CSCI 3155: Principles of Programming Languages
Exam preparation #2
2nd July 2007

Exercise 1. Generics/templates present a useful means for writing generic ADTs without losing type information.

(a) In C++ or Java, implement a generic *Stack* class. The class should store reference/pointer types only (in Java, this is implicitly ensured).

(b) Assume that you do allow null references/NULL pointers to be pushed. What reasonable options exist to signal an error if a client tries to pop off an empty stack? If you suggest the use of an exception, should the exception be checked or unchecked (language facilities permitting)?

(c) Assume that you do not allow null/NULL references/pointers to be pushed. What reasonable options exist to signal an error if a client tries to pop off an empty stack? If you suggest the use of an exception, should the exception be checked or unchecked (language facilities permitting)?

(d) Assume that, as an embarrassing implementation artifact, your stack only has a limited capacity (such as the size of the machine's main memory). What reasonable options exist to signal an error if a client tries to push more entries than your implementation can fit? If you suggest the use of an exception, should the exception be checked or unchecked (language facilities permitting)?

Exercise 2. Consider the following SML function:

```
val fold : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

This function allows us to compute

$[\![$`fold (op+) (0) []`$]\!] = 0$
$[\![$`fold (op+) (0) [1]`$]\!] = 1$
$[\![$`fold (op+) (0) [1,2]`$]\!] = 3$
$[\![$`fold (op+) (0) [2,3,4,7,9]`$]\!] = 26$
$[\![$`fold (op*) (1) [1,2,3]`$]\!] = 6$

Such functions are very popular in SML programming (in fact, SML provides such functions for all container types of its standard library). Effectively, the function "folds" the elements of its parameter list together, using the binary function provided to it:

```
⟦fold (op+) (0) [2,3,4]⟧ = ⟦ 0+2+3+4 ⟧
```

Note how the second parameter is used as a neutral element:

```
⟦fold (op+) (0) []⟧ = ⟦ 0 ⟧
⟦fold (op+) (0) [1]⟧ = ⟦ 0+1 ⟧
```

(a) What is the type of "fold (op+)"?

(b) Implement the function fold, without using the built-in hd or tl functions.

Exercise 3. Consider subrange and function types. Determine which of the following function types are subtypes of each other (annotate with :>, <: or // to indicate that they are neither sub- nor supertype).

```
[1 TO 15] -> [2 TO 20]          [1 TO 15] -> [1 TO 10]
[1 TO 10] -> [100 TO 200]       [10 TO 10] -> [100 TO 100]
[1 TO 15] -> [2 TO 20]          [1 TO 15] -> [2 TO 10]
[1 TO 15] -> [2 TO 20]          [1 TO 25] -> [1 TO 10]
[5 TO 10] -> [1 TO 20]          [1 TO 25] -> [1 TO 10]
```

Exercise 4. Functional languages such as SML usually support both tuple and list types.

(a) Give two differences between tuples and lists.

(b) Implement an `'a ilist` that allows the storage of both int and an arbitrary value of type 'a.

(c) Write a function sum : real ilist → real that sums up all elements in a real ilist.

(d) Write a function intify : 'a ilist → int list that eliminates all 'a entries in your numlist but retains all int entries in order. What does your function do on an int ilist?

Exercise 5. The Eiffel programming language supports an interesting type called like Current. This type represents the type of the current class, *even after inheritance*. For example, if class A has a field next of type like Current and B is a subclass of class A, then B will inherit a field next of type like Current. For class A, the field next will have type A, for class B, the field will have type B.

(a) One possible use of this feature is to implement a clone function (to duplicate the current object) that always has the proper return type. In Eiffel terms,

```
class interface CLONEABLE
feature
    clone : like Current -- returns clone of current object
end
```

This defines a method clone with the like Current type.

Assume that a class A implements the above interface, and class B inherits from A. Ignoring all other changes, will A :> B?

(b) Another possible use of this feature is the implementation of typesafe equals_to methods, to compare the current object to an object *of the same type only* (or of a subtype, of course). In Eiffel terms,

```
class interface CLONEABLE
feature
    compare_to (other : like Current) : BOOLEAN
    -- determines whether the other object is the same
    -- as this object
end
```

This defines a method compare_to with a parameter of type like Current.

Assume that a class A implements the above interface, and class B inherits from A. Ignoring all other changes, will A :> B?

Exercise 6. Classify the following features as to whether they are compatible with *referential transparency*, and explain:

(a) Integer addition
(b) Variable assignment
(c) Dynamic scoping
(d) String concatenation
(e) Pass-by-result
(f) print statements

Exercise 7. Standard ML supports exceptions with the syntax

