Unit-Test-based programming

Miguel Á. Lechón*

April 1, 2014

Adding is when two plus two equals four.
POPULAR SAYING

Abstract

This article introduces a new subparadigm of Declarative Programming, in which code is generated from a description of its intended behavior, specified through unit tests. Yes, for realsies[1].

Some sensitive readers may be taken aback by the conceptual flawlessness as well as by the unbeatable simplicity of the method exposed here. Please, stop reading now if you are prone to emotional instability in the face of plain, unadulterated beauty.

1 Introduction

Programmers automate stuff. Ambitious programmers dream of automating the automation of stuff. Many have tried; all have failed. Until today.

In this article I will review how functions can be represented using non-botanical trees[1] and how non-botanical trees can be enumerated[2], leading to the fact that functions themselves can be enumerated in order of increasing complexity.

Taking advantage of this result, it is possible to automatically generate functions that satisfy arbitrary sets of restrictions, posed as unit tests. By construction, these functions fulfill several desirable properties, such as minimal Kolmogorov Complexity and maximal Occam's Razority.

2 Motivation

This year marks the sixtieth anniversary of the birth of the field of Genetic Programming. While successful as a pastime and as a mechanism to spin Mendel in his grave, its promises of automatic function generation have failed to materialize.

Some researchers venture that exploration on non-linear genotype-phenotype mappings will lead the field to a second blossoming, while other, perhaps more rational, researchers realize the inadequacy of random

*e-mail: miguel.lechon+UTBP@gmail.com
[1] https://github.com/debiatan/utbp

recombination as an exploration technique. As Ernst and Sullivan famously stated in a classic opinion piece[3] on the uses of stochastic methods in the healthcare industry,

> Parents are strangely reassured when the predicted outcome of a critical medical intervention is measured in units other than fractional children alive.

2.1 Motivating example

Let us imagine a small language consisting of a handful of primitives[2] that operate on natural numbers and lists of natural numbers. Among them, we find to our disposal:

car: Given a list, returns its head.

cdr: Given a list, returns its tail.

succ: Returns the successor of a number.

if: Ternary conditional. It evaluates the truth value associated to its first argument and, if found true, it returns the result of evaluating its second argument. Otherwise, it returns the evaluation of its third argument.

Let us further imagine that we intend to program the non-primitive function **length**, that returns the number of elements of a list. A possible definition in the Unit-Test-based programming (UTBP) paradigm would be:

```
@UTBP
def length(l):
    """
    length(()) == 0
    length((2, 2, 2, 2, 2)) == 5
    """
```

[2]for more information, refer to section 3.3.

This version of UTBP is built on top of Python, as a library, hence the syntax of the example.

Notice the `@UTPB` Python decorator, indicating that this function belongs to the Unit-Test-based elite. Notice also how all function logic has been replaced by a documentation string listing assertions for the function to satisfy. Calling this function with arguments of different lengths dispels most doubts on its correctness, but skeptical users may extract some comfort from the examination of its underlying implementation:

```
>>> print(length)

    define
    ├── length
    └── lambda
        ├── l
        └── if
            ├── l
            ├── succ
            │   └── length
            │       └── cdr
            │           └── l
            └── 0
```

Readers familiar with LISP-like languages will recognize this as a recursive definition of **length** in which parentheses have been dumped in favor of a tree-like graphical representation. This code states in unambiguous terms that **length** of the empty list is 0 and that **length** of any other list is 1 plus the **length** of its tail.

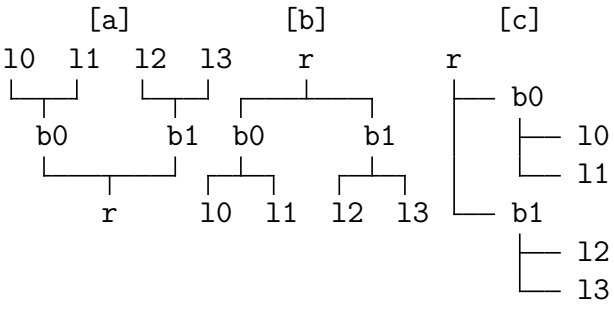
The next section provides a more formal treatment of the foundations of the UTBP paradigm.

3 Materials and methods

UTBP is aimed at programmers who care about results and eschew implementation details. If you count yourself among their numbers, you may skip this section.

