

Semantics of Expressions

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 code up the meaning of expression when it is a variable, value, and binary in this course.*

Expression = Variable | Value | Binary | Unary

- For expressions, the result is not a new state, it is just a value from a mathematical set.
- 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*

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
    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:
        print("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)

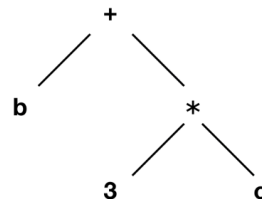
if __name__ == "__main__":
    main1()

```

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. It 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.

Side Effects

- *Note: Our discussion above assumes there are no side effects caused by evaluating the expressions.*
- A **side effect** occurs during the evaluation of an expression if, **in addition to returning a value**, the expression **alters the state** of the program.
- A typically example is the post- and pre- increment. $x = 1; ++x + x++;$
- *If there are side effects, the above semantics are not specific enough.*
- The way to address side effects has two steps:

Update the meaning function. Instead of producing a value, it returns a value and a state.

$$M : Expression \times State \rightarrow Value \times State$$

- The meaning rule should be more specified. Take the binary expression for instance, the meaning rule also define the order to determine the value of two terms.

If the Expression is a Binary, **the meaning of term1 in the current state is first determined**, giving a value $v1$ and a state $s1$. **Then the meaning of term2 in state $s1$ is determined**, giving a value $v2$ and a state $s2$. Then **Meaning Rule of Binary Expression** determines the meaning of the expression by *applying the Operator op to the resulting values $v1$ and $v2$ in state $s2$* , and the resulting state is $s2$.

- *Note: We need to know the definition of side effect and the ways C and Java use to handle it. The implementation of side effects in meaning functions is out of the scope of the course.*

Pre- and Post-Increment can generate Undefined Behavior in C

- Like the dangling else problem, when pre- and post-increment are evaluated is something that needs to be handled correctly by the code-generator/interpreter. C does not specify exactly when these operations should take place, so different compilers can lead to different outcomes.
- Example:
 - Let `int x = 3; x += x++ + ++x;` This is legal in C and Java, so the syntax is correct.

- What about the semantics? After the two statements have been executed, what is the value of x? And does it matter if we consider it part of a C program or part of a Java program?
 - Yes, it matters.
 - If it is a C program, the value of x depends on the compiler.
 - If it is a **Java** program, the value of x is **precisely defined**, which is 11. $x = 3 + 3 + 5$.
- Java: **Increment and decrement** operators have a **higher precedence than** the other **mathematical operators**. They are applied from **left to right in the order in which they occur** unless one of them has been given a higher precedence through the parentheses.
- Example:
 - `if (x++ == x++) {System.out.println("Yes");}`
This condition is always false in Java.
 - `if (++x == x++) {System.out.println("Yes");}`
This condition is always true in Java.
- C Example:
 - Here is a C program that declares an int with value 4, and increment it and assignment it to itself. (show funnyIncrement.c)

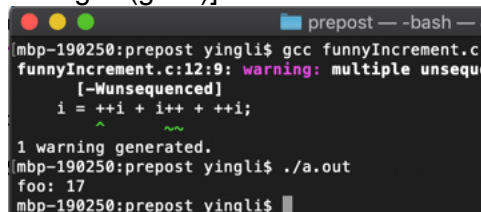
```
// funnyIncrement.c
//
// Created by Ying Li on 10/6/19.
//

#include <stdio.h>

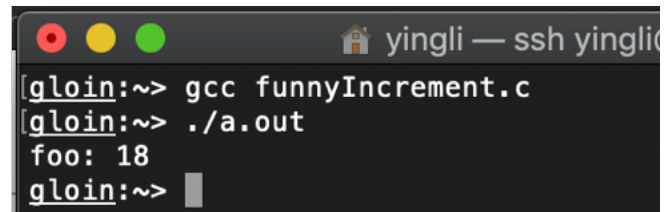
void foo () {
    int i = 4;
    i = ++i + i++ + ++i;
    printf("foo: %d\n", i);
}

int main () {
    foo();
    return 0;
}
```

- What are the possible outputs of the program? [17 from clang (Ying's mac), 18 from gcc (gloin)]

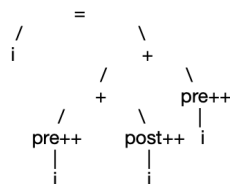


```
mbp-190250:prepost yingli$ gcc funnyIncrement.c
funnyIncrement.c:12:9: warning: multiple unsequenced operations on 'i' [-Wunsequenced]
    i = ++i + i++ + ++i;
          ^
1 warning generated.
mbp-190250:prepost yingli$ ./a.out
foo: 17
mbp-190250:prepost yingli$
```



```
gloin:~> gcc funnyIncrement.c
gloin:~> ./a.out
foo: 18
gloin:~>
```

- The confusion here is when to execute the pre-increment operators.
- If we draw a simplified concrete syntax tree, it will be a tree like this



- Unlike operator precedence, we cannot infer the order of operations here directly from the tree.

- C language specification **does not define** the behavior of the program where there is **more than one pre- or post- increment operator applied to a single variable within a single expression**.
- The behaviors end up being **compiler specific**. The semantic meaning of the program is undefined.