CS212 Unit 3

46

Peter Norvig

Contents

1 CS212 Unit 3

Contents

1.1 1. 1 Introduction

Hi. Welcome back.

As you can see from this spread, this lesson is going to be about tools. Now, we homo sapiens fancy ourselves as tool makers. We think that's what distinguishes us from the other animals. In the physical world, we've invented lots of tools. These general ones like screw drivers and pliers, you're probably very familiar with, and some very specific tools -- look at this one. You may not know what this does, but look how nicely it fits into my hand. Or look at this one. What do you think this one does? If somebody can tell me on the forum what this one does, I'll send them a swell prize. Or look at this one -- surgeons forceps. Now, these have been refined over the years and now we can refine them where they serve as a third hand. It's got this trick where I can attach it on, and now I don't have to put my hand on it anymore, because it's got a little locking mechanism.

So our tools get better and better over time.

Now, there's a saying that a poor craftsman blames his tools. I always thought that all that meant was you shouldn't complain, you shouldn't whine, but I realized it means more than that. It means that the tools become a part of you, and if you're a good craftsman then the tools are a part of you. You know what to do with them, and so you're not going to end up blaming your tools. That's what we're going to learn in this lesson about software tools.

We're going to learn about some general software tools, about some very specific software tools, and specifically, we're going to talk about two.

We're going to talk about, first, language. Now, language is perhaps homo sapien's greatest invention--our greatest tool. In computer programming, well, a computer program is written in a language, but a computer program can also employ language as a tool to do its job.

The other tool we're going to talk about is functions. Of course, you've been using functions all along, but we're going to talk about new techniques for using functions and learn why they're more general and malleable than other tools. So see you in the lesson.

1.2 2. 2 Language

You already know one language we've been using--the Python language obviously. That has a syntax for statements and expressions, but there are also some subparts of Python that seems like quite a different language. One example would be the language for formatting strings. Here's an example from the Python reference language.

```
print '%(language)s has %(#)03d quote types.' % {
            'language': "Python", "#": 2}
```

You see that this rather bizarre string here with %(#)03d and so on. There's a language of format strings itself to produce this type of output.

There are other parts of the Python language that allow you to customize it to your use.

You can create your own type of classes, and they can have their own type of expression, including operator overloading. We're used to expressions like `x + y`. Say if `x` is 2 and `y` is 3 then that would be 5.

But you can also say `x + y` where `x` and `y` are your own types of data types. They might be matrices. They might be components of some kind of building that you're building, and this can say to put them together, it may be two shapes that you can concatenate on the screen. You can define your own language over the types of objects you want to deal with.

You can go one step farther than that and define your own domain specific language.

Here's an example:

This is a language for describing an optimization problem having to do with the price on octane and various types of fuels, and you can describe the parameters and then build the language processor to take this as input and compute the output.

Of course, you could do that with regular Python statements just as well, but here what we've done is design the language specifically for this problem and then written the problem description in that language.

In this unit we'll cover

- what is a language,
- what is a grammar,
- the difference between a compiler and an interpreter, and
- how to use languages as a design tool.

1.3 3. 3 Regular Expressions

We'll start with a language that many of you may be familiar with -- the language of regular expressions.

You've seen them if you've taken CS101. Maybe you've seen them elsewhere. In any event, we'll give an overview of them.

There's a language for regular expressions. They can be expressed as strings.

For example, the string `a*b*c*` describes a language, and that language consists of any number of `a`'s followed by any number of `b`'s followed by any number of `c`'s. Elements of this language include the strings `abc`, `aaa`, `bcc`.

Stars can be any number, so it could be zero of them. Say, just `b` would be an example. The empty string all by itself would be an example. `ccccc` would be an example. An so on.

Now, there's a whole language of symbols like + and ? and so on for regular expressions. To make sense of them, we have to be able to describe what are the possible grammars and then what are the possible languages that those grammars correspond to.

A grammar is a description of a language, and a language is a set of strings.

Now, this form of description of the grammar as a long sequence of characters is convenient when you're quickly typing something in, but it can be difficult to work with. Grammar expressions get long. So we're going to describe the possible grammars in a format that's more compositional.

In other words, what I'm going to describe is an API, which stands for application programming interface. This is the interface that a programmer uses rather than the UI or user interface that a user uses when you click with your mouse.

We'll describe a series of function calls that can be used to describe the grammar of a regular expression. We'll say that a regular expression can be built up by these types of calls.

First, a literal of some string 's'. For example, if we say `lit('a')` then that describes the language consisting of just the character string "a" and nothing else.

We have the API call `seq(x, y)`. We could say `seq(lit('a'), lit('b'))`, and that would consist of just the string "ab." So far not very interesting.

Then we could say `alt(x, y)`. Similarly, `alt(lit('a'), lit('b'))`, and that would consist of two possibilities -- either the string "a" or the string "b." We'll use the standard notation for the name of our API call here.

`star(x)` stands for any number of repetitions -- zero or more. `star(lit('a'))` would be the empty string or "a" or "aa" and so on.

We can say `oneof(c)` and then string of possible characters. That's that same as the alternative of all the individual characters. `oneof('abc')` matches "a" or "b" or "c." It's a constrained version of the alt function.

We'll use the symbol "eol," standing for "end of line" to match only the end of a character string and nowhere else. What matches is the empty string, but it matches only at the end. The only example we can give is "eol" itself, and we can give an example of `seq(lit('a'), eol)`, and that matches exactly the "a" and nothing else at the end.

Then we'll add `dot`, which matches any possible character -- `a`, `b`, `c`, any other character in the alphabet.

In the chart below, s is a string (representing something that must be matched exactly), c is also a string (but represents a set of characters; we match any one of them), and x and y are patterns (that is, regular expressions) formed by any of the elements in the chart.

API		
Grammar	Example	Language
`lit(s)`	`lit('a')`	`{a}`
`seq(x,y)`	`seq(lit('a'), lit('b'))`	`{ab}`
`alt(x,y)`	`alt(lit('a'),lit('b'))`	`{a,b}`
`star(x)`	`star(lit('a'))`	`{'',a,aa,aaa,...}`
`oneof(c)`	`oneof('abc')`	`{a,b,c}`
`eol`	`eol`	`{''}`
	`seq(lit('a'), eol)`	`{a}`
`dot`	`dot`	`{a,b,c,....}`

1.4 4. 4 Specifications

Now, of course Python has a regular expression module called the "re module". If you really want to use regular expressions, you import re and get started.

The point here is not so much to use regular expressions as to show you what it's like to build a language processer. So to get started doing that, I started by writing a test function that defines some regular expressions using this API.

```python
def test_search():
    a, b, c = lit('a'), lit('b'), lit('c')
    abcstars = seq(star(a), seq(star(b), star(c)))
    dotstar = star(dot)
    assert search(lit('def'), 'abcdefg') == 'def'
    assert search(seq(lit('def'), eol), 'abcdef') == 'def'
    assert search(seq(lit('def'), eol), 'abcdefg') == None
    assert search(a, 'not the start') == 'a'
    assert match(a, 'not the start') == None
    assert match(abcstars, 'aaabbbccccccccdef') == 'aaabbbcccccccc'
    assert match(abcstars, 'junk') == ''
    assert all(match(seq(abcstars, eol), s) == s
                    for s in 'abc aaabbccc aaaabcccc'.split())
    assert all(match(seq(abcstars, eol), s) == None
                    for s in 'cab aaabbcccd aaaa-b-cccc'.split())
    r = seq(lit('ab'), seq(dotstar, seq(lit('aca'), seq(dotstar, seq(a, eol)))))
    assert all(search(r, s) is not None
                for s in 'abracadabra abacaa about-acacia-flora'.split())
    assert all(match(seq(c, seq(dotstar, b)), s) is not None
                for s in 'cab cob carob cb carbuncle'.split())
    assert not any(match(seq(c, seq(dot, b)), s)
                for s in 'crab cb across scab'.split())
    return 'test_search passes'

```

First I defined a, b, and c as being these literals. Then I combine some more complicated examples. Then I showed some example of assertions, of what we want to be true.

We've defined two functions that match the functions in the re module. The first function is called `search`. It takes a pattern and a text, and in the regular expression module this function returns something called a "match object."

What we'll have it do is return a string which will be the earliest match in the text for the pattern, and if there is more than one match at the same location, it'll be the longest of those.

For example, `search('def', 'abcdef')` would return `'def'` because it's found there.

The next function is called `match`. It also takes a pattern and a text and returns the string that matches. But for match, the match must occur at the very start of the string.

`match('def', 'abcdef')` would return `None`, indicating that there is no match. But `match('def', 'def fn(x):')` in this string that has `'def'` at the beginning would return that match.

Here are some examples of the types of things that I want to be able to do. That gives me the start of a specification for how I want to write my search and match functions.

