UNIVERSITY
OF OSLO

Master's thesis

Psnodig: The many faces of code

A tool for converting source code to presentation targets

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

A tool for converting source code to presentation targets

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Research Questions 2
1.3 Contributions 2
1.4 Chapter Overview 3
1.5 Project Source Code 4

2 Background **5**
2.1 Pseudocode 5
2.1.1 Text based pseudocode 6
2.1.2 Image based pseudocode 7
2.1.3 LaTeX 9
2.2 Transpiling 11
2.2.1 Generators 11
2.2.2 Haskell's strengths 13

3 (Problem) Analysis **15**
3.1 The problem at hand 16
3.2 Translating code to natural language: Pseudogen 17
3.3 Psnodig vs. Pseudogen 19
3.4 Translating code to flowcharts: Code2Flow 20
3.5 Psnodig vs. Code2Flow 20

4 Design **23**
4.1 Psnodig 23
4.1.1 Syntax 23
4.1.2 Interpreter 24
4.1.3 Gourmet 24
4.1.4 TBP Writer 24
4.1.5 IBP Writer 24

5 Implementation **29**

6 Evaluation **31**

7 Discussion **33**

8 Conclusion **35**

Listings

2.1 Haskell example to convert data type to string 14

Chapter 1

Introduction

Pseudocode is commonly used to provide a description of an algorithm at a suitable level of abstraction. It is meant to work as a comprimise between a low-level implementation in a specific programming language, and a verbal, natural language description of a problem solution [**LinfoAlgorithmsIntro2007**].

An advantage of pseudocode is the lack of standardisation, therefore authors are not tied down to the syntax of any particular programming language. This gives them complete freedom to omit or de-emphasize certain aspects of their algorithms. Consequently, pseudocode is first and foremost aimed to serve as a tool for presentation.

However, as pseudocode is not executable, there is no omniscient way of verifying its correctness. This can, in turn, lead to accidental inclusion of critical inaccuracies. When writing code in IDEs, programming languages are often accompanied by static analysis tools that detect anti-patterns and warn about practices [**linter**]. Psudocode writers, on the other hand, are left to their own devices, as there can be no anti-patterns or bad practices without standardisation.

1.1 Motivation

Correct presentations are important in education, including at university level, 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 additionally their first impression of an algorithm is an incorrect presentation, their path is already hampered. One of our research questions is: can we transpile source code to pseudocode, whilst maintaining a similar level of abstraction?

In this thesis we present a tool we call **Psnodig** (pronounced snoo-dee), which allows for transpiling source 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 imperative programming language syntax, like Go or Python.

Another of our research questions is: can we transpile source code to flowcharts whilst maintaining a similar level of abstraction? The positive effect of teaching algorithms with flowcharts as an alternative to traditional code has been researched since at least the 1980's and is still being researched this decade [**DBLP:journals/software/Scanlan89**][**7096016**][**flowchartsHighschool**], yet direct translation from source code to flowcharts does not seem to be 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 authors wanting to present their algorithms, as well as students wishing to get a better understanding of them.

We aim to promote algorithmic thinking through various forms of representation. We believe that this can aid in better understanding and modification of code, which in turn can lead to more efficient and effective programming practices. By not having to worry about syntactic intricacies, the audience can focus entirely on logic underlying the algorithms.

1.2 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 abstraction?
>
> **RQ2:** Can we transpile source code to flowcharts whilst maintaining a similar level of abstraction?
>
> ~~**RQ3:** Can people in academia find such a tool to be helpful in better understanding small-to-medium sized code chunks?~~ Keep, replace or remove entirely?

(så er spørsmålet: hvor enkelt er det egentlig å skulle måle dette? hva vil det i det hele tatt si å ha similar level of abstraction?)

1.3 Contributions

The main contribution of this thesis is the Psnodig tool for transpiling executable source code to various presentation-only versions of said code, to give people an easy and accessible way to look at their code from a different perspective. By using the Psnodig tool, we hope people can spend more time writing code and less time mastering LaTeX libraries, writing boilerplate code and worrying about maintaining multiple sources.

The Psnodig tool is written entirely in the Haskell programming language[1]. The tool comes with a parser for the Gourmet programming language, which offers 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[2]. It is also accompanied by a Gourmet writer, as well as writers for pseudocode and flowcharts in LaTeX, utilising the Algorithm2e package[3] and the TikZ package[4] respectively.

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 abstract syntax tree (AST) representation, so that you can run your code.
- 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 ASTs with text based pseudocode, utilising the Algorithm2e package in LaTeX.
- A writer for presenting ASTs with flowcharts, utilising the TikZ package, also in LaTeX.

