ITB001 Problem Solving and Programming
Semester 1, 2006

Text Processing Assignment
Part 2: Counting Words and Encryption

Due: Friday, 14th April 2006
Weight: 15%

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 2

In this part of the assignment we develop another procedure for analysing text and devise a procedure for manipulating text. Specifically, we will produce a program for counting the number of (natural language) words in a text message and for encrypting and decrypting messages. Both of these problems can be solved using recursive (or iterative) procedures that operate on lists of characters. This part of the assignment comprises four distinct tasks.

Task 2a requires you to write a procedure that counts the number of (natural language) words in a text message.

Task 2b requires you to write a procedure that enciphers a text message using a particular encryption key, so that it can be read only by someone with the corresponding decryption key.

Task 2c involves writing a small procedure to calculate what the decryption key is for a given encryption key. Combining this with your solution to Task 2b allows you to both encipher and decipher text messages.

Task 2d requires you to clearly document your solutions to Tasks 2a and 2b, so that another programmer can understand how you achieved your solution.

Tasks 2a, 2b and 2c can all be completed independently so if you get temporarily stuck on one you can switch to another. Your solutions to Tasks 2a, 2b and 2c will be needed for Part 4 of the overall text processing assignment.

Requirements for Task 2a

The aim of this task is to write a procedure called `word-count` which, given a character string representing a text message, returns the number of 'natural language' words in the message. (Not computer 'words' as in Part 1 of the text processing assignment.)

To do this, however, we need to decide what we consider to be a 'word'. For instance, is a hyphenated term such as 'stand-alone' one word or two? Punctuation marks are also problematical. Is the abbreviation 'e.g.' a single word? Is a stand-alone symbol composed entirely of punctuation marks, e.g., ':–)', a word at all?

Here we will adopt a common solution and define a word to be any space-separated sequence of characters. Let the symbol '□' represent the space character here. For instance, we consider 'stand-alone:–)' to be one word, 'stand□alone:–)' to be two words, and 'stand□alone□:–)' to be three words. Thus, space characters act as word separators.

However, it doesn't matter how many spaces appear in between two words. For instance, we consider 'stand□alone' to be two words, but so is 'stand□□□□alone'. This means your `word-count` procedure cannot rely on the number of space characters in the string to determine the number of words.

Below are some examples you can use to test your solution to Task 2a. (They are the same ones as in the template file. However, we will use different tests when marking your solution, so you should make sure that your procedure works for other cases as well.)

- `(word-count "See Spot run!")` returns `3`
- `(word-count "  See     Spot  run  !")` returns `4`
- `(word-count "SeeSpotrun!")` returns `1`
- `(word-count "     ")` returns `0`
- `(word-count "")` returns `0`
- `(word-count "How r u? ;-)")` returns `4`
- `(word-count "The (quick) brown fox jumps over the (lazy) dog.")` returns `9`

Hint: Imagine you are scanning the characters in the string from left to right. In this case you know that you have found a new word if the current character is not a space but the previous character was a space. To allow for the possibility that the first character in the string is also the first character of a word, you can assume initially that the previous character was a space.

Requirements for Task 2b

Information security is one of the most important topics in contemporary Information Technology. In this task your goal is to develop a procedure called `encipher` for enciphering text messages with a particular encryption key, so that they can be read only by someone with the corresponding decryption key. To do this we will use an extended form of 'Caesar cipher'.

The Caesar cipher is one of the oldest techniques for securing messages. Given a message and a number to use as an encryption key it works by 'shifting' each letter in the original message along the alphabet by the number of places specified by the key. For instance, if we encipher the message 'Beware the Ides of March' using 3 as our key we get the encrypted message 'Ehzduh wkh Lghv ri Pdufk' because 'B' becomes 'E', 'e' becomes 'h', and so on. (For this example we have left the spaces unchanged.) The original message can be retrieved by shifting the characters in the encrypted message by -3 spaces. Thus -3 is the decryption key corresponding to encryption key 3 in this case.

