CS430 Programming Languages

Exam 2 Study Guide

1. **Chapter 16: Declarative Languages**
   1. Explain the formal basis for logic languages and for rule-based languages.
      1. **Logic** : Based on First Order Predicate Logic
         1. Prolog
         2. Inference by goal-directed resolution
      2. **Rule** : Based on Production system model
         1. OPS5, CLIPS, JESS
         2. Inference by data driven deduction
   2. List and explain the differences between declarative and imperative languages, including their design objectives, computational model, and implementation.
      1. Computational Model
         1. Imperative – Von Neumann Architecture
            1. Machine architecture dictates design
            2. Uses variables tied to memory (state)
            3. Algorithms are sequences of CPU instructions
         2. Declarative – Solid theoretical basis (FOPL or PPS)
            1. Computationally equal to turning machine
            2. Unconcerned with the architecture of the machines on which the programs run
      2. Design Objectives
         1. Imperative
            1. Efficiency is primary concern
            2. Suitability for software development is secondary
            3. Control through iteration and selection
         2. Declarative
            1. Clean and rational semantics is important
            2. High level operations are important (more powerful than imperative)
            3. Flexibility of expression is important
            4. No algorithms – inference engines instead
      3. Implementation
         1. Imperative
            1. Faster execution (compiled)
            2. Concurrency is difficult
         2. Declarative
            1. Slower execution (interpreted, inference engine)
            2. Automatic concurrency (referential transparency, monotonic logic)
   3. **Evaluate Prolog expressions. – look on canvas**
   4. Describe typical applications of declarative languages, and explain their strengths and weaknesses.
      1. Applications:
         1. Expert Systems, natural language processing, planning systems, diagnosis, theorem proving
      2. Strength:
         1. Semantics are sound, based on FOPL or PPS
         2. Programs are concise (no algorithms to write)
         3. Processing is naturally parallel (referential transparency)
      3. Weaknesses:
         1. Prolog negation problem (closed world assumption)
         2. Inference can be expensive
         3. Resolution order depends on program sequence (Violates the declarative programming paradigm).
2. **Chapter 5: Variables**
   1. List and describe the essential attributes of a variable.
      1. Name – The identifier of the variable (Anonymous variables may not have explicit names)
      2. Address - the memory location to which a variable has been bound
      3. Value – The value stored in the variables memory location
      4. Type – The data type associated with the variable
      5. Lifetime – The period between allocation (memory binding) and deallocation
      6. Scope – the range of source code from which the variable can be accessed
   2. Explain and evaluate language options regarding variable naming
      1. Name Length
         1. Too short names not meaningful
         2. Too long and they take away from readability
      2. Connectors
         1. Common connectors include: **hyphens, underscores or blank spaces**
         2. Most languages allow hyphen or underscore
            1. C++, java, perl
         3. Some allow no connectors
            1. Pascal, Fortran 77
      3. Case Sensitivity
         1. **Advantages**: expands range of useable names
         2. **Disadvantages**: May cause confusion
            1. Detracts from readability
            2. Programmers must remember cases of variables
         3. Java, C++, PERL, and C are case sensitive
         4. Cobol, Eiffel are insensitive
      4. Special Words - A word that has a predefined meaning in a language
         1. Reserved Words – Cannot be used besides its meaning in the language (EX. public in java)
            1. Disadvantage: excess reserved words limits expressivity
         2. Keywords – Can be used as keyword or name of variable in a language (EX. REAL in FORTRAN)
            1. Disadvantage: Can be confusing, detracting from readability
      5. Alias – an alternate reference used to access a memory location
