To keep things simple, we place some restrictions on $P_0$. In a print statement, instead of multiple things to print, as in Python 2.5, 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 2.5. 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”.

```python
class Module(Node):
    def __init__(self, doc, node):
        self.doc = doc
        self.node = node

class Stmt(Node):
    def __init__(self, value):
        self.value = value

class Printnl(Node):
    def __init__(self, nodes, dest):
        self.nodes = nodes
        self.dest = dest

class Assign(Node):
    def __init__(self, nodes, exprs):
        self.nodes = nodes
        self.exprs = exprs

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 UnarySub(Node):
    def __init__(self, value):
        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:

```python
>>> 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.

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 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.

```
num_nodes(Add)
```

```
num_nodes(left)
```

```
num_nodes(right)
```

```
[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

```
[num_nodes(x) for x 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, Assign):
        return 1 + num_nodes(n.nodes[0])
    elif isinstance(n, Discard):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, Const):
        return 1 + num_nodes(n.expr)
    elif isinstance(n, Names):
        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.
1.4. Learn the x86 assembly language

This section gives a brief introduction to the x86 assembly language. There are two variations on the syntax for x86 assembly: the Intel syntax and the AT&T syntax. Here we use the AT&T syntax, which is accepted by the GNU Assembler and by gcc. The main difference between the AT&T syntax and the Intel syntax is that in AT&T syntax the destination register on the right, whereas in Intel syntax it is on the left.

The x86 assembly language consists of hundreds of instructions and many intricate details. However, for our current purposes we can focus on a tiny subset of the language. The program in Figure 5 serves to give a first taste of x86 assembly. This program is equivalent to the following Python program, a small variation on the one we discussed earlier.

```python
x = - input()
print x + input()
```

Perhaps the most obvious difference between Python and x86 is that Python allows expressions to be nested within one another. In contrast, an x86 program consists of a flat sequence of instructions. Another difference is that x86 does not have variables. Instead, it has a fixed set of registers that can each hold 32 bits. The registers have funny three letter names:

```
eax, ebx, ecx, edx, esi, edi, ebp, esp
```

When referring to a register in an instruction, place a percent sign (%) before the name of the register.

When compiling from Python to x86, we may very well have more variables than registers. In such situations we use the stack to store the variables. In the program in Figure 5 the variable `x` has been mapped to a stack location. The register `esp` always contains the address of the item at the front of the stack. In addition to local variables, the stack is also used to pass arguments in a function call. In this course we use the cdecl convention that is used by the GNU C compiler. The instruction `pushl %eax`, which appears before the call to `print_int_nl`, serves to put the result of the addition on the stack so that it can be accessed within the `print_int_nl` function. The stack grows down, so the `pushl` instruction causes `esp` to be lowered by 4 bytes (the size of one 32 bit integer).

Because the stack pointer, `esp`, is constantly changing, it would be difficult to use `esp` for referring to local variables stored on the
The x86 assembly code for the program

```
x = input(); print x + input();
```

stack. Instead, the ebp register (bp is for base pointer) is used for this purpose.

The stack memory is conceptually a stack at two-levels. We have already seen that words (32-bit values) are pushed and popped via the stack pointer. Additionally, each function call pushes an activation record that it uses to store its local state. The function call’s activation record is then popped when it returns. The first two instructions set up of the activation record. In particular, the instruction pushl %ebp saves the current value of the base pointer, that is, the base pointer of the previous activation record. Then, the instruction movl %esp,%ebp puts a copy of the stack pointer into ebp delineating the start of this call’s activation record. We can then use ebp throughout the lifetime of the call to this function to refer to local variables on the stack. And we see that ebp points to the head of a linked-list of activation records. In Figure 5, the value of variable x is referred to by -4(%ebp), which is the assembly way of writing the C expression *(ebp - 4) (the 4 is in bytes). That is, it loads the data from the address stored in ebp minus 4 bytes.

The eax register plays a special role: functions put their return values in eax. For example, in Figure 5 the calls to the input function put their results in eax. It is a good idea not to put anything