Unfortunately, messages encrypted using Caesar ciphers are easy to decipher. A particular weakness of the approach is that a given letter is always encrypted in the same way. For instance, each letter 'e' in the original message above became 'h' in the encrypted message. This makes Caesar ciphers vulnerable to attacks that analyse the frequency of letters. Since 'e' is the letter most commonly used in the English language, the four 'h's in the encrypted message above are a strong clue that 'h' was 'e' in the original message and that the encryption key was 3.

To guard against this, we will develop a more sophisticated solution. Instead of using a single number as the key, we will use a list of numbers, each of which is used to decide how far to shift one character in the original message. However, since a list of numbers is hard to remember, we will use a (non-empty) character string to produce the numbers, thus allowing us to use an easily remembered word or phrase as our encryption key.

To get started, we firstly note that all text characters are actually represented as numbers within a computer. This is helpful for solving our problem because we can refer to a character by its numeric code and use this code to calculate what the corresponding encrypted character must be. A typical computer character set contains 256 different characters, but most of these are non-printable 'control' characters. To simplify our task, we will work with a set of 95 printable characters only. The characters and their corresponding numeric codes are shown in the appendix on page 13.

To convert between characters and their numeric codes you should use the two Scheme procedures shown overleaf. (They are already in the template file for this part of the assignment.) Using these two procedures instead of Scheme's predefined `char->integer` and `integer->char` procedures simplifies the arithmetic needed to complete this task.

```
; Given a printable character, return its
; numeric code
(define [printchar->integer char]
  (- (char->integer char) 32))

; Given a numeric code between 0 and 94,
; inclusive, return the corresponding
; printable character
(define [integer->printchar code]
  (integer->char (+ code 32)))
```

The easiest way to understand our extended Caesar cipher is through an example. Imagine that you wanted to encipher the text message 'Beware the Ides of March' using the encryption key 'Julie'. Treating the key as a list of character codes, as per the table on page 13, gives us the numerical key '$42, 85, 76, 73, 69$'. Thus to encrypt the original message we must shift 'B' by 42 places, 'e' by 85 places, 'w' by 76 places, and so on. However, once we reach the second letter 'e' in the original message we have used up all the numbers in our key. To solve this we simply start from the beginning of the key again. Thus we shift the second 'e' by 42 places, the space by 85 places, the 't' by 76 places, and so on until we have exhausted all the characters in the original message.

There is one additional complication, however. At the beginning of the process described above we needed to shift letter 'B' by 42 places. Since B's character code is 34 this is straightforward because $34 + 42 = 76$, thus yielding letter 'l'. However, the next letter in the original message, which is to be shifted by 85 places, is 'e' whose character code is 69. Adding these two numbers would produce $69 + 85 = 154$, which is outside our range of character codes (0 to 94, inclusive). To solve this we must use modulo (or 'clock' or 'circular' or 'wraparound') arithmetic.[1] When a number exceeds our upper limit we cycle back to the beginning again.

The integer remainder operator 'rem' can be used to do this by dividing the sum of the two numbers by the total number of character codes (which is 95 for the simplified character set we are using). In this example we get $(69 + 85) \operatorname{rem} 95 = 59$. Thus shifting letter 'e' by 65 places yields the character '['. Similarly, the next character in the original message is 'w', which has character code 87, and is to be shifted by 76 places. The calculation $(87 + 76) \operatorname{rem} 95 = 68$ thus tells us that the corresponding encrypted character is 'd'.

Completing this process encrypts our original message 'Beware the Ides of March' as 'l[dKX0uaRKJ?QOYJeSi3,hPR', when using the key 'Julie'. This is a considerably more secure solution than the simple Caesar cipher described above. Notice that the first letter 'e' was encrypted as character '[', but the second one became the digit '0',

[1]Modulo arithmetic is a very important concept in computer science because the available range of numbers in most programming languages is limited by the computer architecture's word size and numeric overflows produce modulo results. (Scheme is an exception to this rule, however.)

and the third one became letter 'K', thus making a decryption attack based on letter frequencies much harder.

Your task is to implement this process as a procedure called encipher which accepts two arguments, a text message to be encrypted and an encryption key, both represented as character strings. Procedure cipher must return the encrypted message, as a character string.

Below are some examples you can use to test your solution to Task 2b. (They are the same ones as in the template file. However, we will use different tests when marking your solution, so you should make sure that your procedure works for other cases as well.) Some further tests making use of your solution to Task 2b appear in the section for Task 2c.

