Programming Languages Design Workshop

Vincenzo Ciancia

October 23, 2025

Contents

Chapter 1: Introduction to Programming Language Design

Section 1: What is a Programming Language?

A **programming language** is a formal language comprising a set of strings (instructions) that produce various kinds of machine output. Programming languages are used in computer programming to implement algorithms.

Key Characteristics of Programming Languages

- **Syntax**: The form or structure of the expressions, statements, and program units
- **Semantics**: The meaning of the expressions, statements, and program units
- **Type System**: The set of types and rules for how types are assigned to various constructs in the language
- **Runtime Model**: How the language executes on a computer, including memory management
- **Standard Library**: Common functionality provided out of the box

Programming Languages vs. Natural Languages

Programming languages differ from natural languages in several important ways:

1. **Precision**: Programming languages are designed to be precise and unambiguous
2. **Vocabulary**: Programming languages have a limited vocabulary defined by the language specification
3. **Grammar**: Programming languages have a strict, formal grammar with precise rules
4. **Evolution**: Programming languages evolve through explicit design decisions, not organic usage
5. **Purpose**: Programming languages are designed to instruct machines, not primarily for human communication

Section 2: The Importance of Programming Language Design

Why should we care about programming language design?

Programming Languages Shape How We Think

Programming languages are not just tools for instructing computers; they are frameworks for human thinking. Different languages emphasize different concepts and approaches:

- **Imperative languages** (C, Pascal) focus on step-by-step instructions
- **Functional languages** (Haskell, Lisp) emphasize expressions and function composition
- **Object-oriented languages** (Java, C++, Python) organize code around objects and their interactions
- **Logic languages** (Prolog) express programs as logical relations

Language Design Affects Software Quality

The design of a programming language can significantly impact:

- **Reliability**: How easy is it to write correct code?
- **Maintainability**: How easy is it to understand and modify existing code?
- **Performance**: How efficiently can the code be executed?
- **Security**: How easily can programmers avoid security vulnerabilities?
- **Developer Productivity**: How quickly can developers write and debug code?

The Evolution of Programming Languages

Programming languages have evolved dramatically over time, reflecting changes in hardware, software engineering practices, and problem domains:

- **1950s**: Assembly languages and early high-level languages (FORTRAN, LISP)
- **1960s**: ALGOL, COBOL, and structured programming concepts
- **1970s**: C, Pascal, and the rise of procedural programming
- **1980s**: C++, Ada, and the adoption of object-oriented programming

- **1990s**: Java, Python, Ruby, and the focus on portability and productivity
- **2000s**: C#, JavaScript frameworks, and web-centric languages
- **2010s**: Go, Rust, Swift, and the focus on safety and concurrency
- **2020s**: Continued evolution with AI assistance, type inference improvements, and more

Section 3: Modern Language Design Principles

What principles guide the design of modern programming languages?

Abstraction

Abstraction is the process of removing details to focus on the essential features of a concept or object.

Examples in programming languages: - Functions abstract away implementation details - Classes abstract data and behavior - Interfaces abstract expected behaviors - Modules abstract related functionality

Expressiveness

Expressiveness refers to how easily and concisely a language can express computational ideas.

Factors that contribute to expressiveness: - Rich set of operators and built-in functions - Support for higher-order functions - Pattern matching - Concise syntax for common operations

Safety

Safety features help prevent programmers from making mistakes or make it easier to find and fix errors.

Safety mechanisms in modern languages: - Static type checking - Bounds checking - Memory safety guarantees - Exception handling systems - Null safety features

Performance

Performance considerations affect how efficiently a language can be implemented and executed.

Performance factors: - Compilation vs. interpretation - Memory management approach - Static vs. dynamic typing - Optimization opportunities - Support for concurrency and parallelism

Consistency

Consistency in language design makes languages easier to learn and use correctly.

Consistency principles: - Similar concepts should have similar syntax - Minimal special cases - Orthogonal features (features that can be used in any combination) - Principle of least surprise (intuitive behavior)

Section 4: Using Python to Explore Programming Language Concepts

Why use Python for studying programming language design?

Python's Suitability for Language Implementation

Python is well-suited for implementing language interpreters and exploring language concepts:

- **Readability**: Python's clean syntax makes interpreter code easier to understand
- **High-level constructs**: Python provides lists, dictionaries, and other structures useful for language implementation
- **Dynamic typing**: Simplifies working with diverse language constructs (NOT REALLY! left here for discussion)
- **Rich standard library**: Includes parsing tools, regular expressions, and other useful utilities
- **Interactive development**: Makes experimenting with language features easier

Python 3.10+ Features Relevant to Language Design

Recent Python versions have introduced features that make it particularly interesting for PL experiments:

- **Type hints**: Allows for static type checking while maintaining dynamic execution
- **Pattern matching**: Provides elegant structural decomposition similar to functional languages
- **Dataclasses**: Simplifies creating data-carrying classes with minimal boilerplate
- **Functional programming tools**: Map, filter, reduce, lambdas, and comprehensions
- **AST module**: Allows inspection and manipulation of Python's abstract syntax tree

Section 5: Implementing Language Features in Python

Let's explore how we can implement core language components in Python.

Representing Syntax: Abstract Syntax Trees (ASTs)

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code. Here's a simple example of representing expressions:

```python
from dataclasses import dataclass
from typing import Union, List

# Define the node types
@dataclass
class Number:
    value: float

@dataclass
class Variable:
    name: str

@dataclass
class BinaryOp:
    left: 'Expr'
    operator: str
    right: 'Expr'

# Define the expression type
Expr = Union[Number, Variable, BinaryOp]

# Example: 2 + (x * 3)
expr = BinaryOp(
    left=Number(2.0),
    operator='+',
    right=BinaryOp(
        left=Variable('x'),
        operator='*',
        right=Number(3.0)
    )
)
```

Pattern Matching for AST Processing

Python 3.10's pattern matching provides a more elegant way to implement evaluators:

```python
    match expr:
        case Number(value):
```

```python
            return value
        case Variable(name):
            if name not in environment:
                raise NameError(f"Variable '{name}' not defined")
            return environment[name]
        case BinaryOp(left, operator, right):
            left_val = evaluate_with_match(left, environment)
            right_val = evaluate_with_match(right, environment)

            match operator:
                case '+':
                    return left_val + right_val
                case '-':
                    return left_val - right_val
                case '*':
                    return left_val * right_val
                case '/':
                    return left_val / right_val
                case _:
                    raise ValueError(f"Unknown operator: {operator}")
```

Section 6: Course Structure

This course will introduce you to programming language design concepts through hands-on implementation in Python.

