The Essentials of 61A

December 12, 2023

1 Introduction

Programming consists of a progamming language, problem solving and abstraction. This is the content of 61A. In slightly more detail, a programming language is the interface between people and computing where we can easily manage and compute with data. Inherent in the process of programming is problem solving with various structures and just as importantly programming is the process of abstraction which is both how we all think and is central to engineering design.

We will introduce some frames of common and powerful threads. The full truth is often more complicated than brief (though powerful) statements which communicate continue the main thread.

One challenge is the difficulty of defining useful things. Is a hammer something that hammers? Or is it a piece of wood, with an steel top?[1] This challenge will lead to some back and forth in both these notes and in your gaining of understanding. What roles does the piece of wood serve or the steel serve in making the hammer effective. [2]

A central dogma in computer science is the notion of a **computer being a processor and memory where any memory location can be accessed in one step.** This one is quite clean and powerful and false, but computer architects have given us a quite good approximation of this ideal.

A second central frame is the stored program: **the instructions are stored in memory along with data.**

Then again, even these idealized frames are useless to any normal human wishing to do powerful things with a computer.

Hence, we have programming languages to interface with these basic views.

And again more fundamentally, we have the notion of problem solving including correctness, encodings, and abstraction. These are human things that should be expressed in a human language, in this course, English, and often using basic mathematical knowledge. [3]

[1] There is this thing called epistomology which is a "theory of knowledge" where such examples are discussed at length.

[2] There are corresponding concepts in programming like syntax, semantics and implementations.

[3] Truly basic and deep, essentially the notion of the natural numbers as infinite and successive.

That part is always there and then we have the part where our approaches or thinking is translated to code.

So, we will begin with English (and mathematical principles) as our language for reasoning and thought in the first section or two. Then we will get into computer languages and the connection to our way of thinking as humans (and mathematical principles.)[4]

1.1 Mathematical induction, recursion, and recursive structures.

The memory is of arbitrary size, or inputs to programs are of arbitrary size. This brings up the notion of size which is measured by the natural numbers: $0, 1, \ldots,$. That is, we have 0, and then 1, and for every natural number there is a next one. They are also called the counting numbers as one can count with them, including how many objects one has in an input to a program.

Statements about correctnes of a computer program are thus statements about natural numbers. Specifically, the statment one wishes to verify is that your program is correct for inputs for any size.

As an example, consider finding the maximum element of a set of objects, S. A method would be to removing a number, x, to form a set S/x, and then finding the maximum element, y, of S/x. We finish by returning the maximum of x and y. Of course, if there is one element in S, then it is the maximum element.

Here, we implicitly used the concept of maximum which assumes an ordering on the objects, and asks for an *element in the set that is larger than all the other elements.* To be sure, the objects themselves could have been numbers or words as each have natural orderings where the word larger has meaning. We also use the idea of *transitivity* of an ordering which is the property that if a is greater than b and b is greater than c, then a is greater than c.

Finally, we use something called the principle of induction. The number of objects is a natural number: $0, 1, 2, \ldots$, and we can use the assumption that our procedure is correct for smaller sets to conclude that it is correct for a set of the "current" size.

That is, to argue correctness,assume that the correct maximum is found by this procedure for smaller sets, and then can conclude that the maximum of x and y is the global maximum since if x is larger than y, then x is larger than z for any z in S/x as y is larger than z . If there is one element than, that is clearly the maximum element.

This argument argues the procedure works for size 1 and then shows that if it works for size k, it also works for size $k + 1$. This form of argument proves that the procedure works for sets of arbitrary size. This type of argument follows from the definition of natural numbers, which again is start at 0, and then there is always a next natural number.

In this particular case, we started the argument at 1 to show our procedure works for all sizes that are 1 or larger. Specifically, we showed that it works for 1 (as the maximum for a 1 element

[4]Don't be frightened, the mathematical principle is large just the notion that the natural numbers start with 0, and there is always a next one. That is, 1 is after 0, 2 is after 1, and so on.

set is the element), and then for any $k+1$, the procedure works under the assumption it works for size k.

An alternative procedure for finding a maximum is to split the sets into two sets of equal (or almost equal) size, finding the maximum of each, and returning the maximum of the pair. The argument is that the resulting value is at least the value of any element of either set, as each is the maximum of the smaller set. Again, this works due to the idea of natural numbers, that is, we assume our procedure works for smaller sets and produces the correct maximum value and then make an argument that it produces the correct maximum for the entire set. The argument uses transitivity once again.

This is sometimes called the principle of strong induction. If something works for 0, and if for any value k, one can argue it correctness using assumptions about correctness for smaller values of k, then it works for all natural numbers.

The two methods above give different ways to break down sets. We can flip it around and think about defining a set. A view that follows our first procedure, is that a set is either a single element, or a set plus a single element. In the second view, a set is a single element, or the union of two sets. Either view makes our intuitive notion that a set is a bunch of elements concrete and precise.

Both can be viewed as *recursive* definitions as we used the concept of a set in the definition of set, e.g., a set is the union of two sets, or a set is a set and one more element.

This link between the natural numbers and solving problems or defining the objects we work with is necessary, deep and magical. The power is essentially just the magic of there is always a successor to each natural number.[5] And you learned about it in elementary school!

You might have noticed, that we gave a procedure in English. For computers, we use programming languages to express our solutions to problems. As you saw even in this simple example, problem solving is centrally tied to induction or alternately recursion.[6] And even here, the structure of a set requires the idea of induction or recursionw to define. In programming languages, recursive definitions arise almost immediately and we will get to it shortly.

