The Education Column

BY

Juraj Hromkovič and Dennis Komm

ETH Zürich, Switzerland and PH Graubünden, Chur, Switzerland
juraj.hromkovic@inf.ethz.ch and dennis.komm@phgr.ch

A Minimal Instruction Set for Education

Tobias Kohn
University of Utrecht
t.kohn@uu.nl

Abstract

Computer science education is often caught between the desire to teach algorithmics and the necessity to build a solid basis of programming to properly implement algorithms. We thus raise the question of how much programming we need before we can start tackling interesting computational problems and demonstrate how important computational concepts emerge almost naturally with a minimal set of programming constructs needed.

1 Introduction

The quest of a minimal instruction set is an old and solved problem. How many instructions does a computing system need in order to be Turing complete? In a nutshell, combine increment/decrement with a while-loop and you are good to go (with numerous alternatives existing, of course, some rather surprising [1, 3]).

Let us consider the same question from a different perspective, though. Picture a group of students with virtually no prior experience in programming. Your objective is to enable them to solve 'interesting' computational problems. How interesting? Think, e.g., 'graph algorithms' or 'Project Euler' [4] kind of problems. Your target audience starts from scratch and you want to minimise the time needed for teaching programming and get to solve problems as quickly as possible.

In this setting the set of minimal instructions is no longer dictated by pure logic and a desire for Turing completeness. Rather, psychological factors quickly dominate the discussion. For instance, as trained professionals we would hardly seek to include multiplication into a minimal computing instruction set, whereas we can safely include it into our education setting because students are already highly familiar with it and the necessary amount of teaching is practically null.

On the flip side, we find that the modulo or remainder operation is surprisingly expensive. Even though division and computing the remainder might even be one and the same underlying program running on an actual machine, the modulo operation is significantly harder to understand, apply and hence teach.

So, can we discuss core concepts of computer science with merely a handful of well chosen, 'easily teachable' instructions and programming constructs?

Much of what follows is highly reminiscent of functional programming, although it is not. Our contemplation is based on imperative programming with Python. We embrace its lists as mutable data structures and use iteration instead of recursion. The similarity to functional programming might arise due to two factors: a limited and cautious use of variables and avoidance of list indices.

2 From Code to Data and Back Again

We usually start our introductory programming courses with turtle graphics. The virtual turtle robot is a simple and concrete device that leaves a trace so as to draw figures. Both the set of its capabilities as well as its purpose are thus immediately obvious and in stark contrast to full-blown computing devices with their unbounded complexities.

The set of initial commands consists essentially of `forward(s)` and `back(s)` for movement, and `left(a)` and `right(a)` for turning (on the spot). When talking about finding a minimal set of instructions half of these four instructions are evidently superfluous—even more so when considering that `left(a)` can equally be written as `right(-a)`. In fact, a single unified command that moves the turtle one step forward and turns it by a given angle might suffice.

This example neatly demonstrates how 'engineering minimalism' differs from 'educational minimalism'. The commands `left(a)` and `right(a)` express the same concept and, moreover, directly reflect concepts from the students' everyday experience. Reducing this to a single `turn()` command would rather increase the difficulty as the student would also have to consider which (arbitrary) direction is meant by a 'positive' angle.

The idea of a single unified command comes up later in the course when we introduce for-loops. While for-loops lend themselves to a wide variety of applications, one that stands out is the representation of figures (e.g., draw by the turtle) through data. The introduction of the turtle naturally leads to programs that build intricate figures out of long sequences of instructions. By means of a for-loop and a list, we can now abstract from those long sequences of instructions and retain the data part of them only. The only problem is that the sequences of instructions alternate between movement and rotation. As a first step we therefore have to keep one of these fixed, i.e., stipulate that all rotations are replaced by a left-turn by 45° with zero-movement interspersed where necessary, or, alternatively, keep all forward instructions to ten steps, say (Program 1).

Luckily, Python allows us to create lists of tuples very easily so that we can include an argument for both the forward motion and the angle of rotation. Nonethe-

Program 1 Two ways of drawing a square using a simple for-loop.

```
for a in [90, 90, 90, 90]:
    forward(10)
    left(a)
```

```
for s in [10, 0, 10, ..., 0]:
    forward(s)
    left(45)
```

less, from an abstract point of view, we end up with a 'single instruction' (or a single parametrised action if you prefer), something like `move_and_turn(s, a)`. Alternatively, you might also choose to use coordinates along with the instruction `setpos(x, y)` as the basis for all drawings.

Hence, the step from an instruction- to a data-based format brings us back to the necessity of having a single instruction and therefore considering an extremely minimal set of instructions that suffice to draw as many figures as possible. This example also illustrates the power that comes from single-instruction machines. The extremely regular interface allows for the introduction of a new level of abstraction.