3.1 NBTSASWs

Let us consider a tree with root `r` and two branches `b0` and `b1`, each of them with two leaves (`l0` and `l1`, `l2` and `l3`). There are three main ways of portraying this tree:

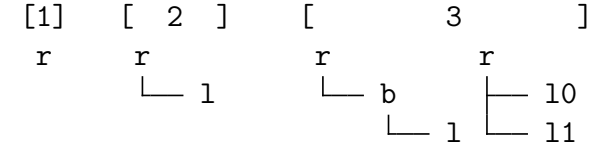


Tree `a` is drawn in a *botanical* fashion. Tree `b` is clearly of a *non-botanical* kind, since its root points upwards. Tree `c` is a *non-botanical tree struggling against strong winds*.

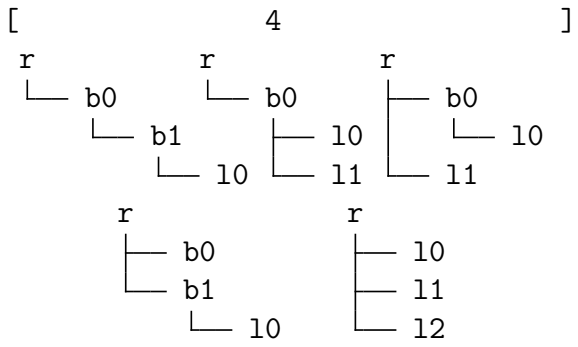
Throughout the rest of the article, I will use NBTSASWs exclusively because their aspect ratio suits two-column layouts best.

3.2 Tree enumeration

There is one tree of size one, one tree of size two and two trees of size three.



There are five trees of size four.



The cardinalities for the rest of the sizes are given by the sequence of Catalan numbers, just like all problem in combinatorics[4]. As a Catalan-born, I am very proud of that sequence. It is a bit upsetting, however, that Eugène C. Catalan turned out to be Belgian.

3.3 Primitive types, operations and constants

The trees listed so far are just templates and the labels on their nodes are but placeholders. For a tree to represent a function, each of its nodes has to be associated with either an operation of cardinality matching the number of descendants of that node or with a constant, in case the node lacks progeny.

3.3.1 Types

The two only necessary types for this article are nested lists and natural (non-negative) numbers. Natural numbers are really not that necessary, but I will spare them if only to avoid writing *23* as:

(())

3.3.2 Arithmetic functions

The primitive arithmetic operations are **succ** (successor) and **pred** (predecessor). They are preferred over the more common nomenclature *increment* and *decrement*, since this latter alternative implies the use of variables and, once you start relying on mutation, you may as well end statements with semicolons;

succ is a unary operation that returns the natural number immediately following its argument.

```
succ  -> 1      succ  -> 2
└── 0            └── succ
                     └── 0
```

pred is also unary and returns the natural number immediately preceding its argument. **pred** of 0 is undefined because accepting the mind-bending abstraction of negative quantities ultimately leads to irrational and complex thoughts.

```
pred  -> 2      pred  -> 0
└── 3            └── succ
                     └── 0
```

3.3.3 List functions

The three UTBP functions that operate on lists are LISP's classical **cons**, **car** and **cdr**.

cons takes two arguments. The first one can be either a natural number or a list, while the second one has to be a list. It returns a list whose head is equal to the first argument as whose tail matches the second argument.

```
cons  -> (10 30)   cons  -> ((5) 23)
├── 10             ├── (5)
└── (30)           └── (23)
```

car returns the head of the list provided as argument. **car** of **()** is undefined.

```
car  -> 30      car  -> 5
└── (30)        └── (5 23)
```

cdr returns the tail of the list provided as argument.

```
cdr  -> ()      cdr  -> (23)
└── (30)        └── (5 23)
```

3.3.4 Special forms

The special form **if** takes three arguments of any type and evaluates the truth value associated with the first one. If it happens to be true, it returns the result of evaluating its second argument. Otherwise, it returns the outcome of evaluating its third argument.

```
if  -> 42        if  -> 23
├── succ         ├── 0
│   └── 0        ├── 42
├── 42           └── 23
└── 23
```

3.3.5 Constants

UTBP's two constants are **0** and **()**. They are also the only values whose associated truth value is *false*.

