Programming Languages Design Workshop

Vincenzo Ciancia

April 3, 2025

Contents

Chapter 1: Programming Language Design and Implementation

Section 1: What is a Programming Language?

- A formal system for instructing computers
- Consists of syntax and semantics
- Bridge between human thought and machine execution
- Examples: Python, Java, C++, JavaScript, FSharp, Haskell, Microsoft Word, Excel, Ableton Live, Mathematica, LaTeX, HTML, CSS, SQL, …
- Everything is a formal language!

Formal system for instructing computers

A programming language is a formal system for instructing computers to perform specific tasks. It provides a structured way to communicate with machines using a well-defined set of rules and conventions. These languages enable humans to express complex algorithms and computations in a way that computers can understand and execute.

Syntax and semantics

The *syntax* and *semantics* of a programming language form its core components. Syntax refers to the set of rules that define how programs are written, including grammar, punctuation, and structure. For example, in Python, the syntax for arithmetic expressions follows specific rules:

- Operators must be placed between operands: `2 + 3` (valid) vs `+ 2 3` (invalid)
- Parentheses must be balanced: `(2 + 3) * 4` (valid) vs `(2 + 3 * 4` (invalid)
- Operator precedence determines evaluation order: `2 + 3 * 4` evaluates to 14, not 20

Semantics, on the other hand, determines the meaning of these syntactically correct programs and how they behave when executed.

Here's an example of a semantic valuation function for arithmetic expressions:

> `V` is the semantic evaluation function that maps syntactic expressions to their computed values

- Define valuation function `V(expr)`:
  - Input: `expr` (arithmetic expression)
  - Output: integer result
- Handle different expression types:
  - If `expr` is a number:
    - Return the number's value
  - If `expr` is of form `(a + b)`:
    - If `type(V(a))` and `type(V(b))` are numeric: Return `V(a) + V(b)`
    - Else: raise `TypeError`
  - If `expr` is of form `(a - b)`:
    - If `type(V(a))` and `type(V(b))` are numeric: Return `V(a) - V(b)`
    - Else: raise `TypeError`
  - If `expr` is of form `(a * b)`:
    - If `type(V(a))` and `type(V(b))` are numeric: Return `V(a) * V(b)`
    - Else: raise `TypeError`
  - If `expr` is of form `(a / b)`:
    - If `type(V(a))` and `type(V(b))` are numeric:
      - If `V(b) == 0`: raise `DivisionByZeroError`
      - Else: Return `V(a) / V(b)`
    - Else: raise `TypeError`

4. Handle operator precedence:
   - Multiplication and division have higher precedence than addition and subtraction; this is usually decided by the grammar of the language and the parser

- Parentheses override default precedence; program semantics is responsible for this

Example evaluation: Let's evaluate the expression `(2 + 3) * 4` using our valuation function:

1. Parse the expression into its components:
   - Outer operation: `*`
   - Left operand: `(2 + 3)`
   - Right operand: `4`
2. Evaluate the left operand `(2 + 3)`:
   - Parse the sub-expression:
     - Operation: `+`
     - Left operand: `2`
     - Right operand: `3`
   - Apply the addition rule:
     - `V(2)` returns 2
     - `V(3)` returns 3
     - `V(2 + 3)` returns `2 + 3 = 5`
3. Evaluate the right operand `4`:
   - `V(4)` returns 4
4. Apply the multiplication rule:
   - `V((2 + 3) * 4)` returns `5 * 4 = 20`

Final result: 20

Bridge between human thought and machine execution

Programming languages serve as a crucial bridge between human thought and machine execution. They allow developers to express abstract ideas and problem-solving strategies in a format that can be translated into machine code. This translation enables computers to perform complex operations and solve real-world problems efficiently.

The world of programming languages is vast and diverse, encompassing general-purpose languages like Python and Java, domain-specific languages like SQL and HTML, and even languages embedded in applications like Microsoft Word macros and Excel formulas. This diversity demonstrates how programming languages have become fundamental tools in virtually every aspect of modern computing.

Everything is a formal language

The concept of formal languages extends beyond traditional programming languages. Everything from mathematical notation to musical scores are formal languages, as they all provide structured systems for expressing and communicating complex ideas in a **machine-executable** format.

Most computer programs have a notion of "document" or "project", which is a collection of files that are used to create a larger work. These documents are essentially formal languages, as they are structured systems for expressing and communicating complex ideas. The user interface of a program helps building these documents (the syntax) and the program itself interprets these documents (the semantics).

This broader perspective helps us understand the fundamental role of formal systems in human-computer interaction and information processing.

Programming language design and the AI era

The advent of artificial intelligence is reshaping the landscape of programming language design in profound ways. As AI systems become more sophisticated, they're influencing both how we design languages and how we interact with them.

Let's first establish a key point: AI enables a conversational programming interface does not mean "no code". Rather, AI can help creating a computer program, which however the human must understand and eventually modify. This is needed to create and execute *unambiguous instructions*.

How can anyone (AI or human) ever prove that it has executed a task correctly, if the task is not well defined?

As an example, the reader can consider ComfyUI, which uses a graph-based, low-code, but still completely formal language to create and execute complex generative AI tasks. See https://www.comfy.org/

Indeed, AI is nevertheless changing the way we program, and therefore, will inevitably also change the landscape of programming language design. Among the things that are changing, we find:

- **AI-Assisted Development**:
  - AI-powered code completion and suggestion tools are becoming integral to modern IDEs
  - AI can help identify potential bugs and suggest optimizations
  - Natural language processing enables **conversational** programming interfaces
- **AI-Driven Language Evolution**:
  - AI can analyze code patterns to suggest language improvements
  - Automated testing of language features and syntax changes
  - AI-assisted language standardization and documentation
- **Future Directions**:
  - Integration of natural language processing into language design
  - **Development of AI-specific type systems and semantics**
  - Creation of languages that can evolve and adapt autonomously <– this was actually suggested by the DeepSeek AI.

As we move forward, the relationship between AI and programming language design will continue to deepen, creating new opportunities and challenges for language designers and developers alike.

But the most fundamental, key consideration is that languages need to be designed for both human and AI comprehension. They serve, and will always serve, as a bridge between human thought and machine execution.

Mathematics, as a prime example of a formal language, has been invented by humans, but it has always been used by both humans and machines; it is a universal language to express and reason about complex ideas.

So will be (more and more declarative, more and more high-level) programming languages, as they will be used by both humans and AI systems to formalise complex processes and avoid ambiguity, which is the key issue for both human-driven and ai-driven process design and execution.

Section 2: Key Design Considerations

- Expressiveness and power
- Imperative or Functional?
- Scoping rules
- Memory management
- Error handling mechanisms

Expressiveness and Power

A language's expressiveness refers to its ability to concisely represent complex ideas and operations. More expressive languages allow developers to achieve more with less code, although this sometimes comes at the cost of readability and maintainability. The power of a language is closely related to its expressiveness, but also includes its computational capabilities and the range of problems it can effectively solve. Modern languages like Python strike a balance between expressiveness and clarity.

Imperative, Functional, Object-Oriented, Declarative, Reactive?

Programming languages can be designed around different paradigms, each with its own strengths and trade-offs. The imperative paradigm focuses on explicit control flow and state changes, while functional programming emphasizes immutability and pure functions. Object-oriented programming organizes code around objects and their interactions, and declarative programming focuses on what to achieve rather than how to achieve it. Reactive (a.k.a. "dataflow") programming deals with data streams and change propagation, making it particularly suitable for modern user interfaces.

The choice of paradigm depends on the problem domain, with many modern languages supporting multiple paradigms to provide flexibility in solving different types of problems.

