Foundations of Programming Languages Knowledge Area

Preamble

- What characterizes this knowledge area?

This knowledge area leverages discrete mathematics and logic to provide a basis for the understanding of the foundations of programming languages, their implementation, and their formal description. Programming languages vary based upon the problem domains and evolve based upon societal needs and technological advancement. Modern day software are complex, interleaving multiple problem domains. Despite differences in evolving languages and language paradigms, underlying abstract models of computation and program development are shared. In addition, underlying processors and their interface with programming tools are getting intertwined and progressively complex. Understanding these shared abstractions and programming paradigms promotes faster learning of new changes and new programming languages.

- Brief description of the knowledge area.

Foundations of Programming Languages is not about programming in a particular language, rather it is about underlying concepts and principles of programming languages, the strengths and weaknesses of various programming paradigms, how programming languages interface with other entities such as operating systems and hardware, how programming languages are implemented, and how we can formally specify the definition of a programming language and the behavior of a program. The concepts covered are applicable to many different languages and an understanding of these principles assists in being able to move readily from one language to the next, and to be able to select a programming paradigm and a programming language to best suit the problem at hand.

- Brief statement of learning outcomes for the courses arising from this knowledge area.

Two example courses are presented at the end of this knowledge area to illustrate how the content may be covered. The first is an introductory course which covers the CS Core and KA Core content. This is a course focused on the different programming paradigms and ensue familiarity with each to a level sufficient to be able to decide which paradigm is appropriate in which circumstances.

The second course is an advanced course focused on the implementation of a programming language and the formal description of a programming language and a formal description of the behavior of a program.

While these 2 courses have been the predominant way to cover this knowledge area of the past decade, it is by no means the only way that the content can be covered. An institutions could, for example, choose to cover only the CS Core content (28 hours) in a shorter course, or in a course
which combines this CS Core content with Core content from another knowledge area such as Software Engineering. Natural combinations are easily identifiable since they are the areas in which the Foundation s of Programming Languages knowledge areas overlaps with other knowledge areas. A list of such overlap areas is provided at the end of this knowledge area.

- How has this KA changed since CS 2013?

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. Over the course of a career, a computer scientist will need to learn and work with many different languages, separately or together. Software developers must understand the programming models, new programming features and constructs, underlying different languages and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs and must understand the principles underlying how programming language features are defined, composed, and implemented to improve execution efficiency and long-term maintenance of developed software. The effective use of programming languages and appreciation of their limitations, also requires a basic knowledge of programming language translation and program analysis, of run-time behavior and components such as memory management and interplay of concurrent processes communicating with each other through message-passing, shared memory, and synchronization. Finally, some developers and researchers will need to design new languages, an exercise which requires familiarity with basic principles.

Changes since the 2013 ACM/IEEE-CS Computer Science curricular guidelines (CS2013) include a change in name of the Knowledge Area from Programming Languages to Foundations of Programming Languages to better reflect the fact that the KA is about the fundamentals underpinning programming languages and related concepts, and not about any specific programming language. Changes also include a redistribution of content formerly identified as core tier-1 and core tier-2 within the Programming Language Knowledge Area (KA). These are now CS Core hours and KA Core hours. All computer science graduates are expected to have the CS Core hours, and those graduates that specialize in a knowledge area are also expected to have the KA core hours. Content that is not identified as either CS Core hours or KA Core hours are non-core topics. Specifically, the changes are:

- Object-Oriented Programming +1 CS Core hours, -2 KA Core hours
- Functional Programming +1 CS Core hour
- Logic Programming +3 CS Core hours
- Event-Driven and Reactive Programming +2 CS Core hours
- Parallel and Distributed Computing +1 CS Core hours, +2 KA Core hours
- Type Systems +1 CS Core hours, -1 KA Core hours
- Language Translation and Execution +4 CS Core hours, -3 KA Core hours
- Program Representation +2 KA Core hours
In addition, some knowledge units are renamed to more accurately reflect their content:

- Static Analysis is renamed to Program Analysis and Analyzers
- Concurrency and Parallelism is renamed to Parallel and Distributed Computing
- Runtime Systems is renamed to Runtime Behavior and Systems
- Basic Type Systems and Type Systems were merged into a single topic and named Type Systems

Three new knowledge units have been added to reflect their continuing and growing importance as we look toward the 2030s:

- Scripting +2 CS Core hours
- Formal Development Methodologies
- Embedded Systems and Hardware Interface

**Allocation of Core Hours**

FPL. Fundamentals of Programming Languages (26 CS Core hours, 18 KA Core hours)

<table>
<thead>
<tr>
<th>Knowledge Unit</th>
<th>CS Core hours</th>
<th>KA Core hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPL/Object-Oriented Programming</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>FPL/Functional Programming</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>FPL/Logic Programming</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FPL/Scripting</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FPL/Event-Driven and Reactive Programming</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FPL/Parallel and Distributed Computing</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>FPL/Type Systems</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>FPL/Language Translation and Execution</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FPL/Program Representation</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>FPL/Syntax Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Compiler Semantic Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Program Analysis and Analyzers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Code Generation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Runtime Behavior and Systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Advanced Programming Constructs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Language Pragmatics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Formal Semantics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Formal Development Methodologies</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/Embedded Computing and Hardware Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL/FPL and SEP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note:

