CHAPTER 1

Integers and variables

The main concepts in this chapter are

abstract syntax trees: Inside the compiler, we represent programs with a data-structure called an abstract syntax tree, which is abbreviated as AST.

recursive functions: We analyze and manipulate abstract syntax trees with recursive functions.

flattening expressions into instructions: An important step in compiling high-level languages to low-level languages is flattening expressions trees into lists of instructions.

selecting instructions: The x86 assembly language offers a peculiar variety of instructions, so selecting which instructions are needed to get the job done is not always easy.

test-driven development: A compiler is a large piece of software. To maximize our productivity (and minimize bugs!) we use good software engineering practices, such as writing lots of good test cases before writing code.

1.1. ASTs and the $P_0$ subset of Python

The first subset of Python that we consider is extremely simple: it consists of print statements, assignment statements, some integer arithmetic, and the input() function. We call this subset $P_0$. The following is an example $P_0$ program.

```
print - input() + input()
```

Programs are written one character at a time but we, as programmers, do not think of programs as sequences of characters. We think about programs in chunks like if statements and for loops. These chunks often have parts, for example, an if statement has a then-clause and an else-clause. Inside the compiler, we often traverse over a program one chunk at a time, going from parent chunks to their children. A data-structure that facilitates this kind of traversal is a tree. Each node in the tree represents a programming language construct and each node has edges that point to its children. When
a tree is used to represent a program, we call it an *abstract syntax tree* (AST). While trees in nature grow up with their leaves at the top, we think of ASTs as growing down with the leaves at the bottom.

![Diagram of an abstract syntax tree](image)

**Figure 1.** The abstract syntax tree for `print - input() + input()`.

Figure 1 shows the abstract syntax tree for the above $P_0$ program. Fortunately, there is a standard Python library that turns a sequence of characters into an AST, using a process called *parsing* which we learn about in the next chapter. The following interaction with the Python interpreter shows a call to the Python parser.

```python
>>> import compiler
>>> ast = compiler.parse("print - input() + input()")
>>> ast
Module(None,
       Stmt([Printl([Add((UnarySub(CallFunc(Name('input'), [], None, None)),
                        CallFunc(Name('input'), [], None, None))],
              None)]))
```

Each node in the AST is a Python object. The objects are instances of Python classes; there is one class for each kind of node. Figure 2 shows the Python classes for the AST nodes for $P_0$. For each class, we have only listed only listed its constructor, the `__init__` method. You can find out more about these classes by reading the file `compiler/ast.py` of the Python installation on your computer.

To keep things simple, we place some restrictions on $P_0$. In a `print` statement, instead of multiple things to print, as in Python 2.x, you only need to support printing one thing. So you can assume
the nodes attribute of a Printnl node is a list containing a single AST node. Similarly, for expression statements you only need to support a single expression, so you do not need to support tuples. $P_0$ only includes basic assignments instead of the much more general forms supported by Python 2.x. You only need to support a single variable on the left-hand-side. So the nodes attribute of Assign is a list containing a single AssName node whose flag attribute is OP_ASSIGN. The only kind of value allowed inside of a Const node is an integer. $P_0$ does not include support for Boolean values, so a $P_0$ AST will never have a Name node whose name attribute is “True” or “False”.

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 Assign(Node):
    def __init__(self, nodes, expr):
        self.nodes = nodes
        self.expr = expr
class AssName(Node):
    def __init__(self, name, flags):
        self.name = name
        self.flags = flags
class Discard(Node):
    def __init__(self, expr):
        self.expr = expr

class Const(Node):
    def __init__(self, value):
        self.value = value
class Name(Node):
    def __init__(self, name):
        self.name = name
class Add(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right
class UnarySub(Node):
    def __init__(self, expr):
        self.expr = expr
class CallFunc(Node):
    def __init__(self, node, args):
        self.node = node
        self.args = args

Figure 2. The Python classes for representing $P_0$ ASTs.

1.2. Understand the meaning of $P_0$

The meaning of Python programs, that is, what happens when you run a program, is defined in the Python Reference Manual [20].

Exercise 1.1. Read the sections of the Python Reference Manual that apply to $P_0$: 3.2, 5.5, 5.6, 6.1, 6.2, and 6.6. Also read the entry for the input function in the Python Library Reference, in section 2.1.