3.3.6 Recursion

Expressing iteration using trees, even if they are NBTSASWs, feels unnatural. Iteration also requires variables. Readers are welcome to do whatever they please in the privacy of their homes, but the public use of variables carries a death sentence in many countries.

3.4 Functions as trees

As academics, we are supposed to derive all possible interpretations and connotations from the methods section by ourselves, but this article is all about providing examples for automatic inference, thus:

What is a proper tree representation of the function that *takes a list* **`l`** *and an index* **`i`** *as arguments and returns the element in the* **`i`***th position of* **`l`**?

```
0       define
1       ├── index
2       └── lambda
3           ├── l i
4           └── if
5               ├── i
6               ├── index
7               │   ├── cdr
8               │   │   └── l
9               │   └── pred
10              │       └── i
11              └── car
12                  └── l
```

Lines 0–3 specify the name of the function (**`index`**) and of its arguments (**`l`** and **`i`**). The **`if`** clause (line 5) tests whether **`i`** is 0 and if so, makes sure that the zeroth element of the list is returned (line 11). Otherwise, **`index`** is called recursively with arguments **`cdr l`** (tail of l) and **`pred i`** (i+1).

It is the responsibility of the caller to make sure that the magnitude of argument **`i`** does not exceed the length of list **`l`**. Fortunately for the caller, we already reviewed an automatic implementation of the length of a list in section 2.1.

4 Properties

4.1 Seamless integration with Python

This article presents a new programming paradigm. Instead of introducing it in an abstract manner, I have implemented it as an extension of Python, a multiparadigm language with a large user base (eighth most popular language at TIOBE at the time of writing[6]). In this way, I hope to decrease the chance of my effort of many nights ending in a drawer, collecting drawer smell.

We have already established how to define the function that computes the **`length`** of a list under the UTBP paradigm, and we have also examined the NBTSASW associated to the **`index`** function (section 3.4), but we still need to provide its UTBP definition:

```
@UTBP
def index(l, a):
    """
    index(((4, 7), 8), 0) == (4, 7)
    index((4, 7, 8), 1) == 7
    index((4, 7, 8), 2) == 8
    """
```

The attentive reader will notice how these unit tests are carefully crafted to:

- Return different indices of the array. If we always returned the first one, the easiest implementation would be **`car l`**.

- Use nested lists. That way, we let the UTBP framework infer the types of inputs and outputs so as to provide a sufficiently general implementation.

4.2 Low barrier to entry

Python code is often described as *executable pseudocode*, so one could argue that its users are not programmers that code, but *pseudo-programmers* that *pseudocode*.

It is a trivial exercise to build a system that generates code from preconditions and post-conditions inside a well-behaved language such as Scala[5], but it is a much harder task to build a tool that does not rely on its users knowing, or even caring, about preconditions or fancy languages. Today's world is in need of dumber tools for careless people.

4.3 Conciseness

Novice programmers are usually required to specify the desired behavior of a target function three times: first by documenting it, then by providing examples of its use and finally by actually implementing it. UTBP-style definitions prey on that redundancy.

4.4 Guaranteed correctness

UTBP's search routine will only stop evaluating functions once it finds one that passes all provided unit tests. If a know-it-all were to try the following definition:

```
@UTBP
def impossible(a):
    """
    impossible(0) == 0
    impossible(1) == 0
    """
```

the UTBP routine would never stop, avoiding the generation of an incorrect answer.

4.5 Minimal Kolmogorov Complexity

The solutions generated under the UTBP paradigm possess the minimum amount of nodes necessary to fulfill a given specification using a restricted set of operations and constants, thus making them minimal under the Kolmogorov Complexity measure.

Some readers will regard this use of Kolmogorov Complexity as a *bastardization* of the strict meaning of the concept, but I doubt Kolmogorov would complain, being the son of an unmarried woman[3] himself.

4.6 Maximal Occam's Razority

Routines that achieve a given task with a guaranteed minimal description length are maximally parsimonious and thus preferable.

4.7 Avoidance of Terminator-like Judgment Days

UTBP does not make computers more intelligent; it simply accentuates their stubbornness. If an unfortunate turn of events leads an instance of the UTBP search engine to stumble on a homicidal routine, the built-in language barrier (code generated is executed inside a LISP-like virtual machine) will prove insurmountable.

Let me be clear on this, your computer will probably plot elaborate plans to end your life, but it will not be able to act on them.

