GIST A4

BY ANDREW SIKOWITZ
OVERVIEW FOR A4

• Implement Dictionary and Set modules (well… functors)
• Use them to create a search engine (almost like Google, but for text files, not websites)
• Bisect: Glass box testing framework that checks code coverage on tests
• You will extend your project in A5, with a new Dictionary implementation
  • Build your project (and test suite) with this in mind
A2 DELIVERABLES

• Zip file, created by `make zip`, which requires work in:
  • [listDictionary.ml]: Implement a Dictionary with an association list
  • [dictionarySet.ml]: Implement a Set with a Dictionary
  • [engine.ml]: Crawl through text files and use a Dictionary and Set to map words to the files that contain them
  • [test.ml]: Test suite for everything you implement
  • [authors.ml] and [authors.mli]: Assignment metadata.
  • Also, the <report/> directory, which will contain Bisect’s generated files (more on this later)
Before getting started…

• A new Makefile addition
  • `make bisect`: Build, run tests, then generate the bisect report in the `<report/>` directory
  • You can open `<report/index.html>` to see the code coverage overview

• Regular Expression Level Up! next Monday

• Bisect
  • A code coverage tool, that will analyze what lines of code are executing when running tests
  • Example shown during presentation…
Dictionary Module

- You don’t implement this, but you will need to know its contents
- Comparable: Signature with a type and function for comparing values of that type
- Formattable: Signature with a type and function for representing that type as a string
- KeySig: Signature representing the type of keys in a dictionary, comparable + formattable
- ValueSig: Signature representing the type of values in a dictionary, just formattable
- Dictionary: Signature representing a dictionary. Look here for documentation
- DictionaryMaker: A functor **signature** that takes in a (K:KeySig) and (V:ValueSig), producing a Dictionary
ListDictionary Module

- Implement the functor Make, which is a DictionaryMaker (as per the previous slide)
- Decide on a type [t] that is an association list with the correct types
- Decide how you want to implement the Dictionary, documenting AF and RI
- Implement RI via [rep_ok] and AF via [format]
- Based on them, implement the rest of the functions (documented in <dictionary.mli>)
- Write tests for all the exposed functions!
ListDictionary Tips

• Once again, make sure to **carefully** read the function specifications in `<dictionary.mli>`
  • For example, `[to_list]` returns a **sorted** list representing the Dictionary

• Try to make your implementations tail recursive (details later)

• Make sure you compare keys via the input KeySig module’s `[compare]` function, not the built in comparison operations (e.g. Pervasives.compare, =, <, <=, etc.)
  • Hint: List.assoc, List.assoc_opt, List.mem_assoc, List.remove_assoc all use Pervasives.compare
Implementing [rep_ok]

- [rep_ok x] returns [x] if the representation invariant is satisfied, and raises a Failure exception if it is not
- You can use [rep_ok] in debugging to make sure you never break the RI
  - e.g. check [rep_ok] is satisfied at function start and end for functions taking in a Dictionary
- However, before submitting or running load tests, consider removing [rep_ok] usages
  - You can also replace [rep_ok]’s implementation with the identity function to keep the calls
    - But make sure to keep your original implementation commented out, for graders to read
    - It may have non-constant running time and slow down your program
Implementing [format]

• Meant to be used with [Format.fprintf]

• [Format.fprintf fmt str arg1 ... argN]:
  • [fmt]: a “formatter”, basically specifies where to output
    • [Format.std_formatter] is a formatter that outputs to stdout
    • [Format.str_formatter] is a formatter that outputs to a string buffer
  • [str]: a formatted string, specifying what to output
    • Like C’s printf, you can embed values with “%c” for some character c
      • “%d” for integers, “%s” for strings, “%B” for bools, etc.
      • See https://caml.inria.fr/pub/docs/manual-ocaml/libref/Printf.html for a list of all of them
  • [arg1 ... argN]: embedded values, number and types of which based on the % flags in [str]
• Example (with \[\text{fmt}\] given as argument):
  • \text{Format.printf fmt }"(\%d, \%d, \%s)"\ 5\ 7\ "abc"\ (*\ prints\ "(5, 7, abc)"\ *)