Application domains

Application domains

Programming languages are often designed with specific application domains in mind, tailoring their features and capabilities to particular problem spaces. Some notable examples include:

- **Web Development**:
  - JavaScript/TypeScript: The backbone of modern web development, enabling interactive front-end experiences and server-side applications
  - Svelte: A modern framework for building efficient web applications with a compiler-based approach
  - PHP: Widely used for server-side web development and content management systems
- **Scientific Computing**:
  - Python: With libraries like NumPy, SciPy, and Pandas, it's become a dominant language for data analysis and scientific computing
  - MATLAB: Specialized for numerical computing and matrix operations
  - R: Designed for statistical computing and data visualization
- **AI and Machine Learning**:
  - Python: The de facto language for AI/ML with frameworks like TensorFlow and PyTorch
  - Julia: Gaining popularity for high-performance scientific computing and machine learning
- **Music and Multimedia**:
  - Max/MSP: A visual programming language for music and multimedia
  - Pure Data: An open-source alternative for real-time audio and video processing
  - SuperCollider: A language for audio synthesis and algorithmic composition
- **Mathematical Computing**:
  - Mathematica: A powerful system for symbolic mathematics and technical computing
  - Maple: Another computer algebra system for mathematical problem solving
- **Systems Programming**:
  - C/C++: The foundation for operating systems and performance-critical applications
  - Rust: A modern systems programming language focusing on safety and concurrency
- **Mobile Development**:
  - Swift: Apple's language for iOS and macOS development
  - Kotlin: The preferred language for Android development

Each domain-specific language or framework brings specialized features and abstractions that make it particularly suited for its target application area, while general-purpose languages like Python and Java find use across multiple domains.

Scoping Rules

Scoping rules determine how and where variables and functions are accessible within a program. Lexical (static) scoping, used in most modern languages, determines variable visibility based on the program's structure. Dynamic scoping, less common today, but used in operating system shells, determines visibility based on the program's execution state. Proper scoping rules are crucial for managing complexity, preventing naming conflicts, and supporting modular programming. Python's scoping rules, with their global, nonlocal, and local scopes, provide a good example of how scoping can be implemented effectively.

Memory Management

Memory management is a critical aspect of language design that affects both performance and developer productivity. Languages can be manual (like C), garbage-collected (like Java and Python), or use reference counting (like Python and Swift). The choice of memory management strategy impacts the language's runtime performance, memory safety,

and the complexity of writing correct programs. Modern languages often provide automatic memory management while allowing manual control when needed for performance optimization.

Error Handling Mechanisms

Error handling is essential for building robust and reliable software. Languages implement various error handling strategies, including exceptions (like in Python and Java), return codes (like in C), and monadic error handling (like in Haskell). The choice of error handling mechanism affects how easily developers can write correct code and how gracefully programs can recover from unexpected situations. Python's exception handling system (similarly to other object-orientedlanguages), with its try/except blocks and exception hierarchy, is widely regarded as both powerful and intuitive.

Section 3: Language Paradigms

- Imperative (C, Python)
- Object-Oriented (Java, C#)
- Functional (Haskell, Lisp)
- Logical (Prolog)
- Declarative (SQL, VoxLogicA)
- Reactive (React, Svelte, Vue)

Imperative (C, Python)

Imperative programming is one of the most common programming paradigms, focusing on describing how a program operates through a sequence of statements that change a program's state. Languages like C and Python follow this approach, where developers explicitly specify the steps needed to achieve a result. This paradigm is particularly useful for tasks that require precise control over program flow and memory management. However, it can lead to complex code when dealing with large systems, as the focus on state changes can make reasoning about program behavior more challenging.

```
// Pseudo-code example of imperative programming:
// Calculating factorial

function factorial(n):
    // Initialize a variable to store the result
    result = 1

    // Use a loop to iteratively calculate factorial
    for i from 1 to n:
        result = result * i

    // Return the final calculated value
    return result

// Example usage
number = 5
factorialOfNumber = factorial(number)
print("Factorial of", number, "is", factorialOfNumber)
```

Object-Oriented (Java, C#)

Object-oriented programming (OOP) organizes software design around data, or objects, rather than functions and logic. Languages like Java and C# are built around this paradigm, which emphasizes concepts like encapsulation, inheritance, and polymorphism. OOP is particularly effective for modeling real-world systems and managing complex software

projects through modular design. However, it can introduce complexity through deep inheritance hierarchies and can sometimes lead to over-engineering of simple problems.

```
// Pseudo-code example of object-oriented programming with subclassing

class Animal:
    // Base class with common attributes and methods
    name: String
    age: Integer

    constructor(name, age):
        this.name = name
        this.age = age

    method makeSound():
        print("Generic animal sound")

    method describe():
        print("Name:", this.name, "Age:", this.age)

class Dog(Animal):
    // Subclass inheriting from Animal
    breed: String

    constructor(name, age, breed):
        // Call parent constructor
        super(name, age)
        this.breed = breed

    // Override parent method
    method makeSound():
        print("Woof! Woof!")

    method describe():
        // Extended description specific to Dog
        super.describe()
        print("Breed:", this.breed)

// Example usage
myDog = Dog("Buddy", 3, "Labrador")
myDog.describe()    // Prints dog's details
myDog.makeSound()   // Prints dog-specific sound
```

Functional (Haskell, Lisp)

Functional programming treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. Languages like Haskell and Lisp embody this paradigm, emphasizing pure functions, immutability, and higher-order functions. This approach can lead to more predictable and testable code, as functions with no side effects are easier to reason about. However, it can be challenging for developers accustomed to imperative programming and may require a different way of thinking about problem-solving.

```
// Pseudo-code example of functional programming with immutability and higher-order functions

// Pure function: deterministic and without side effects
let square(x) -> Number:
```

```
    return x * x

// Higher-order function: transforms a list using a given function
let map(list, transform) -> List:
    match list:
        | Empty -> []
        | [Head | Tail] ->
            [transform(Head)] ++ map(Tail, transform)

// Immutable data and functional transformation
let numbers = [1, 2, 3, 4, 5]
let squaredNumbers = map(numbers, square)

// Function composition as a first-class operation
let compose(f, g) -> Function:
    return λx: f(g(x))

// Recursive function using tail-call optimization
let factorial(n, accumulator = 1) -> Number:
    match n:
        | 0 -> accumulator
        | _ -> factorial(n - 1, n * accumulator)

// Functional application demonstrating composition
let result = compose(square, factorial)(3)  // Computes factorial then squares
```

Logical (Prolog)

Logical programming is based on formal logic, where programs consist of a set of facts and rules. Prolog is the most well-known language in this paradigm, where computation is performed by querying these facts and rules. This approach is particularly useful for problems involving symbolic computation, artificial intelligence, and knowledge representation. However, it can be less efficient for general-purpose programming tasks and may require a different mindset from traditional programming approaches.

```
// Pseudo-code example of logical programming with Prolog

// Facts about family relationships
// Facts about family relationships
parent(john, mary).
parent(john, tom).
parent(mary, alice).

// Rules for determining family relationships
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
```

Declarative (SQL)

Declarative programming focuses on what the program should accomplish rather than how to achieve it. Languages like SQL follow this paradigm, where developers specify the desired results rather than the step-by-step process to achieve them. This approach can lead to more concise and maintainable code, especially for data manipulation and querying tasks. However, it may provide less control over the underlying implementation details and can be less suitable for tasks requiring precise control over program execution. This does *not* mean that declarative programming is “less efficient”; quite the opposite, an SQL *query execution plan* can be highly optimised by the database engine,

and be way more efficient than handwritten code.

