UNIVERSITY
OF OSLO

Master's thesis

Psnodig: A tool for code presentation in academia

Sergey Jakobsen

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Autumn 2023

Sergey Jakobsen

Psnodig: A tool for code presentation in academia

Contents

Chapter 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.

An advantage with pseudocode is the lack of standardisation, thus authors are not tied down to the syntax of any particular programming language. This gives the author complete freedom to omit or de-emphasize certain aspects of her algorithms. Thus, pseudocode is used exclusively for presentation. However, as pseudocode is non-executable, there is no omniscient way of verifying its correctness. This can, in turn, lead to accidental inclusion of critical inaccuracies, particularly when working with lesser-known algorithms.

1.1 Motivation

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 [1]. If then their first impression of an algorithm is an incorrect presentation, their path is already hampered.

In this thesis, we present a tool, "Psnodig", which allows for transpiling executable code to other, perhaps less technical presentations. The presentation targets in this thesis are pseudocode and flowcharts. The target audience is students who are already familiar with some C-like programming language syntax. The reader might be familiar with the term "compiling", where the goal is to take a piece of non-executable code and transform it into a piece of executable code. Transpiling however, in this context, is the act of translating one piece of code to a different piece of code, while keeping a similar level of abstraction [kilde?].

One of our output targets is pseudocode. We are not attempting to create a standardisation or ground truth for pseudocode, but instead we lean on the

algorithm2e[1] library, which is an aleady established environment for writing algorithms in LaTeX. One thing we *can* ensure, is consistency, removing the burden of "should this be in italics or boldface or not" from the author. In addition, the author can write her algorithm in an executable programming language, and when sure that it yields the desired result, she can safely use Psnodig to receive a LaTeXfile and a companion PDF of her work.

Other freely available alternatives for transpiling source code to pseudocode is Code Kindle[2] and Pseudogen, a tool introduced by Oda et. al[4]. Both solutions use statistical machine translation, which is a technique to train a model on previously translated and analyzed information and conversations[3]. With Pseudogen, code is transpiled to purely natural language. Code Kindle's results are less verbose, but in most cases still a description accompanying the original source code.
The other target output of Psnodig is flowcharts. This is a representation very different to the original source code, and the main difficulty consists of keeping the level of abstraction. The main benefit of this representation, is to see the code from a completely different angle. This can be refreshing when looking at the code from the same angle for an extended period of time yields no result.

Currently, to our knowledge, the only tool for converting source code to flowcharts is Code2Flow[4]. This is a DSL with support for most common programming concepts like statements, loops, conditionals, and more. The positive effect of teaching with flowcharts as an alternative to traditional code has been researched since the 1980's by Scanlan[5], and also in recent times by Giordano et. al[2], yet direct translation from source code to flowcharts does not seem very widespread.

We spend much more time reading code than we do writing code[3, p. 14], and tools like IDEs and linters can only help us so much long term when it is the logic of our programs that we fail to grasp. We believe that the Psnodig tool can be an alternative for both authors wanting to present their algorithms, as well as students wishing to get a better grasp of them.

1.2 Psnodig vs. Pseudogen

Let us take a more thorough look at Pseudogen, and how it differs to Psnodig. Their transpiler is currently designed to work with a subset of the Python programming language[5]. The output target is pure natural language, precisely what you are reading now.

Despite being a programming language notoriously known for using plain English where many other programming languages use more technical notation

[1] https://www.ctan.org/pkg/algorithm2e
[2] https://devpost.com/software/code-kindle
[3] https://en.wikipedia.org/wiki/Statistical_machine_translation
[4] https://app.code2flow.com/
[5] https://www.python.org/

(*and* instead of &&, *or* instead of || etc.), Python still bears the mark of being a programming language. People not familiar with programming and/or mathematics might struggle to understand what the % (modulo) is and what it is for.

That is where Pseudogen comes in. Not only is mathematical notation like

> if n % 3 == 0:

transpiled to

> if n is divisible by 3,

but also programming language specific elements like

> raise TypeError('n is not an integer')

is transpiled to

> throw a TypeError exception with a message ...