1.4 Chapter Overview

Chapter 2 provides a clear definition of pseudocode which we carry with us for the remainder of the thesis, as well as some background on transpiling.

Chapter 3 breaks down the exact problem we are looking at, in addition to analysing selected tools which already offer some of the functionality we aim to contribute with.

Chapter 4 delves further into how we intend to solve the problem we introduce in the previous chapter, and compares our tool to what we believe are the shortcomings of its competitors.

Chapter 5 provides concrete implementation details of Psnodig.

Chapter 6 covers how we evaluated Psnodig, how it really works in practice, strengths, weaknesses, and how we attempted to make sure it actually works as intended.

Chapter 7 discusses how the solution holds up agains the problem, and whether or not it fills the holes we believe exist in the alternatives.

[1]https://www.haskell.org/
[2]https://www.uio.no/studier/emner/matnat/ifi/IN2010/index-eng.html
[3]https://www.ctan.org/pkg/algorithm2e
[4]https://www.overleaf.com/learn/latex/TikZ_package

Chapter 8 concludes the work of this thesis, discussing the research questions and future work.

1.5 Project Source Code

All the source code from the master thesis can be found on Github[5] (NOTE: nå ligger den på uio enterprise-githuben. burde være mulig å overføre den til github.com slik at den forblir tilgjengelig også etter at jeg leverer oppgaven og mister uio-rettighetene :smilefjes:).

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

Chapter 2

Background

This chapter will cover concepts that one should be familiar with in order to fully understand the rest of this thesis. We start by providing a definition for pseudocode, to avoid confusion further down the line. We also discuss transpiling, how other transpilers work, and why Haskell is a good tool for the job.

2.1 Pseudocode

Pseudocode is a technique for describing computer programs in a more abstract way than programming languages allow, ignoring specific syntax and keywords. This can make programs easier to understand for both non-programmers and programmers alike, particularly when working with unfamiliar algorithms [**LinfoAlgorithmsIntro2007**].

Since it does not follow any precise syntax rules, pseudocode is subsequently not executable. This is not a bug, but rather a feature of pseudocode: it is intended for presenting ideas of code, not demonstrating results of code [**LogicsofSpecificationLanguages**]. As such, pseudocode is an abstract concept, and can technically be anything, as long as it aims to aid others in understanding what a particular piece of code does.

When explaining a solution to a non-technical audience, the use of pseudocode is standard practice. Specifically, the pseudocode should encapsulate the crucial elements or the core functionality of the program. This focused presentation provides clarity on the essential aspects of the solution. Thus, even individuals without a programming background can provide feedback based on their understanding of the problem and its proposed solution.

Now, since pseudocode has many faces, we must define what we percieve pseudocode to be in the context of this thesis, and what exactly we mean when we refer to “pseudocode” in later parts of the thesis. To avoid further confusion, we will make a distinction between two particular types of pseudocode: text based- and image based pseudocode.

2.1.1 Text based pseudocode

The most common form of pseudocode is likely text based pseudocode (TBP), commonly found in text books on algorithms, published papers, as well as informal scribbling before attempting to solve a problem [**payAttentionToMLPs**][**BOOK:intro/Cormen**
It is also the form that most closely resembles source code, given that it usually includes line numbers, assign statements and generally presents the problem solution in an imperative matter [**proposalForParadigmGeneralPseudocode**].

Since there is no proper set of rules commanding how text based pseudocode should look like, we are prone to viewing different variations of the same algorithms across different literatures. A frequently presented algorithm is Binary Search, which in the context of computer science is a search algorithm that finds the position of a target value within a sorted array [**BOOK:intro/Cormen/Leiserson**].

In a note made for the Algorithmic Problem Solving course at the University of Waterloo, professor Naomi Nishimura has made a note[1] where she presented four different variants of the Binary Search algorithm, all written in pseudocode. The algorithms are written with a total interval of 26 years from the oldest to the newest.

