Computer Science 15–212: Final Examination

Robert Harper

May 6, 1997

Name

Name: ____

Andrew ID: ____

Instructions

- This is a open-book, open-notes examination.
- There are 15 pages in this examination, including three worksheets.
- This examination consists of 5 questions worth a total of 100 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, indicate clearly that you have done so on the front.
- Write your name and Andrew id in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- Two worksheets are attached to the end of this examination for your personal use.

Grading

Question	1	2	3	4	5	Total
Score						
Maximum						100

1 Using Abstractions [20 Points]

Assume that you are given an implementation `Set` of the signature `SET` as follows:

```
signature SET = sig
  type 'a set
  val empty: 'a set
  val isEmpty: 'a set -> bool
  val insert: 'a set * 'a -> 'a set
  val choose: 'a set -> 'a * 'a set
  val union: 'a set * 'a set -> 'a set
  val map: ('a -> 'b) -> 'a set -> 'b set
end

structure Set :> SET = struct ... end
```

The operations are as follows: `empty` is the empty set, `isEmpty` tests for the empty set, `insert` adds an element to a set, `choose` selects an arbitrary element of a given set and returns that element together with the set of remaining elements, `union` forms the union of two sets and `map` applies a function to every element of a given set.

Give a definition of the function `power` of type

```
'a Set.set -> 'a Set.set Set.set
```

that returns the power set of a given set. Your solution should be written entirely in terms of the operations provided by the structure `Set`:

2 Persistent and Ephemeral Data Structures [20 Points]

Consider the following "mutable cell" abstraction. Mutable cells are similar to ML `ref`'s, except that they may or may not contain a value. In particular, we are not required to provide an initial value, and, moreover, the cell may be reset to an uninitialized state.

```
signature CELL = sig
  type 'a cell
  (* create a new, uninitialized cell *)
  val new : unit -> 'a cell
  (* get the value (if any) of the given cell *)
  val get : 'a cell -> 'a option
  (* set the contents of the cell to the specified value *)
  val set : 'a cell * 'a -> unit
  (* return the cell to its un-initialized state *)
  val reset : 'a cell -> unit
end

structure Cell :> CELL = ...
```

The following declaration sequence illustrates the use of cells.

```
val c = Cell.new ();
val x = Cell.get c;
val _ = Cell.set (c, 3);
val y = Cell.get c;
val _ = Cell.reset c;
val z = Cell.get c;
```

After evaluation of this sequence of declarations, the variables `x` and `z` have the value `NONE` and the variable `y` has the value `SOME 3`.

A mutable cell is an ephemeral data structure — performing a `set` operation replaces the old contents of the cell with the new value. In some situations it is useful to be able to "snap back" the contents of a cell to a previous state, undoing the effects of intervening assignments. One way to implement this idea is to use continuations. Specifically, your task is to implement the function

```
val set_temporarily : 'a Cell.cell * 'a * ('a Cell.cell -> 'b) -> 'b
```

When passed a cell `c`, a new value `v`, and a continuation `k`, the function `set_temporarily` sets `c` to `v`, passes the (modified) cell `c` to `k`, and, when `k` returns, resets ("snaps back") `c` to its previous state. The result of a call to `set_temporarily` is the value returned by the continuation. Continuing the above example, the following declarations

```
val a = set_temporarily (c, 6, fn c => Cell.get c);
val b = Cell.get c;
```

bind `a` to `SOME 6` and `b` to `NONE`.

1. [10 Points]

   Give a definition of the function `set_temporarily` described above. *Ignore the possibility that the continuation may raise an exception.*

   ```
   fun set_temporarily (c : 'a Cell.cell, v : 'a, k : 'a Cell.cell -> 'b):'b =
   ```

2. [10 Points]

   Re-implement `set_temporarily`, taking into account the possibility that the continuation may raise an exception. If it does, the cell should be reset to its prior state, and the call to `set_temporarily` should re-raise the same exception raised by the continuation.

