Semantics (V)

Conditional Semantics

- In addition to the assignment statement, conditional statement (if statement) is another type of statement.
- It meaning function also takes in a state and generate another state.

$$M : Statement \times State \rightarrow State$$

- A conditional statement has three elements: an expression, then branch statement, and else branch statement.

$$Conditional = Expression\ test; Statement\ thenbranch, elsebranch$$

- The **meaning of a conditional depends on the meaning of the expression**.
  - If the expression is true, the meaning of the conditional is the meaning of the then branch.
  - If the expression is false, the meaning of the conditional is the meaning of the else branch.

$$M(Conditional\ c, State\ state) = \begin{cases} M(c.\text{thenbranch}, state), & \text{if } M(c.\text{test}, state) \text{ is true} \\ M(c.\text{elsebranch}, state), & \text{otherwise} \end{cases}$$

- To extend our simple interpreter for the conditional statement, we need to add conditional class, the meaning function to statements including assignments, conditionals, and extendible to other statement types, the meaning function of conditional, and a main function to test them.

```
# Conditional has three attribute: condition,
# thenbranch, and elsebranch
class Conditional:
    def __init__(self, condition, thenbranch, elsebranch):
        self.condition = condition
        self.thenbranch = thenbranch
        self.elsebranch = elsebranch

# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement (statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    else:
        raise Error("ERROR: wrong type of statement”)
```

```
# Meaning function of Contional
# M(Statement statement, State state)
def M_Conditional (statement, state):
    if M_Expression(statement.condition, state).value:
        state = M_Statement(statement.thenbranch, state)
    else:
        state = M_Statement(statement.elsebranch, state)
    return state


def main3():
    # if (a < b) {
    #   min = a;
    # }
    # else {
    #   min = b;
    # }
    # {<min, 0>, <a, 3>, <b, 5>}
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    expr = BinaryExpression(vara, '<', varb)
    thenbranch = Assignment(varmin, M_Expression(vara, state))
    elsebranch = Assignment(varmin, M_Expression(varb, state))
    cond = Conditional(expr, thenbranch, elsebranch)
    newState = M_Statement(cond, state)
    print(newState)

if __name__ == "__main__":
    #main1()
    #main2()
    main3()
```

Skip Semantics

- Exercise: write a main function to let the simple interpreter determine the meaning of the following conditional statement. `{<min, 0>, <a, 3>, <b, 5>}`

```
if (a < 5) {
      if (b > 3) {
            min = 100;
      }
}
else {
      min = 50;
}
```

- We need another type of statement here, **Skip**, for the then branch of the outer conditional statement if (a < 5).
- The else branch of the inner conditional statement if (b > 3) has nothing. From the perspective of a compiler/interpreter, this is a skip statement which takes in a state and returns the same state without modifying anything.

```
# Skip
class Skip:
    pass

# Meaning function of Skip
# M(Statement statement, State state)
def M_Skip (statement, state):
    return state;
```

- Then, we need to update the meaning function of statement to let it be able to handle Skip.

```
# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement (statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    elif isinstance(statement, Skip):
        return M_Skip(statement, state)
    else:
        raise Error("ERROR: wrong type of statement”)
```

- We can then implement the main function as

```
def main4():
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    inexpr = BinaryExpression(varb, '>', Value(3))
    outexpr = BinaryExpression(vara, '<', Value(5))
    inthenbranch = Assignment(varmin, Value(100))
    inelsebranch = Skip()
    outthenbranch = Conditional(inexpr, inthenbranch, inelsebranch)
    outelsebranch = Assignment(varmin, Value(50))
    cond = Conditional(outexpr, outthenbranch, outelsebranch)
    newState = M_Statement(cond, state)
    print(newState)
```

Block Semantics

- The conditional and skip semantics work well for the above if statement in which each branch only contains one statement. However, it’s more common in programs that a brach contains more than one statement, which we call a block.

- A block is a set of statements.

$$Block = Statement^*$$

- We need a meaning function for a block.
  - If the set is empty, the block has no statement. So the meaning of the program is not changed by the block.
  - Otherwise, the meaning of the block is the aggregated results of all statements in the block.

- The implementation of the meaning function

$$M(Block\ \mathrm{b}, State\ \mathrm{state}) = \begin{cases} state, & \text{if } \mathrm{b} = \{\} \\ M(b_n, M(b_{n-1}, \cdots, M(b_1, \mathrm{state})\cdots)), & \text{if } \mathrm{b} = \{\ b_1, b_2, \cdots, b_n\} \end{cases}$$

  - If the set is **empty**, the **state** does **not change**.
  - If there are $\mathbf{n}$ **statements** in the block, the meaning function of each statement is called. The **inner most meaning function is the first statement** in the block, and the **outer most is the last statement**.
  - Depending on the type of each statement, different meaning functions are called.

- To extend the simple interpreter, we need a block class, a meaning function for block, a main function to test the code, and extend the meaning function of statement to support blocks.

```
# Block
class Block:
    def __init__(self, statements):
        self.statements = statements

# Meaning function of Block
# M(Statement block, State state)
def M_Block (block, state):
    if (len(block.statements) == 0):
        return state
    return M_Block(Block(block.statements[1:]), M_Statement(block.statements[0], state))

# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement (statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    elif isinstance(statement, Skip):
        return M_Skip(statement, state)
    elif isinstance(statement, Block):
        return M_Block(statement, state)
    else:
        raise Error("ERROR: wrong type of statement”)


def main5():
    # if (a < 5) {
    #   min = 100;
    #   b = 0;
    # }
    # else {
    #   min = 50;
    #   b = 1;
    # }
    # {<min, 0>, <a, 3>, <b, 5>}
```

```
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    expr = BinaryExpression(vara, '<', Value(5))
    ifblckstatmnt1 = Assignment(varmin, Value(100))
    ifblckstatmnt2 = Assignment(varb, Value(0))
    elsblckstatmnt1 = Assignment(varmin, Value(50))
    elsblckstatmnt2 = Assignment(varb, Value(1))
    ifblck = Block([ifblckstatmnt1, ifblckstatmnt2])
    elsblck = Block([elsblckstatmnt1, elsblckstatmnt2])
    cond = Conditional(expr, ifblck, elsblck)
    newState = M_Statement(cond, state)
    print(newState)

if __name__ == "__main__":
    main5()
```