Lecture 07: Control_Flow

Vincenzo Ciancia

May 15, 2025

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>
```

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

Implementation: If-Then-Else

```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
```

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
```

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>
```

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

Implementation: While Loops

```
@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
```

Example:

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

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
```

```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)
```

- **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() {
```

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

AST Node for Operators

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

```
@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).

Operator Class

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

```
@dataclass
class Operator:
    arity: int
    fn: Callable[[list[EVal]], EVal]
```

- `arity` is the number of arguments the operator expects.
- `fn` is the function implementing the operator's semantics.

Semantics of Operator Application

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):
                if op_val.arity != len(arg_vals):
                    raise ValueError(
                        f"Operator '{op}' expects {op_val.arity} arguments, got {len(arg_vals)}"
                    )
                return op_val.fn(arg_vals)
            raise ValueError(f"{op} is not an operator")
```

- The evaluator checks that the operator exists and that the number of arguments matches its arity.
- If the check passes, the operator's function is applied to the evaluated arguments.
- If the check fails, a runtime error is raised.

Example: Operator Application

```
# 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 arity, and then apply the addition function to the results.

Runtime Type and Arity Checks

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

Example 1: If-Then-Else

```
var x = 1;

if x == 1 then print 42 else print 0
```

Output:

```
42
```

Example 2: While Loop

```
var n = 3;

while n > 0 do print n; n <- n - 1
```

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;
print a
```

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.

10. Exercises

1. **Write a program that prints the first 5 even numbers using a while loop.**
2. **Modify the language to support nested if-then-else statements.**
3. **Experiment with block-local variables: what happens if you declare a variable inside an if or while block?**
4. **Implement a program that computes the factorial of a number using a while loop.**

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 **memorized** (stored in the 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.

Example (JavaScript):

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

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.