Lecture 08: Functions

Vincenzo Ciancia

October 02, 2025

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:

```python
@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:

```
@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