CS 538

Homework #1

Due: Wednesday, February 25, 2004

(Not accepted after Wednesday, March 3, 2004)

1. Most procedural programming languages, including C, C++ and Java, group state-
ments into a block. Within a block statements are executed sequentially. That is, a
sequence of statements {S_1 ; S_2 ; ...; S_n} specifies that statement S_1 be executed, then
S_2, and finally S_n.

Some programming language designers have suggested an alternate to a sequentially
executed block of statements—a concurrent block. A sequence of statements {[[S_1 ; S_2;
...; S_n]](delimited by [ and ]]} specifies the execution statements S_1 through S_n in any
order. Thus S_2 might be completely executed first, then S_4, then S_1, etc. Eventually all
n statements are executed, but the exact order is unspecified by the programmer.

(a) What advantages (if any) are there in using a concurrent block rather than an ordi-
nary sequential block in a program?

(b) Is a concurrent block always equivalent to a sequential block? That is, for any set of
statements S_1 to S_n (in C or Java—your choice), does {S_1 ; S_2; ...; S_n} and {[[S_1 ; S_2;
...; S_n]] always compute the same result?
If so, explain carefully why. If not, give a simple example that demonstrates the
inequivalence.

(c) A concurrent block seems well suited for parallel architectures—individual state-
ments can be executed in parallel on different processors or threads. If we know the
order of execution of individual statements in a concurrent block is immaterial (all
execution orders produce the same result), does this guarantee that if individual
statements are executed in parallel, we will get the same result as if the statements
had been executed sequentially?
If so, explain carefully why. If not, give a simple example that demonstrates the
inequivalence.

2. In the early days of programming language design, procedure calls were explained
using a macro-expansion model. That is, the effect of a procedure call P(e_1,e_2,...,e_n)
was defined to be equivalent to expanding the body of P at the point of call, with all
occurrences of P’s first formal parameter replaced by e_1, P’s second formal parameter
replaced with e_2, etc. Any calls that appear within P were also modeled using macro
expansion.

Note that it isn’t necessary to implement calls this way—we can simply use macro-
extension as a way to explain the effect of a procedure call.
(a) A procedure body often contains references to identifiers that are not defined locally. Such identifiers are said to be **free** (because they are not bound to identifiers defined within the procedure body. For example, in the following simple C procedure, identifier `c` is free:

```c
void p(int a) {
  int b;
  return a+b+c;
}
```

Recall that a free variable in a procedure can be bound either statically or dynamically. Which binding method should be used if the macro-expansion model of calls is followed? Explain why.

(b) A number of parameter passing mechanisms, including call by value, call by reference and call by name have been used in programming languages. Which of these (if any) correspond to the macro-expansion model of calls? For each parameter passing mode you should explain carefully why it corresponds to the macro-expansion model, or give a simple example illustrating why it fails to match that model.

3. We have seen that lazy evaluation is able to suspend a computation of a value until it is actually needed. Eager evaluation initiates a computation as soon as its input values are available, even before we know whether the value will actually be needed and used.

Consider a new form of evaluation—the future. Using a future, a computation of a value is initiated as soon as we reach it (just like ordinary evaluation). However, a reference to the computation (called a future) is returned immediately, even before the computation is completed. This future can be assigned to a variable or stored in a data structure. When the computation eventually does complete, the future is automatically transformed into the value that was computed.

Form the user’s point of view, one of two things can happen. If the value of the future is needed before the computation it represents is completed, the user is suspended and restarted after the computation of the value completes. If the user uses the value of the future after the computation of it is completed, the user sees the fully-computed value, and may use it in any way.

(a) Under what circumstances is use of a future preferable to using a suspension (i.e., lazy evaluation)?

(b) Under what circumstances is use of a future preferable to using speculative (eager) evaluation?

(c) What are the drawbacks to using futures? That is, why shouldn’t futures be used for computation of all values?
4. Many programming languages allow creation of a new name for an existing data type using some form of type declaration. In C and C++, `typedef` can be used:

```c
typedef int integer;
```
defines `integer` to be a synonym for `int`.

Assume that we wish to take an existing data type, like `int` or `float`, and create a brand new type that inherits all of the literals, operators and library subroutines of the old type, but which is type-inequivalent to it. For example, we might use

```c
newtype meters is created from float;
```
to create a type `meters` that is different from `float`, but which has literals like `2.1` or `10e-5`, operators like `+` or `*`, and library routines like `sqrt` or `toString`.

Give a set of type rules that can be used to determine if a program that uses both an original type (like `float`) and a newly created type (like `meters`) is type-correct. Illustrate your rules on the following code fragment:

```c
meters m; float f;
m = m + 1.0; // Correct
f = m - 1.0; // Error; float & meters are different types
```

5. Structural equivalence in languages that contain structs (like C and C++) is defined as follows.

Struct `S1` is structurally equivalent to struct `S2` if `S1` and `S2` contain the same number of fields and corresponding fields (in order of declaration) in `S1` and `S2` are structurally equivalent. (The names of corresponding fields within the two structs need not be the same.)

Two pointers are structurally equivalent if the types they point to are structurally equivalent. That is, `S1*` is structurally equivalent to `S2*` if and only if `S1` is structurally equivalent to `S2`. Two arrays are structurally equivalent if they have the same size and their component types are structurally equivalent. Each scalar type is structurally equivalent only to itself.

(a) Assume a C-like language in which structs may contain fields declared to be scalars (`int`, `float`, etc.), arrays, pointers and structs. Give an algorithm that decides if two structs, `S1` and `S2`, are structurally equivalent.

(b) Most languages, including C and C++, state that the order in which fields are declared is unimportant. That is, rearranging field declarations in a struct has no effect other than possibly changing the size of the struct.

Given this observation, it might make sense to change the rule for structural equivalence of structs so that two structs are structurally equivalent if they contain the same number of fields and it is possible to reorder the fields of one struct so that corresponding fields are structurally equivalent after reordering. That is, the order of fields in a struct no longer matters (nor does the name of fields). Thus the following two structs are now considered structurally equivalent:
struct S1{
    int f1;
    float f2;
}

struct S2{
    float g1;
    int g2;
}

Update your algorithm of part (a) to implement this revised definition of structural equivalence. Illustrate your algorithm on the following set of structs:

struct S1{
    S1* f1;
    S2* f2;
    S3* f3;
}

struct S2{
    S4* g1;
    S1* g2;
    S2* g3;
}

struct S3{
    S4* h1;
}

struct S4{
    S3* j1;
}