CS 333 Programming Languages, Spring 2018

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

This course is a survey of programming languages and paradigms. We will focus on the design of programming languages and compare and contrast different language families including imperative, object-oriented, functional, and logic paradigms. Topics include syntax, context-free grammars, parsing, semantics, abstract representations of programming processes and structures, memory management, and exceptions. Students will undertake small programming projects in various languages and more extensive projects in two languages of their choice. Students will present the characteristics of their chosen languages to their peers at the end of the term.

Prerequisites: CS 231 or permission of instructor.

Desired Course Outcomes

A. Students demonstrate an understanding of different language paradigms and implement algorithms in each paradigm.

B. Students demonstrate an ability to independently learn programming languages.

C. Students demonstrate an ability to describe the syntax, semantics and functionality of different languages in a common, rigorous manner.

D. Students demonstrate an understanding of the relationship between language and design.

E. Students work with partners to learn one or more languages and present them to the class.

F. Students present algorithms, languages, and their characteristics in an organized and competently written manner.

1 Programming Languages Overview

What is your favorite language, what do you want to learn?

Administrivia, first project, first homework

In the beginning...

- Gear-based computers (Babbage, Lovelace)
- Cable-based computers (ENIAC)
- Punch cards (IBM)

Assembly language: mnemonics to represent the binary machine code, memory locations, registers

- 2-pass assembler allows the programmer to avoid using memory locations explicitly

Grace Hopper was a pioneer in the development of programming languages

- She worked on the Mark I computer at Harvard
- She worked for the Eckert-Mauchly Computer Corporation (ENIAC designers)
- She helped develop the UNIVAC I computer (first computer to be on the market in 1950)
- She created a compiler at E-M: she published the first compiler in 1952 (after 3 years of being told it couldn't be done)
- She led the release of the first compiled languages and consulted on COBOL
- She developed some of the first validation code and standards for both COBOL and fortran

What did Grace Hopper enable us to do? What were the technical obstacles she had to overcome?

- Syntax: a programming language must be describable in an unambiguous manner
- Names and types: entities must be named, the types and scope of entities must be clear
- Semantics: the meaning of a program must be defined regardless of the underlying architecture
- Compiler/Interpreter: all of these rules must be encapsulated in an interpreter or compiler

Why do we develop programming languages? What are some examples?

- Fortran: scientific computing
- COBOL: business programming
- Algol: general purpose programming
- LISP: AI programming (list processing)
- C: systems programming / general purpose
- Prolog: natural language processing / logic problems
- SEQUEL: database management
- HTML/CSS: layout design

2 Design Goals and Examples

Discussion: design goals for programming languages?

- Simplicity and readability
- Clarity about binding: language definition, implementation, code writing, compile, load, run
- Reliability: across time, across platforms; exception handling; memory management
- Support
- Abstraction: data and procedural abstraction, libraries
- Orthogonality: e.g. what can be passed as an argument to a function?
- Efficient implementation

Interpreted v. Compiled

Examples of languages for various design goals

Programming languages in context

- Chip layout
- Analog circuit design
- CPU modeling
- Hardware components
- Operating System
- User interface
- Compilers
- Applications: GUIs, music, video, motion, models, neural network design
- Theory

C/Project Overview

- Preprocessor commands
- Standard form for a C program: includes, main function, argc/argv
- Simple compile command

3 C Overview and PL Concepts

C/Project Overview

- review: standard form for a C program: includes, main function, argc/argv
- memory model for C
- variables, pointers, arrays, addresses
- the stack
- memory allocation: malloc and free
- how to interpret & (address of) and * (dereference this value)

4 PL Concepts and Grammars

Programming Language Concepts

- Syntax: structure of the language; lexical analyzer (tokens) v. program structure
- Names: rules for naming, what can we name?
- Scope: where an entity is available
- Visibility: which entity is referenced by a name
- Binding: when the connection between name and entity occurs
- Types: what types are available?
- Semantics: meaning in terms of impact on machine state
- Organization: how do we build abstractions?
- Memory management: how does the language handle memory?

