solution we will supply our own copies of these three files, so you do not need to resubmit your previous solutions. Importantly, this means that your solutions to Tasks 4a and 4b must *not* rely on any other utility procedures that you may have produced as part of your solutions to previous parts of the assignment. Be careful to use just those procedures listed above.

What to do if you are missing parts of the assignment

Although Part 4 of the assignment requires you to reuse your solutions to Parts 1 to 3 of the assignment, this does not prevent you from completing Part 4 if you have not completed Parts 1 to 3. This highlights the importance of *modularity* in large-scale program development.

The instructions for Parts 1 to 3 of the assignment clearly defined the *signatures* for each of the required procedures, i.e.,

- the *name* of the procedure,
- the type of each of its required *arguments*, and
- the type of the *result* returned by the procedure.

This means that even if you have not completed one of these procedures, you can still supply a 'dummy' or 'stub' procedure with the same signature to take its place. A dummy procedure need not have the full functionality of the one it replaces, but it allows development of the rest of the program to proceed even in the absence of part of the code. This is a common technique used in the construction of large-scale computer programs. In this case it will allow you to complete a solution to Part 4 of the assignment, even if you do not have fully-working solutions to Parts 1 to 3.

As an example, consider that Task 4a below requires you to reuse procedure `lookup` from Part 3 of the assignment. The signature of this procedure, as described in the instructions for Task 3a, says that it accepts a word (string) and a dictionary as arguments and returns a string. The string returned is either a synonym found in the

dictionary, or the special value `"???"` if the word was not in the dictionary. If you have not successfully completed this procedure you can put a dummy procedure with the same signature in your file `ITB001-3.scm` to allow you to develop your solution to Task 4a. A number of different dummy procedures are possible, provided they match the desired signature. For instance, the following procedure always says that the given word is not in the dictionary.

```
(define [lookup word dictionary]
  "???")
```

Notice that this dummy procedure does not even make use of its arguments, so it is obviously inadequate as a solution to Task 3a. Nevertheless, it can be used during testing of your solution to Task 4a in place of the missing procedure.

This dummy procedure could be replaced with another one during testing of Part 4a. The following procedure always says that the given word was found in the dictionary and is its own synonym.

```
(define [lookup word dictionary]
  word)
```

Again, this procedure is not an adequate implementation of procedure `lookup`, but is helpful when testing the procedure developed in Task 4a if no other `lookup` procedure is available. More elaborate dummy procedures could also be developed. For instance, a dummy procedure with memory could be produced that alternates between the two responses in the two dummies shown above.

Thus, even if you haven't successfully completed some previous part of the assignment, you should still be able to complete Part 4. Task 4c gives you the opportunity to explain your dummy procedures if you use them.

Requirements for Task 4a

The aim of this task is to develop an interactive spellchecker for text messages. The spellchecker asks the user to confirm the correctness of, or provide replacements for, any word in the message that can't be found in a given dictionary.

Your task is to develop a procedure called `spell-check` which accepts a text message, represented as a character string, and a dictionary, implemented as per your solution to Part 3 of the assignment. Procedure `spell-check` examines each word in the message. It leaves words that are found in the dictionary unchanged, but it asks the user to confirm or replace words that are not found. The procedure returns the string produced by replacing words as per the dictionary's contents and the user's responses.

For the purposes of this procedure we will assume that 'words' are contiguous sequences of alphabetic and numeric characters. Thus, in the string `"See you 'l8r'! :-)"` procedure `spell-check` will check words '`See`', '`you`' and '`l8r`', but will leave all whitespace and punctuation marks (including the smiley face) unchanged.

For each alphanumeric word procedure `spell-check` is required to see whether or not the word appears in the given dictionary. If so it leaves the word unchanged. Otherwise, it asks the user whether the unknown word should be accepted as it stands, or replaced with a correction. If a word x appears in the text message but not in the dictionary then the following message should be displayed:

```
Word not recognised: x
Hit RETURN to accept word or enter replacement
```

The user can then accept the word by hitting the carriage-return key, or can type a new word to replace unknown word x. (Hint: You should use the pre-defined procedure `read-line` to read the user's response as a string.)

When all words have been checked, and words not in the dictionary either confirmed by the user or replaced, procedure `spell-check` should return the resultant string.

Example The following interaction shows a possible application of procedure `spell-check` using the dictionary `french->english` defined in Part 3 of the assignment to confirm that all words in the text message are in French. Text entered by the user is underlined.

1. `(spell-check "Oui, Je suis." french->english)` is entered by the user
2. `"Oui, Je suis."` is returned by the computer because all the words appear in the dictionary

Example In the following exchange one of the words in the message is not found in the dictionary, so the user enters a replacement. (English words are not recognised by the French-to-English dictionary.)

