**Final Study Guide**

Does the languageallow user defined types? Imrpoves readability/writeability. OO!=good

Functions – 2nd class citizens C++, 1st class in Haskell

What tools exist for encapsulation. | User defined type constructer. Return types. Operates on type return other kind of type. Vector int.

Critical for generic programming style – diff style. Move pattern matching, duck typing c++

**Template:** Work with concepts. Any type that does this thing. Just have properties.

**Expression:** Means of expressing computation, Combination of values and operators that has a value

**Operator Overloading**: Can users overload operators? Does the language overload operators? How does this affect the language?

**Side Effects:** An observable change of global state made by a function call. Ex. Output param, global var, class var, IO

**Referential Transparency:** If an expression has no side effects it can be thought of as a reference to its value. Functions are called **pure** if they have this property. Four major causes of impurity: Error, Non-determinism, Context, and Destruction. Destruction that increases entropy cannot undo. Can’t retract output

**Short-Circuit Evaluation:** If the value of an expression can be and is determined without revaluating the whole expression it is called short circuit evaluation. Common with Boolean algebra 0\*X – 0, 1+X=1. Also possible with memorized pure functions

**Lazy vs Eager Evaluation:** Eager – expressions become values at earliest opportunity. Lazy – expressions become values at the latest possible moment. Very common in functional languages. Allows expression of infinite objects – but not their evaluation

**Arithmetic Expressions:** Unary, Binary – (Infix, Prefix, Postfix), Ternary

**Boolean Expressions:** Comparisons – Two way and Three way comparison, Boolean algebra

**Assignment:** Procedural Languages – Write to memory, always a side effect. Functional and Logic – Create a new name binding to a constant value.

**Type Conversions:** Narrowing, widening, casting. Which operations are allowed? Which are implicit and which explicit? Are mixed mode expressions allowed?

**Control – Selection**: Statements that allow conditional execution

**Two Way:** If – then – else. What is the type of the control expression? What is the form of the clause? Single statement? Block?

**Nesting ambiguity:** if e1 if e2 c1 else c2. Where does the else belong? Only occurs if clause form permits

**Multiple Selection:** Switch – Case – Patterns. What is the form of the selection expression? How are the cases specified? What if nothing matches?

**Multiple Selection implementation:** Nested ifs, Tree – (Command Pattern), Jump Tables

**Iteration:** Used for repetition, How is it controlled, Where does the control structure appear, Can be replaced by recursion

**Counter Controlled Loops:** Has a loop variable, Loop variable has a begin, end, and step. For, For each

**Logically Controlled Loops:** Has a condition, Is the test before or after the loop, While(e), do..while(e)

**Loop Control Statements:** User located loop exits – break, last. Skip statement- continue. Used to eliminate some uses of goto

**Unconditional Branch:** Most flexible and powerful of statements. Other control structures can be implemented in terms of goto. Some languages don’t have it.

**Goto Considered Harmful:** Dijkstra’s 1968 seminal paper

**Guarded Commands:** Introduced by Dijkstra in 1975. A block of statements with Boolean guards. One expression whos guard is true is executed.

**Subprograms – Subprogram:** Basic subprograms have: Single entry point, suspend the caller, Multiple entry points gives coroutines, Avoiding suspension gives concurrency

**Definition:** The definition includes: Interface, Actions

**Call and Return:** Call – the request to enter a subprogram. Return – The resumption of the calling program (possibly with a value). A subprogram is active between call and return

**Procedures and Functions:**  Procedures do not return, are intended as extension points for statements in the language and mostly a feature of older languages. Functions return, modeled on math functions, and generally should not have side effects.

**Coroutines:**  Include yield and resume. Yield returns a value but maintains current state. Resume restarts co-routine after last yield. Call and return still exist and define the lifetime.

**Side effects:** Ways in which a CS function is not like a math function. Context: global variables, static local variables. Error, Non-determinisms, Destruction – I/O, out parameter.

**Referencing Environments:** Set of bindings visible to a subprogram – local variables, nested subprograms.

**Closures:** A subprogram and its referencing environment are called a closure

**Return Values:** What are the types of return. What are the number of return types

**Formal and Actual:** The parameter definitions in the header are called formal. The parameter values in a call site are called actual.

