Assignment 1: Integer expressions and abstract syntax trees

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

The main concepts for this week are

- abstract syntax trees (AST),
- recursive functions over ASTs (to generate C code), and
- test-driven development.

1 Syntax of P_0

The first subset of Python that we consider is extremely simple: it consists of print statements with integers and arithmetic operations. We call this subset P_0. The following is an example P_0 program.

```
print 1 - 2 + 3 / -4
```

In general, the syntax of the source code for a language is called its *concrete syntax*. The concrete syntax of P_0 specifies which programs, expressed as sequences of characters, are P_0 programs. The concrete syntax of P_0 is shown in Figure 1. (If you need a refresher on how a context-free grammar specifies a language, read Section 3.1 of [1].)

```
program ::= module
module ::= simple_statement+
simple_statement ::= "print" expression ("," expression)*
unary_op ::= "+" | "-"
binary_op ::= "+" | "-" | "*" | "/" | "%" | "**"
expression ::= integer
             | unary_op expression
             | expression binary_op expression
             | "(" expression ")"
```

Figure 1: Concrete syntax for the P_0 subset of Python.

It would be tedious to write a compiler in terms of sequences of characters. Fortunately, there is a standard Python library that parses a sequence of characters into a much more useful data-structure, an *abstract syntax tree* (AST), which is an in-memory structure that represents a program at a more conceptual level, with a node for each expression in the program. The structure forms a tree, with each node containing references to the nodes for its subexpressions. Figure 2 shows the abstract syntax tree for the above example P_0 program. The following interaction with the `python` interpreter shows a call to the Python parser and the textual representation of the resulting AST.

```
>>> import compiler
>>> ast = compiler.parse("print 1 - 2 + 3 / -4")
>>> ast
Module(None,
       Stmt([Printnl([Add((Sub((Const(1), Const(2))),
                           Div((Const(3), UnarySub(Const(4))))))],
                     None)]))
```

The AST produced by the Python parser is represented using several classes, one for each kind of node in the AST. Thus, each node in the AST is represented by an object, an instance of one of these classes. Figure 3 shows the Python classes for representing the AST nodes of the P_0 subset of Python.

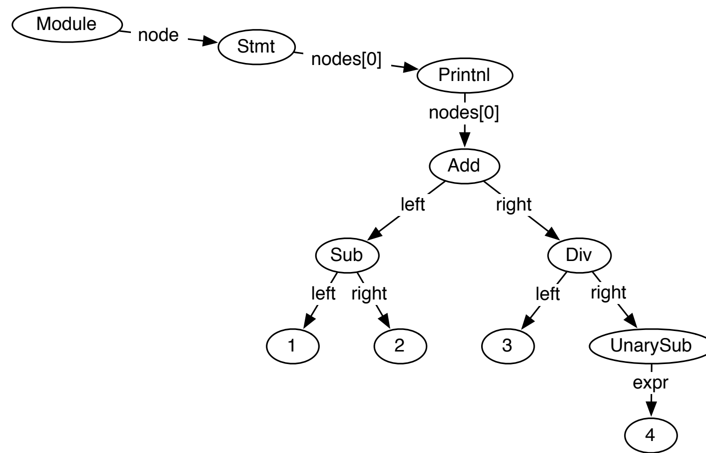

Figure 2: The abstract syntax tree for `print 1 - 2 + 3 / -4`.

```
class Module(Node):
    def __init__(self, doc, node):
        self.doc = doc
        self.node = node
class Stmt(Node):
    def __init__(self, nodes):
        self.nodes = nodes
class Printnl(Node):
    def __init__(self, nodes, dest):
        self.nodes = nodes
        self.dest = dest
class Const(Node):
    def __init__(self, value):
        self.value = value
class UnaryAdd(Node):
    def __init__(self, expr):
        self.expr = expr
class UnarySub(Node):
    def __init__(self, expr):
        self.expr = expr
class Add(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class Sub(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class Mul(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class Div(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class Mod(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
class Power(Node):
    def __init__(self, (left, right)):
        self.left = left
        self.right = right
```

Figure 3: The Python classes for representing P_0 ASTs.

2 Semantics of P_0

The *meaning* of Python programs, that is, what happens when you run a Python program, is defined in the Python Reference Manual [3].

Exercise 2.1. Read the sections of the Python Reference Manual that apply to P_0: 3.2, 5.2.2, 5.4, 5.5, 5.6, and 6.6.