1.5 5. 5 Concept Inventory

I always like to start out with an inventory of the concepts that are going to be used.

So far we have the notion of a pattern or a regular expression, of a text to match against, the result, which will also be a string of some kind. It doesn't seem like there's all that many more concepts. One thing that it looks like we'll need is some notion of a partial result, and some notion of control over the iteration.

- pattern
- text -- result
- partial result
- control over iteration

What do I mean by that?

Well, some of the matches are easy. If we search for a literal 'def' within 'abcdef,' it's easy to imagine just going down the line of this string and saying does it match here? No. Here? No. Here? No. Here? Yes. We'll return that result.

But what if we're matching with the pattern -- let's say we have the expression `'a*b+'` -- any number of a's followed by one or more b's?

In our API notation, we would write that as `seq(star(lit('a')), plus(lit('b')))`.

Let's say we're matching that against the string `'aaab'`.

Now if we had a control structure that says sequence, look to match the first, and then look at the second, and if the first -- `star(lit('a'))` -- only had one possible result, then it would say, yes, it matches here right at the start, now look for something after that.

Does it match `plus(lit('b'))`?

No, it doesn't.

I've got to have iteration over the possible results.

I have to say that `star(lit('a'))` can match in more than one location.

It can match with zero instances of a, with 1, with 2, with 3, and it's only after 3 thatthen we can match the second part, find the b, and then find that the whole expression matches.

That's going to be the tricky part -- worrying about this control when one part of a sequence doesn't match or similarly when we have an alternative between two possibilities -- `a` or `b` or `alt(a, b)`.

This trickiness seems like it's going to be difficult, but it all resolves itself after wemake one good choice.

It takes some experience to see what that choice can be, but if we decide to represent these partial results as a set of remainders of the text, then everything falls into place.

What do I mean by remainder?

I mean everything else after the match.

If we match a literal `a` the remainder after we match zero characters of this string would be `aaab` -- three `a`'s followed by a `b`.

The remainder after we match one a would be two `a`'s followed by a `b` and so on.

What I'm going to do is define an auxiliary function called `match_set`, and it takes a pattern and a text, and it returns this set of remainders.

When given just this pattern here as the input, `star(lit('a'))`, and this text as the `text` then the remainder would be the set consisting of three `a`'s and a `b`, two `a`'s and a `b`, one `a` and a `b`, or just `b`.

In other words, `star(lit('a'))` could have consumed 0, 1, 2, or 3 a's, and that's the remainder that's left over.

So the result will be this set: `{aaab, aab, ab, b}`.

1.6 6. 6 Matchset

Here's the code:

```
def matchset(pattern, text):
    "Match pattern at start of text; return a set of remainders of text."
    op, x, y = components(pattern)
    if 'lit' == op:
        return set([text[len(x):]]) if text.startswith(x) else null
    elif 'seq' == op:
        return set(t2 for t1 in matchset(x, text) for t2 in matchset(y, t1))
    elif 'alt' == op:
        return matchset(x, text) | matchset(y, text)
    elif 'dot' == op:
        return # Enter code here
    elif 'oneof' == op:
        return # Enter code here
    elif 'eol' == op:
        return set(['']) if text == '' else null
    elif 'star' == op:
        return (set([text]) |
                set(t2 for t1 in matchset(x, text)
                    for t2 in matchset(pattern, t1) if t1 != text))
    else:
        raise ValueError('unknown pattern: %s' % pattern)

```

It's just 20 lines or so.

I've left out a couple pieces to give you something to do, but let me first try to explain how it works.

We take a pattern and a text. We're breaking down the pattern into its components. The components are an operator and then and x and a y part, depending on the definition.

For example the literal will only have an x component. Sequence and alternative will have both and x and a y. We'll see how that component is written later, but for now just assume it's going to pick out the right pieces. Then we decide which operator we have, literal, sequence, alt, and so on, and return a proper set accordingly.

For example, if the operator is asking for literal string x, we ask does the text start with x? If it does, then the remainder will be a singleton set, a set of just one element, which is the remainder of the text after we've broken off the length of text characters.

If we matched a three-character sequence for x, we return the text without the first three characters.

If x was 1 character, we return the text without the first character. Otherwise, we return the null set. There's no match.

The alternative is particularly nice. It says if we have an alternative between x and y we just return the union of the those two match sets. This character, the or bar, means "union" in Python, the union of two sets.

Sequence is a complicated one. It says if we're asked for a sequence of x and y, we first find a matching set for x. That's going to be a set of possible remainders, and we go through all of those. Then take the remainder of that text, which would be t1. Then match y against that remainder. For each of those alternatives, that's what we're going to return in this set.

For example, if this is our pattern -- `seq(star(lit('a')) plus(lit('b')))` -- it looks like that, and we're matching against `"aaab"` then the x is the `star(lit('a'))`, and `y` is the `plus(lit('b'))`, and the matchset for `x` is this set here (`{aaab, aab, ab, b}`).

We try to match `y`, the `plus(lit('b'))`, against all of these match sets, and it'll fail to match against each of these three (`aaab, aab, ab`). It will match against this one (`b`), and so now we have a match that consumes the entire string. The result from the match of this sequence of x and y will be the set consisting of just the empty string, because we've matched off all the a's and one b, and there's no remainder left over.

Note that there's a big difference between the outcome of saying here's a result consisting of one string, the empty string, versus the set that is the empty set. The empty set is a failed match, and the set consisting of the empty string is the successful match.

Now let's see if you can fill in your code for these two missing cases here.

Remember, you're going to be returning a set of possible remainders if the match is successful.

1.7 7. 6 Matchset (Answer)

Here's my answer. A dot matches any character in the text. If the text has at least one character, then there's going to be a match, and the match is going to have a remainder, which is all the rest of the text.

That remainder is a set, and it's the set of one element, and the element is the text from the first character on. In other words, dropping off the 0th character. We're going to return that if there is any text at all. That is, if the text is not empty. Otherwise, if the text is empty then we're going to return the null set. We defined the null set down here.

How about for oneof? Oneof takes a string of possible characters and what it should return is similar to dot. It should return the remaining characters if the first character is equal to one of the characters in x. We're going to return exactly the same thing, a set consisting of a signal answer which is the original text with the first character dropped off, and we're going to return that. What I'd like to be able to say if the text starts with any of the characters in x. It turns out that I can actually say exactly that -- `text.startswith(x)`, if I arrange to have x be a *tuple* of characters rather than a character string.

Here I have the documentation from the Python manual for the string that starts with method, and it says it's true if the string starts with a prefix, so we can ask does some string start with a particular string -- yes or no? But prefix can also be a *tuple* of prefixes to look for. All we have to do is arrange for that x to be a tuple, and then it automatically searches for all of them.

What I'm saying is I want the API function oneof('abc'). That should return some object, but we're not yet specifying what that object is, such that the operator of that object is `oneof`, and the `x` for that object should be the tuple `('a', 'b', 'c')`. It's sort of a little optimization here that I've defined the API to construct an object of just the right form so that I can use it efficiently here. Just go ahead and check does the text start with any one of the possible x's. Otherwise, return no. If you didn't know about this form of `text.startswith`, you could have just checked to see if any of the character `c` for `c` in `x`. We'd say return the rest of the string if any of the characters in `x` if the text starts with any one of those characters. Otherwise, return "no."

1.8 8. 7 Filling out the API

This looks like a good time to define the rest of the APIs for the constructors for patterns. Here I've defined literal and sequence to say they're going to return tuples where the first element is the operator, the second -- if it exists -- is x, and then the final -- if it exists -- is the y. Those are for the components that take an operand or two, but `dot` and end of line don't, so I just defined dot to be a one element tuple. See if you can go ahead and fill in the remainder of this API.

1.9 9. 7 Filling out the API (Answer)

Here are the definitions I came up with. Note that I decided here to define `plus` and `opt` in terms of things that we had already defined -- `sequence` and `alt` and `star` and `lit`. You could've just defined them to be similar to the other ones if you prefer that representation. It's just a question of whether you want more work to go on here in the constructors (`plus` and `opt`) for the patterns or here in the definition of `match_set`.

1.10 10. 8 Search and Match

Let's just review what we've defined in terms of our API. We have a function match and a function search, and they both take a pattern and a text, and they both return a string representing the earliest longest match.

```
match(p, t) -> '...'¶
search(p, t) -> '...'¶
```

But for match the string would only return if it's at the start of the string. For search, it'll be anywhere within the string. If they don't match, then they return `None`.