Sometimes it is difficult to understand the technical jargon in programming language reference manuals. A complementary way to learn about the meaning of Python programs is to experiment with
the standard Python interpreter. If there is an aspect of the language that you do not understand, create a program that uses that aspect and run it! Suppose you are 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 \( 2 \times 2^{30} \) produces \(-2147483648\) [13]. Does the same thing happen in Python? Let us 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 \(-2^{31} \text{ to } 2^{31} - 1\) 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 \(-2^{31} \text{ to } 2^{31} - 1\) are undefined.

The built-in Python function `input()` reads in a line from standard input (stdin) and then interprets the string as if it were a Python expression, using the built-in `eval` function. For \(P_0\) we only require a subset of this functionality. The `input` function need only deal with integer literals. A call to the input function, of the form "input()", is parsed into the function call AST node `CallFunc`. You do not need to handle general function calls, just recognize the special case of a function call where the function being called is named "input".

**Exercise 1.2.** Write some programs in the \(P_0\) subset of Python. The programs should be chosen to help you understand the language. Look for corner cases or unusual aspects of the language to test in your programs. Later in this assignment you will use these programs to test your compiler, so the tests should be thorough and should exercise all the features of \(P_0\). If the tests are not thorough, then your compiler may pass all the tests but still have bugs that will be caught when your compiler is tested by the automatic grader. Run the programs using the standard Python interpreter.
1.3. Write recursive functions

The main programming technique for analyzing and manipulating ASTs is to write recursive functions that traverse the tree. As an example, we create a function called num_nodes that counts the number of nodes in an AST. Figure 3 shows a schematic of how this function works. Each triangle represents a call to num_nodes and is responsible for counting the number of nodes in the sub-tree whose root is the argument to num_nodes. In the figure, the largest triangle is responsible for counting the number of nodes in the sub-tree rooted at Add. The key to writing a recursive function is to be lazy! Let the recursion do the work for you. Just process one node and let the recursion handle the children. In Figure 3 we make the recursive calls num_nodes(left) and num_nodes(right) to count the nodes in the child sub-trees. All we have to do to then is to add the two numbers and add one more to count the current node. Figure 4 shows the definition of the num_nodes function.

![Schematic for a recursive function processing an AST.](image)

When a node has a list of children, as is the case for Stmt, a convenient way to process the children is to use List Comprehensions described in the Python Tutorial [21]. A list comprehension has the following form

\[
[\text{compute for variable in list}]
\]

This performs the specified computation for each element in the list, resulting in a list holding the results of the computations. For example, in Figure 4 in the case for Stmt we write

\[
[\text{num_nodes}(x) \text{ for } x \text{ in n.nodes}]
\]
from compiler.ast import *

def num_nodes(n):
    if isinstance(n, Module):
        return 1 + num_nodes(n.node)
    elif isinstance(n, Stmt):
        return 1 + sum([num_nodes(x) for x in n.nodes])
    elif isinstance(n, Printnl):
        return 1 + num_nodes(n.nodes[0])
    elif isinstance(n, Assign):
        return 1 + num_nodes(n.nodes[0]) + num_nodes(n.expr)
    elif isinstance(n, AssName):
        return 1
    elif isinstance(n, Discard):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, Const):
        return 1
    elif isinstance(n, Name):
        return 1
    elif isinstance(n, Add):
        return 1 + num_nodes(n.left) + num_nodes(n.right)
    elif isinstance(n, UnarySub):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, CallFunc):
        return 1
    else:
        raise Exception('Error in num_nodes: unrecognized AST node')

Figure 4. Recursive function that counts the number of nodes in an AST.

This makes a recursive call to num_nodes for each child node in the list n.nodes. The result of this list comprehension is a list of numbers. The complete code for handling a Stmt node in Figure 4 is

    return 1 + sum([num_nodes(x) for x in n.nodes])

As is typical in a recursive function, after making the recursive calls to the children, there is some work left to do. We add up the number of nodes from the children using the sum function, which is documented under Built-in Functions in the Python Library Manual [19]. We then add 1 to account for the Stmt node itself.

There are 11 if statements in the num_nodes function, one for each kind of AST node. In general, when writing a recursive function over an AST, it is good to double check and make sure that you have written one if for each kind of AST node. The raise of an exception in the else checks that the input does not contain any other node kind.