**Study Guide:**

*Syntax and Semantics of Prog Langs:*

* Syntax - The language you write programs in
  + Valid keywords and symbols
  + Rules specifying how to combine symbols and keywords
  + Rules are specified as a grammar
  + Grammar rules implemented in a parser
* Semantics - The “meaning” of the language
  + Operational Semantics
    - To reason about the execution of a specific program, we need a mathematical specification of the behavior of all possible programs we could written in our language
  + Axiomatic Semantics
  + Types - an additional static check carried out before program is ran
* Context-free grammars (CFGs)
  + A set of recursive rules used to generate patterns of strings and further
  + Set of terminal symbols (tokens)
    - The symbols that can actually appear in programs
      * Ex. 1, (, ), data, x, y
      * {a, b}
  + Set of nonterminal symbols
    - Don’t actually appear in programs
    - Represent the “internal structure” of a program
    - Different “kinds of thing” in program
      * Ex. f x = x + 1
      * Patterns, operands, variable, expression which result in a function, function definition are all nonterminal symbols
      * {S, A}
    - Different “categories” represented by different non-terminal symbols
    - “Special” non-terminal representing a complete program (start symbol)
  + Set of grammar rules
    - Specifying how to build up syntactically valid programs from terminal (and non-terminal) symbols.
* Simplest language
  + Term t defining terms like true, false etc is here
  + Ex. if pred then 0 else then false
* Inference rules
  + Axiom idea from here
  + t ∈ T
    - “t is a member of the set T”
    - T represents a set of all valid terms.
  + All the axioms for type T here
  + Type checking is important
  + I know how to do this
* Small-step operational semantics
  + Describes how such an execution in terms of successive reductions of an expression, until we reach a number, which represents the result of the computation
  + Two results
    - Good: Reached a value, or
    - Bad: Getting stuck
  + t → t’ → t’’ → … → t value or stuck
  + Ex. if iszero (pred 0) then succ 0 else 0 → if iszero 0 then succ 0 else 0
* Typing
  + Rule our such “bad” programs statically (before run time)
  + If program passes type checker without getting stuck it is guaranteed to have eros
  + They categorize the different sorts of values in our program
  + Well-typed or not well-typed
  + Ex. pred (if false then 0 else succ 0): Nat
* Type Safety
  + Type Safety = Progress + Type Preservation
    - Progress
      * If t : T then either t is a value or t can reduce to t’
      * Only says something about well-typed terms
      * The actual reduction
      * Presents “getting stuck” at a non-value where no rules apply
      * Keep applying progress by reduction
    - Type Preservation
      * If t : T and t → t’ then t’ : T is also well-typed
      * Says something about t and its reduction term t’
      * The term reduced to/types are preserved by reductions
      * Keep checking types be type preservation
  + Formalize correctness guarantees obtained if program passes type checker
  + Formalize relationship between typing rules (t:T) and evaluation rules (t → t’)
  + Type safety is a property of the language, not of individual programs
  + Does not say anything about terms that are not well-typed
  + Doe not guarantee termination or determinism
* Metaprogramming
  + Implementing the typing and reduction rules in Haskell

*Untyped Lambda Calculus*

* Language based entirely on functions and that's all
* Three types
  + Variables (x)
  + Abstractions (x.t)
  + Application (t t)
* Think of these with AST
* Remember bounds, variables captured
* Free variables are when they are not bound by any abstraction
  + A term with no free variables is a closed term
* Alpha-renaming
  + The “good” renamings are those that preserve binding structure of term (capture avoiding)
* Alpha-equivalence
  + Two terms that can be made to look identical by an alpha-renaming are called alpha-equivalent
  + This is a equivalence relation
* Substitution
  + I know this
* Reductions
  + I know this
* Remember these
  + tru = x.y.x
  + fls = x.y.y
  + Church Numerals
    - c0 = s.z.z
    - c1 = s.z.s z
    - c2 = s.z.s (s z)
    - c3 = s.z.s (s (s z))
* Know these but don’t need to remember
  + not = b.b fls tru
    - Means “if b then fls else tru”
  + and = b1.b2.b1 b2 fls
    - Means “if b1 then b2 else fls”
  + succ = n.s.z.s (n s z)
    - succ c2
  + add = i.j.i succ j
    - add c3 c20
  + times = i.j.i (add j) c0
    - times c2 c4
  + iszero = n.n (x.fls) tru
    - iszero c0
  + fst = p.p tru
  + snd = p.p fls
  + pair = x.y.z.z x y
  + pred = n. fst (n ss zz) where ss = pair c0 c0 and zz = p.pair (snd p) (succ (snd p))
    - pred c0
  + minus = i.j j pred i
    - minus c3 c2
* Full Beta Reduction
  + Can reduce any redex in any order
* Call-by-name
  + Leftmost, outermost redex reduce first, but no reduction allowed inside a
* Call-by-value
  + Like call by name but the RHS of a redex must be evaluated to a normal form (value) before substituting
  + Just worry about this one
* Fix
  + General recursion keyword in UTLC
  + Does not work for STLC because fix is not well-typed unless if new axioms are added to account for fix