- (encipher "Fred" "   ") returns "Fred" because using spaces, which have character code 0, as a key doesn't shift any characters
- (encipher "abcdefg" "!") returns "bcdefgh" because '!' has character code 1, so it causes each letter in the message to be moved 'up' one place
- (encipher "Beware the Ides of March" "#") returns "Ehzduh#wkh#Lghv#ri#Pdufk"
- (encipher "Beware the Ides of March" "Julie") returns "l[dKX0uaRKJ?QOYJeSi3,hPR"
- (encipher "short text" "very long key") returns "jNbltldT`t"
- (encipher "See Spot run" "YourKey") returns "-U[r~ViNohh:"
- (encipher "See Spot run" "MyKey") returns "!_1eM>i@elCh" showing that encrypting the same message with different keys produces different results
- (encipher "!_1eM>i@elCh" "R&T:&") returns "See Spot run" which reveals that we can also use our encryption procedure to decrypt if we can find the right key
- (encipher (encipher "Gaius Julius Caesar was here!" "Julie") "U*36:") returns "Gaius Julius Caesar was here!" because "U*36:" is the decryption key corresponding to encryption key "Julie"
- (encipher "The quick brown fox jumps over the lazy dog!" "qwerty") returns "F`Krfo[[QrWlaoTr[ijwPhbjewUiZlqlNXtfSr_rYiYx"
- (encipher "F`Krfo[[QrWlaoTr[ijwPhbjewUiZlqlNXtfSr_rYiYx" ".(;-+&") returns "Thf quicl browo fox kumps pver tie lazz dog!" which shows the effect of trying to decrypt a message using a slightly imperfect decryption key (one character is wrong in this case)

Requirements for Task 2c

In Task 2b we developed a procedure which encrypts text messages using a character string as the key. Each character in the key was treated as a number (using the codes on page 13) to tell us how many places to shift each character in the message. To retrieve the original message from an encrypted one we need to shift the characters back to their original position. This can be done by finding the decryption key corresponding to the key used to encrypt the original message. Your task is to define a procedure called `decryption-key` which accepts an encryption key and returns the corresponding decryption key.

This is a straightforward task, thanks to the way our encryption procedure works. Recall the example from Task 2b in which we used key 'Julie' to encipher the message 'Beware the Ides of March' as 'l[dKX0uaRKJ?QOYJeSi3,hPR'. To do this we began by using letter 'J''s character code, 42, to determine how many places to shift the first letter in the message, 'B' (code 34), to produce the encrypted character 'l' (code 76).

Now, keeping in mind that we are using circular arithmetic, we can get back to the original letter 'B' again by shifting the encrypted character 'l' by a further $95-42 = 53$ places. This is because $(76 + 53) \operatorname{rem} 95 = 34$.[2]

Although this sounds complicated, all it really means is that to produce a decryption key from an encryption one, your `decryption-key` procedure merely needs to replace each character c, with numeric code n, in the encryption key by the character whose numeric code equals $95 - n$.

Below are some specific examples and algebraic properties you can use to test your solution to Task 2c, including some that make use of your solution to Task 2b. (The examples are the same ones as in the template file. However, we will use different tests when marking your solution, so you should make sure that your procedure works for other cases as well.)

- `(decryption-key "MyKey")` returns `"R&T:&"`
- `(decryption-key "YourKey")` returns `"F0*-T:&"`
- `(decryption-key "abcdefghijklmnopqrstuvwxyz")` returns `">=<;:9876543210/.-,+*)('&%"`
- `(decryption-key "qwerty")` returns `".(:-+&"`
- `(decryption-key "Julie")` returns `"U*36:"`
- `(decryption-key (decryption-key "Brutus"))` returns `"Brutus"` which reveals the symmetry between encryption and decryption keys
- In general, for any key k, it should be the case that `(decryption-key (decryption-key` k`))` returns k

[2]In mathematical terms, 53 is the *additive inverse* of 42 in modulo 95 arithmetic. In other words, if we add 42 to any number in the range 0 to 94, we can get the original number back again by adding a further 53.