The oldest variant is from 1974, presented in The Design and Analysis of Computer Algorithms by Aho et al. [**BOOK:DesignAnalysis/Aho**]:

```
procedure SEARCH(a, f, l):
if f > l then return "no"
else
    if a = A[⌊(f + l)/2⌋] then return "yes"
    else
        if a < A[⌊(f + l)/2⌋] then
            return SEARCH(a, f, ⌊(f + l)/2⌋ - 1)
        else return SEARCH(a, ⌊(f + l)/2⌋ + 1, l)
```

Then, roughly 17 years later, Lewis et al. present it like this in Data Structures and Their Algorithms [**BOOK:DSA/Lewis**]:

```
function BinarySearchLookUp(key K, table T[0..n-1]): info
{Return information stored with key K in T, or Λ if K is not in T}
    Left ← 0
    Right ← n - 1
    repeat forever
        if Right < Left then
            return Λ
        else
            Middle ← ⌊(Left + Right) / 2⌋
            if K = Key(T[Middle]) then return Info(T[Middle])
            else if K < Key(T[Middle]) then Right ← Middle - 1
```

[1]https://student.cs.uwaterloo.ca/ cs231/resources/pseudocode.pdf

```
                    else Left ← Middle + 1
```

The desire for automatic generation of TBP has been in the wind for some time, with the intention of presenting ideas without having to worry about syntax of a particular programming language [**desireToGetPseudocodeGeneration**]. Text based pseudocode allows authors to draft ideas in an imperative way, just like we write recipes for baking bread and building legos. Here, the author is free to omit boilerplate code, include mathematical notation and necessary abstractions, and even resort to natural language where deemed appropriate [**BOOK:intro/Cormen/Leiserson**][**DBLP:conf/els/Nuallain15**].

As previously mentioned, pseudocode has a well-established history in university curricula. When learning algorithms, data structures, or programming concepts, the focus is really on the underlying ideas. These concepts are generally more important than the specifics of how they are implemented in a specific programming language. Thus, learning with TBP serves to maintain a similar level of abstraction, without demanding familiarity with a particular programming language. This approach prioritises concept comprehension over language-specific knowledge.

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. 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.

Its usefulness is also backed by the numerous other attempts at translating source code to TBP in the past [**PSEU:/Kreher/Stinson**][**DBLP:conf/aswec/AlhefdhiDHG18**].

2.1.2 Image based pseudocode

Not all programming languages share the same execution flow. For instance, in VHDL all processes are executed simultaneously[3], whilst rewriting rules in Maude are applied non-deterministically (if multiple rules can apply to a term, any one of them may be chosen in an arbitrary order). Some languages, 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.

This way of executing a program opens up for the possibility of image based pseudocode (IBP), which still includes text, but also complements it with boxes, arrows and perhaps pretty colours. When code stretches over enough lines, code becomes uniform in appearance and challenging to differentiate. By contrast, IBP distinctively isolates each statement, providing more of a birds eye perspective on the program.

[2]https://devpost.com/software/code-kindle

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

In fact, images in computer science is nothing new. One of the most notable examples we have are the ones we use for finite state automata (FSA). An FSA is a machine which either accepts or rejects a given string, by running each symbol through a state sequence uniquely determined by said string. We differentiate betwee deterministic and non-deterministic FSAs, though it is not of importance in our context. What they share, is a number of states, a start state, a transition function and an accept state [**introToAutomataTheory**].

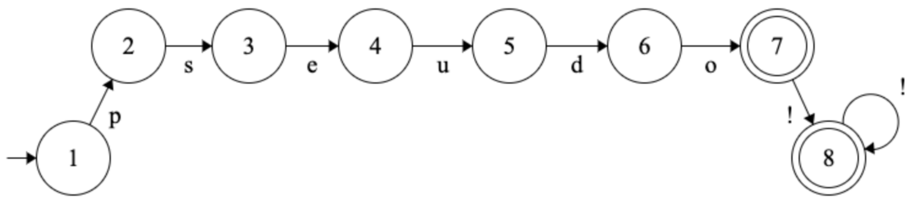


Figure 2.1: An example finite state automata.

Figure 2.3 shows an example of an FSA which accepts the word “pseudo” followed by an arbitrary amount of exclamation marks. The FSA has 8 states, and the leftmost arrow indicates that **1** is the starting state. From here, we can get to the second state if our string starts with the symbol “p”. Thus, all strings that do not begin with a “p” are rejected at this point. States 7 and 8 have an additional ring within their circle, which means that they are accepting states. If a combination of symbols have not been rejected at this point, and is finished, it is accepted. State 8 has an arrow leading to itself via the symbol “!”, meaning that it can end with as many exclamation marks as possible.

A string like “pseudo!!p!!!” is not accepted, however, despite starting with “pseudo!!” and ending with “!!!”. Once a string has reached state 8, it can *only* be followed by exclamation marks, or else it is rejected.

Warren McCulloch and Walter Pitts were among the first researchers to introduce a concept similar to finite automata, all the way back in 1943 [**McCulloch43**]. Their paper presents a simplified computational model of biological neurons.

Throughout the remainder of this thesis, when we talk about “image based pseudocode”, we are referring to flowcharts. The title does not technically belong to flowcharts alone, but it is the way we will go about things.