Course Topics

Throughout this course, we will build a simple programming language interpreter while covering key topics:

1. **Language Syntax and Semantics**
    - Parsing and lexical analysis
    - Abstract syntax trees
2. **Language Features**
    - Environment (how to assign names to things?)
    - State (how to represent changing values?)
    - Control flow (if, while, for, etc.)
    - Static scoping (how do variable names related to nested environments)

Projects and Exercises

The course will include:

- Regular programming exercises to reinforce concepts
- Progressive development of a language interpreter
- Exploration of existing language implementations
- Analysis of language design trade-offs

Knowledge Prerequisites

To get the most out of this course, you should have:

- Basic Python programming experience
- Understanding of fundamental programming concepts (variables, functions, control flow)
- Familiarity with basic data structures (lists, dictionaries, trees)
- Interest in how programming languages work "under the hood"

No prior experience with compiler or interpreter development is required.

Additional Resources

Online Resources

- Python Documentation
- Python Type Hints
- Pattern Matching in Python 3.10
- The AST Module
- Building a Simple Interpreter

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.

```
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`:

```
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:

```
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:

```
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:

```python
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: Building a Mini Interpreter

Section 1: Introduction to Interpreters

An interpreter is a program that executes source code directly, without requiring compilation to machine code. Let's explore how interpreters work and how to build a simple one in Python.

The Role of Interpreters

Interpreters serve several important purposes in programming language implementation:

1. **Direct Execution**: Execute source code without a separate compilation step
2. **Immediate Feedback**: Provide instant results for interactive programming
3. **Portability**: Run on any platform that supports the interpreter
4. **Simplicity**: Often easier to implement than full compilers
5. **Debugging**: Allow for interactive debugging and inspection

Languages like Python, JavaScript, and Ruby primarily use interpreters, while others like Java use a hybrid approach with compilation to bytecode followed by interpretation.

Components of an Interpreter

A typical interpreter includes the following components:

1. **Lexer (Tokenizer)**: Converts source code text into tokens
2. **Parser**: Transforms tokens into an abstract syntax tree (AST)
3. **Evaluator**: Executes the AST to produce results
4. **Environment**: Stores variables and their values
5. **Error Handler**: Manages and reports errors

We'll implement each of these components in our mini interpreter.

Section 2: Using Lark Parser

Instead of manually implementing a lexer and parser, we'll use the Lark parsing library, which provides a powerful and declarative way to parse expressions.

Why Use Lark?

Lark is a modern parsing library for Python that offers several advantages:

1. **Declarative Grammar**: Define your language using EBNF notation
2. **Automatic Tokenization**: No need to write a manual lexer
3. **Parse Tree Generation**: Automatically creates parse trees
4. **Pattern Matching**: Easy to transform parse trees into ASTs

Installing Lark

First, ensure Lark is installed in your environment:

```
pip install lark
```

Defining the Grammar

We define our arithmetic expression grammar using Lark's EBNF format:

```
from __future__ import annotations
from lark import Lark, Token, Tree
```

```python
# Define the grammar in Lark's EBNF format
grammar = r"""
    expr: bin | mono
    mono: ground | paren
    paren: "(" expr ")"
    bin: expr OP mono
    ground: NUMBER

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

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

Understanding the Grammar

The grammar defines:

- **expr**: An expression can be binary operation or monadic (single value)
- **mono**: A monadic expression is either a ground value or parenthesized expression
- **ground**: A basic number literal
- **bin**: A binary operation (left operand, operator, right operand)
- **NUMBER**: A regex pattern matching digits
- **OP**: Operators (+, -, *, /, %)
- **WS**: Whitespace is imported and ignored

Creating the Parser

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

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

Examining Parse Trees

Lark provides a convenient method to visualize the parse tree:

```python
print(parse_tree.pretty())

# Output:
# expr
#   bin
#     expr
#       mono
#         paren
#           expr
#             bin
#               expr
#                 mono
#                   ground        1
```

```
#               +
#               mono
#                 ground  2
#     -
#     mono
#       ground    3
```

Section 3: Transforming Parse Trees to ASTs

While Lark creates parse trees automatically, we need to transform them into our own Abstract Syntax Tree (AST) representation for evaluation.

Defining AST Nodes

We define our AST using Python's type system and dataclasses:

```
from dataclasses import dataclass
from typing import Literal

# Define operator type
type Op = Literal["+", "-", "*", "/", "%"]

@dataclass
class Number:
    value: int

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

type Expression = Number | BinaryExpression
```

Understanding AST vs Parse Tree

- **Parse Tree**: Generated by Lark, contains all grammar details
- **AST**: Simplified tree with only semantically relevant information
- **Transformation**: We convert parse trees to ASTs using pattern matching

Transforming Parse Trees with Pattern Matching

We use Python's pattern matching to transform Lark's parse trees into our AST:

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

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

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

```python
        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),
            )
```

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

Complete Parsing Function

Combining Lark parsing with our transformation:

```python
def parse_ast(expression: str) -> Expression:
    parse_tree = parser.parse(expression)
    return transform_parse_tree(parse_tree)

# Example usage
ast = parse_ast("(1+2)-3")
```

Section 4: Evaluating Expressions

Now that we have an AST, we can evaluate it to produce a result.

The Evaluation Process

Evaluation is the process of computing the result of an expression. It typically involves:

1. Walking the AST recursively
2. Computing the value of each node based on its type and children
3. Combining results according to the language semantics

Implementing an Evaluator

We implement the evaluator using pattern matching:

```python
def evaluate(ast: Expression) -> int:
    match ast:
        case Number(value):
            return value
```

```python
        case BinaryExpression(op, left, right):
            left_value = evaluate(left)
            right_value = evaluate(right)
            match op:
                case "+":
```

```python
                return left_value + right_value
            case "-":
                return left_value - right_value
```

```python
            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
```

Convenience Function

We can combine parsing and evaluation:

```python
def evaluate_string(expression: str) -> int:
    ast = parse_ast(expression)
    return evaluate(ast)

# Example
result = evaluate_string("(1+2)*3")  # Returns 9
```

Section 5: Building a REPL

Now we'll create a Read-Eval-Print Loop (REPL) for interactive use.

Implementing the REPL

```python
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)
```

Running the REPL

```python
# Start the interactive interpreter
REPL()
```

Example Session

```
Enter an expression (exit to quit): 1+2
3
Enter an expression (exit to quit): (10+20)*2
60
```

```
Enter an expression (exit to quit): 100/0
Division by zero
Enter an expression (exit to quit): 17%5
2
Enter an expression (exit to quit): exit
```

Section 6: Debugging Parse Trees

Lark provides useful tools for debugging and understanding parse trees.

Printing Parse Trees

You can inspect the raw parse tree structure:

```python
def print_tree(tree: Tree | 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)

# Usage
parse_tree = parser.parse("(1+2)-3")
print_tree(parse_tree)
```

Pretty Printing

Lark's built-in pretty printer is even more convenient:

```python
parse_tree = parser.parse("(1+2)-3")
print(parse_tree.pretty())
```

This shows the hierarchical structure with indentation, making it easy to understand how the parser interpreted the expression.

