Lecture 07: Control_Flow

Vincenzo Ciancia

May 02, 2025

WORK IN PROGRESS.

Section 1: Introduction to Booleans and Conditionals

In this chapter, we extend our mini-interpreter to support **boolean expressions** and **conditional (if-then-else) expressions**. These features are essential for making decisions in programs and are foundational to all programming languages.

Why Booleans and Conditionals?

- **Booleans** allow us to represent truth values (`True` and `False`).
- **Conditionals** enable programs to choose between different actions based on boolean conditions.
- Together, they make our language expressive enough to encode logic and control flow.

Section 2: Boolean Expressions

Syntax and Semantics

We extend our language with boolean literals and boolean operations:

- Boolean literals: `True`, `False`
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical operators: `and`, `or`

AST Representation

We add new AST node types for booleans and binary operations:

```
@dataclass
class Boolean:
    value: bool


@dataclass
class BinOp:
    op: str  # '+', '-', '*', '/', '==', '!=', '<', '>', '<=', '>=', 'and', 'or'
    left: Expression
    right: Expression
```

Section 3: Conditional Expressions

Syntax and Semantics

We add the `if` expression to our language:

```
if: "if" expr "then" expr "else" expr
```

This allows us to write expressions like:

```
if x > 0 then x else -x
```

AST Representation

```
@dataclass
class If:
    cond: Expression
    then_branch: Expression
    else_branch: Expression
```

Parser Extensions

The parser is extended to recognize boolean literals, comparison and logical operators, and conditional expressions. For demonstration, we use a simple tuple-based parser:

```
def parse(expr):
    if isinstance(expr, int):
        return Number(expr)
    if isinstance(expr, bool):
        return Boolean(expr)
    if isinstance(expr, str):
        return Var(expr)
    if isinstance(expr, tuple):
        tag = expr[0]
        if tag == 'let':
            _, name, value_expr, body_expr = expr
            return Let(name, parse(value_expr), parse(body_expr))
        if tag == 'if':
```

Evaluation Extensions

The evaluation function is extended to handle booleans, binary operations, and conditionals:

```
def eval_expr(expr: Expression, env: Environment) -> DVal:
    if isinstance(expr, Number):
        return expr.value
    if isinstance(expr, Boolean):
        return expr.value
    if isinstance(expr, Var):
        return env(expr.name)
    if isinstance(expr, Let):
        value = eval_expr(expr.expr, env)
        extended_env = bind(env, expr.name, value)
        return eval_expr(expr.body, extended_env)
    if isinstance(expr, BinOp):
        # Short-circuit for 'and' and 'or'
        if expr.op == 'and':
            left = eval_expr(expr.left, env)
            if not left:
```

Section 5: Short-Circuit Evaluation

Short-circuit evaluation means that in logical expressions like `and` and `or`, the second operand is only evaluated if necessary:

- For `and`, if the first operand is `False`, the result is `False` (second operand not evaluated).
- For `or`, if the first operand is `True`, the result is `True` (second operand not evaluated).

This is important for avoiding errors and improving efficiency.

Example: Avoiding Division by Zero

```
# let x = 0 in if (x != 0) and (10 // x > 1) then 1 else 2
ast = parse(('let', 'x', 0, ('if', ('and', ('!=', 'x', 0), ('>', ('/', 10, 'x'), 1)), 1, 2)))
print(eval_expr(ast, empty_env))  # Output: 2 (no error)
```

Section 6: Non-trivial Examples

Example 1: Simple Conditional with Binding

```
# let x = 10 in if x > 5 then x else 0
ast1 = parse(('let', 'x', 10, ('if', ('>', 'x', 5), 'x', 0)))
print(eval_expr(ast1, empty_env))  # Output: 10
```

Example 2: Nested Binding and Conditionals

```
# let x = 3 in let y = 4 in if (x < y) and (y < 10) then x + y else 0
ast2 = parse(('let', 'x', 3, ('let', 'y', 4, ('if', ('and', ('<', 'x', 'y'), ('<', 'y', 10)), ('+', 'x', 'y'), 0))))
print(eval_expr(ast2, empty_env))  # Output: 7
```

Example 3: Short-circuiting to Avoid Errors

```
# let x = 0 in if (x != 0) and (10 // x > 1) then 1 else 2
ast3 = parse(('let', 'x', 0, ('if', ('and', ('!=', 'x', 0), ('>', ('/', 10, 'x'), 1)), 1, 2)))
print(eval_expr(ast3, empty_env))  # Output: 2
```

Section 7: Conclusion and Next Steps

Adding booleans and conditionals makes our mini-language much more expressive, enabling it to represent logic and control flow. In future chapters, we will build on this by adding:

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

These features will allow us to write even more powerful and interesting programs in our mini-language.