We've also defined an API of functions to create patterns -- a literal string, an alternative between two patterns x and y, a sequence of two patterns x and y, and so on. That's sort of the API that we expect the programmer to program to. You create a pattern on this side and then you use a pattern over here against a text to get some result. Then below the line of the API -- sort of an internal function -- we've defined `matchset`, which is not really designed for the programmer to call it. It was designed for the programmer to go through this interface (`match` and `search`), but this function is there. It also takes a pattern and a text. Instead of returning a single string, which is a match, it returns a set of strings, which are remainders. For any remainder we have the constraint that the match plus the remainder equals the original text. Here I've written versions of search and match. We already wrote matchset. The one part that we missed out was this component that pulls out the x, y and op components out of a tuple. I've left out two pieces of code here that I want you to fill in.

1.11 11. 8 Search and Match (Answer)

Let's do match first. Match interfaces with the matchset, finds the set of remainders. If there is a set of remainders, then it finds the shortest one.

The shortest remainder should be the longest text, and then we want to return that.

We want to return the text, and it's a match not a search, so the match has to be at the beginning of the text, and it would go up to match. So we match from the beginning of the text. How far do we want to go? Well, everything except the remainder.

How much is that? Well, we can just subtract from the length of the text the length of the shortest, and that gives us the initial piece of the longest possible match.

Here search calls into match. What we do is we start iterating. We start at position number zero. If there is a match there for the text starting at position zero, then we want to return it. If not, we increment `i` to 1, and we say does the text from position 1 on -- is there a match there and so on. We just keep on going through until we find one.

Here what we want to say is `if the match is not None, then return m`. Notice that it would be a bad idea to say `if m return m`. Normally it's idiomatic in Python to say that if we're looking for a true value. But the problem here is that the empty string we want to count as a true value. A pattern might match the empty string. That counts as a match, not as a failure. We can't just say `if m` because the empty string counts as false. We have to say `if m is not None`.

1.12 12. 9 Compiling

Let's quickly summarize how a language interpreter works.

For regular expressions we have patterns like `(a|b)+`, which define languages. A language is a set of strings like `{a, b, ab, ba, ...}` and so on, defined by that pattern.

Then we have interpreters like `matchset`, which in this case takes a pattern and a text and returns a list of strings or a set of strings.

So we saw that matchset is an interpreter because it takes a description of the language, namely a pattern as a data structure, and operates over that pattern. Here's the definition of `matchset`.

You see it looks at the pattern, breaks out its components, and then the first thing it does is this big case statement to figure out what type of operator we have and to do the appropriate thing.

There's an inherent inefficiency here in that the pattern is defined once, and it's always the same pattern over a long string of text and maybe over many possible texts.

We want to apply the same pattern to many texts. Yet every time we get to that pattern, we're doing this same process of trying to figure out what operator we have when, in fact, we should already know that, because the pattern static, is constant.

So this is kind of repeated work. We're doing this over and over again for no good reason.

There's another kind of interpreter called a "compiler" which does that work all at once. The very first time when the pattern is defined we do the work of figuring out which parts of the pattern are which so we don't have to repeat that every time we apply the pattern to a text.

Where an interpreter takes a pattern and a text and operates on those, a compiler has two steps. In the first step, there is a compilation function, which takes just the pattern and returns a compiled object, which we'll call `c`. Then there's the execution of that compiled object where we take `c` and we apply that to the text to get the result.

Here work can be done repeatedly every time we have a text.

Here the work is split up. Some of it is done in the compilation stage to get this compiled object. Then the rest of it is done every time we get a new text. Let's see how that works.

Here is the definition of the interpreter. Let's focus just on this line here:

```
def matchset(pattern, text):
    "Match pattern at start of text; return a set of remainders of text."
    op, x, y = components(pattern)
    if 'lit' == op:
        return set([text[len(x):]]) if text.startswith(x) else null
```

This says if the op is a literal, then we return this result.

The way I'm going to change this interpreter into a compiler is I'm going to take the individual statements like this that were in the interpreter, and I'm going to throw them into various parts of a compiler, and each of those parts is going to live in the constructor for the individual type of pattern.

We have a constructor -- literal takes a string as input and let's return a tuple that just represents what we have, and then the interpreter deals with that.

Now, we're going to have literal act as a compiler. What it's going to do is return a function that's going to do the work.

```
def lit(s): return lambda text: set([text[len(s):]]) if text.startswith(s) else null
```

What is this saying?

We have the exact same expression here as we had before, but what we're saying is that as soon as we construct a literal rather than having that return a tuple, what it's returning is *a function from the text to the result that matchset would have given us.*

1.13 13. 10 Lower Level Compilers

We can define a pattern -- let's say pattern is `lit('a')`.

```
>>> pat = lit('a')
```

Now what is a pattern? Well, it's a function. It's no longer a tuple.

```
>>> pat
<function <lambda> at 0x101b7bd70>
```

We can apply that pattern to a string and we get back the set of the remainders.

```
>>> pat('a string')
set([' string'])
```

It says, yes, we were able to successfully parse `a` off of a string, and the remainder is a string.

We can define another pattern. Let's say pattern 2 equals plus(pat).

```
>>> pat2 = plus(pat)¶
>>> pat2¶
<function <lambda> at 0x101b7bcf8>¶
¶
```

Pattern 2 is also a function, and we can call pattern 2 of let's say the string of five a's followed by a b.

```
>>> pat2('aaaaab')¶
set(['b', 'ab', 'aaab', 'aaaab', 'aab'])¶
¶
```

- Now we get back this set that says we can break off any number of a's because we're asking for a and the plus of that and the closure of a. These are the possible remainders if we break off all of the a's or all but one or all but three, and so on. Essentially we're doing the same computation that in the previous incarnation with an interpreter we would have done with:

```
>>> matchset(pat2, 'aaaab')¶
¶
```

Now we don't have to do that. Now we're calling the pattern directly. So we don't have matchset, which has to look at the pattern and figure out, yes, the top-level pattern is a plus and the embedded pattern is a lit. Instead the pattern is now a composition of functions, and each function does directly what it wants to do. It doesn't have to look up what it should do.

In interpreter we have a way of writing patterns that describes the language that the patterns below to. In a compiler there are two sets of descriptions to deal with. There's a description for what the patterns look like, and then there's a description for what the compiled code looks like.

Now, in our case -- the compiler we just built -- the compile code consists of Python functions. They're good target representations because they're so flexible. You can combine them in lots of different ways. You can call each other and so on. That's the best unit that we have in Python for building up compiled code.

There are other possibilities. Compilers for languages like C generate code that's the actual machine instructions for the computer that you're running on, but that's a pretty complicated process to describe a compiler that can go all the way down to machine instructions. It's much easier to target Python functions.

Now there's an intermediate level where we target a virtual machine, which has its own set of instructions, which are portable across different computers. Java uses that, and in fact Python also uses the virtual machine approach, although it's a little bit more complicated to deal with. But it is a possibility, and we won't cover it in this class, but I want you to be aware of the possibility.

Here is what the so-called byte code from the Python virtual machine looks like. I've loaded the module dis for disassemble and dis.dis takes a function as input and tells me what all the instructions are in that function.

Here's a function that takes the square root of x-squared plus y-squared.

```
>>> import dis¶
>>> import math¶
>>> sqrt = math.sqrt¶
>>> dis.dis(lambda x, y: sqrt(X ** 2 + y ** 2))¶
·· 1·········· 0 LOAD_GLOBAL············· 0 (sqrt)¶
·············· 3 LOAD_GLOBAL············· 0 (x)¶
·············· 6 LOAD_CONST·············· 1 (2)¶
·············· 9 BINARY_POWER¶
············· 10 LOAD_FAST··············· 1 (y)¶
············· 13 LOAD_CONST·············· 1 (2)¶
············· 16 BINARY_POWER¶
············· 17 BINARY_ADD¶
············· 18 CALL_FUNCTION··········· 1¶
············· 21 RETURN_VALUE¶
¶
```

This is how Python executes that. It loads the square root function. It loads the x and the 2, and then does a binary power, loads the y and the 2, does a binary power, adds the first two things off the top of the stack, and then calls the function, which is the square root function with that value, and then returns it.

This is a possible target language, but much more complicated to deal with this type of code than to deal with composition of functions.

1.14 14. 11 Alt

Let's get back to our compiler.

```
def matchset(pattern, text)
    ...
    elif 'seq' == op:
        return set(t2 for t1 in matchset(x, text) for t2 in matchset(y, t1))
```

Again, in `matchset` I pulled out one more clause. This is a clause for sequence, and this is what we return. If I want to write the compiler for that sequence clause, I would say let's define seq(x, y).

```
def seq(x, y): return lambda text: set().union(*map(y, x(text))
```

It's a compiler so it's going to return a function that operates on x and y, take as input a text and then returns as result. We could take exactly that result. While I'm moving everything to this more functional notation, I decided let's just show you a different way to do this. This way to do it would be fine, but I could have the function return that. But instead, let's have it say what we're really trying to do is form a union of sets. What are the sets? The sets that we're going to apply union to. First we apply x to the text, and that's going to give us a set of remainders. For each of the remainders, we want to apply y to it. What we're saying is we're going to map y to each set of remainders. Then we want to union all those together. Now, union, it turns out, doesn't take a collection. It takes arguments with union a, b, c. So we want to turn this collection into a list of arguments to union. We do that using this apply notation of saying let's just put a star in there. Now, we've got out compiler for sequence. It's the function from text to the set that results from finding all the remainders for x and then finding all the remainders from each of those after we apply y. Unioning all those together in union will eliminate duplicates.

