CS 381: Final Exam Review

* Pure language
  + It always returns the same output for the same inputs
  + Does not do anything else (no “side effects”)
  + HASKELL: “function” == pure function.
  + **Haskell** is both Object language and Metalanguage.
* Functional Programming
  + **Type**
    - Goal: Argument -> Result
    - Atomic Type (e. g. Int, Char)
      * Arg: Apply function to it
      * Res: output of another function.
    - Algebraic Data Type
      * Arg: Pattern matching
        + Case analysis
        + Decompose into parts
      * Res: build with data constructor
    - Function Type
      * Arg: Apply it to something
      * Res: Function composition or partial application
      * Build with lambda abstraction.
  + **Type Inference**
    - If a literal, data constructor, or named function: write down type.
      * Pick an application ()
      * Recursively infer their types ( and )
      * should be a function type
      * Unify , yielding type variable assignment
      * Return ( with type variable is substituted)
    - Else: type error.
  + Expression
  + Value
  + Function
* Syntax (Structure)
  + **Grammar**
    - Metalanguage for describing syntax.
    - A program is in the language IFF it can be generated by the grammar.
  + **Abstract Syntax** (defining semantics)
    - Captures the essential structure of programs.
    - Typically tree-structured
    - Ex: if t t t
  + **Concrete Syntax** (written source code)
    - Describes how programs are written down.
    - Typically linear (e.g. as text in a file).
    - Ex: if t then t else t
  + Parsing
    - Transforms: *concrete syntax* (src) -> *abstract syntax tree* (ast).
    - Steps:
      * Lexical analysis (chunk character stream into tokens)
      * Generate parse tree (parse token stream into intermediate “concrete syntax tree”)
      * Convert to AST
    - Pretty Printing (opposite of parsing: *abstract syntax tree* (ast) -> *concrete syntax* (src).
  + Translating grammars into Haskell data types (grammar -> Haskell)
    - For each basic nonterminal, choose a built-in type, e.g. **Int**, **Bool**
    - For each other nonterminal, define a data type
    - For each production, define a data constructor
    - The nonterminals in the production determine the arguments to the constructor
    - Special rule for lists:
    - in grammars, *s* ::= *t*\* is shorthand for: *s* ::= \_ | *t s* or *s* ::= \_ | *t* **,** *s*
  + **Abstract Syntax Tree (AST)**
    - Captures the essential structure of a program
    - (Everything needed to determine its semantics).
  + **Object Language**
    - Language we’re defining.
  + **Metalanguage**
    - The language we’re using to define the structure and meaning of the object language.
* Denotational Semantics (Meaning)
  + Formal Specification:
    - Denotational Semantics: relates terms (AST) directly to denotation (value in Semantic domain).
    - Operational Semantics: describes how to evaluate a term.
    - Axiomatic Semantics: describes the effects of evaluating a term.
  + Desirable properties of a denotational semantics
    - **Compositionality**: a program (AST)’s denotation is built from the denotations of its parts (sub-AST)
      * supports modular reasoning, extensibility
      * supports proof by structural induction
    - **Completeness**: every value in the semantic domain is denoted by some program
      * ensures that semantic domain and language align
      * if not, language has expressiveness gaps, or semantic domain is too general
    - **Soundness**: if two programs are “equivalent” then they have the same denotation
      * equivalence: e.g. by some syntactic rule or law
      * ensures the equivalence relation and denotational semantics are correct
* Semantic Domain
* Valuation Function
  + : abstract syntax -> semantic domain  
    data Term = ... -- abstrax Syntax, T  
    type Value = ... -- semantic domain, V  
    sem :: Term -> Value -- Valuation function, [[a]] : T -> V
* Type Systems
  + Type
  + Static Typing
  + Dynamic Typing
  + Typing Relation
* Naming and Scope
  + Name
  + Declaration
  + Binding
  + Reference
  + Shadowing
  + Dynamic Scope
  + Static Scope
  + Environment
  + Closure
* Parameter Passing
  + Call-by-value
  + Call-by-name
  + Call-by-need (lazy)
* Logic Programming
  + **Atom -**  Primitive Value
    - lowercase (str of char, num, \_)
    - Single quoted strings (‘Hello World!’)
    - numeric literals (123, -345)
    - empty list ([])
  + **Variable**
    - Used in rules and queries.
    - UPPERCASE letter (str of char, num, \_).
    - Underscore (\_: “Don’t care”).
  + **Predicate ≌ relation ≌ set**
    - Def: Basic entity in Prolog
    - Defined in file, queried in REPL
    - Predicates with the same name but different arities are different predicates!
    - Unary: hobbit(bilbo).
    - Binary: likes(frodo, ring).
  + **Goal / query**
    - Different from passing arg to func.
    - ?-
  + Database
  + **Fact**
    - The *predicate* matches the *goal*.
  + **Rule**
    - head :- body
    - The head is true if body is true.
  + Goal search
  + Arithmetic Equality (A =:= B)
    - Arithmetic expressions to check if **numerically equal** (A=5;B=5; A==B).

**?-** X is 3\*5.

X = 15.

* + - Nested predicates (structured data) – written infix.
  + **Unification (A=B)**
    - Assignment of variables that makes its arguments **syntactically** **equal**.

**?-** X = 3\*5.

X = 3\*5.

* + **Cut (!)**
    - Special atom used to prevent backtracking.
    - Always succeeds; commits current goal search matches and assignment made so far.
    - Green Cut:
      * Doesn’t affect the member of a predicate
      * To prevent “false.”s (for efficiency)
    - Red Cut:
      * !greencut
      * Cuts too early -> affects logic of the predicates.
  + SWI-Prolog logistics

[filename]. Loads def from “filename.pl”

listing(P). List facts and rules related to predicate P

trace. Turn on tracing

nodebug. Turn off tracing

help. View documentation

halt. quit.

Other shit:

-- map: basically splits the functions and turns into list.

map :: (a->b) -> [a]->[b]

map f [] = []

map f (x:xs) = f x : map f xs

--example:

map f [2,3,4] = [f 2, f 3, f 4]

-- foldr: loop for aggregating elements in a list

foldr :: (a->b->b) -> b -> [a] -> b

foldr f y [] = y

foldr f y (x:xs) = fx (foldr f y xs)

--example:

foldr f y [2,3,4] = f 2 (f 3 (f 4 y))

filter :: (a -> Bool) -> [a] -> [a]

(.) :: (b->c) -> (a->b) -> a->c

f . g = \x -> f (g x)