Lecture 06: State

Vincenzo Ciancia

May 02, 2025

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:

TODO: also use an assignemnt in the example

```
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: vardecl

        | assign

        | print


vardecl: "var" IDENTIFIER "=" expr

assign: IDENTIFIER "<-" expr

print: "print" expr
```

AST Representation

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

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


@dataclass
class Assign:
    name: str
    expr: Expression


@dataclass
class Print:
    expr: Expression


@dataclass
class CommandSequence:
```

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.

```
@dataclass(frozen=True)
class State:
    store: Callable[[int], Any]
    next_loc: int


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


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

Environment and Binding

Our environment maps names to locations:

```
# Environment now maps to locations (store addresses)
type Environment = Callable[[str], int]


def empty_environment() -> Environment:
    def env(name: str) -> int:
        raise ValueError(f"Undefined identifier: {name}")
    return env


def bind(env: Environment, name: str, loc: int) -> Environment:
    def new_env(n: str) -> int:
        if n == name:
            return loc
        return env(n)
    return new_env
```

This approach maintains the functional style of our previous chapters while allowing for

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) -> int:
    # ...
```

Variable lookup now has two steps:

1. Use the environment to find the location
2. Use the state to look up the value at that location

```
case Var(name):
    # Look up variable's location and then its value in the state
    try:
        loc = lookup_env(env, name)
        return lookup(state, loc)
    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:

```
env, state = execute_command(cmd, env, state)
```

- `env` is the updated environment (with new variable bindings, if any)
- `state` is the updated state (with new or updated values)

You can access the first and second components of the tuple as `env` and `state` respectively.

> ***Note:*** *The `print` command is not part of the mathematical semantics; it is just an aid for using the interpreter.*

The Variable Declaration Command

The variable declaration command (`var x = expr`) creates a new variable and initializes it. If the variable already exists, it is an error.

The Assignment Command

The assignment command (`<-`) only updates an existing variable. If the variable does not exist,

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;

var y = x + 5;

print y;

x <- 20;

print x;

print y
```

This example demonstrates that changing `x` doesn't automatically change `y`, even though `y` was defined in terms of `x`. The output is:

```
15

20

15
```

The value of `y` remains 15 because the expression `x + 5` was evaluated at the time of declaration.

Example 3: 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, closer to real languages

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 more complex conditionals.

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