There have also been multiple attempts at creating flowchart editors, most notably by Carlisle et al. and Charntaweekhun et al. [**carlisle2004**][**charntaweekhun2006**]. These allows authors to visualise their ideas, rather than keeping it all text based. Benefits of learning with help from visual aid is well documented, which is one of the reasons introductory math books are always so colourful [kilde?].

One of few editors that generates flowcharts directly from code is Code2Flow[4].

[4]You can try the editor for free at https://app.code2flow.com/

This is a DSL with support for most common programming concepts like statements, loops, conditionals, and more. The editor comes with a comprehensible guide on its syntax. It is also a highly customisable tool, letting you change the flowcharts' fonts, colours, sizes and even edge height.

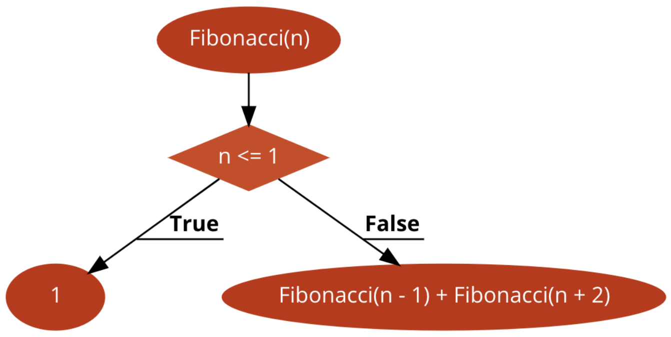


Figure 2.2: Image based pseudocode illustrating an algorithm to retrieve the nth number in a Fibonacci sequence, created with Code2Flow.

Given the imperative nature of flowcharts, the way they walk through problems step-by-step, there have also been attempts at converting image based pseudocode to text based pseudocode. Wu et al. proposed a structure identification algorithm, which can take an identified flowchart as input, and automatically generate code in return [**codeFromFlowcharts**]. This gives even more ground to perceive flowcharts as an image based form of pseudocode.

There have been multiple studies documenting the preference for IBP when it comes to studying algorithms, already back in the 80s by Scanlan et al. He documented how his students overwhelmingly preferred structured flowcharts to pseudocode for comprehending algorithms. Using multiple algorithms of varying complexity, the students most notably indicated that the flowcharts took less time to comprehend, provided fewer errors in understanding, and reduced the number of times they had to look at the algorithms [**DBLP:journals/software/Scanlan89**].

In newer times, Nita et al. attempted to analyse student's understanding of algorithms with pseudocode and flowcharts. The students were subjected to algol-like TBP, and IBP. Their conclusion was that the students found it easier to understand the selected algorithms in image format, as compared to a text based approach [**Nita˙2020**].

2.1.3 LaTeX

LaTeX is a document preparation system that is widely used for the production of scientific documents[5]. It is an open-source typesetting system recognized for its capabilities in creating visually appealing documents that meet typographic

[5]In fact, this thesis is written in LaTeX, through the editor Overleaf.

standards.

Unlike traditional word processors like Microsoft Word or Google Docs, LaTeX operates more similarly to how we write code. The user typesets the document in plain text, with various commands that describe its structure and presentation. All documents must have a “document class”, which is a set of formatting instructions that dictate how a document will look. Additionally, all content in a LaTeX file must be within the commands “\begin{document}” and “\end{document}”.

LaTeX builds upon the TeX typesetting system created by Donald Knuth. It added a collection of macros that simplified the use of TeX, and made it more accessible to non-technical users.

A distributed collection of macros in LaTeX is called a **package**. They allow users to add functionality or modify the behaviour of LaTeX, including refining typography, changing the layout of elements, creating graphics and more. In LaTeX documents, they are included using the **\usepackage{}** command, and placed before **\begin{document}**.

For this thesis, there are two LaTeX packages that are central: **Algorithm2e** and **TikZ**. Algorithm2e is a package to typeset algorithms or pseudocode. TikZ, on the other hand, is probably the most complex and powerful tool to create graphic elements in LaTeX. In this thesis, Algorithm2e will be used for TBP and TikZ for IBP.

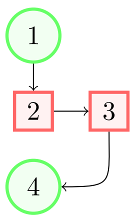
Algorithm2e provides an algorithm environment you can access through **\begin{algorithm2e}**[6]. TikZ provides an environment you can access through **\begin{tikzpicture}**[7]. We must also remember to include **\usepackage{algorithm2e}** and **\usepackage{tikz}** in our preamble.

