CS 538

Homework #1

Due: Monday, March 3, 2008

(Not accepted after Monday, March 10, 2008)

1. Many programming languages allow large programs to be subdivided into **units** (often calls **packages** or **modules**). Each unit is given a name and may be separately compiled. A unit contains a collection of definitions, including types, constants, variables, classes, and subprograms.

   A member of a unit may be accessed directly using the unit name and the member's name (for example `U.a` means identifier `a` in unit `U`). Java uses this notation to access members of a package.

   If we intend to use a number of names defined in one or more units (or to use a given name several times), it is useful to **import** the contents of a unit into the current scope. Assume that

   `import` u_1, u_2, ... , u_n;

   causes all the declarations that appear in u_1 through u_n to be imported into the current scope. After an `import` statement imported names may be used without qualification (just like locally declared identifiers).

   Two potential ambiguities arise:
   (i) What if the same identifier is declared in more than one imported unit?
   (ii) What if an imported identifier clashes with an identifier already visible using normal scoping rules?

   For each of these two potential ambiguities, propose and analyze possible resolutions to the ambiguities. Which resolutions do you recommend, and why?

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-expansion 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:

```
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-result, 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. Many main-stream programming languages, including Java and C#, have begun to add constructs that support concurrent (parallel) execution. Java's thread mechanism is a good example of this.

   Another proposed concurrent execution mechanism is **method level parallelism**. Ordinary method calls are *synchronous*. This means that the caller of a method suspends execution while the called method executes, and the caller then resumes after the called method returns.

(a) Methods that are procedures (i.e., return `void`) are called for side-effects since they return no explicit value to the caller. This suggests it may be possible for the caller of a procedure to resume execution **immediately**, and run while the called procedure is executing. This execution mechanism is what we mean by method-level parallelism; the calling method and the called method may execute in parallel.

   Under what circumstances is it safe to use method-level parallelism? That is, under what circumstances will concurrent execution of the two methods give the same result as ordinary synchronous execution?

(b) Of course many method calls are to functions not procedures. For these calls, the caller expects to receive a return value. Is method-level parallelism of any value for function calls? That is, does it ever make sense to resume execution of the caller even though the method just called has yet to return its result value?

4. Recall that lazy evaluation is interesting and useful because it can sometimes delay or entirely avoid a slow or expensive computation. Some function programming languages, most notably **Haskell**, are entirely lazy. In a lazy language, a value definition like

   ```
   a = b + c + d;
   ```

   does not mean "compute the sum of `b`, `c` and `d`, and assign it to `a`." Rather it means "here is rule for computing `a` if you ever need the value of `a`."

   Computations are all deferred until an unavoidable statement like `print` or `return` is reached. Then deferred evaluations are forced to compute the demanded value. Thus

   ```
   print(a);
   ```

   would force us to consult the definition of `a` and use it to compute `a`.

   Lazy evaluation has not appeared in modern procedural languages like Java or C#. This suggests there may be drawbacks in suspending computations in these sorts of languages. What problems do you see in adding lazy evaluation to Java or C++ (your choice)? Are there ever any computations that can and should be implemented lazily? How do you recognize such computations?

5. Structural equivalence in languages that contain structs (like C, 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*` (a pointer to type `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, 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 S{
   int f1;
   float f2;
}

struct T{
   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. Is `S1` structurally equivalent to `S2`? Why?

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

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

struct S3{
   S4* h1;
   int h2;
}

struct S4{
   S3* j1;
   int j2;
}
```