Syntax: a precise description of all its grammatically correct programs

Hierarchy of grammars

- regular: very restrictive grammars, corresponds to a finite-state automata, non-recursive
- context-free: a push-down automata, allows recursive definitions
- context-sensitive: undecidable, left side can be multiple symbols, right side has to be no less than left.
- unrestricted: no restrictions

Most programming languages are a subset of context-free grammars that can be parsed in a single pass.

- LL(k) grammars can be parsed by looking ahead by k symbols.
- LL(k) grammars cannot contain left recursion.
- Most compilers (e.g. gcc) recursive descent LL(1) parsers.

Context Free Grammars

- Terminal symbols: keywords, legal characters
- Non-terminal symbols: symbols for abstract concepts
- Production rules: define relationships between non-terminal and terminal symbols
- A start symbol

Backus-Naur Form: a context-free grammar variation

Integer and Float examples: don't want to define all possible names or numbers explicitly

Extended BNF

- Curly brackets are zero or more instances of what is inside
- Parentheses indicate one of the options must be chosen

5 Grammars and Regular Expressions

Example: a grammar for scientific notation

Ambiguous grammars

- Expression example using the Op non-terminal, need additional rules to specify precedence
- Dangling else example: need additional rules to specify while if the else belongs to.
- C: else is matched with the textually nearest if
- Java: if statement has to have an else if it is the single statement after the if

Lexical Syntax

Lexical analysis can take up to 75% of compilation time

Tokens: identifiers, literals, keywords, operators, punctuation

A lexical analyzer takes a text file and converts it into tokens, handles most terminal rules

Regular Expressions

Special characters

- [ ] - used to specify a set of alternatives
- \- used as an escape character to permit use of the other special characters
- ^ - negates an expression when inside brackets, permits you to specify strings that don't include a certain expression, is the start of the string otherwise
- $ - the end of the string
- . - matches almost any character except line breaks
- — - specifies an or between expressions
- ? - makes an expression lazy (first possible completion) instead of greedy (largest possible completion)
- * - tells the engine to match the prior expression zero or more times
- + - concatenation (and)
- ( ) - groups tokens
- { } - can be used to specify ranges for repetition.

6 PL Concepts and Grammars

Do some examples of regular expressions with egrep. Use testbaba.txt and htmltest.htm.

Flex: sections divided by %%

- Definitions: macros and global C code
- Rules: a regular expression and C code that should do something if matched
- User Code: generally the main function for the lexical processor.

Do the bab examples in flex

CLite Grammar

Introduce CLite grammar: pp 16 of the lecture notes

Assignment: a = b + 3 * c; // pp 22 of lecture notes

7 Abstract Syntax, Names

Maybe review concrete syntax examples

Assignment: a = b + (c - d);

Assignment: a = b + c - d; // look at the different parse trees

Abstract Syntax

- Simplified version of the parse tree, collapses parts of the concrete syntax parse tree
- Precedence handled by the concrete syntax, built into the abstract syntax
- Abstract grammar for CLite: pp 20 of lecture notes
- Non-terminals are more directly connected to their semantic meaning
- Organization of the grammar explicitly links inputs and outputs of functional operations

Go back to example on pp 22: a = b + 3 *c;

Example on pp 21 of lecture notes: drawing abstract syntax as boxes with fields for key information.

```
if( a < 0) {
    a = -a + 3*b;
}

Statement
    Conditional:
        Expression
            Binary
                Operator: RelationOp: <
                Expression: VariableRef; Variable; String a
                Expression: Value; IntValue; Integer 0
        Statement
            Assignment
                VariableRef; Variable; String a
            Expression
                Binary
                    Operator; ArithmeticOp; +
                    Expression
                        Unary
                           UnaryOp; -
                           Expression: VariableRef; Variable; String a
                    Expression
                        Binary
                            Operator; ArithmeticOp; *
                            Expression; Value; IntValue; Integer 3
                            Expression; VariableRef; Variable; String b
        Statement; NULL
```

Converting a concrete parse tree to an abstract syntax tree.

- Discard all separator or terminating symbols or tokens
- Discard all non-terminals that are trivial roots. A trivial root is a symbol with only one subtree (child).
- Replace all remaining non-terminals with the operators which are a leaf or one of their immediate subtrees.