Before continuing, we consider the problem of sorting a set of objects. One procedure is to find the maximum element, set it aside, sort the remaining objects and the place the maximum element last. Alternatively, one could split the set of objects into two, sort each and merge the sorted sets by repeatedly choosing the least element of the two sorted sets and placing that least element next in the ordering of the set. Again, we have given two methods to produce our output, one that removes a single element and sorts the remaining, the other splits it into two and relies on sorting each. The procedures work since we decompose the problem to eventually reach single element sets. That is, we get to a smallest set which is trivial to sort, and from there we can build our solution. Again, the correctness follows from correctness on smaller sets, and the idea of smaller comes from the concept of the natural numbers.

[5]Technically, this notion is one of Peano's axioms which which forms the basis of mathematical reasoning.

[6]Essentially, recursion is the idea of defining a concept in terms of itself. It avoid circularity throuh size, or the ordering of the natural numbers.

1.2 The recursive leap of faith.

In the maximum and sorting examples, we assumed the procedure worked on smaller inputs. We refer to this as *recursive leap of faith* and it seems magical. But the magic is from the notion of natural numbers. While 0 is just 0, 1 contains the concept of 0, or a 100 contains the concept of all the previous numbers. When we say 100, we don't need to say 99, 98, 97 and so on. The "so on" itself is, in a sense, a leap of faith. The program works for smaller inputs or inputs of size 99, 98, 97 and so on is the same concept. Indeed, it is not even metaphor, it is the same mathematical concept.

1.3 Humanity, languages, problem solving, and abstraction.

In the previous section, we gave several algorithms that could be implemented on a computer. Two for finding the maximum element of a set and two for sorting a set. We described them using English, and argued correctness using English.

Throughout this course, you will develop your strategies using English (or whatever language you like to reason in). That will remain the starting point. Eventually, you will implement your strategies in a programming language.

Above, we used the power of names in our solution. For example, for maximum, we referred to a *first object* to specify it. Moreover, we worked in several contexts: in sorting method, we first found the maximum then sorted the remaining elements. Each time, finding the maximum the notion of the first object was local to that particular context of which set we were working in.

A bit less technically, we often use names. There is someone often called Bob. Sometimes we are speaking of a Bob in the room we happen to be in as there is more than one Bob in the world. This localization of reference is interesting and important. Sometimes, in our local context, we refer to the President. If there is no person who has the role of president in that room, we typically understand that we are referring to Joe Biden (at the time of this writing.) The concept of naming, of localizing names, and sometimes finding the least enclosing context to find a name are all natural to us. They are also fundamental to programming languages.

Another aspect of English (or human languge) in human endeavors is our ability to break down systems into parts by abstracting what they do or what they are. For example, a chair is an object with legs and an object we sit on. A chair consists of of legs, which are made of possibly wood or metal. The concept of a chair is an abstraction. Actions also can be abstractions. Driving consists of steering, stepping on the gas pedal, and braking, and of course, hopefully watching where you are driving. But, we simply say to someone, please drive here. Indeed, this notion of abstraction applies at many levels; a restuarant contains cooks, hosts, servers, managers, logistics. All this is setup into a sytem that allows us to eat (hopefully) delicious food. Of course, there are other systems, ranging from the design of a car, to a healthcare system, or even just organizing your household. But this compartmentalization is there in so much of what we do as people.

In computer science, we refer to the act of defining this compartmentalization as abstraction.

Abstraction is also key to programming, both for problem solving (where we used the tool of finding a maximum to sort) and in organizing our systems. Thus programming languages must have the power that we humans have in our languages, but they should also be simple, where possible.

Let's get to it.

2 Programming Language.

A programming language consists of a few fundamentals. Let's see.

2.1 Values,expressions and evaluation.

For programming, essential components are:

- **values,** such as numbers 1 or 2 or 1.5, or text such as "hello" or "world"
- **expressions,** such as $3 + 3$, which *evaluates* to a value of 6.

Oh my. Already we have the hammer issue. Here, an expression is a thing, "3+3", but its value is 6. The connection here is the process of *evaluation.* It is both central to understanding programming and natural enough, so let's move forward.[7]

2.2 Types, objects.

The concept of types of values is both central and a bit of an aside. With values, one sees different kinds of values. Integers, decimal numbers, and pieces of text, or strings. Moreover, we see that the definition of operations depend on the type. For example, addition means something for both integers and for decimal numbers. Moreover, one can think of addition of strings as putting them together or concatenating.

This raises a choice in terms of presentation. Types of values is always fundamental and programming languages typically (perhaps always) have some way to define your own types of values.

Here, I might encourage a reader to go ahead and look at the discussion of objects in section **??**.

This is, again, the hammer problem; how do you define a hammer without understanding what it means to swing it. We urge some patience in the reader with our jumping forward and then revisiting the basics as the concepts are so intertwined. Some back and forth by the reader may be helpful.

[7] We can't resist noting that at this point, we have a calcular which is woefully short of programming.

2.2.1 Notation and Evaluation.

Here, we jump into a bit of notation. The notion of value is translated into specific expression as syntax and meaning. That is, in text, a value may be 3 or 4. We say this using the notation `<value>` being possibly many things in text. That is `<value>` could be 0 or `<value>` could be written as 1.

Some call this a grammar, as in english text, a `<sentence>` could follow the pattern: `<subject> <predicate>`. Then one can fill in subject with specific nouns, and the `<predicate>` could be a `<verb>` and `<object>`.

