**CS 471 Midterm Study Guide**

1. **Lab 1**

Simple C Aliasing Problem – Printing our name with ASCII integer values, printing the location of the array, location of the array pointer, moving the array to another segment by making it static, little endian = Intel big endian = Motorola, the last byte can be set to 0 and still work.

1. **Chapter 1 – criteria, or domains, or Language Categories**

Programming Domains

* Scientific Applications
  + Large numbers of floating point computations; use of arrays
  + Fortran
* Business Applications
  + Produce reports, use decimal numbers and characters
  + COBOL
* Artificial Intelligence
  + Symbols rather than numbers manipulated; use of linked lists
  + LISP
* Systems Programming
  + Need efficiency because of continuous use
  + C
* Web Software
  + Eclectic collection of languages: markup (HTML), scripting (PHP), general-purpose (Java)

Language Evaluation Criteria

* Readability: the ease with which programs can be read and understood
  + Overall simplicity
    - A manageable set of features and constructs
    - Minimal feature multiplicity
    - Minimal operator overloading
  + Orthogonality
    - A relatively small set of primitive constructs can be combined in a relatively small number of ways
    - Every possible combination is legal
  + Data types
    - Adequate predefined data types
  + Syntax considerations
    - Identifier forms: flexible composition
    - Special words and methods of forming compound statements
    - Form and meaning: self-descriptive constructs, meaningful keywords
* Writability: the ease with which a language can be used to create programs
  + Simplicity and orthogonality
    - Few constructs, a small number of primitives, a small set of rules for combining them
  + Support for abstraction
    - The ability to define and use complex structures or operations in ways that allow details to be ignored
  + Expressivity
    - A set of relatively convenient ways of specifying operations
    - Strength and number of operators and pre-defined functions
* Reliability: conformance to specifications (i.e., performs to its specifications)
  + Type checking
    - Testing for type errors
  + Exception handling
    - Intercept run-time errors and take corrective measures
  + Aliasing
    - Presence of two or more distinct referencing methods for the same memory location
  + Readability and writability
    - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability
* Cost: the ultimate total cost
  + Training programmers to use the language
  + Writing programs (closeness to particular applications)
  + Compiling programs
  + Executing programs
  + Language implementation system: availability of free compilers
  + Reliability: poor reliability leads to high costs
  + Maintaining programs
* Portability
  + The ease with which programs can be moved from one implementation to another
* Generality
  + The applicability to a wide range of applications
* Well-definedness
  + The completeness and precision of the language’s official definition

Language Categories

* Imperative
  + Central features are variables, assignment statements, and iteration
  + Include languages that support object-oriented programming
  + Include scripting languages
  + Include the visual languages
  + Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
* Functional
  + Main means of making computations is by applying functions to given parameters
  + Examples: LISP, Scheme, ML, F#
* Logic
  + Rule-based (rules are specified in no particular order)
  + Example: Prolog
* Markup/programming Hybrid
  + Markup languages extended to support some programming
  + Examples: JSTL, XSLT

1. **History of early languages**
2. **History of early languages**

* Fortran
  + Fortran I: 1957
    - Designed for the IBM 704 which had index registers and floating point hardware.
    - Led to the idea of compiled programming languages.
    - First implemented version of Fortran.
    - Names could have up to six characters
    - Post-test counting loop (DO)
    - Formatted I/O
    - User-defined subprograms
    - Three-way selection statement (arithmetic IF)
    - No data typing statements
    - No separate compilation
    - Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of 704
    - Code was very fast
    - Quickly became widely used
  + Fortran II: 1958
    - Independent compilation
    - Fixed the bugs
  + Fortran IV: 1960-1962
    - Explicit type declarations
    - Logical selection statement
    - Subprogram names could be parameters
    - ANSI standard in 1966
  + Fortran 77: 1978
    - Character string handling
    - Logical loop control statement
    - If-then-else statement
  + Fortran 90
    - Modules
    - Dynamic arrays
    - Pointers
    - Recursion
    - Case statement
    - Parameter type checking
  + Fortran 95
    - Relatively minor additions, plus some deletions
  + Fortran 2003
    - Support for OOP, procedure, pointers, interoperability with C
  + Fortran 2008
    - Blocks for local scopes, co-arrays, do concurrent
  + Fortran evaluation
    - Highly optimizing compilers (all versions before 90) – types and storage of all variables are fixed before runtime
    - Dramatically changed forever the way computers are used

Lisp

* LISt Processing language
  + Designed at MIT by McCarthy
* AI research needed a language to
  + Process data in lists (rather than arrays)
  + Symbolic computation (rather than numeric)
* Only two data types: atoms and lists
* Syntax is based on lambda calculus
* Lisp evaluation
  + Pioneered functional programming
  + No need for variables or assignment
  + Control via recursion and conditional expressions
  + Still the dominant language for AI
  + Common Lisp and Scheme are contemporary dialects of Lisp
  + ML, Haskell, and F# are also functional programming languages, but use very different syntax