Concrete Syntax examples

Abstract Syntax examples

8 Quiz

HW Review and Quiz

9 Names

Naming is about binding meanings to symbols: keywords, identifiers

Most languages don't allow redefinition of keywords. Python does, COBOL doesn't

- Concept is pre-defined identifiers versus reserved keywords

Naming syntax

- Case-insensitive: Pascal, Fortran, Ada, VHDL
- Case-sensitive: C, Python, Java
- Mixed case sensitivity: PHP
- Bounded length: Fortran (6 characters through F77, 31 in F90)
- Special characters: C, Python, Java (underscore), Cobol (dash)
- languages do not allow names to start with a digit (why?)

Identifiers can indicate more than bindings: e.g. in Fortran first letter defines default type.

- This is the reason i is used as a loop variable so often.

Variables bind a name to a memory location. Variables have a number of properties

- Type
- Value
- Address
- Scope
- Lifetime

The first four properties can be static or dynamic

- Static binding: takes place before run-time
- Dynamic binding: takes place during run time

l-value versus r-value: examples in Python, Java, C

- l-value is the variable's address, short for left-hand side value
- r-value is the variable's value, what is stored at the address, short for the right-hand side value

Scope: the set of statements that can access a specific name binding. *nested* v. *disjoint*

- Python example: scope of a for loop variable
- C example: scope of a for loop variable
- Java example: scope of a for loop variable, note semantic v. syntactic nesting

Visibility and name clashing; specifying scope in Java (this), Python (global), Ada (named blocks)

10 Names and Symbol Tables

No-forward reference rule: how is it relaxed in Java, Python, get around it in C

Lifetime examples

- Fortran, COBOL: lifetime of program
- C: variable, look at extern, globals, static locals, static globals
- Java: static changes lifetime properties
- Python: class globals have a different lifetime than objects or object fields
- Most current languages link lifetime and scope

Symbol tables

- representing them as name/value pairs, possibly name/type/value pairs
- symbol tables are maintained by either the compiler or the interpreter
- When enter a new scope, push a new, empty dictionary on the symbol table stack
- When existing a scope, pop
- When a name is bound to a value, insert the pair into the current dictionary
- Given a name, look in the current dictionary, repeat with enclosing dictionaries according to the rules

11 Names and Symbol Tables

Overloading names and operators

- variables: shadowing, mechanisms for specifying, Java example, Python example (global), ADA
- functions: concept of a signature, does it include the return type? Python doesn't allow overloading names
- functions: ADA is the only language to allow differentiation based on return type
- functions: difficulty one: which version to use if return value is not assigned
- functions: difficulty two: when the l-value receiving the return value is not the same, which version do you use?
- operators: mechanisms for defining new meanings. C++ and Java allow overloading for class types
- operators: C doesn't allow overloading,

Java allows functions and variables to share a name, because a function is not a basic type (you can't create a variable of type function). Python and C don't allow this, because a variable can hold a reference to a function.

Types: static typing v. dynamic typing (actual typing versus duck typing)

Basic types

- What are the basic types?
- is String a basic type? Compare Java, Python, C
- are int and float defined the same across all languages? Show the table

12 Types

Implicit conversion

- C generally implicitly converts to the longer type
- Java defines 11 categories of conversion (see notes)
- Java defines five contexts in which the compiler can make implicit conversions
- Some languages do not allow implicit conversions (e.g. Ada, VHDL)

Pointers and Arrays as types

- All languages have pointers (or references). Not all languages have explicit operators for them.
- Arrays in typed v. untyped languages
- Arrays that store information about themselves: dope vector
- Pointer arithmetic in C

13 Types

Pointers and Arrays as types

- Accessing arrays: what is the first element? Java, C, Python it's 0, Matlab and Fortran it's 1. (note Fortran to C conversion trick)
- Storage order of multi-D arrays: Fortran is the outlier and stores them column-major

Strings

- C strings v. String types in Java, Python where arrays keep track of their length
- Haskell treats strings as a char array

Structures

- User-defined collection of data
- In most languages with structures, the dot operator specifies a field of a structure.
- The semantic meaning of the dot operator is different across languages, dereference or not?