Now it's your turn to do one. This was the definition of alt in the interpreter matchset. Now I want you to write the definition of the compiler for alt, take a pattern for (x, y), and return the function that implements that.

1.15 15. 11 Alt (Answer)

Here is the answer.

```
def matchset(pattern, text):
    op, x, y = components(pattern)
    if 'lit' == op:
        return set([text[len(x):]]) if text.startswith(x) else null
    elif 'seq' == op:
        return set(t2 for t1 in matchset(x, text) for t2 in matchset(y, t1))
    elif 'alt' == op:
        return matchset(x, text) | matchset(y, text)

...
def lit(s): return lambda t: set([t[len(s):]]) if t.startswith(s) else null
def seq(x, y): return lambda t: set().union(*map(y, x(t)))
def alt(x, y): return lambda t: x(t) | y(t)
```

The structure is exactly the same. It's the union of these two sets. The difference is that with a compiler the calling convention is pattern gets called with the text as argument. In the interpreter the calling convention is matchset calls with the pattern and the text.

1.16 16. 12 Simple Compilers

Here's the whole program.

```
def match(pattern, text):
    "Match pattern against start of text; return longest match found or None."
    remainders = pattern(text)
    if remainders:
        shortest = min(remainders, key = len)
        return text[:len(text)-len(shortest)]

def lit(s): return lambda t: set([t[len(s):]]) if t.startswith(s) else null
def seq(x, y): return lambda t: set().union(*map(y, x(t)))
def alt(x, y): return lambda t: x(t) | y(t)
def oneof(chars): return lambda t: set([t[1:]]) if (t and t[0] in chars) else null
dot = lambda t: set([t[1:]]) if t else null
eol = lambda t: set(['']) if t == '' else null
def star(x): return lambda t: (set([t]) |
                               set(t2 for t1 in x(t) if t1 != t
                                   for t2 in star(x)(t1)))
```

Now, compilers have a reputation as being difficult and more complicated than interpreters, but notice here that the compilers is actually in many ways simpler than the interpreter.

It's fewer lines of code over all. One reason is because we didn't have to duplicate effort here of saying first we need constructors to build up a literal and then within matchset have an interpreter for that literal. Rather we did it just once. Just once! We said the constructor for literal returns a function which is going to be the implementation of the compiler for that type of pattern. It's very concise. Most of these are one-liners. Maybe I cheated a little bit and I replaced the word "text" with the word "t" to make it a little bit shorter and fit on one line.

There's only one that's complicated. That's the star of x, because it's recursive. The ones I haven't listed here is because they're all the same as before. Before we get into star(x) let me note that.

I didn't have to put down search here, because search is exactly the same as before.

I didn't have to put down plus, because plus is exactly the same as before. It's defined in terms of star.

What is the definition of star? One thing we could return is the remainder could be the text itself. Star of something -- you could choose not to take any of it and return the entire text as the remainder. That's one possibility. The other possibility is we could apply the pattern x. From `star(x)` apply the pattern x to the text and take those sets as remainders. For every remainder that's not the text itself -- because we already took care of that. We don't need to take care of it again. For all the remainders that are different from the whole text then we go through and we apply star(x) to that remainder. We get a new remainder and that's the result. That's all we need for the compiler result.

Oh, one piece that was missing is how do interface the match function, which takes a pattern and a text, with this compiler where a pattern is applied to the text. That's one line, which is slightly different. Here before we called matchset. In the previous implementation we had

```
def match(pattern, text):
    remainders = matchset(pattern, text)
    ...
```

Your job then is to replace that with the proper code for the implementation that calls the compiler.

1.17 17. 12 Simple Compilers (Answer)

The answer is that the interface with the compiler is we just call a pattern with text as the input. That's all we need to do.

```
def match(pattern, text):¶
    remainders = pattern(text)¶
    ...¶
¶
```

1.18 18. 13 Recognizers and Generators

So far what we've done is call the recognizer task. We have a function match which takes a pattern and a text, and that returns back a substring of text if it matches or None.

It's called a recognizer, because we're recognizing whether the prefix of text is in the language defined by the pattern.

There's a whole other task called the generator in which we generate from a pattern a complete language defined by that pattern.

For example, the pattern a or b sequenced with a or b -- `(a|b)(a|b)`. That defines a language of four different strings -- `{aa, ab, ba, bb}`, and we could define a function that takes a pattern and generates out that language. That all seems fine.

One problem, though. If we have a language like a* then the answer of that should be the empty string or a or aa or aaa and so on -- `{'', a, aa, aaa, ...}`. It's an infinite set. That's a problem. How are we going to represent this infinite set?

Now, it's possible, we could have a generator function that generates the items one at a time. That's a pretty good interface, but instead I'm going to have one where we limit the sizes of the strings we want. If we say we want all strings up to n characters in length, then that's always going to be a finite set.

I'm going to take the compiler approach. Rather than write a function "generate," I'm going to have the generator be compiled into the patterns. What we're going to write is a pattern, which is a compiled function, and we're going to apply that to a set of integers representing the possible range of lengths that we want to retrieve. That's going to return a set of strings.

```
pat({int}) --> {str}
```

So for example, if we define pattern to be `a*` -- we did that appropriately -- and then we asked for pattern, and we gave it the set `{1, 2, 3}`,

```
pat = a*¶
pat({1,2,3}) --> {a, aa, aaa}¶
```

then that should return all strings which are derived from the pattern that have a length 1, 2, or 3. So that should be the set `{a, aa, aaa}`. Now let's go ahead and implement this.

1.19 19. 14 Oneof and Alt

Okay. Here's the code for the compiler.

```
def lit(s):         return lambda Ns: set([s]) if len(s) in Ns else null¶
def alt(x, y):      return lambda Ns: # your code here¶
def star(x):        return lambda Ns: opt(plus(x))(Ns)¶
def plus(x):        return lambda Ns: genseq(x, star(x), Ns, startx=1) #Tricky¶
def oneof(chars):   return lambda Ns: # your code here¶
def seq(x, y):····· return lambda Ns: genseq(x, y, Ns)¶
def opt(x):········ return alt(epsilon, x)¶
dot = oneof('?')··· # You could expand the alphabet to more chars.¶
epsilon = lit('')·· # The pattern that matches the empty string.¶
¶
null = frozenset([])¶
¶
def test():¶
¶
···· f = lit('hello')¶
···· assert f(set([1, 2, 3, 4, 5])) == set(['hello'])¶
···· assert f(set([1, 2, 3, 4]))··· == null¶
¶
···· g = alt(lit('hi'), lit('bye'))¶
···· assert g(set([1, 2, 3, 4, 5, 6])) == set(['bye', 'hi'])¶
···· assert g(set([1, 3, 5])) == set(['bye'])¶
¶
···· h = oneof('theseletters')¶
···· assert h(set([1, 2, 3])) == set(['t', 'h', 'e', 's', 'l', 'r'])¶
···· assert h(set([2, 3, 4])) == null¶
¶
···· return 'tests pass'¶
¶
```

Now, remember the way the compiler works is the constructor for each of the patterns takes some arguments -- a string, and x and y pattern, or whatever -- and it's going to return a function that matches the protocol that we've defined for the compiler.

The protocol is that each pattern function will take a set of numbers where the set of numbers is a list of possible lengths that we're looking for. Then it will return a set of strings.

What have I done for `lit(s)`? I've said we return the function which takes a set of numbers as input, and if the length of the string is in that set of number -- if the literal string was "`hello`" and if `hello` has five letters and if 5 is one of the numbers we're trying to look for -- then return the set consisting of a single element -- the string itself. Otherwise, return the null set.

`star` I can define in terms of other things.

`plus` I've defined in terms of a function sequence that we'll get to in a minute. It's a little bit complicated. It's really the only complicated one here. We can reduce all the other complications down to calling plus, which calls `genseq()`. `seq` does that too.

I've introduced epsilon, which is the standard name in language theory for the empty string. So it's the empty string. It's the same as just the literal of the empty string, which matches just itself if we're looking for strings of length 0.