Scheme

* Developed at MIT in mid 1970s
* Small
* Extensive use of static scoping
* Functions as first-class entities
* Simple syntax (and small size) make it ideal for educational applications

Common Lisp

* An effort to combine features of several dialects of Lisp into a single language
* Large, complex, used in industry for some large applications

ALGOL

* ALGOL 60 was the result of efforts to design a universal language
* Goals of the language
  + Close to mathematical notation
  + Good for describing algorithms
  + Must be translatable to machine code
  + It was the standard way to publish algorithms for over 20 years
  + First machine-independent language
  + First language whose syntax was formally defined BNF

COBOL

* Design goals
  + Must look like simple English
  + Must be easy to use, even if that means it will be less powerful
  + Must broaden the base of computer users
  + Must not be biased by current compiler problems
  + First language required by DoD

1. **Compilation and Interpreter Environments**

* Compilation translates high-level programs into machine code. Slow translation but a fast execution. Compilation process has several phases; lexical analysis, syntax analysis, semantic analysis, and code generation
* Interpretation has no translation, slower execution, often requires more space, rare for traditional high-level cases,

1. **Operational Semantics**

Consider the language construct: *Repeat STMT until expr.* Without using if-then-else statements (you can use GOTO) write an operational semantic equivalence.

L1:

STMT

t = expr

JEQ t 0 L1 //continue loop while zero flag is 0 (t not equal to expr)

Do STMT while (expr)

L1:

STMT

T = expr

JNE t 0 L1 //loop while zero flag is not 0

*If (expr) STMT1 else STMT2*

t = expr

JEQ t 0 L1 //else – if expr is false jump to L1: STMT2

STMT1 //perform STMT1 if true

JMP L2 //skip over L1:

L1: STMT2

L2: NOP

*For (expr1; expr2; expr3) STMT*

EXPR1 //set expr1 (i.e. int i = 0)

L1:

T = expr2 //set t = expr2 (i.e. i < 5)

JEQ t 0 L2 //jump out if zero flag is set (counter reaches 5)

STMT

Expr3 //perform expr3 (i.e. i++)

JMP L1 //loop back to L1

L2: NOP

1. **Given BNF, give a parse derivation for a sentence**

S -> A a

A -> bA | cA | B

B -> d | e

bcda

S -> Aa -> bAa -> bcAa -> bcBa -> bcda

1. **Dynamic Binding**

Sub1() int x, y;

Sub2() int a, b, y;

Sub3() int b, c;

Sub1() calls sub3 () calls sub2(). What variables are visible to sub2 using sub*i.v* as notation.

Work backwards through the calls, no duplicate variables from previous calls

Sub2.a, sub2.b, sub2.y, sub3.c, sub1.x

Sub1() calls sub2() calls sub3() Visible to sub3?

Sub3.b, sub3.c, sub2.a, sub2.y, sub1.x

1. **Coding problem**
2. **2 of implicit/explicit, stack/heap/data, narrow/widen**

* An explicit declaration is a program statement used for declaring the types of variables (you just declare a variable)
* An implicit declaration is a default mechanism for specifying types of variables through default conventions, rather than declaration statements (you assign it a value)
* Narrowing – converting one type to another with a smaller range
* Widening – converting to a type that at minimum encompasses the old type’s range
* Heap – malloc, i.e. New String()
* Stack – basic arrays
* Data – static and global variables

1. **2 of weakest Precondition (axiomatic semantics)**

if (x > 5)

x = x +7

else

x = x – 3

{x > 100 .or. x < -100}

Break this into 2 parts

Positive {x > 5}

{x > 93}

{x > 93 or x < -107} AND {x > 5}

{x + 7 > 100 or x + 7 < -100}

X = x + 7

{x > 100 or X < -100}

Negative

{x < -97}

{x > 103 or x < -97} AND {x <= 5}

{x - 3 > 100 or x - 3 < -100}

X = x – 3

{x > 100 or x < - 100}

Combination (positive OR negative)

{x > 93 or x < -97}

1. **Variables**

* Variables are characterized by attributes
  + To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility
* A variable is an abstraction of a memory cell
* Variables can be characterized as a sextuple of attributes:
  + Name – not all variables have them
  + Address – the memory address with which it is associated, can have different addresses at different times and places, aliases hurt readability
  + Type – determines the range of values of variables and the set of operations that are defined for values of that type
  + Value – the contents of the location with which the variable is associated, the i-value is its address, r-value is its value
  + Abstract memory cell – the physical cell or collection of cells associated with a variable

1. **LHS/RHS**

SP+index+offset

1. **LR(), LALR(), LL() differences, implementation, and uses. Comparison of Context Free vs LR/LALR/LL**

LL – Left to Right, Left most parse, recursive descent parsing

LR – Left to Right, Right most parse, table driven parsing

LALR – Look ahead Left to Right

Context Free Grammar

LL() = top down, recursive descent parsing

LR() = bottom up, table driven

LL grammar cannot have: left recursion, common left factors

CFG(s) = O(n^3)

LALR(s) = O(n)

LR(s) = O(n)

LR1(s) = O(n)