Demystifying Expressions in Project 1

March 16, 2021

Many of you had questions and misunderstandings about the role of the "expression" module in our template codebase. This post aims to clarify these questions and describe expressions in more technical detail.

In order to execute an SMC protocol, all parties need to agree on the arithmetic circuit they are computing. Thus, in your implementation you need to be able to represent arithmetic circuits in a convenient way, for some definition of convenient. This is what the expression module template is for. An expression is a representation of an arithmetic circuit that can be passed around and is easy to work with.

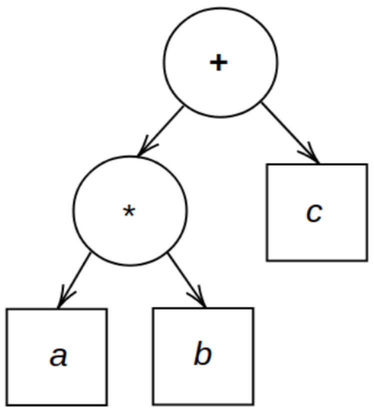
Expressions cannot be immediately interpreted

An arithmetic circuit is a function, not *a value* of a function. For example, f(a, b) = a + b. When an expression representing f(a, b) is constructed, you cannot compute its concrete output: the values of a and b are simply not known at construction time. Thus, in an implementation of expressions, you have to represent the *structure* of f(a, b) in an abstract way, not its concrete output value.

Expressions are abstract syntax trees

We now know that an expression should somehow encode the structure of an arithmetic circuit. How can this be done? In fact, the only way to do this in a way that is compatible with our integration tests is to build an abstract syntax tree. An abstract syntax tree is a tree data structure that is used internally by compilers and interpreters to represent a structure of an expression in a (programming) language. This is, in fact, exactly what we want to do! We have a special language of arithmetic circuits, and we want to be able to represent them in a convenient way.

Consider a circuit f(a, b, c) = a * b + c. It can be represented using the following abstract syntax tree:



The intermediate nodes of the tree (circles) represent operations, and the leaf nodes (squares) contain the terms—the most basic atoms—of our language. In this example, the terms are variables such as a, b, c.

How can this be implemented in Python? The simplest way to do so is the following:

```
# Intermediate tree node representing addition operation
class AddOp:
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Intermediate tree node representing multiplication operation
class MultOp:
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Leaf node representing a variable
class Variable:
    def __init__(self, name=None):
        self.name = name
```

Using this implementation we can represent f(a, b, c) = a * b + c as follows:

```
AddOp(MultOp(Variable("a"), Variable("b")), Variable("c"))
```

This basic implementation is already sufficient to represent any arithmetic circuit in Python. However, an implementation like this would not pass our integration tests as we expect a slightly different interface for the creation of circuits. In fact, we expect a more intuitive way to create circuits that the Python language supports: we want to be able to write an expression in natural notation. For example, we want to simply write `a * b + c` to represent f(a, b, c) = a * b + c. How to enable this way of constructing expressions? First, we have to leverage Python's operator overriding using `__add__` and `__mul__` methods. Second, we have to use an abstract base class for all expressions:

```
# This is the base class for all expressions.
# Similar to an abstract class in Java or C++.
class Expression:
    def __add__(self, other):
        return AddOp(self, other)
    def __mul__(self, other):
        return MultOp(self, other)

# Intermediate tree node representing addition operation
# (Note it is now an instance of Expression)
class AddOp(Expression):
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Intermediate tree node representing multiplication operation.
# (Note it is now an instance of Expression)
class MultOp(Expression):
    def __init__(self, a, b):
        self.a = a
        self.b = b

# Leaf node representing a variable.
# (Note it is now an instance of Expression)
class Variable(Expression):
    def __init__(self, name=None):
        self.name = name
```

Now, we are able to define f(a, b, c) = a * b + c using the natural notation:

```
Variable("a") * Variable("b") + Variable("c")
```

This is not the only way to implement the functionality, but rather the most idiomatic (standard) for Python.

This example shows a construction of an abstract syntax tree that represents arithmetic operations on some “variables.” Here, *Variable* is the most basic term of a language. In the case of our SMC protocol, the terms are a *Scalar* and a *Secret*, thus the tree implementation has to be slightly adapted.

How to use expressions in a run of the protocol

To run the protocol, *each party* has to execute an appropriate sub-protocol (addition, scalar addition, scalar multiplication, multiplication using the Beaver scheme) for *each operation* in the expression. How do we traverse expressions to run a sub-protocol for every operation?

To traverse an expression, we run a tree traversal algorithm: starting from the root node, we recursively visit each sub-expression until we reach the leaves of the tree. An idiomatic way to do so in Python uses the so-called "visitor" pattern:

```python
def traverse(expr):
    if isinstance(expr, AddOp):
        # Evaluate the addition, and
        # traverse further by calling traverse(expr.a) and traverse(expr.b)
        return value

    if isinstance(expr, MultOp):
        # Evaluate the multiplication,
        # traverse further by calling traverse(expr.a) and traverse(expr.b)
        return value

    if isinstance(expr, Variable):
        # Handle the base of recursion.
        # For example, retrieve the value assigned to the variable from somewhere.
        return value
```

In our case, we do not have *Variables* as leaves—we have Secrets and Scalars that are treated differently, and we do not simply “evaluate” additions or multiplications—rather we run SMC sub-protocols.