See also the voxlogica project: http://www.voxlogica.org

```
-- Pseudo-code example of declarative programming with SQL

-- Define a table for employees
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2),
    hire_date DATE
);

// Insert sample records
INSERT INTO employees (name, department, salary, hire_date) VALUES
    ('John Doe', 'Engineering', 75000.00, '2022-01-15'),
    ('Jane Smith', 'Marketing', 65000.50, '2021-11-03');

// Example query to demonstrate declarative programming
SELECT name, department, salary
FROM employees
WHERE department = 'Engineering'
ORDER BY salary DESC;
```

Reactive (React, Svelte, Vue)

Reactive programming is a paradigm that focuses on data flows and the propagation of change. Languages and frameworks like React, Svelte, and Vue implement this approach, where the UI automatically updates in response to changes in the underlying data. This paradigm is particularly effective for building modern web applications and user interfaces, as it simplifies the management of complex state changes and UI updates. Reactive programming often uses concepts like observables, streams, and declarative bindings to create responsive and efficient applications. While powerful for UI development, it can introduce complexity in managing state across large applications and may require specific architectural patterns to maintain code organization.

```
<script>
    let count = 0;
    $: doubled = count * 2;

    function increment() {
        count += 1;
    }
</script>

<button on:click={increment}>
    Count: {count} (Doubled: {doubled})
</button>
```

Section 4: Implementation Concerns

- Compilation vs. Interpretation
- Memory management
- Garbage collection
- Runtime environment
- Performance optimization

Compilation vs. Interpretation

The choice between compilation and interpretation is a fundamental decision in language implementation. Compiled languages (like C and Rust) translate source code into machine code before execution, resulting in faster runtime performance but requiring a separate compilation step. Interpreted languages (like Python and JavaScript) execute code directly through an interpreter, enabling faster development cycles and platform independence but typically with slower execution speeds. Many modern languages (like Java and C#) use a hybrid approach, compiling to an intermediate bytecode that is then interpreted or just-in-time (JIT) compiled, offering a balance between performance and flexibility.

Memory Management

Memory management is crucial for both performance and safety in programming languages. Manual memory management, as seen in C and C++, gives developers fine-grained control but increases the risk of memory leaks and segmentation faults. Automatic memory management through garbage collection, used in Java and Python, simplifies development by automatically reclaiming unused memory but can introduce unpredictable pauses in program execution. Reference counting, another automatic approach used in Python and Swift, provides more predictable memory reclamation but can struggle with circular references.

Garbage Collection

Garbage collection is a critical component of automatic memory management, responsible for reclaiming unused memory. Different garbage collection algorithms (like mark-and-sweep, generational, and reference counting) offer various trade-offs between throughput, latency, and memory efficiency. Languages like Java and C# use sophisticated garbage collectors that can be tuned for specific workloads, while languages like Go use concurrent garbage collection to minimize pause times. The choice of garbage collection strategy significantly impacts a language's performance characteristics and suitability for different application domains, from real-time systems to long-running server applications.

Runtime Environment

The runtime environment provides essential services for program execution, including memory management, type checking, and security features. Managed runtime environments, like the Java Virtual Machine (JVM) and .NET Common Language Runtime (CLR), offer platform independence and additional safety features but introduce overhead. Native runtime environments, as used in C and Rust, provide direct access to system resources but require more careful programming. Modern runtime environments often include features like just-in-time compilation, dynamic optimization, and extensive standard libraries to improve performance and developer productivity.

A special mention goes to HTML5, the latest version of the Hypertext Markup Language, which provides a runtime environment for web applications. It includes features like local storage, web workers, and web sockets, enabling developers to create more powerful and responsive web applications. HTML5 also supports offline applications, multimedia, and advanced graphics, making it a versatile runtime environment for modern web development and also for desktop applications, thanks to frameworks such as Electron or Tauri.

Performance Optimization

Performance optimization in programming languages involves various techniques at different levels of the implementation stack. Compiler optimizations, such as inlining, loop unrolling, and dead code elimination, can significantly improve execution speed without changing the source code. Runtime optimizations, like JIT compilation and adaptive optimization, can dynamically improve performance based on actual usage patterns. Language designers must balance optimization opportunities with compilation speed, memory usage, and the complexity of the implementation. Modern languages often provide profiling tools and optimization flags to help developers identify and address performance bottlenecks in their code.

Section 5: Syntax and Semantics

- Syntax: Rules for valid program structure
- Semantics: Meaning of valid programs

- Abstract Syntax Trees (AST)
- Context-free grammars
- Parsing techniques

Syntax: Rules for valid program structure

Syntax defines the formal rules that govern how programs are written in a language. It specifies the valid combinations of symbols, keywords, and structures that constitute a well-formed program. Syntax rules are typically defined using formal grammars, such as context-free grammars, which provide a precise specification of the language's structure. These rules determine everything from basic elements like variable declarations and function definitions to more complex constructs like control flow statements and class definitions. While syntax rules are often rigid, many modern languages include syntactic sugar - features that make the language more expressive and easier to use without changing its fundamental capabilities.

Semantics: Meaning of valid programs

Semantics defines the meaning of syntactically valid programs - what happens when a program is executed. While syntax determines whether a program is well-formed, semantics determines what the program actually does. There are different approaches to defining semantics, including operational semantics (describing how a program executes), denotational semantics (mapping programs to mathematical objects), and axiomatic semantics (describing program behavior through logical assertions). Understanding semantics is crucial for writing correct programs, as it helps developers predict how their code will behave and ensures that programs produce the intended results.

Context-free grammars

Context-free grammars (CFGs) are formal systems used to describe the syntax of programming languages. They consist of a set of production rules that define how symbols in the language can be combined. CFGs are particularly useful because they can be parsed efficiently and provide a clear, mathematical foundation for language design. They consist of terminal symbols (the actual characters/tokens in the language), non-terminal symbols (abstract syntactic categories), and production rules (how non-terminals can be expanded into sequences of terminals and other non-terminals). While CFGs are powerful, some language features (like indentation-sensitive syntax in Python) require extensions to the basic CFG model.

Parsing techniques

Parsing is the process of analyzing a sequence of tokens to determine its grammatical structure according to a given formal grammar. There are several parsing techniques used in language implementation, including top-down parsing (like recursive descent) and bottom-up parsing (like LR parsing). Top-down parsers build the parse tree from the root down to the leaves, while bottom-up parsers work from the leaves up to the root. Modern language implementations often use parser generators (like ANTLR or Yacc) that can automatically generate efficient parsers from a grammar specification. The choice of parsing technique can affect both the performance of the language implementation and the complexity of the grammar that can be supported.

Abstract Syntax Trees (AST)

An Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of source code. It abstracts away from the concrete syntax (like punctuation and specific keywords) and focuses on the essential structure of the program. ASTs are used extensively in compilers and interpreters, as they provide a convenient intermediate representation for performing various transformations and analyses. Each node in the tree represents a construct occurring in the source code, with the children of the node representing its components. ASTs are particularly useful for implementing features like syntax highlighting, code formatting, and static analysis tools.

Example of a parse tree:

```
   Expr
  /  |  \
```

```
  3   +   Expr
        /   |   \
       (    Expr   )
           /  |  \
          4   *   5
```

Example of an AST:

```
  (+)
 /   \
3     (*)
       /   \
      4     5
```

Normally there is a translation step between the parse tree and the AST.