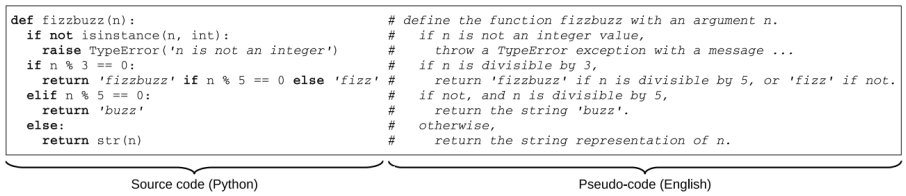

Figure 1.1: Example of source code written in Python and corresponding pseudo-code written in English from Oda et. al

For an audience with little to no programming language expereince, this is likely fine. A boss that wishes to see what her engineers are spending their time on, a curious George wanting to get insight into TikToks algorithms, and anyone in between.

Psnodig, however, offers pseudocode for a different crowd: people, primarily the ones involved in academia, who already have some experience with writing and reading code. If you know that the % symbol stands for modulo, and that it represents the action of returning the remainder of a division operation, it is no longer benefitial to constantly read the verbose description. When reading a novel in italian after successfully learning the language, you would likely prefer each page driving the story forward, rather than having an english word-for-word translation on every other page.

While Pseudogen drives the abstraction levels of Python down to natural language, Psnodig instead wishes to stay closer to the code, while still allowing for an extra layer of abstraction when the implementation is clumsy or too language-specific.

1.3 Psnodig vs. Code2Flow

Let us take a more thorough look at Code2Flow, and how it differs to Psnodig. Code2Flow lets the user create flowcharts with natural language, decorated with a C-like syntax.
It mainly consists of

- start- and end expressions, drawn as red ovals
- other expressions, drawn as blue rectangles
- conditionals, loops and match statements, drawn as red red rhombuses
- comments, drawn as orange rectangles

Expressions are separated by semicolons. This is the default colour scheme, though there are also three others the user can pick.

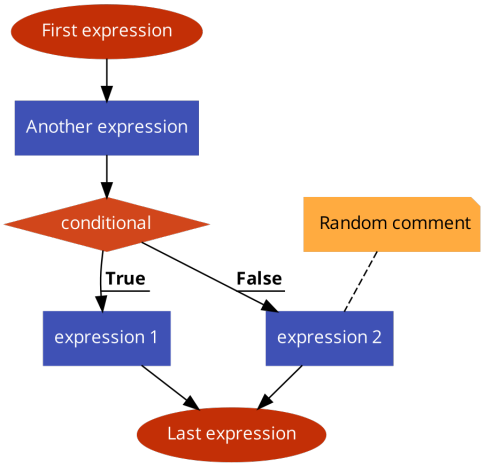

Figure 1.2: Example of a program written with Code2Flow

The above diagram is the result of writing the following code in the online code2flow interpreter:

```
First expression;
Another expression;
if (conditional) {
  expression 1;
} else {
  expression 2; // Random comment
}
Last expression;
```

Using flowcharts to visualise a program is nothing new, but again we have a situation where the autor has to re-write their algorithm in yet *another* language, and hope they did not drift too far from their original code. Naturally, these

flowcharts cannot be tested with any input, and are meant to be presentation-only.

Psnodig, on the other hand, makes a point of transpiling syntax-error-free code, which the author can first test on whatever input they want. First then are they allowed to transpile their code, which does not require them to write anything again, unless they want to tweak any details.

1.4 Psnodig

The Psnodig tool in itself is really just a syntax. This syntax includes the standard building blocks like statements, expressions, function declarations, structs etc. But, it also includes two new types of statements, which are specific to our DSL.

Hash-Statement, which can be written like this with BNF:

HashStmt ::= # <Stmt>.

These statements are read and and processed by the interpreter, but ignored during transpiling. They work much like macros do in e.g. the C programming language, but they are declared inside functions, as Psnodig only allows struct- and function declarations to lie in the global scope. They are limited to the line reside in, which makes it easy for the author to decide which lines should be included or ignored when they wish for a different presentation of their code.

