CS 33101     Structure of programming languages: 3 credit hours
Course Type:  Required core course
Nature of Course: Theiry + limited labs
Mode of Delivery: Hybrid (online + in class tutorials + multilanguage programming labs)

Instructor’s Name:  ___Purva Gawde / Aditi Singh  (mentor: Professor Arvind Bansal)

(Textbook Title, Author, Year)
(Other Supplemental Material)


Power point slides from the above book

Course Content:
Introduction to syntax, semantics, behavior and implementation issues in imperative, functional, logic and object-oriented languages; type theory, concurrency, data dependency, nondeterminism and internet languages

Prerequisites or co-requisites:  C (2.000) or better in CS 23001. Required

Topics Covered from ACM Body of Knowledge

Core Tier 1:
1.  PL/Basic Type Systems
   a.  A type as a set of values together with a set of operations
   b.  Primitive types (e.g., numbers, Booleans)
   c.  Compound types built from other types (e.g., records, unions, arrays, lists, functions, references)
   d.  Association of types to variables, arguments, results, and fields
   e.  Type safety and errors caused by using values inconsistently given their intended types
2.  Goals and limitations of static typing
   a.  Eliminating some classes of errors without running the program
   b.  Memory allocation optimization
   c.  Code optimization
   d.  Dynamic checks for range, upcasting and string operations
3.  Dynamic typing
4.  Difference between Interpreters, static Compilers and just-in-time compilation
5.  Abstract Syntax Tree; Contrast with Concrete Syntax
6.  Context free and attribute grammars
7. Scope and binding resolution
8. Language translation pipeline: parsing, optional type-checking, translation, linking, execution
   a. Execution as native code or within a virtual machine
   b. Alternatives like dynamic loading and dynamic (or “just-in-time”) code generation
9. Run-time layout of memory: call-stack, heap, static data
   a. Implementing loops, recursion, and tail calls
   b. Implementing different types of parameter passing
10. Side-effect and aliasing
11. Memory management
   a. Manual memory management: allocating, de-allocating, and reusing heap memory
   b. Automated memory management: garbage collection as an automated technique using the notion of reachability
12. Effect-free programming
   a. Function calls have no side effects, facilitating compositional reasoning
   b. Variables are immutable, preventing unexpected changes to program data by other code
   c. Data can be freely aliased or copied without introducing unintended effects from mutation
13. Design and implement a class. [Usage]
   a. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Usage]
14. Correctly reason about control flow in a program using dynamic dispatch. [Usage]
15. Compare and contrast programming in various paradigms: imperative, functional, logic and object oriented

Core Tier 2:
1. Correctly reason about variables and lexical scope in a program using function closures. [Usage]
2. Generic Types and data abstractions
   a. Definition
   b. Parameteric polymorphism, subtypes and adhoc polymorphism
3. Subtyping (cross-reference PL/Type Systems)
   a. Subtype polymorphism; implicit upcasts in typed languages
   b. Relationship between subtyping and inheritance
4. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Familiarity]
5. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Usage]
5. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language.

6. Defining higher-order operations on aggregates such as map, apply

7. Events and event handlers

Elective Topics from ACM Body of Knowledge:

- **Advanced Programming Constructs**
  - Control Abstractions: Blocks, functional abstractions, iterative constructs, iterators and exception handling
  - Module system
  - String manipulation and dynamic code evaluation
- Procedure calls and method dispatching
- Separate compilation; linking
- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting),
- Data layout for objects and activation records
- Just-in-time compilation
- Syntax vs. Semantics: operational, axiomatic, denotational and behavioral
- PL: functional programming: Lambda Calculus
- Principles of language design such as orthogonality
- PL-functional programming: Evaluation order, precedence, and associativity
- Eager vs. delayed evaluation
- Defining control and iteration constructs
- PL/logic programming: Clausal representation of data structures and algorithms
- PL/logic programming: Unification
- PL/logic programming: Backtracking and search

Topics Beyond ACM Body of Knowledge:

- Implementation model of functional, logic and object oriented languages
- Control and data level parallelism in programs
- Control and data dependency in programs
- Deterministic and nondeterministic programs, the notion of guards, completeness and correctness of programs
- Internet based programming language concepts
  - Code and data migration
  - Virtual machines instructions
  - Multimedia languages
- Event based program model
Learning Outcomes:

Core Tier 1 from ACM knowledge base
1. For both a primitive and a compound type, informally describe the values that have that type. [Familiarity]
2. For a language with a static type system, describe the operations that are forbidden statically, such as passing the wrong type of value to a function or method. [Familiarity]
3. Describe examples of program errors detected by a type system. [Familiarity]
4. For multiple programming languages, identify program properties checked statically and program properties checked dynamically. [Usage]
5. Give an example program that does not type-check in a particular language and yet would have no error if run. [Familiarity]
6. Explain how programs that process other programs treat the other programs as their input data. [Familiarity]
7. Describe an abstract syntax tree for a small language. [Usage]
8. Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Familiarity]
9. Distinguish syntax and parsing from semantics and evaluation. [Familiarity]

Core Tier 2 from ACM knowledge base
1. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Familiarity]
2. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Usage]

Electives from ACM knowledge base
- Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Familiarity]
- Generate the low-level code for calling functions/methods in modern languages. [Usage]
- Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Familiarity]
- Discuss benefits and limitations of automatic memory management. [Familiarity]
- Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers. [Familiarity]
- Discuss advantages, disadvantages, and difficulties of just-in-time compilation.
- Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, finite mapping (for arrays and associations), monomorphism vs. polymorphism, nd types in various imperative languages
- Type checking in statically types languages and type inference in polymorphic languages
- Write a lambda-calculus program and show its evaluation to a normal form. [Usage]
- Discuss the different approaches of operational, denotational, and axiomatic semantics. [Familiarity]
- Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
- Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]
- Use parametricity to establish the behavior of code given only its type. [Usage]
- Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Familiarity]
- Use crisp and objective criteria for evaluating language-design decisions. [Usage]
- Give an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Usage]
- Show uses of delayed evaluation, such as user-defined control abstractions. [Familiarity]
- Use a logic language to implement a simple conventional algorithm such as matrix multiplication, different types of sorts and string manipulations [Usage]

Additional Learning Outcomes Beyond ACM Knowledge Base

- Prepare students to learn new trendy language that integrate multiple combinations of programming paradigms by discussing fundamental characterization of various paradigms
- Discussing how a program can be constructed scientifically using post conditions and Pre-conditions using axioms and guards
- Making students familiar with synchronization issues and multimedia compression issues for modeling real world multimedia applications
- Discuss how the different types of dependency effect parallelization and concurrency of deterministic programs