CS430 Programming Languages Exam 1 Study Guide

This exam will include matching, short answer, true/false, and/or code evaluation type questions. No extensive coding problems will be included. Exam questions will be similar to lab and class activity questions and problems.

1. Evolution of Programming Languages (Chapter 2)
   1. Describe the general evolution of programming languages
   2. The major features of successive generations of languages
   3. Explain how imperative languages reflect the Von Neumann architecture
   4. Explain the significant features introduced by major languages such as Fortran, Algol, Cobol, C, and Java
      1. Fortran
         1. Index Registers, floating point hardware
      2. Algol
         1. 1st language with formally defined syntax
         2. 1st machine independent language
         3. Mathematical Notations and good for describing algorithms
      3. Cobol
         1. 1st macro high language
         2. Hierarchical data structures
         3. Nested selection statements
         4. Decimal arithmetic (but not 1st to do it)
      4. PL/I
         1. Scientific and Business programs with one language
      5. Simula 6
         1. Early stages of object oriented coding
         2. Classes, objects and inheritance
      6. SmallTalk
         1. 1st full implementation of object oriented language
            1. Data abstraction
            2. Inheritance
            3. Dynamic type binding
         2. Pioneered GUI still used today
      7. C
         1. Powerful set of operations
         2. Designed for system programming
         3. Initially spread through unix
      8. C ++
         1. Evolved from C
   5. Describe the major programming paradigms (imperative, functional, declarative) and their primary languages
      1. Algorithmic Languages
         1. Fortran, C, C++, Java
      2. Functional Languages
         1. LISP, Scheme
      3. Declarative Languages
         1. Prolog, OPS5
   6. Define important terms and concepts
2. Syntax and Semantics (Chapter 3)
   1. Define syntax, static semantics, and dynamic semantics
      1. Syntax: Form of expressions; the set of rules governing the form of programs
      2. Semantics: Meaning of expressions
         1. Static Semantics: Set of rules that describe elements of the form:
            1. Outside the scope of grammars such as BNF but are still checked by the compiler
            2. EX. Data type rules and naming rules
         2. Dynamic Semantics: The set of rules governing the programs behavior
      3. Backu-Naur Form (BNF)
         1. Standard form for defining programming syntax
         2. Meta Language
         3. John Backus and Peter Naur in 1960
   2. Define non-terminal, terminal, derivation, and parse tree
      1. Non-terminal: Is the left hand side of a BNF equation. The part before the arrow.
         1. “<identifier>”. Identifies a specific grammar of a language
      2. Terminal: Can be any variable in the language. A variable a or an operator +
      3. Derivation: A series of transformations leading to a sentence (when no non-terminals left)
         1. Each transformation
      4. Parse Tree: a tree representation of a derivation
   3. Define ambiguity in a grammar, and modify a grammar to remove ambiguity
      * 1. Ambiguity: An ambiguous grammar allows multiple derivations for some sentence
           1. 2 different parse trees return the same result
           2. Can be eliminated by rewriting rules
   4. Define associativity in a grammar, and change associativity within a grammar
      1. Associativity: Determines the order of association when multiple operations are present
         1. EX. a+ b + c could equal (a+b) + c or a+ (b+ c)
         2. Can have left or right associativity
            1. Whichever part of a parse tree is the lowest is the association.

Ex. a + b + c, if a is the lowest branch of the tree then its left associative

If c is the lowest branch its right associative

* + - * 1. Can also be changed by rewriting grammar rules
  1. Define precedence in a grammar, and change precedence within a grammar
     1. Operation Precedence: Ca be established by assigning tokens to precedence levels
        1. In a situation where grammar has rules 2 rules for the same <identifier>, the one on top has higher precedence
  2. Be able to:
     1. Interpret and write BNF descriptions of languages
     2. Create derivations and parse trees for sentences of a given grammar
     3. Create left-most and right-most derivations
     4. Rewrite rules to change ambiguity, associativity, and precedence

1. Chapter 4: Parsing (Chapter 4)
   1. Define syntax analysis, lexical analysis and parsing, including the purpose and product of each
      1. Syntax Analysis:
         1. Goals: to check correctness and create a parse tree for translation
         2. Usually based on a formal description such as BNF
      2. Syntax Analysis Components:
         1. Lexical Analyzer
            1. Identify atomic elements (identifiers, operators, reserved words ect)
         2. Parser
            1. Detect syntax errors
            2. Generate Parse tree
      3. Lexical Analysis:
         1. “front end" for the parser
         2. Goal:
            1. Identify lexemes and tokens
         3. Called by the parser when it needs the next token
         4. Grabs lexemes from input stream and builds a symbol table
         5. 2 Methods:
            1. Formal grammar

Automatically interpreted by lexical analysis tools

* + - * 1. State Diagram

Recognizes identifiers and integer literals

* + 1. Parser
       1. Find all syntax errors in program; for each error produce an appropriate message and recover
       2. Produce a parse tree for the program to be used by the translator
       3. Uses lexemes and symbol table from lexical analyzer to check syntax and build parse tree
       4. 2 categories:
          1. Top Down

Produce parse tree form root down

Creates left most derivation

* + - * 1. Bottom Up

Produces parse tree beginning at the leaves

Creates right most derivation

* 1. Define token and lexeme, and relate them to lexical analysis
     1. Lexemes: pieces of the source program
     2. Tokens: Categories to which lexemes belong
     3. Ex.
        1. int is a lexeme and type is its token
        2. + and = are lexemes and operator is its token
     4. Lexical Analyzer must take the lexemes and tokes of the formal grammar and build a symbol table with them so the parse tree can then check that the syntax is right and build its parse tree

1. Scripting Languages
   1. Explain the characteristics of scripting languages that differentiate them from other general-purpose languages
      1. Used for shorter and hoc programming languages
      2. Simple relaxed syntax (less coding)
      3. Powerful operators
      4. Interpreted execution (executed line by line)
      5. Dynamic data type binding
   2. Name several scripting languages
      1. Perl, python, javaScript, bash, awk
   3. Explain the properties of the Perl language
      1. Data Types
         1. Scalar - $count
         2. Array - @scores
         3. Hashtable - %dictionary
         4. String – ‘abc’ or “abc”. (need “ “ to interpret a new line)
      2. Dynamically typed
         1. Can change between stings and numbers easily, no variable types
      3. Scoping
         1. my $x is a local variable
         2. $y is a global variable
      4. Arrays
         1. @ before the variable
         2. Need to reference each element of the array as a scalar $
         3. Dynamically sized
         4. $# is the index of the leas element in the array
      5. Hashes
         1. Need % in front of the variable
         2. Need to reference a single element as a scalar $
      6. Functions (subroutines)
         1. @\_ is the argument array
   4. Be able to:
      1. Interpret Perl expressions
2. Functional Languages (Chapter 15)
   1. Describe the formal basis for functional languages
   2. Explain the differences between functional and imperative languages related to design objectives, computational model, and implementation
   3. Evaluate basic Dr.Racket expressions