         1. Pointers, reference variables
         2. Common blocks (FORTRAN), redefines(COBOL), unions(C,C++), variants (Ada)
         3. Procedure parameters (passed by reference)
         4. **Advantages**: Efficient use of storage (by assigning mutually exclusive variables to the same address)
            1. Convenience of pointers, procedure side effects
         5. **Disadvantages**: Reduced readability because there are more chances for error and program readers must remember all aliases
      6. Value – the contents of the memory location to which the variables assigned
         1. Value is subject to interpretation according to variable data type
         2. Initialization: binding of a variable to a value when its bound to memory
            1. Often done on declaration statement
            2. Some languages initialize everything (JAVA), some do not (C)
      7. Named Constants – a variable that is bound to a value only once, when its bound to memory
         1. Advantages
            1. Readability – constant values are identified
            2. Modifiability – used to parameterize programs
   3. Explain and evaluate strong vs. weak typing
      1. **Strong Typing:** a programming language in which all type errors are detected (at either compile time or runtime)
         1. Fortran, Perl, C, C++ are not
         2. Java and Ada are mostly strong
         3. **Advantage:** Allows detection of the variable misuses that result in type errors
         4. **Disadvantage:** Reduces expressivity by limiting automatic type assignment and conversion
      2. **Weak Typing:** the allowance of type conversions
         1. Can Be very dangerous
         2. Ex. C++: int and double can be converted both ways
            1. Java only int can be converted to double
      3. **Safety Type:** The degree to which a language deters type errors
      4. **Type Binding** – variable type bindings may be made during compile time (static) or at runtime (dynamic)
         1. Statically bound types may be specified : Implicitly, Explicitly or Inferred from context
         2. Dynamically bound types are inferred by assignment
      5. **Explicit Type Declaration** – a program statement that declares the types of variables (EX. int sum = 0, Initialization optional)
      6. **Implicit Type Declaration –** a default standard for specifying the types of variables by form or context (EX. Fortan integers must start with a capital I or N, else they are reals)
         1. **Advantage:** expressivity (minimizes coding)
         2. **Disadvantage:** reliability, since type assumptions may be incorrect
      7. **Static Type Inferencing** - Types are determined from context of the reference (Used by ML, Miranda, and Haskell)
      8. Strongly typed languages are more likely to catch and generate an error if an argument passed does not closely match the expected type.
         1. Ex. Java, Ada, Pascal, LISP are strongly typed
      9. Weakly typed languages may produce unpredictable results or may perform implicit type conversions
         1. Ex. Perl
   4. Describe binding times and the things that are bound at various times
      1. **Binding –** an association between an entity and an attribute or semantics
         1. Variable bound to memory cell
         2. Symbol bound to operation (+ to add)
         3. Reserved word to its semantics
      2. **Binding Time -** When a binding occurs
         1. **Language Design time –** Binds operator symbols to operations
         2. **Language Implementation Time –** Binds data type to internal representation
         3. **Compile Time –** Bind variable to a type
         4. **Load Time –** Bind a static variable to a memory cell
         5. **Runtime –** Bind a dynamic variable to a memory cell
   5. Explain and evaluate static vs. dynamic binding in general.
      1. **Static Binding** – Occurs before execution and remains unchanged throughout execution
         1. Language Design, Language Implementation, Compile
      2. **Dynamic Binding –** Occurs during execution or can change during execution
         1. Load and Run time
      3. **Dynamic Type binding -** Type is determined during assignment
         1. Javascript, Perl, PHP
         2. **Advantage**: expressivity (minimal code), and felxibility (generic program units)
         3. **Disadvantage**:
            1. High run time cost

