CS 381: Final Exam Review

* HASKELL = 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**
    - Can be combined in 2 ways
      * Sum: contains value from one domain or the other
        + Haskell Either a b or new data type
      * Product: contains a value from both domain
        + Haskell: (a,b) or new data type
      * Can errors occur?
        + Haskell: maybe
      * Does the language manipulate state or use names?
        + Use function type
      * Example stateful domains
        + **Read-only state**: State -> Value -- immutable
        + **Modify as only effect**: State -> State
        + **Modify as side effect**: State -> (State,Value) -- mutable
  + **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**
    - A set of syntactic terms (ASTs) that share the same behavior
    - Defines the interface for these term (in what context can they appear?)
  + Type Error
    - Occurs when a term cannot be assigned a type
    - A violation of type interface between terms.
    - If not caught/prevented -> crash/unpredictable evaluation.
  + Type System
    - Detects and prevents/reports type errors.
    - Type safe:
      * If an implementation can detect all type errors
      * Statically: by proving (absence of type errors)
      * Dynamically: detecting and reporting (at runtime)
  + **Static Typing** (Syntactic terms (AST ->type))
    - “Which programs have meaning?
    - Type errors -> reported(compile time), prevent execution
    - Type checker -> proves no type error (at runtime)
  + **Dynamic Typing** (Runtime values (AST -> value))
    - “What is the meaning of this program?”
    - Type errors -> reported(runtime), exception thrown
    - Type checker -> integrated in runtime.
  + Benefit of Static Typing
    - Usability and comprehension
      * Machine-checked documentation
        + Guaranteed to be correct and consistent with implementation.
      * Better tool support
        + Code completion, navigation
      * Supports high-level reasoning
        + Named abstractions provided for shared behavior
    - Correctness
      * A partial correctness proof (no runtime type errors)
        + Improves robustness, focus testing on more interesting errors.
    - Efficiency
      * Improved code generation
        + Can apply type-specific optimizations
      * Type erasure
        + No need for type information or checking at runtime.
  + **Typing Relation**
    - Typing is just a semantic with a different semantic domain.
    - Defining a static type system
      * Define the abstract syntax, E
        + The set of AST
        + Data Exp = …
      * Define the structure of types, T
        + Another abstract syntax
        + Data Type =
      * Typing relation, E: T
        + typeOf :: Exp -> Type
    - Then we can define a dynamic semantic that assumes there are no type errors.
* Naming and Scope
  + **Name**
    - **Declaration (add)**
      * Introduces a new name.
    - **Binding (set)**
      * Associate a name with a thing
    - **Reference (get)**
      * Use the name to stand for the bound thing.
  + Scope
    - The part of the program where that name can be referenced.
    - Block
      * Shared scope of a group of declared names.
    - Shadowing
      * When an inner block temporarily hides a name in an outer block (let).
    - **Static Scope (compile time)**
      * typeOf Exp -> Type
      * Function Call
        + Save stack (current)
        + Restore stack (function)
        + Push frame (Parameter -> stack)
        + Run func(return val)
        + Restore stack (saved) & resume
      * Tradeoffs:
        + Names are not part of public interface

No risk of name collision (predicted behavior)

Improved modularity (name change w/o clients breaking)

* + - * + Only supports planned extensibility.
    - **Dynamic Scope (runtime)**
      * Function Call
        + Push frame (Parameter -> stack)
        + Run frame (return val)
        + Pop frame (stack) & resume
      * Tradeoffs:
        + Supports ad-hoc extensibility
        + All names are part of public interface

Risk of name collision (unintended behavior)

Bad modularity (hard to refactor and understand)