In any case, here we go with values and expressions.

- **Value:**

  ```
  <value> = 0
  ....
  ```

  Here the list is long. That is, `<value>` could be 'hello' or 'world' or 1, and so on.

- **Expression:**

  ```
  <expression> = <value>
  <expression> = <name>
  <expression> = <expression> <operator> <expression>
  <expression> = ( <expression> <operator> <expression> )
  ```

  Here, `<name>` could be something like 'a', or 'bob', and `<operator>` could be '+', or '*'. Examples include :'3', '3 + 3', or the name 'a'. The list of possible operators is more extensive. Expressions **evaluated** into values.

2.2.2 Recursion already.

In the previous notation, we see the following.

```
 <expression> = <expression> <operator> <expression>
   <expression> = {\bf (} <expression> <operator> <expression> {\bf )}
```

Here, the definition of expression uses the definition of expression itself. That is, this is a "recursive" definition. Let's consider the form $(3+4)+2$. Here, $3+4$ is an expression because each of 3 and 4 are values and as such expressions. Moreover 2 since it is a value. Thus the $3+4+2$ is an expression that consists of the expression $3+4$, an operator, and the expression 2.

One can get arbitrarily complicated here. You perhaps were forced to do such things with complicated mathematical expressions as a child. We will allow you to do so on your own. The resulting pictures are often called expression trees. For example, we could draw one for $(3 \times 4) + 2$ as follows.

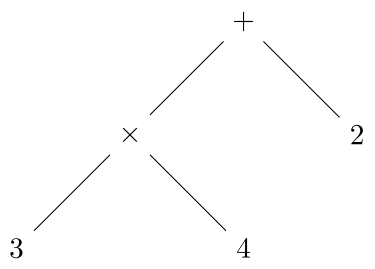

Figure 1: The expression tree for the expression, $(3 \times 4) + 2$.

But a power here is and my very strong advice that you don't have to make it complicated.

Instead, we can stick to the evaluation rule where one evaluates the expression as either a value if it is one, or uses the evaluation rule "recursively" on each sub-expression to produce a value and then using the evaluation rule that comes with the operator, e.g., '+', that then combines the resulting value.

This pattern of recursive definition and evaluation is absolutely central.

And frankly, understanding it in this context, will get you a long way in this course. Indeed, we will start introducing a mantra about expressions whose echos you will hear throughout these notes. That is:

An *expression* is a value or *one or two expressions with an operator.*

Here again an expression is defined in terms of expressions and a base case of a value.

2.2.3 Play around.

At this point, if not previously you should be using the interpreter. In particular, go to code.cs61a.org. Play with the interpreter. Evaluate a few expressions, and values!

Generate expressions recursively. That is, think of $(3 + 4)$, as $((1 + 2) + 4)$. Imagine the possibilities. But do it step by step to understand the power of recursion.

No. Don't read on. Really, just go and do this!

2.3 Everything, all at once, some of how and why.

To be a program one continues with:

- **bindings**, such as $a = 3$ or $b = 4$, and $c = a + b$, where *the name a is bound to 3*, the name *b is bound to 4* and the expression $a + b$ is *evaluated* by evaluating a to be the value 3 and b is evaluted to the value 4 and finally the expression evaluates to 7. Then c is bound to the value 7.

We must stop again and consider humans who name things, for example, “Bob”. Its an association.

Bob can be complicated, but in programming, an association between names and values is straightforward, e.g., the name a corresponds to the value. Yet it is very powerful and essential.

Still, these names form a frame for us to deal with the world. And in computing, the notion of frame is central and we make it explicit.

A **frame** is an association between a set of names and values. [8]

Play around: Again, use the interpreter: perhaps at code.cs61a.org. Assign a variable. Use it in an expression, evaluate the expression.

Depending on circumstances, our behavior differs, as should a program. Thus, programs have a necessary component:

- **control**, such as *if speed > speed_limit, then slowdown.*

The importance of this is self-evident, and we leave it at that.

To review, we have values, expressions (which need evaluating), bindings (which require frames), and control. We snuck in the notion of instructions or statements: $a = 3$, $c = a + b$, and **if a > 3, then b = 5**.

Both of these are **statements**. The former are assignement statements and the latter is a conditional statement (with an assignment statement inside). Recall, expressions are evaluated. We say a statement is **run** or **executed** as it does not simply return a value, but either controls the program or creates a binding.

Notice the conditional statement **If a > 3, then b = 5** requires that **a** has been bound. Thus, the statement is run in the frame where **a** is bound.

Returning to the notion of “Bob”. But there can be more than one “Bob”. But in a small group or a room “Bob” is just this Bob.

This is powerful concept for us in our communication and thought and requires the introduction of another essential component:

- ***function***, such as “square(x)” corresponding to a procedure that computes $x * x$.

This is fancy and really to some people, completes the notion of a programming language. So, let’s take a moment.

Ok now, what is a function in a programming language?

For the square function, the relevant concepts are the name of the function **square**, the name x which is called its *formal parameter*, and an expression or set of statements that eventually give $x * x$.

[8] The name frame, in fact, comes from a theory of knowledge. Meaning comes from the “frame”. That is, a leg in the frame of a table, its meaning in the frame of the phrase, “You don’t have a leg to stand on.” which is both a leg of a person and means that your argument is not supported.

More precisely, a function itself is a set of*formal parameters* and *a body* or in this class[9] a *suite* of statements.

The function is a new kind of value in a program. And it can be bound to a name. In our example, the function is bound to the name ***square***.

