Master’s thesis

Psnodig: Title WIP

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: Title WIP

A tool for converting source code to presentation targets

Contents

Listings

List of Figures

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 [23].

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 [30]. 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 algorithms and data structures have proved challenging for undergraduates [12]. If their first impression of an algorithm is an incorrect presentation, their path is already hampered.

In this thesis we present a tool **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.

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 [38, 16, 39], 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 [28, p. 14]. Tools like IDEs and linters can only help us so much 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 Goals

The goal of this thesis is to construct a tool with the following properties:

- **Presentable**, the user can lift her source code to a higher level of abstraction.
  - **Modifiable**, the presentation result should be modifiable.
- **Adaptable/Compatible**, the user can add parsers to work with Psnodig.
- **Executable**, the user can run the source code they have written.

`Et litt viktig punkt som vi tilbyr er at brukeren får den tilhørende LaTeX-filen, slik at de kan endre detaljer om de vil. Dette tilbyr (så vidt jeg vet) ingen andre. Jeg har pakket det inn som en underkategori av ‘‘presentable’’, men jeg føler det bør presiseres tydeligere. Eller?`

Standard tools traditionally fulfill one or the other. Programming languages, in which you can write programs and test them, do not necessary have the appropriate abstraction level to be understood by students at all levels. Pseudocode, on the other hand, is intentionally not executable, and thus the presented ideas cannot be tested directly.

The perk of centralising the resources to a single tool makes the job easier for everyone involved.

In chapter 3 we will analyse some tools which perform well in one or more area, but fail to satisfy all three. With Psnodig, we hope to fulfill this task.

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] It comes with a parser for a simple imperative language we call Gourmet, to serve as a proof of concept. The language offers a rich enough syntax for writing all algorithms introduced in the introductory course to algorithms and data structures at the University of Oslo.[2]

The tool is also accompanied by multiple *writers*, which are able to transform a Psnodig abstract syntax tree (AST) to Gourmet source code, as well as pseudocode and flowcharts in LaTeX. The latter two utilise the Algorithm2e[3] and TikZ[4] packages, respectively.

To summarise, the contributions include:

- Psnodig, a tool for transpiling code from one representation to another. It also comes with an interpreter which works on the intermediate 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 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:).

[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
[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 later. 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, void of a predefined set of rules. Authors can ignore specific syntax and keywords, and focus more on getting their ideas across. This can make programs easier to understand for both non-programmers and programmers alike, particularly when working with unfamiliar algorithms [23].

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 [5]. 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 confusion, we delineate between two distinct types of pseudocode: Traditional pseudocode and flowcharts.

2.1.1 Traditional pseudocode

The most conventional form of pseudocode, commonly found in text books on algorithms, published papers, as well as informal scribbling before attempting to solve a problem [25, 11]. 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 [41].

Since there is no proper set of rules commanding how 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 is a search algorithm that finds the position of a target value within a sorted array. If the target value is not found, some sort of default value is usually returned [11].

In a note made for the Algorithmic Problem Solving course at the University of Waterloo, professor Naomi Nishimura presented four different variants of the Binary Search algorithm, all written in pseudocode.[1] 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. [1, p. 139]:

```
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 [22, p. 182]:

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

[1]The note can be found at https://student.cs.uwaterloo.ca/~cs231/resources/pseudocode.pdf

The wish for automatic generation of pseudocode has been desired for some time, with the intention of presenting ideas without having to worry about syntax of a particular programming language [43]. Traditional 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 [11, 34].

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 pseudocode 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 are PseudoEditor[2], Code Kindle[3], and Pseudogen, a tool introduced by Oda et. al [35]. The two latter 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 pseudocode in the past [21, 3].

2.1.2 Flowcharts

Not all programming languages share the same execution flow. For instance, in VHDL all processes are executed simultaneously[4]. In languages with term rewriting, like Maude [9], rewriting rules 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 almost follow the execution flow by just 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 converting source code to flowcharts, which still includes text, but also complements it with boxes, arrows and different colours. When code stretches over enough lines, it becomes uniform in appearance and challenging to differentiate. By contrast, flowcharts capture the control flow of the program explicitly and makes it visually appar-

[2]`https://pseudoeditor.com/`

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

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

ent.

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 [17].

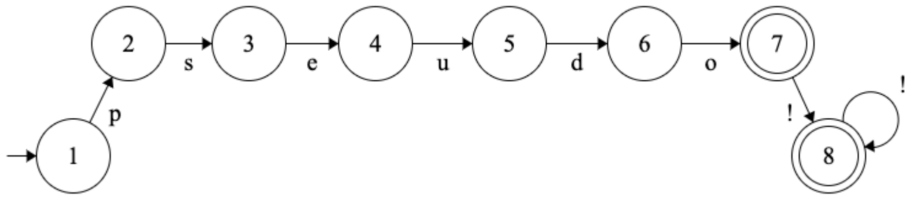


Figure 2.1: An example finite state automata.

Figure 2.1 shows an example of an FSA which accepts the word “pseudo” followed by an arbitrary number 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.

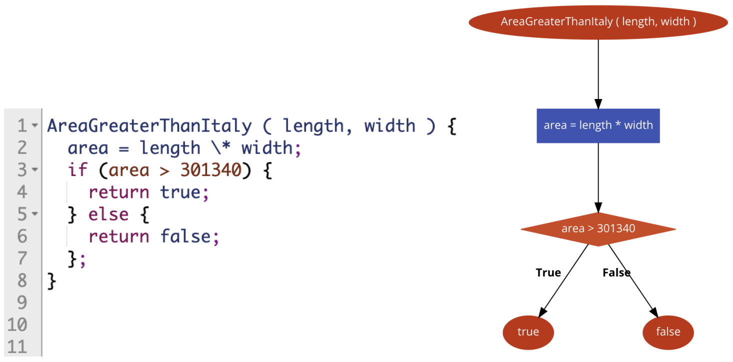
Warren McCulloch and Walter Pitts were among the first researchers to introduce a concept similar to finite automata, all the way back in 1943 [29]. Their paper presents a simplified computational model of biological neurons.

There have been multiple attempts at creating flowchart editors, most notably by Carlisle et al. [6] and Charntaweekhun et al. [7]. 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. When it comes to computer science, visualisations are especially common in the context of machine learning [44, 33, 15].

One of few editors that generates flowcharts directly from code is Code2Flow[5].

[5]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.



```
AreaGreaterThanItaly ( length, width ) {
  area = length \* width;
  if (area > 301340) {
    return true;
  } else {
    return false;
  };
}
```

(a) Code2Flow source code (b) Compiled result

Figure 2.2: An algorithm written with Code2Flow, and the resulting flowchart

Given the imperative nature of flowcharts, the way they walk through problems step-by-step, there have also been attempts at converting flowcharts to pseudocode. Wu et al. [42] proposed a structure identification algorithm, which can take an identified flowchart as input, and automatically generate code in return. This gives even more ground to perceive flowcharts as an image based form of pseudocode.

There have been multiple studies documenting the preference for flowcharts when it comes to studying algorithms, already back in the 80s by Scanlan et al. [38] 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.

More recently, Nita et al. [32] attempted to analyse student's understanding of algorithms with pseudocode and flowcharts. The students were subjected to algol-like pseudocode and flowcharts. Their conclusion was that the students found it easier to understand the selected algorithms in image format, as compared to a text based approach.

2.1.3 LaTeX

LaTeX is a document preparation system that is widely used for the production of scientific documents[6]. It is an open-source typesetting system recognized for its capabilities in creating visually appealing documents that meet typographic

[6]In fact, this thesis is written in LaTeX.

standards.

LaTeX operates similarly to traditional programming, as it requires the user to write code to produce a document. The user typesets the document by typing commands in plain text, specifying the structure and styling of the content. This code is then compiled (see more in Section 2.2) to produce a formatted document, typically in PDF format.

This is a contrast to more ubiquitous word processors like Microsoft Word or Google Docs, which abide to WYSIWYG principles. This means that they display the final product as it is being edited, and allow users to manipulate the document directly through the GUI.

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.

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 pseudocode and TikZ for flowcharts.

2.1.4 Examples

Figure 2.3 shows how we can use the Algorithm2e package with LaTeX to write pseudocode. Figure 2.3 (a) shows the source code, and Figure 2.3 (b) shows the result of compiling said source coude.

The first lines show how algorithm2e can be loaded and cofigured. Some keywords must be declared, like **\SetKwProg{Prog}{Title}** to denote title of our algorithm. **\Prog** is what we write, and **Title** is what we see. Declaring a function keyword is optional, and all it really does is add a monospaced font. We can also opt to exclude semicolons.

We can add a description of input and output through **\KwIn** and **\KwOut**, respectively. We can also add a caption to the algorithm as a whole with **\caption** at the end.

We can write the algorithm itself with natural language, but also utilise the embedded keywords of Algorithm2e. In the example we have used **\uIf**, **\uElse** and **\Return**, as well as mathematical symbols like $\leftarrow$ and $\cdot$ by wrapping them in dollar signs.

