Name: ***Solutions***

Due Monday, December 14, at 10:30am in Craig Chambers's office.

Take in a single 5-hour period, self-timed. You may refer to any of your notes, lecture slides, or course readings, but you may not use the computer to experiment or verify your answers. You may use a computer to compose any subset of your answers. Turn in a hard-copy of all your answers.

150 points total.

1) In Scheme, an arbitrarily nested list can be used to represent a tree-like structure. The `leaves` function takes such a nested list and returns a flat list collecting together the non-`nil` leaves of the original nested list. E.g.

```
(leaves '(3 (4 5) ((6) ()) 7)) ⇒ (3 4 5 6 7)
```

a) [4 pts] Implement `leaves` in Scheme.

```
(define (leaves l)
   (cond ((null? l) l)
         ((atom? l) (list l))
         (else (append (leaves (car l))
                       (leaves (cdr l))))))
```

b) [2 pts] Why is `leaves` of a list impossible to implement in ML?

Arbitrarily-nested lists are not homogenous and so don't typecheck. Lists nested uniformly are OK; lists whose elements have different nesting depth aren't.

c) [4 pts] A variation on `leaves` is possible in ML using explicit trees. Implement `leaves` in ML over the following `Tree` data type declaration:

```
datatype 'a Tree = EmptyTree
                 | Leaf of 'a
                 | Node of 'a Tree * 'a Tree;
```

(Recall that `@` is the infix `append` operator in ML.)

```
fun leaves(EmptyTree)    = []
  | leaves(Leaf(x))      = [x]
  | leaves(Node(t1,t2)) = leaves(t1) @ leaves(t2);
```

d) [3 pts] What is the most general type of the ML `leaves` function?

```
leaves: 'a Tree -> 'a list
```

2) [6 pts] Write an ML signature `SYMBOLTABLE` and a corresponding structure `SymbolTable` which implements a symbol table. A symbol table maintains a mapping from identifiers (i.e. strings) to data; symbol tables are polymorphic in the type of the data stored in the symbol table. The symbol table should support the following values and operations:

- the `empty_table` value,
- the `lookup` function which takes an table and an identifier and either returns the associated data or raises the `NotFound` exception,
- the `insert` function, which takes a table, a key, and a value, and returns a new table with the additional key/value binding, and
- the `NotFound` exception.

Use the appropriate encapsulation declarations to hide your choice of implementation of the symbol table. Your implementation need not be very efficient; linear-time lookup and/or insertion is acceptable.

```
signature SYMBOLTABLE = sig
  type 'data Table;
  val empty_table: 'data Table;
  val lookup: 'data Table -> string -> 'data;
  val insert: 'data Table -> string -> 'data -> 'data Table;
  exception NotFound;
end;

structure SymbolTable:>SYMBOLTABLE = struct
  type 'data Table = (string * 'data) list;

  exception NotFound;

  val empty_table = nil;

  fun lookup nil key = raise NotFound
    | lookup ((k,v)::assocs) key =
       if key = k then v else lookup assocs key;

  fun insert assocs key value = (key,value)::assocs;
end;
```

3) a) [2 pts] What is the main feature provided by abstract data types that is not provided by concrete data types?

Encapsulation of implementation details.

b) [3 pts] What advantage do abstract data types bestow on **clients** that is not provided by concrete data types?

Clients only have to pay attention to the published interface; they are not bothered with implementation details.

c) [3 pts] What advantage do abstract data types bestow on **implementors** that is not provided by concrete data types?

Implementors can feel free to change implementation details that are hidden from clients. Clients will not be affected.

d) [3 pts] What is the main advantage for software maintenance of object-oriented programming over procedural programming extended with abstract data types? How do the language features in object-oriented languages contribute to this advantage?

The main advantage is that programs are easier to extend. Programs can be extended by defining new subclasses of existing classes; client code need not be affected if the new subclass supports the interface of the existing superclass.