A simple TBP example with Algorithm2e is provided in footonte 6. An identifier “i” is declared, and a nested control sequence is defined.

```
$i \leftarrow 10$;
if $i \geq 5$ then
  | $i \leftarrow i - 1$;
else
  | if $i \leq 3$ then
  |   | $i \leftarrow i + 2$;
  | end
end
```

In footnote 7 we are introduced to a small example of IBP with TikZ. We see a flow going from one circle, passing two rectangles and ending on another circle. The path is denoted by arrows.

[6]https://www.overleaf.com/learn/latex/Algorithms#The_algorithm2e_package
[7]https://www.overleaf.com/learn/latex/TikZ_package



2.2 Transpiling

The reader might be familiar with the concept of *compiling*. Traditionally, this is a process where a *compiler* reads a program in a high-level language and translates it to an executable target program [**DBLP:books/aw/AhoSU86**].

A transpiler, however, is a tool that converts input source code into output source code, maintaining a similar abstraction level [**DBLP:conf/els/MarcelinoL22**]. The first transpiler to our knowledge was developed in 1978 by Intel, with the aim of translating assembly source code from the 8080/8085 processor to the 8086 processor [**intel1979**].

An example of transpiling involves the JavaScript programming language[8], commonly used in web development. It is a language in constant development, frequently updating its features. The issue with this is that not all browsers are always compatible with its newest features. Therefore, there exists a transpiler Babel[9] which converts modern JavaScript into a backwards compatible version. According to Nicolini et al., without a transpiler almost 14% of web users risk facing a JavaScript bug when accessing a website with new JavaScript features [**DBLP:journals/software/NicoliniHF24**].

Another example is a transpiler presented by Lunnikiv etl. al, where Python is converted to Rust as an intermediate source code step. The paper shows how pre-existing Python implementations that depend on optimised libraries can be transpiled to Rust semi-automatically [**DBLP:conf/samos/LunnikiviJ020**]. This way, the user can keep writing Python, whilst additionally allowing for the performance optimisation given by Rust.

2.2.1 Generators

Given how syntactically rich programming languages tend to be (in that they tend to have many keywords etc), it seems like too difficult of a task to have 1-to-1 mappings from the input source language to the output source language. There are infinite combinations of programs we can write[kilde?], thus we could benefit from some sort of intermediate representation. One of these methods is using so-called *generators*.

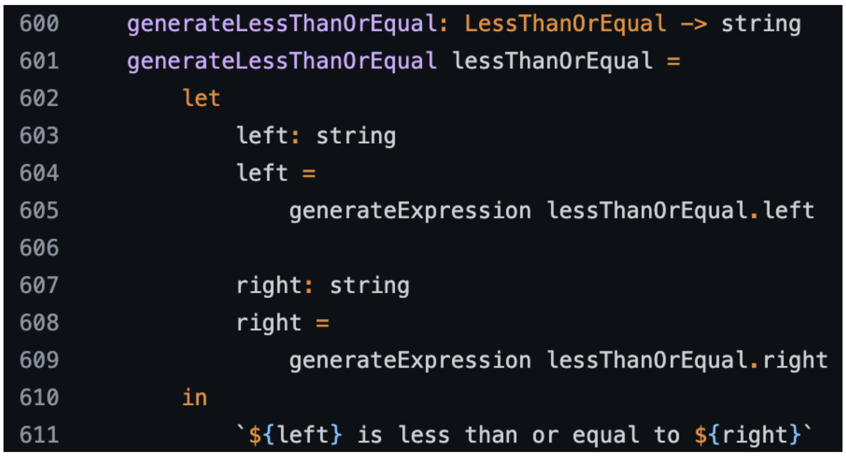
A generator is basically a stand-alone parser or writer. A parser generator translates a program written in a source language to an abstract syntax tree

[8]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview
[9]https://babeljs.io

(AST). A writer generator translates an AST to a program in a target language.

There are several benefits of using generators. One of them is flexibility. Generators can handle a wide range of source code structures and translate them into various target languages. Another one is naturally the modularity. If we wish to add another reader, we can simply implement one and put it “on top” of the rest, without having to modify anything else. Thus we only need to maintain our reader if we wish to change the representation.

An example of this is the programming language Derw[10], an ML language mainly inspired by Elm. By utilising generators[11], it has multiple writers, among others bytecode, JavaScript, and even English. Figure 2.5 shows how expressions like 6 <= 8 are translated. The token **lessThanOrEqual** has a left- and right pointer, corresponding to the respective integers. These are extracted, and put on each side of the string “is less than or equal to”.



```
generateLessThanOrEqual: LessThanOrEqual -> string
generateLessThanOrEqual lessThanOrEqual =
    let
        left: string
        left =
            generateExpression lessThanOrEqual.left

        right: string
        right =
            generateExpression lessThanOrEqual.right
    in
        `${left} is less than or equal to ${right}`
```

Figure 2.3: An excerpt from the english generator in Derw, showing how expressions with “less than or equal” are converted[12].