Sometimes it is difficult to understand the technical jargon in programming language reference manuals. Another way to learn about the meaning of Python programs, and perhaps a more fun way, is to experiment with the standard `python` interpreter. If there is an aspect of the language that you don't understand, create a program that uses that aspect and run it! Suppose you're not sure about a particular feature but have a guess, a *hypothesis*, about how it works. Think of a program that will produce one output if your hypothesis is correct and produce a different output if your hypothesis is incorrect. You can then run the `python` interpreter to validate or disprove your hypothesis.

For example, suppose that you are not sure what happens when the result of an arithmetic operation results in a very large integer, an integer too large to be stored in a machine register ($> 2^{31} - 1$). In the language C integer operations wrap around, so 2×2^{30} produces -2147483648. Does the same thing happen in Python? Let's try it and see:

```
>>> 2 * 2**30
2147483648L
```

No, the number does not wrap around. Instead, Python has two kinds of integers: *plain integers* for integers in the range -2^{31} to $2^{31} - 1$ and *long integers* for integers in a range that is only limited by the amount of (virtual) memory in your computer.

> *For P_0 we restrict our attention to just plain integers and say that operations that result in integers outside of the range -2^{31} to $2^{31} - 1$ are undefined.*

Exercise 2.2. Write five programs in the P_0 subset of Python that help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs. Run the programs using the standard `python` interpreter. Add your test programs to the `test` directory.

3 Recursive functions and compiling to C

The main programming technique for analyzing and manipulating ASTs is to write recursive functions that perform traversals of the tree. The `ast_to_string` function in Figure 4 is a typical example. The function checks what kind of AST node it is dealing with and makes a recursive call to generate strings for the children nodes. It then collects the results to form the string for the current AST node.

There are several Python functions that are particularly useful when dealing with ASTs. The `map` function, `filter`, and `reduce` functions are great for handling nodes that have a *list* of children, as is the case for `Stmt` and `Printnl`. You can find out about these functions in the section on Functional Programming Tools in the Python Tutorial [4]. Another useful tool is the `join` method for strings, which is documented under String Methods in the Python Library Manual [2].

```
from compiler.ast import *

def ast_to_string(ast):
  if isinstance(ast, Module):
    return 'Module(None, ' + ast_to_string(ast.node) + ')'
  elif isinstance(ast, Stmt):
    return 'Stmt([' + ', '.join(map(ast_to_string, ast.nodes)) + '])'
  elif isinstance(ast, Printnl):
    return 'Printnl([' + ', '.join(map(ast_to_string, ast.nodes)) + '], None)'
  elif isinstance(ast, Const):
    return '%d' % ast.value
  elif isinstance(ast, UnarySub):
    return 'UnarySub(' + ast_to_string(ast.expr) + ')'
  elif isinstance(ast, Add):
    return 'Add(' + ast_to_string(ast.left) \
             + ', ' + ast_to_string(ast.right) + ')'
  ...
```

Figure 4: Recursive function that traverses an AST and produces a string representation.

(Note: Change the example to `num_nodes` and use list comprehensions instead of map and filter.)

Exercise 3.1. Create a Python script named `compile.py` that reads in a file that containing a P_0 program and compiles it to a C file. The script must take a command-line argument that specifies the input file (a file with a `.py` extension) and the script must output the resulting C program standard output (stdout), for example by using the Pyhon `print` statement. Use *test-driven development* to create the compiler. Initially, the compiler should do nothing, and just output an empty file. Use the provided script `run_tests.py` to compare the output of the generated C files to that of the standard `python` interpreter for all the test programs in directory `test`. Feel free to modify and customize the script as necessary. The compiler should initially fail all of the tests. Now begin to write a recursive function that translates a P_0 AST to the C programming language. Write the function incrementally, adding one or two cases at a time and re-running the tests to make sure that the new cases that you've added make the appropriate tests begin to succeed. The compiler is finished when all of the tests succeed.

References

[1] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003.

[2] G. van Rossum. *Python Library Reference*. Python Software Foundation, `http://docs.python.org/lib/lib.html`, 2.5 edition, September 2006.

[3] G. van Rossum. *Python Reference Manual*. Python Software Foundation, `http://docs.python.org/ref/ref.html`, 2.5 edition, September 2006.

[4] G. van Rossum. *Python Tutorial*. Python Software Foundation, `http://docs.python.org/tut/tut.html`, 2.5 edition, September 2006.