Course wrap-up

- Considered several paradigms (functional, imperative, OO)
- Considered pure vs hybrid languages w.r.t. paradigms, and associated advantages/drawbacks
- Considered various specific language features and capabilities, with their associated advantages/drawbacks
- Considered possible/common implementations of different features

Understanding implementations

- Understanding how different features are implemented lets us understand how to use them most effectively, avoiding key pitfalls and taking full advantage of benefits
- Also allows us to see how we might achieve similar benefits when using languages that don’t directly support a desired feature/capability
- Also just plain interesting seeing how on earth some of this stuff can be made to work, creativity at low level to make high level abstractions possible

Life long learning and comp sci

- This field changes rapidly, all the time
- To stay relevant, you'll be constantly learning new languages, new features, possibly new paradigms
- (my opinion) being curious about how things work will make for a much more rewarding (and much less frustrating) career, being willing to experiment with and explore language features and implementations helps with that every bit as much as reading/google-searches

Lisp recap

- assuming fluency with basic syntax
- types and type checking
- sequence type heirarchy (sequences, lists, arrays, etc)
- list implementation and implications
- forms of equality testing
- lexical vs dynamic scoping (let, special, defvar, setf, etc)
- tail recursion (accumulators, equivalence to loops, optimization of stack use, etc)

Lisp recap continued

- parameter passing mechanisms (&optional, &key, &rest)
- hash tables, structures, symbols and property lists
- higher order functions (funcall, apply, eval, map, etc)
- lambda functions, let over lambda (and labels)
- homoiconic languages, macros, parsing lisp in lisp
- pure FP vs lisp, making some lisp features more pure
- underlying C implementation of lisp, implications
- misc lisp: pipes/file io, packages, compilation, gotos etc

Grammars and languages

- formal definitions often based on three tiers of grammar
  - regular expressions for tokenizing/scanning
  - context free grammars for parsing
- augmented grammars for context sensitive rules/meaning
- automation of tokenizing/parsing:
  - use of lex for tokenizing
  - use of yacc grammar rules for parsing
  - use of C segments to augment yacc's grammar rules
  - assuming basic fluency in lex/yacc

Context free grammars/parsing

- (assuming regular expressions are largely review)
- CFG use of tokens and nonterminals
- CFG rules of form
  - nonterminal --> sequence of tokens and nonterminals
- derivations
  - step by step
  - from start nonterminal to desired token sequence
  - leftmost and rightmost derivations

Parse trees

- from start nonterminal (tree root) to tokens (leaves)
- inorder traversal => leaves (tokens) in desired sequence
- relationship between ambiguity of grammar and existence of multiple parse trees for token sequences
- tree reveals grammar's order of evaluation
  - lower precedence operations wind up higher in tree
  - associativity also reflected

Language attributes and binding

- static vs dynamic binding
- fixed dynamic vs truly dynamic
- names/symbols
- types
- storage
- scope
- lifetime

Types and type checking

- static vs dynamic typing
- automatic vs programmer-controlled runtime checking
- explicit vs implicit conversions
- widening vs narrowing conversions
- name vs structural type compatibility

Primitive types

- common types, operations, implementations
- characters
- integers
- real numbers
- booleans
- ordinals
- rational, complex, ...?
- fixed size vs variable size: pros/cons

Composite types

- common types, operations, implementations
- strings
- arrays (1-D, multi-D)
- hash tables
- records/structures
- unions
- pointers

Operators/operations

- unary, binary, ternary
- associativity
- precedence
- infix, prefix, postfix
- operator overloading
- side effects
- short circuiting
- parameter evaluation (w.r.t. order of evaluation)

Implementation of control

- underlying test and branch operations
- emulation with limited language set (e.g. if/goto + labels)
- possible structuring of different loop types
- possible structuring of switch/case styles

Control structures:

- blocks
  - single/multiple entry/exit points
  - scoping rules, storage/initialization rules, return values
- selection
  - single-way, two-way, multi-way, switch/case
- iteration/loops
  - top/bottom tested
  - for, while, do-while, repeat-until, foreach
  - break, continue, last/final
  - block-related issues (scoping, entry/exit points, etc)

Subroutines

- nested declarations
- parameter passing mechanisms
- call/return mechanisms
- scoping rules
- overloading
- lexical/dynamic scoping
- higher order functions
- generic/templated functions

Macros

- different forms of preprocessing in languages
- degree of syntax variations they open for programmer
- complications they add to debugging
- lisp-style macros
- C-style #defines
- C++-style templates

Higher order functions

- degree of support in language
- lisp-style
- C-style using function pointers
- C++ expansion with templates

Variadic functions

- degree of support in language
- lisp style (&rest)
- C style (stdarg macros)
- C++ style (recursive templated functions)

Dynamic/nested scopes

- possible implementation approaches
- dynamic scope
  - pass entire environment description to called function
  - use stacks for each individual name
  - give called function access to stack frame from caller, with information on which variables stored/where
- nested scope
  - giving nested function access to its lexical parent's stack frame

Dynamic memory management

- management of heap space
- maintaining data on free/allocated memory items
- responding to free/allocate requests
- memory leaks, garbage collection
- fragmentation issues/handling
- information to track in free/allocated chunks
- gnu-y algorithms/approaches

Smart pointers

- managing the various pointer problems (wild, null, dangling pointer access, memory leaks)
- reference counts/tombstones
- C++ approach (shared_ptr, weak_ptr)

ADTs and OO

- language support vs enforcement
- OO: pure vs hybrid
- ADTs + inheritance + dynamic dispatch
- issues of multiple inheritance
- implementation issues (storage and method calls):
  - C++ examples (this pointer, vtables)