UNIVERSITY
OF OSLO

Master's thesis

Psnodig: The many faces of code

Sergey Jakobsen

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2024

Sergey Jakobsen

Psnodig: The many faces of code

Contents

Listings

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 [**10.1145/2157136.2157148**]. 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 rather 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[**DBLP:conf/kbse/OdaFNHSTN15**]. 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, though 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[**DBLP:journals/software/Scanlan89**], and also in recent times by Giordano et. al[**7096016**], 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[**martin2008clean**], 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.

[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/

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

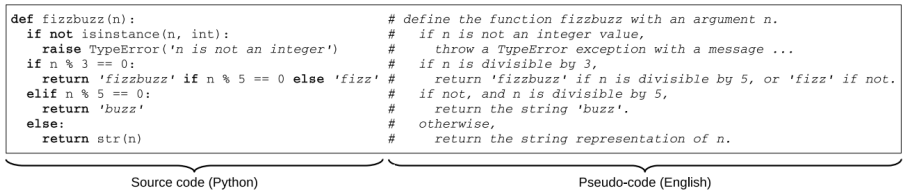

Figure 1.1: Example of source code written in Python and corresponding pseudo-code written in English from Oda et. al

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.

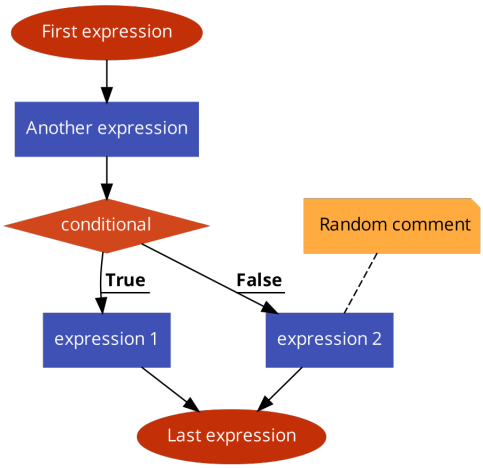

Figure 1.2: Example of a program written with Code2Flow

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, however 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. However, 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 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. Also comes with an interpreter which works on the intermediate AST representation.
- 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 a writer to convert the AST back to Gourmet code.
- A writer for presenting an AST with text-based pseudocode, utilising the Algorithm2e package in LaTeX.
- A writer for presenting an AST with image-based pseudocode (diagrams), utilising the TikZ package, also in LaTeX.

[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 5 covers ..

Chapter 6 covers ..

Chapter 7 covers ..

Chapter 8 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
- Important definitions

This chapter will cover concepts that one should be familiar with in order to fully understand the rest of this thesis. We start by defining pseudocode to avoid confusion further down the line, as well as discussing transpiling, why Haskell is a good implementation language, and how other transpilers work.

2.1 Pseudocode

Pseudocode is a loose term used to describe a presentation-only version of code ...

History? Use (particularly in education)?

To avoid confusion, we make a distinction between two particular types of pseudocode: text based and image based.

2.1.1 Text based pseudocode

Black n white stuff

2.1.2 Image based pseudocode

Not all programming languages share the same execution flow. For istance, in VHDL all processes are executed simultaneously[1], whilst in Maude rules are applied in an arbitrary order(source??). Some, on the other hand, like Python, will execute their programs line for line. This means that we can follow the execution flow simply by looking at the order functions are called and the order of statements within those functions.

[1] https://www.people.vcu.edu/ rhklenke/tutorials/vhdl/modules/m12_23/sld008.htm

This way of executing a program opens up for the possibility of flowcharts, which still includes code, but also complements it with boxes, arrows and perhaps pretty colours. When code stretches over enough lines, it all starts looking similar and confusing. Flowcharts, on the other hand, does a good job of isolating each statement, and giving a more birds eye view perspective of the code.

2.2 Transpiling

We have a program, we transform that into an intermediate representation, and from that intermediate representation we can get an entirely new representation.

An alternative to transpiling is the well-known phenomenon of *compiling*. The difference is that we do not stay on the same abstraction level, but rather we make it much more specific. An example of this is compiling a Java file to bytecode. To humans, the bytecode reads like the most foreign language, though the JVM understands it perfectly, and is able to execute it.

2.2.1 Generators

One technique of utilising parsers and writers is to apply generators. A generator is basically a stand-alone parser or writer. A parser generator, in the context of Psnodig, would translate an example program to an AST, whilst a writer generator would translate an AST to a program.

An example of this is the programming language Derw[2], an ML language mainly inspired by Elm. By utilising generators, it has multiple writers, among others bytecode, JavaScript, and even English.

Another example is Pandoc[3], which works with markdown languages. It has a "core language" which all parsers and writers must oblige to, and looks like this

```
Plain [Inline]
Para [Inline]
LineBlock [[Inline]]
CodeBlock Attr String
RawBlock Format String
BlockQuote [Block]
OrderedList ListAttributes [[Block]]
BulletList [[Block]]
DefinitionList [([Inline], [[Block]])]
Header Int Attr [Inline]
HorizontalRule
Table [Inline] [Alignment] [Double] [TableCell] [[TableCell]]
Div Attr [Block]
Null
```

[2]https://github.com/eeue56/derw
[3]https://github.com/jgm/pandoc

Every inputlanguage must be able to parse to (at least) these data types, and every writer must be able to work with these data types.