   ```
   fun set_temporarily (c: 'a Cell.cell, v : 'a, k : 'a Cell.cell -> 'b):'b =
   ```

3 Implementing an ADT [20 Points]

Consider the following simple signature of string dictionaries:

```
signature DICT = sig
    type dict
    val empty : dict
    (* insert(D,s) inserts the string s into the dictionary D. *)
    val insert : (dict*string) -> dict
    (* Check whether the given string is in the dictionary *)
    val find : (dict*string) -> bool
  end
```

One useful implementation of a dictionary is a data structure called a *trie* (pronounced "try"). A trie is a tree structure that is used to represent sets of strings in which many of the strings share a common prefix. The idea of a trie is to minimize redundancy by sharing common prefixes among the entries in the dictionary. For example, in a dictionary of English words, both "anther" and "anthem" would occur. We may save space by only saving one copy of the common prefix "anthe", followed by two branches - one containing the suffix "r" and one containing the suffix "m". We can also take advantage of the fact that some prefixes are (or contain) words, such as "an" in "anthe". Since common prefixes are shared among words, and some prefixes may be words ("an"), but not all prefixes are words ("anthe"), we must mark those prefixes that correspond to words in the dictionary.

Tries may be implemented using the following type declaration:

```
datatype Trie =  EMPTY | TERM of Trie | TRIE of char*Trie*Trie*Trie
```

See figure 1 for an example of a trie created by the following sequence of `insert` operations:

```
val D = insert(empty,"an");
val D = insert(D,"anthem");
val D = insert(D,"anther");
val D = insert(D,"and");
val D = insert(D,"any");
```

What strings does a trie contain? First, the trie `EMPTY` contains no strings at all. Second, a word w is an element of the trie `TERM` T if either w is the null string, or w is a string in T. Finally, a word w is in the trie `TRIE` (c, L, E, G) if either w is a string in L, or w is a string in G, or $w = c\,w'$ where w' is a string in E.

Returning to the trie in Figure 1, notice that the `TERM` nodes indicate stopping places - that is, points in a prefix where a valid word has been completed. In terms of the formal definition, the set of strings in the **N** trie is $\{\} \cup \{\mathrm{n}, \mathrm{nd}, \mathrm{nther}, \mathrm{nthem}, \mathrm{ny}\} \cup \{\}$. (The two empty sets represent the sets of strings from the left and right nodes, both `EMPTY`). Note that the string "n" is in the **N** trie only because the middle node is a `TERM` node. This means that ϵ is in the middle sub-trie, which when concatenated with "n" yields "n". The strings in the trie as a whole are the strings in the **A** trie, which are simply the strings in the **N** trie with "a" glued to the front. Since the **T** node does not have a `TERM` node as "middle" child, the string "nt" is not in the **N** trie, and hence the word "ant" is not in the trie as a whole.

To facilitate search, tries satisfy the representation invariant that for every node of the form `TRIE` (c, L, E, G) in a trie, the strings in L start with a character less than c in alphabetical order,