Section 7: Extending the Interpreter

Our mini interpreter is basic but extensible. Here are some ideas for enhancements:

Possible Extensions

1. **Variables**: Add variable storage and assignment
2. **Functions**: Support function definitions and calls
3. **More Operators**: Add comparison (<, >, ==) and logical operators (and, or, not)
4. **Control Flow**: Implement if-then-else expressions
5. **Type System**: Add type checking before evaluation
6. **Error Messages**: Improve error reporting with line numbers and context

Modifying the Grammar

To extend the language, you can modify the Lark grammar:

```python
# Add comparison operators
grammar = r"""
    expr: comparison | bin | mono
```

```
    comparison: expr COMP mono
    mono: ground | paren
    paren: "(" expr ")"
    bin: expr OP mono
    ground: NUMBER

    NUMBER: /[0-9]+/
    OP: "+" | "-" | "*" | "/" | "%"
    COMP: "==" | "!=" | "<" | ">" | "<=" | ">="

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

Then update your AST and transformer accordingly.

Section 8: Learning Resources

To learn more about interpreters and language implementation:

Recommended Reading

- **"Crafting Interpreters"** by Robert Nystrom: A comprehensive guide to implementing interpreters
- **"Programming Language Pragmatics"** by Michael Scott: Covers theoretical aspects of language design
- **Lark Documentation**: https://lark-parser.readthedocs.io/

Key Takeaways

1. **Lark simplifies parsing**: No need to write manual lexers and parsers
2. **Pattern matching is powerful**: Transforming parse trees to ASTs is elegant with pattern matching
3. **Recursive evaluation**: AST evaluation naturally uses recursion
4. **Extensibility**: The architecture supports easy addition of new features

Conclusion

We've built a mini interpreter that:

1. Uses Lark to parse arithmetic expressions into parse trees
2. Transforms parse trees into ASTs using pattern matching
3. Evaluates ASTs recursively to produce results
4. Provides an interactive REPL for testing

This foundation can be expanded to build more complex languages and tools.

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**:

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

4. **Network operations**:

```python
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:

```python
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:

```python
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:

```python
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:

```python
def bind(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:

```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)
```

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:

```python
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:

```python
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)
```

```python
    # 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:

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

```python
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

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

```python
            try:
                # Get operator from environment
                operator = 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:

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

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

```
### Adding Variables

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

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

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

Update evaluate function

```python
def evaluate(ast: Expression, env: Environment) -> MVal:
    match ast:
        case Variable(name):
            value = 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 9: Practical Implementation and Testing

Our environment-based interpreter includes practical components for execution and testing, demonstrating how theoretical concepts translate to code.

Parsing and AST Construction

In our implementation, we use the Lark parser to convert text expressions into parse trees:

```python
def parse_ast(expression: str) -> Expression:
    """Parse a string expression into an AST"""
    parse_tree = parser.parse(expression)
    return transform_parse_tree(parse_tree)
```

The `transform_parse_tree` function then converts these parse trees into our AST representation, ready for evaluation.

Interactive REPL

A Read-Eval-Print Loop allows interactive testing of the interpreter:

```python
def REPL():
    """Read-Evaluate-Print Loop with environment"""
    env = create_initial_env()

    print("Mini-interpreter with environment (type 'exit' to quit)")
    print("Available operators: +, -, *, /, %")

    while True:
        expression = input("Enter an expression (exit to quit): ")
        if expression == "exit":
            break
```

```python
        try:
            ast = parse_ast(expression)
            result = evaluate(ast, env)
            print(result)
        except Exception as e:
            print(f"Error: {e}")
```

The REPL shows how our environment-based interpreter integrates with user interaction.

Automated Tests

Testing ensures the interpreter behaves as expected across various expressions:

```python
def run_tests():
    """Run test expressions to verify the parser and evaluator"""
    test_expressions = [
        "1+2",
        "3*4",
        "5-3",
        "10/2",
        "10%3",
        "(1+2)*3",
        "1+(2*3)",
        "10/(2+3)",
        "10%(2+3)",
    ]

    env = create_initial_env()

    for expr in test_expressions:
        try:
            ast = parse_ast(expr)
            result = evaluate(ast, env)
            print(f"{expr} = {result}")
        except Exception as e:
            print(f"{expr} -> Error: {e}")
```

These tests validate core functionality while providing examples of valid expressions.

Section 10: Conclusion and Next Steps

Summary of Key Concepts

In this chapter, we've explored:

1. **Semantic domains** as mathematical structures that give meaning to programs
2. **Environment-based interpretation** as a flexible approach to language implementation
3. **Functional representations** of environments and state
4. **Primitive operations** for manipulating environments and memory
5. **The distinction** between denotable and memorizable values

These concepts form the foundation for understanding more complex language features in subsequent chapters.

Looking Forward

The environment-based approach introduced here will be extended in future chapters to support:

1. **Variables and assignment**: Using environments to bind identifiers to memory locations

2. **Scoping mechanisms**: Creating nested environments for block-structured code
3. **Functions and closures**: Capturing environments for later execution
4. **Typing systems**: Adding constraints on what values expressions can produce

By building on these semantic foundations, we can construct a rich understanding of programming language design and implementation.

Exercises for the Reader

1. Extend the interpreter to support variables using the Variable AST node described in Section 7
2. Add support for multi-character operators (e.g., "**" for exponentiation)
3. Implement a memory system with the primitives described in Section 8
4. Add support for conditional expressions (if-then-else)
5. Implement a simple block structure with local variables

These exercises will deepen your understanding of language semantics and interpreter design while preparing you for more advanced topics in the next chapters.

Chapter 5: Variable Binding and Let Expressions

Section 1: Introduction to Variable Binding

In this chapter, we extend our mini-interpreter to support **variable binding** through `let` expressions. This is a fundamental capability that allows programs to:

1. Name and reuse values
2. Create modular components
3. Build abstractions that hide implementation details

The extension represents our first step toward a more practical and powerful language.

Variable Binding vs. Assignment

It's important to distinguish between binding and assignment:

- **Variable Binding**: Associates a name with a value in a specific scope or environment

  ```
  let x = 10 in ...
  ```

- **Assignment**: Changes the value associated with an existing name

  ```
  x = 10
  ```

In our mini-language, we implement variable binding but not assignment. This makes our language more functional in nature and simplifies reasoning about program behavior.