- Some topics from one or more of the first three Knowledge Units (Object-Oriented Programming, Functional Programming, Event-Driven and Reactive Programming) are likely to be integrated with topics in the Software Development Fundamentals Knowledge Area in a curriculum’s introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- Different programming paradigms correspond to different problem domains. Most languages have evolved to integrate more than one programming paradigms such imperative with OOP, functional programming with OOP, logic programming with OOP, event and reactive modeling with OOP. Hence, the emphasis is not on just one programming paradigm but a balance of all major programming paradigms.
- With multicore computing, cloud computing and computer networking becoming commonly available in the market, it has become imperative to understand the integration of “Distribution, concurrency, parallelism and code mobility” along with other programming paradigm as a core area. This paradigm is integrated with almost all other major programming paradigms.
- With increased emphasis on data management and Artificial Intelligence and introduction of “Data Management” and “Artificial Intelligence” as knowledge areas, concepts suitable for AI programming and data management have become imperative. Hence, predicate-based database and AI programming present in logic programming paradigm has been introduced as a core CS area along with functional programming.
- With ubiquitous computing and real-time temporal computing getting into daily human life such as health, transportation, smart homes, it has become important to cover the software development aspect of “Embedded Computing and hardware interfaces” under programming languages. Some of the topics covered will require and interface with concepts covered in knowledge areas such as “Architecture and Organization”, “Operating Systems”, and “Systems Fundamentals”.
- Some topics from the Parallel and Distributed Computing Knowledge Unit are likely to integrated within the curriculum with topics from the Parallel and Distributed Programming Knowledge Area.
- Some topics from the Hardware Interface Knowledge Unit are likely to integrated within the curriculum with topics from the System Fundamentals Knowledge Area.
Description of Knowledge Units

**FPL/Object-Oriented Programming**

[5 CS Core hours, 3 KA Core hours]

- **Topics**
  - **[CS Core]**
    - Object-oriented design
      - Decomposition into objects carrying state and having behavior
      - Class-hierarchy design for modeling
    - Definition of classes: fields, methods, and constructors
    - Subclasses, inheritance (including multiple inheritance), and method overriding
    - Virtual methods
    - Dynamic dispatch: definition of method-call
    - Exception handling
    - Object-oriented idioms for encapsulation
      - Privacy, data hiding, and visibility of class members
      - Interfaces revealing only method signatures
      - Abstract base classes, traits and mixins
    - Dynamic vs static properties
    - Composition vs inheritance
  - **[KA Core]**
    - Subtyping *(cross-reference: FPL/Type Systems)*
      - Subtype polymorphism; implicit upcasts in typed languages
      - Notion of behavioral replacement: subtypes acting like supertypes
      - Relationship between subtyping and inheritance
      - Collection classes, iterators, and other common library components [cross-reference: FPL/Functional Programming]

- **Illustrative learning outcomes**
  - **[CS Core]**
    1. Compose a class through design, implementation, and testing to meet behavioral requirements. [Creating]
    2. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses. [Applying]
    3. Predict and validate control flow in a program using dynamic dispatch. [Analyzing, Evaluating]
    4. Compare and contrast
      a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and
b) the object-oriented approach-defining a class for each data variant with the
class definition providing a method for each operation.
Understand both as defining a matrix of operations and variants. [Analyzing]
[cross-reference: FPL/Functional Programming]
5. Compare and contrast the benefits and costs/impact of using inheritance (sub-
classes) and composition (in particular how to base composition on higher order
functions). [Analyzing]

- [KA Core]
  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 ex-
     pects the supertype). [Understanding]
  2. Use object-oriented encapsulation mechanisms such as interfaces and private
     members. [Applying]
     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. [Enumerate, Applying]
     [cross-reference: FPL/Functional Programming]

FPL/Functional Programming
[4 CS Core hours, 4 KA Core hours]
- Topics
  - [CS Core]
    o Lambda expressions and evaluation
      - Variable binding and scope rules
      - Parameter passing
      - Nested lambda expressions and reduction order
    o Effect-free programming
      - Function calls have no side effects, facilitating compositional reasoning
      - Immutable variables and data copying vs. reduction
      - Use of recursion vs loops
    o Processing structured data (e.g., trees) via functions with cases for each data vari-
      ant
      - Associated language constructs such as modeling function composition using
        functions as parameters, map, reduce/fold, filter, iterators, and recursion
      - Functions defined over compound data in terms of functions applied to the
        constituent pieces
    o Using higher-order functions (taking, returning, and storing functions), for exam-
      ple map, reduce
• [KA Core]
  o Function closures (functions using variables in the enclosing lexical environment)
    • Basic meaning and definition -- creating closures at run-time by capturing the environment
    • Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
    • Using a closure to encapsulate data in its environment
    • Lazy versus eager evaluation
  o Defining and implementing functions which can accept a function as a parameter, combined with another function, or return a function.
  o Evaluating λ-expressions using SECD machine
    • Definition and significance of each stack
    • Storing environment as closure
    • Call back implementation

• [Non-Core]
  o Graph reduction machine and call-by-need
  o Implementing lazy evaluation using STG machine and ABC machine
  o Integration with logic programming paradigm using concepts such as equational logic, narrowing, residuation and semantic unification [cross-reference: FPL/Logic Programming]
  o Integration with other programming paradigm such as imperative and object-oriented [understanding]
  o Exploiting concurrency using concurrency constructs and message passing libraries such as Mvar in Haskell, multithreading and mutex locks in ruby, etc. [understanding]

