

Haskell Macroprocessor

COMP40009 – Computing Practical 1

18th – 22nd October 2021

Aims

- To provide experience with recursive list processing in Haskell.
- To provide an introduction to macroprocessors.

Introduction

A *macroprocessor*, or *preprocessor*, is a program that transforms a text file by replacing specially identified *keywords* within the file by a specified piece of text. The keyword replacement rules are provided in a separate information (“info”) file. In this exercise we will adopt the convention that **keywords are strings that begin with the character ‘\$’** and that an info file consists of one keyword definition per line. Furthermore, each definition can be assumed to comprise a keyword followed by a single space character and then the keyword’s *definition*, which is a sequence of words, possibly including punctuation and whitespace characters. For example, given the info file:¹

```
$name William Shakespeare
$birth-date 1564
$town Stratford upon Avon
```

and the text file:

```
Welcome to $town, where $name was born in $birth-date.
```

a macroprocessor will generate the output text: **Welcome to Stratford upon Avon, where William Shakespeare was born in 1564.** Note that both the text and info files may include arbitrary whitespace and punctuation characters. These must be faithfully preserved during the replacement process.

The same text file can be used to generate many “welcome” messages, simply by changing the info file. For example, given a similar info file for Thomas Hardy, we might instead generate:

```
Welcome to Stinsford, where Thomas Hardy was born in 1840.
```

¹Note: Stratford upon Avon (the town) is not written with hyphens; see <https://www.telegraph.co.uk/style-book/places-and-people/>

Macroprocessors have many uses, for example in web programming, code generation and “mail merging” (an application we’ll revisit later on).

This exercise requires you to implement a very simple macroprocessor in Haskell. To help you a skeleton file `MP.hs` and several text and info files are provided in your repository.

Breaking up text

The main problem here is that the whitespace (e.g. " ", "\t" (tab), "\n" (newline) etc.) and punctuation (e.g. ".", ";", "\'" etc.) **characters must be preserved** during the replacement. This means you have to be able to spot variable references embedded within arbitrary whitespace and punctuation, as in

Keywords (e.g. `$x`, `$y`, `$z...`) may appear anywhere, e.g. `<$here>`.

for example. Similarly, the layout of a keyword definition in an info file must be faithfully reproduced. For example, in

```
$rule Reproduce this precisely -- or else!!
```

the spacing (there are seven space characters in the definition) and punctuation **must be reproduced exactly**.

In order to implement the replacement correctly, you therefore need to break up the text file into its component *words* and *separator* characters. The processing of the info file is a little simpler, in principle, as the variable name is guaranteed to be the first word on a line and is separated from its definition by a *single* space character.

In this exercise you’re going to **break up a piece of text using a `splitText` function** that returns a pair comprising the **separator characters**, i.e. a `String`, and **the list of words**, a `[String]`, in the order that they were encountered in the input. The arguments to `splitText` are the list of valid separator characters (here written `[Char]` rather than `String`, but we could have chosen either) and the input text, which is a `String`. Thus:

```
splitText :: [Char] -> String -> (String, [String])
```

As an example, consider the string `"some words; some more."`. Given a list of separator characters, `" ;."`, say, as its first argument, we’d like `splitText` to break the text up to return:

```
{("~;~~.",["some","words","","some","more",""])} 
```

You can see how the original text can be constructed from this, by interspersing the separators (first element of result pair) and the text words (second element).

Let’s now work out how to define the function recursively. The input text begins with a non-separator (`'s'`) so if we **recursively split the remainder of the string** then we will get back **a pair comprising the separators** (`" ; ."`) and words (`["ome", "words", "", "some", "more", ""]`). Importantly, note the `"`. This appears in the list of words because there are two consecutive separators (`';'` and `' '`) in the string, so there must be a null word between them. This is almost the answer we want, but it is missing the `'s'` on the front of the first