4) ML-style pattern-matching and overloaded dynamically-dispatched methods are superficially similar.

a) [4 pts] What feature do multimethods and ML-style pattern-matching share, that is not shared by singly-dispatched methods?

"Dispatching" or testing on the "class" or tag of all the arguments, not just the first.

b) [4 pts] What important feature do multiply- and singly-dispatched methods support that ML functions defined through pattern-matching do not?

Being extended to handle new cases (subclasses) w/o modifying the existing cases.

Being able to inherit cases from one data constructor to a more specific one.

5) Haskell's type classes give a flavor of object-oriented programming. But it has strengths and weaknesses relative to true OO classes/objects.

a) [4 pts] Explain what language properties enable data structures like lists of mixed integers and floats to be programmed (and statically typechecked!) easily in OO languages but not in Haskell.

OO languages support run-time subtyping from a statically declared type, while Haskell does not. E.g., in the OO language, I can instantiate the `list[T]` parameterized type as `list[num]`, and then store any subtype of `num` into the list at run-time. In Haskell, I have to pick a particular type to instantiate the `List` type class to, and then the list must be homogeneous, holding only values of that single type at run-time.

b) [5 pts] Haskell's type classes allow operations to be attached to type classes that mention the type class's type variable multiple times, e.g.,

```
class Foo f where
  bar :: (f,f) -> f
  bar (x1,x2) = ...
```

```
data Instance = ...

instance Foo Instance where
  bar (x1,x2) = ...
```

Give the analogous Cecil `Foo[f]` and `Instance` objects and two `bar` methods. Make sure that the `bar` methods are just as precise in terms of its allowed combinations of argument types and the result type corresponding to each combination of argument types as the Haskell declaration, except that Cecil's `bar` methods should allow appropriate subtype polymorphism. Hint: the type parameter `f` will need to be bounded using an F- or signature-bound.

```
abstract object Foo[f <= Foo[f]];
method bar(x1:`f <= Foo[f], x2:`f):f { ... }

template object Instance isa Foo[Instance];
method bar(x1@:Instance, x2:Instance):Instance { ... }
```

c) [3 pts] Why is a simpler bounding technique, like plain bounded polymorphism, insufficient to program this example in an OO language?

Because it's not possible to express that whatever the first argument is, the second argument and result is the same kind of thing.

d) [3 pts] Eiffel's use of covariant typing and `like Current` makes this example easy to program. In pseudocode:

```
class Foo {
  method bar(x2:like Current):like Current { ... }
};

class Instance isa Foo {
  method bar(x2:Instance):Instance { ... }
};
```

What is a key drawback of Eiffel's typing model? How does it apply to this example?

It is unsound. I can invoke `((Foo)(new Instance)).bar(new Foo)`, which the type system allows but which may do bizarre things at run-time.

e) [4 pts] How can the use of multimethods improve on Eiffel's approach while preserving simple type declarations? How does this change the meaning of the program compared to that implemented in part b?

Just add dynamic dispatching on the argument to `Instance's` `bar` method, so it only applies if the argument is an `Instance` as well. But it allows applying `bar` to mixes of `Instances` and other subclasses of `Foo` that aren't `Instances`, which is statically disallowed by the solution in part b.

f) [4 pts] Contrast Haskell's notion of inheritance of implementation (i.e., default implementations of type class operations) with an OO language's notion, pointing out where one approach has additional expressiveness over the other.

Haskell allows methods to be put at the top of the class hierarchy (when the method is initially declared) and at the leaves (when an instance is specified). OO languages also allow methods to be put in any intermediate class in the hierarchy, to be inherited by a subset of the leaf classes.