• Illustrative learning outcomes
  • [CS Core]
    1. Develop basic algorithms that avoid assigning to mutable state or considering reference equality. [Creating]
    2. Develop useful functions that take and return other functions. [Creating]
    3. Compare and contrast
       a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant, and
       b) the object-oriented approach-defining a class for each data variant with the class definition providing a method for each operation.
Understand both as defining a matrix of operations and variants. [Analyzing] [cross-reference: FPL/Object-Oriented Programming]

- [KA Core]
  4. Explain a simple example of lambda expression being implemented using SECD machine showing storage and reclaim of the environment [Understanding]
  5. Correctly interpret variables and lexical scope in a program using function closures. [Understanding]
  6. Use functional encapsulation mechanisms such as closures and modular interfaces. [Applying]
  7. Compare and contrast stateful vs stateless execution [Analyzing]
  8. 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. [Enumerate, Applying] [cross-reference: FPL/Object-Oriented Programming]

- [Non-core]
  9. Illustrate graph reduction using a λ-expression using a shared subexpression [Understanding]
  10. Illustrate the execution of a simple nested λ-expression using an ABC machine [Understanding]
  11. Illustrate narrowing, residuation and semantic unification using simple illustrative examples [Understanding]
  12. Illustrate the concurrency constructs using simple programming examples of known concepts such as a buffer being read and written concurrently or sequentially [Understanding]

FPL/Logic Programming
[3 CS Core hours, Non-core]

  - Topics
    - [CS Core]
      - Universal vs. existential quantifiers
      - First order predicate logic vs. higher order logic
      - Expressing complex relations using logical connectives and simpler relations
      - Definitions of Horn clause, facts, goals and subgoals
      - Logic programs as set of Procedures containing set of Horn clauses having same name and number of arguments
      - Nondeterministic nature of logic programs
- Unification and Unification algorithm; unification vs. assertion vs expression evaluation
- Mixing relations with functions
- AND-OR tree
- Query reduction using AND-OR tree and unification as means of parameter passing
- Modeling logical programs using first order predicate calculus
- Generating alternate solutions using backtracking and depth-first search of AND-OR tree
- Cuts

• [Non-Core]
- Memory overhead of variable copying in handling iterative programs
- Programming constructs to store partial computation and pruning search tree
- Mixing functional programming and logic programming using concepts such as equational logic, narrowing, residuation and semantic unification [*cross-reference FPL/functional programming*]
- Warren Abstract Machine and the use trail stack for backtracking
- Recursive programming
- Constraint logic programming
- Program Analysis to reduce copying overheads and optimization
- Concurrency in logic programs: AND-OR parallelism models, Associative models, etc.
- Inductive logic programming
- Higher order logic programming
- Integration with other programming paradigms such as object-oriented programming
- Advance programming constructs such as difference-lists, creating user defined data structures, set of, etc.

• Illustrative Learning Outcomes
  • [CS Core]
    1. Use a logic language to implement a conventional algorithm. [Applying]
    2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts. [Applying]
    3. Use a simple illustrative example to show correspondence between First Order predicate Logic (FOPL) and logic programs using Horn clauses [Applying]
    4. Use examples to illustrate unification algorithm and its role of parameter passing in query reduction [Applying]
5. Use simple logic programs interleaving relations, functions, and recursive programming such as factorial and Fibonacci numbers and simple complex relationships between entities, and illustrate execution and parameter passing using unification and backtracking [Applying]

- [Non-core]
6. Illustrate computation of simple programs such as Fibonacci and show overhead of recomputation, and then show how to improve execution overhead [Understanding]

**FPL/Scripting**
[2 CS Core hour]
- Topics
  - Divide, combine, conquer
  - Concurrency
  - Error/exception handling
  - I/O redirection
  - System commands
  - Environment variables
  - File test operators
  - Data structures
    - Arrays and lists
    - Slices
    - List Comprehensions
  - Regular expressions
  - Dynamic typing
  - Function declarations
  - Processes and threads
  - Code objects

- Illustrative Learning Outcomes
  1. Create and execute automated scripts to manage various system tasks. [Applying]
  2. Solve various text processing problems through the [Applying]

**FPL/Event-Driven and Reactive Programming**
[2 CS Core hour, 2 KA Core hours]
This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.
Topics

[CS Core]
- Procedural programming vs. Reactive programming: advantages of reactive programming in capturing events
- Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers
- Behavior model of event-based programming
- Canonical uses such as GUIs, mobile devices, robots, servers
- Finite state machines [cross-reference: AL/Automata and Complexity]

[KA Core]
- Using a reactive framework
  - Defining event handlers/listeners
  - Parameterization of event senders and event arguments
  - Main event loop not under event-handler-writer's control
  - Externally-generated events and program-generated events
- Separation of model, view, and controller

Illustrative Learning Outcomes

[CS Core]
1. Implement event handlers for use in reactive systems, such as GUIs. [Applying]
2. Examine why an event-driven programming style is natural in domains where programs react to external events. [Analyzing]