At-Statement, which can be written like this with BNF:

AtStmt ::= @ { text } { [<Stmt>] }

These statements consist of two parts, pure text in the first scope, followed by new statements in the second scope. The second part is meant for the interpreter, whilst the first part is meant for the transpilers. This allows the author to abstract over implementation-specific details and/or messy code, which is not crucial for the program's logic. The statement list can also be an empty list, which also makes this a way of letting the author explain things solely with natural language when deemed necessary.

In addition to the syntax, the tool comes with a parser and interpreter for the Gourmet programming language, designed as a proof of concept for this thesis. It also comes with two writers, one presenting the source code with pseudocode, and the other presenting the source code in the form of a flowchart. The main benefit is that we can write our code once in Gourmet, test it, and when we are satisfied, transpile it to pseudocode and/or flowcharts thorugh the command line, rather than having to re-write it manually.

To the best of our knowledge, a tool which combines these methods does not already exist.

1.5 Research Questions

In this thesis, we aim to answer the following research questions:

RQ1: Can we transpile source code to pseudocode whilst maintaining a similar level of abstract?

RQ2: Can we transpile source code to flowcharts whilst maintaining a similar level of abstract?

RQ3: Can people in academia find such a tool to be helpful in better understanding small-to-medium sized code chunks?

1.6 Contributions

The main contribution of this thesis is the Psnodig tool for transpiling executable source code to various presentation-only versions of said code, giving people in academia an easy and accessible way of looking at their code from a different angle. By using the Psnodig tool, people can spend more time writing code and less time mastering LaTeXlibraries, writing boilerplate and worrying about consistency.

The Psnodig tool is written entirely in the Haskell programming language[6], offering a syntax rich enough for writing all algorithms and data structures introduced in the introductory course to algorithms and data structures at the University of Oslo[7]. The tool comes with a parser for the Gourmet programming language, as well as a writer for both pseudocode and flowcharts in LaTeX, utilising the Algorithm2e package[8] and the TikZ package[9].

To summarise, the contributions include:

- Psnodig, a tool for transpiling code from one representation to another through a pre-defined syntax.
- The Gourmet programming language, inspired by Go and Python, as a proof of concept. This includes a parser for converting tokens to an AST, as well as an interpreter for executing the nodes in said AST, both implemented in Haskell.
- A LaTeXwriter for presenting an AST with pseudocode, utilising the Algorithm2e package.
- A flowchart writer for presenting an AST with diagrams, utilising the TikZ package.

[6] https://www.haskell.org/
[7] https://www.uio.no/studier/emner/matnat/ifi/IN2010/index-eng.html
[8] https://www.ctan.org/pkg/algorithm2e
[9] https://www.overleaf.com/learn/latex/TikZ_package

1.7 Chapter Overview

Chapter 2 introduces ..

Chapter 3 defines ..

Chapter 4 covers ..

Chapter N concludes the work of this thesis, discussing the research questions and future work.

1.8 Project Source Code

All the source code from the master thesis can be found on Github[10].

[10] https://github.com/dashboard

Chapter 2

Background

- Previous work within the scope of the thesis
- Relevant related work like Pandoc and Derw

Chapter 3

(Problem) Analysis

- What is the problem I am looking at?
- Analysing papers
- What is relevant to include?
- Builds on the background chapter

Chapter 4

Design

- This is how I intend to solve the problem!
- Builds on the analysis chapter

Chapter 5

Implementation

- Which tools am I using?
- Concrete implementation of the Psnodig tool
- Testing: How can I be certain that Psnodig works?

Chapter 6

Evaluation

- Use cases
- Examples of how it works in practice
- Strengts, weaknesses etc.
- Maybe: Test it on students that take/have taken IN2010? let them write algorithms and let the transpiler do the work

Chapter 7

Discussion

- How does my solution hold up against the problem?
- Does it solve *anything*?
- Is it better than what is currently available?

Chapter 8

Conclusion

- Future work
- Optimisations etc.
- Conclude RQs
- Important: Remind the reader of all the good stuff!

Bibliography

[1] 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.

[2] D. Giordano and F. Maiorana. "Teaching algorithms: Visual language vs flowchart vs textual language". In: *2015 IEEE Global Engineering Education Conference (EDUCON)*. 2015, pp. 499–504.

[3] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[4] 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.

[5] D. A. Scanlan. "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison". In: *IEEE Softw.* 6.5 (1989), pp. 28–36.