Assignment 3: floats, bools, and polymorphism

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

The main concepts in this assignment are

- Polymorphism in dynamically typed languages and type-based dynamic dispatch
- Order of evaluation
- Translating from one intermediate language to another.

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 and we expand the built-in `input` function to handle Boolean and floating-point constants, 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.

```plaintext
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:

```python
>>> False < True  # True
>>> 2 if 0.1 else 3  # 2
>>> False or False  # False
>>> 1 and 2  # 2
>>> 1 or 2  # 1
```

Note that the result of a logic operation such as and or or does not necessarily return a Boolean value. Instead, $e_1$ and $e_2$ evaluates expression $e_1$ to a value $v_1$. If $v_1$ is equivalent to False, the result of the and is $v_1$. Otherwise $e_2$ is evaluated to $v_2$ and $v_2$ is the result of the and. The or operation works in a similar way except that it checks whether $v_1$ is equivalent to True.

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

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

you can write

```python
>>> 1 < 2 < 3  # True
```

**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 at least 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).
3 New Python AST Classes

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):
    # for Boolean literals and for the 'input' function
    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.

4 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.
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*.

**polymorphism**  
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.

```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.

```python
>>> 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).

```c
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?

## 5 Compiling chained comparisons

Compiling the normal two-argument comparisons to C is straightforward, but compiling the chained comparisons is more challenging because C does not support chained comparisons and because expressions in the middle of a chain of comparisons are only evaluated once. It is therefore incorrect to duplicate the middle expressions when compiling to C. For example, the following translation is incorrect.

```python
1 < input() < 3
```

```c
1 < input() && input() < 3  /* incorrect translation */
```
To translate chained comparisons to C, we need a way to evaluate an expression once and use the result in multiple places. The most straightforward language feature that could provide this ability is something called a let expression. A let expression has the form

let variable = expression in expression

The behavior of a let expression is to evaluate the first expression and bind the result in the variable. The second expression is then evaluated with the variable in scope. The result of the entire let expression is the result of the second expression. Consider the following example

let x = input() in 1 < x && x < 3

If the input is 2, then this evaluates to True. Unfortunately, C does not provide let expressions. However, GNU C includes an extension called statement expressions that provide the same functionality (and more).

```python
class Let(Node):
    def __init__(self, var, rhs, body):
        self.var = var
        self.rhs = rhs
        self.body = body

class PrimitiveOp(Node):
    def __init__(self, name, nodes):
        self.name = name
        self.nodes = nodes
```

Figure 3: New Python classes for IR$_0$.

Exercise 5.1. 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.

Python uses a different format for printing floating point numbers than the default format used by the C printf function. A C function that mimics the Python formatting is available on the course web page.

Write a new pass in your compiler that goes between parsing and translating to C. The output of the new pass should be an AST in an intermediate representation we will call IR$_0$. This IR differs from the $P_1$ AST in that it adds the Let and PrimitiveOp classes shown in Figure 3 and it removes the following classes:

- Add, Sub, Mul, Div, Mod, Power, UnaryAdd, UnarySub
• And, Or, Not
• Compare
• CallFunc (which was used for the input function)

The semantics of the Let feature is described above. The PrimitiveOp node is meant as a replacement for the removed arithmetic, logic, comparison, and input operations. The semantics of a PrimitiveOp node is straightforward: it evaluates all of the expressions in its nodes attribute and then applies the operation indicated by the name attribute. The logic and comparison primitive operations return a Boolean. The input function can return a Boolean, integer, or floating-point number depending on the content of the input. Your compiler should follow the overall organization shown in Figure 4. The shaded boxes are the focus of this assignment.

Figure 4: Overview of the compiler organization.