Psnodig: From source to pseudo

Sergey Jakobsen

June 1, 2023

1 Introduction

Pseudocode is commonly used for providing a description of an algorithm at a suitable level of abstraction. It is meant to be a comprimise between a low-level implementation in a specific programming language, and a word-for-word solution to problems in a natural language.

One of the advantages with pseudocode is its versatility. It can be used in various domains, including programming, mathematics, and other scientific fields. It is a helpful tool in educational and scientific literature to introduce concepts, particularly algorithms, to diverse audiences.

An advantage with pseudocode is that there is no standardisation, and authors are not tied down to the rules and syntax of any particular programming language. Because of this, pseudocode cannot be executed, and is used exclusively for presentation. Having a non-executable language gives the author complete freedom to omit details or de-emphasize certain aspects of an algorithm.

However, since pseudocode is non-executable, it leaves the author (and the audience) unable to test it. This can lead to accidental inclusion of critical inaccuracies. Futhermore, a student with no prior experience or knowledge to a certain algorithm is not likely to identify non-obvious errors. This can lead to them transferring it to their own work later.

Correct presentations are especially important in academia, where the goal is to teach students concepts they were previously unfamiliar with. Traditionally, concepts within the algorithms and data structures realm have proved challenging for undergraduates [2]. If then their first impression of an algorithm is an incorrect presentation, their path is already hampered.

In this essay I will present the background and motivation (Section 2) for doing this task, and how I propose it could be solved (Section 3).

2 Background

In this section I will introduce the motivation behind the thesis (Section 2.1), definitions (Section 2.2) and current approaches to the problem (Section 2.3).

2.1 Motivation

With this thesis, I wish to explore possibilities of automatically creating pseudocode from source code (which is executable), for two main reasons:

1. To reduce the amount of human error in the process of writing pseudocode
2. To lessen the author's workload

When presenting an algorithm you have created, it is common to present it using pseudocode to reach a broader audience. After all, everyone might not be familiar with whatever esoteric programming language you hold dearest to your heart. When translating your source code to pseudocode, you are met with an unlimited amount of choices. What could be emitted? What must be included? Which part can be abstracted and which part might as well be written in plain English?

Taking the source code of a given language and translating it to pseudocode automatically will at least lead to a certain degree of consistency. One must assume that struggling students will appreciate predictability when introduced to a large amount of previously unfamiliar pseudocode.

When working with a common programming language in an IDE, one can often get valuable help from a linter[1] to keep the code clean. For instance, a general linter can pick up on unused variables and missing statements. When writing pseudocode, however, authors are left entirely to their own devices.

Though consistent, the generated pseudocode might not always give a perfect representation of the author's code. However, the basic building blocks will be in place, so the author can just tweak what she wishes was different, rather than having to write out the entire thing from scratch.

A tool that automatically translates source code to pseudocode would likely reduce the amount of extra work the author has to do, as they can now to a greater extent worry about the code they have that is executable. Also, when performing the translation manually, the author must always keep track of changes, by now maintaining both the source code and the pseudo-equivalent pseudocode.

2.2 Definitions

In this section, I will define some terms that will be referred to later in the essay.

Algorithm According to Donald Knuth [5], an algorithm consists of five important features:

1. Finiteness
2. Definiteness
3. Input
4. Output
5. Effectiveness

Algorithms are supposed to terminate after a finite amount of steps. Exactly how many steps, however, is arbitrary, as long as it eventually ends. The steps must also be precise and unambigous, leaving no room for artistic freedom from the reader.

Algorithms must take nil or more inputs, and are also expected to return an output of some sort. Lastly, algorithms are expected to solve problems effectively. They should be able to solve the problem they're trying to address in a sensible amount of time (though this will depend on the problem at hand).

Some examples of well-known algorithms are Dijkstra's algorithm for finding the shortest path in a graph, Prim's algorithm for finding a minimal spanning tree in a graph, and insertion sort, which is used to order the elements of a collection [4].

Pseudocode Pseudocode is an abstract representation of an algorithm, portrayed in a simplified manner to be read by humans rather than computers. It is, as previously mentioned, purposely non-executable, to allow the author to emphasise what she herself deems important regarding her algorithm. It is not tied to a specific programming language, but rather a way to express the *behaviour* of algorithms in a way that is easier for humans to understand.

Pseudocode commonly includes natural language, common programming constructs, and often an imperative formatting to describe the steps of an algorithm in an intuitive way, mimicking that of a typical reciepe. It allows the author to focus on the logic and structure of her algorithm without adhering to the strict syntax of a specific programming language.

