CS 320 – 08 December 2016

(extending handout `16_10_20_TopicsSoFar.pdf`)

Topics Mentioned/Discussed So Far

1. **avoid recomputation**, **memoization/dynamic programming**

2. **higher-order functions**, **functions as first-class objects**

3. **infinite lists** (defined but not evaluated)

4. **lazy evaluation** (infinite lists evaluated as much as needed)

5. **lazy evaluation/call-by-name evaluation**
   versus **eager evaluation/strict evaluation/call-by-value evaluation**

6. **recursion** more powerful than other mechanisms for repetitive execution

7. **reasoning about infinite lists** –
   what list is defined by "`nats = 0:(map (1+) nats)`"?
   what list is defined by "`twos = 1:(map (2*) twos)`"?
   what list is defined by "`fibs = 0:1:(zipWith (+) fibs (tail fibs))`"?

8. **tail recursion** versus **recursion in general**

9. **tail recursion** via **continuation-passing style** or **accumulating-parameter style**,

10. **type inference**, **type checking**

11. **polymorphic types**, **monomorphic types**

12. **scoping**, **encapsulation**, **implementation hiding**, **help functions**

13. **local definitions** versus **global definitions**

14. **imperative programming**/**programming with side effects**

15. **(pure) functional programming**/**programming without side effects**,

16. **referential transparency**/**functional programming** versus **referential opaqueness**/**imperative programming**

17. **pattern matching**

18. **list comprehension**

19. **vocabulary**/**lexing**/ **parsing**/**tokens**

20. **regular expressions**/**regular grammars** and **regular languages**, **context-free grammars**/**BNF** and **context-free languages**

21. **associativity** and **commutativity** of binary operators

22. **rules of precedence** of binary operators, **implicit parenthesization**

23. **rules of association** of binary operators, **implicit parenthesization**

24. **function application** (in program code) is left-associative: "MNP" means "$((MN)P)$"

25. **arrow constructor "$\rightarrow$"** (in type expressions) is right-associative: "$t_1 \rightarrow t_2 \rightarrow t_3$" means "$(t_1 \rightarrow (t_2 \rightarrow t_3))$"

26. **curried**/**consumes arguments sequentially**

versus **uncurried/consumes arguments simultaneously**

27. **user-defined types**, **in Haskell with keywords:** **type**, **newtype**, **data**

28. **recursively defined datatypes** (also called **algebraic datatypes**), introduced with keyword **data**

29. **binary trees**, different versions (labels at the internal nodes only, labels at the leaf nodes only, labels at both), polymorphic or monomorphic

30. **constructors** and **selectors** on binary trees (and other recursively defined datatypes)

31. **foldr**, **foldl**, **foldl'** on lists and their generalizations on recursively defined datatypes (e.g., `trees` using keyword **data**)

32. **map**, **filter**, and other list functions and their generalizations on recursively defined datatypes (e.g., `trees` using keyword **data**)

33. **type classes** and **instance of classes**

34. **most general types** (also called **principal types**)

35. **substitution** and **unification** (background for implementation of type-inference, and call-by-value and call-by-name interpreters)

36. **formal semantics** of programming languages ("denotational smeantics" versus "operational semantics", specified by rules of inference for **operational semantics**)

37. **lambda calculus** (its syntax and reduction rules, "safe substitution" to avoid name capture, **normal order reduction** = leftmost outermost reduction, **applicative order reduction** = leftmost-innermost reduction) *

38. **lambda calculus** in relation to programming languages (desugaring/translating high-level programming constructs in Haskell or Python – at least in the **pure functional** parts – into equivalent lambda-calculus constructs)

* Disallowing reductions under lambdas in normal order = call-by-name.
Disallowing reductions under lambdas in applicative order = call-by-value.

39. **referential transparency** and **side effects**

40. **programming paradigms** (**functional** versus **imperative** versus **declarative** versus **logic programming**)

41. **reasoning about programs** (all remaining items to the end of this list)

42. **loop invariants** (for reasoning about loops, iterative parts, or recursive parts, within the same program)

43. **invariants** (equivalent to **induction hypotheses** in proofs by inductions)

44. **proofs by induction** (to prove correctness of an iteration or a recursion within the same program)

45. **proofs by induction** (to prove equivalence of two separate programs)

46. **proofs by induction** (to prove equational properties relating two or more programs)

47. **simple induction**, **strong induction**, **structural induction**

48. **termination conditions** for proofs by induction (equivalent to **escape clauses** required to terminate iterations and recursions – a well-ordering on the induction parameters)