1. `(spell-check "Je say Non!" french->english)`
2. `Word not recognised: say`
   `Hit RETURN to accept word or enter replacement`
   is displayed by the computer to indicate that the word is not found in the given dictionary
3. `dis` is entered by the user as a replacement for the unknown word
4. `"Je dis Non!"` is returned as the spell-checked message

Example In the following example two unknown words are encountered, one of which is replaced and one of which is left unchanged.

1. `(spell-check "Je suis un Rock Star" french->english)`
2. `Word not recognised: Rock`
   `Hit RETURN to accept word or enter replacement`
3. `Movie`
4. `Word not recognised: Star`
   `Hit RETURN to accept word or enter replacement`
5. User types carriage return, in order to accept the word
6. `"Je suis un Movie Star"` is returned

Example In the following example we use the other dictionary from Part 3 of the assignment.

1. `(spell-check "c you 2moro, mate :-)" sms->english)`
2. `Word not recognised: you`
   `Hit RETURN to accept word or enter replacement`
3. `u`
4. `Word not recognised: mate`
   `Hit RETURN to accept word or enter replacement`
5. User types carriage return
6. `"c u 2moro, mate :-)"` is returned

Requirements for Task 4b

So far we have been developing individual procedures which accept text messages as arguments. Now we will create text message 'data objects' to which we can apply various operations using the 'message passing' technique.

Your task is to define a procedure `message` which accepts a string representing a text message and returns a text message object which responds appropriately to six possible messages which tell it either to show the current value of the message, show the message's cost, encrypt the message, decrypt the message, translate the message, or check the message's spelling.

However, rather than directly sending messages to the object, you are required to also create 'syntactic sugar' procedures for each of these six operations to make text message objects easy to manipulate. These procedures, and their required behaviours when applied to text objects, are as follows.

- `(show` *object*`)` returns the current value of the message as a string.
- `(price` *object*`)` returns a string of the form "x `cents for` y `words`", where x is the cost of sending the message as calculated by procedure `text-cost` and y is the number of words in the message as determined by procedure `word-count`. However, if there is only one word in the message the string returned should be of the form "x `cents for 1 word`". To do this calculation we need to know some properties of the computer architecture and communications medium, as in Part 1 of the assignment. For the purposes of this part of the assignment we will assume these values are fixed as 8 bits per byte, 32 bits per word, and 3 cents per (computer) word.
- `(encrypt` *object key*`)` encrypts the message with the given key and returns the new value of the message.
- `(decrypt` *object key*`)` decrypts the message with the given key and returns the new value of the message.
- `(translate` *object dictionary*`)` translates the message using the given dictionary and returns the new value of the message.
- `(spellcheck` *object dictionary*`)` performs an interactive spellcheck on the message using the given dictionary, prompting the user to confirm or replace unknown words, and returns the new value of the messsage when finished.

Example In the following example we create and manipulate two different text messages. Again we assume the dictionaries are the same ones defined in the tests for Part 3 of the assignment. User inputs are underlined.

1. `(define i-say (message "u r funny! :-)"))` creates the first text object

2. `(show i-say)` returns `"u r funny! :-)"`
3. `(price i-say)` returns `"12 cents for 4 words"`
4. `(define you-say (message "Oui.  Je suis rotfl!"))` creates the second text object
5. `(encrypt i-say "secret")` returns `"ieVrLjbT]se/!n"`
6. `(price i-say)` returns `"12 cents for 1 word"` showing that although the number of English words has changed the message still occupies the same number of computer words and thus costs the same amount to transmit
7. `(decrypt i-say "secret")` returns `"u r funny! :-)"`
8. `(translate i-say sms->english)` returns `"you are funny! I'm happy"`
9. `(price i-say)` returns `"18 cents for 5 words"`
10. `(show you-say)` returns `"Oui.  Je suis rotfl!"` demonstrating that none of the changes to the first text object have affected the second
11. `(translate you-say french->english)` returns `"Yes.  I am rotfl!"`
12. `(spellcheck you-say sms->english)` initiates the following interactive exchange,
    (a) `Word not recognised: Yes`
    `Hit RETURN to accept word or enter replacement`
    is displayed
    (b) User types carriage return to accept the word
    (c) `Word not recognised: I`
    `Hit RETURN to accept word or enter replacement`
    (d) `Je` is entered by the user as a replacement word
    (e) `Word not recognised: am`
    `Hit RETURN to accept word or enter replacement`
    (f) `suis`

    and the procedure then returns `"Yes.  Je suis rotfl!"`
13. `(price you-say)` returns `"15 cents for 4 words"`

Requirements for Task 4c

As in previous parts of the assignment, we emphasise the importance of clearly communicating the results of your work to others. In particular, technical documentation describing computer programs is typically required to explain

1. *what* the program does,
2. *how* it achieves its goals, and
3. *why* it should be considered trustworthy.