Section 6: Type Systems

- Static vs. Dynamic typing
- Strong vs. Weak typing
- Type inference
- Type safety
- Generics and polymorphism
- Union and intersection types

Static vs. Dynamic typing

Static typing requires variable types to be explicitly declared and checked at compile time, while dynamic typing determines types at runtime. Statically typed languages like Java and C++ provide early error detection and better tooling support, but can be more verbose. Dynamically typed languages like Python and JavaScript offer more flexibility and faster development cycles, but may lead to runtime errors that could have been caught earlier. The choice between static and dynamic typing often depends on the project's requirements and the development team's preferences.

```java
// Example of static typing in Java
public class Main {
    public static void main(String[] args) {
        // Explicit type declaration required
        int number = 42;          // 'number' is statically typed as int
        String text = "Hello";    // 'text' is statically typed as String

        // Type checking at compile time
        // number = "Hello";      // This would cause a compile-time error
        // text = 42;             // This would also cause a compile-time error

        System.out.println(number + " " + text);
    }
}
```

```python
# Example of dynamic typing in Python
def main():
    # No explicit type declaration
    value = 42          # 'value' is initially an integer
    print(type(value))  # Output: <class 'int'>

    value = "Hello"     # 'value' is now a string
    print(type(value))  # Output: <class 'str'>
```

```
    value = 3.14        # 'value' is now a float
    print(type(value))  # Output: <class 'float'>

if __name__ == "__main__":
    main()
```

Strong vs. Weak typing

Strong typing enforces strict type rules and prevents implicit type conversions, while weak typing allows more flexible type handling. Strongly typed languages like CSharp, Java, FSharp and Haskell help prevent type-related bugs and make code more predictable, but can require more explicit type conversion code. Weakly typed languages like JavaScript and PHP can be more convenient for quick scripting but may lead to unexpected behavior due to implicit type coercion. The strength of a type system affects how easily developers can reason about code behavior and maintain type safety.

A modern trend in language design is to delegate strong type checking to the editor, which highlights type errors at compile time. This is in contrast to languages like C# or Java, where the compiler performs the type checking. A poorly typed program will still run, which may be a balanced choice for development. Note that industrial programming environments also require programmers to write unit tests, which can help catch errors, and also in that case, programs can be executed (e.g. for debugging) also when tests are not passing. Typing-in-the-editor is somehow aligned to testing in this respect.

Type inference

Type inference allows the compiler to automatically deduce types based on context, reducing the need for explicit type annotations. Languages like Haskell use sophisticated type inference systems that can often determine types without explicit declarations. This feature combines the benefits of static typing with the convenience of dynamic typing, making code more concise while maintaining type safety. However, complex type inference can sometimes make error messages harder to understand and may require explicit type annotations in certain cases.

Generics and polymorphism

Generics allow writing code that works with multiple types while maintaining type safety, while polymorphism enables objects to take on multiple forms. Languages like Java and C#, and functional languages like OCaml, Haskell, FSharp, use generics to create reusable, type-safe components, while polymorphism allows for more flexible and extensible designs. These features help reduce code duplication and create more maintainable systems, but can introduce complexity in understanding and implementing type hierarchies and generic constraints.

Union and intersection types

Union types allow a value to be one of several types, while intersection types combine multiple types into one. Languages like TypeScript and Python's type system support these advanced type features, enabling more precise type definitions and better code documentation. Union types are particularly useful for handling multiple possible input types, while intersection types can describe objects that must satisfy multiple interfaces. These features enhance type safety and expressiveness but can increase the complexity of type checking and inference. Nevertheless, the expressivity of such type systems provides a strong foundational alternative to object-oriented programming, often resulting in more maintainable and scalable systems.

Section 7: Modern Language Trends

- Concurrency and parallelism
- Functional programming features
- Type inference and safety
- Domain-specific languages
- Interoperability with other languages

Concurrency and Parallelism

Modern programming languages are increasingly focusing on concurrency and parallelism to handle the demands of multi-core processors and distributed systems. Concurrency allows multiple tasks to make progress simultaneously, while parallelism enables the execution of multiple tasks at the same time. Languages like Go and Rust have built-in support for concurrent programming through goroutines and async/await patterns, respectively. These features help developers write efficient, scalable applications that can take full advantage of modern hardware. However, managing shared state and avoiding race conditions remain significant challenges in concurrent programming, requiring careful design and the use of synchronization primitives.

Asynchronous programming

Asynchronous programming has become a crucial feature in modern languages to handle I/O-bound operations and improve application responsiveness. Languages like JavaScript, Python, and C# provide async/await syntax to write asynchronous code that appears synchronous, making it easier to reason about and maintain. This approach allows programs to perform non-blocking operations, such as network requests or file I/O, without freezing the user interface or wasting CPU cycles, and at the same time avoiding the pitfalls and complexity of parallel semantics. Asynchronous programming models help create more efficient and responsive applications, particularly in web development and server-side programming, where handling multiple concurrent requests is essential. However, managing asynchronous code flow and error handling can introduce complexity, requiring careful design and the use of appropriate patterns and libraries.

Functional Programming Features

Functional programming concepts are being incorporated into many mainstream languages, bringing benefits like immutability, pure functions, and higher-order functions. Languages like Python, JavaScript, and C# now support features such as structured types, lambda expressions, map/filter/reduce operations, and pattern matching. These features enable developers to write more concise, declarative code and can lead to programs that are easier to reason about and test. The adoption of functional programming principles also promotes better code organization and can help reduce side effects, leading to more predictable program behavior.

Type Inference and Safety

Type systems in modern languages are becoming more sophisticated, with features like type inference reducing the need for explicit type annotations while maintaining type safety. Languages like TypeScript and Swift use type inference to automatically determine types based on context, making code more concise without sacrificing safety. Advanced type systems also include features like algebraic data types, type classes, and dependent types, which can catch more errors at compile time. These developments help create more robust software by preventing common runtime errors and making code more self-documenting.

Domain-Specific Languages

Domain-specific languages (DSLs) are specialized programming languages designed for particular application domains. They offer higher-level abstractions and more expressive syntax for specific tasks, such as SQL for database queries or HTML for web page structure. Modern language design often includes facilities such as Algebraic Data Types (ADTs) or Monadic syntax (see also computation expressions in FSharp), for creating embedded DSLs within general-purpose languages, allowing developers to create custom syntax and semantics tailored to their specific needs. This approach combines the power of general-purpose programming with the convenience and expressiveness of domain-specific solutions.

Interoperability with Other Languages

As software systems become more complex, the ability to interoperate with code written in other languages has become increasingly important. Many modern languages provide foreign function interfaces (FFIs) or support for calling functions from other languages. For example, Python's C API allows integration with C/C++ code, while WebAssembly enables running code from multiple languages in web browsers. This interoperability allows developers to leverage

existing libraries and frameworks while still using their preferred language for new development, promoting code reuse and reducing development time.

Chapter 2: Types and Pattern Matching in Python

Section 1: Python's Type System

What are Type Annotations?

Python's type system allows developers to add optional type hints to variables, function parameters, and return values. These annotations help catch errors early, improve code documentation, and enhance IDE support without changing the runtime behavior of the code.

```python
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

Type annotations are part of the Python Enhancement Proposal (PEP) 484 and have been continuously improved in subsequent PEPs. They provide a way to make Python code more robust through static type checking, although Python remains dynamically typed at runtime.

Basic Types in Python

Python provides several built-in types for annotations:

- **Primitive types**: `int`, `float`, `bool`, `str`
- **Collection types**: `list`, `tuple`, `dict`, `set`

Example from `types_and_matching.py`:

```python
my_tuple: tuple[int, str, float] = (1, "hello", 1.0)
my_list: list[int] = [1, 2, 3, 4, 5]
```

These annotations tell the type checker that `my_tuple` is a 3-element tuple containing an integer, a string, and a float, while `my_list` is a list containing only integers.

Generic Types

Generic types allow you to create reusable, type-safe components. In Python 3.12+, the syntax for generic classes uses square brackets:

```python
class Stack[T]:
    def __init__(self) -> None:
        self.items: list[T] = []

    def push(self, item: T) -> None:
        self.items.append(item)

    def pop(self) -> T | None:
        return self.items.pop() if self.items else None
```

In this example from our code: - `T` is a type parameter representing any type - `Stack[T]` is a generic class that can be specialized for specific types - `list[T]` indicates a list containing elements of type `T`

To use this generic class:

```python
stack_1 = Stack[int]()  # A stack of integers
stack_1.push(1)  # Valid
stack_1.push("hello")  # Type error: expected int, got str
```

Union Types and Optional Values

Union types allow a variable to have multiple possible types, expressed using the `|` operator (introduced in Python 3.10):

```python
def process_data(data: int | str) -> None:
    # Can handle both integers and strings
    pass
```

From our example code:

```python
def pop(self) -> T | None:  # Use | for union types
    return self.items.pop() if self.items else None
```

This indicates that the `pop` method returns either a value of type `T` or `None` if the stack is empty.

Type Aliases

Type aliases help simplify complex type annotations:

```python
type JsonData = dict[str, int | str | list | dict]
type A = int | dict[str, A]  # Recursive type
```

In `types_and_matching.py`, we use type aliases for recursive types:

```python
type MyBaseList[T] = None | MyList[T]
type Expr = int | Sum
```

These aliases make the code more readable and allow for recursive type definitions.

Literal Types

Literal types restrict values to specific constants:

```python
y: Literal["hello", "world"] = "hello"  # Valid
z: Literal[1, 2, 3] = 9  # Type error
```

From our example code:

```python
@dataclass
class Human:
    name: str
    drivingLicense: Literal[True, False]
```

This constrains the `drivingLicense` attribute to be either `True` or `False` only.

Section 2: Structural Pattern Matching

Introduction to Pattern Matching

Introduced in Python 3.10, the `match` statement provides powerful pattern matching capabilities, similar to switch statements in other languages but with more expressive power:

```python
def describe(value):
    match value:
        case 0:
            return "Zero"
        case int(x) if x > 0:
            return "Positive integer"
        case str():
            return "String"
        case _:
            return "Something else"
```

Pattern matching allows for more concise and readable code, especially when dealing with complex data structures and multiple conditions.

Basic Patterns

Basic patterns match against simple values and types:

```
match x:
    case 0:
        print("Zero")
    case int():
        print("Integer")
    case str():
        print("String")
    case _:  # Wildcard pattern
        print("Something else")
```

The _ pattern is a wildcard that matches anything and is often used as a catch-all case.

Sequence Patterns

Sequence patterns match against sequence types like lists and tuples:

```
def process_lst(lst: list[int]) -> None:
    match lst:
        case []:
            print("List: empty")
        case [head, *tail]:
            print(f"List: head {head}, tail {tail}")
```

This example shows destructuring a list into its head (first element) and tail (remaining elements), demonstrating how pattern matching facilitates recursive list processing.

Class Patterns and Attribute Matching

Pattern matching works particularly well with dataclasses:

```
def greet(person: Person | str) -> None:
    match person:
        case Person(name="Alice", age=25):
            print("Hello Alice!")
        case Person(name=x, age=y):
            print(f"Hello {x} of age {y}!")
        case str():
            print(f"Hello {person}!")
```

This example shows matching against specific attribute values and binding attributes to variables.

Complex Pattern Matching Examples

Recursive Pattern Matching Pattern matching excels at handling recursive data structures:

```
def sum_list_2(lst: MyBaseList[int]) -> int:
    match lst:
        case None:
            return 0
        case MyList(head=x, tail=y):
            return x + sum_list_2(y)
```

This function processes a custom linked list structure using pattern matching to handle the base case (None) and recursive case elegantly.

Parsing Expressions Pattern matching can implement simple interpreters:

```
def eval_expr(expr: Expr) -> int:
    match expr:
        case int(x):
            return x
        case Sum(left=x, right=y):
            return eval_expr(x) + eval_expr(y)
```

This code evaluates a simple arithmetic expression tree, showing how pattern matching simplifies traversal of complex structures.

Section 3: Combining Types and Pattern Matching

Algebraic Data Types in Python

Python can implement algebraic data types (ADTs) using classes, dataclasses, and union types:

Sum Types (Tagged Unions) Sum types represent values that could be one of several alternatives:

```
@dataclass
class Human:
    name: str
    drivingLicense: Literal[True, False]

@dataclass
class Dog:
    name: str
    kind: DogKind
    colour: Literal["brown", "black", "white"]

type Record = Human | Dog
```

This example defines two distinct record types (`Human` and `Dog`) and a union type `Record` that can be either of them.

Pattern Matching with Sum Types Pattern matching works seamlessly with sum types:

```
def describe_person(person: Human | Dog) -> str:
    match person:
        case Human(name=name, drivingLicense=True):
            return f"Person {name} has a driving license"
        case Human(name=name, drivingLicense=False):
            return f"Person {name} does not have a driving license"
        case Dog(name=name, kind=kind, colour=colour):
            return f"Dog {name} is a {kind} and has a {colour} colour"
```

This function handles different record types with specific patterns for each case.

Product Types Product types represent combinations of values:

```
@dataclass
class Person:
    name: str
    age: int
```

A dataclass like `Person` is a product type, representing a combination of a string and an integer.

Type Checking and Pattern Matching

Type checkers like mypy can catch errors in pattern matching code:

```
def describe_person_2(person: Record) -> str:
    if isinstance(person, Human):
        return person.name + person.colour  # Type error: Human has no attribute 'colour'
    else:
        return person.name + person.kind
```

Type checking helps identify incorrect attribute access, which might otherwise lead to runtime errors.

Section 4: Applications and Best Practices

When to Use Type Annotations

Type annotations are particularly valuable in: - Large codebases with multiple developers - APIs and libraries that will be used by others - Performance-critical code where type-specific optimizations matter - Complex data processing pipelines

Domain-specific languages, interpreters, ADTs!!

When to Use Pattern Matching

Pattern matching excels at: - Processing complex recursive data structures - Implementing interpreters and compilers - Handling case-based logic with destructuring - Processing structured data like JSON or ASTs

Best Practices for Type Annotations

1. Be consistent with type annotations across your codebase
2. Use type aliases for complex or repetitive type expressions
3. Leverage tools like mypy, pyright, or pylance for static type checking
4. Balance between type precision and code readability
5. Document non-obvious type constraints with comments

Best Practices for Pattern Matching

1. Order cases from most specific to most general
2. Use the wildcard pattern (_) as the last case
3. Consider using guard clauses for complex conditions
4. Break complex pattern matching into smaller functions
5. Leverage destructuring to avoid redundant variable assignments

Exercises

- Consider the expression evaluator written in Lecture 01. Turn it into a parser that converts a string of the form "x op y op z …" separated by spaces into an AST in the sense of Lecture 02. Concatenate the new parser and the evaluation function that takes AST as input to define a mini-interpreter.
- Add the "==" boolean operator to the AST and the evaluator.

Additional Resources

- PEP 484 – Type Hints
- PEP 585 – Type Hinting Generics In Standard Collections
- PEP 604 – Allow writing union types as X | Y
- PEP 634 – Structural Pattern Matching: Specification
- PEP 636 – Structural Pattern Matching: Tutorial

- Mypy Type Checker Documentation
- Real Python: Python Type Checking
- Real Python: Structural Pattern Matching in Python

Chapter 3: Parsing and Mini-Interpreters

Section 1: Introduction to Context-Free Grammars and Parsing

Lark is a parsing toolkit that works with Context-Free Grammars (CFGs). Context-free grammars are a formal grammar formalism that can express the syntax of most programming languages and many natural language constructs.

Key Features of Lark's Parsing Approach

- **Multiple Parsing Algorithms**: Lark implements Earley and LALR(1) algorithms for parsing CFGs.
- **Ambiguity Support**: Unlike PEG parsers, Lark's Earley algorithm can handle ambiguous grammars gracefully.
- **Complete Language Coverage**: Lark can parse all context-free languages, making it capable of handling almost any programming language.
- **EBNF Notation**: Lark uses Extended Backus-Naur Form (EBNF) for grammar definition, which provides powerful and concise syntax.

Section 2: Working with Lark

Lark is a modern parsing library for Python that implements parsers for Context-Free Grammars with additional features. It provides a clean interface for defining grammars and working with parse trees.

Defining a Grammar in Lark

