EECE 437 — Software Architecture and Design Fundamentals
Assignment 1

January 31, 2017

Data types are the basic data abstraction concepts of any programming language. A data type is characterised by the following.

- The information it can theoretically represent. For example, the Boolean type abstracts all named entities that may range over the set of Boolean values $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$.
- The operations it supports. For example, Boolean variables and constants support logical conjunction (AND), disjunction (OR) and negation (NOT)
  - Some of these operations are closed within the type; i.e. they produce a value of the same type.
  - Some of these operations produce values of other types; e.g. the operation $x \leq y$ operates over numbers represented by x and y and produce a value of type Boolean.
- Composition and connections of types can form other types.
- The machine representation that limits the capacity of the type. For example, an `int` type is limited to 32 bits under some programming languages and computer architectures.

Consider the following.

- Primitive types of Boolean, integer, and double.
- Unary operations that take one operand such as NOT, MINUS, CEILING and FLOOR.
- Operations that take two operands such as AND, OR, XOR, IMPLIES, PLUS, MINUS, MULTIPLY, DIVIDE, LESSTHAN, MORETHAN, and EQUAL.
- A pair structure that allows building a custom type out of two other types.

  - operators `.first` and `.next` that allows access to the first element and the second element in the pair.

- Array types that allow building sequence types based on other types.
  - Operator `.size` that allows to access the size of the array
  - Operator `[i]` that allows access to element *i* of the array

Problem 1 – due Monday February 6, 2017

- Show in one or more diagram(s) the primitive types, examples of instantiations, and interconnections between the instances across the types using the different operators.
- Build a class hierarchy with the needed constructors in a programming language of your preference where each type and each "operation kind" is represented as a class.
- Test your system for the following declaration statements (or their equivalents in your designed class hierarchy). The prefix `TA` stands for "type abstraction". (More test cases will be provided on moodle).

```
TAInt x;
TADouble d;
TACeiling y(d); // denotes y is the ceiling of d
TABool b;
TALessThan t1 (x,y); // denotes x < y
TAAnd t2 (b, t1); // denotes b and t1
```

- Make sure the class hierarchy and the constructors you build produce static compilation errors when the operations are used in error. For example, the following declaration list should produce errors.

```
TAInt x;
TADouble d;
TALessThan t1 (x,d); // x and d are not of the same type and
   //thus TALessThan should not accept to perform the construction
```

- *Hint:* do not start *coding* before you *design* the class hierarchy in full. All questions and design ideas are welcome on moodle and are worth class grades.
- *Hint:* do not start *coding* before you *read* problems 2 and 3.

Problem 2, Due Wednesday February 8, 2017

- Augment all your primitive, array, and pair types with a static name property. Then provide a `list` functionality in all your classes.

  - The `list` functionality will print the static name if one exists,
  - Otherwise it will print an open parenthesis, print the name of the operation, print a space, call list of each operand and print a space, then print a close parenthesis.

- Design and implement a `set` functionality for each of the primitive types. The `set` functionality takes a corresponding value and saves it as the state of the instance.
- Design and implement an `evaluate` functionality for all the classes in your class hierarchy.
- The following should work as expected.

```
TAInt x("x");
TADouble d("d");
TACeiling y(d); // denotes y is the ceiling of d
TABool b("b");
TALessThan t1 (x,y); // denotes x < y
TAAnd t2 (b, t1); // denotes b and t1
t2.list();// should print: (& b (< x (ceiling d ) ) )
x.set(5);
d.set(2.3);
b.set(true);
t1.evaluate();
t1.printState();// should print: false
d.set(5.3);
t1.evaluate();
t1.printState();// should print: true
```

- Same hints apply.

Problem 3, Due Monday February 13, 2017

If you reading this before answering and implementing Problems 2 and 3, then you are fine. Otherwise, the problem now begins. Your design at the beginning should have considered the following.

- Consider adding one (or more) class(es) to your class hierarchy that allow(s) instantiations of pairs of objects. A pair takes two objects of any type allowed in the system including primitive types, pairs, and arrays.
  - That is pairs are allowed to be nested.
  - A pair supports a `.first` and a `.next` operator.
- Consider adding one (or more) class(es) to your class hierarchy that allow(s) instantiations of arrays of objects of the same type. That includes arrays of primitive types, arrays of pairs, and arrays of arrays. An array supports a `.size` and an access (`[i]`) operator.

```
TAConstant N(16); // surprise
TAConstant one(1);
TAInt x("x");
TADouble d("d");
TAPair p(x,d);

//The following line is tricky:
// what type should the second argument be declared as?
TAArray a("a", x.type(), N); //instantiate an array of integer with capacity 16
TAInt i("i");
TAPlus exp(i,one);
TAArrayAccess ai(a, exp);
TAPair p2(ai, x);
i.set(1);
ai.set(7); // The type of the argument of set for array access is also tricky
```

- Names are mandatory for instances of primitive and array types. Names are optional for instances of pair and operation types. Names should also uniquely identify instances. That is a runtime error should be reported in case two instances had the same name.

Help and advice

This section will be filled as needed after questions and discussions on moodle and in class.

Problem 4, Due Wednesday February 15, 2017

1. Provide a vision statement on how your design and implementation consider possible extensions and uses of your designed system.
2. Summarize the principle design decisions that you made.
3. Summarize the working design decisions that you made.
4. Maybe you should have started from here!