[KA Core]
3. Define and use a reactive framework [Using]
4. Describe an interactive system in terms of a model, a view, and a controller. [Understanding]

FPL/Parallel and Distributed Computing
[3 CS Core hours, 2 KA Core Hours]
Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this non-core Knowledge Unit aims only to complement the material included elsewhere in the body of knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.
[Cross-reference: PD/Parallel and Distributed Computing, SF/Parallelism]
Topics

[CS Core]
- Definitions of basic concepts such as nondeterministic programming, racing, sequential consistency, concurrency, threads, message passing, data parallelism, types of task parallelism, data dependency, control dependency, synchronization, loop unrolling, and granularity with sufficient illustrations
- Guarded commands: understand difference between necessary and sufficient conditions, logically build guarded programs using post-condition and pre-conditions
- Constructs for thread-shared variables and shared-memory synchronization
- Actor models
- Synchronous programming
- Asynchronous programming
- Concurrent programming and synchronization constructs such as cobegin-coend, monitor, fiber, synchronized methods
- Communicating sequential processes
- Models for passing messages between sequential processes
- Loci of control
- Compare and contrast shared memory model and message passing model
- Data dependencies: producer-consumer relationship, Anti-dependency, output dependency
- Control dependency and loop unrolling
- Remote Procedure Calls (RPC) and types of parameter-passing in distributed languages such as call by value and call by reference
- Programming models for mobile codes in languages such as Java and Emerald
- Parameter passing in distributed computing in languages such as Emerald: call-by-object reference, call by visit, call by move
- Distributed objects as in Java and Emerald; Remote method invocation
- Code and data mobility such as client server model, remote evaluation model, code-on-demand model, migrating agent model

[KA Core]
- Futures
- Language support for data parallelism such as forall, loop unrolling, map/reduce
- Effect of memory-consistency models on language semantics and correct code generation
- Representational State Transfer Application Programming Interfaces (REST APIs)
- Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing
- Overheads of message passing
Granularity of program for efficient exploitation of concurrency.

Illustrative Learning Outcomes

- [CS Core]
  1. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Understanding]
  2. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, synchronization constructs, and data-parallelism primitives. [Applying]
  3. Use a message-passing model to analyze a communication protocol. [Applying]
  4. Use synchronization constructions such as monitor/synchronized methods in a simple program [Applying]
  5. Modeling data dependency using simple programming constructs involving variables, read and write [Applying]
  6. Modeling control dependency using simple constructs such as selection and iteration [Applying]

- [KA Core]
  7. Explain how REST API's integrate applications and automate processes [Understanding].
  8. Explain benefits, constraints and challenges related to distributed and parallel computing [Understanding].

FPL/Type Systems

[3 CS Core hours, 4 KA Core hours, Non-core]

The KA Core hours would be profitably spent both on the KA Core topics and on a less shallow treatment of the CS Core topics and learning outcomes.

Topics

- [CS Core]
  - A type as a set of values together with a set of operations
    - Primitive types (e.g., numbers, Booleans)
    - Compound types built from other types (e.g., records, unions, arrays, lists, functions, references) using set operations
  - Association of types to variables, arguments, results, and fields
  - Type safety as an aspect of program correctness [cross-reference: FPL/Formal Semantics]
  - Type safety and errors caused by using values inconsistently given their intended types
Statically-typed vs dynamically-typed programming languages
Type equivalence: structural vs name equivalence
Goals and limitations of static and dynamic typing
  • Detecting and eliminating errors as early as possible
Generic types (parametric polymorphism) [cross-reference: FPL/Formal Semantics]
  • Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion
  • Comparison of monomorphic and polymorphic types
  • Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
  • Generic parameters and typing
  • Use of generic libraries such as collections
  • Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
  • Prescriptive vs. descriptive polymorphism
  • Implementation models of polymorphic types

[KA Core]
Complementary benefits of static and dynamic typing
  • Errors early vs. errors late/avoided
  • Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections
  • Typing rules
    o Rules for function, product, and sum types
  • Hindley-Milner type system
    o Use of unification to find most general types
    o Lambda calculus plus let-polymorphism
  • Grant-Reynolds type system (System F/Polymorphic Lambda Calculus)
    o Rules for function and type abstraction
    o Relation to Hindley-Milner type system
  • Avoid misuse of code vs. allow more code reuse
  • Detect incomplete programs vs. allow incomplete programs to run
  • Relationship to static analysis
  • Decidability
  • Use of sophisticated (complex) type systems, e.g., Rust.
• [Non-core]
  o Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
  o Type checking
  o Subtyping [cross-reference: FPL/Object-Oriented Programming]
    • Subtype polymorphism; implicit upcasts in typed languages
    • Notion of behavioral replacement: subtypes acting like supertypes
    • Relationship between subtyping and inheritance
  o Type safety as preservation plus progress
  o Type inference
  o Static overloading
  o Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) [cross-reference: FPL/Formal Methods]
  o Dependent types (universal quantification as dependent function, existential quantification as dependent product) [cross-reference: FPL/Formal Methods]

• Illustrative Learning Outcomes
  • [CS Core]
    1. Describe, for both a primitive and a compound type, the values that have that type. [Understanding]
    2. Describe, for a language with a static type system, the operations that are forbidden statically, such as passing the wrong type of value to a function or method. [understanding]
    3. Describe examples of program errors detected by a type system. [Understanding]
    4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically. [Enumerate]
    5. Describe an example program that does not type-check in a particular language and yet would have no error if run. [Understanding]
    6. Use types and type-error messages to write and debug programs. [Applying]