```latex
\documentclass{standalone}
\usepackage[linesnumbered, ruled]{algorithm2e}
\SetKwProg{proc}{Procedure}{}{}
\SetKwFunction{AreaGreaterThanItaly}{AreaGreaterThanItaly}
\DontPrintSemicolon
\begin{document}
\begin{algorithm}[H]
\KwIn{Two numbers $length$ and $width$}
\KwOut{A boolean revealing wether the input's area is greater than that of the Italian Republic}
\proc{$\AreaGreaterThanItaly(length, width)$}{
    $area \gets length \cdot width$ \;
    \uIf{$area > 301340$}{
        \Return true
    } \uElse{
        \Return false
    }
}\caption{Algorithm to determine if area is greater than Italy}
\end{algorithm}
\end{document}
```

(a) LaTeX source code

```
Algorithm 1: Algorithm to determine if area is greater than Italy
  Input: Two numbers $length$ and $width$
  Output: A boolean revealing wether the input's area is greater than
          that of the Italian Republic
Procedure AreaGreaterThanItaly($length, width$)
    $area \leftarrow length \cdot width$
    if $area > 301340$ then
        return true
    else
        return false
```

(b) Compiled result

Figure 2.3: A program written in LaTeX, using the Algorithm2e package

Figure 2.4 shows how we can use the TikZ with LaTeX to draw flowcharts. Figure 2.4 (a) shows the source code, and Figure 2.4 (b) shows the result of compiling said source coude.

Again, the first lines show how the package can be loaded and configured. After loading TikZ, we specify what we want to use from the package, which in this case is **shapes** and **arrows**. We also specify the node distance, in centimeters.

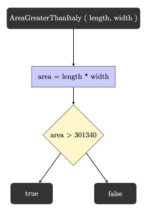
Flowcharts with TikZ are constructed in three primary steps:

1. Defining the style of nodes and edges
2. Declaring the nodes
3. Declaring the edges

Each node style has a label and a shape, along with more specific metadata like height and colour. When declaring nodes in the next step, we give them an identifier, a style and the text which occurs in them. When declaring edges, we can simply point out the style, the child and the parent. Each edge must be drawn individually. In this example the edges have direction, but this can be changed by omitting the `->` option on line 10.

```
1  \documentclass{article}
2  \usepackage{tikz}
3  \usetikzlibrary{shapes}
4  \begin{document}
5  \begin{tikzpicture}[node distance=2.8cm]
6
7  \tikzstyle{startstop} = [rectangle, rounded corners, minimum width=2cm, minimum height=1cm, text centered, draw=black, text=white, fill=black!80]
8  \tikzstyle{statement} = [rectangle, minimum width=4cm, minimum height=1cm, text centered, draw=black, fill=blue!20]
9  \tikzstyle{decision} = [diamond, text centered, draw=black, fill=yellow!30]
10 \tikzstyle{edge} = [thick, ->, >=stealth]
11
12 \node (start) [startstop] {AreaGreaterThanItaly ( length, width )};
13 \node (decl) [statement, below of=start] {area = length * width};
14 \node (if) [decision, below of=decl] {area $>$ 301340};
15 \node (true) [startstop, below of=if, xshift=-2cm] {true};
16 \node (false) [startstop, below of=if, xshift=2cm] {false};
17
18 \draw [edge] (start) -- (decl);
19 \draw [edge] (decl) -- (if);
20 \draw [edge] (if) -- (true);
21 \draw [edge] (if) -- (false);
22
23 \end{tikzpicture}
24 \end{document}
```

(a) LaTeX source code



(b) Compiled result

Figure 2.4: A program written in LaTeX, using the TikZ package

2.2 Haskell

As previously mentioned, we opted for the Haskell programming language to implement Psnodig. Knowing the ins and outs of Haskell is not crucial for understanding the thesis. However, there are some aspects of the language that are key to the implementation, and majority of the provided listings in this thesis will be in Haskell.

2.2.1 Data types

Types in Haskell are also called **data types**, as the **type** keyword is used to create type aliases.

A common type in any programming language is the boolean, shown in Listing 2.1. All types have one or more **value constructors**, which specify the different values a certain type can have. In this case, the Boolean type can have one of two values: True or False. The pipe operator functions as an “or” [24, be stian om å sjekke hvilken side dette eksemplet er på i boka].

```
data Boolean = False | True
```

Listing 2.1: Recreating the Boolean data type with Haskell

With Haskell, it is straightforward to create your own data types, which can then be used to model ASTs. For instance, we can create our own calculator language in just a few lines of code, as shown in Listing 2.2.

```
data Program = Program Expression

data Expression =
      CompoundExpression Integer Operator Expression
    | IntExpression Integer

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

Listing 2.2: Example data types in Haskell

From this, we can construct an AST as the one we see in Listing 2.3.

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

Listing 2.3: An AST constructed with data types presented in Listing 2.2

Naturally, we could create much bigger calculations than this. If we wish to expand our operators data type, we only have to add a pipe and the operator name, as shown in Listing 2.4.

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

```
        | Exponent
```

Listing 2.4: An extended version of the Operator data type presented in Listing 2.2

2.2.2 Pattern matching

Another integral part of Haskell, is that its strong type system opens for clean and efficient pattern matching. This is a very useful method for deconstructing and working with data, and for making decisions based on the data's shape.

Pattern matching is demonstrated in Listing 2.5, with a function converting each value of the Operator type (introduced in Listing 2.2) to its string equivalent. It shows a function `convert` that takes a value of type Operator as input, and returns a value of type String.

```
convert :: Operator -> String
convert Plus     = " + "
convert Minus    = " - "
convert Times    = " / "
convert Division = " * "
```

Listing 2.5: Haskell function converting values of one type to another

If the function shown in Listing 2.5 was to work on the extended Operator data type from Listing 2.4, the compiler would let us know that our pattern matching is in, as there is no case for the `Exponent` value. This is one of the features that make pattern matching so powerful and safe. For some types though, like `Integer`, it can be exhausting to define injective functions. Listing 2.6 shows how the underscore can be used to capture all remaining values of a type.

```
convertInt :: Integer -> String
convertInt 1 = "one"
convertInt 2 = "two"
convertInt _ = "in integer other than one or two"
```

Listing 2.6: A simple Haskell function converting values of type Integer to its string equivalent

2.2.3 Testing

Lastly, Haskell allows us to utilise the QuickCheck, which is a testing library suited for automatic property-based testing.[7] QuickCheck can be used to prove various properties of our tool [8].

[7]Full documentation can be found at `https://hackage.haskell.org/package/QuickCheck`

2.3 Compilers

A compiler is, in simple terms, a tool that reads a program in a high-level language and translates it to an executable target program [2]. It consists of a frontend and a backend. The frontend is often referred to as the analysis part, whilst the backend is referred to as the synthesis part.

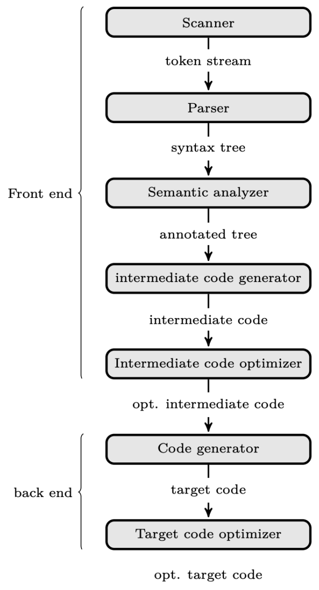


Figure 2.5: The phases of a typical compiler.[8]

The frontend of a compiler is responsible for reading the character stream of a source program, and converting them into appropriate tokens. These tokens are then used to create an intermediate representation of the source program. It is during the analysis part that a compiler will detect a program's syntactic errors, if there are any.