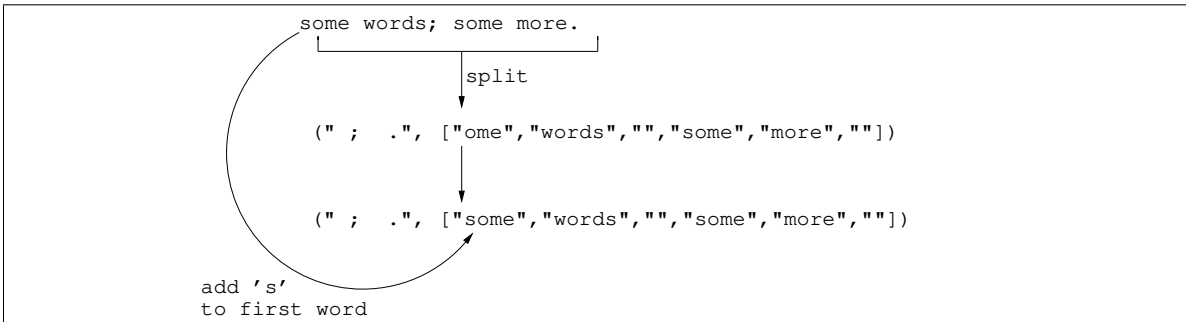


Figure 1: Text splitting at a non-separator

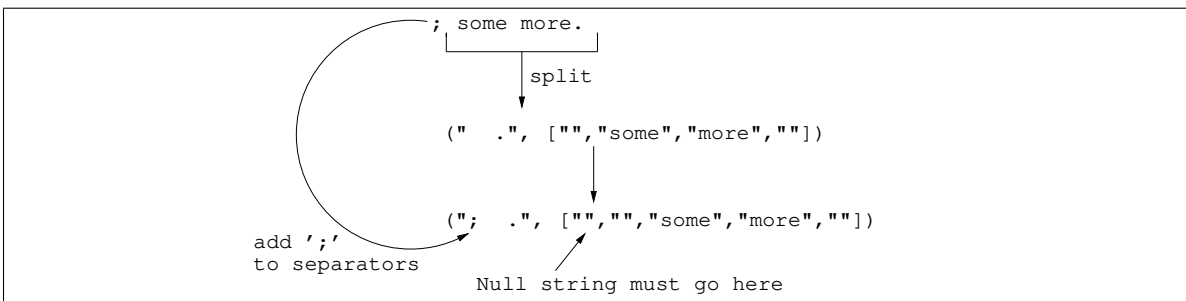


Figure 2: Text splitting at a separator

'some', so to patch up, we just add it (:) to the first word in the list returned. This is shown in Figure 1.

What happens if the first character in the input string is a separator? In order to get this right we need to think about how the original text will be rebuilt from its constituents. Given the pair `("< >", ["", "bracketed", "text", ""])`, for example, the rule is that the **first word appears before the first separator in the rebuilt string**, so the above will be rebuilt as `"<bracketed text>"`, not `"< bracketed>text"`. Note that the `"` at the end of the list arises because the string ends with a separator. Thus, **when we encounter a separator at the head of the input string the first word in the list of resulting words must be ""**. This is shown in Figure 2.

What's the base case? If the input string is null then `("", [])` would appear to work, and it can, although it's a little messier. To keep it simple we'll instead return `("", [""])`. When we use `splitText` to process keyword definitions you'll see that this is tantamount to imposing a *precondition* that each definition contains at least one word.

Preamble

The skeleton file `MP.hs` contains the type synonyms shown below.

```

type FileContents = String

type Keyword      = String
type KeywordValue = String
type KeywordDefs  = [(Keyword, KeywordValue)]

```

A top-level function `main` is provided for performing the file input/output (I/O). Haskell I/O will be covered later in the course, but suffice it to say that `readFile` delivers the contents of a file as a string and `writeFile` does the opposite. The `main` function will look for the arguments passed to it on the command line, which should be the names of the text, info and output files.

Once you have everything working, you can generate your own output, either by invoking the program from the Linux prompt. Imagine the example below (*where \$ is your terminal prompt*)

```
$ runghc MP welcome.txt shakespeare.info output.txt
```

or from within GHCi:

```
*MP> :main welcome.txt shakespeare.info output.txt
```

Getting started

As per the previous exercise, you will use the `git` version control system to get the repository with the skeleton files for this exercise and its (incomplete) test suite. You can get your repository with the following (remember to replace the `username` with your own username).

```
git clone https://gitlab.doc.ic.ac.uk/lab2120_autumn/haskellmacroprocessor_username.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files². On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -pC0F0A13C
```

(*n.b.*, `C0F0A13C` is the password). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named “skel”, because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

What to do

- ✓ Write a Haskell function `lookUp :: String -> [(String, a)] -> [a]` that, given a `search string` and a `list of string/item pairs`, returns the `list of items whose associated string matches the search string`. For example,

```
*MP> lookUp "A" [("A",8),("B",9),("C",5),("A",7)]  
[8,7]
```

Hint: try doing this with a `list comprehension` as it is much easier this way.

- ✓ Define the `splitText` function that will `break up a string as described above`. For example,

²<https://www.7-zip.org/download.html>

```
*MP> splitText " .," "A comma, then some words."
( " ,      .", ["A","comma","","then","some","words",""])
```

(The list of separators returned in the example comprises one space, a comma, then three spaces and the full stop.) Note that the empty strings ("") are important as they correspond to the locations of the separator characters , and . in the example.

