

## CS3723 HOMEWORK #5, DUE: 2pm, 3/13/08

Note: If you collaborated with your classmates or used their notes, please note which classmates you collaborated with. If you use an external source, besides the text book, lectures, notes provided by the instructor, and your own intellect, please cite that source. Use quote marks if you are quoting material word-for-word from any source (including the text book).

### TYPES (Chapter 6)

- What is a type?
  - List three reasons types are used in programming languages.
- Define *type error*.
  - Give an example of an operation and operands that would produce a hardware exception.
  - Give an example of an operation and operands that would not produce a hardware exception but would not behave as expected because of the type of the operands.
  - Which of these two examples are type errors? (answer 'neither,' 'first,' 'second,' or 'both')
- Define *type safety*.
  - List three language features found in C, C++, and/or Pascal that cause those languages to not be type safe.
- List one advantage to using run-time type instead of compile-time type checking?
  - List two advantages of compile-time type checking over run-time type checking?
  - Why does compile-time type-checking need to be conservative if it is to be safe?
- What is *type inference*? How does it differ from simple type checking?
- What is polymorphism?
  - What is parametric polymorphism? Give an example of an ML program exhibiting parametric polymorphism and explain how the example demonstrates parametric polymorphism.
  - What is subtype polymorphism? Give an example of a Java program exhibiting subtype polymorphism and explain how the example demonstrates subtype polymorphism.
  - What is ad-hoc polymorphism? Give an example of a Java program exhibiting ad-hoc polymorphism and explain how the example demonstrate ad-hoc polymorphism.

### TYPE INFERENCE EXERCISES (Chapter 6)

For each of the following ML functions, (1) draw an "Parse Graph,"<sup>1</sup> (2) label each node with that nodes type (if it is trivially known) or a type variable, (3) list the set of a type constraints derivable from (the application and abstractions nodes in) the "parse graph," (4) solve constraints to determine the type of each variable, and (5) determine the type of the function.

- ```
fun g(x) = 10.0 - x;
```

(You can ignore the ad-hoc polymorphism and use 'real \* real -> real' as the type of '-'.)
- ```
fun g(f,x,y) = f(x) div y;
```
- ```
fun g(f,x,y) = (2+f(3*x),x,y);
```

(You can use 'int \* int -> int' as the type of '+' and '\*'.)
- ```
fun g(x,xs) = x::xs;
```

(The type of :: is 'a \* 'a list -> 'a list where 'a parameterizes the polymorphic 'a list' type.)

---

<sup>1</sup>Also called "abbreviated parse tree"; Mitchell, p. 137, p. 157ff. This terminology is non-standard; a more accurate description might be "type-annotated abstract syntax tree."