A function is only interesting in that it can be ***called***.

A ***call expression*** uses a function to compute or process. We will describe the process using the language of evaluation and execution.

For example for the function "square(x)", one can have a call expression, "square(3)". The evaluation of this expression is done by binding the name "x" to 3, and then evaluating a statement "x*x" which is, in turn, done by evaluating each of the occurences of x to the value 3 and then multplying to two together to obtain the value 9.

Indeed, one can use our concepts to do more. If we had already had a statement, $a = 3$, we could evaluate the call expression "square(a)" to obtain the value 9. Or evaluate the call expression "square(a+a)" as the value 36, as $a + a$ evaluates to the value 6 and the square of that value is 36.

In general, a call expression to a function consists of the function name and expressions corresponding to the **parameters** of the function. Each expression is evaluated and bound to the name of the formal parameter. Then the statements in the body are run using these bindings.

Using the notion of ***frame***, a function call creates a frame where its parameters are bound to the expressions in the call and then runs its statements using that frame.

In terms of "Bob", we can now have "Bob" just to mean what the name is bound to in that function call. That is, "Bob" can mean something very particular in a particular context.

There is one more concept that we all use in a local context. For example, consider lecture, who is the President? Typically, there is no one in the room that is bound to the notion of the President, and most of us would respond with "Joe Biden" or whomever is the current President.

In a function, the code may also contain names that are not bound in the formal parameters. In this case, that name is evaluated in the frame that existed when the function was defined. That is, the frame that contains the name of the function itself. In our square example, when square is defined, the name square is bound to a value which is a function whose parameter(s) is "x" and body is "x*x".

An example, might be a function which "times_x(x)" whose body contains "y*x". When evaluating the call expression "times_x(3)", a frame is created where "x" is bound to 3, and when evaluating the expression "y*x", the name "y" is not found in the frame created for the function call expression for "times_x(3)" and thus is searched for in the frame where "times_x(x)" is defined.

Ok. That was a bit hard. It's fine. We will come back to the details.

Still, the example motivates the definition of an **environment** as follows:

An *environment* is a frame and possibly a parent environment.

Moreover, one searches for a name in the current frame and then in the parent environemt.

[9] This is due to the use of python

Does this seem familiar. Indeed, this definition is *recursive* where concept, the environment, is defined with the concept environment itself. Moreover, the usage of the environment in finding bindings is recursive as well. That is, look in the current frame and failing there, look in the parent environment.

A final and very important point is that a function is a value. And that it can be defined anywhere. This is not in all programming languages, but it is in many and central to this class. Central enough to make the following definition.

A function is a value and such is *first class.*

This allows, for example, for the function to be passed as an argument to another function. This will be very useful, shortly.

Now we are basically finished with what a programming language is.

We will review them, with some notation that describes the way these concepts appear specifically in python.

First. Quickly think about the following concepts: values,expressions,bindings,conditionals, and functions. Moreover, think about these concepts with the idea of their meaning in terms of evaluation, names, frames and environments.

Here, we have a lot of how humans generalize and abstract. We make manipulations (as expressions do), we name things, and our names have context (as functions give us.) An one more thing that is about how we think abstractly and that will come. The notion is that a chair has a leg, and a human has a leg. They mean different things. This is done with a concept called objects and their type. Humans and chairs are different types of things. That is soon to come in our discussion of programming.

It is a lot, but it is also, a very long way towards what a program is. Let's continue.

2.4 Everything, all at once : Notation and Pythonics.

Recall, like a hammer, there has to be a mix of what something does, what it is, and how it is done. Thus, in what is below, will also discuss both expressions, values, conditionals, bindings, and functions as well as the notion of evaluation, frames, and environmenments together.

Here again are the concepts we introduced with some notation that is a partial specification of them in python. It is a compromise between. It does cover all the concepts discussed. Its may be intimidating, do try to parse it. It's worth it. [10]

- **Value:**

```
<value> = 0
....
```

[10] Also, for what it's worth, python has some specifics regarding indentations as having meaning. At this point, we don't follow that structure in this semi-formal treatment.

Here the list of possible values is long, even infinite. 1,2,1.5, etc.

- **Expression:**

  ```
  <expression> = <value>
  <expression> = <name>
  <expression> = <expression> <operator> <expression>
  ```

  Here, ¡name¿ could be something like 'a', or 'bob', and ¡operator¿ could be '+', or '*'. Examples include :'3', '3 + 3', or the name 'a'. The list of possible operators is more extensive.

  Expressions **evaluated** into values.

- **Assignment Statement:**

  ```
  <name> = <expression>
  ```

  Assignment statements create **a binding** between the ¡name¿ and the value of the expression.

- **Conditional Statement:**

  ```
  <cond> = {\bf if}  <boolean>: <statements>
  ```

  There are other forms, but this is sufficient as an example. A ¡boolean¿ is an expression that can be interpreted as true or false, e.g, 'a ¿ 3'. The interpretation can be broad, but we leave that for the future.

- **Function:**

  - **Definition.**

    ```
    {\bf def} <fname> (<params>): <statements>
    ```

    This statement creates a "function" value which consists of the parameters, the statements, and tracks the (parent) frame where the **def** statement is executed. ¡params¿ is a list of names.

    The name ¡fname¿ is bound to this function value.

  - **Call Expression.**

    ```
    <fname> (<exp1>,...)
    ```

Here fname is bound to a function value. The call expression creates a frame and an eviroment consisting of the frame and the parent environment association. The parameters are bound to the values of the corresponding expressions in the new frame. There is a type of statement: **return ¡exp¿** statement where the code stops executing and the value of exp becomes the value of the call expression.