The idea is that people who use different programming languages should be able to understand the same pseudo code, due to its flexibility. You can expect to find different pseudocode describing the same algorithm across different articles, papers and websites. Sometimes, all

[1] https://en.wikipedia.org/wiki/Lint_(software)

mathematical abstractions are removed, and pseudocode is instead presented solely with natural language.

Figure 1 shows two different approaches to the same problem. The algorithm is supposed to solve a variation of the infamous FizzBuzz problem[2]. The user is given an integer, and met with exactly four conditions: if the number is divisible by 3 and 5, the algorithm must return the word fizzbuzz. If it is only divisible by 3, it should return "fizz", and if only divisible by 5, it should return "buzz". If none of the cases match, it should just return the original integer.

```
Procedure Fizzbuzz(n):
  if n modulo 3 is 0 && n modulo 5 is 0 ->
    return "fizzbuzz"
  else if n modulo 3 is 0 ->
    return "fizz"
  else if n modulo 5 is 0 ->
    return "buzz"
  else ->
    return n
```

```
Algorithm Fizzbuzz(number n)
  if n is divisible by 3 and 5,
    return "fizzbuzz"
  if n is divisible by 3,
    return "fizz"
  if n is divisible by 5,
    return "buzz"
  otherwise, return the number
```

Figure 1: A solution to the FizzBuzz problem, shown with two different representations of pseudocode

On the left, one can recognise mathematical/programming language concepts like arrows, modulo and double ampersands. On the right, the entire algorithm is written in plain english, like an old-fashioned recipe you would find laying around your parents' house.

Nonetheless, the algorithm's control flow is conveyed, and no prior (or at least extensive) experience with a specific programming language is necessary to grasp the concepts introduced. Therefore, it is also difficult to say which one is better, as different people have shown to prefer different approaches. [8]

Interpreter An **interpreter** is a computer program that parses the source code of a program into an abstract syntax tree (AST), before evaluating and executing AST node by AST node. The interpreter also maintains a runtime environment, which includes values of variables, functions etc. Examples of interpreted languages are Python[3] and Ruby[4].

Transpiling A source-to-source translator is a tool that takes the source code of a program written in a certain programming language, and translates it to source code in another language. From here on out I will refer to source-to-source compiling as **transpiling**.

If you are not familiar with transpiling, it might help to think of compiling. During compilation, source code is translated to code on a significantly lower level. For instance, a file containing Java source code can be compiled to byte code, which is then executed. Transpiling, on the other hand, does not execute programs, and has no concern as to whether or not the code it reads is syntactically sound.

Instead, it builds an AST and translates each node into the corresponding node from the other language, attempting to keep the abstraction level similar.

Transpiling is valuable for many reasons. For one, it can be used to update an existing Python 2 code base to Python 3, or from C to C++. Frameworks and languages are constantly being updated and re-released, and with time, support for earlier versions ends [9]. Therefore, it is sometimes not only enjoyable and advantageous to upgrade a code base, it is also strictly necessary.

Transpiling the code base (or parts of it) is therefore an appropriate alternative to refactoring it manually, which requires a lot of time and domain knowledge.

[2] https://leetcode.com/problems/fizz-buzz/
[3] https://www.python.org/
[4] https://www.ruby-lang.org/en/

Pandoc A tool for transpiling is **Pandoc**. It is a Haskell[5] library for converting from one markup format to another, and a command-line tool that uses this library. After installing Pandoc from the website[6], you can use the command line effectively translating your files from one markup format to another.

One way of doing this, is writing directly in the terminal. For instance, you could write

```
## This is one format
A *simple* example
- This is a list entry
- And this is another
```

Pandoc converts to HTML by default, so you will now be greeted with

```
<h2 id="this-is-one-format">This is one format</h2>
<p>A <em>simple</em> example</p>
<ul>
<li>This is a list entry</li>
<li>Ant this is another</li>
</ul>
```

You can also put the input in a file, say "example.md". While you are in the folder of that file, you can translate it to HTML with the command line. By writing

```
pandoc example.md -o example.html
```

you will receive a file called "example.html". Contrary to the command line output shown above, example.html will contain a lot of meta information and styling.

Domain-Specific Language A **Domain-Specific-Language** (DSL) is a programming language designed for a specific application domain, rather than general-purpose programming. [3]

DSLs are created to solve very specific problems, that need more focus than other, existing languages, are willing to focus on. An example of a DSL is HTML, which cannot be used for (or at least is not intended for) computations, but is excellent for making websites.