Dynamic type checking

Interpretation or extensive runtime support

* + - * 1. Reduced reliability – type error detection at compile time is difficult or impossible
  1. Explain and evaluate alternatives related to storage location and lifetime.
     1. Static Storage Binding
        1. Static storage is bound before execution, usually at compile time
           1. Other possibilities are language design time or implementation time
           2. Ex. C static variables, Fortran 77 variables
        2. Advantages:
           1. Addressing efficiency (direct addressing)
           2. Run-time efficiency (pre allocation)
           3. History-sensitive subprograms
        3. Disadvantages:
           1. Lack of support for recursion
           2. Enlarged executable modules
     2. Static Dynamic Storage Binding
        1. Variables are bound to memory cells when their declarations are elaborated (can be implicit or explicit)
           1. Call by value parameters and local variables in C subprograms and Java methods(explicit)
           2. Return values from C procedures (implicit)
        2. Advantages:
           1. Conserves storage (variables bound only when required)
           2. Permits recursion
        3. Disadvantages:
           1. Allocation and deallocation imply runtime overhead (stack operations)
           2. Requires indirect addressing of variables (via stack), which is less efficient than direct
     3. Heap Dynamic Storage Binding
        1. The heap is a memory pool available for allocation of variables
        2. Variable lifetime is independent of procedures
        3. **Explicit heap-dynamic** storage is allocated and deallocated by explicit directives during execution
           1. **Ex.** malloc() in C and all Java objects and arrays (via new)
           2. Referenced through pointers or references
        4. **Implicit heap-dynamic** is allocated and deallocated automatically by assignment operations
           1. **Ex.** strings and arrays in Perl and Javascript
        5. **Advantage:** 
           1. Efficient use of storage – variables are allocated only when needed
           2. Flexibility for data structure creation
        6. **Disadvantages**:
           1. Pointers are less efficient that static references, and have lower reliability
           2. Large runtime overhead for heap management
           3. Possible heap exhaustion
  2. Differentiate between variable lifetime and scope.
     1. **Variable lifetime** – the time during which a variable is bound to a specific memory cell
        1. Lifetime is period between allocation and deallocation
     2. **Scope –** the range of source code statements over which a variable is visible
        1. Range it can be referenced
     3. Scope and lifetime are usually closely related, but are different concepts
  3. Define scope and scoping rules, and interpret a program based upon scoping alternatives.
     1. **Scope –** the range of source code statements over which a variable is visible
     2. Scope rules determine how references are associated with variables
     3. **Static (Lexical) Scope –** based strictly on program text
        1. Variables are discovered by searching local declarations, then increasingly larger enclosing scopes
        2. **Static ancestor**: an enclosing static scope
        3. **Static parent**: the nearest static ancestor
        4. **Static Scope Evaluation** 
           1. **Advantages:** readability, static scope is intuitive
           2. **Disadvantages:** Requires many parameters or global variables
     4. **Dynamic Scope –** Based on calling sequences (temporal vs spatial context)
        1. References to variables are resolved by searching through chain of procedures calls on the stack
        2. **Dynamic Scope Evaluation** 
           1. **Advantages**: convenience, utility routines always access data of their callers
           2. **Disadvantages**: poor readability, variable mappings are not obvious
        3. **Hidden variables** - Reusing the name of a variable from an ancestor scope hides (shadows) the variable
           1. Ex. in Java: this.name
     5. **Referencing Environment** – the collection of all names that are visible to a statement
        1. **For static scoped languages**: local variables plus all visible variables of ancestor scopes
        2. **For dynamic-scoped language**: local variables plus all visible variables of *active* procedures (i.e., those on the stack

1. **Chapter 6: Data Types**
   1. Describe the history of primitive data types (integer, real, decimal, Boolean), and the languages in which they originated.
      1. Integer
         1. Reflection of the hardware (machine code mapping is trivial)
         2. Limited range but accurate
         3. Stored as 2’s compliment
         4. Most languages have several integer types
         5. Comes from Fortan 1 (1957)
      2. Real - Comes from Fortan 1 (1957)
      3. Decimal
         1. For business applications (money)
         2. Stores fixed number of decimal digits (1 digit per byte or 2 digits per byte)
         3. Cobol (1960)
            1. Advantage: more accurate, no binary conversions or loss of accuracy
            2. Disadvantage: less efficient