  • [KA Core]
    7. Explain how typing rules define the set of operations that are legal for a type. [Understanding]
    8. List the type rules governing the use of a particular compound type. [Enumerate]
    9. Explain why undecidability requires type systems to conservatively approximate program behavior. [Understanding]
    10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections. [Enumerate]
11. Discuss the differences among generics, subtyping, and overloading. [Understanding]
12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software. [Understanding]

- [Non-core]
13. Define a type system precisely and compositionally. [Enumerate]
14. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Enumerate]
15. Precisely describe the invariants preserved by a sound type system. [Understanding]
16. Prove type safety for a simple language in terms of preservation and progress theorems. [Evaluating]
17. Implement a unification-based type-inference algorithm for a simple language. [Applying]
18. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs. [Understanding]

**FPL/Language Translation and Execution**
[4 CS Core hours, Non-core]

- Topics
  - [CS Core]]
    - Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
    - Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution
    - BNF and extended BNF representation of Context-free grammar
    - Parse tree using a simple sentence such as arithmetic expression or if-then-else statement
    - Ambiguity in Parsing due to lack of precedence order and resolution
    - Execution as native code or within a virtual machine
    - Alternatives like dynamic loading and dynamic (or "just-in-time") code generation
    - Control-flow diagrams using selection and iteration
    - Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string
    - Direct, indirect and indexed access to memory location
    - Runtime representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects
- Abstract low level machine with simple instruction, stack and heap to explain translation and execution
- Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap [cross reference: AR/Memory System Architecture and Organization, OS/Memory Management]
  - Translating selection and iterative constructs to control-flow diagrams
  - Translating control-flow diagrams to low level abstract code
  - Implementing loops, recursion, and tail calls
  - Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine
- Memory management [cross reference: OS/Memory Management, FPL/Hardware Interface]
  - Low level allocation and accessing of high level data structures such as basic data types, n-dimensional array, vector, record, and objects
  - Return from procedure as automatic deallocation mechanism for local data elements in the stack
  - Manual memory management: allocating, de-allocating, and reusing heap memory
- [Non-core]
  - Memory management [cross reference: OS/Memory Management, FPL/Hardware Interface]
    - Automated memory management: garbage collection as an automated technique using the notion of reachability
  - Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
  - Secure compiler development [cross-reference: SEC/Foundational Security]

Illustrative Learning Outcomes

[CS Core]
1. Differentiate a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Understanding]
2. Differentiate syntax and parsing from semantics and evaluation. [Understanding]
3. Use BNF and extended BNF to specify the syntax of simple constructs such as if-then-else, type declaration and iterative constructs for known languages such as C++ or Python [Applying]
4. Illustrate parse tree using a simple sentence/arithmetic expression [Applying]
5. Illustrate translation of syntax diagrams to BNF/extended BNF for simple constructs such as if-then-else, type declaration, iterative constructs, etc. [Understanding]
6. Illustrate ambiguity in parsing using nested if-then-else/arithmetic expression and show resolution using precedence order [Understanding]
7. Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions. [Applying]
8. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model. [Understanding]
9. Investigate, identify, and fix memory leaks and dangling-pointer dereferences. [Applying]

[Non-core]
10. Discuss the benefits and limitations of garbage collection, including the notion of reachability. [Understanding]

FPL/Program Representation
[3 KA Core hours]

- Topics
  - BNF and regular expressions
  - Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
  - Components of a language
    - Definitions of alphabets, delimiters, sentences, syntax and semantics
    - Syntax vs. semantics
  - Types of Semantics: operational, axiomatic, denotational, behavioral Define and using abstract syntax trees; contrast with concrete syntax
  - Program as a set of non-ambiguous meaningful sentences
  - Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling
  - Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation
  - L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values
  - Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables
• Scope rules: static vs. dynamic; visibility of variables; side-effects
• Environment vs. store and their properties
• Data and control abstraction
• Mechanisms for information exchange between program units such as procedures, functions and modules: nonlocal variables, global variables, parameter passing, import-export between modules
• Types of parameter passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations
• Side-effects induced by nonlocal variables, global variables and aliased variables
• Data structures to represent code for execution, translation, or transmission
• Low level instruction representation such as virtual machine instructions, assembly language, and binary representation \[cross-reference: AR/Machine Level Representation of Data\]
• Lambda calculus, variable binding, and variable renaming.
• String-based mobility in mobile coding.