DSLs can be put into two categories: internal and external. Internal DSLs are embedded within a general-purpose language, utilising the syntax and semantics of the latter. External DSLs, however, are standalone languages that have their own syntax and semantics, and require their own parsers. [3]

2.3 Related work

There have been some approaches to this problem, though with different cases in mind. In this section I will mention two of them: The programming language Drew, which compiles to plain english, and a 'source code to pseudocode'-generator made with statistical machine translation.

2.3.1 Drew

Derw[7] is a programming language aimed at making complex apps in a simple way. It is interesting to this thesis because it can compile to plain English. It takes a lot of inspiration from the programming language Elm[8], particularly in the sense of giving very explanatory error messages.

Derw's compiler has three main phases: lexing, parsing and compiling. It targets multiple programming languages, therein Derw, Elm, and English. There are multiple generators which can be used during compilation, one for each language. In principle, a generator takes the parsed AST, and generates the correct output based on the corresponding AST nodes.

[5] https://www.haskell.org/
[6] https://pandoc.org/
[7] https://github.com/eeue56/derw
[8] https://elm-lang.org/

2.3.2 Pseudocode-generator

In a 2015 paper, Oda et al. propose a method to automatically generate pseudocode from source code, specifically adopting the statistical machine translation (SMT) framework. [7]

Statistical machine translation is a type of machine translation that translates text from one language to another, with the help of statistical models. The process involves analysing large amounts of text data of the relevant languages to learn patterns and generate translations that are likely to be correct, statistically speaking. [6] An example of a tool for statistical machine translation is Google Translate[9].

The objective of SMT in this case is to generate the most probable target sentence t given a source sentence s. This is made possible by introducing phrase pairs, where Python syntax is mapped being mapped to some corresponding text in English. For instance, the following source code

```
if x % 5 == 0:
```

produces the following pseudocode

```
if x is divisible by 5,
```

with help from the following mappings:

- ”if” → ”if”
- ”x” → ”x”
- ”% 5” → ”by 5”
- ”== 0:” → ”is divisible”.

Then, to achieve a gramatically sound sentence, a method called ”reordering” is done. This is carried out by multiple machine learning models that measure the fluency of the sentence, calculate the probabilities of the individual phrases in the sentence, and calculate the probability of the arranging each phrase in a particular order.

Unfortunately, the tool is not publically available, so there is no way of testing it right now, to my knowledge.

3 Proposed solution

One of the key problems with the pseudocode generator is that SMT is developed with natural language in mind. Therefore, when generating ”pseudocode” in English or Japanese from Python source code, the pseudocode is purely language. This is fine, for a person that has never read any code at all. But our target audience is people who are already somewhat familiar with programming.

Also, the paper assumes pseudocode to be a ”line-to-line” translation between the source code and natural language. We, on the other hand, want to let the author abstract away certain details of her algorithms. For instance, there are parts of an algorithm where the implementation is not particularly important to the final result (i.e. the way a pivot is chosen in the sorting algorithm quicksort [4]). Rather than troubling the reader with this code, the author should be given freedom to simply put it in a function (or sum it up with natural language) in the pseudocode.

Also, the papers are specific to programming languages Derw and Python, respectively. I will also base the reader on a programming language (see Section 3.1), but also make it parameterisable through the use of monadic parser combinators.

[9] https://translate.google.com/

3.1 DSL

I will design an external DSL (called Psnodig) for transpiling source code to psuedocode. The output shall be exported to one or more target formats, e.g. LaTeX and HTML. Although intended to be Turing complete, Psnodig is not intended for general purpose programming.

Psnodig will have an imperative syntax, to give the user an intuitive way of writing algorithms, like you would for any other receipe. It will take syntactic inspiration from existing languages like Kotlin[10], Python[11] and Janus [10]. It is will be as expressive as it needs to be to work with algorithms and data structures, thus containing identifiers, records, data types like integers and floats, various operators and so on.

To settle on a definite syntax I will study relevant literature, particularly Algorithms Unlocked [1] and Algorithm Design and Applications [4] to help me prioritise what is necessary to include. I will also study the algorithms introduced in Algorithms and Datastructures[12] and Algorithms: Design and Efficiency[13] and their presentation, to see what is already being applied in academia. I wish to keep the language simple and avoid superflous keywords, to the extent possible.

I will also take inspiration from programming languages like Prolog[14] and Python[15], which have an English-like syntax.