Section 2: Let Expressions

Syntax and Semantics of Let Expressions

We extend our grammar with `let` expressions:

```
let: "let" IDENTIFIER "=" expr "in" expr
```

A `let` expression has three components: 1. The name (identifier) being bound 2. The expression that provides the value to bind 3. The body expression where the binding is in scope

For example, in `let x = 10 in x + 5`, we: - Bind `x` to the value `10` - Evaluate the body `x + 5` in this extended environment, resulting in `15`

AST Representation

To support `let` expressions in our abstract syntax tree, we add a new class:

```python
@dataclass
class Let:
    name: str
    expr: Expression
    body: Expression
```

And we extend our `Expression` type to include `Let` and `Var` (variable reference) nodes:

```
type Expression = Number | BinaryExpression | Let | Var
```

Variable References

To use bound variables, we need to support variable references in our language:

```python
@dataclass
class Var:
    name: str
```

When evaluating a variable reference, we look up its value in the current environment:

```
case Var(name):
    x = lookup(env, name)
    match x:
        case int():
            return x
        case _:
            raise ValueError(f"Unexpected value type: {type(x)}")
```

Section 3: The Environment

Environment as a First-Class Citizen

In our mini-interpreter, the environment is a first-class entity that's explicitly passed around during evaluation. This makes the binding context explicit in our semantics.

The key operations on environments are: - **Creation**: Building an initial environment with primitive operations - **Lookup**: Finding the value associated with a name - **Extension**: Adding new bindings to create a derived environment

Environment Implementation

Our environment is implemented as a function that maps names to values:

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

We extend the environment when evaluating a `let` expression:

```
case Let(name, expr, body):
    value = evaluate(expr, env)
    extended_env = bind(env, name, value)
    return evaluate(body, extended_env)
```

This creates a new environment that includes the original bindings plus the new binding.

Static vs. Dynamic Scoping

Our implementation uses **lexical (static) scoping**, where variable references are resolved in the environment where they are defined, not where they are used.

This is in contrast to **dynamic scoping**, where variable references are resolved in the current execution environment.

Lexical scoping is more predictable and easier to reason about, which is why it's the dominant approach in modern programming languages.

Static vs. Dynamic Scoping Examples Consider this expression:

```
let x = 5 in
  let f = (let y = 10 in y + x) in
    let x = 42 in
      f
```

With **static scoping** (our implementation), this evaluates to: - `f` is bound to the result of `(let y = 10 in y + x)` where `x` is 5, so `f` is 15 - Even though we later rebind `x` to 42, the reference to `x` inside `f` still refers to the original binding - Final result: 15

With **dynamic scoping**, this would evaluate to: - `f` is bound to the expression `(let y = 10 in y + x)` (not its value yet) - When `f` is evaluated, it looks for the current binding of `x`, which is 42 - Final result: 52

Another example:

```
let a = 1 in
  let f = (a + 10) in
    let a = 100 in
      f
```

With **static scoping**: - `f` evaluates to 11 using the binding `a = 1` - The later binding of `a = 100` has no effect on `f`

With **dynamic scoping**: - `f` would be 110 since it would use the latest binding of `a` when evaluated

Examples

Let expressions dramatically increase the expressiveness of our language. Consider a few examples:

```
let x = 10 in x + 5
```

Evaluates to 15 by binding x to 10 and adding 5.

```
let x = 1 in let y = 2 in x + y
```

Nested binding.

Section 5: Conclusion

Adding variable binding through let expressions marks a significant step in the evolution of our mini-language. In future chapters, we'll build on this foundation to add more capabilities:

- Conditionals and booleans
- Iteration (while and for loops)
- Functions and application
- Recursion

Parser and Grammar Extensions

Compared to Lecture 4, we've made the following extensions to the parser and grammar:

1. Added a new production rule for `let` expressions:

```
let: "let" IDENTIFIER "=" expr "in" expr
```

2. Extended the expression rule to include both variables and let expressions:

```
?expr: bin | mono | let
mono: ground | paren | var
var: IDENTIFIER
```

3. Added new pattern matching cases in the parse tree transformation function:

```
case Tree(data="var", children=[Token(type="IDENTIFIER", value=name)]):
    return Var(name=name)

case Tree(
    data="let",
    children=[
        Token(type="IDENTIFIER", value=name),
        expr,
        body,
    ],
):
    return Let(
        name=name,
        expr=transform_parse_tree(expr),
```

```
        body=transform_parse_tree(body),
    )
```

4. Added evaluation rules for the new AST node types:

```
case Let(name, expr, body):
    value = evaluate(expr, env)
    extended_env = bind(env, name, value)
    return evaluate(body, extended_env)
case Var(name):
    x = lookup(env, name)
    match x:
        case int():
            return x
        case _:
            raise ValueError(f"Unexpected value type: {type(x)}")
```

These extensions allow our language to support both variable definitions and references while maintaining the core evaluation semantics from Lecture 4 that will be useful in the second part of the course.

Chapter 6: State

Section 1: Introduction to State

In this chapter, we extend our mini-language with the concept of **state**. Until now, our language has been purely functional, where expressions are evaluated to produce values without side effects. By adding state, we can:

1. Store and update values in memory
2. Observe changes to data over time
3. Model real-world systems that have changing state

Expressions vs. Commands

Our language extension introduces a new distinction:

- **Expressions**: Compute values (e.g., `x + 1`, `let x = 42 in x * 2`)
- **Commands**: Perform actions with side effects (e.g., `var x = 42`, `x <- 42`, `print x`)

This distinction is common in many programming languages:

```
# Expression (computes a value)
x + 1

# Command (performs an action)
var x = 42
x <- 42
```

Section 2: Commands and Command Sequences

To implement state, we introduce two new syntactic categories:

1. **Commands**: Individual actions that can modify state
2. **Command Sequences**: Ordered lists of commands to be executed in sequence

Commands in Our Language

We implement three basic command types:

1. **Variable Declaration**: Declares a new variable and initializes it

   ```
   var x = 42
   ```

2. **Assignment**: Updates a variable with a new value (only if already declared)

   ```
   x <- 10
   ```

3. **Print**: Outputs the value of an expression

   ```
   print x + 1
   ```

Command Sequences

Command sequences represent multiple commands separated by semicolons:

```
var x = 10;
var y = x + 5;
print y
```

This sequence declares `x`, then declares `y` as `x + 5` (which is 15), and finally prints the value of `y`.

Grammar Extensions

We extend our language grammar to support commands and sequences:

```
?program: command_seq

?command_seq: command
           | command ";" command_seq

?command: assign
        | print
        | vardecl

assign: IDENTIFIER "<-" expr
print: "print" expr
vardecl: "var" IDENTIFIER "=" expr
```

AST Representation

We represent commands and command sequences with these AST node types:

```python
@dataclass
class Assign:
    name: str
    expr: Expression

@dataclass
class Print:
    expr: Expression

@dataclass
class VarDecl:
    name: str
    expr: Expression
```

```python
@dataclass
class CommandSequence:
    first: Command
    rest: Optional[CommandSequence] = None

type Command = Assign | Print | VarDecl
```

Section 3: The Store Model of State

To represent state, we use the “store model”:

1. **Environment**: Maps variable names to locations (memory addresses)
2. **Store**: Maps locations to values

This two-level indirection allows multiple variables to refer to the same location or for a variable’s value to change without changing its location.