As a perhaps final step along this ladder we might wish to differentiate further between possible actions. So, let us move away from the idea of a single instruction that churns through a list of data and enrich the data with (textual) hints as to which action to perform.

For the sake of simplicity, let us stick with the commands we have already introduced. A minimalistic program might then look as shown in Program 2. If you look closely you will discover that we have come full circle here. The 'data' in the list we are processing is in fact program code in what looks very much like a dialect of Lisp or Logo [5].

Program 2 A simple interpreter for a Logo-like language.

```
for (cmd, arg) in [('fd', 50), ('rt', 90), ('fd', 20)]:
    if cmd == 'lt':
        left(arg)
    if cmd == 'rt':
        right(arg)
    if cmd == 'fd':
        forward(arg)
```

So what did we gain if we end up still writing our drawings as program code, only now with the additional overhead of the for-loop that executes the instructions? We have stumbled on one of the most fundamental ideas in computer science: the 'equivalence' of data and code. Replace the textual instructions above with numeric constants and you are but a small step away from Gödel numbers.

To put it differently, at this stage our students have written their first interpreter. Their programs no longer draw specific pictures, but are in principle capable of

running any other program that draws a picture. Although this is far away from Turing completeness and lacks most attributes of an actual computer, we nonetheless have just introduced the concept of a *universal machine*.

3 Iterating Over Lists

Our journey from sequences of instructions to data processing above was made possible by the use of lists and for-loops.[1] It is quite natural, of course, that abstracting from code to data meant that a sequence of instructions turns into a sequence of data. Moreover, Python's lists are very versatile data structures that can take the role of arrays, lists, stacks, sets or even maps. Let us therefore spend some time taking a closer look at Python's lists.

True to the overall topic of this article we argue that a set of three list operations, namely *iteration through a list*, *appending an element to a list* and *testing for inclusion* (i.e., whether a list contains a specific element) is sufficient for tackling a range of interesting problems. Most importantly, all three of these basic operations are conceptually simple enough to be integrated into teaching relatively early on.

The choice to use for-loops and lists comes at a price, though. The resulting loops are clearly bounded and we forgo Turing completeness (i.e., we only have primitive recursion and cannot implement partial recursive functions). On the other hand, considering that educational problem instances are typically quite small and by choosing sufficiently large numbers, we still achieve a working approximation to Turing completeness.

Lists in Python. Lists in Python are implemented as arrays that can grow in size. Accessing elements (both retrieval and modification) are thus in $O(1)$ and appending elements is in amortised $O(1)$ [6]. Using for-loops together with 'append' allows us to easily implement 'map' and 'filter' as shown in Program 3. Naturally, writing 'reduce' in Python is not much more difficult and follows the same pattern as 'map' as shown in Program 4.

There is direct syntactic support for checking whether an element occurs in a list (which is performed through a linear $O(n)$ search). Although this could also be done with a 'reduce' pattern, we include it in our 'instruction set' as it allows us to conveniently use lists like sets with minimal teaching effort.

Based on the basic elements of 'filter', 'map' and 'reduce' we can implement functions like 'pop' and 'cons' (Program 5) known from functional programming.

[1] NB: Python's for-loop corresponds to 'for each' loops in other languages and does not have the generality of for-loops in C, say.

Program 3 Implementations of 'filter' and 'map' in Python.

```
def filter(p, lst):
    mod_lst = []
    for i in lst:
        if p(i):
            mod_lst.append(i)
    return mod_lst
```

```
def map(f, lst):
    mod_lst = []
    for i in lst:
        mod_lst.append( f(i) )
    return mod_lst
```

Program 4 A generic implementation of 'reduce' or 'fold' and a more concrete instance of 'sum'.

```
def reduce(f, lst, init = 0):
    acc = init
    for i in lst:
        acc = f(acc, i)
    return acc
```

```
def sum(lst):
    acc = 0
    for i in lst:
        acc += i
    return acc
```

Variables and conditionals. Apart from the three list processing primitives introduced above we also make use of variables and conditional execution, thereby adding two further concepts. Furthermore, the code examples also use function definition and the return-statement, but those are not necessarily needed when using and implementing the presented concepts with students.

The most problematic aspect of these constructs is variable assignment, in particular updating a variable's value. To mitigate this issue we restrict variable modification to a small set of operations such as `acc += x` or `acc *= 2` etc.

