

# Assignment 3: floats, bools, and polymorphism

ECEN 4553 & 5013, CSCI 4555 & 5525  
Prof. Jeremy G. Siek

The main concepts for this chapter are

- Polymorphism in dynamically typed languages
- Type-based dynamic dispatch

## 1 Syntax of $P_1$

The  $P_0$  subset of Python only dealt with one kind of entity: plain integers. In this chapter we add floating-point numbers and Booleans. We also add in a few more operations that return or use Boolean values, creating what we call the  $P_1$  subset of Python. The concrete syntax for  $P_1$  is shown in Figure 1. In addition, all of the syntax from  $P_0$  is carried over to  $P_1$  unchanged.

```
unaryop ::= "not"
binaryop ::= "<" | ">" | "==" | ">=" | "<=" | "!=" | "or" | "and"
expression ::= float | "True" | "False"
              | expression "if" expression "else" expression
```

Figure 1: Concrete syntax for the  $P_1$  subset of Python. (In addition to the syntax of  $P_0$ .)

## 2 Semantics of $P_1$

All of the arithmetic operations of Python work on integers, floating point numbers, and even Booleans. True is treated as if it were the integer 1 and False is treated as 0. Furthermore, numbers can be used in places where Booleans are expected. The number 0 is treated as False and everything else is treated as True. Here are a few examples:

```
>>> False < True
True
>>> 2 if 0.1 else 3
2
>>> 0 or 0
0
>>> False or False
False
```

An unusual feature of Python is that it provides succinct syntax for sequences of comparisons. For example, instead of writing

```
>>> 1 < 2 and 2 < 3
True
```

you can write

```
>>> 1 < 2 < 3
True
```

Compiling the normal two-argument comparisons to C is straightforward, but compiling the chained comparisons is more difficult, and is left as an extra credit exercise.

**Exercise 2.1.** Read the sections of the Python Reference Manual that apply to  $P_1$ : 3.2, 5.2.2, 5.9, and 5.10.

**Exercise 2.2.** Write five programs in the  $P_1$  subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs. Add your test programs to the `test` directory (which should include the tests from the previous assignment) or add them to the `test_extra` directory if the test uses chained comparisons, and, or or.

### 3 Compiling Polymorphism

One of the defining characteristics of Python, and part of what makes it a dynamic language, is that a Python expression may result in different types of objects and that the type may be determined during program execution (at run-time). In general, the ability of a language to allow multiple types of values to be returned from the same expression, or be stored at the same location in memory, is called *polymorphism*.

<b>pol•y•mor•phism</b> noun the occurrence of something in several different forms
--

The term “polymorphism” can be remembered from its Greek roots: “poly” means “many” and “morph” means “form”.

The following is an example of polymorphism in Python.

```
3 if randint(0,1) else 3.14159
```

This expression sometimes results in the integer 3 and sometimes in the floating-point number 3.14159.

```
>>> from random import randint
>>> 3 if randint(0,1) else 3.14159
3
>>> 3 if randint(0,1) else 3.14159
3
>>> 3 if randint(0,1) else 3.14159
3.1415899999999999
```

In contrast, the analogous expression in the C language always returns a floating-point number (either 3.0 or 3.14159).

```
rand() ? 3 : 3.14159
```

In C, the 3 is implicitly cast to a float so that the result will be of just one type. The lack of implicit polymorphism in C represents a small challenge; how can we compile away the polymorphism inherent in Python?

Figure 2 shows the additional Python classes used to represent the AST nodes of  $P_1$ . Several of these classes deserve some explanation. First, there is not a specific class for floating-point numbers; instead the `Const` class is used for floats as well as integers. The `Name` class is for variables, which we have not yet discussed, but Python represents `True` and `False` as variables with names `'True'` and `'False'`. The `Compare` class is for representing comparisons such as `<` and `>`. The `expr` attribute of `Compare` is for the first argument and the `ops` member contains a list of pairs, where the first item of each pair is a string specifying the operation, such as `'<'`, and the second item is the argument. The `Or` and `And` classes each contain a list of arguments, held in the `nodes` attribute.

```
class Name(Node):
    def __init__(self, name):
        self.name = name
class Compare(Node):
    def __init__(self, expr, ops):
        self.expr = expr
        self.ops = ops
class Or(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class And(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Not(Node):
    def __init__(self, expr):
        self.expr = expr
class IfExp(Node):
    def __init__(self, test, then, else_):
        self.test = test
        self.then = then
        self.else_ = else_
```

Figure 2: The Python classes for  $P_1$  AST nodes.

**Exercise 3.1.** Extend your compiler to handle the  $P_1$  subset of Python. You may use the parser from Python's `compiler` module, or for extra credit you can extend your own parser. Don't worry about throwing exceptions in erroneous situations, just print an error message. The chained comparison, `and`, and `or` expressions are tricky to compile so they are extra credit. Test your compiler against the programs in `test` and for extra credit, `test_extra`.