For dot -- dot matches any character. I've decided to just return a question mark to indicate that. You could return all 256 characters or whatever you want. Your results would start to get bigger and bigger. You can change that if you want to.

I left space for you to do some work.

Give me the definitions for `oneof(chars)`.

If we ask for `oneof('abc')` , what should that match?

What it should match is if 1 is an element of Ns then it should be abc. Otherwise, it shouldn't be anything.

Similarly for `alt`. Give me the code for that.

1.20 20. 14 Oneof and Alt (Answer)

```
def lit(s):········ return lambda Ns: set([s]) if len(s) in Ns else
null¶
```

```
def alt(x, y):····· return lambda Ns: x(Ns) | y(Ns)¶
def star(x):······· return lambda Ns: opt(plus(x))(Ns)¶
def plus(x):······· return lambda Ns: genseq(x, star(x), Ns, startx =
1) #Tricky¶
def oneof(chars):·· return lambda Ns: set(chars) if 1 in Ns else null¶
def seq(x, y):····· return lambda Ns: genseq(x, y, Ns)¶
def opt(x):········ return alt(epsilon, x)¶
dot = oneof('?^')··· # You could expand the alphabet to more chars.¶
epsilon = lit('')·· # The pattern that matches the empty string.¶
¶
```

The answer is if we want an alternative of the patterns x and y, then we use a protocol. We say let's apply x to the set of numbers. That gives us a set of strings that matches. We'll just union that with the set that comes back from applying y to the set of numbers. Now, for one of char -- this will be a list of possible characters that we're trying to match one of. If 1 is in our list of numbers, then we should return all of those, and we have to return them as a set. We'll say the set of character if 1 is in there. Otherwise, there are no matches at all, so return null.

1.21 21. 15 Avoiding Repetition

That's the whole compiler. I want to show you just a little bit of the possibility of doing some compiler optimizations. Notice this sort of barrier here where we introduce lambda, where we introduce a function. Remember I said that there's two parts to a compiler. There's the part where we're first defining a language. When we call lit and give it a string, then we're doing some work to build up this function that's going to do the work every time we call it again. Anything that's on the right of the lambda is stuff that gets done every time. Anything that's to the left is stuff that gets done only once.

Notice that there is a part here building up this set of `s` that I'm doing every time, but that's wasteful because s doesn't depend on the input. `s` is always going to be the same.

I can pull this out and do it at compile time rather than do it every time we call the resulting function.

I'll make this set of s and I'll give that a name -- `set_s`. Over here I'll do `set_s` equals that value. It looks like I'd better break this up into multiple lines.

Now I pulled out that precomputation so it only gets done once rather than gets done every time. You could look around for other places to do that.

I could pull out the computation of this set of characters and do that only once as well.

That's a **lifting operation** that stops us from repeating over and over again what we only need to do once. That's one of the advantages of having a compiler in the loop. There is a place to do something once rather than to have to repeat it every time.

1.22 22. 16 Genseq

Now there's only one bit left -- this generate sequence. Let's talk about that. Now sequence in this formulation is a function that takes x and y, two patterns, and what it returns is a function, and that function takes a list of numbers and returns a set of text that match. So sequence is delaying the calculation. It's computing a function which can do the calculation later on. Genseq does the calculation immediately. It takes x and y and a set of numbers, and it immediately calculates the set of possible text. Now the question is what do we know about genseq in terms of the patterns x and y and the set of possible numbers. We know at some point we're going to have to call the pattern x with some set of numbers. We're not yet quite sure what. That's going to return a list of possible text. Then we're going to have to call y with some other set of numbers. Then we're going to have to concatenate them together and see if they make sense, if the concatenation of some x and some y, if that length is within this allowable set. Now, what do we know about what these Ns should be in terms of this set of possible numbers here regardless of what this set is. This could be a dense set, so we could have Ns equals 0, 1, 2, all the way up to 10 or something. Or it could be a sparse set. It could be, say, only the number 10. But either way, the restriction on x and y is such that they have to add up to no more than 10. But x could be anything. If the list of possible numbers that we want to add up to is only 10, that doesn't constrain x at all other than to be less than 10. This N should be everything up to the maximum of N sub s. Then what should y be? Well, we have two choices. One, we could for each x that comes back we could generate the y's. Or we could generate the y's all at once and then try to combine them together and see if they match up. I think that's actually easier. So for the y's also, they can be any size up to the maximum. Then we take the two together, add up the x match and the y match and see if that length is within N. In this example, if Ns is equal to 10, here we want to have the Ns be everything from 0 up to 10 inclusive in both cases, and we get back some results like, say, a, abb, acde, and so on, and some other result over here -- ab, bcd.

Then for each of them we add them up, and if we say abb plus ab and check to see if that's in Ns. If it is, we keep it. If it's not, we don't keep it. Here is candidate solution for genseq. We take x, y, and a set of numbers, and then we define Ns as being everything up to the largest number, including the largest number. We have to add 1 to the maximum number in order to get a range going from 0 up to and including the largest number. Now that we know the possible values of the numbers that we're looking for for the sizes of the two components-the x and the y components -- then we can say m1 is all the possible matches for x, m2 is all the possible matches for y. If the length of m1 plus m2 is in the original set of numbers that we're looking for, then return m1 plus m2. This seems reasonable. It looks like it's doing about what we're looking for to generate all sequences of x and y concatenated together. But I want you to think about it and say, have we really gotten this right? The choices are is this function correct for all inputs? Or is in incorrect for some? Does it return incorrect results? Or is it correct when it returns, but doesn't doesn't always return? Think about that. Think about is there any result that looks like it's incorrect that's being formed. Think about does it infinite loop or not. Think about base cases on recursion and saying is there any case where it looks like it might not return. This is a tricky question, so I want you to try it, but it may be difficult to get this one right.

1.23 23. 16 Genseq (Answer)

The answer is that it is correct when it returns.

All the values it builds up are correct, but unfortunately it doesn't always return. Let's try to figure out why.

In thinking about this, we want to think about recursive patterns.

Let's look at the pattern x+. We've defined x+ as being the sequence of x followed by x*. And now for most instances of x that's not a problem.

If we had plus(lit('a')), it not going to be a problem. That's going to generate a, aa, aaa, and so on.

But consider this -- let's define a equals lit('a'), pat equals plus(opt('a')).

Now, this should be the same. This should also generate a, aa, aaa.

The way we can see that is we have a plus so that generates any number of these. If we pick a once, we get this. It we pick a twice we get this. If we pick a three times we get this. But the problem is there's all these other choices in between.

`opt(a)` means we we can either be picking a or the empty string. As we go through the loop for plus, we could pick empty string, empty string, empty string. We could pick empty string an infinite number of times. Even though our N is finite -- at some point we're going to ask for pattern of some N -- let's say the set {1, 2, 3} -- we won't have a problem with having an infinite number of a's, but we will have a problem of choosing from the opt(a) the empty part. If an infinite number of times we choose the empty string rather than choosing a, then we're never going to get past three as the highest value. We're going to keep going forever. That's the problem. We've got to somehow say I don't want to keep choosing the empty string. I want to make progress and choose something each time through. So how can we make sure that happens?

1.24 24. 17 Induction

What I have to do is I have to check all my cases where I have recursion and eliminate any possibility for infinite recursion. Now, there are two possibilities in the star function and the plus function. Those are the two cases where regular expressions have recursion. But now star I defined in terms of plus, so all that's left is to fix plus to not have an infinite recursion. Here's how I define plus. Basically, I said that x+ is defined as x followed by x*, and the x* is in turn defined in terms of x+. The problem was that I was going through and saying, okay, for x when I'm doing plus(opt(a)), for my x I want to choose opt(a). Okay, I think I'll choose the empty string. So I chose that, and now I'm left in a recursion where I have an x*, which is defined in terms of x+, and I haven't made any progress. I have a recursive call that's defined in terms of itself. We know in order to make sure that a recursion terminates, we have to have some induction where we're reducing something. It makes sense here that what we're going to reduce is our set Ns. One way to guarantee to do that is to say when I'm generating the x followed by the x* let's make sure that the x generates at least 1 character. If we can guarantee that x generates a character, then when we go to do the x* we've reduced our input. So we have this inductive property saying that now our set of Ns will be smaller. It's smaller by 1 each time, and if it started out finite and we reduce by 1 every time, then eventually we're going to terminate. Let's see how we can implement that idea of saying every time we have an x+ we have a x and an x*, we have to choose at least 1 character for x. Note that that's not limiting us in any way. That hasn't stopped us from generating all possible expressions, because if we were going to generate something, it would have to come from somewhere -- either from this x or this x -- and we might as well make sure it comes from here rather than adding an infinite number of nothings before we generate something.