There are (many) details left out, but these are the essentials.

2.4.1 Play around.

Use the interpreter: perhaps at code.cs61a.org. Write a function, perhaps absolute value, it should have a conditional in it.

Also, it may be a good time now to explore Python tutor.

Cut and paste your code into it, it gives some nice visualizations of frames as functions are called. Try to explore your understanding of the above discussion in these visualizations. Think about your personal model of what happens with assignments, conditionals, function definition and calls, and their evaluations.

3 Data and aggregates.

Again, to organize ourselve, we often write things down. We need paper (or these days, computers or phones.)

The space for writing these notes is fundamental as is organizing them. Space and storage is fundamental to programming as well.

In this section, we formulate this programming language concept.

3.1 Minimal and everything.

Values like numbers are simple: 1, 2, 1.5, etc.[11]

A simple extension to values is pairs of values.

```
<pair> = <value>,<value>
<value> = <pair>
```

This defines a pair and adds it as a fundamental value in our (abstract) programming language. We could thus do something like, $p = (3, 4)$.

Of course, given a pair, we may wish to access its first value or its second, which we do do by using a functions on the pair, p, **first(p)** and **second(p)** which evaluate to 3 and 4, respectively.

[11]Strings are seemingly simple, but they are actually data in the sense of that a string could consists of many characters. In this section, we think of values so far as just numbers and then try to introduce data.

A pair can be seen to represent perhaps a pair of points. But the notion that a pair can contain any value, including a pair, makes it far more interesting! That is, we could have a pair $a = (3, (4, 5))$, where $(4, 5)$ is a pair as well, then **first(second(a))** evaluates to 5.

Again, as with expressions, can make arbitrarily "interesting" structures. Indeed, one could represent expressions, which were recursive structures of arbitrary complexity, as pair structures. That is, '(3 + 4) + 2' could be coded into pairs as $(+((+, (3, 4)), 2))$. Here, one could 'evaluate' the structure using the rule that the operation in first position in the pair to the values that are yielded by the expressions of the two elements of the second pair. Again, the elements of the second pair represent expressions themselves, which we evaluate by applying the same rule, recursively.[12]

This is great, and let's continue.

First, we need to define another kind of value.

```
<value> = None
```

It is nothingness, emptiness. But don't feel sad, we make lists as follows.

A list is either empty or a first and a rest whose rest is a list.

The first and rest can be implemented using a pair, and the empty could be implemented as a None in, say, python.

Eureka! We now have a way to have a value represent an arbitrary sequence of data. Cool beans! We can even write programs that process a lot of data. For example, here is a function that adds up all the elements of a list.

```
def sum(lst):
   if lst == None:
      return 0
    else:
      return first(lst) + sum(rest(lst))
```

Here, we use the syntax in the expression, 'lst == None', which is true if the lst is empty.

Moreover, we call the function sum from the function sum. This seems circular, but since the list is smaller (that is, does not have the first element) we march toward a list with none, and can return from that, and from the previous call, and from the previous call, etc. So it all works out. Again, the pattern is recursive in that the function is defined in terms of itself. And moreover, the actual action of the function during the call follows the same pattern.

But this shows you why emptiness is so important. For this pattern to work, one must march toward empty. Recall, the natural numbers must start somewhere.

Notice, the function uses a conditional: in the empty case, we return, otherwise we do a recursive call on a smaller list. The first case is called a **base** case for this pattern.

Invariably, a student will point out that a list could be circular,for example, the rest of the list could be the list itself. In this case the program runs infinitely or would typically fail at some point in any implementation. Yes. The list must not be circular for this approach to work.

[12]Pairs as with expressions give rise to pictures of trees.

This seems simple. But it is powerful and a pattern that is oft used. Another closely related pattern is calling a function on each element of a list as follows.

```
def apply_f(lst, f):
   if lst == None:
      return 0
    else:
       f(first(lst))
       apply_f(f(rest(lst)))
```

Voila. Neato.

Exercise: Modify the code to return a list which contains the values returned in the sequence of function calls. Recall, that the shorthand we use for making a pair is $a = (x, y)$ where x and y are values. [13]

3.1.1 Making a pair: below the line.

Above, we skipped ahead and used the notion of a tuple, a type of value that is built into python and we will get to in more detail later. One could implement a tuple using just the essentials of values, expressions, bindings, control and functions.[14]

One could implement a pair as follows.

```
def make_pair(a,b):
   def dispatch(msg):
       if msg == 'first':
          return a
       else:
          return b
       return dispatch

 def first(pair):
     pair('first')

 def first(pair):
     pair('second')
```

This is a bit of an odd exercise. But the concept of "below the line" is a nice one in that it makes clear what is fundamental and has to be explicitly included in a programming language or could be built out of what is there. To be sure, there are many levels between the ideal model of a computer as processor and unit access memory and programming languages, and many levels

[13]This works in python. The pair is called a tuple.

[14]In some languages, for example, in C, functions are not by themselves values and function definition cannot be defined anywhere in the code. But this treatment, assumes that functions are "first class" and can be used as a value.

below even the idealized model and the transistor. Still it is enlightening to understand the process at this level.

In this case, the creation of the frame for the function call to make_pair creates a frame where the values assigned to the parameters a and b are stored for later access using the dispatch function that can find a and b in its environment diagram.

3.1.2 Mutability

Mutability is essential and a bit of a nightmare. In a sense, assigning a to 3 and then assigning it to 4 could be confusing, as "Wasn't it 3?".