Illustrative Learning Outcomes
1. Illustrate the scope of variables and visibility using simple programs [Understanding]
2. Illustrate different types of parameter passing using simple pseudo programming language [Understanding]
3. Explain side-effect using global and nonlocal variables and how to fix such programs [Understanding]
4. Explain how programs that process other programs treat the other programs as their input data. [Understanding]
5. Describe a grammar and an abstract syntax tree for a small language. [Understanding]
6. Describe the benefits of having program representations other than strings of source code. [Understanding]
7. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator. [Applying]

FPL/Syntax Analysis
[Non-core]
• Topics
  • Regular grammars vs. context-free grammars
  • Scanning and parsing based on language specifications
  • Lexical analysis using regular expressions
  • Tokens and their use
  • Parsing strategies including top-down (e.g., recursive descent, parser combinators, Earley parsing, or LL) and bottom-up (e.g., LR or GLR) techniques.
Lookahead tables and their application to parsing

- Language theory
  - Chomsky hierarchy
  - Left-most/right-most derivation and ambiguity
  - Grammar transformation

- Parser error recovery mechanisms
- Generating scanners and parsers from declarative specifications

Illustrative Learning Outcomes
1. Use formal grammars to specify the syntax of languages. [Applying]
2. Illustrate the role of lookahead tables in parsing [Understanding]
3. Use declarative tools to generate parsers and scanners. [Applying]
4. Recognize key issues in syntax definitions: ambiguity, associativity, precedence. [Enumerate]

FPL/Compiler Semantic Analysis
[Non-core]

- Topics
  - Abstract syntax trees; contrast with concrete syntax
  - Defining, traversing and modifying high-level program representations
  - Scope and binding resolution
  - Static Semantics
    - Type checking
    - Define before use
    - Annotation and extended static checking frameworks
  - L-values/R-values [cross reference: SDF/Fundamental Programming Constructs]
  - Call semantics
  - Parameter passing mechanisms
  - Declarative specifications such as attribute grammars and their applications in handling limited context-base grammar

Illustrative Learning Outcomes
1. Describe an abstract syntax tree for a small language [Understanding]
2. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Applying]
3. Describe semantic analyses using an attribute grammar. [Understanding]
FPL/Program Analysis and Analyzers
[Non-core]

- Topics
  - Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment.
  - Undecidability and consequences for program analysis
  - Flow-insensitive analysis, such as type-checking and scalable pointer and alias analysis
  - Flow-sensitive analysis, such as forward and backward dataflow analyses
  - Path-sensitive analysis, such as software model checking and software verification
  - Tools and frameworks for implementing analyzers
  - Role of static analysis in program optimization and data dependency analysis during exploitation of concurrency [cross-reference: ç]
  - Role of program analysis in (partial) verification and bug-finding [cross reference: FPL/Code Generation]
  - Parallelization
    - Analysis for auto-parallelization
    - Analysis for detecting concurrency bugs

- Illustrative Learning Outcomes
  1. Define useful program analyses in terms of a conceptual framework such as dataflow analysis. [Enumerate]
  2. Explain the difference between dataflow graph and control flow graph. [Understanding]
  3. Explain why non-trivial sound program analyses must be approximate. [Understanding]
  4. Argue why an analysis is correct (sound and terminating). [Analyzing]
  5. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Understanding]
  6. Use the results of a program analysis for program optimization and/or partial program correctness. [Applying]

FPL/Code Generation
[Non-core]

- Topics
  - Instruction sets [cross reference: AR/Assembly Level Machine Organization]
  - Control flow
  - Memory management [cross reference: AR/Memory System Organization and Architecture, OS/Memory Management]
• Procedure calls and method dispatching
• Separate compilation; linking
• Instruction selection
• Instruction scheduling (e.g., pipelining)
• Register allocation
• Code optimization as a form of program analysis

Illustrative Learning Outcomes
1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Enumerate]
2. Generate the low-level code for calling functions/methods in modern languages. [Creating]
3. Discuss why separate compilation requires uniform calling conventions. [Understanding]
4. Discuss why separate compilation limits optimization because of unknown effects of calls. [Understanding]
5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization. [Understanding]

FPL/Runtime Behavior and Systems
[Non-core]
• Topics
  • Process models using stacks and heaps to allocate and deallocate activation records and recovering environment using frame pointers and return addresses during a procedure call including parameter passing examples.
  • Schematics of code lookup using hash tables for methods in implementations of object-oriented programs
  • Data layout for objects and activation records
  • Object allocation in heap
  • Implementing virtual entities and virtual methods; virtual method tables and their application
  • Runtime behavior of object-oriented programs
  • Compare and contrast allocation of memory during information exchange using parameter passing and non-local variables (using chain of static links)
  • Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
• Just-in-time compilation and dynamic recompilation
• Interface to operating system (e.g., for program initialization)
• Interoperability between programming languages including parameter passing mechanisms and data representation
  o Big Endian, little endian
  o Data layout of composite data types such as arrays
• Other common features of virtual machines, such as class loading, threads, and security checking
• Sandboxing

Illustrative Learning Outcomes
1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Analyzing]
2. Discuss benefits and limitations of automatic memory management. [Understanding]
3. 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. [Understanding]
4. Compare and contrast static allocation vs. stack-based allocation vs. heap-based allocation of data elements. [Analyzing]
5. Explain why some data elements cannot be automatically deallocated at the end of a procedure/method call (need for garbage collection). [Understanding]
6. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation. [Understanding]
7. Discuss use of sandboxing in mobile code [Understanding]
8. Identify the services provided by modern language run-time systems. [Enumerate]

