Spring, 1996 Name: ______________________

Com S 541 — Programming Languages 1

Test on the Abstraction Principle

Special Directions for this Test

This test has 10 questions and pages numbered 1 through 6.
You are allowed to use two pages of notes. These notes are to be handed in at the end of the test.
Before you begin, please take a moment to look over the entire test so that you can budget your time.

1. (5 points) Fill in the blank in the following.

   What kind thing that we studied in chapter 2 of Schmidt's book is the best description of a Smalltalk class? ______________

2. (10 points) Circle the letter of the answer that best describes the abstraction principle.

   (a) Programs should be written at as abstract a level as possible, because this eases the task of understanding the program. Hence, a programming language should have good abstraction mechanisms.

   (b) Phrases of any semantically meaningful syntactic class should be able to be named in a language.

   (c) A language should provide abstract data type facilities, which allow for information hiding. This allows a program to work at a higher level of abstraction, by using data that is a better match to the problem.

   (d) Programs should be written using the techniques of abstract algebra, especially the techniques of homomorphic functions, which abstractly map one algebraic structure to another. Thus programming languages should provide support for homomorphisms, like the map function.

   (e) Visual programming tools, like visual BASIC, are better than non-visual tools, because they work at a more abstract level.

   (f) Every published paper should start with an abstract.

3. (20 points) Fill in the empty areas of the following table with the correct abstraction names and best evaluation semantics (for an imperative programming language). For evaluation mechanisms, you can write "either" if the particular evaluation mechanism doesn't matter. Otherwise, write the name of the particular mechanism (you're supposed to know what these are).

syntactic class	abstraction name	best evaluation semantics
location	alias	either
numeral		
expression		
command		
declaration		
type structure		

4. (10 points) In Unix, the Bourne shell (a command language) allows one to fork (i.e., start) an independent process with the following abstract syntax.

$C \in$ Command
$C ::= C$ `&`

For example, one can write

```
emacs &
```

which starts an independent process running emacs. Briefly describe (in 1 or 2 sentences) what would have to be true about the Bourne shell for it to satisfy the abstraction principle?

5. (10 points) Briefly answer (in 1 or 2 sentences) the following questions. (i) Would a lazy evaluation semantics work for Haskell modules? (ii) Why or why not?

6. (10 points) Briefly answer (in 1 or 2 sentences) the following questions. (i) What is a module in C? (ii) What is the syntax for the invocation (import) of a module in C?

7. (10 points) Briefly answer (in a few sentences) the following question. What is the difference between a module and a record?

8. (10 points) Briefly answer (in a few sentences) the following questions. (i) What is the purpose of a δ attribute? (I.e., what is it used for?) (ii) What is the purpose of a $\delta class$ attribute?

9. Suppose the abstract syntax for type structures is modified to be the following.

   T $\in$ Type-structure   D $\in$ Declaration
   X $\in$ Identifier-expr   E $\in$ Expression
   N $\in$ Numeral   I $\in$ Identifier
   T ::= **newint** (E) | **record** D **end** | X
   D ::= D_1 , D_2 | D_1 ; D_2 | **fun** I = E
   | **const** I = N | **proc** I = C | **var** I : T
   | **class** I = T | **module** I = { D } | **import** X

   The idea is that, for example, in the following declaration the initial value of `x` is 541, and the initial value of `y` is 342.

   var `x` : **newint**`(541);`
   var `y` : **newint**`(342)`

   Briefly answer the following questions.

   (a) (10 points) Does this syntax prevent uninitialized variable errors? If so, how; if not, why not?

   (b) (15 points) Give the type rules for the syntactic class Type-structure.

10. This problem is about adding variant records (i.e., tagged discriminated unions) to the language of Schmidt's chapter 2. We will model variant records on the Haskell `data` declarations, which are a kind of immutable variant record; however, we will add mutation. Consider the following abstract syntax:

T ∈ Type-structure   F ∈ Variant-field
I ∈ Identifier
T ::= **variant** F **end**
F ::= **const** I = N | **var** I : T | F_1 , F_2

For example, consider the following declaration.

```
module vehicle = { var mpg : newint };
class truck = record import vehicle;
                     var payload : newint
              end;
class car = record import vehicle;
                   var passengers : newint;
                   var license : newint
            end;
var v : variant var c : car,
                var t : truck
        end
```

In the above declaration, `v` is a variant record, which can be either a car or a truck.

(a) (10 points) Describe any new type attributes, and give a type rule for (i) the variant-field (F) syntax, and (ii) the type structure (T) syntax that describes a variant record.

(This question continues on the next page...)

(b) (20 points) Give an abstract syntax and a type rule for an expression (or expressions) that can extract the relevant information from a variant record (i.e., what the tag is, and the relevant value is). If you need new type attributes, also describe those. Ideally, your syntax should make it impossible to try to extract a value with the wrong tag. For example, if `v` happens to hold a truck, then one should not be able to try to extract the license number. (Hint: there is some expression in Haskell that does this kind of thing. Also, you might want to use more than one thing to the left of the $\vdash$ in some of your rules.)