For pairs, adding the power of changing the value of one of the elements is mutation. That is, given a pair $p = (3, 4)$, calling the function $set_first(p, 5)$ *mutates* the pair. The function $first(p)$ would then return 5 rather than 3.

This very simple notion allows for greater efficiency and for maintaining and updating data. For example, if the pair represents a person's name and a zipcode, one can update the zipcode for the person.

3.1.3 A Visualization: box-and-pointers

The value of understanding from their very definition is clear. An expression is a value or an two expressions with an operator. A pair is a pair. It is what it is. And we encourage you to focus heavily on that. It is basic dogma in this course that a list is a first and a rest or it is empty.

But common visualizations of structures can be useful in understanding and communicating and debugging. Moreover, mutation(and sharing[15]) can makes things very complicated. The box-and-pointer diagram is quite useful visualization to deal with subtleties.

We have seen already an expression can be written $(1+(2+3))$. A linked list can also be written as $(1, 2, 3)$, even though it consists of three pairs, $p1, p2, p3$, where $p1 = (1, p2)$, $p2 = (2, p3)$ and $p3 = (3, None)$. But it represents the sequence $(1, 2, 3)$ and thus is often written that way rather than belaboring the implementation.

But sometimes it is useful to see how it is represented visually. Particularly, when one mutates or shares or perhaps even has circular lists.

There is the notion of box and pointer diagrams that is illustrated in Python tutor that can be quite useful. For example, for the list, (1,2,3), we have the following:

3.1.4 Play around.

Use the interpreter: perhaps at code.cs61a.org. Maybe cut and paste some of the code above. Play around with a pair. Make the concept of a linked list, which is a pair whose second element is a list or empty. Write a function to find an element in your list.

[15] An example of sharing is a pair $p = (1, 2)$ and pairs $a = (p, 3)$ and $b = (p, 4)$. The pair p is shared by the pair a and b. Mutating p, changes the notion of a and b.

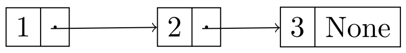

Figure 2: A box and pointer diagram of a list that consists of three pairs.

Put your code in python tutor and see what it draws. This is a box-and-pointer diagram.

3.2 Efficient Aggregates.

Python has a built-in way to have a sequence which is also called a list[16], which can be constructed in the following manner.

```
list(<exp1>,<exp2>,...)
[<exp1>,<exp2>,...]
```

It constructs a sequence of values. For a list bound to the name a, one can access element 2, using the expression, $a[2]$. They are 0-indexed, and thus $a[0]$ refers to the first element of the sequence.

They are interesting in that the time to access any element is considered to be constant, much like the model where one can access any memory location in one time unit. Compare this to the linked list that is formed from pairs, where accessing the nth element requires $O(n)$ time to traverse the list.[17]

While this power is interesting, we do believe that even here when solving many problems, a useful view is a list is a first element, in this case, $a[0]$, and the rest, $a[1:]$. The notation "1:" is called a slice indicates that this value is the sequence of elements in a starting at position 1. In general, one has the following syntax for generating slices.

```
<list>[<start>:<end>:<step>]
```

This will produce a subsequence of the original list consisting of elements at indices, `<start>`,`<start + step>`,..., where the final element is at index strictly less than `<end>`. One can have fun with it, e.g, $a[0::2]$ gives the even elements as the missing `<end>` value is assumed to be the length of the sequence. But these are details for you to play with on your own.

Assignment can be done as well.

```
<list>[<exp>] = <exp>
```

One can assign to slices, but it is best to not do so. That is discussed further in the ugly section.

[16]For old-timers, this is very bad. Normally, such things were called arrays to distinguish them from list. The efficient access property here is very nice, but one can get a long way with just the list abstraction.

[17]The $O(\cdot)$ notation is shorthand for the time scales with the function inside the notation. That is, $O(n)$ means the time for processing an n-length input grows linearly with n. $O(n^2)$ means it grows quadratically with n.

3.2.1 To copy or to not copy, that is the question.

Assignment to a list (as with assignment to a pair) raises a fundamental issue in programming with agregates. When do you make a copy?

In python $a[1:]$ gives you a slice which is a copy of the data in a. That is, it makes a new list containing all the elements in a from 1 onwards. Thus, executing $b = a[1:]$ and $b[0] = 4$, will not change the contents of a.

For a list made of pairs, one might have $b = second(a)$ which happens to be a pair which is not copied. Then changing the value in that will change the contents of a.

Making these choices is something that the programmer must pay attention to when thinking of what a program means and different choices are made due to considerations of semantics and efficiency.

3.2.2 Play around.

Again, with the interpreter, indeed, you should always be reading with the interpreter.

Make a list, a list of lists of things. Look at the representation in python tutor. Assign a value to an element of the list.

From here, do keep that interpreter open and type in expressions relevant to our discussion.

3.3 Dictionaries

Dictionaries build on the concept of an array or, in python, a list that one can access the ith element of a list in constant time.

The python list associates an index to a value. A dictionary generalizes this by associating a key of arbitrary type to a value.

We have already seen this idea, in the concept of a frame. That is, a variable name is bound to a value. One can look up that value by referencing the variable name.

This power is extremely useful and in python a dictionary can be created, accessed, and assigned as follows.

```
<variable> = {}
<variable>[<key>]
<variable>[<key>] = <value>
```

One application is in perhaps building a programming language itself as noted. One can have the keys be the variable names, and the values be the values associated with that variable name.