Another example is Pandoc, which works with markdown languages. It has a “core language” which all parsers and writers must work with:

```
Plain [Inline]
Para [Inline]
LineBlock [[Inline]]
CodeBlock Attr String
RawBlock Format String
BlockQuote [Block]
OrderedList ListAttributes [[Block]]
BulletList [[Block]]
DefinitionList [([Inline], [[Block]])]
```

[10]https://github.com/eeue56/derw
[11]https://github.com/eeue56/derw/tree/main/src/generators
[12]https://github.com/eeue56/derw/blob/main/src/generators/English.derw

```
    Header Int Attr [Inline]
    HorizontalRule
    Table [Inline] [Alignment] [Double] [TableCell] [[TableCell]]
    Div Attr [Block]
    Null
```

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. However, writing a document in a rich format like Latex, and later converting it to a different markup language might tends to pose problems due to the different philosophies that underlie each language. Yet Pandoc is an excellent interpreter of lightweight markup languages like Markdown, which are “neutural” by design [**dominici2014**].

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[13]. 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 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. If we wish to expand our operators data type (with for instance an exponent), we only have to add a pipe and the operator name, like so:

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

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

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 instance, 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)    = " / "
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.

Lastly, we can utilise the QuickCheck[14], which is a testing library suited for automatic property-based testing in Haskell. With this we can prove different properties of our tool [**DBLP:conf/icfp/ClaessenH00**], and perhaps also of other (primarily) parsers and (maybe) writers, given that they have to pass through Haskell ADTs anyway.

[14]https://hackage.haskell.org/package/QuickCheck

Chapter 3

(Problem) Analysis

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

One of our output targets is pseudocode. We are not attempting to create a standardisation or ground truth for pseudocode. 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 LaTeX file and a companion PDF of her work.

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 perspective. This can be refreshing when you have been trying to debug code for an extended period of time with no result [source?].

Currently, to our knowledge, the only freely available tool solely dedicated to converting source code to flowcharts is Code2Flow[2]. This is a DSL with support for most common programming concepts like statements, loops, conditionals, and more. We are aware of other tools which also convert code to diagrams, like Mermaid.js[3] and Diagrams[4], but they are either very general purpose, or focus on different types of diagrams.

[1]https://www.ctan.org/pkg/algorithm2e
[2]https://app.code2flow.com/
[3]https://mermaid.js.org/
[4]https://diagrams.mingrammer.com/

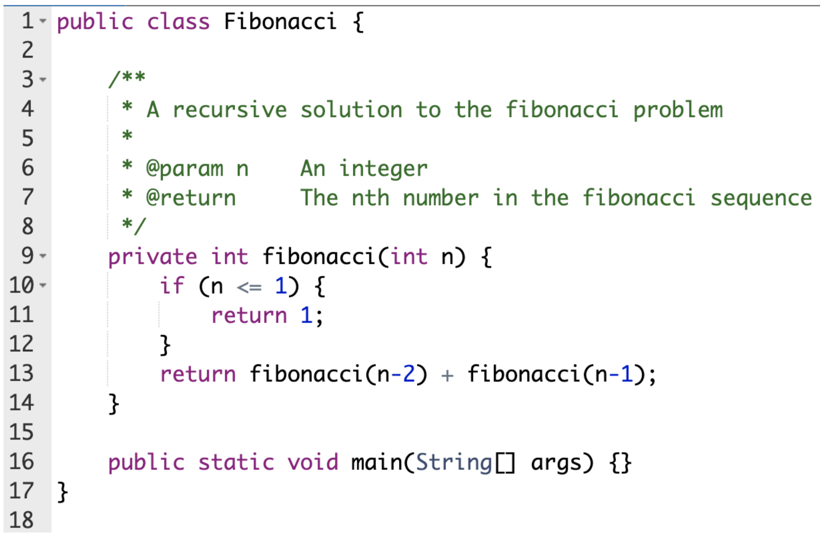
In this chapter, we will be doing problem analysis. We will dive deeper into the problem we intend to fix, and we will discuss what the current solutions lack. While the alternatives we present boast about successfull outcomes, there are some shortcomings when comparing them to what we wish to achieve,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