Functions

- Functions can be treated as a proper type: C, Python, Lisp
- Some languages allow functions to be manufactured in code: lambda functions, and eval()
- Some languages do not allow functions as a type: Java

Variants

- A variant is a structure that can be interpreted in multiple ways: union in C
- Variants are one method of polymorphism

14 Polymorphism

Polymorphism: allowing the same name to attach to different types

- Python, PHP are examples of languages that have polymorphic variables
- C has the void * that can reference any type
- OOP languages use inheritance, interfaces, or abstract classes
- C++ and Java have generics/templates

Quiz

15 Type Checking and Semantics

Writing?

- What are good principles for tutorial writing?
- Know your audience
- Define your terms, given your audience

Type Checking

- Rule 1: All referenced variables must be defined
- Rule 2: All declared variables must have unique names
- Rule 3: A program is valid if its declarations are valid and all statements are valid with respect to the type map
- Rule 4: Statements are valid if their individual expressions are valid
- Rule 5: An expression is valid if its subclasses are valid
- Rule 6: An expression's type follows a type rule system (e.g. widening/narrowing conversions)

The abstract syntax tree is our guide for type checking a program.

Semantics

- Operational semantics: what does the program the compiler created do?
- Axiomatic semantics: a formal specification of each statement as an axiom
- Denotational semantics: representing the abstract syntax as a mathematical transformation of the computer's state

Expressions and operator precedence

- abstract syntax trees define the order of operations and types
- pre- and post-increment operators can have different behaviors depending on the tree traversal
  - Option 1 Pre-: execute pre-increment operators prior to all other accesses, picking an order for tree traversal
  - Option 2 Pre-: execute pre-increment operators when the node is reached during traversal (picking an order)
  - Option 1 Post-: execute all post-increment operators after the expression has been calculated and assigned
  - Option 2 Post-: execute all post-increment operators when the node is reached during tree traversal
  - Option 3 Post-: execute post-increment operators after the value has been computed, but before assignment

Because C does not define which option to use, we have to use operational semantics to determine the meaning of a statement with pre/post operators.

16 Semantics

Lazy evaluation in if-statements

- Don't both doing more if you know the final value
- Reformulate the expression as nested ifs
- Example: if a ptr is not null then check its value in the expression
- Haskell has a more extensive concept of lazy evaluation

Program State and Denotational Semantics

- program state is all variables with active bindings and their value
- arguably includes the program counter, stack pointer, flags, and other registers
- each token in an abstract syntax is a transformation on the program's state

$$\begin{aligned} M : Program &\rightarrow State \\ M : Statement \times State &\rightarrow State \\ M : Expression \times State &\rightarrow Value \end{aligned} \tag{1}$$

We can express every program, statement, or expression as a function that returns either a value or a state

Program initializes the state of the computer. Initial state functions can be different.

$$State\ MProgram(\text{Program.body}, InitialState(\text{Program.decpart})) \tag{2}$$

Statements

- Skip: identity transformation
- Assignment: statement or expression? Puts a value in a target location.
  - C: can use an assignment in an if statement, can do multi-assignment, can use as an r-value.
  - Copy semantics versus reference semantics, think about a language with both
  - assignments in many languages can have side effects (anything that allows aliasing)
  - Put function implements *overriding union*
- Conditional: a test expression and two statements, the test determines which statement applies
  - A conditional can have side-effects as well in the expression
  - Show how to reformulate the function to avoid temporary variables

17 Semantics

Quick review: Skip, Assignment, Side-effects Statements

- Skip: identity transformation
- Assignment: statement or expression? Puts a value in a target location.
  - C: can use an assignment in an if statement, can do multi-assignment, can use as an r-value.
  - Copy semantics versus reference semantics, think about a language with both
  - assignments in many languages can have side effects (anything that allows aliasing)
  - Show how to reformulate to avoid temporary variables
  - Put function implements *overriding union*
- Conditional: a test expression and two statements, the test determines which statement applies
  - A conditional can have side-effects as well in the expression
  - Show how to reformulate the function to avoid temporary variables
- Looping: while loop is all that is necessary, all other loops can be created with additional assignments
  - Show version with and without side-effects
  - Looping is implemented functionally with recursion
  - For loop with side effects requires three functions
  - How do we encode scope in semantics? Use push/pop to maintain a stack of states
- Block
  - A series of statements for sequential semantics is implementable using recursion
    - If a block defines a new scope context, add an encompassing function that uses push and pop
  - Not all languages use sequential semantics
  - VHDL uses concurrent semantics: all statements execute simultaneously in parallel