With regard to the *Basic Problem Solving Process* explained in the Week 3 ITB001 tutorial, these concepts are covered by the need to

1. restate the problem in your own words,
2. give a detailed algorithm describing your solution, and
3. present your strategy for testing your program,

respectively.

Your task in this part of the assignment is to produce a document which describes your solution to Task 4a, paying particular attention to the aspects listed above. You should produce a file named `ITB001-4a.doc`. (Although the file is your answer to Task 4c, it describes how you developed your solution to Task 4a, so we use '`4a`' in the file name.)

As you will discover, testing interactive programs is much less convenient than testing the 'purely functional' ones we have developed in previous parts of the assignment, due to the need to respond to prompts from the computer. Clearly describing your testing regime is especially important in this situation.

You may produce this file using Microsoft Word or as a plain text document using emacs, vi, Notepad or some other text editor. However, the Online Assessment System expects the file to be named as shown above. Therefore, if you are submitting a plain text file, you may need to rename the file before submitting it to OAS, e.g., to change the extension from '`.txt`' to '`.doc`'. Ask the teaching staff if you don't know how to do this. (If you want to submit your document in some other format, such as PostScript or Portable Document Format, this is fine, but again the file has to have the name expected by OAS.)

Criterion Referenced Assessment for Part 4

Part 4 of the ITB001 text processing assignment will be marked on the *correctness*, *clarity* and *conciseness* of your solutions. With respect to the Faculty of Information Technology's Graduate Capabilities, assessment of this part of the assignment will use the following guidelines.

Graduate Capabilities	%	High achievement	Good	Satisfactory	Unsatisfactory
GC1: Knowledge and Skills	60	Program works to specification	Program has a few minor defects	Program has several minor defects	Program has major defects or is largely incomplete
GC2: Critical and Creative Thinking	20	Solutions are concise and elegant	Solutions follow sound practices	Some solutions are clumsy or convoluted	Solutions show a poor understanding of the principles
GC3: Communication	20	– Program code is well modularised and clearly commented – Documentation is clear and precise	– Program code is clearly commented – Documentation is precise but unclear in places	– Program code is commented but unclear in parts – Documentation is complete but hard to understand	– Code is confusing or inadequately commented – Documentation is largely incomplete or is misleading
GC4: Lifelong Learning	0	– Your ability to complete technical tasks punctually is assessed indirectly through the application of late penalties – Your ability to use technical documentation effectively is assessed indirectly through your use of the manuals and textbooks needed to successfully complete the assignment			

Submission Process for Part 4

This part of the assignment will be submitted as two separate files, a Scheme file containing your solutions to Tasks 4a and 4b, and a document file containing your answer to Task 4c. Your program code will be tested automatically, so please adhere to the following rules.

- Complete your solutions to Tasks 4a and 4b using template file `ITB001-4.scm` from the ITB001 OLT page for this part of the assignment.

- If you have any test examples in the program file, please comment them out with semi-colons so that they do not interfere with the testing software.

- Note: Comment out with semi-colons any procedure or code in the program file that is syntactically incorrect, i.e., creates an error when you press 'Run' in DrScheme, because this will cause the automatic testing software to reject your whole program.

- Complete your solution to Task 4c as a separate document file, named `ITB001-4a.doc`.

- Submit both files `ITB001-4.scm` and `ITB001-4a.doc` via the Online Assessment System (`http://www.fit.qut.edu.au/students/oas/index.jsp`). Once you have logged into OAS using your QUT Access user name and password, one of the assignment items that will be available to you will be for ITB001. You should choose the action '`Submit`' and submit your solution files. If you make a mistake, such as submitting an empty file or a draft version, you are able to re-submit as many times as you like before the due date and time. You can re-submit by choosing the action '`Re-submit`' and entering the name of the file that you would like to submit. Each re-submission overwrites any previous submissions.

- You must submit your solution before midnight to avoid incurring a late penalty. You should take into account the fact that the network might be slow or temporarily unavailable when you try to submit. Network slowness near the deadline will not be accepted as an excuse for late assignments. To be sure of avoiding a late penalty, submit your solution well before the deadline.

- Standard FIT late penalties will apply as follows: 1 day late, 10% of the given mark is deducted; 2 days late, 20% of the given mark is deducted; 3–4 days late, 30% of the given mark is deducted; 5–7 days late, 40% of the given mark is deducted; 8–10 days late, 50% of the given mark is deducted; and 11 or more days late, 100% of the given mark is deducted.

- This is an individual assignment. Please read, understand and follow QUT's guidelines regarding plagiarism. Scheme programs submitted in this assignment will be subjected to analysis by the MoSS (Measure of Software Similarity) plagiarism detection system (`http://www.cs.berkeley.edu/~aiken/moss.html`).