The State as a Functional Dataclass

Our implementation uses a `State` dataclass to manage the store in a functional style:

- The store is **not** a dictionary, but a function from locations to values (or raises an error if not found), just like the environment is a function from names to locations. This ensures a fully functional approach.

```python
@dataclass
class State:
    store: Callable[[int], MVal]
    next_loc: int

def empty_store() -> Callable[[int], MVal]:
    def store_fn(loc: int) -> MVal:
        raise ValueError(f"Location {loc} not allocated")
    return store_fn

def empty_state() -> State:
    return State(store=empty_store(), next_loc=0)
```

```python
def allocate(state: State, value: MVal) -> tuple[Loc, State]:
    loc = Loc(state.next_loc)
    prev_store = state.store

    def new_store(l: int) -> MVal:
        if l == loc.address:
            return value
        return prev_store(l)

    return loc, State(store=new_store, next_loc=loc.address + 1)
```

```python
def update(state: State, addr: int, value: MVal) -> State:
    prev_store = state.store

    def new_store(l: int) -> MVal:
        if l == addr:
            return value
        return prev_store(l)

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

def access(state: State, addr: int) -> MVal:
    return state.store(addr)
```

Denotable Values (DVal)

Denotable values are those that can be bound to identifiers in an environment. Our language now has three distinct semantic domains:

- **EVal (Expressible Values)**: Values that expressions can evaluate to (integers in our language)
- **MVal (Memorizable Values)**: Values that can be stored in memory (equal to EVal in our implementation)
- **DVal (Denotable Values)**: Values that can be bound to identifiers in the environment

DVal is broader than MVal because not everything that can be bound to a name can be stored in memory. Specifically, DVal includes:

- **Numbers**: Simple integer values (EVal)
- **Operators**: Functions that take two integers and return an integer
- **Locations**: Memory addresses (wrapped in a Loc class)

The distinction between these domains is crucial for implementing state correctly:

- Expressions evaluate to EVal (integers)
- Memory cells store MVal (integers)

- Variable names can refer to DVal (integers, operators, or locations)

```
@dataclass
class Loc:
    address: int

type DenOperator = Callable[[int, int], int]
type EVal = int  # Expressible value type (for expressions)
type MVal = EVal  # Main value type for store and evaluation (expressible)
type DVal = EVal | DenOperator | Loc  # Denotable values: can be associated with names
```

Rather than representing memory locations as plain integers, we wrap them in the `Loc` class for two important reasons:

1. **Pattern Matching**: It allows the pattern matcher to distinguish locations from integer values
2. **Type Safety**: It prevents accidentally using a location as an integer or vice versa

This distinction becomes essential in variable lookup. When we look up a variable, we need to determine if the value bound to it is:

- A direct integer value (for operators like +, -, etc.)
- A location that requires a further lookup in the store

Without the `Loc` wrapper class, the pattern matcher wouldn't be able to distinguish between an integer value and an integer location in memory.

Section 4: Evaluating Expressions with State

When evaluating expressions in a stateful language, we need to pass both the environment and the state:

```
def evaluate_expr(expr: Expression, env: Environment, state: State) -> EVal:
    # ...
```

Variable lookup now has two steps:

1. Use the environment to find the location or directly bound value
2. If it's a location, use the state to look up the value at that location

```
case Var(name):
    try:
        dval = lookup(env, name)
        match dval:
            case int():
                return dval
            case Loc(address=addr):
                return access(state, addr)
            case _:
                raise ValueError(f"Variable '{name}' does not refer to a value")
    except ValueError as e:
        raise ValueError(f"Variable error: {e}")
```

Section 5: Executing Commands

Commands modify the state or produce output. The `execute_command` function returns both the updated environment and state as a tuple:

```
def execute_command(
    cmd: Command, env: Environment, state: State
) -> tuple[Environment, State]:
    # ...
```

The Variable Declaration Command

The variable declaration command (`var x = expr`) creates a new variable and initializes it:

```
case VarDecl(name, expr):
    value = evaluate_expr(expr, env, state)
    loc, state = allocate(state, value)
    new_env = bind(env, name, loc)
    return new_env, state
```

The Assignment Command

The assignment command (`<-`) only updates an existing variable:

```
case Assign(name, expr):
    try:
        dval = lookup(env, name)
        match dval:
            case Loc(address=addr) as loc:
                value = evaluate_expr(expr, env, state)
                state1 = update(state, addr, value)
                return env, state1
            case _:
                raise ValueError(
                    f"Assignment target '{name}' is not a variable"
                )
    except ValueError:
        raise ValueError(f"Assignment to undeclared variable '{name}'")
```

The Print Command

The print command evaluates the expression and prints its value:

```
case Print(expr):
    # MORALLY THIS IS THE IDENTITY FUNCTION
    value = evaluate_expr(expr, env, state)
    print(value)
    return env, state
```

Command Sequences

Executing a command sequence involves:

1. Executing the first command
2. Executing the rest of the sequence with the updated environment and state

```
def execute_command_seq(
    seq: CommandSequence, env: Environment, state: State
) -> tuple[Environment, State]:
    # Execute the first command
    env1, state1 = execute_command(seq.first, env, state)

    # If there are more commands, execute them with the updated environment and state
    if seq.rest:
        return execute_command_seq(seq.rest, env1, state1)

    return env1, state1
```

Section 6: Examples of State in Action

Let's look at some examples that demonstrate the power of state:

Example 1: Basic Declaration, Assignment, and Printing

```
var x = 42; print x
```

This simple example shows creating a variable and reading its value. The output is `42`.

Example 2: Updating Variables

```
var x = 10; print x; x <- 20; print x
```

This example demonstrates updating a variable's value. The output is:

```
10
20
```

Example 3: Multiple Declarations and Operations

```
var x = 10; var y = 20; var z = x + y; print z; x <- 30; print x + y
```

The output of this program is:

```
30
50
```

Notice that changing `x` doesn't automatically update `z`, even though `z` was defined in terms of `x` and `y`.

Example 4: Let Expressions in Commands

```
var x = let y = 5 in y * 2; print x
```

This example shows how we can use let expressions within commands. The output is `10`.