```
grammar = r"""
    ?expr: bin | mono
    mono: ground | paren
    paren: "(" expr ")"
    bin: mono op expr
    ground: NUMBER

    NUMBER: /[0-9]+/
    op: "+" | "-" | "*" | "/" | "%"

    %import common.WS
    %ignore WS
"""
```

Grammar Components Explained

- **Rules**: Non-terminals like `expr`, `mono`, and `bin` define the structure of the language.
- **Terminals**: Items like `NUMBER` or literals like `"("` match specific patterns in the input.
- **Special Rules**: The `?expr` rule is an anonymous rule that doesn't create a separate node in the parse tree.
- **Directives**: `%import common.WS` and `%ignore WS` handle whitespace elegantly.

Left-Associativity and Left Recursion

The grammar uses a left-recursive rule for binary operations to achieve left-associativity:

```
bin: expr OP mono ## left-associative
```

This approach ensures that expressions like `1 + 2 + 3` are evaluated from left to right as `(1 + 2) + 3`, which is the standard evaluation order for most arithmetic operations. Left recursion is traditionally challenging for many parser generators, but Lark can handle it efficiently, allowing for a more natural and intuitive grammar definition. For parser generators that cannot handle left recursion efficiently, there are transformations of the grammar that use only right recursion.

Creating a Parser and Parsing Input

```
from lark import Lark

# Create the Lark parser
parser = Lark(grammar, start="expr")

# Parse an input string
parse_tree = parser.parse("(1 + 2) - 3")

# Print the parse tree
print(parse_tree.pretty())
```

Section 3: Understanding Parse Trees

Lark generates parse trees that represent the structure of the input according to the grammar rules. Each node in the tree corresponds to a rule in the grammar or a terminal value.

Parse Tree Structure

For the expression `(1 + 2) - 3`, the parse tree looks like:

```
bin
+-- mono
|   +-- paren
|       +-- bin
|           +-- mono
|           |   +-- ground
|           |       +-- 1
|           +-- op
|           |   +-- +
|           +-- mono
|               +-- ground
|                   +-- 2
+-- op
|   +-- -
+-- mono
    +-- ground
        +-- 3
```

Navigating the Parse Tree

Parse trees can be navigated and inspected programmatically:

```
def print_tree(tree: ParseTree | Token):
    match tree:
        case Tree(data=data, children=children):
            print("Tree", data)
            for child in children:
                print_tree(child)
        case Token(type=type, value=value):
            print("Token", type, value)
```

Section 4: Pattern Matching with Parse Trees

Python 3.10's pattern matching feature works exceptionally well with parse trees, allowing for declarative tree traversal:

```
match parse_tree:
    case Tree(data='bin', children=[left, Tree(data='op', children=[Token(value=op)]), right]):
        print(f"Found binary operation: {op}")
        print(f"Left: {left}")
        print(f"Right: {right}")

    case Tree(data='mono', children=[Tree(data='ground', children=[Token(value=number)])]):
        print(f"Found number: {number}")

    case _:
        print("Unknown tree structure")
```

Section 5: Transforming Parse Trees to ASTs

Parse trees often contain more detail than needed for interpretation. We can define custom types and use pattern matching to convert parse trees into a more abstract syntax tree (AST):

```
from dataclasses import dataclass
from typing import Literal

# Define AST types for the expression language
type Op = Literal["+", "-", "*"]

@dataclass
class Number:
    value: int

@dataclass
class BinaryExpression:
    op: Op
    left: Expression
    right: Expression

type Expression = Number | BinaryExpression
```

This AST representation is cleaner and more suitable for evaluation than the raw parse tree.

Section 6: Converting Parse Trees to ASTs

With our AST structure defined, we can convert parse trees into ASTs using pattern matching:

```
def transform_parse_tree(tree: Tree) -> Expression:
    match tree:
        case Tree(data="mono", children=[subtree]):
            return transform_parse_tree(subtree)

        case Tree(data="ground", children=[Token(type="NUMBER", value=value)]):
            return Number(value=int(value))

        case Tree(data="paren", children=[subtree]):
            return transform_parse_tree(subtree)

        case Tree(
            data="bin",
            children=[
```

```
                left,
                Token(type="OP", value=op),
                right,
            ],
        ):
            return BinaryExpression(
                op=op,
                left=transform_parse_tree(left),
                right=transform_parse_tree(right),
            )

        case _:
            raise ValueError(f"Unexpected parse tree structure")
```

This approach provides much better type safety and makes the intent clearer than the Transformer class approach.

Section 7: Building a Mini-Interpreter

By combining parsing with evaluation, we can create a mini-interpreter:

1. **Parse**: Convert the input string into a parse tree
2. **Transform**: Convert the parse tree into an AST
3. **Evaluate**: Traverse the AST to compute a result

```
# Function to parse a string and return an AST
def parse_ast(expression: str) -> Expression:
    parse_tree = parser.parse(expression)
    return transform_parse_tree(parse_tree)

# Example
expression = "(1 + 2) - 3"
ast = parse_ast(expression)
print(ast)

# Evaluate function
def evaluate(ast: Expression) -> int:
    match ast:
        case Number(value):
            return value
        case BinaryExpression(op, left, right):
            left_value = evaluate(left)
            right_value = evaluate(right)
            match op:
                case "+":
                    return left_value + right_value
                case "-":
                    return left_value - right_value
                case "*":
                    return left_value * right_value
                case "/":
                    if right_value == 0:
                        raise ValueError("Division by zero")
                    return left_value // right_value
                case "%":
                    if right_value == 0:
```

```
                    raise ValueError("Division by zero")
                return left_value % right_value
```

The implementation handles division by zero by checking for zero divisors before performing division or modulo operations and raising a `ValueError` when detected.

REPL Implementation

The interpreter is equipped with a Read-Evaluate-Print Loop (REPL) that allows users to interactively input expressions and see their evaluated results:

```
def REPL():
    exit = False
    while not exit:
        expression = input("Enter an expression (exit to quit): ")
        if expression == "exit":
            exit = True
        else:
            try:
                print(evaluate_string(expression))
            except Exception as e:
                print(e)
```

This pattern provides an interactive environment for testing the interpreter, handling errors gracefully, and allowing users to experiment with different expressions.

Additional Resources

- Lark Documentation
- Context-Free Grammars (Wikipedia)
- EBNF Syntax (Wikipedia)
- Python Pattern Matching Documentation

Chapter 4: Semantic Domains and Environment-Based Interpreters

Section 1: Introduction to Semantic Domains

In programming language semantics, **semantic domains** are mathematical structures used to give meaning to syntactic constructs. They provide the foundation for defining the behavior of programs in a precise, mathematical way.

Key Semantic Domains in Programming Languages

- **Syntactic Domains**: Represent the structure of programs (tokens, parse trees, syntax trees).
- **Semantic Domains**: Represent the meaning of programs (values, environments, states).

The relationship between these domains is at the heart of language semantics:

```
Syntax -> Semantics
Program -> Meaning
```

Core Semantic Domains

In our mini-interpreter, we'll work with several fundamental semantic domains:

- **Expressible Values**: Values that can be produced by evaluating expressions
- **Denotable Values (DVal)**: Values that can be bound to identifiers in the environment
- **Memorizable Values (MVal)**: Values that can be stored in memory/state
- **Identifiers**: Names of constants, variables, functions, modules, etc.
- **Locations**: Memory addresses of variables
- **Environment**: Maps identifiers to denotable values, representing the binding context
- **State**: Maps memory locations to memorizable values, representing the program's memory

While these domains often overlap, they aren't necessarily identical. Understanding the differences is crucial for language design.

Side Effects and Pure Functions

A fundamental distinction in programming language semantics is between **pure** computations and those that produce **side effects**.

Pure Functions A **pure function** is a computation that: 1. Always produces the same output for the same input 2. Has no observable effects beyond computing its result

In mathematical terms, a pure function is simply a mapping from inputs to outputs, like mathematical functions (e.g., sin(x), log(x)).