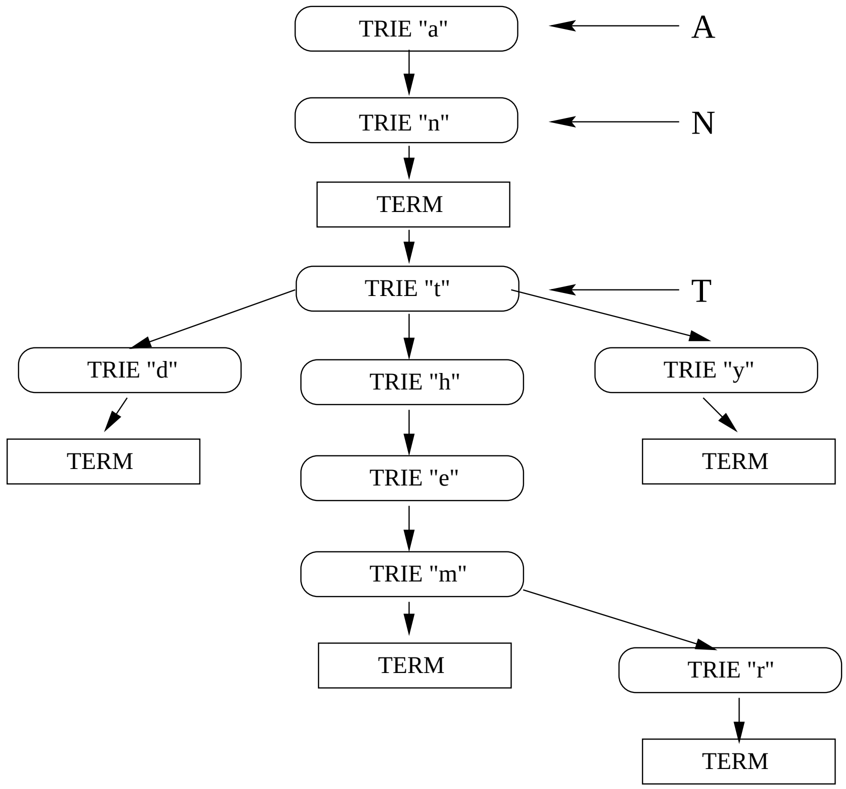

Figure 1: Example Trie (`EMPTY` nodes are omitted)

and that the strings in G start with a character greater than c in alphabetical order. Consequently, all strings *starting* with c have their "tails" (obtained by dropping the first character) in E.

Complete the following implementation of the structure Trie. The functions insert and find are already defined for you in terms of functions `insert'` and `find'` that operate on lists of characters instead of strings. You should implement functions `insert'` and `find'`.

For extra credit, implement the function `toList` of type `dict -> string list` that returns a list of all words in the trie.

```
structure Trie :> DICT  =
  struct
    (* Representation invariant as described above. *)
    datatype Trie =  EMPTY | TERM of Trie | TRIE of char*Trie*Trie*Trie
    type dict = Trie
    fun insert'









    fun find'









    fun insert (T,s) = insert' (T,explode s)
    fun find (T,s) = find' (T, explode s)
    fun toList = (Extra Credit)









  end
```

4 Streams [20 Points]

In this question we will implement a simulator for a simple computer. The computer will have fixed number of registers that hold integer values. Computation is performed by changing the values in the registers according to a specific set of instructions. The computer recognizes the following instructions:

```
datatype instr =
  Add of int * int * int
| Sub of int * int * int
| Mul of int * int * int
| Const of int * int
| Loop of int * instr list
```

A program consists of a list of instructions. The `Add`, `Sub`, and `Mul` instructions are executed by performing the appropriate arithmetic operation on the contents of the first two registers, storing the result in the third register. The `Const` instruction loads the integer value given as first argument into the register specified as the second argument. The `Loop` instruction repeatedly executes the program (instruction list) given as second argument as long as the register specified in the first argument contains a non-zero value.

As an example, the following program computes the factorial of register one in register two:

```
[ Const(1, 2),        load 1 into register 2
  Const(1, 3),        load 1 into register 3
  Loop(1, [           loop until register 1 is zero
    Mul(1, 2, 2),     multiply register 2 by register 1
    Sub(1, 3, 1)      decrement register 1
  ])
]
```

Note that register three is used solely to hold the value 1.

A simulator for this machine can be implemented using streams. The *loop unit* of the simulator takes a program (list of instructions) and generates a stream consisting only of the arithmetic and constant instructions — the loops are "unrolled" by the loop unit into streams of individual instructions. The *processing unit* executes each of the instructions in the stream until no further instructions remain to be executed.

Returning to the factorial example, the loop unit will generate the following stream of instructions:

```
Const(1, 2),
Const(1, 3),
Mul(1, 2, 2),
Sub(1, 3, 1)
Mul(1, 2, 2),
Sub(1, 3, 1)
   ⋮
Mul(1, 2, 2),
Sub(1, 3, 1)
```

The number of multiply and subtract instructions is equal to the value of register one at the start of the program.