Often, the analysis part involves a symbol table, which maintains information about syntactic entities of the source program. This is passed along with the intermediate representation to the synthesis part, for optimisation reasons. Common entities are bindings and typing.

The backend of a compiler is responsible for producing the desired target program from the intermediate representation. This target program is intended to be executable. For instance, source code written in C is compiled down to an executable binary.

[8]The image is borrowed by Martin Steffen's script on compiler construction: `https://www.uio.no/studier/emner/matnat/ifi/INF5110/v24/script/`

2.3.1 Source-to-source compilers

A transpiler, formally **source-to-source compiler**, is a tool that converts input source code into output source code, whilst maintaining a similar abstraction level [27]. 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 [19].

The JavaScript programming language[9] has a rich history of transpiling. As a language in constant development, it faces the issue where not all browsers are always compatible with its newest features. Therefore, there exists a transpiler Babel[10] which converts modern JavaScript into a backwards compatible version. According to Nicolini et al. [31], without a transpiler almost 14% of web users risk facing a JavaScript bug when accessing a website with new JavaScript features.

Not only JavaScript can be transpiled to JavaScript. In fact, the list of other programming languages and tools that can be transpiled to JavaScript is so extensive that it is potential for its own thesis.[11] However, we can bring forward a few notable exambles.

Unlike JavaScript, TypeScript is structurally typed. TypeScript is syntactically a superset of JavaScript, as it adds a static typing layer. The primary purpose of these types is to enhance the development experience by catching potential errors during compilation, and making the code more maintainable. However, before the code is run, TypeScript is transpiled into plain JavaScript, and the types are stripped away [4].

AlaSQL is an open-source SQL database for JavaScript.[12] It is technically a transpiler since we work in a JavaScript environment, but it allows you to write CRUD operations in SQL, like displayed in Listing 2.1.

```
alasql("CREATE TABLE
        cities (city string, pop number)");

alasql("INSERT INTO cities
        VALUES ('Paris', 2249975),
               ('Berlin', 3517424),
               ('Madrid', 3041579)");

const res = alasql("SELECT *
                    FROM cities
                    WHERE pop < 3500000
                    ORDER BY pop
                    DESC");
```

[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview
[10]https://babeljs.io
[11]https://gist.github.com/matthiasak/c3c9c40d0f98ca91def1 provides a list of 320 languages and tools that compile to JavaScript.
[12]https://github.com/AlaSQL/alasql

Listing 2.7: JavaScript code to create, populate and select a table with AlaSQL

Despite all the commands being written in SQL, `res` has a JavaScript value of a list with two objects, as seen in Listing 2.2.

```
[
    {
        "city": "Madrid",
        "pop": 3041579
    },
    {
        "city": "Paris",
        "pop": 2249975
    }
]
```

Listing 2.8: JSON list with two objects

JavaScript is practically the only Turing-complete programming language that can be used across browsers for web development. Rather than reinventing the wheel, developers have created transpilers to compile their favourite languages to JavaScript, which allows them to write code in their preferred languages, also for web development. Notable examples inlcude GopherJS, Scala.js and Opal, which transpile Go, Scala and Ruby, respectively, to JavaScript.

`https://ieeexplore.ieee.org/abstract/document/9930246/references#references` `refererer til AlaSQL-githuben som en kilde, istedenfor fotnote. Kan jeg gjøre det samme? Bør fotnoter egentlig bare være digresjoner og sånt? I så fall blir det ganske mange slike kilder, da jeg linker til ting som Babel, men også emnesider i kap. 4.3! Jeg ser forskjellige måter å gjøre det på i ulike artikler. Jeg er redd for å ‘‘forsøple’’ teksten med for mange fotnoter..`

Transpilation is not exclusive to JavaScript, however. It is a common practice in many other programming languages that must interact with or be portable across diverse systems. For instance, Haskell uses GHC (Glasgow Haskell Compiler) to compile its code, which at one point converted its code into C rather than direct generation of native code. This enabled Haskell to run on any platform with a C compiler. It also benefits directly from others' improvements in C code generation [36].

Another example is a transpiler presented by Lunnikiv et. al [26], 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. This way, the user can keep writing Python, whilst additionally allowing for the performance optimisation given by Rust.

2.3.2 Parsers and code generators

We remember that the frontend of a compiler is tasked with reading source code, and — given that it is syntactically correct — build an intermediate representation. The backend of a compiler is tasked with converting that intermediate representation into target code.

Having in depth knowledge about the entire pipeline of a compiler is not necessary to understand the rest of the thesis. However, the first and the last parts of a compiler will be central topics. Specifically I am referring to the parser and the code generator.

These two play a vital role in a transpiler. In fact, they are all you really need to build a simple transpiler (in addition to a defined intermediate representation). When the parser has converted the source code to an intermediate representation, the code generator can convert that intermediate representation into target code. Given the fact that even the simplest language could write infinitely many different programs, we must lean on some kind of intermediate representation.

Technically, the parser and the code generator are completely independent from one another. The only thing they must have in common is the ability to read-/write the same intermediate representation. There are several advantages to this, like flexibility and modularity. If we want our transpiler to read or write another language, we can just create an additional parser or code generator. When we add a parser, we do not have to do any changes to our code generator, and vice versa, because they work on the same intermediate representation, independent of how the source- and target programs look like.

An example of this is the programming language **Derw**, an ML language mainly inspired by Elm.[13] Its compiler can only parse Derw code, however it comes with multiple code generators (referred to as just "generators") which as of writing this target JavaScript, TypeScript, Elm and even English and Derw itself. As it is open sourced, anyone can fork the repository and add their own code generator, if they wish.

Listing 2.3 shows how expressions like 6 <= 8 are converted. 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".[14]

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

[13]https://github.com/eeue56/derw

[14]The entire English code generator can be found at https://github.com/eeue56/derw/blob/main/src/generators/English.derw

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

Listing 2.9: The function that converts a "less than or equal"-expression in Derw to English

Another example is **Pandoc**, which is a software that converts between different markdown formats [13]. It includes a Haskell library, as well as a command-line program. It is able to a document from 45 source formats to 63 target formats.[15] Additionally, Pandoc is able to convert documents in LaTex, Groff ms and HTML into PDFs.

At its core, Pandoc is really just an abstract syntax tree (AST) of Haskell data types. A Pandoc document has the type **Pandoc Meta [Block]**. The first attribute is *Meta*, metadata for the document, like its title, its author(s), the date it was written and more. The second attribute is a list of *Block*. A block is a more intricate data type, which is shown in its entirety in Listing 2.4.

```
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
```

Listing 2.10: The "Block" data type of Pandocs native representation

The modular design of Pandoc means that adding an input or output formats only requires adding a program that can convert *to* this native representation, and a program that can convert *from* this native representation. This is much like parsers and code generators of compilers, but as they are much less intricate, the Pandoc documentation refers to them simply as **readers** and **writers**.

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. As the native representation is less expressive than many of the formats it converts between, the user cannot always expect perfect conversions. Yet Pandoc is an excellent interpreter of lightweight markup languages like Markdown, which are "neutural" by design [13].