6) Consider the following concrete data types and associated instance variables: `Employee` (`name`, `salary`), `Professor` (`name`, `salary`, `school`, `class_list`), `Teacher` (`name`, `salary`, `school`, `class_list`), `Student` (`name`, `school`, `class_list`), and `CollegeStudent` (`name`, `school`, `class_list`, `major`). (The professor & teacher class lists refer to the classes being taught, while those for the students refer to the classes being taken.)

a) [6 pts] Organize these ADTs into a reasonable inheritance hierarchy, implemented using Cecil objects and fields. Feel free to introduce appropriate abstract objects to maximize sharing of fields and other (implied) code. Ensure that your classification makes intuitive sense. You do not need to include any type declarations (use the dynamically typed subset of Cecil).

Option 1: factor out `school` and `class_list` into a common abstract object `PersonAtSchool`:

```
abstract object Person;
  field name(@Person);

template object Employee isa Person;
  field salary(@Employee);

abstract object PersonAtSchool isa Person;
  field school(@PersonAtSchool);
  field class_list(@PersonAtSchool);

template object Teacher isa Employee, PersonAtSchool;

template object Professor isa Teacher;

template object Student isa PersonAtSchool;

template object CollegeStudent isa Student;
  field major(@CollegeStudent);
```

Option 2: make separate `school` and `class_list` fields for the two senses:

```
template object Teacher isa Employee;
  field school(@Teacher);
  field class_list(@Teacher);

template object Student isa Person;
  field school(@Student);
  field class_list(@Student);
```

b) [3 pts] Think about adding the `TeachingAssistant` object using multiple inheritance to combine the properties of a `Teacher` and a `Student`. What problem arises due to multiple inheritance?

In option 1, the problem is that a TA would get a single `class_list` field, even though a TA is taking one set of classes and teaching another.

In option 2, there would be two `class_list` fields and two `school` fields, but accessing `class_list` or `school` would become ambiguous.

7) The Pizza language supports both object-oriented features and "algebraic types," which are in essence ML `datatype` declarations and functions defined by pattern matching over the `datatype`'s cases.

a) [3 pts] Why does Pizza support both, according to the authors?

Classes make it easy to add subclasses, if the operations are relatively fixed, while algebraic types achieve the reverse.

b) [4 pts] How would extending Pizza to support multimethods (or even just the ability to add new methods to existing classes from the outside, without modifying the classes) eliminate some of the need for algebraic types?

Adding new methods would make it easy to extend a class with new operations, eliminating the advantage of algebraic types over classes.

c) [3 pts] What additional challenges would be introduced by adding multimethods to Pizza?

There are modular typechecking and encapsulation challenges with multimethods (and even the restricted case of adding new operations to existing classes).

8) a) [4 pts] Argue why dynamic dispatching is the key to the power of object-oriented programming.

DD is the key since it allows subclasses/children to be used wherever superclasses/parents are expected, and still enable the subclasses/children to have their own implementations of their interface (by selecting the method defined for the subclass/child, not the one of the superclass/parent). Also allows code written for one class/interface to be reused for any other class that supports the same (or augmented) interface.

b) [4 pts] Argue why abstract classes and factoring are the keys to the power of object-oriented programming.

Abstract classes and factoring take DD one step further, sharing code across multiple similar classes, making new code easy to add in one place and propagating automatically to all descendant classes. DD is required to make shared code able to access descendant-specific operations ("sends to self").

9) a) [4 pts] Argue how Cecil's model of methods follows the design principle of uniformity, merging separate, similar constructs together into one general mechanism.

Cecil's methods unify undispatched procedures, singly-dispatched methods, functions overloaded on arguments, some aspects of pattern matching, constructors, control structures, and built-in operations. Dispatching is the one mechanism applied uniformly to select the appropriate case.

b) [4 pts] Cecil's named methods are not themselves first-class; only anonymous closure objects are first-class. But closure objects cannot have dispatched, specialized formals; only methods can. Is this glitch in uniformity a limitation in practice? If so, what kinds of programs cannot be written? If not, how can the glitch be worked around to avoid losing expressiveness?