3.1 The problem at hand

As such, we can comfortably regard the succinctness of figure 2.1 displaying TBP, to figure 2.2 displaying an implementation of said TBP in a popular programming language [**javaIsAPopularProgrammingLanguage**]:

```
Algorithm 1: A recursive solution to the fibonacci problem
  Input: An integer $n$
  Output: The $nth$ number in the fibonacci sequence
Procedure Fibonacci($n$)
    if $n \leq 1$ then
        return 1
    return Fibonacci (n - 2) + Fibonacci (n - 1)
```

Figure 3.1: Text based pseudocode illustrating an algorithm to retrieve the nth number in a fibonacci sequence.



```java
public class Fibonacci {

    /**
     * A recursive solution to the fibonacci problem
     *
     * @param n    An integer
     * @return     The nth number in the fibonacci sequence
     */
    private int fibonacci(int n) {
        if (n <= 1) {
            return 1;
        }
        return fibonacci(n-2) + fibonacci(n-1);
    }

    public static void main(String[] args) {}
}

```

Figure 3.2: An algorithm written in the Java programming language, to retrieve the nth number in a fibonacci sequence. It follows the commenting guidlines JavaDoc[5].

[5]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

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[6] 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)
```

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

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
```

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 us 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 is 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 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[7]. 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 (*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,

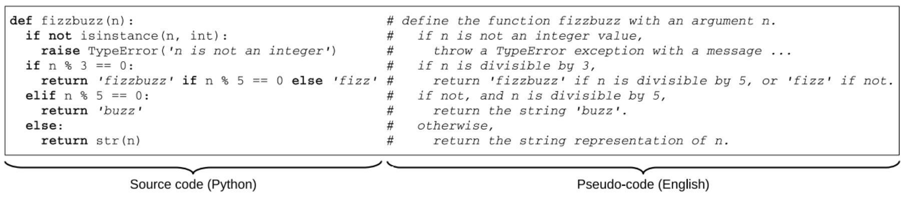
but also programming language specific elements like

raise TypeError('n is not an integer')

[7]https://www.python.org/

is transpiled to

> throw a TypeError exception with a message ...



```
def fizzbuzz(n):                                  # define the function fizzbuzz with an argument n.
  if not isinstance(n, int):                      #   if n is not an integer value,
    raise TypeError('n is not an integer')        #     throw a TypeError exception with a message ...
  if n % 3 == 0:                                  #   if n is divisible by 3,
    return 'fizzbuzz' if n % 5 == 0 else 'fizz'   #     return 'fizzbuzz' if n is divisible by 5, or 'fizz' if not.
  elif n % 5 == 0:                                #   if not, and n is divisible by 5,
    return 'buzz'                                 #     return the string 'buzz'.
  else:                                           #   otherwise,
    return str(n)                                 #     return the string representation of n.
```

Figure 3.3: 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.

3.4 Translating code to flowcharts: Code2Flow

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

3.5 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.

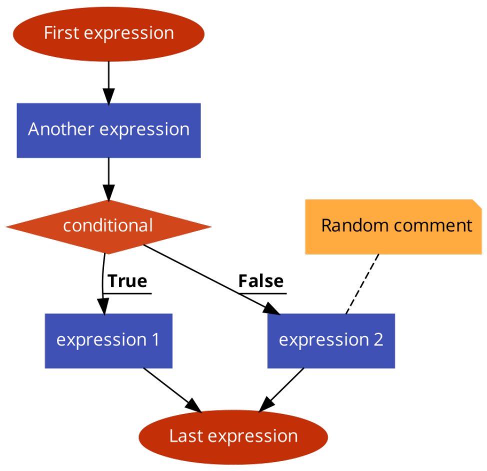


Figure 3.4: Example of a program written with Code2Flow

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.

Chapter 4

Design

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

This is when I start comparing Psnodig to the other tools, showing why Psnodig is preferrable etc.

4.1 Psnodig

4.1.1 Syntax

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.

4.1.2 Interpreter

4.1.3 Gourmet

4.1.4 TBP Writer

4.1.5 IBP Writer

One of the biggest difficulties with the TikZ library, is that we first have to "declare" all of our nodes, before adding edges between them. This means we have to store them somehow, and then later knowing where each edge is coming from and going to. This could be done with a Map.