- ✓ Define a function `combine :: String -> [String] -> [String]` that will `combine the components of a string from its constituent separator characters and words`, as generated by a call to `splitText`. To do this follow the rules outlined above but note that the result is `a list of strings that`, when concatenated (`concat`) will rebuild the original string. Note that this means the function should satisfy the *axiom*³

```
let (ts, ws) = splitText seps s in concat (combine ts ws) ≡ s
```

- ✓ Define a function `getKeywordDefs :: [String] -> KeywordDefs` that takes the contents of an information file in the form of a list of lines (each line is a string), and which `returns a list of keyword/definition pairs`. For example,

```
*MP> getKeywordDefs ["$x Define x", "$y 55"]
[("$x","Define x"),("$y","55")]
```

Hint: You may find it easier to `use splitText to turn each line into space-separated words`; the keyword will then be the first word in the line and its definition the concatenation (`concat`) of the remaining words. If you do it this way, you do not need to worry about punctuation and other whitespace characters in the definition; it's only the first space that's important.

- ✓ Define a function `expand :: FileContents -> FileContents -> FileContents` that takes the `contents of a text file and an info file` and `combines` them using the above functions to build a string representing the output file. Note that this function is called by the top-level `main` function that has been defined for you. You can use `main` to test your `expand` function, but you will need to `inspect the output file` in order to do this. Alternatively you can test it directly by passing in smaller test strings. For example,

```
*MP> expand "The capital of $1 is $2" "$1 Peru\n$2 Lima."
"The capital of Peru is Lima."
```

Hint: You'll first need to `split the text file into its component words and separators`. To do this you may assume that the only separators in the text file will be `" \n\t.,:;!\"'()<>/\\\"`; a predefined variable, `separators`, containing these characters has been declared in the skeleton file for you. You might also find it useful to `define a helper function to replace the words of the template file one by one`. Recall that the word may be empty ("") so you'll need to check this first. If the word is non-empty

³Optional note (you can ignore this if you wish): You will notice that `splitText` returns a pair, but that the “inverse” function, `combine` consumes its two arguments separately. It would be really nice to be able to compose the two directly, but unfortunately the types do not match. A beautiful way to fix this is to use Haskell's `uncurry` function. If `f` is a function of type `a -> b -> c` then `uncurry` turns it into a function of type `(a, b) -> c`. This is an example of a *higher-order* function, i.e. a function that takes another function as an argument and/or returns a function as its result. Using `uncurry` we can write the above axiom as:

```
concat (uncurry combine (splitText seps s)) ≡ s
```

You may find `uncurry` useful in defining your `expand` function below, but it is up to you whether you use it.

and begins with a '\$' then it is a keyword and should be replaced by looking it up in the list of `keywordDefs` generated from the info file. Otherwise, the word is returned intact.

Extension: an enhanced expander

Now assume that an info file can contain multiple sets of keyword definitions, each separated by a '#' character. Modify your implementation of `expand` so that it **expands the input file once for each set of definitions**. For example, given the info file:

```
$name William Shakespeare
$birth-date 1564
$town Stratford upon Avon
#
$birth-date 1840
$town Stinsford
$name Thomas Hardy
#
$name Charles Dickens
$town Landport
$birth-date 1812
```

your program should now expand the given text file *three* times. The three pieces of expanded text should be **separated by a line of five '-' characters**. Thus, expanding the “welcome” text file with the above info file should generate (precisely):

```
Welcome to Stratford upon Avon, where William Shakespeare was born in 1564.
-----
Welcome to Stinsford, where Thomas Hardy was born in 1840.
-----
Welcome to Landport, where Charles Dickens was born in 1812.
-----
```

Note that this functionality is similar to that of a “mail merger”, which generates multiple instances of a letter template given a file of names, addresses and other information about the target recipients. In that case each letter instance is separated by “page break” characters so that, when printed, each letter starts on a separate sheet of paper. Here, we’ve used ‘-----’ as the separator, but the principle is the same.

Submission

As with all previous exercises, you will need to use the commands `git add`, `git commit` and `git push` to send your work to the GitLab server. Then, as always, log into the LabTS server, <https://teaching.doc.ic.ac.uk/labts>, click through to your `haskellmacroprocessor` exercise https://gitlab.doc.ic.ac.uk/lab2122_autumn/haskellmacroprocessor_username and request an auto-test of your submission.

IMPORTANT: Make sure that you submit the correct commit to CATE – you can do this by checking that the key submitted to CATE matches the Commit Hash of your commit on LabTS/GitLab.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

- F - E: Very little to no attempt made.
Submissions that fail to compile cannot score above an E.
- D - C: Implementations of most functions attempted;
solutions may not be correct, or may not have a good style.
- B: Implementations of all functions attempted, and solutions
are mostly correct. Code style is generally good.
- A: There are no obvious deficiencies in the solution or
the student's coding style. In addition, there is
evidence of productive testing.
- A*: As for an A -- plus the student has done additional work
beyond the basic spec, e.g. by considering (and clearly
commenting) interesting variations or extensions to the
given functions; e.g. based on their own research.