1.25 25. 18 Testing genseq

Here's what gensequence looks like. We have a recursive base case that says, if there are no numbers that we're looking for, we can't generate anything of those lengths, and so return the empty set. Then we say the xmatches we get by applying x to any number up to the maximum of Ns, including the maximum of Ns, but then we got to do some computation to figure out what can be the allowable sizes for y, and we do that by saying, let's take all the possible values that came back from the xmatches and then for each of those values and for each of the original values for the lengths that we're looking for, subtract those off and say, total is going to be one of the things we got from x and one of the things we got from y, that better add up to one of the things in Ns. Then we call y with that set of possible ends for y and then we do the same thing that we were going to do before. We go through those matches, but this is going to be with a reduced set of possibilities and count those up, and now, the thing that makes it all work is this optional argument here, saying the number that we're going to start at for the possible sizes, for x in the default case, that's 0, and so we start the range at 0. But in the case where we're calling from +, we're going to set that to 1. Let's see what that looks like. Here's the constructors, the compilers for sequence and plus. For a regular sequence, there is no constraint on this start for x. X can be any size up to the maximum of the N's. But for plus, we're going to always ask that the x part have a length of at least 1, and then the y part will be whatever is left over. That's how we break the recursion, and we make sure that genseq will always terminate. Now this language generation program is a little bit complex. So I wanted to make sure that I wrote a test suite for it to test the generation. So here I've just defined some helper functions and then wrote a whole bunch of statements here. If we check one of 'ab' and limit that to size 2, that should be equal to this set. It's gone off the page. Let's put it back where it belongs. One element of size 0, 2 elements of size 1, and 4 elements of size 2, just what you would expect. Here are sequences of a star, b star, c star of size exactly 4. Here they are and so on and so on. We've made all these tests. I should probably make more than these, but this will give you some confidence that the program is doing the right thing if it passes at least this minimal test suite.

1.26 26. 19 Theory and Practice

This is a good time to pause and summarize what we've learned so far. We've learned some theory and some practice. In theory, we've learned about patterns, which are grammars which describe languages, where a language is a set of strings. We've learned about interpreters over those languages, and about compilers, which can do the same thing only faster. In terms of practice, we've learned that regular expressions are useful for all sorts of things, and they're a concise language for getting work done. We've learned that interpreters, including compilers, can be valuable tools, and that they can be more expressive and more natural to describe a problem in terms of a native language that makes sense for the problem rather than in terms of Python code that doesn't necessarily make sense. We learned functions are more composable than other things in Python. For example, in Python we have expressions, and we have statements. They can only be composed by the Python programmer whereas functions can be composed dynamically. We can take 2 functions and put them together. We can take f and call g with that and then apply that to some x. We can do that for any value of f and g. We can pass those into a function and manipulate them and have different ones applying to x. We can't do that with expressions and statements. We can do it with the values of expressions, but we can't do it with expressions themselves. Functions provide a composability that we don't get elsewhere. Functions also provide control over time, so we can divide up the work that we want to do into do some now and do some later. A function allows us to do that. Expressions and statements don't do that because they just get done at 1 time when they're executed. Functions allow us to package up computation that we want to do later.

1.27 27. 20 Changing seq

Now one thing I noticed as I was writing all those test patterns is that functions like seq and alt are binary, which means if I want a sequence of 4 patterns, I have to have a sequence of (a, followed by the sequence of (b, followed by sequence of (c,d), and then I have to count the number of parens and get them right. It seems like it'd be much easier if I could just write sequence of (a, b, c, d). And we talked before about this idea of refactoring, that is changing your code to come up with a better interface that makes the program easier to use, and this looks like a good example. This would be a really convenient thing to do. Why did I write seq this way? Well, it was really convenient to be able to define sequence of (x,y) and only have to worry about exactly 2 cases. If I had done it like this, and I had to define sequence of an arbitrary number of arguments, then the definition of sequence would have been more complex. So it's understandable that I did this. I want to make a change, so let's draw a picture. Imagine this is my whole program and then somewhere here is the sequence part of my program. Now, of course, this has connections to other parts of the program. Sequence is called by and calls other components, and if we make a change to sequence, then we have to consider the effects of those changes everywhere else in which it's used. When we consider these changes, there are 2 factors we would like to break out. One is, is the change backward compatible? That is, if I make some change to sequence, am I guaranteed that however it was used before, that those uses are still good, and they don't have to be changed? If so, then my change will be local to sequence, and I won't have to be able to go all over the program changing it everywhere else. So that's a good property to have. So for example, in this case, if I change sequence so that it still accepted than that would be a backwards compatible change as long as I didn't break anything else. And then the second factor is whether the change is internal or external. So am I changing something on the inside of sequence that doesn't effect all the callers, than that's okay. In general, that's going to be backwards compatible. Or am I changing something on the outside -- changing the interface to the rest of the world? In this case, going from the binary version to this n_ary version, I can make it backwards compatible if I'm careful. It's definitely going to be both an internal and external change. So I'm going to have to do something to the internal part of sequence. And then I'm also changing the signature of the function, so I'm effecting the outside as well. I can make that effect in a backwards compatible way. Thinking about those 2 factors, what would be the better way to implement this call? Let's say we're dealing with the match-set version where we're returning a tuple, would it be better to return the tuple sequence (a, b, c, d) or the tuple sequence of (a, sequence of (b, sequence of (c, d)? Tell me which of these do you prefer from these criteria.

1.28 28. 20 Changing seq (Answer)

The answer is this approach is much better because now from the external part everybody else sees exactly the same thing. But internally, I can write the calls to the function in a convenient form and they still get returned in a way that the rest of the program can deal with, and I don't have to change the rest of the program.

1.29 29. 21 Changing Functions

Let's go about implementing that. Let's say I had my old definition of sequence of (x,y), and we say return the tuple consisting of a sequence x and y. Now I want to change that. How would I change that? Well, instead of x and y, I think I'm going to insist that we have at least 1 argument, so I can say x and then the rest of the args, and I can say, if the length of the rest of the args equals 1, then I've got this binary case and then I can take sequence of x. The second arg is now no longer called y. It's called args at 0. Else: now I've got a recursive case with more than 2 args, and I can do something there. So it's not that hard, but I had to do a lot of violence to this definition of sequence, and come to think of it, I may be repeating myself because I had to do this for sequence, and I'm also going to have to do it for alt, and if I expand my program, let's say I start wanting to take on arithmetic as well as regular expressions, then I may have functions for addition and multiplication and others, and I'm going to have to make exactly the same changes to all these binary functions. So that seems to violate the "Don't Repeat Yourself" principle. I'm making the same changes over and over again. It's more work for me. There's a possibility of introducing bugs. Is there a better way? So let's back up and say, what are we doing in general? Well, we're taking a binary function f of (x,y), and we want to transform that somehow into an n_ary function -- f prime, which takes x and any number of arguments. The question is, can we come up with a way to do that to automatically -- change one function or modify or generate a new function from the definition of that original function.

1.30 30. 22 Function Mapping

What's the best way to do that? How can I map from function f to a function f prime? One possibility would be to edit the bytecode of f. Another possibility would be to edit the source string of f and concatenate some strings together. Another possibility would be to use an assigment statement to say f = some function g of f to give us a new version of f. Which of these would be the best solution?

1.31 31. 22 Function Mapping

I think it's pretty clear that this one is the best because we know how to do this quite easily. We know how to compose functions together, and that's simple, but editing the bytecode or the source code, that's going to be much trickier and not quite as general, so let's go for the solution.

1.32 32. 23 n_ary Function

What I want to do is define a function, and let's call it n_ary, and it takes (f), which should be a binary function, that is a function that takes exactly 2 arguments, and n_ary should return a new function that can take any number of arguments. We'll call this one f2, so that f2 of (a, b, c) is = f(a, f(b, c)), and that will be true for any number of arguments -- 2 or more. It doesn't have to just be a, b, c. So let's see if you can write this function n_ary. Here's a description of what it should do. It takes a binary function (f) as input, and it should return this n_ary function, that when given more than 2 arguments returns this composition of arguments. When given 2 arguments, it should return exactly what (f) returns. We should also allow it to take a single argument and return just that argument. That makes sense for a lot of functions (f), say for sequence. The sequence of 1 item is the item. For alt, the alternative of 1 item is the item. I mentioned addition and multiplication makes sense to say the addition of a number by itself is that number or same with multiplication. So that's a nice extension for n_ary. See if you can put your code here. So what we're doing is, we're passed in a function. We're defining this new n_ary function, putting the code in there, and then we're returning that n_ary function as the value of that call.

1.33 33. 23 n_ary Function (Answer)

Here's the answer. It's pretty straight forward. If there's only 1 argument, you return it. Otherwise, you call the original f that was passed in with the first argument as the first argument, and the result of the n-ary composition as the other argument.