[15]See the entire list at `https://hackage.haskell.org/package/pandoc`

An example of conversion with Pandoc is provided in Figure 2.6. The input format is Markdown, and the output format is LaTeX. Figure 2.6 (b) shows the intermediate, native representation in Haskell. Meta contains title and date, while the list entries in Block are of type Header and Para.

```
---
title: Programming languages
date: 2023-02-01
---

# Introduction

This is a paragraph about programming languages
```

(a) A Markdown program

```
Pandoc
  (Meta {unMeta = fromList
    [ ("title", MetaInlines [Str "Programming",
                             Space,
                             Str "languages"])
    , ("date", MetaInlines [Str "2023-02-01"])
  ]})

  [ Header 1 ("introduction", [], []) [Str "Introduction"],
    Para [Str "This", Space, Str "is", Space,
          Str "a", Space, Str "paragraph", Space,
          Str "about", Space, Str "programming",
          Space, Str "languages"]
  ]
```

(b) The internal Pandoc AST of (a)

```
...

\title{Programming languages}
\author{}
\date{2023-02-01}

\begin{document}
\maketitle

\section{Introduction}\label{introduction}

This is a paragraph about programming languages

\end{document}
```

(c) A LaTeX program, built from the AST in (b)

Figure 2.6: A Markdown program converted to LaTeX with Pandoc

Chapter 3

Analysis

In this chapter, we will be doing problem analysis. We will dive deeper into the problem we intend to fix, and why we think it is a problem in the first place. We will discuss various existing approaches that already solve part of the problem, and why these are great efforts. We finish by discussing why believe that the existing approaches fall short of what we aim to achieve.

3.1 Problem Definition

Now is as good a time as any to outline the problem at hand more concretely. Simply put, we aim to present computer programs at different abstraction levels, without losing the underlying ideas. This can be - and traditionally also has been - done "manually" by the program authors. For instance: writing a computer program, testing it, and later rewriting the source code to some sort of pseudocode. We believe that this process could be automated to some extent, to the benefit of all involved.

There are several use cases where displaying source code with an alternative representation could be useful. Mainly within education, where the goal is to teach students concepts in an agonstic way. However, it could also be used by researchers exchanging ideas, across preferred programming languages.

Already in the 80's Clements et al. predicted that computers would be an integral part of the classroom [10]. Slowly but surely also the art of computer programming has been introduced into school curriculums, at a younger and younger age.

Instructing 10-year-olds in complex concepts like pointers and multi-threading may not be the most effective use of time. However, there is value in familiarizing them with computer programming, which is a fundamental aspect of our increasingly digital world.

If the teacher is to transpile her code to flowcharts, it offers a visual approach to understanding programming concepts. Consequently, the whole classroom can focus on the underlying logic than on the syntactic quirks of the teacher's preferred programming language.

This is also the case when teaching computer programs at a higher level, like university. Opting for pseudocode instead of a specific programming language levels the playing field, as well as helping students coming from a mathematical background.

Lastly, researchers might wish to exchange ideas despite using different programming languages. Just like every animal in the kingdom can be called upon in latin, every idea can be presented more neutrally with pseudocode.

`For tynt eller ok?`

In the context of our problem, and the remainder of this thesis, we choose to also treat flowcharts as a form of pseudocode. As such, we have to make a distinction between flowcharts and traditional pseudocode, which more closely resembles source code. From now, when discussing both forms in the same context, traditional pseudocode will be referred to as **Text based pseudocode** (TBP) and flowcharts as **Image based pseudocode** (IBP).

3.2 Comparing source code with TBP and IBP

In this section we aim to show how the same computer program can be presented in three different levels of abstraction. The first one will be source code written in a popular programming language, while the latter two will be TBP and IBP.

The program in question `beskrivelse programmet. Mulig bucket sort eller en tre-algoritme kan egne seg her.`

Listing ?? shows the program written in the programming language Go/Java. Figure ?? and Figure ?? show the same program, but with TBP and IBP, respectively.

The main similarity between the source code and the TBP is that both versions resemble an executable program. People familiar with programming will likely understand the logic conveyed by the pseudocode, even if they have no prior experience with the Algorithm2e library. Both versions present each instruction as sequential lines, and employ syntax like `if` and `else`. `Forklare likhetene nærmere?`

The main difference is the format. For one, the TBP version describes the input and output, which allows us to have rather succinct parameter names. In the Go/Java example, even though the parameter names are explanatory, we cannot always be 100% sure about how the author intends for them to be applied. This is not an issue with TBP.

Another thing to notice is the difference in notation. While in Go/Java we have to write terms like `!=` and `ceil(x)`, TBP allows us to use more precise mathematical notation like $\neq$ and $\lceil x \rceil$.

Additionally, since the TBP is intentionally not executable, we allow ourselves to omit implementation specific details that are not crucial to understand the program's logic. This makes for a more concise presentation, minimising the chance of confusion by whoever is trying to understand the program.

The same can be said for the IBP version, which does not particularly resemble the original source program. In appearance, they are on very different abstraction levels. Whereas the code is written in sequential lines, the flowchart shows colourful shapes wandering off in different directions.

The spacing and colourfulness of IBP can make it easier to isolate parts of the program, and to see more precisely how each part works individually. It also makes it easier to follow each path of the program, compared to the Go/Java program, where the order of function calls and scoping of if-statements can confuse even more seasoned programmers.

The notation stays much the same, but since the flowcharts are not executable, we can again opt for more precise mathematical notation to describe expressions, and exclude static properties like types.

3.3 Related work

This section will cover selected related software, that has already solved parts of the problem in its own right. We have separated the section into two further subsections, to analyse contributions related to TBP and IBP individually.

3.3.1 Source code to TBP

Despite TBP being used in so many text books, online courses and published papers, the amount of source code-to-pseudocode-editors currently available on the internet is anything but overwhelming. Additionally, there is no undisputed choice that is commonly used. There are, however, a couple of candidates who stick out if we look closely enough.

Naive approach

The naive approach would be to do this manually. Perhaps the wording is too harsh, but in this context, `naive` really just corresponds to `manual`. By manually writing our own pseudocode, we are void of any restrictions, and can do it just the way we like.

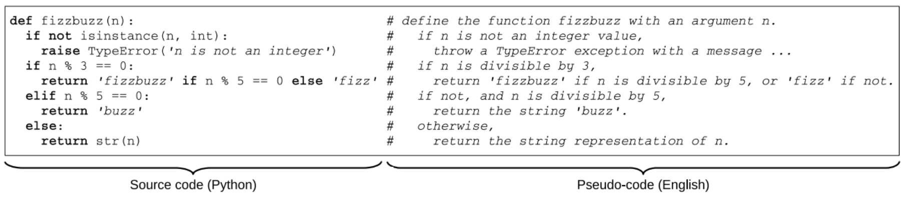
The downside is the extra effort of writing both the original algorithm, in addition to spending time on writing the pseudocode. Doing it this way means we also have to maintain both versions manually. We might change our source code but forget to update the pseudocode, and we are likely to spend some time on trying to abstract the pseudocode in the first place.

Pseudogen

The Psuedogen transpiler is currently designed to work with a subset of the Python programming language [35]. The output target is purely natural language, precisely like majority of this thesis. As mentioned in Section 2.1.1, the tool is developed using statistical machine translation. This is a technique of translating from one language to another based on statistical models, and is generally used to translate between natural languages (which makes Python such a natural choice).

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 unfamiliar with programming and/or mathematics might still struggle to understand some of the more technical aspects of its syntax.

Figure 3.1 shows an example taken from the initial 2015 paper where Pseudogen was first presented. It displays Python source code on the left, and pseudocode on the right. The program in question is an algorithm that solves the **FizzBuzz** problem, commonly presented in entry level interview settings and beginner programming exercises.



```
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 3.1: Example of source code written in Python and corresponding pseudocode created by Pseudogen

What the examples in the 2015 paper, as well as a video on their website show, is really a line-for-line translation to English. [1] 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 (without having to refactor the code base to Cobol).

However, since Pseudogen will translate each line in a servile manner, all error handling is translated too. Listing 3.1 shows some error handling in Python, and Listing 3.2 shows the result from transpiling it with Pseudogen. The result is overwhelmingly verbose, which also defeats much of the point with Python, whose syntax tends to be elegant and succinct, already closely resembling natural language.

[1]The Pseudogen website can be found at https://ahclab.naist.jp/pseudogen/

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

Listing 3.1: Error handling in Python

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

Listing 3.2: The result of transpiling the code in Listing ?? with Pseudogen

Another example is visible in Listing 3.3 and Listing 3.4, where list comprehension has been translated very literally. It is plausible to assume people with backgrounds in academia might favour the Python version to the transpiled one, as it closely resembles how we would write set comprehension in mathematics.

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

Listing 3.3: A list comprehension of applying f(n) to integers in the range -10 to 10, and placing the results in a list