- Give example with e.g. HTML and MD for one of the data types?

2.2.2 Haskell's strengths

As previously mentioned, we opted for the Haskell programming language when implementing Psnodig. There are several reasons as to why, but the primary one is that it is widely perceived as a fitting tool when working with programming languages, and particularly when working with interpreters[4]. At the end of the day, programming languages are just tools, and we believe this is the best one for this particular job.

With Haskell, it is straightforward to create your own **data types**, which are then used to model abstract syntax trees (ASTs). For instance, we can create our own calculator language in just a few lines of code:

```
data Program = Program Expression

data Expression =
      CompoundExpression Integer Operator Expression
    | IntExpression Integer

data Operator =
      Plus
    | Minus
    | Times
    | Division
```

From this, we can construct the following AST:

```
Program (CompoundExpression 1 Plus
            (CompoundExpression 2 Minus
                (IntExpression 3))
```

As you can also see, we could create much bigger calculations than this, and if we wish to include extra operators, like for instance exponents in the future, we simply add a `| Exponent` to the Operator data type.

Another benefit of using Haskell, is that its strong type system opens for clean and efficient pattern matching. This is very useful, both when writing the interpreter, but also when adding new, potential readers. For example, if we wish to transpile the above AST to text, we could start with writing a function to convert the operators:

```
f :: Operator -> String
f (Operator Plus)     = " + "
f (Operator Minus)    = " - "
f (Operator Times)    = " / "
```

[4] https://github.com/Gabriella439/post-rfc/blob/main/sotu.md#compilers

```
f (Operator Division) = " * "
```

Listing 2.1: Haskell example to convert data type to string

The function **f** takes something of type **Operator** as input, and returns something of type **String**. It will pattern match on the input, and return a correspondng value, making it bijective. We could add case of `f _ = ""`, which would return the empty string for any other kind of operator, though this would be redundant as we have not defined any other type of operator anyway.

Chapter 3

(Problem) Analysis

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

In this chapter, we will be doing problem analysis. We will dive deeper into the problem we intend to fix, and we will discuss what the current solutions lack. While the alternatives we present boast about successfull outcomes, there are some shortcomings when comparing them to what we wish to achieve, as well as comparing the respective target audiences.

3.1 The problem at hand

The issue of translating code to some sort of pseudo code has been around for a while (source?), often intended to help people less familiar with code, or perhaps those with no familiarity to code whatsoever, to understand what is being done behind the scenes, and better understanding what product is being built etc. (source??).

We, on the other hand, believe that people who *do* have some familiarity with code could also benefit from being exposed to it. For one, it can be a nice tool for debugging a-little-too-fancy code you did not write yourself, by getting a more abstract view of it. It can also aid beginners in seeing the flow of their code, making them understand how they can improve their code. Unfortunately, there does not seem to be any tools freely available on the market with this target audience in mind: people who also code, but would like to see their code from a different perspective.

Another aspect of pseudo code, is that since it is intended to be a presentation-only tool, you cannot actually verify its semantical properties, that is, if it even does what you want it to do. As Donald Knuth famously put it,

I have only proven the algorithm correct, not tested it.[**DBLP:books/aw/Knuth68**]

Therefore, by manually translating executable code into a non-executable form, we are no longer able to test it, and the work of maintaining both quickly turns into a hassle. We believe that everyone benefits from there persisting a stronger relationship between the original code and the presentation-only pseudocode.

Ever since the concept of pseudocode was introduced, there have been attempts at creating tools to automate the process of translating code to pseudocode. The most noteworthy attempt to deliver pseudocode in text format was presented in a 2015 paper, a tool called Pseudogen. When it comes to translating code to flow charts, we decided to look at Code2Flow, which is widely used in practice today, even in PIT, the Norwegian police's IT service.

3.2 Translating code to natural language: Pseudogen

Pseudogen boasts about generating pseudocode from Python. What the examples in the 2015 paper, as well as a video on their website[1] show, is rather a line-for-line translation to English. This could be desired in cases where the business people on the team are particularly curious about what the product is really doing under the hood (and the boss cannot afford Cobol developers). Since Pseudogen will translate each line in a slave-like manner, we also translate all error handling. For example, the following two lines

```
except ValueError as e:
    print(e)
```

will translate to

```
# If ValueError, renamed to e, exception is caught.
    # Call the function print with an argument e
```

One can speculate as to whether or not anyone gained much knowledge from that, though that is, luckily, not our task. We can also assume the business people would not be overly interested in each single piece of error handling anyway, but the absence of possibility for abstraction can make the translated transcript overwhelmingly verbose.

