

Semantics of Conditional, Block, Loop Statements

Conditional Semantics

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

$$M : \text{Statement} \times \text{State} \rightarrow \text{State}$$

- 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.

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

- 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. Remember to extend the meaning function

for binary expression to support relational operators: >, <, ==, and !=.

```
# 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:
        print("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
```

- 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;
}

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()
```

- Complicating factor: **Lazy evaluation**
 - Give if (false && foo()) . . ., Is foo() called?
 - The answer depends on the language. Most languages now use **lazy evaluation** (short circuit evaluation). When evaluating && expressions, **do it from left to right and**

- stop as soon as you know the value of the expression. Therefore, in the example above, we don't need to call `foo()`. [C, Java, Python uses lazy evaluation]
- Similarly, `if (true || foo())` does not require a call to `foo()`. This is important if `foo()` has side effects.
 - **Why is this important?**
 - Consider the example `if (x != 0 && y/x > 3) . . .`
 - If `x` is equal to `0`, the lazy evaluation prevents the execution of the second expression and so avoid the DivideByZero error.
 - Therefore, the order of expressions in a lazy evaluation matters when the expressions have side effects.
 - Why care? [make the program more efficient]

```
//LazyEval.java
public class LazyEval {
    public static void main (String[]
args) {
        int x = 0;
        int y = 4;
        if (y/x > 3 || x != 0) {
            System.out.println("Hi,
there!");
        }
    }
}
```

```
# LazyEval.py
x = 0
y = 4

if (x != 0 and y/x > 3):
    print("Hi there!")
```

Skip Semantics

- 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:
        print("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)
```

- Caveat: the parser Stephanie wrote for Cliter doesn't actually handle Skip statements. Also, it always requires an else branch. So this is useful conceptually, and useful if we code the AST directly, but not useful when we are parsing statements in test.c

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 branch 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 (as presented by Tucker & Noonan)

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

- If the set is **empty**, the **state does not change**.
- If there are **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.
- Writing it more like a recursive function call (and borrowing some Python notation for slicing), we say

$$M(Block\ b, State\ state) = \begin{cases} state & , \text{ if } b.statement = \{\} \\ M(Block(b.statement[1:]), M(b.statement[0], state)) & , \text{ otherwise} \end{cases}$$

- 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:
        print("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))
    ifblkstatmnt1 = Assignment(varmin, Value(100))
    ifblkstatmnt2 = Assignment(varb, Value(0))
    elsbkstatmnt1 = Assignment(varmin, Value(50))
    elsbkstatmnt2 = Assignment(varb, Value(1))
    ifblk = Block([ifblkstatmnt1, ifblkstatmnt2])
    elsbk = Block([elsbkstatmnt1, elsbkstatmnt2])
    cond = Conditional(expr, ifblk, elsbk)
    newState = M_Statement(cond, state)
    print(newState)

if __name__ == "__main__":
    main5()

```

Loop Semantics

- Another type of statements that heavily relies on blocks is loop.
 - While loop semantics
 - A typical **while loop** is composed of an expression, test, and a loop body, a block of statements.
- Loop = Expression test; Statement body*

- The **meaning of a while loop depends on the meaning of the expression.**

$$M(\text{Loop } l, \text{State } state) = \begin{cases} M(l, M(l.\text{body}, state)), & \text{if } M(l.\text{test}, state) \text{ is true} \\ state, & \text{otherwise} \end{cases}$$

- If the expression is **true**, it is the **meaning of the body** of the loop.
 - Otherwise, the **meaning is the state**.
- Note that, **l.body** here **is a statement**. It can be an assignment, an expression, a conditional, a skip, or a block.
 - It also means that it is okay that if there is an assignment, expression, or conditional after the expression *without using curly braces*.
 - If there are more than one statement, curly braces should be used to form a block.
- To extend the simple interpreter, we need a While loop class, a meaning function for While loop, a main function to test the code, and extend the meaning function of statement for the loop. This is part of the HW, so I don't include the code here.
- **For loop semantics**
 - Cliter has one feature that Clite doesn't – for loops!
 - The abstract syntax for a for loop requires a number of different fields: an initial statement, an expression, the body statement, and the post statement.

For = Statement initial; Expression test; Statement body; Statement post

- The meaning of the for loop is a combination of things.
 - The initial statement modifies the state prior to evaluation of the test expression.
 - An expression that indicates whether or not the body should be evaluated
 - The body and post statements modify the state only if the test expression is true.
 - The post statement modifies the state after the application of the body to the state.
- In Cliter, we specify the pre- and post- parts are Statements. Like C, we can support multiple statements in the pre- and post- parts, but our syntax requires the programmer to use a block statement if they want multiple statements. This is different from C because C is actually looking for expressions with side effects. Since we don't have side effects, we need to allow assignment statements. Since we are using statements, we are going to be as consistent as possible and require blocks to group statements.
- We need more than one meaning function for a For loop – one to evaluate the initial statement, and one to recursively evaluate the condition, the body, and the post statement.

$$M(\text{For } f, \text{State } state) = M(f, M(f.\text{initial}, state))$$

$$M(\text{For } f, \text{State } state) = \begin{cases} state, & M(f.\text{test}, state) = \text{False} \\ M(f, M(f.\text{post}, M(f.\text{body}, state))), & \text{otherwise} \end{cases}$$