Semantics (III)

Expression Meaning Function

- We know that recursion is heavily used in the denotational semantics.
- For example, the rhs of an assignment can be a value, variable, binary or unary expression.
- So, to figure out the meaning of an assignment, we need to figure out the meaning of the rhs of the assignment.
- In Clite, an expression can be a variable, value, binary, or unary expression. *We will discuss the meaning of expression when it is a variable, value, and binary in this course.*

$$Expression = Variable | Value | Binary | Unary$$

- For expressions in Clite, the result is not a new state, it is just a value from a mathematical set. In languages with side-effecting expressions, evaluation may also change the state.

$$M(Expression\ \mathrm{e}, State\ \mathrm{state}) = \begin{cases} e, & \text{if } e \text{ is a } Value \\ state(e), & \text{if } e \text{ is a } Varial \\ ApplyBinary(e.\mathrm{op}, M(e.\mathrm{term1}, state), M(e.\mathrm{term2}, state)), & \text{if } e \text{ is a } Binar \\ ApplyUnary(e.\mathrm{op}, M(e.\mathrm{term}, state)), & \text{if } e \text{ is a } Unary \end{cases}$$

- We need to break it down by the type of expression and figure out the value.
  - If the Expression is a Value, then its *meaning is* the meaning of *the Value itself*
  - If the Expression is a Variable, then its *meaning is* the *Value of the Variable* in the current state
  - If the Expression is a Binary, then the *meaning of* each of its operands term1 *and* term2 *is first determined*. Then Meaning Rule of Binary Expression determines the meaning of the expression by *applying the Operator* op *to the Value of those two operands.*

Meaning Rule of Binary Expression

- The meaning rule defined a way to decide the value of an expression
- If either operand term1 or term2 is undefined, the expression is *semantically meaningless*
- If the operator is an integer operator, then integer arithmetic add (int+), subtract (int-), or multiply (int*) perform on the integer operands, resulting in an *integer result*. If the operator is divide (int/), then the result is the *same as a mathematical divide with truncation toward zero.*
- If the operator is a floating point operator, then floating point arithmetic *using the IEEE standard* is performed on the float operands, resulting in a float *result*
- If the operator is a relational operator, then the operands are compared with a *result of either* true *or* false
- meaning of an expression for a dynamic typed language would be different.

Simple Interpreter

- Using the expression semantics we discussed above, we can implement a semantic interpreter to generate/check the meaning of an expression.
- Now let's implement a simple interpreter for a language with dynamic typing, no side effects, and only one scope.

```
# Value
class Value:
    def __init__(self, value):
        self.value = value;

    def __str__(self):
        return str(self.value)

# Variable has the attribute name
class Variable:
    def __init__(self, name):
        self.name = name

# BinaryExpression has the attribute left,
# right, and operator
class BinaryExpression:
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right


# Maintain a dictionary that maps
# variables to their corresponding values
class State:
    def __init__(self):
        self.state = {}

    def setValue(self, var, value):
        self.state[var.name] = value

    def getValue(self, var):
        return self.state[var.name]


# expr should be a BinaryExpression
# state should be a State
# returns a value
def M_BinaryExpression(expr, state):
    if expr.operator == '+':
        val = M_Expression(expr.left, state).value + M_Expression(expr.right,
state).value
    elif expr.operator == '-':
        val = M_Expression(expr.left, state).value - M_Expression(expr.right,
state).value
    elif expr.operator == '*':
        val = M_Expression(expr.left, state).value * M_Expression(expr.right,
state).value
    elif expr.operator == '/':
        val = M_Expression(expr.left, state).value / M_Expression(expr.right,
state).value
    else:
        raise ValueError(f"Unsupported operator: {expr.operator}")
    return Value(val)
```

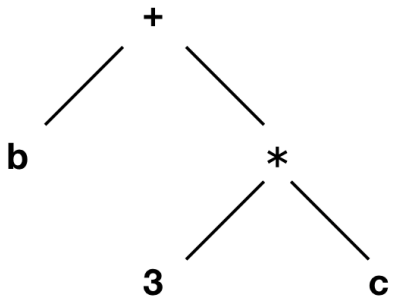
```
# Meaning function of Expression
# M(Expression expr, State state)
def M_Expression (expr, state):
    if isinstance(expr, Value):
        return expr
    elif isinstance(expr, Variable):
        return Value(state.getValue(expr))
    elif isinstance (expr, BinaryExpression):
        return M_BinaryExpression(expr, state)
    else:
        raise Error("ERROR: wrong type of expression")


def main1():
    # b + 3 * c {<b, 1>, <c, 3>}
    varb = Variable('b')
    varc = Variable('c')
    val3 = Value(3)
    expr = BinaryExpression(varb, '+', BinaryExpression(val3, '*', varc))
    state = State()
    state.setValue(varb, 1)
    state.setValue(varc, 3)
    val = M_Expression(expr, state)
    print(val)

if __name__ == "__main__":
    main1()
```

Explain the expression in the main1 function of simple interpreter

$b + 3 * c$ $\{\langle b, 1\rangle, \langle c, 3\rangle\}$



- The AST is generated by the syntactic analyzer in the compilation.
- The structure of the tree defines the precedence of the operators.
- Then the semantic analyzer will check the semantic errors of it. The compiler will interpret each node of the tree and evaluate it based on the semantic rules.
- Each node in the tree is a class we defined in the simple interpreter. So, we create a Value object for 3, a Variable object for each of the two variables b and c.

- Then, following the precedence and the semantic rule for binary expressions, we create a binary expression object for the right subtree 3 * c and using this object coupled with the Variable object b to create another binary expression object b + 3 * c.
- In this way, the meaning of the subtree 3 * c will be determined before determining the meaning for b + 3 * c.
- This also explains how the recursion works here. The meaning function of binary expression evaluate the expression b + 3 * c. It will need to figure out the meaning of the left term b and the meaning of the right term 3 * c first. The right term 3 * c is another binary expression, whose meaning depends on the left term 3 and the right term c.
- So the process of the meaning function of binary expression to determine the meaning of the expression b + 3 * c is to depth-first traverse to the evaluate the meaning of leaves and subtrees before determining the meaning of the expression.