• What about embedding more complex types?
  • Use \"\%a\" and pass as arguments a custom-defined formatting function and the value
  • If the value has type \[t\], format function should have type \[\text{formatter} \to t \to \text{unit}\]

let rec print_list fmt lst =
  match lst with
  | [] -> Format.printf fmt ""
  | h::[] -> Format.printf fmt "\%d" h
  | h::t -> Format.printf fmt "\%d, \%a" h print_list t in
Format.printf Format.std_formatter "[%a]" print_list [1; 2; 3] (*\ prints\ [1, 2, 3]\ to stdout \*)
DictionarySet Module

• Implement the functor Make
  • Takes (E:ElementSig), specifying the type of element in the set (ElementSig is just like KeySig)
  • Takes (DM:DictionaryMaker), which you will use to make a Dictionary module
  • Produces a Set (documentation in <dictionarySet.mli>)

• Most functions can be implemented in very few lines, using Dictionary functions
• Make sure to write your AF and RI, and implement [rep_ok] and [format] as before
• Write tests for all the exposed functions!
Engine Module

• Implement the functor Make:
  • Takes (S: Set) with type Elt.t = string – a set with string elements
  • Takes (D: Dictionary with type Key.t = string and type Value.t = S.t) – a dictionary mapping strings to sets with string elements
  • Produces an Engine (documentation in <engine.mli>)

• The most in-depth part of this assignment

• Requires you to use new libraries: Unix, Pervasives’ I/O, possibly Str
Engine Module Breakdown

• The bulk of the work is in [index_of_dir]
  • Crawl through the filenames in a directory using Unix.(opendir, readdir, closedir)
  • Crawl through the text in a file using Pervasives.(open_in, input_char / input_line, close_in)
  • Pick out the “words” (specifically defined) in the text of each file
  • Return a mapping of words to the set of files that had an occurrence of them

• [words], [to_list], and [format]: Use your Dictionary functions

• [or_not] and [and_not]: Use your Set functions
[index_of_dir] tips

• Note that crawling through a directory and through a file are very similar:
  • open, read next, and close functions
  • [readdir] and [input_char] / [input_line] raise [End_of_file] when iteration is over

• Be careful about filenames vs. file paths
  • [readdir] returns a file name, but [open_in] expects a path
  • Which format should your [idx] store?

• Indexing directories with large text files can be slow, but must not cause stack overflow
  • Use tail recursion. This applies to ListDictionary and Set functions used while indexing too!
[index_of_dir] tips continued

- Looking at `<engine.mli>`:
  - Convert all words to lowercase via `String.lowercase_ascii` before putting them in your index
  - Do not change the case of filenames

- Parsing words:
  - Make sure to read the definition of a “word” very carefully
  - Practice OCaml regular expressions via the `Str` module beforehand, or don’t use regex at all!
OCaml Regular Expressions

• Define a regular expression with \[regexp\ pattern\], where pattern : string
• Look for a match with \[string\_match\] or \[search\_forward\] (or other functions)
• Once a match is found, get the actual string matched with \[matched\_string\]
• Also: \[split\], split on a pattern to produce a list of strings
  • This one is great! Simple and easy to use
  • The others ones… less so
• Make sure to test your regular expressions before moving on!
  • You can make your word-finding function top-level and expose it in <engine.mli> if you want
OCaml Regular Expressions Tips (and Warnings)

• When making a group, you must escape the parentheses: \(\langle\text{pattern here}\rangle\)
• To use *, +, or ? on a group of characters, you must put them in a group (as above)
• \text{string_match r s i} will return false if the pattern doesn’t start at index \(i\)
  • You shouldn’t try \text{string_match} on every possible \(i\)
  • If you find yourself wanting to do that, use \text{search_forward} instead
• \text{search_forward r s i} will find matches for all indices \(\geq i\)
  • But it will raise \text{Not_found} if a match is not found, so you have to catch that
• \text{matched_string s} Call it on the same string you called \text{string_match} or \text{search_forward}
  • You can only call this after calling one of them
  • \text{matched_group i s}: Like \text{matched_string}, but will pick out a specific group within your regular expression
FINAL TIPS

• Look at .mli files (again). Every function is very well-documented
• Try to implement all the functionality, including the excellent scope
  • You will have to do it for A5, anyway
• Implement and test piece by piece
  • This project can be broken into small, testable portions pretty well