Lecture 05: Binding

Vincenzo Ciancia

May 01, 2025

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:

```
@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:

```
@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.