The glitch isn't a real limitation, just an annoyance. The effect of a first-class named multimethod (or collection of multimethods) can be simulated by a closure whose body just sends the corresponding message, e.g.

```
&(a,b,c){ foo(a,b,c) }
```

is a first-class version of the `foo` method.

c) [Up to 5 pts extra credit] Propose a revised design for Cecil that would resolve this glitch.

Here's one possible idea:

Make generic functions (the set of methods of a given name) first-class. E.g., `foo` above names an object, which is a collection of methods. A method declaration adds a case to the named generic function, and the method "name" is an expression, not just a constant identifier. Similarly, a message send expression would use a full expression as its name, not just an identifier. The anonymous closure expression would be generalized to be an anonymous generic function expression, which can have several cases and which can have dispatching done on any of the cases. `eval` goes away; closures are just applied directly to arguments. This model is closer to Scheme's treatment of functions.

E.g., here's an anonymous generic function, assigned to a local named `equal`, which is then assigned to a different local (it's just a regular value), and then eventually invoked.

```
let equal := &(x1,x2){
     method(@Point, @Point) { ... compare points .. }
     method(@Int, @Int) { ... compare ints ... }
     ... };

let eq := equal;

eq(3, 4)
```

10) [3pts] What feature is offered by constraint logic programming languages that is not found in imperative, functional, or object-oriented languages?

The ability to run “functions” forwards, backwards, or as generators.

The ability to produce multiple possible solutions.

The ability to produce unbounded sets of answers described by constraints.

11) Consider the definitions of `1` and `succ` in the untyped lambda calculus:

```
1 ≡ λS.λZ.S Z
  ≡ λS.(λZ.(S Z))

succ ≡ λN.λS.λZ.S (N S Z)
     ≡ λN.(λS.(λZ.(S ((N S) Z))))
```

a) [4 pts] Evaluate the lambda term `(succ 1)`, showing each β-reduction step and each possible β-redex considered at each reduction.

```
succ 1 =

(λN.λS.λZ.S (N S Z))@(λS.λZ.S Z) ⇒

λS.λZ.S ((λS.λZ.S Z)@S Z) ⇒

λS.λZ.S ((λZ.S Z)@Z) ⇒

λS.λZ.S (S Z)
```

b) [6 pts] Consider the following typing of these terms in the simply typed lambda calculus:

```
nat ≡ (*→*)→*→*
    ≡ (*→*)→(*→*)

1 ≡ λS:*→*.λZ:*.S Z
  ≡ λS:*→*.(λZ:*.(S Z))

succ ≡ λN:nat.λS:*→*.λZ:*.S (N S Z)
     ≡ λN:nat.(λS:*→*.(λZ:*.(S ((N S) Z))))
```

Using the type checking rules for the simply typed lambda calculus used in class, draw a proof tree deducing the type of the `succ` term, in the empty typing environment.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{S}{:}{*}\to{*} \qquad \dfrac{\dfrac{\Gamma \vdash \mathtt{N}{:}\mathtt{nat} \qquad \Gamma \vdash \mathtt{S}{:}{*}\to{*}}{\Gamma \vdash (\mathtt{N\ S}){:}{*}\to{*}} \qquad \Gamma \vdash \mathtt{Z}{:}{*}}{\Gamma \vdash (\mathtt{N\ S\ Z}){:}{*}}}{\Gamma \equiv \mathtt{N{:}nat},\ \mathtt{S}{:}{*}\to{*},\ \mathtt{Z}{:}{*} \vdash (\mathtt{S\ (N\ S\ Z)}){:}{*}}}{\mathtt{N{:}nat},\ \mathtt{S}{:}{*}\to{*} \vdash (\lambda \mathtt{Z}{:}{*}.\mathtt{S\ (N\ S\ Z)}){:}{*}\to{*}}}{\mathtt{N{:}nat} \vdash (\lambda \mathtt{S}{:}{*}\to{*}.\lambda \mathtt{Z}{:}{*}.\mathtt{S\ (N\ S\ Z)}){:}\mathtt{nat}}}{\varnothing \vdash (\lambda \mathtt{N{:}nat}.\lambda \mathtt{S}{:}{*}\to{*}.\lambda \mathtt{Z}{:}{*}.\mathtt{S\ (N\ S\ Z)}){:}\mathtt{nat} \to \mathtt{nat}}
$$

c) [2 pts] Using this typing, does the term `(succ 1)` typecheck? If so, give the result type; if not, explain why not.