3.2 Output

I will not attempt to create a universal ground truth for pseudocode. As mentioned in Section 2.2, different styles work for different use cases and audiences. The transpiler must, however, export the source code to *something*. Therefore, there will be a default target language to which Psnodig files are exported. The target language will likely be along the lines of this:

Algorithm: FizzBuzz

Input: A natural number n

Output: If n is divisible by 3 and 5, return FizzBuzz. If n is only divisible by 3 or 5, return Fizz or Buzz, respectively. If it is divisible by neither, return the number as it is

```
1  Procedure FizzBuzz(n)
2      if n mod 3 and 5 is 0 then
3          return FizzBuzz
4      else if n mod 3 is 0 then
5          return Fizz
6      else if n mod 5 is 0 then
7          return Buzz
8      else
9          return n
10     end
```

From here, there are two interesting options to explore: parameterising the writer, or allowing authors to include their own writer.

Parameterising the writer

Since different authors might wish to present their code's behaviour differently, the default pseudocode will ocassionaly require manual refinement. For instance, say the default pseudocode transpiles this

```
fun sortArray array =
```

to this

[10] https://www.kotlinlang.org/
[11] https://www.python.org/
[12] https://www.uio.no/studier/emner/matnat/ifi/IN2010/
[13] https://www.uio.no/studier/emner/matnat/ifi/IN3130/
[14] https://www.swi-prolog.org/
[15] https://www.python.org/

```
Procedure sortArray (array)
```

But the user is not fond of the word "Procedure" in her pseudocode. Instead, she wants to see the word "Algorithm", like this:

```
Algorithm sortArray (array)
```

Then perhaps there could be a way of parameterising the writer, so that "fun" maps to "Algorithm" rather than "Procedure".

Including their own writer

The other approach is to allow authors including their own writer in our program. This way, they will be sure that the resulting pseudocode is what they had in mind. Also, they will not have to twist and turn the default writer, and can instead focus on improving their own.

nevn noe om derw og generators og sånt.

3.3 Running Psnodig

When the user wants a psnodig program interpreted, she can write the following command to execute the code (psnodig programs will end with .psn):

```
psnodig file.psn
```

or

```
psnodig file.psn run
```

When no argument is provided, the default will be to execute the code. However, if she instead wishes to generate a LaTeX/PDF file of pseudocode based on her program, she can write the following command:

```
psnodig file.psn pseudo
```

This will not run the program, but instead transpile the source code to a LaTeX file, or alternatively a PDF document with compiled code from the LaTeX file.

The advantage of this is that the user can run and test her program to verify that it works as intended, and that she attains the desired results. Then, when she is satisfied, she can choose to translate her work into pseudocode.

3.4 Technical approach

The plan is to write an interpreter in the Haskell programming language. I will make use of the Parsec[16] library for monadic parser combinators, to write a lexer and parser in one. Then, I will write the interpreter in Haskell.

By using Parsec, I can not only parameterise the writer, but also the reader, with higher-order functions and combinators provided by the library. For instance, a function can take certain parameters, and modify the parser accordingly.

I can also adopt the Derw approach of allowing generators to be put on top of Psnodig. That way, authors can add their own specifications, and though it will take more work on their half, it will gaurantee them their desired results.

One alternative to creating my own DSL would be to expand an existing programming language, like Scheme[17]. Nevertheless, I have chosen a different route, because I want full ownership of the tool, and I want the possibility to tailor the DSL to my thesis.

[16] https://hackage.haskell.org/package/parsec
[17] https://www.scheme.org/

References

[1] T. H. Cormen. *Algorithms Unlocked*. MIT Press, 2013.

[2] H. Danielsiek, W. Paul, and J. Vahrenhold. "Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 21–26.

[3] M. Fowler. "Domain-Specific Languages Guide". In: Martin Fowler, 2019.

[4] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications*. Wiley, 2014.

[5] D. E. Knuth. *The Art of Computer Programming, Volume 1*. 3rd. Addison-Wesley, 1997, pp. 4–6.

[6] P. Koehn. *Statistical Machine Translation*. Vol. 5. 4. 2009, pp. 27–28.

[7] Y. Oda et al. "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by M. B. Cohen, L. Grunske, and M. Whalen. IEEE Computer Society, 2015, pp. 574–584.

[8] S. Siozou, N. K. Tselios, and V. Komis. "Effect of algorithms' multiple representations in the context of programming education". In: *Interact. Technol. Smart Educ.* 5.4 (2008), pp. 230–243.

[9] T. Tran. "Top 51 Software Development Frameworks For 2023". In: *Orient Software* (2022).

[10] T. Yokoyama. "Reversible Computation and Reversible Programming Languages". In: *Proceedings of the Workshop on Reversible Computation, RC@ETAPS 2009, York, UK, March 22, 2009*. Ed. by I. Ulidowski. Vol. 253. Electronic Notes in Theoretical Computer Science 6. Elsevier, 2009, pp. 71–81.