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 (AST).

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

flattening expressions into instructions: One 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.

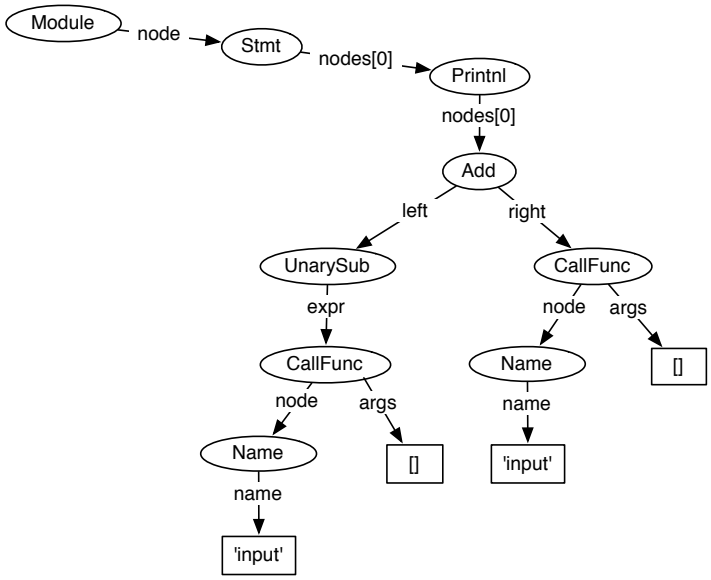

FIGURE 1. The abstract syntax tree for `print - input() + input()`.

Figure 1 shows the abstract syntax tree for the previous P_0 program. 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.

```
>>> import compiler
>>> ast = compiler.parse("print - input() + input()")
>>> ast
Module(None,
       Stmt([Printnl([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 langauge construct. In the above interaction we invoked `compiler.parse`, but in your compiler I recommend using an alternative function that takes it's input from a file: `compiler.parseFile`. Figure 2 shows the Python classes for the AST nodes for P_0. For each class, we have 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 2.7, 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. The P_0 subset only includes basic assignments instead of the much more general forms supported by Python 2.7. 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
# CallFunc is for calls to the 'input' function
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 2×2^{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} 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 your tests but still have bugs that are caught by the automatic grader. Run your test 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.

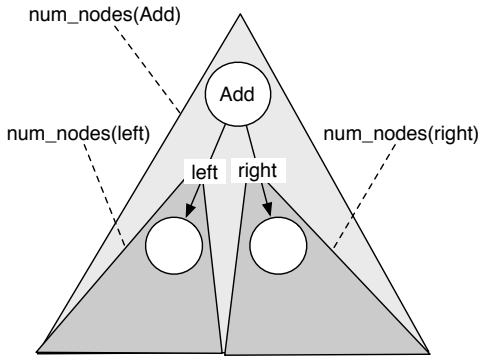

FIGURE 3. Schematic for a recursive function processing an AST.

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

[*compute* `for` *variable* `in` *list*]

```
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 + num_nodes(n.node)
    else:
        raise Exception('Error in num_nodes: unrecognized AST node')
```

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

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]
```

This code 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 kinds of nodes.

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 purposes a tiny subset of the language will do. 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.

```
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. Recall that a stack is a data structure that supports pushing and popping values in a last-in-first-out (LIFO) manner. 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 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 stack. Instead, the `ebp` register (`bp` is for base pointer) is used for this purpose.

The stack is conceptually a two-dimensional stack. 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* to store its local state. The function call's activation record is popped when the function 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 important into `eax` before making a function call, as it will be overwritten. In general, a function may overwrite any of the *caller-save* registers, which are `eax`, `ecx`, and `edx`. The rest of the registers are *callee-save* which means that if a function wants to use those registers, it has to first save them and then restore them before returning. In Figure 5, the first thing that the `main` function does is save the value of the `ebp` register on the stack. The `leave` instruction copies the contents of the `ebp` register into the `esp` register, so `esp` points to the same place in the stack as the base pointer. It then pops the old base pointer into the `ebp` register. Appendix 6.4 is a quick reference for the x86 instructions that you will likely need. (They are the ones I used in the reference compiler.) For a complete list of x86 instructions, see the Intel manuals [**9, 10, 11**].