Instead of 'append' we could also have gone for list concatenation along the lines of `list = list + [item]` or its short form `list += [item]`. At first glance this seems not only to be more powerful, but also easier as you do not need a special function, but rely on the concept of 'adding' instead. However, there are some subtle rules to consider for variable assignments in Python in terms of scoping. Python infers the scope of variables from context, unless explicitly de-

Program 5 The 'pop' function to split a list into its head and tail is a combination of 'filter' and 'reduce' from above whereas 'cons' follows a 'reduce' pattern.

```
def pop(lst):
    head = None
    tail = []
    for item in lst:
        if head == None:
            head = item
        else:
            tail.append(item)
    return head, tail
```

```
def cons(head, tail):
    result = [ head ]
    for i in tail:
        result.append(i)
    return result
```

clared as global, which adds a large and complex topic to the teaching curriculum. In short, using 'append' avoids some common pitfalls, helps us reduce the use of variable (re)assignments and is therefore in line with our aim to strive for a minimum set of instructions from an educational perspective.

Implementing Dijkstra's algorithm. As a proof of concept we present an implementation of Dijkstra's algorithm for finding the shortest path in an (undirected) graph in Program 6. The graph itself is defined close to its mathematical formulation with two lists representing vertices and edges. However, the actual vertex coordinates are used in the edge set, too, so as to avoid the use of indices.

Program 6 An implementation of Dijkstra's algorithm in Python.

```
vertices = [ (0, 0), (10, -2), (1, 15), (10, 10), ... ]
edges = [ ( 7, (0, 0), (10, -2)), (14, (0, 0), (1, 15)), ... ]
visited = [(0, 0)]
visited_dists = [((0, 0), 0)]

def dijkstra():
    for k in range(1, 50):
        for (anchor, cur_dist) in visited_dists:
            for (w, p1, p2) in edges:
                if cur_dist + w == k:
                    if anchor == p1 and p2 not in visited:
                        (x, y) = p2
                        visited.append((x, y))
                        visited_dists.append(((x, y), k))
                    if anchor == p2 and p1 not in visited:
                        (x, y) = p1
                        visited.append((x, y))
                        visited_dists.append(((x, y), k))
```

The algorithm presented here works without queues, minima or relaxation. This is achieved by iterating over possible path lengths (k in Program 6), i.e., first establishing all paths with length 1, then with length 2, etc. Once a vertex has been added to the 'visited' set, no other path can reach it with a shorter path length.

On the flip side, we achieve this 'simplification' and reduction to our minimal set of instructions by sacrificing performance. During each iteration for a specific path length, we iterate over all edges to find suitable candidates. For larger graphs, this is clearly not practical. However, the graphs used in education tend to be small enough, particularly when considering that we aim to write such programs as early in the curriculum as possible.

Although we have also implemented Prim's algorithm for finding a minimum spanning tree with a similar approach, we omit it here for the sake of brevity.

However, note that these algorithms frequently form a basis for solving 'interesting' graph problems (see, e.g., Skiena and Revilla [7]).

4 Accessing Items by Index

The attentive reader will have noticed that we have not made use of Python's facilities to directly access any element in the list by its index, although we mentioned $O(1)$ support. Moreover, the syntax for indexed access includes slices with custom strides. Retrieving every other element from a list `L` is thus as simple as `L[::2]` and getting a list with its elements in reversed order is `L[::-1]`.

Experienced programmers find thus a very convenient and simple syntax in Python's slices. Novice programmers, on the other hand, tend to struggle with the concept of accessing an element by its index. Consider: the syntax for building lists and stating its member elements almost coincides with the syntax for accessing an element by its index:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
print( primes[3], primes[4] )
```

Given the syntactic similarity we should not be surprised that some students explain the above code as: 'First prints the element 3 because it is actually in the list and then throws an error or something because 4 is not in the list' (cf. Boulay [2]).

At this point we have not even touched upon the famous issue of starting the indexing with zero and the 'off-by-one' problems this quickly leads to. How quickly can you tell which elements are to be found in the slice `primes[2:5]`? It is the three numbers $5, 7, 11$—the number 13 at index 5 is excluded as Python's 'intervals' always include the first but exclude the last element.

Dealing with a lack of indices. We recently asked students as part of a programming challenge to determine whether a given short sequence occurs in a longer list of numbers. For instance, the sequence $2, 4$ does indeed occur in the following list:

```
[1, 4, 2, 5, 2, 2, 4, 7, 2, 3, 4]
```

With the tools provided and, in particular, the absence of indices and slices it may seem very hard to solve this challenge for a general case. Perhaps the canonical approach is to build a finite state machine for the short sequence—an approach, however, that does not scale well with the length of the sequence to find.

Program 7 demonstrates an alternative solution with nothing more than the three basic list operations introduced earlier. The bulk of the program consists in listing every possible sequence of the correct length occurring in the given list. The actual test whether the sequence then occurs happens in the very last line.

The approach in Program 7 can equally be used to solve a challenge posed as part of the Swiss Olympiad in Informatics competition 2021/22 [8, 'Peaks'].

Program 7 Determining whether a list contains a given sequence.