[3] https://en.wikipedia.org/wiki/Kolmogorov#Early_life

4.8 Unparalleled performance

The implementation of the UTBP paradigm presented in this article is the first of its kind. No benchmarking is necessary to show that its speed is beyond comparison.

4.9 Unparalleled performance

The UTBP search algorithm is fully parallelizable, but it is not parallelized.

5 Examples

5.1 Boolean functions

5.1.1 Unary Boolean functions

Unary and binary Boolean functions are easy to describe using unit tests, since one can enumerate all their possible inputs. **not** is described as:

```
@UTBP
def logical_not(a):
    """
    logical_not(1) == 0
    logical_not(0) == 1
    """
```

Which generates the following code:

```
define
├── logical_not
└── lambda
    ├── a
    └── if
        ├── a
        ├── 0
        └── 1
```

5.1.2 Binary Boolean functions

I will skip the obvious unit-test Python descriptions for **or**, **xor**, **and** and **nand**, but I will list their outcome for the reader's enjoyment:

```
define
├── logical_or
└── lambda
    ├── a b
    └── if
        ├── a
        ├── a
        └── b
```

```
define
├── logical_xor
└── lambda
    ├── a b
    └── if
        ├── a
        ├── logical_not
        │   └── b
        └── b
```

```
define
├── logical_and
└── lambda
    ├── a b
    └── if
        ├── a
        ├── b
        └── 0
```

```
define
├── logical_nand
└── lambda
    ├── a b
    └── logical_not
        └── logical_and
            ├── a
            └── b
```

The definitions of **xor** and **nand** show that the function search engine makes use of previous UTBP definitions to provide simpler expressions than those possible using only the initial set of operations.

5.1.3 N-ary Boolean functions

A classic in the Genetic Programming literature is the Boolean Parity Problem. In it, the target function takes a list of N Boolean digits and decides if the number of ones it contains is odd. An initial attempt using the UTBP framework would be:

```
@UTBP
def logical_parity(l):
    """
    logical_parity((0,)) == 0
    logical_parity((1,)) == 1
    logical_parity((0, 0)) == 0
    logical_parity((0, 1)) == 1
    """
```

And the shortest routine satisfying it:

```
define
├── logical_parity
└── lambda
    ├── l
    └── car
        └── if
            ├── cdr
            │   └── l
            ├── cdr
            │   └── l
            └── l
```

This code assumes input lists of up to two elements, which makes sense from the low vantage point of the computer, but ends up returning a list instead of Booleans when that condition is violated.

Enumerating the solutions to a handful of longer examples

```
    """
    ...
    logical_parity((0, 0, 0)) == 0
    logical_parity((0, 0, 1)) == 1
    logical_parity((0, 1, 1)) == 0
    """
```

takes care of the problem:

```
define
├── logical_parity
└── lambda
    ├── l
    └── if
        ├── l
        ├── logical_xor
        │   ├── logical_parity
        │   │   └── cdr
        │   │       └── l
        │   └── car
        │       └── l
        └── 0
```

This code computes the parity of the tail and **xor**s it with the head. Spotless.

5.2 Arithmetic functions

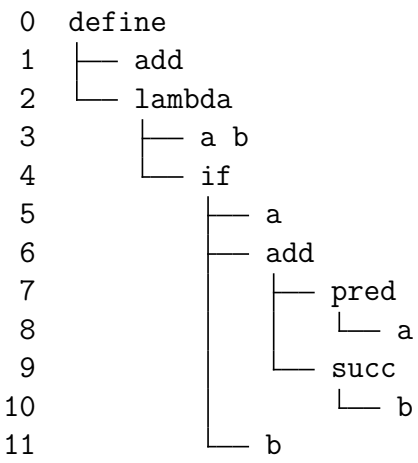
Hermann Grassmann showed in the 1860s that many arithmetic operations can be derived from the *successor* operation. UTBP can easily replicate some of his findings without giving much thought to the task.