- `(encipher (encipher "Agent 86 to Control" "Kaos") (decryption-key "Kaos"))` returns `"Agent 86 to Control"` which shows how we can encrypt a message and decrypt it again by finding the right decryption key
- `(encipher (encipher "Agent 86 to Control" "Kaos") (decryption-key "chaos"))` returns ``")`srL}?H[b{+-aj%Uol"`` which shows that if we don't know which key was used to encrypt the original message we can't easily find a suitable decryption key
- `(encipher (encipher "Testing, One, Two, Three" (decryption-key "qwerty")) "qwerty")` returns `"Testing, One, Two, Three"`
- In general, for any key k and message m, it should be the case that `(encipher (encipher` m k`) (decryption-key` k`))` returns m
- Similarly, for any key k and message m, `(encipher (encipher` m `(decryption-key` k`))` k`)` should return m

Hint: This problem can be solved as a straightforward 'list processing' exercise, using the `printchar->integer` and `integer->printchar` procedures defined above, and Scheme's pre-defined `string->list` and `list->string` procedures.

Requirements for Task 2d

An important aspect of programming is the ability to explain your solutions to other programmers. Communicating technical knowledge clearly is always necessary when working in a team. It's also important to leave behind clear documentation when you complete a programming job so that someone else who has to make changes to your program at some later date can understand what you have done.

Your task in this part of the assignment is to produce two documents which describe your solutions to Tasks 2a and 2b above. To do this you should use the step-by-step *Basic Problem Solving Process* explained in the Week 3 ITB001 tutorial. A copy of the process can be downloaded from the Lecture Materials page on the ITB001 OLT site. Follow this process to document your solution to Tasks 2a and 2b. You should produce two files, named `ITB001-2a.doc` and `ITB001-2b.doc`.

You may produce these files using Microsoft Word or as plain text documents using emacs, vi, NotePad or some other text editor. However, the Online Assessment System expects the two files to be named as shown above. Therefore, if you are submitting a plain text file, you may need to rename the file before doing so, 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 documents in some other format, such as PostScript or Portable Document Format, this is fine, but please let one of the lecturers know that you have done so.)

Criterion Referenced Assessment for Part 2

Part 2 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 2

This part of the assignment will be submitted as three separate files, a Scheme file containing your solutions to Tasks 2a, 2b and 2c, and two document files containing your answer to Task 2d. Your program code will be tested automatically, so please adhere to the following rules.

- Complete your solutions to Tasks 2a, 2b and 2c using the template file from the ITB001 OLT page for the relevant 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.
- 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 2d as two separate document files, as described above.
- Submit all three files 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. (E-mails sent to the teaching staff at 12:01am saying that you are having trouble submitting your file are not appreciated.)
- Standard QUT late penalties will apply. For more information about these penalties please visit `http://www.fit.qut.edu.au/forstudents/current/policy.jsp#assessment`.
- This is an individual assignment. Please read, understand and follow QUT's guidelines regarding plagiarism, which can be accessed via ITB001's OLT site.

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`).

Appendix: Character Codes

The following table shows the character codes for the 95 printable characters used in this part of the assignment, as accessed by the procedures `printchar->integer` and `integer->printchar` defined above.

Code	Char	Code	Char	Code	Char
0		32	@	64	`
1	!	33	A	65	a
2	"	34	B	66	b
3	#	35	C	67	c
4	$	36	D	68	d
5	%	37	E	69	e
6	&	38	F	70	f
7	'	39	G	71	g
8	(	40	H	72	h
9	)	41	I	73	i
10	*	42	J	74	j
11	+	43	K	75	k
12	,	44	L	76	l
13	-	45	M	77	m
14	.	46	N	78	n
15	/	47	O	79	o
16	0	48	P	80	p
17	1	49	Q	81	q
18	2	50	R	82	r
19	3	51	S	83	s
20	4	52	T	84	t
21	5	53	U	85	u
22	6	54	V	86	v
23	7	55	W	87	w
24	8	56	X	88	x
25	9	57	Y	89	y
26	:	58	Z	90	z
27	;	59	[	91	{
28	<	60	\	92	|
29	=	61	]	93	}
30	>	62	^	94	~
31	?	63	_		