```
    # 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
```

Listing 3.4: The result of transpiling the code in Listing ?? with Pseudogen

It is clear that the target audience for Pseudogen's output formats must be people with little to no experience with reading/writing code. It seems like an excellent tool for translating Python to English, and shows that something like this is indeed possible. However, Psnodig's intended target audience is broader, and thus we believe Pseudogen alone is not enough to solve the problem we are dealing with.

PseudoEditor

Har ikke fått denne til å funke enda! Synes ikke den gir ikke mening i det hele tatt. Gjerne sjekk den ut på https://pseudoeditor.com/app/. Selv ikke å poste eksemplet fra https://pseudoeditor.com/guides/merge-sort fungerer..

Føler derimot at jeg bør ha flere enn ett eksempel på 'source code -> pseudocode'. Eller?

3.3.2 Source code to IBP

Even though there is research arguing for the good effects of flowcharts in computer science, there have been few documented efforts towards developing a tool that can effectively translate source code to flowcharts. However, there does exist some software that lets us write flowcharts through a DSL, and we will look at two of them.

Naive approach

Yet again, calling it a naive approach might not be entirely accurate, but manually translating our code to flowcharts does introduce some intricacies. For one, like with TBP, we have to maintain both versions, and if we change too much of our main idea, then the time spent on making the IBP version is - to a certain extent - wasted.

Another flaw is that we have to spend time thinking about how the flowchart should look like, which parts could and should be abstracted, how they should be presented instead (or if they should be removed altogether) etc. Which colours should the frames have? How should the arrows look? Which font should be used? By using an automated tool, we do not have to worry about details unrelated to the actual program logic.

The perk of doing it this way, however, is that we can do it entirely our way. We can choose which tools we want to use, and if we are already proficient in making flowcharts based on source code, then this actually seems like a natural choice. If we also like spending time on details like colours, fonts, shapes etc., then at least the naive approach does not limit our creativity in any way.

`Jeg føler egentlig at det er naturlig å starte med det positive, og avslutte med det negative?`

Code2Flow

Code2Flow is a tool that lets us create flowcharts with natural language, decorated with a C-inspired syntax. Their website states that we might get away with pasting syntactically correct C programs, but that this is purely incidental. This goes to show that the Code2Flow team have indeed developed a DSL with their own syntax.

Flowcharts created with Code2Flow have a few, consistent colours to differentiate parts of their corresponding programs. Start- and end expressions are displayed as red ovals, while all remaining expressions are displayed as blue rectangles. Conditionals, loops and match statements are displayed with red rhombuses, and comments are displayed with orange rectangles.

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

Listing 3.5: A Code2Flow program

Listing 3.5 shows a program written with Code2Flow, and Figure 3.2 shows the corresponding flowchart. As we can see, syntactically correct expressions are just any combination of UTF-8 characters. Thus, we have no way to test a Code2Flow program. In fact, Code2Flow will *never* let us know about syntactic errors, and will *always* try to construct whatever flowchart it can.

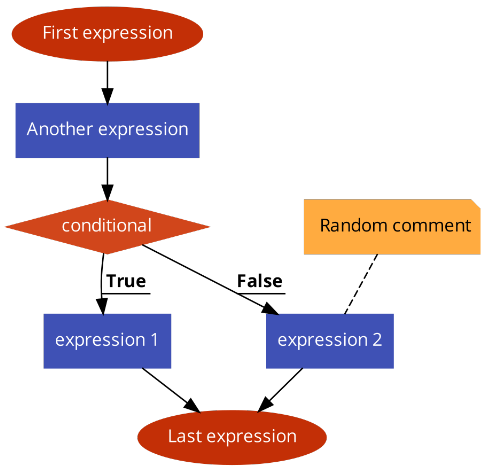


Figure 3.2: The resulting flowchart from transpiling the Code2Flow code in Listing 3.5

If our C program inadvertently creates a "correct" flowchart, we can use a C compiler to test said program on the side. However, we must now maintain both versions, and changes to the C program are not guaranteed to successfully transpile to a new flowchart.

Mermaid.js

Mermaid.js is a DSL for rendering diagrams (including flowcharts) from a Markdown-inspired syntax. Even though we can construct many different diagrams with Mermaid.js, we will focus on the flowcharts.

Like Code2Flow, they also render flowcharts in real time. However, Mermaid.js will warn us about syntax errors, and only re-render syntactically correct programs.

To write a Mermaid.js flowchart, our program must start with `flowchart TD`. Nodes can come in many different shapes, and are denoted by the types of brackets they use. For instance, `Node[ ]` displays a rectangle, `Node(( ))` displays a circle, and `Node{ }` displays a rhombus. Edges also come in many shapes. `-->` displays an arrow, `---` displays a simple link, and `-.->` displays a dotted arrow.

We can also have add text, by writing it either inside the brackets, like `Node[text]`. Arrows can also include text by breaking them up into two parts, like `-- text`

`-->`.[2]

```
    flowchart TD
        A([First expression])
            --> B[Another expression]
        B --> C{contidional}
        C -- True --> D[Expression 1]
        C -- False --> E[Expression 2]
            %% Random comment
        D --> F([Last expression])
        E --> F
```

Listing 3.6: A mermaid.js program

Just like with Code2Flow, the text inside these nodes can be anything. Listing 3.5 shows a program written with Mermaid.js, and Figure 3.3 shows the corresponding flowchart. Contrary to Code2Flow, comments are ignored by the parser, and solely exist to aid the programmer. They must also be on their own lines.

The biggest drawback of Mermaid.js is that the syntax is very different from any programming language. This means pasting our source code will not yield any result, and we have to carefully translate our code every time. This means that it is fully our responsability to maintain the abstraction level we want. Like Code2Flow, Mermaid.js has no way of letting us test the code either.

[2]The full documentation can be found at `https://mermaid.js.org/syntax/flowchart.html`

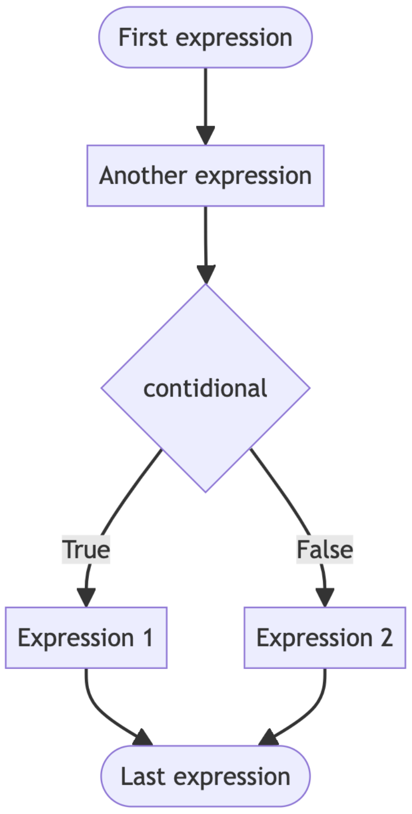


Figure 3.3: The resulting flowchart from transpiling the Mermaid.js code in Listing 3.6

Chapter 4

Design

In this chapter, we introduce Psnodig and relevant design decisions behind the tool. We also describe the design behind the Gourmet programming language, and the programs that generate TBP and IBP.

4.1 Proposed solution

To solve the problem we are dealing with, we introduce Psnodig. We believe that Psnodig offers something unique in the context of this problem. Psnodig is a transpiler, intended to convert executable programs to equivalent presentation-only programs.

The software discussed in Chapter 3 all do a very good job in their own right. The effort invested by developers, researchers and others involved is clearly reflected in both functionality and performance in each application.

However, the results they generate are sealed, and we have no choice but to accept whatever we receive. We believe there is more value in a tool that also lets us modify the final result, in case the translation from source code to PDF too lossy.

Additionally, we also believe these programs lie in the same sphere. Thus, it would make sense to gather and apply them in the same tool, rather than having to switch between different ones. We also see value in users adding their own parsers and code generators.

Where the programs primarily specialise in code generations, the DSLs are not executable. We believe that this is something that could be of tremendous use, and have therefore incorporated an interpreter that works on the internal representation of Psnodig. To the best of our knowledge, a tool which combines all these methods does not currently exist.

Lastly, code generators traditionally do much more than directly translating data types to the target language. Therefore, we will refer to these programs simply as **writers** for the remainder of the thesis, in the context of Psnodig.

4.2 Psnodig

Psnodig is a collection of data types in Haskell, describing a computer program. As such, it does not provide a lot of functionality on its own. It is first when we add parsers and writers that it shows its usefulness.

Since we are free to add parsers and writers at will, only our imagination (and programming abilities) can limit what we use it to convert. However, it is first and foremost intended as a tool for converting executable source programs to a presentation-only target programs.

Psnodig's transpilation flow is as follows: source programs are parsed to an intermediate, internal representation. Later, this representation is used to convert the original program further into a new target program. This means that a source program written in two different languages can produce the same target program, given that they have identical ASTs.

Psnodig also comes with an interpreter, which works on the internal representation. This means that we can add parsers and run the code, without also having to write an interpreter or compiler.

The main benefit is that we can write our code once in a source language, test it with the interpreter, and when we are satisfied, transpile it to TBP and/or IBP thorugh the command line, rather than having to re-write it manually or search for a tool that can do that job for us.

`Den neste delen stemmer ikke foreløpig, men er visstnok veldig enkel å få til.. Bare fikse noe cabal-greier og legge det ut på github :) Akkurat nå må man skrive f.eks. ‘stack run --ast program‘, sikkert ikke krise om det ender slik og?`

Currently, Psnodig offers four command line arguments. If we at some point wish to add more, we can add them to the project's `Main.hs` file.

`psnodig program`, which will parse a program and run it through the interpreter.

`psnodig --ast program`, which will print the program's AST to the terminal, given that it is syntactically correct. This can be a nice way to debug programs, and get a better understanding of how our parsers actually parse our programs.

`psnodig --tbp program`, which will parse the program and produce a LaTeX file of TBP. We can also add another flag `--pdf` to get an accompanying PDF of said LaTeX file.

`psnodig --ibp program`, which will parse the program and produce a LaTeX file of IBP. We can also add another flag `--pdf` to get an accompanying PDF of said LaTeX file.

4.2.1 Syntax

Blir syntax riktig ord å bruke her?

The data types of Psnodig are presented in Listing 4.1. The entry point of a Psnodig program is `Program`, but since it consists of two lists and a `Maybe` data types, a minimal working example is actually an empty file. This flexibility allows our parsers and writers to utilise just as much, or as little, of the Psnodig syntax as we wish.

Most of the syntax will resemble the syntax of common programming languages. However, there are two statements that have been introduced specifically for Psnodig's real use case: Hash- and Annotation statements.