**Positional and Keyword:** If the matching between formal and actual parameters is based only on order then the language uses positional parameters. If each actual parameter can be associated with, a formal parameter name in any order the language used keyword parameters. Ex foo(bar=42)

**Parameter Passing:** Pass by value – Only the value is passed (a copy), Pass by Result – A local variable is created and the value (result) is copied into caller at end of function., Pass by value result – Copy passed to function and Value copied back into caller and also called pass by copy, Pass by reference – Create and copy an alias, Pass by Name – As if parameter was textually substitute. Referencing environment must also be included for name lookups.

**Type Checking Parameters:** Do formal parameters have a type? Do formal and actual parameters have to match?

**Multidimensional arrays as parameters:** A language needs to be able to build the array mapping. This complicates passing arrays. Sens a pointer and do pointer arithmetic. Less flexible functions (Specific array size and layout). More complex built in arrays.

**Subprograms as parameters:** How can subprograms be passed – What restrictions are there? What is the referencing environment? Call statement- Shallow Binding, Passed function definition – deep binding, Specified at call site – ad hoc binding

**Indirect Subprograms:** Function pointers delegates – A callable and assignable object. Virtual functions – Implemented in terms of indirect subprograms.

**Overloaded functions:** Subprograms with the same name and referencing environment. Each must have a different protocol (number, type, and order of arguments).

**Overloaded Operators:** Some languages (C++, Ada, Python, Ruby, others) allow operators to be overloaded. Usually some special function name is invoked by operator syntax.

**Generic Subprograms:** Generic subprograms work on multiple types. The concept of a parameter is what the generic subprogram expects. A type is said to model the concept if it meets the requirements. Generic programs work on all types that model their concept.

**Prologue and Epilogue:** Function call must: Suspend caller, compute and pass parameters, pass return address, Transfer control. Return must: Resolve out parameters, pass return value, return control, resume caller into previous state

**Activation Records:** Data needed by every invocation of a function. Stack local variables. Parameters. Return address. Dynamic Link. Static Link

**Example: Recursive Factorial:**

**Blocks:** Entering a block adds a new activation record. Chains of static links used to lookup non-local names

**Dynamic Scope:** Deep access – lookup names using dynamic links, Shallow access – maintain a stack for each name, Semantics are identical.

Haskell: Factorial: Fact 0 =1; Fact n=Fact (n-1)n;, Fibb 0=1; Fibb 1 = 1; Fibb n = Fibb(n-1 ) + Fibb(n-2);

quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []

quicksort (x:xs) =

let smallerSorted = quicksort [a | a <- xs, a <= x]

biggerSorted = quicksort [a | a <- xs, a > x]

in smallerSorted ++ [x] ++ biggerSorted

List Equivalence listA == listB

Reverse:

reverse' :: [a] -> [a]

reverse' [] = []

reverse' (x:xs) = reverse' xs ++ [x]

Max:

maximum' :: (Ord a) => [a] -> a

maximum' [] = error "maximum of empty list"

maximum' [x] = x

maximum' (x:xs)

| x > maxTail = x

| otherwise = maxTail

where maxTail = maximum' xs

**ABET QUIZ**

Computer language first used BNF to describe syntax: ALGOL

Static Bindings: Value of a global constant. The meaning of a reserved word

**Lifetime of variable:** The time during which the variable is bound to a memory location

**Scope of Variable:** The collection of statements where the variable is visible.

**In statically scoped language:** Variable scope depends on execution order

All non-local scopes are global scopes = False

Which array allocation strategy allows for array growth? Heap Dynamic

Pointer problems that are also problems with references: aliasing

Which is not a step in logic program execution? Hypothesis Formation

Which of the following statements are true for a referential transparent function? The function has no side effects. The function can be memorized (results of one invocation stored and used in lieu of a second invocation).  
1968 Paper Edsger Dijkstra argued against which control structure? Unconditional branches

Which of the following control statements is non-deterministic? Guarded Command

Which of the following refers the names and types of parameters? Formal Parameters

Which of the following parameter passing methods can be used for output parameters? Pass by copy, pass by name, ,by result, and reference

Shallow Binding: Referencing env of the call of statement. Deep Binding: … of the closures subprogram. Adhoc: given as param at call site

Which is not part of an activation record? Spilled Registers

Infinite data structures: Lazy Evaluation