```
raise Match
```

(here, Match is a built-in exception). This expression has the type 'a.

(a) Write a program that determines whether SML's andalso expressions are subject to short-circuit evaluation.

(b) Is short-circuit evaluation compatible with referential transparency? Explain.

Exercise 8. Overloading and overriding are often confused in object-oriented programming.

(a) When do languages resolve *overloading*?

(b) When do languages resolve *overridden methods*?

(c) Which of the two is related to *dynamic dispatch*?

(d) In object-oriented languages, objects have two relevant types: a *dynamic* type and a *static* type. Which of the two is used to resolve dynamic dispatch?

(e) Let $B <: A$. Now consider two methods $voidf(Aa)$ and $voidf(Ba)$. Which of the two methods can conceivably *override* the other?

(f) Let again $B <: A$ and consider two methods $voidf(Aa)$ and $voidf(Ba)$. In some languages, one of these methods can override the other, in other languages, overloading is used instead. Write a program that determines which of the two mechanisms your language uses.

Exercise 9. Below are the first nine Roman numerals, in order, starting at the numeral representing the number 1:

```
I, II, III, IV, V, VI, VII, VIII, IX
```

The same pattern as for `I`,`V`,`X` repeats for `X`,`L`,`C`, representing the numbers 10, 20, ..., 100. Writing one of the latter numerals before one of the earlier numerals allows us to combine the two; for example, `LVI` represents 54, `XCIX` represents `99`, and `XLII` represents 42.

(a) Write a syntax to represent precisely all roman numerals up to `XCIX`. Do not allow any objects that are not Roman numerals to be represented. Use fewer than 20 rules.

(b) Write a denotational semantics that maps roman numerals to the natural numbers.

Exercise 10. Polymorphism is a useful mechanism for re-using code.

(a) What are the two kinds of polymorphism we discussed in class, and what do they do?
(b) Give an example for each of the two kinds of polymorphism.

Exercise 11. Consider the pocket calculator example from the early denotational semantics handout (Handout A).

(a) Implement an SML `datatype` that can express all constructions that the pocket calculator allows.
(b) Implement an SML function that evaluates all constructions that your datatype allows.
(c) Unless you already have done so, ensure that you do not divide by zero. What is the return type of your evaluation function?

Exercise 12. Recall the syntax and semantics of two-valued boolean logic:

$$
\begin{aligned}
[\![\mathsf{TRUE}]\!] &= \top \\
[\![\mathsf{FALSE}]\!] &= \bot \\
[\![\mathsf{NOT}(x)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \bot \\ \bot & \iff [\![x]\!] = \top \end{cases} \\
[\![\mathsf{AND}(x,y)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \top \text{ and } [\![y]\!] = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathsf{OR}(x,y)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \top \text{ or } [\![y]\!] = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathsf{IMPLIES}(x,y)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \bot \text{ or } [\![y]\!] = \top \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\top$ and $\bot$ are two values in some set we use to interpret two-valued boolean logic.

(a) Reconstruct the datatype you used to represent expressions of two-valued boolean logic. Note that we must usually deal with variables occurring in expressions, at least in real programs. Add a value constructor

```
VAR : string -> boolexpr
```

to your datatype.

(b) Observe that the semantics of IMPLIES look very similar to the semantics of OR. Explain how you can use OR and NOT to describe the (semantically) same construct.

(c) In compiler implementation, it is often helpful to simplify expressions as much as possible: if our expressions are less complex, we need less complex logic to deal with them. Write a program map_deimply that traverses through boolexpr and applies the simplification you implemented above.

(d) Implement a function

$$\mathsf{mapb} : (\mathsf{boolexpr} \to \mathsf{boolexpr}) \to \mathsf{boolexpr} \to \mathsf{boolexpr}$$

that performs the *traversal* part of map_deimplify only. Your function takes two parameters, a function f and a boolexpr e. Your function should visit any sub-boolexpr of e and apply f to it.

(e) Reimplement map_deimplify using mapb.

(f) Find one or two logical simplifications you can apply to boolexprs (hint: consider double negation, DeMorgan's laws if you know them, or how AND/OR behave if one of their parameters is TRUE or FALSE). Write a function that applies these simplifications *locally*. Apply your simplification via mapb.