```
seq = [2, 4]
numbers = [1, 4, 2, 5, 2, 2, 4, 7, 2, 3, 4]
part_lists = []
for num in numbers:
    part_lists.append([])
    for lst in part_lists:
        if len(lst) < len(seq):
            lst.append(num)
print(seq in part_lists)
```

Given a sequence of numbers count the number of local maxima, i.e., how often a number N_i is larger than its two immediate neighbours: $N_i > N_{i-1}$ and $N_i > N_{i+1}$. The follow-up challenge asks the student to find a subsequence of length K with the highest amount of local maxima in it. This can, again, easily be solved using the approach introduced above.

As before, however, we also have to concede that there are far superior approaches in terms of performance. That is, our solution is probably not fit to compete in such contests, but again demonstrates that the tools provided suffice to solve 'interesting' problems.

5 Turing's Return

After having implemented searches in graphs and sequences, let us tackle another classic problem: *sorting*. At this point, it should be fairly obvious what minimum sort would look like. So, let us focus on *merge sort* instead.

When implementing *merge sort* we have to deal with two challenges: merging two lists and the recursive structure of the problem decomposition. Merging two lists is difficult because we have to advance the iteration in both list separately and non-uniformly. However, Program 8 shows how we may use the `pop()` function from above to address this and perform explicit iteration.

While merging two lists without list indices is challenging from a technical point of view, we may find ourselves more interested in the main function where we have to deal with the recursive 'tree' structure of the problem. Our approach uses a 'todo' buffer containing all the already sorted lists. The algorithm then picks two lists from this buffer, merges them and replaces the original two lists by the merged one until only one unified list remains. With only an 'append' operation for lists, it may not be immediately obvious how to do this.

Luckily, Python's for-loop iterator and the 'append' function are compatible with each other. We may append elements to the list we are currently iterating over and the for-loop will correctly pick up these additional elements. This gives us a neat *queue* data structure that acts as our buffer (Program 9).

Program 8 Merging two lists for *merge sort* requires 'explicit iteration'.

```
def merge(lstA, lstB):
    result = []
    (a, tailA) = pop(lstA)
    (b, tailB) = pop(lstB)
    for _ in range(len(lstA) + len(lstB)):
        if b == None or (a != None and a <= b):
            result.append(a)
            (a, tailA) = pop(tailA)
        else:
            result.append(b)
            (b, tailB) = pop(tailB)
    return result
```

Program 9 The core function of *merge sort* where we use the list `todo` as a queue. By both iterating over `todo` and simultaneously appending elements to it, we effectively end up with a while-loop.

```
def merge_sort(lst):
    todo = []
    for k in lst:
        todo.append( [k] )
    first = None
    for item in todo:
        if first == None:
            first = item
        else:
            todo.append( merge(first, item) )
            first = None
    return first
```

Because we may extend the list we are iterating over inside the loop, we have a tool to control whether the loop shall terminate at any one point. More importantly, though, we can keep the loop going for an indefinite amount of time! Let us reiterate this point: although Python's for-loops can only iterate over lists, they are still as powerful as while-loops (when combined with if-conditions, of course). In other words, the small set of instructions we chose is Turing complete after all.

6 Conclusion

What a ride! By drawing figures with a turtle we accidentally wrote an interpreter and stumbled on a data-code-equivalency. Moreover, by restricting ourselves to iteration and simple list operations, we implemented our own while-loop and dis-

covered that our minimal set of instructions is Turing complete after all.

All this was achieved with a minimal set of instructions. It is thus perhaps tempting to interpret these ideas so as to de-emphasise the teaching of programming. However, using the small set of programming constructs the way we have in this article requires training in how to use and apply them as well as a thorough and solid understanding of their effects. In reality, this is very difficult to achieve, but puts the focus of learning on the right topics and concepts.

In fact, there is a tendency in programming courses to introduce a wealth of programming constructs early on with endless listings of available functions, methods and data types. Yet, if we are interested in core concepts of computer science, in algorithmics and computational thinking, we might actually be better off with a minimalistic set of instructions and data types. We would even argue that a minimalistic instruction set automatically places much more emphasis on algorithm design and problem solving. On any account, we have demonstrated that core concepts of computer science emerge almost naturally even with such a minimal set of instructions. Programming is not only a prerequisite for implementing algorithms but also a fertile ground for discovering and discussing algorithms.

References

[1] M. Böhme and B. Manthey. The computational power of compiling C++. *Bulletin of the European Association for Theoretical Computer Science*, 81:264–270, 2003.

[2] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[3] S. Dolan. mov is Turing-complete (2013).

[4] C. Hughes et al. Project Euler. `https://projecteuler.net`.

[5] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., USA, 1980.

[6] Python Wiki: Time Complexity. `https://wiki.python.org/moin/TimeComplexity`.

[7] S. S. Skiena and M. A. Revilla. *Programming Challenges*. Springer, 2003.

[8] Swiss Olympiad in Informatics. `https://soi.ch`.