```
data Program = Program [StructDecl] [Function]
               (Maybe FunctionCall)

data StructDecl = StructDecl String [Argument]

data Struct = Struct String [Expression]

data StructField = StructField Expression
                   Expression

data Function = Function String [Argument]
                [Statement]

data FunctionCall = FunctionCall String
                    [Expression]

data Argument = Argument String String

data Statement =
      Assignment AssignmentTarget AssignmentValue
    | Loop Expression [Statement]
    | If Expression [Statement] (Maybe Else)
    | ForEach String Expression [Statement]
    | For String Expression Expression [Statement]
    | CallStmt FunctionCall
    | Return Expression
    | HashStmt Statement
    | AnnotationStmt String [Statement]
    | Break
    | Continue

data AssignmentTarget =
      VariableTarget String
    | ListIndexTarget String [Expression]
    | StructFieldTarget StructField
```

```
data AssignmentValue =
      ExpressionValue Expression
    | StructValue Struct

data Else =
      ElseIf Expression [Statement] (Maybe Else)
    | Else [Statement]

data Expression =
      Constant Value
    | VariableExp String
    | BinaryExp Operator Expression Expression
    | ListIndex String [Expression]
    | CallExp FunctionCall
    | Not Expression
    | StructExpr Struct
    | StructFieldExp StructField

data Operator =
      Plus
    | Minus
    | Times
    | Division
    | LessThan
    | LessThanEqual
    | GreaterThan
    | GreaterThanEqual
    | Equal
    | NotEqual
    | And
    | Or
    | Modulo

data Value =
      Nil
    | Boolean Bool
    | Number Integer
    | Decimal Double
    | Text String
    | List [Expression]
    | HashSet (Set.Set Expression)
    | HashMap (Map.Map Expression Expression)
    | StructVal [(String, Value)]
```

Listing 4.1: Psnodig's data types in Haskell

Hash statements are picked up by the parser and intended to be interpreted, but ignored by writers. This lets us abstract away things that we deem to be obvious to our audience. It could also be things that have already been stated implicitly elsewhere, but still need to be parsed for our programs to run. The name derives from how we can write single line comments in Python with a hash symbol.

A common use case of this is how a lower case `n` is often used to denote amounts in TBP. By using a hash statement, we avoid including superflous length-function calls in cases where it is obvious what `n` symbolises. However, this will never be obvious to the interpreter, and thus we still have to include it in our source programs somehow.

Annotation statements are statements that allow us to add an extra layer of abstraction to our output targets. The first string is what is transpiled, without any conversion, whilst the list of statements is reserved for the interpreter.

A common use case of this is when we wish to swap two elements in a list. In many programming languages, when swapping two elements `a` and `b`, we have to assign `a` to a temporary variable, then assign `a` to `b`, before we can finally assign `b` to that temporary variable. These three implementation-specific lines could easily be abstracted with `swap a and b`.

These two statements are particularly useful when a piece of code is not crucial to the program's logic, or when the code is very implementation specific. The statement list can also be empty, which lets us explain things solely with natural language when deemed necessary. A value of `AnnotationStatement "" []` will be ignored by the IBP writer, but is a way of allowing the TBP writer to add whitespace.

4.2.2 Interpreter

A big selling point of Psnodig is that in addition to being a transpiler, it also comes with an interpreter which works on the AST. For a program to be transpiled, it only needs to be syntactically correct. However, this does not guarantee that the program works as the we intend. Thus, Psnodig provides users with the ability to test their programs before they are transpiled and later presented, which in many cases can be crucial.

Take the program presented in Listing 4.2 as an example. The function `printEvenNumbers` takes two arguments: a list of numbers and the list's length. Then we iterate through this range, and proceed to print every number in the list that is even. However, we check for evenness by doing `if i % 2 == 0`, when really we intend to do `if numbers[i] % 2 == 0`. The difference is subtle, but by running the program we quickly realise the error when the screen displays 47, 79 and 93 rather than 46 and 22.

```
func printEvenNumbers(numbers list, length int) {
    for i := 0, length-1 {
        if i % 2 == 0 {
            print(numbers[i])
        }
    }
    return 1
}

printEvenNumbers([47, 46, 79, 22, 93], 5)
```

Listing 4.2: A syntactically correct program with a subtle logical error

`Dette programmer er skrevet i Gourmet. Bør jeg heller skrive det i Go/Java, siden vi ikke har introdusert noe Gourmetsyntaks enda?`

As seen from the grammar in Listing 4.1, programs consist of three parts: A list of struct declarations (which can be empty), a list of function declarations (which can also be empty), and lastly, an optional function call. The function call works as the interpreter's entry point.

Scope

Psnodig works with both a global and a local scope. All structs and functions are global, and can be accessed from any other functions. For instance, functions can be mutually recursive. Variables, on the other hand, are strictly local, and a variable declared in function `f` cannot be accessed in function `g`, unless passed as an argument.

Listing 4.3 and Listing 4.4 both show syntactically correct programs. However, running the former will yield an error, because `n` is not available in the scope of `g`. The latter program will yield the value of n, as we pass the variable as an argument to `g`, which returns the variable straight back.