18 Semantics

Block

- A series of statements for sequential semantics is implementable using recursion
  - If a block defines a new scope context, add an encompassing function that uses push and pop
  - Go back to the for loop and add push/pop in the top level function of the three functions.
  - Note how semantics with respect to scope is different in different languages. Some for loops create a new scope (C++, Java), some do not (PHP, Python)
- Not all languages use sequential semantics
- VHDL uses concurrent semantics: all statements execute simultaneously in parallel

I/O semantics

- Some languages have I/O built into the language (PHP, Prolog, Python)
- Some languages use a built-in library of functions (C, Fortran, Lisp)
- Some languages use standard classes to handle I/O (Java, C++, Python)

I/O as a set of statements: assign something to a buffer

- with no side-effects, the print statement is equivalent to a skip in terms of its effect on program state
- print buffers are temporary and have no effect on future program functionality
- side-effects will generally be due to expressions within the print statement

File buffers affect data stored long term and could affect future program functionality

- Most languages permit a program to peek at an input buffer before reading it
- Some languages permit a program to read the contents of a buffer just written
- Most languages permit a program to adjust the current buffer pointer (location within the buffer)
- Most languages permit a program to read status flags.

Most I/O semantics can be represented as assignments to variables.

One challenge in file I/O is handling unknown input, input errors, and output errors. If errors occur, the meaning of a program may be undefined.

Next up: exception semantics

19 Exceptions

Exception Semantics attempt to define behavior when errors occur

- C and Fortran have no built-in error handling, C uses an operating system interface; C functions return status
- Ada, C++, Java, Python all have built-in error handling
- The issue with errors is that the scope in which they occur often can't handle the error
- Most error handling is about communication

Error communication strategies: return values, flag variables, exception handlers, throwing exceptions

Show examples in C, C++, Java, Python

20 Homework Review

Homework Review

Quiz

21 Assertions and Functions

Assertions: a standard way of checking that things are ok

- Java: assert is a function that takes a boolean argument. It will throw an exception if the value is false.
- Python: assert is syntactically like an if statement, acts like Java.
- C++: has a macro assert() that also halts if the expression is false. Can be turned off easily.

Functions

Benefits of functions

- Parameterization of functional blocks, avoids code repetition
- Encapsulation of code, avoids code repetition
- Modular, enables top-down design, facilitates large software systems
- Enables unit testing of code, which facilitates large/robust software systems
- Facilitates portability and re-use, avoids code repetition.

Terminology

- Procedure/subroutine: no return value
- Function: return value
- Method: function that is part of a class
- Argument: the value passed to a function
- Parameter: the identifier that receives an argument

Connecting arguments and parameters

- Pass by value: copy the value of the argument to the parameter
- Pass by reference: copy a reference to the argument to the parameter
- Pass by value-result: copy the argument to the parameter, then copy the result back to the argument
- Pass by name: textually substitute the argument expression into the function

Go over pass by value (most languages), pass by reference (e.g. C++).

22 Functions

Connecting arguments and parameters

- Pass by value: copy the value of the argument to the parameter (review)
- Pass by reference: copy a reference to the argument to the parameter (review)
- Pass by value-result: copy the argument to the parameter, then copy the result back to the argument
- Pass by name: textually substitute the argument expression into the function

Emphasize the efficiency of pass by reference, contrast to the dangers

Algol 68: only language to ever use pass by name
Ada: compilers may implement an in-out variable using reference or value-result

Passing big things to functions is costly: using pointers or references is efficient but more dangerous

- const keyword: the value won't be modified
- can be applied to static objects
- can be applied to references
- can be applied to functions
- `inline` is a suggestion that the function be implemented as a macro

Function call implementation: activation records or stack frames.

- Return address
- Return value
- Arguments/parameters
- Local variables
- A pointer to the prior activation record
- A pointer to the function's static context (class context, base context, global or static variables)
- Saved registers

Parent context: return value, arguments, stack frame pointer, static context reference, return address

Function context: stack frame pointer, local variables, registers

Parent context: restore static and local context pointers, discard arguments, deal with the return value

For an interpreted language, a symbol table replaces the stack as the local variable storage.