FPL/Advanced Programming Constructs
[Non-core]
• Topics
  • Encapsulation mechanisms
  • Lazy evaluation and infinite streams
  • Compare and contrast lazy evaluation vs. eager evaluation
  • Unification vs. assertion vs. expression evaluation
  • Control Abstractions: Exception Handling, Continuations, Monads
  • Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
  • Metaprogramming: Macros, Generative programming, Model-based development
  • String manipulation via pattern-matching (regular expressions)
  • Dynamic code evaluation ("eval")
  • Language support for checking assertions, invariants, and pre/post-conditions
• Domain specific languages, such as database languages, data science languages, embedded computing languages, synchronous languages, hardware interface languages
• Massive parallel high performance computing models and languages

Illustrative Learning Outcomes
1. Use various advanced programming constructs and idioms correctly. [Usage]
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

FPL/Language Pragmatics
[Non-core]
• Topics
  • Effect of technology needs and software requirements on programming language development and evolution
  • Problems domains and programming paradigm
    o Criteria for good programming language design
    o Principles of language design such as orthogonality
    o Defining control and iteration constructs
    o Modularization of large software
  • Evaluation order, precedence, and associativity
  • Eager vs. delayed evaluation
  • Defining control and iteration constructs
  • External calls and system libraries

Illustrative Learning Outcomes
1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Understanding]
2. Use crisp and objective criteria for evaluating language-design decisions. [Applying]
3. Implement an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Applying]
4. Illustrate uses of delayed evaluation, such as user-defined control abstractions. [Applying]
5. Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation. [Understanding]
FPL/Formal Semantics
[cross-reference: SE/Formal Methods]
[Non-core]

- **Topics**
  - Syntax vs. semantics
  - Lambda Calculus
  - Constraint checkers
  - Symbolic execution
  - Approaches to semantics: Operational, Denotational, Axiomatic
  - Operational semantics analysis of abstract constructs and sequence of such as assignment, expression evaluation, selection, iteration using environment and store
  - Axiomatic semantics of abstract constructs such as assignment, selection, iteration using pre-condition, post-conditions and loop invariance
    - Loop-invariant conditions and their derivation in iterative constructs
    - Proving program correctness using pre-conditions and post-conditions
  - Denotational semantics
    - Semantic domains, including domain lifting, and semantic algebra
    - Fix-point semantics and continuation
    - Semantics of expression evaluation, assignment, and selection using environment and store
    - Semantics of iterative constructs using fix point semantics
    - Semantics of command sequence using continuation
  - Action semantics
  - Proofs by induction over language semantics
  - Formal definitions and proofs for type systems [cross-reference: FPL/Type Systems]
  - Parametricity [cross-reference: FPL/Type Systems]
  - Using formal semantics for systems modeling
  - Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) [cross-reference: FPL/Type Systems]
  - Dependent types (universal quantification as dependent function, existential quantification as dependent product) [cross-reference: FPL/Type Systems]

- **Illustrative Learning Outcomes**
  1. Construct a formal semantics for a small language. [Creating]
  2. Write a lambda-calculus program and show its evaluation to a normal form. [Applying]
  3. Discuss the different approaches of operational, denotational, and axiomatic semantics. [Understanding]
  4. Use induction to prove properties of all programs in a language. [Applying]
5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system. [Applying]

6. Use parametricity to establish the behavior of code given only its type. [Applying]

7. Use formal semantics to build a formal model of a software system other than a programming language. [Applying]

FPL/Formal Development Methodologies
[cross-reference: SE/Formal Methods]
[Non-core]

- Topics
  - Formal specification languages and methodologies
  - Theorem provers and logics
  - Specification and proof discharge for fully verified software systems (e.g. dafny, Coq, F*)
  - Formal modelling and manual refinement/implementation of software systems (e.g. TLA+)
  - Development with checker assisted Domain Specific Languages (DSLs) and specialized languages (e.g. Liquid Haskell and Ivy)
  - Use of symbolic testing and symbolic model checking systems in the overall software development system.
  - Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

- Illustrative Learning Outcomes
  1. Use formal modeling techniques to develop and validate architectures. [Applying]
  2. Use proof assisted programming languages to develop fully specified and verified software artifacts. [Applying]
  3. Use verifier and specification support in programming languages to formally validate system properties. [Applying]
  4. Integrate symbolic validation tooling into a programming workflow. [Analyzing]
  5. Discuss when and how formal methods can be effectively used in the development process. [Understanding]

FPL/Embedded Computing and Hardware Interface
[Non-core]

- Topics
  - Embedded Systems
    - Microcontrollers
Interrupts and feedback
- Sensors/actuators
- GPU technology
- Field Programmable Gate Arrays
- Energy efficiency
- Simulation of logical circuits with timing control and thread-based programming
- Modeling registers and microcontrollers
- Loosely timed coding and synchronization
- Transaction level modeling: reading, writing, master/slave simulation, unidirectional vs. bidirectional communication
- Modeling generic Transaction Level Model (TLM) component
- Synchronization constructs in synchronous modeling languages such as Estrel
- Emitting and awaiting signals
- Temporal constructs such as triggers, watchdogs, and temporal loops
- Hard and soft interrupts and trap-exits
- Software adapters
- Interrupt handlers in high level languages

- Real-time systems
  - Hard real-time systems vs soft real-time systems
  - Timeliness
  - Time synchronization/scheduling
  - Prioritization
  - Latency
  - Compute jitter

- Memory management [cross-reference: FPL/Language Translation and Execution]
  - Mapping programming construct (variable) to a memory location
  - Shared memory
  - Manual memory management
  - Garbage collection