```
func f() {
    n := 5
    return g()
}

func g() {
    return n
}

f()
```

Listing 4.3: Code with error

```
func f() {
    n := 5
    return g(n)
}

func g(n int) {
    return n
}

f()
```

Listing 4.4: Code without error

Psnodig supports nested scopes as indicated in the `For String Expression Expression [Statement]` structure within the `Statement` data type. The two `Expression` data types define a range, whilst the `String` data type serves as an identifier that binds to all numbers within this specified range (inclusive).

The statements are then executed repeatedly for each value in this range, with the identifier reflecting the current value on each iteration. Once the loop terminates, the identifier is automatically unbound, and no longer exists in any context.

Iterating expressions

The trained eye will notice that Psnodig prohibits two types of for loops. `For String Expression [Statement]` is discused in the previous subsection, but we also allow `ForEach String Expression [Statement]`. This is intended to mimic standard `For each` loops found in languages like Java and Python.

One could say that it is too expressive, technically allowing iteration of non-iterables like arithmetic expressions. However, since all iterables also fall under the `Expression` type, we allow it syntactically, and instead deal with it on the interpreter side.

Standard Library

The interpreter provides several built-in functions, which are also found in many programming languages. They are also reflected in the TBP writer. If a function call fails, due to e.g. wrong number of arguments or arguments having the wrong type, the program will stop and the user will receive an explanatory error message.

`Bør jeg ha med eksempelkall? f.eks. noe ala print("master ", "thesis") -> displays "master thesis" to the terminal and returns 2.`

`Print og append skiller seg fra de andre, da jeg har med en pil som indikerer hva som returneres. Vet ikke om det ser litt rotete/forvirrende ut?`

print(x_1, .., x_n) $\rightarrow$ **n**, which takes $n \in \mathbb{N}$ arguments. The arguments must be of type `Expression`. Each argument is converted to a string and concatenated, separated by a space, before they are printed to the terminal. The function returns the number of arguments passed to it.

length(x), which takes one argument. The argument must be of type `Text`, `List`, `HashSet`, or `HashMap`. The function returns the length of its argument: Number of characters in the text, number of elements in the list or hashset, or number of mappings in the hashmap.

ceil(x) and **floor**(x), which take one number $x \in \mathbb{Q}$ as argument. The return value will be rounded up or down to the nearest $x' \in \mathbb{N}$, respectively.

min(x_1, .., x_n) and **max**(x_1, .., x_n), which takes $n \in \mathbb{N}$ arguments. The arguments themselves must be an $x \in \mathbb{Q}$. The return value will be the smallest value or the largest value, respectively, amongst the arguments.

append(x, xs) $\rightarrow$ **1**, which takes two arguments. The first argument must be an `Expression`, and the second argument must be a `List`. The function will append `x` to the end of `xs`, thus modifying the list locally. The function returns the number 1 if everything went well.

add(x, hs) and **add**(k, v, hm), which is an overloaded function, taking either two or three arguments. In the first case, it adds an `Expression` x to a `HashSet` hs. In the other case it maps an `Expression` k to an `Expression` v in a `HashMap` hm. The function returns 1 in both cases upon success.

get(k, hm), which takes two arguments, an `Expression` k and a `HashMap` hm. If k is a key in hm, the function will return the value that k maps to in hm.

in(x, xs), which takes two arguments, an `Expression` x and either a `List`, `HashSet`, or `HashMap`. The function will check if x exists in xs, and return $True$ or $False$ accordingly.

4.2.3 Testing

Psnodig is not accompanied by any testing framework, but we have done some testing with QuickCheck.

`Bør dette heller flyttes til et annet sted?`

`Har ingen tester i skrivende stund. Forslag til tester:`
`1. 'AST -> Gourmet -> AST'`
`2. Teste biblioteksfunksjoner`

`En annen ting som kan være interessant å teste: Transpile programmer til LaTeX uten å krasje. Opplever neon ganger at det er edge caser jeg har glemt som gjør at vi får feil. Vet ikke helt hvordan dette skulle blitt testet, men det gir vel noe grunnlag for å si at det er ''trygt'' å bruke Psnodig? At det er ''komplett'' på et vis (alle gyldige programmer kan bli transpilet)`

4.3 Gourmet

To make sure Psnodig works at all, we depend on having at least one input target and one output target. When it comes to input language, we have two plausible alternatives: Use an existing programming language, or design a new one. We opted for the latter.

In the context of Psnodig, we just need to build a parser that can translate programs in the source language to the intermediate representation. There are several reasons as to why designing a new language for the purpose of proof of concept is a good choice.

For one, it demonstrates the general effort to add an entirely new input target for Psnodig. This can motivate others to add their own.

Another reason is that selecting a single programming language to encompass all needs is not feasible. For instance, the largest university in southern Norway uses Python for its introductory programming course, [1] whilst the largest university in northern Norway prefers C. [2] The largest university in Greece opts for Java, [3] and Harvard's renowned CS50 course introduces students to both JavaScript and SQL. [4]

`Lars kommenterte en gang at en fotnote burde ligge på utsiden høyresiden av punktum. Gjelder det også over her? er det bedre med feks ‘‘tekst her og tekst der.1 2’’ istedenfor ‘‘tekst her 1 og tekst der 2.’’? hvis det ga mening`

What the languages of introductory courses to computer science do have in common, is that they tend to be within the imperative paradigm of computer programming. We did not want to stray too far away from that, and Gourmet's syntax is mainly inspired by imperative programming languages like Go and Python. In fact, Gourmet started out as a pure subset of the former, hence its name: A gourmet portion of Go.

4.3.1 Lexical Aspects

Identifiers and keywords

An identifier in Gourmet is used to reference either a struct, function or variable. Identifiers are also restricted to begin with a letter, followed by an arbitrary number of letters, numbers, and single quotes. **variable**, **var1able'** and **g0urmetVar1able''''** are all valid Gourmet identifiers, whilst **1variable** and **'variable** are both not. Additionally, an identifier can not shadow keywords.

In total, there are 14 keywords in Gourmet. They are **while**, **if**, **func**, **true**, **false**, **return**, **else**, **for**, **break**, **continue**, **struct**, **not**, **map** and **set**. These keywords are also reserved, which means that we cannot define an identifier **while** or **func**. The complete overview of reserved names is presented in Listing 5.3

We are not allowed to define functions that shadow library functions in Psnodig, and functions cannot be defined multiple times. This effectively means that we cannot overload functions with a different number of arguments, technically removing the ambiguity. In other words, we cannot define and use a custom function called **print**.

However, we can always define a function e.g. **print'** or **print1**. Also, this is only relevant for interpreting programs, as creating functions does not break any syntax rules.

[1]Course page for *Introduction to object-oriented programming* at the University of Oslo: `www.uio.no/studier/emner/matnat/ifi/IN1000/`

[2]Curriculum for *Introduction to programming and the computer's mode of operation* at the University of Tromsø: `www.bibsys-c.alma.exlibrisgroup.com/leganto/readinglist/lists/10569365600002205?institute=47BIBSYS_UBTO&auth=SAML`

[3]Course page for *Introduction to Computer Science* at the University of Athens: `www.dept.aueb.gr/en/dmst/content/introduction-computer-science`

[4]Course page for *Introduction to Computer Science* at the University of Harvard: `www.pll.harvard.edu/course/cs50-introduction-computer-science`

Comments

As there are no data types for comments in Psnodig, they must be handled entirely by Gourmet parser. The language supports both single- and multi-line comments, identical to the ones found in most C-like languages like C itself, Go, Java, and more.

Single-line comments begin with a double forward slash `//`, and extend to the end of that line. Multi-line comments start with a forward slash and a star `/*`, and end with a star and a forward slash `*/`. Multi-line comments cannot be nested, which means that the first `*/` after a `/*` will end that comment, no matter how many `/*` preceeds it.

Whitespace

Whitespace can be defined as spaces, newlines and tabs. Gourmet does not differentiate between either of them, and we can use them in our programs exactly how we wish.

Listing 4.5, Listing 4.6 and Listing 4.7 show three identical programs, besides the use of whitespace. We define a function **f** which again defines **a** to be a list with two elements, before returning **a**. Lastly, the fuction is called. All three programs will be parsed successfully, and have the same internal representation in Psnodig.