Section 7: Differences from Previous Chapters

Adding state represents a significant shift in our language:

Purely Functional (Ch. 5)	Stateful (Ch. 6)
Values are immutable	Values can change over time
No side effects	Commands have side effects
Referential transparency	Expressions may evaluate differently
Environment maps names to values	Environment maps names to locations
Easier to reason about	More expressive

Section 8: Conclusion and Next Steps

Adding state to our mini-language significantly increases its expressiveness, making it capable of modeling real-world problems that involve change over time. In the next chapter, we'll build upon this foundation by adding control flow structures like loops and conditionals.

With state, environment, and control flow, our language will have all the essential ingredients of a complete programming language.

Exercises

1) Interactive REPL

Modify the REPL implementation to operate one command at a time instead of parsing and executing entire programs.

2) Aliasing

Add the command

```
alias x = y
```

where `x` and `y` are variables. The semantics is that after executing the command, both names x and y point to the same location, so assigning to one of them assigns also to the other one.

Make an example where this becomes apparent

Now consider the program

```
var x = 0;
alias y = x;
x <- x+1;
y <- y+1;
var y = x;
x <- x+1;
print y
```

QUESTION: which value is printed?

3) Multiple Assignment

Implement multiple assignment where multiple variables can be assigned at once:

```
x, y <- z + 1, x + 1
```

4) Parallel Assignment

Make the assignment "parallel" so that all right-hand sides are evaluated before any assignments happen; note that the variables x and y must be different in that case:

```
x, y <- y + 1, x + 1
```

This should swap the values of x and y.

5) If-Then-Else Statements

Add if-then-else statements to the language:

```
if x > 0 then
    y <- 1
else
    y <- -1
```

5) Command Sequences in Branches

Extend if-then-else to support command sequences in the branches and consider the implications for variable scoping.

Chapter 7: Control Flow

1. Introduction to Control Flow

In the previous chapters, we introduced **state** to our mini-language, allowing variables to be declared, updated, and printed. However, all programs executed commands in a fixed, linear order.

In this chapter, we enrich our language with **control flow constructs**—specifically, conditionals (`if`/`else`) and loops (`while`). These features allow programs to make decisions and repeat actions, greatly increasing their expressive power.

We also introduce **block-local variables** and extend the language to support boolean values and unified operators (arithmetic, relational, and boolean). This chapter marks a significant step toward a full-featured imperative language.

Summary of new features:

- `if`/`else` and `while` commands
- Boolean values and expressions
- Unified operator handling
- Block-local variable scoping (static/lexical scope)

2. Control Flow Constructs

2.1 If-Then-Else

The `if` command allows a program to choose between two branches based on a boolean condition. The syntax is:

```
if <condition> then <commands> else <commands> endif
```

- The `<condition>` must be a boolean expression.
- Both the `then` and `else` branches can contain sequences of commands.

```python
@dataclass
class IfElse:
    cond: Expression
    then_branch: CommandSequence
    else_branch: CommandSequence

# ...
```

```python
def execute_command(cmd: Command, env: Environment, state: State) -> tuple[Environment, State]:
    match cmd:
        case IfElse(cond, then_branch, else_branch):
            cond_val = evaluate_expr(cond, env, state)
            if not isinstance(cond_val, bool):
                raise ValueError("If condition must be boolean")
            saved_next_loc = state.next_loc
            if cond_val:
                _, state1 = execute_command_seq(then_branch, env, state)
                # Restore next_loc after block
                state2 = State(store=state1.store, next_loc=saved_next_loc)
                return env, state2
            else:
                _, state1 = execute_command_seq(else_branch, env, state)
                state2 = State(store=state1.store, next_loc=saved_next_loc)
                return env, state2
```

Implementation: If-Then-Else This ensures that variables declared inside a branch are only visible within that branch, and their memory is reclaimed after the branch ends.

Example:

```
var x = 1;
if x == 1 then print 42 else print 0 endif
```

This program prints `42` because the condition `x == 1` is true.

2.2 While Loops

The `while` command allows repeated execution of a block of commands as long as a condition holds:

```
while <condition> do <commands> done
```

- The `<condition>` must be a boolean expression.
- The body can be a sequence of commands.

```python
@dataclass
class While:
    cond: Expression
    body: CommandSequence

# ...
```

```python
def execute_command(cmd: Command, env: Environment, state: State) -> tuple[Environment, State]:
    match cmd:
        case While(cond, body):
            cond_val = evaluate_expr(cond, env, state)
            if not isinstance(cond_val, bool):
                raise ValueError("While condition must be boolean")
            saved_next_loc = state.next_loc
            if cond_val:
                _, state1 = execute_command_seq(body, env, state)
                # Restore next_loc after block
                state2 = State(store=state1.store, next_loc=saved_next_loc)
                return execute_command(While(cond, body), env, state2)
            else:
                return env, state
```

Implementation: While Loops **Example:**

```
var n = 3;
while n > 0 do print n; n <- n - 1 done
```

This program prints `3`, `2`, and `1` on separate lines.

3. Block-Local Variables and Scoping

A major semantic change in this chapter is the introduction of **block-local variables**. Variables declared inside a block (such as the body of an `if`, `else`, or `while`) are only visible within that block. This is known as **static (lexical) scoping**.

- When a block ends, its local variables are no longer accessible.
- The implementation reuses memory locations for block-local variables by resetting the next available location counter after a block ends. This models stack allocation and prevents unbounded memory growth.

Implementation: Block-Local Variables and Lexical Scoping Block-local variables are managed using a stack-like memory model. Recall how allocation and deallocation work:

```
@dataclass
class State:
    store: Callable[[int], MVal]
    next_loc: int
```

```
def allocate(state: State, value: MVal) -> tuple[Loc, State]:
    loc = Loc(state.next_loc)
    prev_store = state.store
    def new_store(l: int) -> MVal:
        if l == loc.address:
            return value
        return prev_store(l)
    return loc, State(store=new_store, next_loc=loc.address + 1)
```

- **Allocation**: Each new variable gets a fresh location (`next_loc`), which is incremented.
- **Deallocation**: After a block, `next_loc` is reset, so locations for block-local variables can be reused.

Example:

```
if cond then
    var x = 42; print x
else
    print 0
```

Here, `x` is only accessible inside the `then` branch. After the branch, its memory location can be reused for other variables.

Digression: Memory Safety and Buffer Over-Read

A **buffer over-read** occurs when a program reads data past the end of a buffer (an array or memory region), often due to incorrect pointer arithmetic or missing bounds checks. In C, this is a common source of security vulnerabilities, especially when working with stack-allocated arrays.

Example: Buffer Over-Read in C

```c
#include <stdio.h>

void print_secret() {
    char buffer[8] = "hello";
    char secret[8] = "SECRET!";
    for (int i = 0; i < 16; i++) {
        printf("%c", buffer[i]); // Over-reads into secret
    }
    printf("\n");
}

int main() {
    print_secret();
    return 0;
}
```

Output:

```
helloSECRET!
```

Here, the loop prints not only the contents of `buffer`, but also the contents of the adjacent `secret` array, which is stored on the stack. This is a classic buffer over-read: the program leaks data from memory that should have been inaccessible, illustrating why careful memory management and scoping are crucial.