- Illustrative Learning Outcomes
  1. Design and develop software to interact with and control hardware [Creating]
  2. Design methods for real-time systems [Creating]
  3. Evaluate real-time scheduling and schedulability analysis [Analyzing]
  4. Evaluate formal specification and verification of timing constraints and properties [Analyzing]
FPL/FPL and SEP
[Non-core]

- Topics
  - Impact of English-centric programming languages
  - Impact of syntax on accessibility
  - Epistemology of terms such as “class”, “master”, “slave” in programming languages

- Illustrative Learning Outcomes
  1. Consciously design programming languages to be inclusive and non-offensive [Applying]

List of Professional Dispositions Appropriate for this KA

- **Professional**: Students must demonstrate the highest professional standards when using programming languages and formal methods to build safe systems which are fit for purpose.
- **Inventive**: Programming and approaches to formal proofs is inherently a creative process, students must demonstration innovative approaches to problem solving.
- **Meticulous**: Attention to detail is essential when using programming languages and applying formal methods.
- ** Responsible**: Programmers are responsible for anticipating all forms of user input and system behavior and to design solutions that address each one.
- ** Perseverance**: Students must demonstrate perseverance since the correct approach is not always self-evident and a process of refinement may be necessary to reach the solution.
- ** Accountable**: Students are accountable for their choices regarding how a problem is solved.

Math needed and wanted

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).

Shared Concepts

- **FPL/Event-Driven and Reactive Programming** overlaps with **AL/Automata and Complexity**
- **FPL/Distribution, Concurrency and Parallelism** overlaps with **PD/Parallel and Distributed Computing, SF/Parallelism**
- **FPL/Program Representation** overlaps with **AR/Machine Level Representation of Data**
- **FPL/Language Translation and Execution** overlaps with **AR/Memory System Architecture and Organization, OS/Memory Management, SEC/Foundational Security**
- **FPL/Compiler Semantic Analysis** overlaps with **SDF/Fundamental Programming Constructs**
- **FPL/Code Generation** overlaps with **AR/Assembly Level Machine Organization, AR/Memory System Organization and Architecture, OS/Memory Management**
- **FPL/Formal Semantics** overlaps with **SE/Formal Methods**
- **FPL/Formal Development Methodologies** overlaps with **SE/Formal Methods**
- **FPL/Program Analysis and Analyzers** overlaps with **PD/Parallelism Fundamentals**

**Possible Packaging of Courses**

**Introductory Course** to include the following:

- **FPL/Object-Oriented Programming** 5 CS Core hours, 3 KA Core hours
- **FPL/Functional Programming** 4 CS Core hours, 4 KA Core hours
- **FPL/Logic Programming** 3 CS Core hours
- **FPL/Event-Driven and Reactive Programming** 2 CS Core hours, 2 KA Core hours
- **FPL/Scripting** 2 CS Core hours
- **FPL/Parallel and Distributed Computing** 3 CS Core hours, 2 KA Core hours
- **FPL/Type Systems** 3 CS Core hours, 4 KA Core hours
- **FPL/Language Translation and Execution** 4 CS Core hours
- **FPL/Program Representation** 3 KA Core hours
- **FPL/Advance Programming Constructs** 4 Non-core hours
- **FPL/Hardware Interface** 2 Non-core hours
- **FPL/FPL and SEP** 1 Non-Core hour

Pre-requisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar

**Advanced Course** to include the following:

- **FPL/Language Translation and Execution** 3 Non-core hours
- **FPL/Syntax Analysis** 3 Non-core hours
- **FPL/Type Systems** 3 Non-core hours
- **FPL Compiler Semantic Analysis** 5 Non-core hours
- **FPL/Program Analyzers** 5 Non-core hours
- **FPL/Code Generation** 5 Non-core hours
Pre-requisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar

- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).

- Introductory course.

- Programming proficiency in programming concepts such as:
  - type declarations such as basic data types, records, indexed data elements such as arrays and vectors, and class/subclass declarations, types of variables,
  - scope rules of variables,
  - selection and iteration concepts, function and procedure calls, methods, object creation

- Data Structure concepts such as:
  - abstract data types, sequence and string, stack, queues, trees, dictionaries
  - pointer-based data structures such as linked lists, trees and shared memory locations
  - Hashing and hash tables

- System Fundamentals and Computer Architecture concepts such as:
  - Digital circuits design, clocks, bus
  - registers, cache, RAM and secondary memory
  - CPU and GPU

- Basic knowledge of Operating System concepts such as:
  - Interrupts, threads and interrupt-based/thread-based programming
  - Scheduling, including prioritization
  - Memory fragmentation
  - Latency

**Subcommittee**

Chair: Michael Oudshoorn, High Point University, NC, USA

Subcommittee members

- Annette Bieniusa, TU Kaiserslautern, Germany
- Alan Dearle, University of St. Andrews, Scotland
- Drijesh Dongol, University of Surrey, UK
Michelle Kuttel, University of Cape Town, South Africa
Doug Lea, State University of New York at Oswego, NY, USA
James Noble, Victoria University of Wellington, New Zealand
Mark Marron, Microsoft Research, WA, USA
Peter-Michael Osera, Grinnell College, IA, USA
Michelle Mills Strout, University of Arizona, AZ, USA

Other contributors:
- Anonymous reviewers