23 Functions and Memory Management

Function semantics: return the popped state after executing the function body on the state with added parameters after pushing the old state

```
def MFunc( c, f, state ):
  return MPop( MStatement( f.body, MByValue( f.params, c.args, MPush( state ))))
```

With a return value: return the popped state after executing the function with the given arguments after pushing the existing state with an added return value.

```
def MFunc( c, f, state ):
  return MPop( MStatement( f.body,
                           MByValue( f.params, c.args,
                                     MPush( MAddIdentifier( f.retval, state )))))
```

Remember: functional coding does not have variables

Memory Management

Four types of information in a program.

- Code
- Global Context: statically allocated variables with a global lifetime
- Stack: used for local variables and function calls
- Heap: used for dynamically allocated variables

What is a stack overflow?

Details

- Interpreted or virtual machine languages generally have a fixed size stack space/memory space
- Compiled languages rely on the operating system to detect memory programs/protect the machine
- Heap is needed for dynamic memory allocation, can't do that on the stack
- Modern OS systems often give each program their own virtual memory space

Dynamic Allocation

- Occurs in languages with and without implicit memory management
- Something has to keep track of what has been allocated
- What information is required? Starting address and memory size. Might use a dope vector.

How does malloc avoid searching for free memory?

- keep track of the highest free address and always allocate from there
- with virtual memory and paging, it's not inefficient to not re-use old space unless it's really big
- **chunk**: the data structure to hold memory information, chunks are binned by approximate size
- when malloc needs memory space of a certain size, it can check in the appropriate chunk set first

24 Memory Management and Garbage Collection

Review memory diagram

What is a stack overflow?

Handles

- A pointer to a pointer, generally one of an array maintained by the OS
- Used for memory compaction in older operating systems
- Need to lock them before using them and unlock them after

Garbage Collection

- Reference counting
  - When a new object is created, it stores how many references to itself there are
  - Assignments modify the number of references to objects (increase or decrease)
  - If an object's reference count hits zero in between statements, it can be freed (put on the free list)
  - Benefit: fast garbage collection: no big sweeps through memory.
  - Drawback: circular/ring data structures will never be automatically freed (e.g. doubly-linked list)
- Mark-Sweep
  - Memory consists of nodes/chunks in a list; each node starts marked as free.
  - Allocate memory from a free pointer until memory is used up
  - For each reference in the symbol table, follow it and all child references, marking as non-free
  - Put any node still marked as free back on the free list
  - Benefit: runs only when the heap is full and clears all free memory
  - Drawback: memory can become fragmented; when GC runs, it takes time
- Copy Collection
  - Divide the heap in half, allocate from a free pointer until memory is used up
  - For each reference in the active symbol table, follow it and child references, copy to other half
  - Compact memory as it is copied into the other half
  - Benefit: faster than mark-sweep, no free list to maintain
  - Drawback: uses more memory (1/2 is always free); when GC runs, it takes time

25 Concurrent Programming

Concept of concurrent programming: multiple threads

Issues with concurrent programming

- Deadlock
- Race conditions

Synchronization

- Semaphores: P(s) and V(s)
  - P(s) if $s > 0$ then assign $s = s - 1$, otherwise block the thread that called P
  - V(s) if a thread T is blocked on the semaphore s, then wake up T, otherwise assign $s = s + 1$
  - Most languages that allow parallelism use system classes or libraries to handle synchronization
  - Example adding to a list from the notes
- Barriers: all threads must finish their work before continuing

Threads: independent processes that share global context

- pthread_create: create a thread and give it a function to run, can pass in arguments
- pthread_join: barrier that halts the parent program until the thread is complete
- Go through the summation example
- Using mutex locks with threads: go through the counter example
- These are all system calls in C
- What can we expect for a speedup?

26 Concurrent Programming

Amdahl's law: $S(s) = \frac{1}{(1-p)+\frac{p}{s}}$

- S(s): the expected speedup given resources s
- p: the proportion of the program that can be parallelized
- s: the speedup of the task that benefits from parallelism (e.g. number of threads/CPUs)

Other factors

- I/O bound code
- Memory access time
- Code dependencies
- Actual CPUs, hyperthreading (virtual CPUs)

Threads in Python/Java

- Have to remember that the Python interpreter/JVM is running as well as the program
- Primary purpose is to have threads watching different things like user input

Explicitly Concurrent Langages

- SA-C, a single assignment language: variables represent fixed values, not memory locations
- Erlang, also a single assignment language
- VHDL

Go over lecture note examples.