* + - **Environment (Typing Context)**
      * A mapping from names to things
      * Just a flat stack
      * Type Env = Name -> Thing
  + **Closure** = function + stack (its env)
    - Needed to implement static scoping
* Parameter Passing
  + **Call-by-value (static)**
    - How it works:
      * Evaluate all arg -> #
      * Store (# -> env)
      * Evaluate function
      * Ref -> func(env)
    - Pro:
      * Evaluates all once.
      * Predictable
    - Con:
      * eval(! need)
      * If eval()==error,  
         !terminate || err
  + **Call-by-name (dynamic)**
    - How it works:
      * Stores(arg -> env) (arg = var)
      * Eval(func)
      * ref -> eval(env) everytime.
    - Pro
      * !eval(!need)
      * If unused arg -> error, still good.
    - Con:
      * If call(arg = n times),  
         time = time x n.
  + **Call-by-need (lazy)**
    - How:
      * Stores(arg-> env) (arg = var)
      * Eval(func)
      * ref -> eval(env) once.
      * Result = ref(saved(env))
    - Pro:
      * Evaluates all 0 or 1 times.
    - Con:
      * Needs to cover all cases
      * Less predictable than call-by-value (in terms of space)
* 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”).
    - Cannot be used for predicates.
  + **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.
    - ?-
    - Responds with T/F or provides true binding for each query.
  + **Fact**
    - The *predicate* matches the *goal*.
  + **Rule**
    - head :- body
    - The head is true if body is true.
  + **Basic Algorithm for solving a (sub)goal (S6, p 23)**
    - Linearly search database for candidate facts/rules
    - Attempt to unify candidate with goals
    - If (unification), case:
      * Fact: we’re done with this goal. (**true**)
      * Rule: (predicate of its head matches the goal)
        + Add body of rule as new subgoals to the list.
        + Substitute variables in all goals in the list.
    - Else: keep searching.
    - Backtrack if we reach the end of the database.
      * For each subgoal, prolog maintains
        + The search state (goals + assignments) before it was produced.
        + A pointer to the rule that produced it.
      * If subgoal fails
        + Restore the previous state
        + Resume search for previous goal from the pointer.
      * Else: return **false**.
  + Recursive Rules
    - Always list non-recursive case first (in database and rule bodies)
    - Use helper predicate to enforce progress during search.
  + Relationship to Haskell Data Types
    - Haskell:
      * Build values with **data constructors**
      * Data types **statically define** valid combinations
    - Prolog
      * Build values with predicates
      * Uses **rules to dynamically identify** or enumerate valid combinations.
  + List Patterns [head|tail]
    - Structured data with special syntax.
    - Similar to Haskell but can be heterogeneous.
    - [3,4] == ‘[|]’(3,’[|]’(4,[]))
    - [|] = cons [] = nil
  + 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.
    - Arithmetic operations: + - \* / mod
    - Comparison operations: < > =< >= =:= =\=
  + **Unification (A=B)**
    - Assignment of variables that makes its arguments **syntactically** **equal**.
    - A=B means attempt to unify A and B.

**?-** 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)

data Maybe a = Nothing | Just a

data Expr = Lit Int -- case 1

| Plus Expr Expr -- case 2

-- Expr = type name

-- Plus = data constructor

-- Expr Expr = types of arguments

data List a = Nil -- a = type parameter

| Cons a (list a)

-- first a = reference to type parameter

-- (List a) = recursive reference to type

-- Grammar (BNF notation)

s 2 Sentence ::= n v n | s and s

n 2 Noun ::= cats | dogs | ducks

v 2 Verb ::= chase | cuddle

-- note: '2' are "with in"

-- Right of '2' = syntactic category

-- n v n = nonterminal symbol

-- cats, chase etc. = terminal symbol

-- right of ::= production rules.

-- Abstract Grammar vs. Concrete Grammar

Abstract:

t 2 Term ::= true

| false

| not t

| if t t t

Concrete:

t 2 Term ::= true

| false

| not t

| if t then t else t

| ( t )

--Denotational Semantics

Valuation Function:

data Term = ... -- abstrax Syntax, T

type Value = ... -- semantic domain, V

sem :: Term -> Value -- Valuation function, [[a]] : T -> V

--Dynamic vs Static Scope

Dynamic: called during runtime

sem :: Exp -> Val

Static: called during compile time (src)

typeOf :: Exp -> Type

-------------------------

-- Prolog --

-------------------------

-- Negation as failure (Prolog)

not(P) :- P, !, fail.

not(P).

-- List Patterns

?- story([X,Y,Z|V]).

X = 3,

Y = little,

Z = pigs,

V = [].

[3,4] == ‘[|]’(3,’[|]’(4,[]))

-- [|] = cons

-- [] = nil