But, a more prosaic application may be to associate names with addresses or what have you. We note that in python there are a variety of ways to initalize and express a concept, e.g., one can specify a dictionary as {'satish':'berkeley,ca','biden':'washington,dc'}, where the key 'satish' has value 'berkeley,ca', and so on.

We leave the details of various expressions to later.

4 Functions and abstraction.

There are varieties of abstraction as we discussed in the introduction. For sorting, it may be useful to have a function to find the maximum element in a set. So, we abstract the action of finding the maximum value.

Another type of abstraction is on data, or what it means. For example, one thinks a fraction or equivalently a rational number as a pair of integers: the numerator and denominator. With our constructs, can now put that abstraction into code as follows:

```
def make_rational(num,den):
   d = gcd(num,den)
   return make_pair(num/d,den/d)

def num(rat):
   return first(rat)

def den(rat):
   return second(rat)

def add_rational(x,y):
    n_x, n_y = num(x),num(y)
    d_x, d_y = num(x),num(y)

    return make_rational(n_x* d_y + n_y *d_x, d_y * d_x)
```

This code models how we think of a rational: a pair of integers that correspond to the numerator and denominator. The last function is how we add rationals. Here, we assume that we have a function, gcd, that computes greatest common divisor of two numbers so we can reduce the fraction for the rational.

Notice in the add_rational function, we used the functions num and den rather than take the first and second element of a pair. This is a big idea for computer scientists. Indeed, had we used first, we would have called it an *data abstraction violation*. It's bad.

Why? For example, we could change our mind on the representation of two numbers and use a python list instead. If we used num and den (which are sometimes called accessors), we would not have to change the implementation of add_rational. This is simple enough here, but as one builds more complicated software, this notion of abstraction and obeying the lines drawn, that is, add_rational abstractly thinks of a rational as a numerator and denominator and only deals with the rational on that level. Its more clear and also good engineering design.

5 Objects and Classes

Its clear, a values has a "type". That is, there are integers, decimal numbers, and strings. With agregates, we have a pair which "contains" other values, including a pair itself.

There are certain builtin operations or functions which operate on the built-in types. And we can add functions whose arguments must be certain types to make sense. For example, the notion of **first** makes sense for a pair.

We also saw functional abstraction, where we can view a pair as a concept or type. For example, a pair of integers could be a rational number where the first number is the numerator and the second is the denominator. Then we can add functions that add, divide, or make fractions. This leads to having a set of functions in our program with names such as **add_rational**.

In contrast, for python, we have the notion that the operation '+' can be applied to integers or to decimal numbers or even to strings, the latter being that you obtain a new string which is the concatenation of the string values that the '+' symbol connects.

Indeed, one could have the notion that there is an add function for the type rational that simply does the "right thing" where it produces a rational which is the sum of the rationals given to the generic add function.

To be sure, this is relatively pedestrian. One could have a type called a person. And for some application, the concept of a person contains their name, address, and various other properties for the person, e.g., for medical records it may be their health history.

This type of abstraction is fundamental to our reasoning as humans and to programming.

This way to organize our thoughts is quite useful and is done using object systems in programs.

The idea is each value is an object with a type. We saw, integers and strings. Moreover, an object has an associated set of functions, and variables. In python the variables are called attributes. The type is called the *class* of the variable and the class itself is where the functions are defined and may have attributes or class variables that themselves are values..

At this point, we will just proceed with how one *defines* classes in python, and *instantiates* objects of these classes. The definition and instantiation is somewhat analogous to function definition and calling a function and indeed classes can be built with functions as we will see in our "below the line" discussion.

To sum up, an object has a type or belongs to a class. Classes contain associated functions and data for each object. And they can also have variables that are common to all objects. Finally, classes can be related. The most basic form, is that a class can inherit from another class, for example, a chef is a fancy type of cook. We will proceed to how this is expressed in python.

5.1 Pythonics of classes and objects.

An object can have associated variables and which can be accessed or assigned using the dot notation below:

```
<objname>.<attribute_name>

<objname>.<attibutue_name> = <expression>
```

By default, any user defined object can have attributes and they can be assigned. This is restricted for builtin types and can be restricted for user defined types.

A user can define a class as follows:

```
class <cname>:

   <class body>
```

The class body consists of function definitions or class variables. Some functions are special and have a naming convention where the begin and end with "__". For example, the "__init__" function is typically defined as follows:

```
class <cname>:

      def __init__(<params>):
          <body>
```

An example is now possible.

```
class Rational:

     def __init__(self,num, den):
         d = Rational.gcd(num,den)
         self.num = num/d
         self.den = den/d

     def gcd(n,d):
         if n > d:
             n,d = d,n
         if n == 0:
            return d
         return gcd(d%n,n)

     def numerator(self):
         self.num
```

```
    def __str__(self):
        str(self.num) + '/' + str(self.den)



r = Rational(6,8)

print ("Numerator", r.numerator())

s = r.__str__()

print ("Rational:", r)
```

When the expression Rational(6,8) is evaluated, python first makes an object with type Rational, and then calls the __init__ where self is the first argument of the init function and num is assigned to 6 and den is assigned to 8.

More generally, consider the expression of the form:

```
<cname>(<params>)
```

An object with type `<cname>` is made, and then its __init__ function is called with self bound to the object, and the rest of the paramets in the initializer bound to the `<params>`.

In the example, one sees that one can define other functions, in this case, **gcd**. The value is that the function **gcd** is local to these objects. The way it is acccessed in this case is by using the class name which itself refers to the class object whose functions can be acessed this way. This function does not take the object as an argument and is called a class function.