```
.globl main
main:
        pushl %ebp
        movl %esp, %ebp
        subl $4, %esp
        call input
        negl %eax
        movl %eax, -4(%ebp)
        call input
        addl -4(%ebp), %eax
        pushl %eax
        call print_int_nl
        addl $4, %esp
        movl $0, %eax
        leave
        ret
```

FIGURE 5. The x86 assembly code for the program `x = - input(); print x + input()`.

1.5. Flatten expressions

The first step in translating from P_0 to x86 is to flatten complex expressions into a series of assignment statements. For example, the program

```
print - input() + 2
```

is translated to the following

```
tmp0 = input()
tmp1 = - tmp0
tmp2 = tmp1 + 2
print tmp2
```

In the resulting code, the operands of an expression are either variables or constants, that is, they are *simple expressions*. If an expression has any other kind of operand, then it is a *complex* expression.

EXERCISE 1.3. Write a recursive function that flattens a P_0 program into an equivalent P_0 program that contains no complex expressions. Test that you have not changed the semantics of the program by writing a function that prints the resulting program. Run the program using the standard python interpreter to verify that it gives the the same answers for all of your test cases.

1.6. Select instructions

The next step is to translate the flattened P_0 statements into x86 instructions. For now we will assign all variables to locations on the stack. In chapter 3, we describe a register allocation algorithm that tries to place as many variables as possible into registers.

Figure 6 shows an example translation, selecting x86 instructions to accomplish each P_0 statement. Sometimes several x86 instruction are needed to carry out a Python statement. The translation shown here strives for simplicity over performance. You are encouraged to experiment with better instruction sequences, but it is recommended that you only do that after getting a simple version working. The `print_int_nl` function is provided in a C library, the file `runtime.c` on the course web page.

EXERCISE 1.4. Write a function that translates flattened P_0 programs into x86 assembly. In addition to selecting instructions for the Python statements, you will need to generate a label for the `main` function and the proper prologue and epilogue instructions, which you can copy from Figure 5. The suggested organization of your compiler is shown in Figure 7.

Your compiler should be a (Python) script that takes one argument, the name of the input file, and that produces a file (containing the x86 ouput) with the same name as the input file except that the `.py` suffix should be replaced by the `.s` suffix.

That is, given a wrapper script that invokes your compiler called `pyyc`, invoking

```
$ ./pyyc mytests/test1.py
```

should produce an x86 assembly file `mytests/test1.s`.

```
  tmp0 = input()
  tmp1 = - tmp0
  tmp2 = tmp1 + 2
  print tmp2
⟹
.globl main
main:
        pushl %ebp
        movl %esp, %ebp
        subl $12,%esp           # make stack space for 3 variables

        call input
        movl %eax, -4(%ebp)     # tmp0 is in -4(%ebp)

        movl -4(%ebp), %eax
        negl %eax
        movl %eax, -8(%ebp)     # tmp1 is in -8(%ebp)

        movl -8(%ebp), %eax
        addl $2, %eax
        movl %eax, -12(%ebp)    # tmp2 is in -12(%ebp)

        pushl -12(%ebp)         # push the argument on the stack
        call print_int_nl
        addl $4, %esp           # pop the stack

        movl $0, %eax           # put return value in eax
        leave
        ret
```

FIGURE 6. Example translation to x86 assembly.

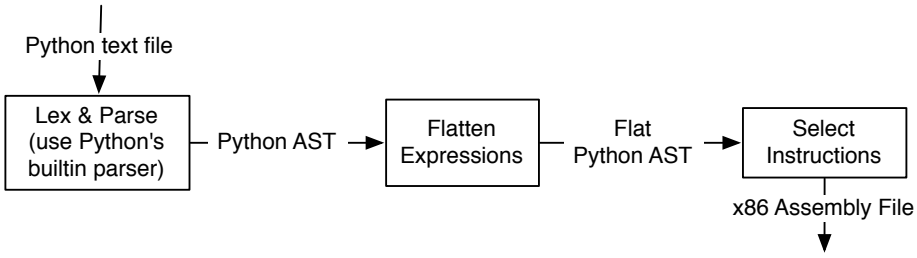

FIGURE 7. Suggested organization of the compiler.