5.2.1 Addition

Back in section 3.3.2, I introduced UTBP's two arithmetic primitives: **succ** and **pred**. From them we can derive addition by writing:

```
@UTBP
def add(a, b):
    """
    add(2, 2) == 4
    add(3, 3) == 6
    """
```

The second assertion is there to prevent UTBP from thinking that adding two numbers means returning always the value 4 (which otherwise would be a more parsimonious interpretation). The resulting code reads:



The mathematically inclined reader will see that the function is equivalent to:

`11` : $add(0, b) = b$

`6-10` : $add(a, b) = add(a - 1, b + 1)$

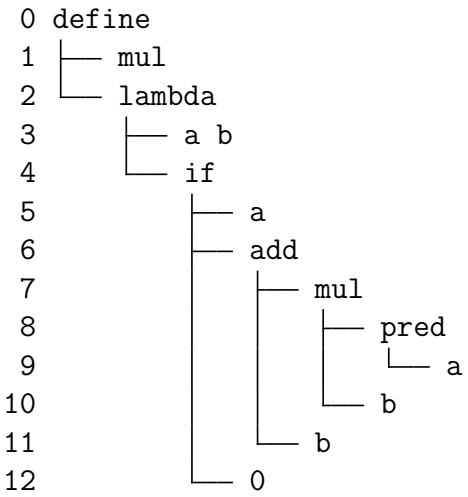
The complexity of this function grows linearly with the size of a in both execution time and memory consumption, which is quite reasonable for the natural numbers people encounter on everyday tasks.

5.2.2 Multiplication

Defining multiplication without first defining addition is very troubling for both humans and UTBP, but once the more basic operation is in place, these two assertions suffice:

```
@UTBP
def mul(a, b):
    """
    mul(2, 2) == 4
    mul(3, 5) == 15
    """
```

This definition leads to:



Which again has a clear mathematical interpretation:

`12` : $mul(0, b) = 0$

`6-11` : $mul(a, b) = add(mul(a - 1, b), b)$

Execution time and memory consumption for this multiplication function grows quadratically with the magnitude of the result. I find this feature useful, since it reminds me of my excesses when I compute my daily caloric intake.

5.2.3 Exponentiation

I am sure the reader gets the idea by now.

5.3 List functions

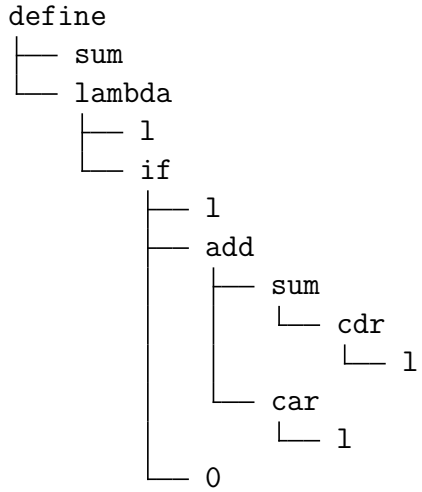
We have already reviewed the definitions of three functions that operate on lists: **length**, **index** and **logical_parity**. Now I provide one more example that requires the use of list primitives as well as of a non-primitive arithmetic function.

5.3.1 Summation

A possible UTBP definition of summation is:

```
@UTBP
def sum(l):
    """
    sum((2, 2)) == 4
    sum((3, 3, 3)) == 9
    """
```

And the function that the two previous assertions generate is correct:



Needless to say, programmers are discouraged from examining the code associated with UTBP definitions. It is not your code that defines you, but your actions.

6 Final remarks

In this article I have presented a new, simpler approach to programming. I am fully aware that my target audience will probably never read it, so I have decided to provide an alternative, more pragmatic video presentation online. It can be found here:

`http://blog.debiatan.net/utbp.html`

References

[1] John McCarthy, *Recursive functions of symbolic expressions and their computation by machine*. Communications of the ACM 3(4):184-195

[2] Eric Lippert, *Every tree there is*[4] 2010.

[3] Philip Ernst and Richard Sullivan, *Probabilistic pediatrics – Trusting your progeny to Monte Carlo.* Journal of Proper Parenthood, May 1998, 79-92

[4] Richard P. Stanley, *Enumerative Combinatorics, Volume 2*, June 2001

[5] E. Kneuss, V. Kuncak, I. Kuraj and P. Suter, *Synthesis Modulo Recursive Functions*, Acm Sigplan Notices, vol. 48, num. 10, p. 407-426, 2013

[6] *TIOBE index*[5], February 2014

[4] `http://blogs.msdn.com/b/ericlippert/archive/2010/04/22/every-tree-there-is.aspx`

[5] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`