You are given the following implementation of machine instructions:

```
signature MACHINE = sig
  (* The type of machine instructions. *)
  datatype instr =
    Add of int * int * int
  | Sub of int * int * int
  | Mul of int * int * int
  | Const of int * int
  | Loop of int * instr list
  (* Return the value of a particular register. *)
  val register: int -> int
  (* Execute a given instruction, updating the registers accordingly. *)
  val execute: instr -> unit
end

structure Machine :> MACHINE = ...
```

The function `register` returns the contents of the specified register. The function `execute` executes one instruction, updating the registers as described above.

1. [5 Points]

   Give an implementation of the function `process` of type

   ```
   Machine.instr Stream.stream -> unit
   ```

   which, when applied to a stream of non-loop instructions executes each in the order given. Your solution should be defined in terms of the implementation `Stream` of finite and infinite streams given in class.

   ```
   fun process instrs =
   ```

2. [15 Points]

   Implement the function `unroll` of type

   ```
   Machine.instr list * Machine.instr Stream.stream -> Machine.instr Stream.stream
   ```

   that, when applied to a list `instrs` of instructions and an instruction stream `rest`, yields the stream consisting of the instructions generated by unrolling all of the loops in `instrs`, followed by the instructions in the stream `rest`. Loop unrolling is controlled by consulting the registers using the `Machine.register` function to determine when to stop unrolling — the instructions specified in a `Loop` instruction are repeatedly issued to the result stream until the specified register's contents becomes zero.

   ```
   fun unroll (instrs, rest) =
       let
           fun gen () =













       in
           Stream.stream gen
       end
   ```

5 Regular Expressions and Grammars [20 Points]

Regular expressions and context-free grammars are two different formalisms for defining languages. In this question you are asked to relate these two formalisms.

First, recall the definitions of regular expressions and grammars. A regular expression has the form $\mathbf{0}$, $\mathbf{1}$, $\mathbf{a}$ (where a is a character), $\mathbf{r}_1 + \mathbf{r}_2$, $\mathbf{r}_1\,\mathbf{r}_2$, or $\mathbf{r}^*$. The language $L(\mathbf{r})$ of a regular expression $\mathbf{r}$ is the set of strings matched by $\mathbf{r}$, as defined in the course notes. A context-free grammar consists of a set of rules of the form $A \to \alpha$ where A is non-terminal (variable) of the grammar and α is any string of terminals (characters) and non-terminals. Each grammar comes with a start symbol S, the non-terminal with which we start derivations in that grammar. The language $L(G)$ of a grammar G is the set of strings derivable from the start symbol S.

Context-free grammars are more powerful than regular expressions in that every language definable by a regular expression is also definable by a context-free grammar. More precisely, given a regular expression $\mathbf{r}$, there is a grammar $G_{\mathbf{r}}$ with start symbol $S_{\mathbf{r}}$ that defines the same language as $\mathbf{r}$ (*i.e.*, $L(G_{\mathbf{r}}) = L(\mathbf{r})$).

1. [5 Points]

   Consider the regular expression $\mathbf{r} = (\mathbf{a}+\mathbf{b})^*(\mathbf{a}+\mathbf{b})$. Give a context-free grammar that accepts the same language as $\mathbf{r}$. Use S as the start symbol of the grammar. Feel free to introduce additional non-terminals and rules as appropriate.

2. [5 Points]

   Exhibit a leftmost derivation of the string $aaba$ using the rules of the grammar given in part (1).

3. [10 Points]

   Define by induction on the structure of $\mathbf{r}$ a grammar $G_{\mathbf{r}}$ (with start symbol $S_{\mathbf{r}}$) that accepts the same language. More precisely, for each regular expression $\mathbf{r}$, give a set of rules (possibly empty!) for $G_{\mathbf{r}}$ so that the grammar accepts the same language. *Hint: Think inductively. Only one or two lines are required for each case.* To get you started we have given one case of the answer:

Regular Expression	*Production Rules*
$\mathbf{1}$	$S_{\mathbf{1}} \to \epsilon$
$\mathbf{0}$	
$\mathbf{a}$	
$\mathbf{r}_1 + \mathbf{r}_2$	
$\mathbf{r}_1\,\mathbf{r}_2$	
$\mathbf{r}^*$	

Worksheet

Worksheet

Worksheet