DUE: Monday 20 February 2006

READING — Due 13 February

- Watt: Section 3.3; Section 5.2
- Scott, 1st Ed: Chapter 11, Sections 11.1-11.2
  Scott, 2nd Ed: Chapter 10, Sections 10.1-10.8
- Lloyd Allison, *Lambda Calculus*. Web document at
  `http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/`
  See link on CSc 520 course homepage.
- Read Robert Harper, "Introduction to Standard ML". It can be found on the Web, or there is a `pdf` version on the CSc520 web page at `www.cs.arizona.edu/classes/cs520/spring06/SML.pdf`.

PROBLEMS

1. Static vs Dynamic Binding

Scott, 1st Ed: Exercise 3.11.
Scott, 2nd Ed: Exercise 3.13.

2. Association Lists

For the program in the preceding question (1), assume dynamic scope and an association-list (a-list) representation of the referencing environment. Trace the changes made to the a-list over the course of the program's execution, by drawing figures whenever there is a change to the a-list, and key these changes to the line in the code where they occur.

3. Deep vs Shallow Binding

Scott, 1st Ed: Exercise 3.17.
Scott, 2nd Ed: Exercise 3.16.

4. Normal-order Evaluation

Scott, 1st Ed: Exercise 11.4.
Scott, 2nd Ed: Exercise 10.4.

5. Lambda Calculus

Define **S** = λx . λy . λz . xz(yz) and **I** = λx . x.

(a) Derive a normal form for (λy . yyy)((λab . a)**I**(**SS**))

(b) Watt text, p. 144, Exercise 5.1

6. Type Equivalence in C and C++

Languages differ in what types they regard as equivalent, for purposes of enforcing strong typing constraints at compile-time. A language that views two datatypes the same if they are built in exactly the same way using the same type constructors from the same simple types is said to use *structural* equivalence for types. A much stricter (i.e., *less* inclusive) type equivalence is *name* equivalence: two named types are equivalent only if they have the *same name* (ADA uses strict name equivalence). Consider the following example

```
type ar1 is array(1..10) of INTEGER;
type ar2 is new ar1;
type age is new INTEGER;
```

Here `ar1` and `ar2` are structurally equivalent but *not* name equivalent. Similarly `age` and `integer` are structurally but not name equivalent.

An important type equivalence algorithm that falls between name and structural equivalence is *declaration* equivalence, used by Pascal and Modula2. In this algorithm, type names that lead back to the same original structure declaration via a series of re-declarations are considered to be equivalent types. In the above example, `ar1` and `ar2` are declaration equivalent, as are `age` and `INTEGER`. However, in

```
type  ar3 is array(1..10) of INTEGER;
      ar4 is array(1..10) of INTEGER;
```

the two types are not declaration equivalent (though they are structurally equivalent). See Scott, pp. 330-334 for further discussion.

The language C uses structural type equivalence for arrays and pointers, but declaration equivalence for structs and unions.

(a) Cite documentation that supports this statement about C from a believable source.

(b) Verify that these equivalences are enforced by a C compiler near you. Your examples should clearly indicate that the compiler allows structurally equivalent arrays/pointers to be treated as type-compatible, while disallowing this for structs/unions. It should also clearly indicate that structs/unions use declaration equivalence, not the more stringent name equivalence.

(c) Why did the language designers make this choice? In your answer, indicate the consequences of making the alternative choice: using full structural equivalence everywhere.

(d) Does this mixing of type-equivalence notions cause any difficulties for a programmer? Explain.

(e) What kind of type equivalence does C++ use? Describe it using examples, and cite your reference.

7. Type Equivalence in ML

What form of type equivalence is used in ML? Show an ML session that provides evidence for your conclusion.

8. Array Updating

Assume all variables, expressions and arrays in a language *L* are of type integer. When necessary, a boolean value of **false** will be represented by integer zero and **true** by any non-zero integer. Consider the fragment

```
A[I] := D;
A[J] := C;
E  := A[I];
```

All array references may be assumed within bounds.

(a) Describe the value of *E* in terms of the values *I, J, C* and *D* without mentioning the array *A* or using any array references. Use as concise an expression for the value as possible. (The idea is to show in a single expression exactly how *E* depends on *I, J, C* and *D*.)

(b) Suppose *L* does not support conditional expressions. How can we simulate the conditional expression assignment *X* := **if** *expr1 = expr2* **then** *expr3* **else** *expr4* without the use of statements which alter the flow of control? Assume that all the *expr*'s return values. Assume that expression evaluation has no side-effects.

(c) Same as (b) but now generalize to allow expression evaluation to have side-effects.

(d) Your simulation in (c) is (probably) unsatisfactory if not all the *expr*'s are well-defined (eventually return values). Explain why, if this criticism applies, or else explain how you avoided the difficulty!