```python
def add(x: int, y: int) -> int:
    return x + y  # Pure: same inputs always give same output
```

Side Effects A **side effect** is any observable change to the system state that occurs during computation, beyond returning a value. Common side effects include:

1. **Memory updates**: Modifying variables or data structures

   ```python
   x = 10  # Changes the program's state
   ```

   Q: in this context, what is a function that does **not** return always the same value for the same inputs?

2. **Input/Output operations**:

   ```python
   print("Hello")  # Affects the external world (terminal)
   ```

3. **File operations**:

```
with open("data.txt", "w") as f:
    f.write("data")  # Changes the file system
```

4. **Network operations**:

```
requests.get("https://example.com")  # Interacts with external systems
```

Memory Updates as Side Effects In our interpreter design, memory (state) updates are a primary form of side effect. When we update state:

```
def state_update(state: State, location: int, value: MVal) -> State:
    new_state = state.copy()
    new_state[location] = value
    return new_state
```

We're representing a change to the program's memory. In a real computer, this would modify memory cells directly. Our functional implementation returns a new state rather than modifying the existing one, but conceptually it represents the same side effect.

Side Effects in Programming Languages Languages differ in how they handle side effects:

- **Purely functional languages** (like Haskell) isolate side effects using type systems and monads
- **Imperative languages** (like C, Python) embrace side effects as their primary mechanism for computation
- **Hybrid languages** (like Scala, OCaml) support both styles

Understanding side effects is crucial for language design because they impact: - Program correctness (pure functions are easier to reason about) - Parallelization (side effects complicate parallel execution) - Optimization (pure functions allow more aggressive optimizations)

In our interpreter, we'll model side effects using explicit state passing, maintaining the mathematical clarity of our semantics while accurately representing the behavior of stateful programs.

Section 2: Denotable vs. Memorizable Values

Denotable Values (DVal)

Denotable values are those that can be bound to identifiers in an environment. In our implementation, they include:

```
type Num = int  # A type alias for integers
type DenOperator = Callable[[int, int], int]
type DVal = int | DenOperator  # Denotable values
```

DVal includes: - **Numbers**: Simple integer values - **Operators**: Functions that take two integers and return an integer

Memorizable Values (MVal)

Memorizable values are those that can be stored in memory (the state). In our implementation:

```
type MVal = int  # Memorizable values
```

MVal only includes integers, not functions. This highlights an important distinction:

> Not everything that can be bound to a name can be stored in memory.

This distinction is crucial for understanding: - Why some languages don't support first-class functions - Why some types require special treatment in memory management - How languages with different type systems handle values differently

Section 3: Environment and State as Functions

In our treatment of semantic domains, we adopt a purely functional approach where environments are represented as functions, and states are represented as dataclasses containing store functions, rather than mutable data structures. This approach aligns with the mathematical view of semantic domains and provides a clean conceptual model for understanding program behavior.

Functional Programming: A Brief Digression

Before diving into our function-based implementation of environments and state, it's worth taking a brief detour to discuss functional programming concepts, as they form the foundation of our approach.

Functional programming is a paradigm where computations are treated as evaluations of mathematical functions, emphasizing immutable data and avoiding side effects. Python, while not a pure functional language, supports many functional programming techniques.

Functions as First-Class Citizens In functional programming, functions are "first-class citizens" — they can be: - Assigned to variables - Passed as arguments to other functions - Returned from functions - Stored in data structures

For example:

```
# Function assigned to a variable
increment = lambda x: x + 1

# Function passed as an argument
def apply_twice(f, x):
    return f(f(x))

result = apply_twice(increment, 3)  # Returns 5
```

Higher-Order Functions: Map A common pattern in functional programming is applying a function to each element in a collection. Python's `map` function does exactly this:

```
numbers = [1, 2, 3, 4, 5]

# Apply a function to each element
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16, 25]

# Equivalent to a list comprehension
squared_alt = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]
```

Function Composition Functional programming emphasizes building complex behaviors by composing simpler functions:

```
def compose(f, g):
    return lambda x: f(g(x))

# Compose two functions
negate_and_square = compose(lambda x: -x, lambda x: x**2)
result = negate_and_square(5)  # -25
```

Pure Functions and Immutability Pure functions always produce the same output for the same input and have no side effects. This property makes them predictable and easier to reason about:

```
# Pure function
def add(a, b):
    return a + b
```

```python
# Impure function (has side effects)
def add_and_print(a, b):
    result = a + b
    print(f"The result is {result}")  # Side effect: printing
    return result
```

Relevance to Semantic Domains These functional programming concepts directly inform our approach to implementing semantic domains:

1. We represent environments and stores as functions, not data structures
2. We use higher-order functions to create updated environments and stores
3. We maintain immutability through functional updates rather than mutations
4. We compose simple operations to build complex behaviors

With this foundation in mind, let's explore how we represent environments and states as functions.

Environment as a Function

An environment is mathematically a function that maps identifiers to denotable values:

```python
type Environment = Callable[[str], DVal]
```

This means an environment is a function that: - Takes an identifier (string) as input - Returns a denotable value (DVal) - Raises an error if the identifier is not defined

While many practical implementations use dictionaries or hash tables for efficiency, conceptually an environment is simply a function:

```
Environment: Identifier → DVal
```

State as a Dataclass

Unlike environment, which is purely a function, a state encapsulates both a store function and allocation information:

```python
@dataclass
class State:
    store: Callable[[Location], MVal]  # The store function
    next_loc: int                      # Next available location
```

This represents a state as: - A dataclass containing a store function and next location counter - The store function takes a location as input and returns the value at that location - The next_loc field tracks the next available memory location for allocation

Conceptually, the store component maintains the mapping:

```
Store: Location → MVal
```

While the next_loc component tracks allocation state.

Section 4: Functional Updates

In a purely functional approach, we don't mutate existing environments or states. Instead, we create new functions or dataclasses that encapsulate the updated behavior.

Environment Updates

Instead of modifying a dictionary, we define a new function that returns the new value for the updated identifier and delegates to the original environment for all other identifiers:

```
def env_extend(env: Environment, name: str, value: DVal) -> Environment:
    """Create new environment with an added binding"""
    def new_env(n: str) -> DVal:
        if n == name:
            return value
        return env(n)
    return new_env
```

This function returns a new environment that: - Returns `value` when asked for `name` - Delegates to the original environment for all other identifiers

This approach: - Preserves referential transparency - Enables easy implementation of lexical scoping - Facilitates reasoning about program behavior - Models the mathematical concept of function extension

State Updates

Similarly, state updates create new State objects with updated store functions:

```
def state_update(state: State, location: Location, value: MVal) -> State:
    """Create new state with an updated value at given location"""
    def new_store(loc: Location) -> MVal:
        if loc == location:
            return value
        return state.store(loc)
    return State(store=new_store, next_loc=state.next_loc)
```

This function creates a new State object that: - Contains a new store function that returns the new value when asked for the specified location - Delegates to the original state's store function for all other locations - Preserves the next_loc value from the original state

Empty Environment and State

The primitives for creating empty environments and states define initial values:

```
def empty_environment() -> Environment:
    """Create an empty environment function"""
    def env(name: str) -> DVal:
        raise ValueError(f"Undefined identifier: {name}")
    return env

def empty_memory() -> State:
    """Create an empty memory state"""
    def store(location: Location) -> MVal:
        raise ValueError(f"Undefined memory location: {location}")
    return State(store=store, next_loc=0)
```

Note that empty_memory returns a State dataclass initialized with an empty store function and next_loc set to 0.

Memory Allocation

In a complete interpreter, we use the State dataclass to implement memory allocation elegantly:

```
def allocate(state: State, value: MVal) -> tuple[State, Location]:
    """Allocate a new memory location and store a value there.
    Returns the updated state and the new location."""
    location = state.next_loc
    new_state = state_update(state, location, value)
```

```
    # Return state with incremented next_loc and the allocated location
    return State(store=new_state.store, next_loc=location + 1), location
```

This function: 1. Gets the next available location from the state 2. Updates the store function to map this location to the provided value 3. Returns a new state with the incremented next_loc and the allocated location

By bundling the store function with the next_loc counter in our State dataclass, we maintain a purely functional approach while elegantly handling the allocation challenge. This design demonstrates how functional programming can manage state without side effects by making state changes explicit in the return values of functions.

Initial Environment Setup

In our functional implementation, the initial environment is built by starting with an empty environment and extending it with each operator:

```
def create_initial_env() -> Environment:
    """Create an environment populated with standard operators"""
    env = empty_environment()
    env = env_extend(env, "+", add)
    env = env_extend(env, "-", subtract)
    env = env_extend(env, "*", multiply)
    env = env_extend(env, "/", divide)
    env = env_extend(env, "%", modulo)
    return env
```

This builds up the environment incrementally, adding each binding through functional extension.

Section 5: Environment-Based Interpretation

Traditional interpreters often use pattern matching on operators directly in the evaluation function. An environment-based approach takes a more abstract view, treating operators as first-class values in the environment.

Traditional Approach (from Chapter 3)

```
def evaluate(ast: Expression) -> int:
    match ast:
        case Number(value):
            return value
        case BinaryExpression(op, left, right):
            left_value = evaluate(left)
            right_value = evaluate(right)
            match op:
                case "+":
                    return left_value + right_value
                case "-":
                    return left_value - right_value
                # ... other operations
```

Environment-Based Approach

```
def evaluate(ast: Expression, env: Environment) -> MVal:
    match ast:
        case Number(value):
            return value
        case BinaryExpression(op, left, right):
```

```
            try:
                # Get operator from environment
                operator = env_lookup(env, op)

                # Ensure it's a DenOperator
                if not isinstance(operator, Callable):
                    raise ValueError(f"{op} is not a function")

                # Evaluate operands and apply operator
                left_value = evaluate(left, env)
                right_value = evaluate(right, env)

                # Apply the operator to the evaluated operands
                return operator(left_value, right_value)
            except ValueError as e:
                raise ValueError(f"Evaluation error: {e}")
```

Benefits of the Environment-Based Approach

- **Extensibility**: New operators can be added to the environment without modifying the evaluator
- **First-class operations**: Operators are values that can be passed, returned, and manipulated
- **Consistent treatment**: Operators and other identifiers are handled uniformly
- **Semantic clarity**: The environment explicitly represents the mapping from names to meanings

Section 6: Implementing an Environment-Based Interpreter

Our `domains.py` file implements a complete environment-based interpreter:

1. **Define semantic domains**: Denotable and memorizable values
2. **Implement operators**: Define functions for arithmetic operations
3. **Create the environment**: Populate with standard operators
4. **Evaluate expressions**: Using the environment to look up operators
5. **REPL**: Interactive environment for testing the interpreter

Section 7: Extending the Interpreter

This approach makes it easy to extend the language with new features:

Adding New Operators

To add a new operator, simply define its function and add it to the environment:

```
def power(x: int, y: int) -> int:
    return x ** y

# Extend environment
env = env_extend(create_initial_env(), "**", power)
```

Adding Variables

To support variables, extend the AST with a variable node and update the evaluator:

```
@dataclass
class Variable:
    name: str
```

```python
# Update Expression type
type Expression = Number | BinaryExpression | Variable

# Update evaluate function
def evaluate(ast: Expression, env: Environment) -> MVal:
    match ast:
        case Variable(name):
            value = env_lookup(env, name)
            # Additional check might be needed if variables can only be Numbers
            if not isinstance(value, int):
                raise ValueError(f"{name} is not a number")
            return value
        # ... existing cases
```

Additional Resources

- Denotational Semantics (Wikipedia)
- Programming Language Semantics (Stanford)
- Functional Programming and Lambda Calculus
- Environment and Store in Programming Languages

Section 8: Primitives for Environment and Memory

In the formal semantics of programming languages, we work with primitives that define how environments and memory operate. These primitives capture the essential operations needed to model variable bindings and memory allocation.

Locations as a Semantic Domain

Before discussing memory operations, we must recognize **Locations** as a fundamental semantic domain:

```
type Location = int  # In our implementation, locations are integers
```

Locations (sometimes called addresses) are abstract entities that serve as references to memory cells. They are a semantic domain distinct from integers used in calculation, even though we might represent them as integers in an implementation. In a language's formal semantics, locations are opaque values that only make sense in the context of memory operations.

Memory (State) Primitives

Memory, also called the store or state, is represented by the State dataclass which contains both a store function mapping locations to memorizable values and a next_loc field. The core operations include:

1. **empty_memory**: Creates an initial, empty memory state

```python
def empty_memory() -> State:
    """Create an empty memory state"""
    def store(location: Location) -> MVal:
        raise ValueError(f"Undefined memory location: {location}")
    return State(store=store, next_loc=0)
```

2. **update**: Modifies the memory at a specific location

```python
def state_update(state: State, location: Location, value: MVal) -> State:
    """Create new state with an updated value at given location"""
    def new_store(loc: Location) -> MVal:
        if loc == location:
            return value
```

```
        return state.store(loc)
    return State(store=new_store, next_loc=state.next_loc)
```

3. **lookup**: Retrieves a value from a location

```
def state_lookup(state: State, location: Location) -> MVal:
    """Look up a value at a given location"""
    try:
        return state.store(location)
    except ValueError:
        raise ValueError(f"Undefined memory location: {location}")
```

Memory in programming languages has a maximum size, determined by hardware or system configuration. When a program needs more memory than available:

- In simple interpreters: An "out of memory" error occurs
- In modern operating systems: Memory expansion mechanisms activate, such as:
  - Virtual memory paging to disk
  - Heap expansion
  - Garbage collection (freeing unused memory)

Environment Primitives

The environment maps identifiers to denotable values. Its essential operations include:

1. **empty_environment**: Creates an initial, empty environment

```
def empty_environment() -> Environment:
    """Create an empty environment function"""
    def env(name: str) -> DVal:
        raise ValueError(f"Undefined identifier: {name}")
    return env
```

2. **bind** (or **extend**): Adds a new binding to an environment

```
def env_extend(env: Environment, name: str, value: DVal) -> Environment:
    """Create new environment with an added binding"""
    def new_env(n: str) -> DVal:
        if n == name:
            return value
        return env(n)
    return new_env
```

3. **lookup**: Retrieves a value bound to an identifier

```
def env_lookup(env: Environment, name: str) -> DVal:
    """Look up an identifier in the environment"""
    try:
        return env(name)
    except ValueError:
        raise ValueError(f"Undefined identifier: {name}")
```

Memory and Environment in Language Semantics

The interaction between environments and memory is central to understanding language features:

- **Variables**: Bind identifiers to locations (in the environment) which then hold values (in memory)
- **Assignment**: Changes memory but not the environment (the variable still refers to the same location)
- **Scoping**: Creates nested environments with different bindings for the same identifier
- **Parameter passing**: Creates bindings between formal parameters and actual arguments

This conceptual separation allows language designers to reason clearly about the effects of operations and ensure language features interact correctly.