Yes, yielding `nat`.

d) [3 pts] Using this typing, does the term `(succ (succ 1))` typecheck? If so, give the result type; if not, explain why not.

Yes, yielding `nat`.

[The definition of add ≡ (λM.λN.M succ N) *doesn't* typecheck in the simply typed lambda calculus, though!]

12) Consider the following PCF expression:

```
P ≡ λb:bool.
      let g = (rec f:nat→nat = λn:nat. f n) in
        if b then g 1 else 0
```

a) [2 pts] What is the type of this expression? (You need not show a proof, just the final type.)

`bool→nat`

b) [3 pts] For what set of V ∈ Val is it possible to prove $\varnothing \vdash$ `(P false)`➥`V` ?

{0}

c) [3 pts] For what set of V ∈ Val is it possible to prove $\varnothing \vdash$ `(P true)`➥`V` ?

None: this computation runs forever

d) [4 pts] What is the denotation (i.e., meaning) of P in the empty environment, i.e., what is $\mathcal{E}[\![P]\!]\,\varnothing$? Give your answer in simplest possible terms.

λv. if v = true then ⊥ else 0

13) Consider extending PCF with a record type and value. The syntax could be extended as follows:

$$E ::= \ldots \mid \{\ I_1 = E_1,\ \ldots,\ I_n = E_n\ \} \mid E.I$$

$$\tau ::= \ldots \mid \{\ I_1 : \tau_1,\ \ldots,\ I_n : \tau_n\ \}$$

For example, the expression $\texttt{P}_1 \equiv \texttt{\{X=3+4,Y=true\}}$ builds a record value of type $\texttt{\{X:int,Y:bool\}}$, and the expression $\texttt{P}_2 \equiv \texttt{(\{X=3+4,Y=true\}).X}$ extracts the X member of the record, evaluating to 7.

a) [8 pts] Specify the static semantics (i.e., typing rules) of the two new kinds of expressions. Show the proof tree deducing the type of the $\texttt{P}_2$ expression above.

$$\frac{\Gamma \vdash \texttt{E}_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash \texttt{E}_n : \tau_n}{\Gamma \vdash (\{\texttt{I}_1 = \texttt{E}_1, \ldots, \texttt{I}_n = \texttt{E}_n\}) : \{\texttt{I}_1 : \tau_1, \ldots, \texttt{I}_n : \tau_n\}}$$

$$\frac{\Gamma \vdash \texttt{E} : \{\texttt{I}_1 : \tau_1, \ldots, \texttt{I}_n : \tau_n\}}{\Gamma \vdash (\texttt{E}.\texttt{I}_i) : \tau_i} \qquad \text{where } i \in 1..n$$

$$\cfrac{\cfrac{\cfrac{\emptyset \vdash 3 : \texttt{nat} \qquad \emptyset \vdash 4 : \texttt{nat}}{\emptyset \vdash (3+4) : \texttt{nat}} \qquad \emptyset \vdash \texttt{true} : \texttt{bool}}{\emptyset \vdash (\{\texttt{X=3+4,Y=true}\}) : \{\texttt{X:nat,Y:bool}\}}}{\emptyset \vdash (\{\texttt{X=3+4,Y=true}\}.\texttt{X}) : \texttt{nat}}$$

b) [10 pts] Specify the dynamic semantics of the two new kinds of expressions, in either operational or denotational style. If using operational semantics, show the proof tree deducing the result of evaluating of the $\texttt{P}_2$ expression. If using denotational semantics, give the meaning of the $\texttt{P}_2$ expression **before simplifying**. Hint: the semantic representation of a record value ($\texttt{Val}_{\{\ldots\}}$) can just be Env, which is already a partial mapping from identifiers to values, represented as a set of Id:Val pairs.