4. Unified Operators and Boolean Expressions

Our language now supports a unified set of operators:

- **Arithmetic:** `+`, `-`, `*`, `/`, `%`
- **Relational:** `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Boolean:** `and`, `or`, `not`

Example:

```
var x = 10;
var y = 5;
if x > y and not (y == 0) then print x / y else print 0
```

5. Grammar Extensions

The grammar is extended to support control flow and booleans:

```
?command: assign
        | print
        | vardecl
        | ifelse
        | while

assign: IDENTIFIER "<-" expr
print: "print" expr
vardecl: "var" IDENTIFIER "=" expr
ifelse: "if" expr "then" CommandSequence "else" CommandSequence
while: "while" expr "do" CommandSequence

?expr: ...
```

6. Abstract Syntax Tree (AST) Extensions

The AST is extended to represent the new constructs:

```python
@dataclass
class IfElse:
    cond: Expression
    then_branch: CommandSequence
    else_branch: CommandSequence

@dataclass
class While:
    cond: Expression
    body: CommandSequence
```

Command sequences allow multiple commands in each branch or loop body.

6.1 AST and Semantics of Operators

Operators in the language are represented in the AST using the `Apply` node:

```python
@dataclass
class Apply:
    op: str
    args: list[Expression]
```

- `op` is the operator name (e.g., ‘+’, ‘and’, ‘==’).

- `args` is a list of argument expressions (one for unary, two for binary operators).

All operators are stored in the environment as `Operator` objects:

```
@dataclass
class Operator:
    type: tuple[list[type], type]  # (argument types, return type)
    fn: Callable[[list[EVal]], EVal]
```

- `type` is a tuple where the first element is a list of argument types (e.g., `[int, int]` for binary integer operators, `[bool]` for unary boolean operators), and the second element is the return type.
- `fn` is the function implementing the operator's semantics.

Operator application is handled in the expression evaluator as follows:

```
def evaluate_expr(expr: Expression, env: Environment, state: State) -> EVal:
    match expr:
        # ...
        case Apply(op, args):
            arg_vals = [evaluate_expr(a, env, state) for a in args]
            op_val = lookup(env, op)
            if isinstance(op_val, Operator):
                expected_types, _ = op_val.type
                if len(expected_types) != len(arg_vals):
                    raise ValueError(
                        f"Operator '{op}' expects {len(expected_types)} arguments, got {len(arg_vals)}"
                    )
                for i, (expected, actual) in enumerate(zip(expected_types, arg_vals)):
                    if type(actual) is not expected:
                        raise ValueError(
                            f"Operator '{op}' argument {i+1} expects type {expected.__name__}, got {type(actual).__name__}"
                        )
                return op_val.fn(arg_vals)
            raise ValueError(f"{op} is not an operator")
```

- The evaluator checks that the operator exists, that the number of arguments matches its type signature, and that each argument matches the expected type.
- If all checks pass, the operator's function is applied to the evaluated arguments.
- If any check fails, a runtime error is raised.

Operator application is checked at runtime for both correct arity and argument types (the type signature), ensuring safe execution and clear error messages. For example, applying `and` to non-booleans or dividing by zero will raise an error.

Example:

```
# Example: evaluating x + y
Apply(op='+', args=[Var('x'), Var('y')])
```

This node will look up the '+' operator in the environment, evaluate `x` and `y`, check the number and types of arguments, and then apply the addition function to the results.

7. Examples of Control Flow in Action

Example 1: If-Then-Else

```
var x = 1;
if x == 1 then print 42 else print 0 endif
```

Output:

```
42
```

Example 2: While Loop

```
var n = 3;
while n > 0 do print n; n <- n - 1 done
```

Output:

```
3
2
1
```

Example 3: Euclid's Algorithm (GCD) Using Subtraction

```
var a = 48; var b = 18;
while b != 0 do
    if a > b then a <- a - b else b <- b - a endif;
print a done
```

Output:

```
6
```

8. Comparison with Previous Chapters

Chapter 6: State	Chapter 7: Control Flow
State and variable assignment	Adds conditionals and loops
No control flow	Programs can branch and repeat
Variables global to block	Block-local variable scoping
Only arithmetic expressions	Boolean and relational expressions
Arithmetic expressions only	Arithmetic, boolean, and relational expressions

9. Conclusion and Next Steps

With the addition of control flow, our mini-language can now express a wide range of algorithms and computations. Block-local variables and unified operators bring us closer to the features of real-world programming languages.

In the next chapter, we will explore more advanced features, such as functions and closures, and discuss the semantic challenges they introduce.

Appendix: Closures, Denotable Values, and State

Closures (functions together with their captured environment) are a prime example of a value that can be **denoted** (named and referenced in the environment), but not **expressed** (evaluated to a simple value) or **stored as a value in the program's state**, unless special provisions are made.

Why Closures Are Special

- In a language with lexical scoping, a closure "remembers" the environment in which it was created, including variables that may have gone out of scope.
- If closures are allowed to be stored in state (e.g., assigned to variables), the variables they capture can outlive their lexical scope, breaking the clean separation between environment (static, lexical) and state (dynamic, mutable).
- This can lead to subtle bugs and makes reasoning about programs more complex.

Special Provisions for Closures

To allow closures to be stored in state safely, languages typically provide:

- **Heap allocation for environments:** Captured variables are stored on the heap, so they persist as long as the closure does, not just for the duration of the block.
- **Garbage collection or reference counting:** To reclaim memory when closures and their environments are no longer accessible.
- **First-class environments:** Environments are represented as first-class values that can be stored, passed, and manipulated at runtime.

```
function makeCounter() {
  let x = 0;
  return function () {
    return ++x;
  };
}
let counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Example (JavaScript): Here, `x` lives as long as the closure does, even after `makeCounter` has returned.

In Our Mini-Language

In the current chapter, only simple values (like numbers and booleans) are allowed to be stored in state. Closures are not yet supported as storable values, which keeps the semantics simple and reasoning about programs tractable. Supporting closures as storable values requires the special provisions described above.

Example: The Danger of Memorizing Closures

Suppose our mini-language supported lambda-abstractions (anonymous functions) and allowed them to capture local variables. Consider the following (hypothetical) example:

```
var x = 0;
if cond then
    var y = 42;
    x <- lambda() { return y }  // y is a local variable, now accessible outside of its scope!
else
    // do something else

print x() // call the closure to read y outside of its scope
```

Here, the lambda-abstraction `lambda() { return y }` captures the local variable `y` declared inside the `then` branch. We then assign this closure to the global variable `x`.

If the language allows closures to be stored in state (e.g., as variable values), the closure may outlive the scope of `y`. Later, calling `x()` would attempt to access `y`, which no longer exists—leading to undefined behavior or runtime errors.

This illustrates why special care is needed when allowing closures to be stored in state: the captured environment must persist as long as the closure does, or else memory safety and correctness are compromised.

Chapter 8: Function Abstraction — Design and Semantics

1. Introduction

In this chapter, we take a major step in the evolution of our mini-language: we add **user-defined functions**. This enables abstraction, modularity, and code reuse, and brings our language closer to real-world programming languages.

What you will learn

- How to extend the parser and grammar to support function definitions and applications
- How to update the abstract syntax tree (AST) to represent functions and closures
- The semantics of function declaration, closure creation, and application
- Subtle issues such as closure escape and stateful closures
- The implications for state and effects in expressions
- The calling convention and possible alternatives
- Extensions and exercises

2. Parser and Grammar Changes

To support user-defined functions, we extend the grammar with new rules for **function declaration** and **function application**.

Function Declaration Syntax

A function is declared as:

```
function f(x, y) = expr
```

- `f` is the function name
- `x, y` are parameters (zero or more)
- `expr` is the function body (an expression)

Function Application Syntax

A function is applied as:

```
f(3, 4)
```

- `f` is the function name
- `3, 4` are argument expressions

Grammar Fragment

```
fundecl: "function" IDENTIFIER "(" param_list ")" "=" expr
param_list: IDENTIFIER ("," IDENTIFIER)*
funapp: IDENTIFIER "(" arg_list ")"
arg_list: expr ("," expr)*
```

These rules are integrated into the parser, allowing functions to be declared in any block and called in any expression.

3. Abstract Syntax Tree (AST) Changes

We introduce new AST nodes to represent functions and their applications:

```
@dataclass
class FunctionDecl:
    name: str
    params: list[str]
    body: Expression
```

```python
@dataclass
class FunctionApp:
    name: str
    args: list[Expression]
```

- **FunctionDecl**: Represents a function's name, parameter list, and body
- **FunctionApp**: Represents a function call with arguments

Semantic Domains: The Role of Closures

To support functions and closures, we update our semantic domain of denotable values as follows:

```python
@dataclass
class Closure:
    function: FunctionDecl
    env: Environment

# Denotable values (DVal): can be associated with names in the environment
# NEW: Closure is now a denotable value!
DVal = EVal | Loc | Operator | Closure
```

- **Closure**: Represents a function paired with its declaration environment (lexical scoping)

4. Semantics of Functions and Closures

Function Declaration and Closure Creation

When a function is declared, its name is bound in the current environment to a **closure**:

- The closure contains the function's code (parameters and body)
- The closure captures the environment at the point of declaration

This enables **lexical (static) scoping**: the function "remembers" the variables in scope when it was defined.

Example: Closure Creation

```
var x = 10;
function f(y) = x + y
```

Here, `f` is bound to a closure that captures the current environment, including the binding of `x`.

Function Application (Step by Step)

To apply a function:

1. Look up the closure for the function name in the environment
2. Evaluate the argument expressions in the *calling* environment and state
3. Extend the closure's environment with parameter-to-argument bindings
4. Evaluate the function body in the extended environment and the current state

This ensures that the function body sees the environment at declaration, not at call.

Example: Lexical Scoping

```
var x = 10;
function f(y) = x + y;
var x = 100;
print f(1)  # prints 11, not 101
```

- The function `f` captures `x = 10` at its declaration, so `f(1)` evaluates to `11`.

5. Problems: Closure Escape and State

The Closure Escape Problem

If closures could be stored in the store (i.e., if they were **memorizable**), they could outlive their declaring block and access or update state that no longer exists.

This would break the stack discipline of our state model and could lead to subtle bugs (see the appendix in Lecture 7 for a discussion of this issue).

Example (not allowed in our language):

```
var a = 0;
if cond then
    var x = 42;
    function f() = x + 1;
    a = f  # not possible: closures are not memorizable
endif
# a would reference x after the block ends!
```

Why closures cannot escape

- In our language, **closures are not memorizable**: they cannot be stored in the store, only in the environment
- There is **no heap**: all memory is stack-allocated, and locations are reused after a block ends
- This ensures that closures cannot outlive their environment or access invalid state

6. Implementation: Closures Cannot Be Returned from Expressions

- In our language, **expressions can only return expressible values**: integers and booleans.
- **Closures are not expressible values**; they are denotable values, which means they can be bound to names in the environment, but not stored in the store or returned as results of expressions.

If closures could be returned from expressions, they could be stored in variables (i.e., in the store), or passed around as values, potentially escaping the block in which they were defined. This would break the stack discipline of our state model, as closures could reference variables that no longer exist (see the closure escape problem).

- **Reading**: When an expression refers to a variable, it looks up the value in the environment. If the value is a closure, it can only be used in a function application context, not as a value to be returned or stored.
- **Returning**: Only expressible values (int, bool) can be returned from expressions. Attempting to return a closure would be a type error in our semantics.

7. Stateful Closures and Effects in Expressions

If a closure captures a variable by environment, and the function body assigns to that variable, then the closure is **stateful**.

Example: Stateful Closure

```
var x = 0;
function inc() = x <- x + 1;
inc(); inc(); print x  # would print 2 if allowed
```

Since functions can be called in expressions, and if functions could have side effects, then **expressions themselves become effectful**. This blurs the line between expressions and commands, and is a key design decision in language semantics.

8. Calling Convention: What Is Passed?

Current Convention

- When a function is called, the **store** (state) at the call site is passed to the function body
- The **environment** is the one captured by the closure at declaration, extended with parameter bindings
- The calling environment is *not* passed

This is a form of **lexical scoping** with call-by-value for arguments.

Possible Alternatives

- **Dynamic scoping**: pass the calling environment instead of the closure's environment
- **Call-by-reference**: pass locations instead of values

Each alternative has different implications for modularity, reasoning, and security.

9. Exercises and Extensions

Exercise 1. Procedures

Extend the language with **procedures**: a variant of functions that can only be called as commands, not as expressions.

For example:

```
procedure p(x) = print x;
p(42)
```

- Procedures can perform side effects but do not return values
- How would you change the grammar, AST, and semantics to support this?

Exercise 2. Stateful Functions

Allow functions to update variables in their environment (i.e., allow assignment in function bodies).

- What changes are needed to the AST and semantics? How does this affect the distinction between expressions and commands?

Summary

- Functions are first-class, block-local, and lexically scoped
- Closures pair function code with the environment at declaration
- Closures are not memorizable, so cannot escape their block or break state
- Operators and closures are both denotable values, but are kept distinct for clarity
- The semantics are modular and support correct state and environment handling for closures