Computer Science 15–212–X: Midterm Examination

October 2, 1997

Name

Name: ________________________________

Andrew ID: ________________________________

Instructions

- This is a closed-book, closed-notes examination.
- There are 12 pages in this examination, including two worksheets.
- This examination consists of 3 questions worth a total of 100 points, plus an extra-credit problem worth 20 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- You do not need to re-state the given invariants, but you should annotate auxiliary functions you define. Your answers should be correct, simple and clean, and they should take advantage of the given invariants. In other respects, your functions do not need to be particularly efficient, and they do not need to be tail-recursive unless that is explicitly stated.
- 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	Total	Extra Credit
Score					
Maximum	35	30	35	100	20

1 Induction and Recursion [35 points+20 extra credit]

A polynomial p in a variable x of degree $n \geq 0$ can be written as

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where $a_n \neq 0$ unless $n = 0$. For the purpose of this question, we assume the coefficients a_i for $0 \leq i \leq n$ are integers and that the variable x also ranges over integers.

Since the name of the parameter does not matter, we can represent such a polynomial by a list

$$\overline{p} = \begin{cases} \texttt{[}\overline{a_0}\texttt{,}\ \overline{a_1}\texttt{,}\ \texttt{...,}\ \overline{a_n}\texttt{]} & \text{if } n > 0\text{, or } n = 0 \text{ and } a_0 \neq 0 \\ \texttt{[]} & \text{if } n = 0 \text{ and } a_0 = 0 \end{cases}$$

with the invariant that $a_n \neq 0$. Make sure your functions below take advantage of the invariants on the arguments and guarantee them on the results.

1. [2 pts] Define the representation type `poly`.

2. [8 pts] Write a function

   ```
   val polyToFun : poly -> (int -> int)
   ```

   which converts a polynomial to the function it represents on integers.

3. [10 pts] Prove that

$$\texttt{(polyToFun } \overline{p}\texttt{)}\ \overline{x} \Longrightarrow \overline{p(x)}$$

for any polynomial p and integer argument x. Make sure to explicitly state both the induction principle you use and your induction hypothesis.

4. [10 pts] Adding two polynomials in the same variable yields another polynomial. Write a function

```
val add : poly * poly -> poly
```

to compute the sum of two polynomials.

5. [5 pts] Using tail recursion, write a function

```
val degree : poly -> int
```

to compute the degree n of a polynomial.

6. [20 pts extra credit] Write a function

```
val binomial : int -> poly
```

to compute the polynomial representation of the binomial $b_n(x) = (1+x)^n$. Avoid unnecessary recomputation to obtain full extra credit.

2 Abstract Types [35 points]

Consider a signature for finite sets of natural numbers, that is, integers greater or equal to 0.

```
signature NATSET =
sig
  type natset
  exception Negative
  val empty : natset
  val singleton : int -> natset (* may raise Negative *)
  val union : natset * natset -> natset
  val intersection : natset * natset -> natset
  val member : int * natset -> bool
end;
```

Each of the operations listed in the signature has an obvious specification. The `member` function is defined only for natural numbers in the first argument, but may return `false` instead of raising an exception when called on a negative integer. The function `singleton` instead is *required* to raise an exception when called with a negative argument.

1. [25 pts] Implement a structure `N :> NATSET` where integer sets are represented as ordered lists of integers without duplicates. Make sure your functions take advantage of this invariant on the arguments and guarantee it for the results.

2. [10pts] Now assume we have a correct implementation `N :> NATSET`. For each of the following declarations,

   - either give its type or indicate a type error and explain the reason for the type error, and
   - if the declaration is well-typed, give the value binding it produces or the exception raised during its evaluation, and
   - if the evaluation produces a value binding, indicate if the value is observable, that is, if the type of the expression is abstract or concrete in the environment of the declaration.

   (a) `val s1 = N.union(N.singleton 4, N.singleton(3));`

   (b) `val s2 = N.union(s1, N.singleton (~2));`

   (c)
   ```
   val s3 = N.member (2,
                let
                    val sg = N.singleton
                 in (N.union(sg 2, N.union (sg ~2, sg 4)))
                    handle N.Negative => N.empty
                end);
   ```

   (d) `local open N in val s4 = singleton(~2) handle Ngative => empty end;`

   (e) `local open N in val s5 = union ([1,2,3], N.union (s1, s1)) end;`

3 Higher-Order Functions [30 pts]

Repeatedly stung by missing, incorrect, or ignored invariants, a software developer decided to enrich his functions with filters which check whether the argument lies in the intended domain of application. For this purpose he wrote the following signature.

```
signature FILTER =
sig
  type 'a filter = 'a -> 'a  (* may raise Domain, otherwise identity *)
  val filter : 'a filter -> ('a -> 'b) -> ('a -> 'b)
  val all : 'a filter
  val both : 'a filter * 'a filter -> 'a filter
  val either : 'a filter * 'a filter -> 'a filter
  val not : 'a filter -> 'a filter
  val predToFilter : ('a -> bool) -> 'a filter
end;
```

A function of type t `filter` must either return its argument or raise the exception `Domain`. We say the filter *succeeds* if it returns its argument and *fails* if it raises the exception. Note that the exception `Domain` is not exported in the signature, since a call to a function outside its domain should be an irrecoverable error. The intended use is to replace suspect function applications e_1 e_2 by `filter` *fil* e_1 e_2, where *fil* is a filter appropriate for e_1.

1. [25 pts] Implement a structure `Filter :> FILTER` satisfying the following specifications.

```
structure Filter :> FILTER =
struct
  type 'a filter = 'a -> 'a
```

(a) `filter` *fil* f returns the function which behaves like f if the filter succeeds and raises exception `Domain` if the filter fails.

(b) `all` is the filter that always succeeds.

(c) `both` (*fil*$_1$, *fil*$_2$) succeeds if both *fil*$_1$ and *fil*$_2$ succeed, otherwise fails.

(d) `either` $(\mathit{fil}_1$`,` $\mathit{fil}_2)$ succeeds if either fil_1 or fil_2 succeeds.

(e) `not` *fil* succeeds if *fil* fails, and fails if *fil* succeeds.

(f) `predToFilter` p returns the filter which succeeds on x if $p(x)$ evaluates to `true` and fails otherwise.

```
end;  (* structure Filter *)
```

2. [5pts] Given a correct implementation of `Filter :> FILTER` in the current environment, write the following two simple filters.

   (a) `val nat : int filter` which checks if the argument is greater or equal to zero.

   (b) `val unitCircle : (real * real) filter` which checks if the argument represents a point on or inside the circle of radius 1 about the origin.

Worksheet

Worksheet