The expression **r.numerator()**, calls the function defined as numerator *with* the first argument, self, being the object r itself. This way of calling a function, using the dot notation on the object itself, generally views the function as a **method**. It can be contrasted with calling the function using the dot notation with the class name. Indeed, one can assess the function through the class name, as Rational.numerator(r), where the object **r** is explicitly passed to the function as the formal parameter, self.

More generally, we have the following patterns:

```
<objectname>.<attribute>
<objectname>.<method_name>(<params>)
<cname>.<function_name>(<params>)
```

The first accesses an attribute, the second is a call expression to the method where the first parameter of the method is asssigned to the object and `<params>` is used to assign the remainder of the parameters, and finally the last expression is how one calls methods defined in a class using the class name. We note that function_name and method_name may be the same name, but the default behavior for a method call is that the object is passed implicitly (as it is available through the `<objectname>`) and the the class function case all parameters must be passed explicitly.

Finally, we discuss the method __str__ in rational. This is just another method, but it is used in python by the str function to produce a string representation of the object. It is, for example, implicitly called by the print function on any object. Thus, it, like, the __init__ function is called by default.

A related function called __repr__ is also implicitly used by python to produce a string representation, in particular, it is used by the interpreter when displaying the value of the object. But we leave that discussion to later

5.1.1 Inheritance: in brief.

Briefly, once one has built a class, it is often nice to "resuse" those methods for related classes. Thus, one has inheritance.

In general, we have the following form:

```
<cname1>(<cname1>):

    <cbody>
```

This makes a new class name, `<cname1>`, that inherits all the methods and class variables defined by class name `<cname2>`.

One can override the "parent" classes methods by writing a method of the same name in `<cbody>`.

Indeed, one could inherit from more than one class, and there are ways to access the parent method from a child method explicity. But, for now, we leave it at this.

5.1.2 Play around.

Play around in the interpreter. Make a class, copy the rational code, add your own method.

5.1.3 Below the line using dictionaries.

One can implement much of the notion of an object by using a dictionary.

That is, a rational could simply be a dictionary with keys, 'num' and 'den'. Since functions are first class objects the methods can also be stored in the dictionary with a key corresponding to the method name and value being the function. One loses some syntactic convenciences provided by the dot notation and also the automatic passing of an instance to methods.

5.2 Below the line using functions.

Conceptually, a frame is a dictionary in that it contains bindings between names and values. Thus, one can use a function call to create an instance of a class which is implemented as a function whose parent is the frame. That is one can write a function, **make_rational** could return a function, where passing 'num' to this function returns the numerator and passing 'den' to this function returns the denominator. One can also invoke methods using the function to call various functions defined inside of the **make_rational** body.

In some sense, this is how **make_pair** worked before. It created, in a sense, an object of type pair.

This illustrates that in some fundamental sense, classes need not be directly implemented in a programming language. On the other hand, the fact that even builtin types like integers and strings exist suggests that types are critical and perhaps that truth should be exposed right away.

In any case, the syntactic convenience of explicitly defining classes is extremely important and thus included in many programming languages.

6 A Fundamental Pattern: Iteration.

Iteration is basically keep doing something over and over until you decide to stop. The most basic construct is a **while** loop. That is, you have a piece of code you repeatedly execute as long a specified condition holds. One could express this recursively, but it can be convenient, efficient, and easier to understand the iterative approach (or vice versa). For example, if one wishes to add up a set of numbers one can maintaining a running **total** variable and iterate over the set of numbers adding each to the running total.[18]

There is not so much more conceptually to say, but there are some interesting ways to express and abstract this concept. Python is fine for illustrating, and has a reasonably sophisticaed abstraction, so we get into that.

A common pattern in iteration is iterating over a sequence, such as a list. Python abstracts this notion in an interesting manner in that the sequence can be generated as one goes.

6.1 Pythonics: iteration.

A while construct in python is as follows:

```
while <condition>:
    <body>
```

It does it what it says. Not too much to say about it.

Python also has another construct which is a for loop.

[18]A recursive expression is to remove a number from a set, add up all the remaining elements and then return the sum of that and the removed element.

```
for <variable> in <iterable>:
    <body>
```

This loops over a sequence of objects in iterable. For example, a list is iterable and in successive executions of the body the variable is assigned to a successive elements of the list. The abstraction of a sequence as an iterator is a bit fancy, but we can go there, as it does give some insight into the power of programming. But it is fine to skip for now as well.

6.1.1 Iterators and Iterables.

From the previous section, an iterable is something that can be used in a for loop. In fact, the interface to an iterable is through the function, **iter** which produces an object which responds the function **next** by returning the next item in the sequence until it runs out.

That is, for a list, $i = iter(lst)$ assigns to i an object. Repeatedly calling $\textbf{next}(i)$ returns values corresponding to the elements of the list referred to by the variable, **lst** until there are no more, in which case, something called an *exception* happens, the details of which will be covered later.

The for loop uses this mechanism generically. Of course, perhaps the simplest iteration is one over a subset of the natural numbers. In python this is done using the construct *range*.

```
range(<start>,<end>,<step>)
```

The range returns a iterable which corresponds to the sequence start,start+step,start + 2*step,..., upto the final value which is strictly below end. One can simply do range(3,6) and the stepsize of 1 is assumed, or range(6) and the start value of 0 is assumed.

Finally, one can build one's own iterable in python. A common way is to use the function map or something called a generator. We will discuss these later.