27 HW Review and Quiz

28 Paradigms: Functional Languages

VHDL Keypoints

- Entity / Architecture concept, can have multiple architectures and specify which one to use
- Statements
- Process statements
- Signals v. variables, purpose of loops is to describe circuits compactly
- For loops map to parallel circults

Functional languages

- Examples: LISP, Scheme, Erlang, Haskell
- Variables represent values, not memory locations
- Functional and imperative languages are equally capable, require different styles of coding

Lambda Expression / Lambda Function

- A nameless mathematical expression that returns a value
- The mathematical expression becomes a named object itself
- Variables in the expression can be substituted for actual values or other expressions
- Loops are generally implemented as recursions
- Assignments are avoided, but allowed

Lambda expressions in Python example: show for loop and recursive loop

Lisp/Scheme

- Cambridge prefix notation, lots of parentheses
- The define function puts a symbol in the symbol table (functions and variables)
- Define makes a connection between a name and a mathematical expression or value
- A list is the core data structure, like Python, lists can hold arbitrary data types
- List manipulation: car, cdr, and cons
  - car is the first element of a list
  - cdr is all of the list but the first element, or the empty list
  - cons makes its first argument the first element of a list using the second as the remainder of the list
  - append concatenates two lists
  - if statements and case statements
- All the other standard structures work

29 Paradigms: Functional Languages

More Lisp

- List manipulation: car, cdr, and cons
  - car is the first element of a list
  - cdr is all of the list but the first element, or the empty list
  - cons makes its first argument the first element of a list using the second as the remainder of the list
  - append concatenates two lists
- if statements
- case statements
- display (print)
- for loops, like Python over a list
- do loop, iteration while a condition is true
- random access data structure: vector

Backward example

Binary search example

Logic Languages: Prolog

- Based on search in a state space: are two nodes connected, and what is the path?
- A logic language is an interface to a search engine
- Specify facts/nodes in a graph
- Specify relationships between nodes
- Specify queries

Prolog: propositional and predicate logic

- predicate/fact: `hobbit(frodo)` means frodo has the property (predicate) hobbit
- Horn clause: defines a new predicate as a function of other predicates
- rule: `hobbitHole(X) :- dry(X), comfortable(X), underground(X)`
- With a Horn clause `p(A) :- q(A)` we can also say that `p(A) or (not q(A))` is also true

Resolution: tracking between Horn clauses, when you can substitute the right side or a term of one clause with the right side of another

Goal: find established facts/predicates that are true that satisfy the query

Go through hobbit hole example

Logic puzzle example

30 Guest Lecture on Time

31 Course Review

Discussion of Monday talk

- How do we currently think about time in programming languages?
- How do we measure time?
- In which applications do we make use of time?
- How could we use time if we had synchronized and syntonized clocks?
- What kind of support could the operating system provide?
- What kind of support could a programming language provide?

Review/What should be in your talk?

- Why do we create different programming languages?
- What are the different design goals of languages?
- Why learn C?
- PL Concepts and grammars
- Regular expressions as a way of defining grammars
- Concrete v. Abstract Syntax
- Naming and Symbol Tables
- Types
- Polymorphism: why not always just use void/Object
- Semantics
- Denotational Semantics: defining each of the standard programming structures
- Exceptions and Assertions
- Functions and function semantics
- Passing information to functions
- Memory management: why do it yourself?
- Concurrent programming, synchronization and locks
- Functional languages
- Logic Languages

32 HW Review, Quiz

33 Talks

PHP (10 min)

SQL (20 min)

HTML (10 min)

CSS (10 min)

34 Talks

Javascript (15 min)

Haskell (25 min)

Bash (10 min)

35 Talks

C++ (30 min)

R (20 min)

36 Talks

Swift (25 min)

Ruby (25 min)

37 Talks

Scala (10 min)

Matlab (10 min)

Python (10 min)

Lisp (10 min)

Fortran (10 min)