Due to every line being translated, succinct and elegant list comprehension like

```
a = [f(n) for n in range(-10, 10)]
```

is translated into this long, tangled spaghetti of words

```
# Call the function f with an argument n for every n in
  range of integers from range 10 negative integer 10,
  substitute the result for a
```

[1] https://ahclab.naist.jp/pseudogen/

The target audience is people who prefer the English translation to the Python code. The two examples we just provided show that the target audience is unlikely to be someone who has any background with at least programming or mathematics, which is in turn the target audience for *our* tool.

The choice of creating a tool like Psuedogen for a programming language like Python makes sense as a prototype, because Python is already so closely related to "natural language". Using syntax like *and*, *or*, colons etc. often makes Python code very easy to read. Thus, imagine a scenario where we declare a list of 10 integers, create a new list by filtering out the even ones, and print the result. In Python, we could do something like

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = [num for num in numbers if num % 2 == 0]

print("Even numbers:", even_numbers)
```

This is already so close to what we would have if we were to write the commands in English, making the task of translation simple. Though it makes you wonder, what is even the point of translating that? As we saw earlier, particularly translation of list comprehensions turn out rather messy compared to their succinct Python counterparts. Another argument against its usefullness is that the listing above is a fully fledged program, ready to be interpreted! If a user is still unsure about what the program is doing, executing it will certainly silence their doubts.

Now, let's analyse the same program written in the Go programming language:

```
numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

var evenNumbers []int

for _, num := range numbers {
    if num % 2 == 0 {
        evenNumbers = append(evenNumbers, num)
    }
}

fmt.Println("Even numbers:", evenNumbers)
```

This program is objectively much less decipherable than its Python counterpart. For starters, it's properties of being statically typed introduces patterns like $[]int\{\}$ and keywords like *var*. Since we do not have list comprehension, we are forced to iterate the "numbers"-list with a for-loop. "range mubers" yields two values for each instance, the current index and the current value. On top of this, a syntactically correct Go program would require this being inside a function, a main function, and boilerplate code like declaring the package and imports like "fmt". A translation of the Go program would surely be more desired than a translation of the Python program. Sadly, Pseudogen does not

offer this.

It also makes you think, did William Shakespeare really write his entire collection of works in pseudocode?

3.3 Translating code to flowcharts: Code2Flow

also say what is wrong about the flowcharts, show examples of code etc. maybe inconsistencies?

Chapter 4

Design

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

THIS is where I start showing examples of how Psnodig is superior etc. In the problem analysis part, I only showcase what I believe lacks in Pseudogen etc.

Chapter 5

Implementation

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

on its own, psnodig is useless! but when paired with parsers and writers, we see its magic. without, you can only but admire its conciseness and love it.
interpreter. what is the point? you see, not all syntactically correct programs are necessarily also semantically correct (cool graphs on this?? figures of some sort??). show an example!
btw (still on interpreter). this gives yet another edge compared to manually writing pseudocode. it's like JVM: write once, run everywhere! except not quite, but at least you can run the same code you transpile, before bothering to transpile it! if you write bubble sort, try it with a carefully (or perhaps not so) selected list, and it fails to sort your example list, then you probably shouldnt transpile it and use in your class, or present in a final masters thesis.

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
- How easy is it to add another reader/writer? How few lines can this be done in?

This should probably be divided into more parts? Like

Part I - Psnodig
Part II - Gourmet (+ parser?)
Part III.I - Latex-writer (algorithm2e)
Part III.II - Flowcharts (tikz)

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.
- Things I would like to have but didn't have time to implement
- Conclude RQs
- Important: Remind the reader of all the good stuff!

The interpreter has a focus on correctness, and is not particularly optimised for speed. Since it works directly on Psnodig datatypes, it is entirely agnostic to how programs are written. There are also likely edge cases that we have not encountered, as is often the case. The design decisions around things like scoping could be tweaked.

Psnodig ignores types, and types like "list" and "Tree" is only there for the LaTeX-writer, since doing e.g. $\backslash Array1$ without having defined $\backslash KwArray\{Array1\}$ is illegal when working with the $algorithm2e$ library.

The syntax of Psnodig could also have been refined, for instance spacing things out more (as in, instead of having "ForEach" and "For" statements, we could make them their own datatype and simply having a "ForStmt" or something), allowing for more statements and expressions, more types of values, more operations, maybe also allowing programs to be more flexible, allowing global variables etc. These are design decisions that must be thought through thoroughly, and whilst we made decisions based on what we thought was sufficiently powerful and flexible, we *were* to a certain degree limited by the time on our hands, and could have expanded the syntax even more.

It would also be cool to expand Psnodig to include things like lambda functions, as well as classes more similar to object oriented languages. Since input and output languages are totally independent of each other, Psnodig can, in theory, be as rich as one wants, and then people can decide for themselves what their parsers should pick up and what their readers should produce.

We carried out very limited user testing, and did not include any of that in the thesis, since it was usually quite informal. It was done with friends and others who showed interest in the thesis throughout its course, and included things like, how easy is it to write a program in Gourmet? How satisfied are you with the LaTeX equivalent of the code you wrote? How did you find debugging, as in how descriptive and helpful were error messages? Etc.