1.34 34. 24 Update Wrapper

Now how do we use this? Well, we take a function we define, say seq of x, y, and then we can say sequence is redefined as being an n_ary function of sequence. Oops -- I guess I got to fix this typo here. From now on, I can call sequence and pass in any number of numbers, and it will return the result that looks like that. So that looks good. In fact, this pattern is so common in Python that there's a special notation for it. The notation is called the decorator notation. It looks like this. All we have to do is say, @ sign, then the name of a function, and then the definition. This is the same as saying sequence = n_ary of sequence. It's just an easier way to write it. But there is one problem with the way we specified this particular decorator, which is if I'm in an interactive session, and I ask for help on sequence, I would like to see the argument list and if there is a doc string, I want to see the documentation here. I didn't happen to put in any documentation for sequence. But when I ask for help, what I get is this. I'm told that sequence is called n_ary function. Well, why is that? Because this is what we returned when we define sequence = n_ary of sequence. We return this thing that has the name n_ary function. So we would like to fix n_ary up so that when the object that it returns has the same function name and the same function documentation -- if there is any documentation -- and have that copied over into the n_ary f function. Now it turns out that there is a function to do exacty that, and so I'm going to go get it. I'm going to say from the functools -- the functional tools package. I want to import the function called update_wrapper. Update_wrapper takes 2 functions, and it copies over the function name and the documentation and several other stuff from the old function to the new function, and I can change n_ary to do that, so once I've defined the n_ary function, then I can go ahead and update the wrapper of the n_ary function -- the thing I'm going to be returning from the old function. So this will be the old sequence, which has a sequence name, a list of arguments, maybe some documentation string, and this will be the function that we were returning, and we're copying over everything from f into n_ary f. Now when I ask for help -- when I define n_ary sequence, and I ask for help on sequence, what I'll see is the correct name for sequence, and if there was any documentation string for sequence, that would appear here as well. So update_wrappers is a helpful tool. It helps us when we're debugging. It doesn't really help us in the execution of the program, but in doing debugging, it's really helpful to know what the correct names of your functions are. Notice that we may be violating the Don't Repeat Yourself principle here. So this n_ary function is a decorator that I'm using in this form to update the definition of sequence. I had to -- within my definition of n_ary -- I had to write down that I want to update the wrapper. But it seems like I'm going to want to update the wrapper for every decorator, not just for n_ary, and I don't want to repeat myself on every decorator that I'm going to define.

1.35 35. 25 Decorated Wrappers

So here's an idea. Let's get rid of this line, and instead, let's declare that n_ary is a decorator. We'll write a definition of what it means to be a decorator in terms of updating wrappers. Then we'll be done, and we've done it once and for all. We can apply it to n_ary, and we can apply it to any other decorator that we define. This is starting to get a little bit confusing because here we're trying to define decorator, and decorator is a decorator. Have we gone too far into recursion? Is that going to bottom out? Let's draw some pictures and try to make sense of it. So we've defined n_ary, and we've declared that as being a decorator, and that's the same as saying n_ary = decorator of n_ary. Then we've used n_ary as a decorator. We've defined sequence to be an n_ary function. That's the same as saying sequence = n_ary of sequence. Now we wanted to make sure that there's an update so that the documentation and the name of sequence gets copied over. We want to take it from this function, pass it over to this function because that's the one we're going to keep. While we're at it, we might as well do it for n_ary as well. We want to have the name of n_ary be n_ary and not something arbitrary that came out of decorator. So we've got 2 updates that we want to do for the function that we decorated and for the decorator itself. Now let's see if we can write decorator so that it does those 2 updates. So let's define decorator. It takes an argument (d), which is a function. Then we'll call the function we're going to return _d, and that takes a function as input. So it returns the update wrapper from applying the decorator to the function and copying over onto that decorated function, the contents of the original function's documentation and name, and then we also want to update the wrapper for the decorator itself. So from (d) the decorated function, we want to copy that over into _d and then return _d. Now which update is which? Well, this one here is the update of _d with d, and this one is the update of the decorated function from the function. So here we're saying the new n_ary that we're defining gets the name from the old n_ary, the name in the documentation string, and here we're saying the new sequence, the new n_ary sequence, gets its name from the old sequence. Here's what it all looks like. If you didn't quite follow that the first time, don't worry about it. This is probably the most confusing thing in the entire class because we've got functions pointing to other functions, pointing to other functions. Try to follow the pictures. If you can't follow the pictures, that's okay. Just type it into the interpreter. Put these definitions in. Decorate some functions. Decorate some n_ary functions. Take a look at them and see how it works.

1.36 36. 26 Decorated Decorators

Okay, now for a quick quiz. We have this definition of decorator, and we've seen how that works. Here's an alternative that was proposed by Darius Bacon, which is this1 line -- return the function that updates wrapper for the decorator applied to the function from the original function, and then 1 more line, which says, decorator = decorator of decorator. Can you get any more recursive than that? The question is, does this work? I want you to give me the best answer. The possible answers are yes, it does; no, it's an error; no, it correctly updates decorator such as n_ary, but not the decorated function such as (seq); or no, my brain hurts.

1.37 37. 26 Decorated Decorators (Answer)

Now if you answered, no, my brain hurts -- well, who am I to argue with that? But I think the best answer is yes, it does in fact work. Even though there's only 1 update wrapper call, both of them happen, and the reason is because decorator becomes a decorator. So this version of decorator updates (seq), and then this version that gets created when we make it a decorator updates the decorator.

1.38 38. 27 Cache Management

If you took CS 101, you saw the idea of memoization. If you haven't seen it, the idea is that sometimes particularly with the recursive function, you will be making the same function calls over and over again. If the result of a function call is always the same and the computation took a long time, it's better to store the results of each value of N with its result in a cache, a table data structure, and then look it up each time rather than try to recompute it.

We can make this function be cached very simply with a couple extra lines of code. We ask if the argument is already in the cache, then we just go ahead and return it. Otherwise, we compute it, store it, and then return it. So this part with the dot, dot, dot, is the body of the function. All the rest is just the boiler plate that you have to do to implement this idea of a cache. We've done this once, and that's fine, but I'm worrying about the principle of Don't Repeat Yourself. There's probably going to be lots of functions in which I want to store intermediate results in a cache, and I don't want to have to repeat this code all of the time. So this is a great idea for a decorator. We can define a decorator called memo, which will go ahead and do this cache management, and we can apply it to any function. The great thing about this pattern of using memoization is that it will speed up any function f that you pass to it because doing a table look-up is going to be faster than a computation as long as the computation is nontrivial, is more than just a look-up.

Now the hockey player, Wayne Gretzky, once said that you miss 100% of the shots you don't take. This is kind of the converse. This is saying you speed up 100% of the computations that you don't make. So here's the memo decorator.

```
@decorator¶
def memo(f):¶
    """Decorator that caches the return value for each call to
f(args).¶
    Then when called again with same args, we can just look it up."""¶
    cache = {}¶
    def _f(*args):¶
        try:¶
            return cache[args]¶
        except KeyError:¶
            cache[args] = result = f(*args)¶
            return result¶
        except TypeError:¶
            # some element of args can't be a dict key¶
            return f(args)¶
    return _f¶
¶
```

The guts of it is the same as what I sketched out previously.

If we haven't computed the result already, we compute the result by applying the function f to the arguments. It gives us the result. We cache that result away, then we return it for this time. It's ready for next time. Next time we come through, we try to look up the arguments in the cache to see if they're there. If they are, we return the result.

And now I've decided to structure this one as a try-except statement rather than an if-then statement. In Python, you always have 2 choices. You can first ask for permission to say are the args in the cache, and if so, return cache or args, or you can use the try-except pattern to ask for forgiveness afterwards to say, I'm first going to try to say, if the args are in the cache, go ahead and return it. If I get a keyError, then I have to fill in the cache by doing the computation and then returning the result.

The reason I use the try structure here rather than the if structure is because I knew I was going to need it anyways for this third case. Either the args are in the cache, or they aren't, but then there's this third case which says that the args are not even hashable.

What does that mean?

Start out with a dictionary d being empty, and then I'm going to have a variable x, and let's say x is a number. If I now ask, is x in d? That's going to tell me false. It's not in the dictionary yet. But now, let's say I have another variable, which is y, which is the list [1, 2, 3] and now if I ask is y in d? You'd think that would tell me false, but in fact, it doesn't. Instead, it gives me an error, and what it's going to tell me is type error: unhashable type: list.

What does that mean?

That means we were trying to look up y in the dictionary, and a dictionary is a hash table -- implemented as a hash table. In order to do that, we have to compute the hash code for y and then look in that slot in the dictionary. But this error is telling us that there is no hash code for a list.

Why do you think that is?

Are lists unhashable because:

- lists can be arbitrarily long?
- lists can hold any type of data as the elements, not just integers?
- lists are mutable?