For instance, imagine you have this program:

```
func Fibonacci(n int) {
    if n <= 2 {
        return 1
    }
    return Fibonacci(n-1) + Fibonacci(n-2)
}
```

This will give us quite a few nodes. I can imagine this:

- func Fibonacci(n int)
- if n <= 2
- return 1
- return Fibonacci(n-1) + Fibonacci(n-2)

Really we get a node per statement, in addition to the function declaration for good measure. Now, nodes with TikZ are written something like this

```
\node (name1) [func] {Fibonacci(n)};
\node (name2) [stmt, below of=name1] {n <= 2};
...
```

As we can see, each node needs their own unique name, so that they can reference each other later. The square brackets denote necessary metadata, like how the node should look like (declared earlier, here we see func and stmt), their position (for instance, name2 is below name1) etc. Lastly, the curly brackets indicate the text withing the box, displayed on the screen. In listing ? we opted for the function name, and the expression inside the if.

Then, after all nodes have been declared, we can start drawing the edges between them, like for instance

```
\draw [arrow] (name1) -- (name2);
\draw [arrow] (name2) -- (name3);
...
```

Thus, we could have a mapping from the function name to the first statement, and thereafter a mapping from that expression to the subsequent expressions. Since we are dealing with an if statement (and followingly an implicit else-statement), a mapping could look something like this

```
name1 -> name2
name2 -> name3
name3 -> name4
name3 -> name5
```

because drawing the edges only really need the labels. The main issue here is that we cannot write the nodes and edges in the same go.

However, when working with large control flows with multiple else-branches, we are met with a new dillemma. Simply putting “a below of=c” and “b below of=c” will but both a and b on the same spot, and the latter will shadow the former.

What we need then is to space them out more evenly, suddenly expanding the metadata to more intricate details like

```
[stmt, below left of=name3, yshift=-0.5cm, xshift=-1.5cm];
```

Additionally, having multiple straight edges from the same source might interfere with each of the edge labels. Thus, we might also have to change the way we draw edges, for instance like this

```
\draw [arrow] (name3) |- (name5);
```

This will curve the edge between name3 and name5, making the flowchart clearer. This is, however, a difficult task to carry out since we never know how programs turn out. Therefore it seems natural to opt for the easiest choice available, and instead, unfortunately, force the authors of tweaking the resulting LaTeX on their own.

Another issue is that all control statements have implicit else branches. Take the Fibonacci code above as an example. Technically, the last return statement is the result of an implicit else in the control statement.

Now, what is actually interesting here (should be in chapt 3 I know), is that Code2Flow is not always too sure about this stuff either. For instance, this code

```
Fib() {
    if a { return x }
    else if b { return y }
    else { return z }

    if c { return k }
}
```

actually yields this flowchart

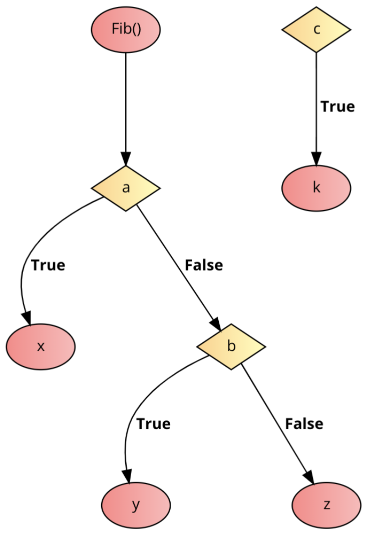


Figure 4.1: IBP from Code2Flow

As you can see, the first control sequence is pointed to by the function name, but not the second. This is because all the branches in the first sequence return something (with an explicit else), thus we cannot ever reach the second one. Therefore, it just dangles in the air.

However, if we remove the explicit else branch, like so

```
Fib() {
    if a { return x }
    else if b { return y }

    if c { return k }
}
```

we see that the last control sequence will still act as some sort of else branch

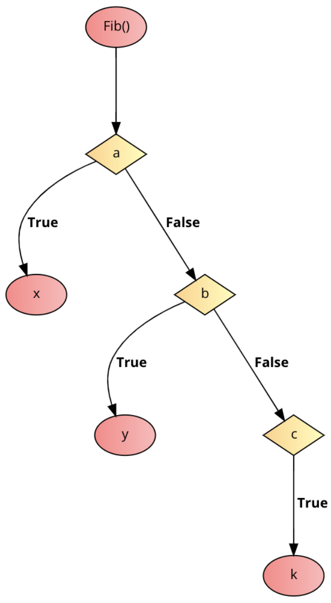


Figure 4.2: More IBP from Code2Flow

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 is 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.

mention some kind of testing with quickcheck, e.g. from gourmet to ast and back to gourmet, to show consistency at least.

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)
when showing flowchart, maybe huge stuff is a bad idea. but we can try to show things like fizzbuzz and fibonacci!

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!

- another future work: even stronger correctness! would be cool to try to revert the produced pseudocode back to source code, thus being even more sure that we are consistent!!!

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.