Uses more storage, slower calculation if decimal ALU not present in CPU

* + 1. Boolean
       1. Either true or false
       2. Useful for decision making in programs
       3. Implemented as:
          1. Bits: more compact, slower access
          2. Bytes: wastes memory, faster access
       4. Algol 68, Ada (1983)
  1. Describe and evaluate the storage methods used for strings.
     1. **Static string length:** 
        1. Length is fixed at compile time
        2. Requires a compile time descriptor
        3. Used In FORTRAN 77, Ada, Cobol
     2. **Limited Dynamic Length String:**
        1. Capacity is fixed at compile time; actual length is dynamic
        2. Managed with a run time descriptor or string terminators
     3. **Dynamic Length Strings**:
        1. Length is dynamic and unlimited
        2. Allocation and processing are complex:
           1. Requires a run time descriptor
           2. Linked list option
           3. Dynamic array option
        3. Used in Perl and Javascript
     4. String implementation
        1. Ada, Fortran 90 – somewhat primitive
           1. Assignment, comparison, catenation, substring
        2. C
           1. Strings stored in **char** arrays, null terminated
           2. String library provides operations
           3. DANGEROUS! (buffer overflows)
        3. C++, Java
           1. Abstract data type; built-in catenation operator
           2. String objects are immutable and static length
           3. StringBuffer objects are dynamic and mutable
           4. Support for regex’s in Pattern class
        4. Perl and Javascript
           1. Dynamic strings
           2. Supports regular expressions for searching
  2. Explain and evaluate choices related to user-defined ordinal types.
     1. **Ordinal Type**: a data type in which the range of possible values can be mapped to positive integers
        1. **User defined ordinal types**
           1. **Enumeration:** enum color = {red, blue, green, yellow};
           2. **Subrange:** type Day\_Of\_Month is range 1 .. 31;
     2. **Enumeration Types –** a type in which the user enumerates all of the possible values as symbolic constants
        1. **Advantages:** 
           1. Readability**,** no need to code a symbol as a number
           2. Reliability, operations and ranges can be checked
        2. **Disadvantages**:
           1. Execution time: run time range checking is expensive
           2. Design issues
     3. **Subrange Types** - a subsequence of an ordinal type
        1. Advantages:
           1. Readability: type name is more informative
           2. Reliability: ranges are restricted at run-time
        2. Disadvantages:
           1. Run-time range-checking is expensive
  3. Explain and evaluate array implementation choices, including subscripting, storage order, jagged arrays, and slices.
     1. **Array -** an aggregate of homogeneous data elements, with a common identifier, in which each element is identified by its position
     2. Reference Syntax
        1. Index
           1. Fortran and Ada use parentheses
           2. Most others use brackets
        2. Number of dimensions
           1. Fortran I: 3, the rest have unlimited dimensions
        3. Subscripting Types
           1. Fortran, C, Java – integer only
           2. Pascal, Ada – ordinal types (int, bool, char, enum)
     3. **Jagged Array** – an array with a variable number of elements in any dimension
        + 1. Supported in C, C++, Java
     4. **Array Slices** – substructure of an array
        + 1. Fortran 90 and Ada (ada can slice single dimension arrays only)
     5. **Two Dimensional Array indexing**
        + 1. Fortran : Column major (fill first column first)
          2. Others: row major (fill first row first)
     6. **Array** **Storage Allocation Modes**
        + 1. Static:

**Advantage**: efficient run-time access since the compiler knows array address; no run-time allocation or deallocation costs

* + - * 1. Stack Dynamic

**Advantage**: efficient use of storage since lifetime is limited to procedure activations

* + - * 1. Heap Dynamic

**Advantage**: flexibility to create/delete data structures as needed

* 1. Describe and evaluate mechanisms used for implementing records and unions.
     1. **Records –** a sequential aggregate of heterogeneous data elements in which the individual elements are uniquely identified
        1. **C struct** 
           1. **a**ccess to elements via dot notation
           2. Assignment is allowed
        2. **Cobol** 
           1. Assignment:

Moves spaces to Customer-record

Move 12345 to zip of Address of Customer-Record

* + - * 1. Operations

Add new-purchases to old-balance giving account balance of Customer-Record

* + 1. **Unions** – a set of variables that share the same memory space
       1. Variables are mutually exclusive: only one can be stored at any point in time
       2. **Non-Discriminated unions** 
          1. Types are not checked (Dangerous)
          2. Fortran, C, C+
          3. Types are enforced, by means of a *tag*
       3. **Discriminated Unions**
          1. Programmers cannot create an inconsistent union – the tags cannot be changed by itself
          2. Access is checked at run-time, to be consistent with tag value
          3. Ada
       4. **Advantages:**
          1. An abstract data type can allow different content for different objects
       5. Unions are generally considered unsafe, because they can be used to bypass strong type checking
          1. Ada prevents errors by type discrimination
          2. Java does not support unions
  1. Explain and evaluate pointers vs. reference types.
     1. **Pointers :** a variable that contains the address of another variable
        1. **C, C++ pointers**
           1. Used for dynamic storage management and addressing
           2. Explicit dereferencing (\*) and address-of (&) operator
           3. Type need not be fixed
        2. **Pointer Arithmetic** 
           1. **Implicit scaling**: for pointers, ++ adds the storage size of the pointer type, rather than 1
        3. **Fortran pointers**
           1. Pointers can only point to variables that have been given the target attribute
           2. Dereferencing is implicit (don’t need ‘\*’)
        4. **Perl pointers**
           1. ‘/’ establishes a pointer (which is scalar)
           2. An extra '$' deference’s
           3. Pointer targets are heap dynamic, since all data resides on the heap
           4. Pointers also refer to subprograms, which can be passed as parameters
        5. **Pascal and Ada pointers**
           1. Pascal: for dynamic storage management only; Explicit dereferencing (postfix **^**) p^ = 0
           2. Ada: pointer data type must be create; Dynamic objects are automatically deallocated at end of pointer lifetime
     2. **Reference types –** pointers but no pointer arithmetic
        1. **C++ reference type**
           1. Constant pointer; implicitly dereferenced
           2. Used for parameter passing (reference passed by value)
        2. **Java reference Variable**
           1. No pointer arithmetic
           2. Can point only to objects (all on heap)
           3. NO explicit deallocator (auto garbage collection)
           4. Dereferencing is always implicit
  2. Define dangling pointer and memory leak, and explain their causes and consequences
     1. **Dangling Pointers**: a pointer who points to the former memory location of a deallocated variable
        1. **Solutions:** Locks and Keys, tombstones
     2. **Memory Leak:** A heap dynamic variable that is no longer referenced by any program pointer
        1. No way to access or deallocate it
        2. **Solutions:** Counters, Garbage collection
  3. Explain and apply algorithms for heap management, including tombstones, locks and-keys, reference counters, and garbage collection.
     1. **Tombstones**: extra heap cell that is a pointer to the heap dynamic variable
        1. Actual pointer variable points only at tombstones
        2. After deallocation tombstone remains, but is set to nil to indicate dead end
     2. **Locks and Keys**: pointer values are represented as (key, address) pairs
        1. Heap dynamic variables are represented as (data, lock) pairs
        2. After deallocation, lock is changed so lock and key no longer match
     3. **Reference Counters**: maintains a counter in every HD cell that stores the number of pointers currently pointing at the cell
        1. When count gets to 0, the cell is returned to the free storage pool
     4. **Mark and Sweep Garbage Collection**: Does nothing until all available cells are allocated, then searches for garbage
        1. Every heap cell has an extra bit used by the collection algorithm
        2. Search Algorithm:
           1. Set all GC bits to "garbage" initially
           2. Trace all pointers into the heap, and mark all reachable cells as "not garbage"
           3. Return all "garbage" cells to the list of available cells
     5. **Pointer Evaluation**
        1. Advantages: Pointers or references are necessary for dynamic data structures -- we can't design a useful (modern) language without them
        2. Disadvantages: Dangling pointers and memory leaks are problems
           1. Heap management is expensive