Now I recognize this might be a hard problem if you're not up on hash tables. This might not be a question you can answer. But give it a shot and give me your one best response.

1.39 39. 27 Cache Management (Answer)

The answer is because lists are mutable. That makes them unlikely candidates to put into hash tables. Here's why. Let's suppose we did allow lists to be hashable. Now we're trying to compute the hash function for y, and let's say we have a very simple hash function -- not a very good one -- that just says add up the values of the elements. Let's also say that the hash of an integer is itself, so the hash code for this list would be equal to 6, the sum of the elements. But now the problem is, because these lists are mutable, I could go in, and I could say, y[0] = 10. Y would be the list [10, 2, 3]. Now when we check and say, is y in d? We're going to compute the hash value 10 + 2 + 3 = 15. That's a different hash value than we had before. So if we stored y into the dictionary when it had the value 6, and now we're trying to fetch it when it has the value 15, you can see how Python is going to be confused. Now, there's 2 ways you could handle that. One -- the way that Python does take is to disallow putting the list into the hash table in the first place because it potentially could lead to errors if it was modified. The other way is Python could allow you to put it in, but then recognize that it's the programmers fault, and if you go ahead and modify it, then things are not going to work anymore, and Python does not take that approach, although some other languages do.

1.40 40. 28 Save Time Now

Now to show off how insanely great memo is, we'll want to have before and after pictures, showing the amazing weight loss of a model that was fat before and was thin after applying memo. Oh! Wait a minute. It's not weight loss. It's time loss that we're going to try to measure. We want to show that before, we have a function f, and that's going to run very slowly, making us sad. And after, we have a function memo of f, and that's going to run very quickly, making us happy. Now we could do that with a timer and say it took 20 seconds to do this and .001 seconds after, but instead of doing it with timing, I think it's a little bit more dramatic just to show the number of function calls, and I could go into my function and modify it to count the number of function calls, but you could probably guess a better way to do that. I'm going to define a decorator to count the calls to a function because I'm probably going to want to count the calls to more than 1 function as I'm trying to improve the speed of my programs. So it's nice to have that decorator around. So here's the decorator countcalls, you pass it a function, and this is the function that it returns. It's going to be a function that just increments entry for this function in a dictionary callcounts. Increment that by 1 and then go ahead and apply the function to the arguments and return that result. We have to initialize the number of calls to each funciton to be 0, and that's all there is to it. So here I've defined the Fibonacci function. Through cursive function it calls itself twice for each call, except for the base case. I can count the calls both with and without the memoized version. So I'm going to do before and after pictures -- before and after memoizing. So here's before. I have the values of n and a value computed for Fibonacci number of n, the number of calls created by countcalls, and then I have the ratio of the number of calls to the previous number. We see the number of calls goes up by the time we get up to n = 20. We got 10,000 calls. We can scroll down, and by the time we're up to n = 30, we have 2.6 million calls. And here's the after. Now we've applied the memo decorator. Now the number of calls is much more modest. Now at 20, we're only at 39 calls, and at 30, we're at 59 calls rather than 2.6 million. So that's pretty amazing weight loss to go from 2.6 million down to 59, just by writing 1 little decorator and applying it to the function. Now just as an aside here, and for you math fans in the audience, I'm back to the before part. This is without the memoization. This number here in this column is the ratio of the number of calls for n = 30 to the number of calls for n = 29. You can see that it's converging to this number 1.6180. Math fans out there, I want you to tell me if you recognize that number. Do you think it's converging to 1 + square root of 5 / 2, or the square root of e?

1.41 41. 28 Save Time Now (Answer)

The answer is 1 + square root of 5 over 2, otherwise known as the Golden Ratio. The Golden Ratio I knew was actually the ratio of success of elements of the Fibonacci sequence -- the sequence 1,1, 2, 3, 5, 8, and so on -- converges to that ratio. But I didn't know that the number of calls in the implementation also converges to that ratio. So that's something new.

1.42 42. 29 Trace Tool

I want to make the point there are different types of tools that you can use in your tool box. We just saw the count calls. I think I would classify that as a debugging tool, and we saw a memo, and I'll classify that as a performance tool. Earlier, we saw n_ary, another decorator, which I can classify as an expressivenes tool. It gives you more power to say more about your language. This gives you no more power but makes it faster. This isn't going to end up in your final program, but helps you develop the program faster. I want to add another tool here in debugging called trace, which can help you see the execution of your program. So I'm going to define a decorator trace, which when we apply to fib, gives us this kind of output. When I ask here for what's the 6th Fibonacci number? It says for each recursive call, we have an indented call with an arrow going to the right saying we're making a call, and for each return, we have an arrow going to the left. When you ask for fib of 6, you keep on going down the list until we get near the end. When we ask for fib of 2, then that's defined in terms of 1 and 0, and they both return 1, so that means fib of 2 returns 2 and so on. We can see the shape of the trace here as we go. It's coming to the right and then returning back and coming to the right some more and returning back. The pattern takes a long time to reveal itself and would take even longer for larger arguments other than 6. But it gives you some idea for the flow of control of the program. So that's a useful tool have, and here's an implementation. It follows the same pattern as usual. Decorator takes a function as input. We're going to create another function, and this is what it's going to look like. We're going to figure out what it is that we're going to print. We're going to keep a variable, which we keep as an attribute of the trace function itself called the level. We'll increment that as we come in, print out some results here. We initialize the trace level to 0 -- the indentation level -- and then finally, we return the function that we just built up. I've left out some bits here, and I want you to fill them in to make this function work properly to show the trace that I just showed.

1.43 43. 29 Trace Tool (Answer)

So the code you had to write was pretty straight forward. Like most decorators, we compute the result here by applying the function to the args, then we return the result down here. Maybe a little bit tricky is what's in this finally clause, which is we're decreasing the trace level. So the indentation level goes up by 1 every time we enter a new function and down by 1 every time we go back. The issue here is, we want to make sure that we do decrement this. If we increment this once, and then calling the function results in an error, and we get thrown out of that error, we want to make sure we put it back where it belongs. We don't want to mess with something and then not restore it. So that's why we put this in a try finally.

1.44 44. 30 Disable Decorator

We're coming to the end of what I want to say about decorators. I wanted to add one more debug tool. That's one I'm going to call disabled. It's very simple. Disabled is another name for the identity function -- that is the function that returns its argument without doing any computation on it whatsoever. Why do I want it and why do I call it "disabled?" Well, the idea is that if I'm using some of these debugging tools like trace or countcalls, I might have scattered throughout my program trace define f and some other traced functions. Then I might decide I think I'm okay now. I think I've got it debugged. I don't want to trace any more. Then what I can do is I can just say "trace = disabled" and reload my program, and now the decorator trace will be applied to the function, but what it will do is return the function itself. Notice we don't have to say that disabled is a decorator, even though we're using it as if it were one, because it doesn't create a new function. It just returns the original function. That way we won't have the trace output cluttering up our output, and the function will be efficient. There won't even be a test to see if we are tracing or not. It'll just use the exact function that passed in.

1.45 45. 31 Back to Languages

Now we're done with decorators, and I want to go back to languages. The first thing I want to say is that our culture is full of stories of wishful thinking. We have the story of Pinocchio, who wishes to be a real boy, and the of Dorothy, who wishes to return from Oz to her home in Kansas by clicking her shoes together. They find that they have the power within them to fulfill their wishes. Programming is often like that. I'm righting the body of the definition of a function, and I wish I had the function "fib" defined already. If I just assume that I did, then eventually my wish will come true. In this case, it was good while writing the right-hand side to just assume I wish I had the function I want and proceed as if you did, and sometimes it's a good idea to say I wish I had the language I want and proceed as if you did. Here's an example. Suppose you had to deal with algebraic expressions and not just compute them the way we can type this expression into Python and see its value if x is defined but manipulate them. Have the user type them in, modify them, and treat them as objects rather than as something to be evaluated. Now, my question for you is it is possible to write a regular expression which could recognize expressions like this that are valid. Is the answer, yes, it is possible to write that? No, it's not possible because our language we're trying to define has the plus and asterisk symbols and those are special within regular expressions? Or no, we can't parse this language because this language includes parentheses, and we can't do balanced parentheses for the regular expressions. If you're not familiar with language theory, this may be a hard question for you, but go ahead and give it a shot anyways.

1.46 46. 31 Back to Languages (Answer)

The answer is that the problem is the balanced parentheses. Regular expressions can handle a set number of nesting parentheses -- one or two or three -- but they can't handle an arbitrary number of nestings. It makes sure that all the left parentheses balance with the right parentheses. We're going to need something else. The thing we traditionally look at is called context-free languages, which are more general than the regular languages. We'll see how to handle that.