Operational semantics:

$$\frac{\rho \vdash \texttt{E}_1 \Rightarrow \texttt{V}_1 \qquad \ldots \qquad \rho \vdash \texttt{E}_n \Rightarrow \texttt{V}_n}{\rho \vdash (\{\texttt{I}_1 = \texttt{E}_1, \ldots, \texttt{I}_n = \texttt{E}_n\}) \Rightarrow \{\texttt{I}_i : \texttt{V}_i \mid i \in 1..n\}}$$

$$\frac{\rho \vdash \texttt{E} \Rightarrow \{\texttt{I}_j : \texttt{V}_j \mid j \in 1..n\}}{\rho \vdash (\texttt{E}.\texttt{I}_i) \Rightarrow \texttt{V}_i} \qquad \text{where } i \in 1..n$$

$$\cfrac{\cfrac{\cfrac{\emptyset \vdash 3 \Rightarrow 3 \qquad \emptyset \vdash 4 \Rightarrow 4}{\emptyset \vdash (3+4) \Rightarrow 7} \qquad \emptyset \vdash \texttt{true} \Rightarrow \text{true}}{\emptyset \vdash (\{\texttt{X=3+4,Y=true}\}) \Rightarrow \{\texttt{X}:7, \texttt{Y}:\text{true}\}}}{\emptyset \vdash (\{\texttt{X=3+4,Y=true}\}.\texttt{X}) \Rightarrow 7}$$

Denotational semantics:

$$\mathcal{E} [\![\{\texttt{I}_1 = \texttt{E}_1, \ldots, \texttt{I}_n = \texttt{E}_n\}]\!] \rho = \{ \texttt{I}_i : \mathcal{E} [\![\texttt{E}_i]\!] \rho \mid i \in 1..n \}$$

$$\mathcal{E} [\![\texttt{E}.\texttt{I}]\!] \rho = (\mathcal{E} [\![\texttt{E}]\!] \rho)(\texttt{I})$$

$$\mathcal{E} [\![\{\texttt{X=3+4,Y=true}\}.\texttt{X}]\!] \emptyset = (\{\texttt{X}:3+4, \texttt{Y}:\text{true}\})(\texttt{X})$$

14) [6 pts] Imagine adding side-effects to the purely functional version of PCF by adding a `set! I E` construct, instead of adding a `ref` data type. Like its Scheme namesake, `set! I E` side-effects `I` to refer to the result of evaluating `E`. A simple approach to defining this language's semantics would be to drop the store idea and just use environments directly. Now each evaluation judgment would have the form $\rho \vdash E \hookrightarrow (V, \rho')$, i.e., evaluation of `E` in an environment ρ produces both a value `V` and an updated environment ρ' reflecting any side-effects that happened during evaluation of `E`. The rule for evaluating the new `set! I E` construct would be the following:

$$\frac{\rho \vdash E \hookrightarrow (V, \rho')}{\rho \vdash (\texttt{set! I E}) \hookrightarrow (V, \rho'')} \qquad \text{where } \rho'' = \rho' \uplus I{:}V$$

Why is this approach doomed to fail? Hint: consider the semantics of lambda abstractions and their application.

It will fail because lambda abstractions capture the environment when they're defined, including the environment's current bindings. When a lambda is applied, the body is evaluated in the context of the captured environment, which in effect rolls back the effect of any `set!`'s on identifiers in scope of the lambda. Moreover, what's the environment returned along with the result of evaluating the lambda's body? If the captured environment, any variables declared after the lambda will be lost after the call returns. If the caller's environment, then any updates to free variables in the lambda's body will be lost. The following program illustrates all the broken cases.

```
let x = 1 in
   let P = λv.(set! x (x+v); x) in
      let y = 2 in
         set! x 2; P(x)+y
```