```
func f() {
    a := [1, 2]
    return a
}

f()
```

Listing 4.5: f with a standard amount of whitespace

```
func f()
{
    a        :=
        [1    ,    2]
    return
    a
}

f    (    )
```

Listing 4.6: f with a lot of whitespace

```
funcf(){a:=[1,2]returna}f()
```

Listing 4.7: f with no whitespace

Jeg har tatt mye inspirasjon herfra: https://github.uio.no/compilerconstruction-inf5110/compila/blob/master/doc/languagespec/compila.pdf uten å kildeføre noe. her er jeg veldig usikker på hva jeg skal gjøre! Har jeg kopiert for mye? Finnes det en ryddig måte å kildeføre noe slikt?

4.3.2 Types

Gourmet is dynamically typed, which means that we do not have to specify the type of variables and return type of functions. In fact, Gourmet does not even *allow* it. When defining structs and function arguments, however, we have to provide type hints on the form **name *type***.

The reason behind this choice is solely to improve code readability. When defining a variable, the value will be clearly present on the right hand side. When working with function arguments, however, we can never be entirely sure what the caller may pass. With type hints, at least we have a grasp of the arguments' *intended* use.

There are, however, five base types that any value will have: **Boolean**, **Number**, **Decimal**, **Text**, and **Nil**. A boolean value is either true or false, a number value is an $n \in \mathbb{N}$, a decimal value is a $q \in \mathbb{Q}$, and a text value is an arbitrary combination of letters within a pair of double quotes. Nil indicates the absence of a value, and is usually serves as a placeholder.

Essentially, types associate data values into classes and provide rules for how these classes should interact [37, 24]. Sometimes, the base types Gourmet offers might not suffice. Therefore, we can create our own types through structs. Structs in Gourmet work exactly the same way they do in languages like C and Go, containing instance variables, but no methods or constructors, contrary to languages like Python and Java.

Listing 4.8 shows how we can create a struct for modelling a tree data structure, and Listing 4.9 shows how they can be initialised.

```
struct Tree {
    value int,
    left Tree,
    right Tree
}
```

Listing 4.8: A Gourmet struct Tree, with instance variables value, left and right

```
func f() {
    tree := struct Tree(10, nil, nil)
    tree' := struct Tree(20, nil, nil)
    tree'' := struct Tree(15, tree, tree')
    return tree''
}
```

Listing 4.9: A function to initialise three tree structs and return the last one

4.3.3 Syntax

Grammar

Gourmet's EBNF grammar is presented in Figure 4.1. EBNF (short for Extended Backus-Naur form) is a notation for expressing a programming language's grammar. It is an extension of BNF (short for Backus-Naur form) that was developed in the 1960s to describe the syntax of the ALGOL programming language [14]. We could have used the original BNF notation, but EBNF allows us to present it more succinctly.

In this variant of EBNF, we use the following meta symbols:

```
-> { } [ ] " |
```

Arrows indicate the application of a rule. Curly brackets indicate repetition of 0 or more times (much like the reflexive arrow in Figure 2.1). Square brackets indicate binary presentness. Vertical bars indicate option. Anything wrapped in double quotes is a keyword.

Non-terminals are written in upper case. There are also five terms written in upper case, that do not have production rules: **NAME**, **TEXT**, **NUMBER**, **DECIMAL**, **STRING_LITERAL**. NAME is an identifier, already described in Section 4.3.1.1. TEXT is any combination of UFT-8 symbols. It is only present once, and is intended for "hiding" code under an abstracting layer of natural language. NUMBER is any $n \in \mathbb{N}$. DECIMAL is any $q \in \mathbb{Q}$. STRING_LITERAL is similar to TEXT, but additionally it is wrapped in double quotes.

Bør jeg forklare dette mer? F.eks. dette med terminals og non-terminals osv, production rules etc.

Precedence and Associativity

The precedence of Gourmet operators is ranked in the following order, from highest to lowest:

1. $*$, / and %
2. + and $-$
3. <, <=, > and >=
4. == and ! =
5. && and ||
6. . (to access fields of a struct)

```
PROGRAM           -> { STRUCTDECL } { FUNCTIONDECL }
                     [ FUNCTIONCALL ]

STRUCTDECL        -> "struct" NAME "{" { ARGUMENT } "}"

FUNCTION          -> "func" NAME "(" [ ARGUMENT { ","
                     ARGUMENT } ] ")" "{" { STATEMENT }
                     "}"
ARGUMENT          -> NAME NAME

STATEMENT         -> ASSIGNMENT | LOOP | IF | FOREACH
                     | FOR | FUNCTIONCALL | ANNOTATIONSTMT
                     | # STATEMENT | "return" EXPRESSION
                     | "break" | "continue"

ASSIGNMENT        -> ASSIGNMENTTARGET ":=" ASSIGNMENTVALUE

LOOP              -> "while" EXPRESSION "{" { STATEMENT } "}"

IF                -> "if" EXPRESSION "{" { STATEMENT } "}"
                     [ ELSE ]

FOREACH           -> "for" NAME ":=" EXPRESSION "{"
                     { STATEMENT } "}"

FOR               -> "for" NAME ":=" EXPRESSION ","
                     EXPRESSION "{" { STATEMENT } "}"

FUNCTIONCALL      -> NAME "(" EXPLIST ")"

ANNOTATIONSTMT    -> "@" "{" TEXT "}" "{" { STATEMENT } "}"

ASSIGNMENTTARGET -> NAME | LISTINDEX | STRUCTFIELD

ASSIGNMENTVALUE   -> EXPRESSION | STRUCT

ELSE              -> "else" IF | "else" "{" { STATEMENT } "}"

EXPRESSION        -> VALUE | NAME | LISTINDEX
                     | EXPRESSION OPERATOR EXPRESSION
                     | FUNCTIONCALL | "not" EXPRESSION
                     | STRUCT | STRUCTFIELD

LISTINDEX         -> NAME "[" EXPRESSION "]" { "["
                     EXPRESSION "]" }

OPERATOR          -> "+" | "-" | "*" | "/" | "<" | "<="
                     | ">" | ">=" | "==" | "!=" | "&&"
                     | "||" | "%"
```

```
STRUCT              -> NAME "(" EXPLIST ")"

STRUCTFIELD         -> EXPRESSION "." EXPRESSION

VALUE               -> "nil" | "true" | "false" | NUMBER
                       | DECIMAL | STRING_LITERAL
                       | "map" "{" [ PAIR { "," PAIR } ] "}"
                       | "set" "{" EXPLIST "}"
                       | "[" EXPLIST "]"

PAIR                -> EXPRESSION ":" EXPRESSION

EXPLIST             -> [ EXPRESSION { "," EXPRESSION } ]
```

Figure 4.1: The EBNF grammar of the Gourmet programming language

7. not

This means that if we wish to calculate the sum of the tree values from Listing 4.9, we cannot write `tree.value + tree’.value + tree’’.value`, because the parser will parse `(p.(value + p’).(value + p’’)).value`. Therefore, we have to include parentheses explicitly: `(tree.value) + (tree’.value) + (tree’’.value)`.

All binary operations are left-associative.

4.4 TBP Writer

Our TBP writer takes an internal representation of Psnodig and produces a LaTeX file. As previously stated, we are not attempting to create a ground truth for pseudocode. Therefore, we have chosen to produce the pseudocode with the Algorithm2e package.

Akkurat nå håndterer vi ikke Input, Output og Caption. Hvordan dette gjøres er nok ikke så farlig. Må bestemme om dette skal bakes inn i Psnodig, eller om vi f.eks. skal hente det fra kommandolinjen idet noen kjører ‘‘stack run -- tbp program.gt’’

Structs and the initial function call are not converted to pseudocode. For one, we believe that function calls are rarely cruical to the algorithm itself, and structs will always be implementation specific. The second reason is that there does not seem to be an obvious way of transpiling them in a way that makes sense.

4.4.1 Algorithm2e

Bør jeg skrive en liten intro om pakken, f.eks. historie og sånt?

Algorithm2e allows us to define our own keywords with the `\SetKw{}{}` command. These are macros, so if we define `\SetKw{KwBreak}{break}`, we can write `\KwBreak`, and `break` will be printed in a bold font.

There are also more specific macros, like `\SetKwProg{proc}{Procedure}{is}{end}`. This is used to initialise programs, and denotes a special syntax for the program. Listing 4.10 shows an example program, and Figure 4.2 shows the subsequent compiled result. For our TBP writer, we decide to ignore the last two parameters of `SetKwProg`, because we do not believe they add enough value to the final result.

```
...
\SetKwProg{proc}{Procedure}{}{}
\SetKwFunction{ExampleProgram}{ExampleProgram}
...
\begin{document}
\begin{algorithm}[H]
  \KwIn{Nothing}
  \KwOut{The number 1}
  \proc{$\ExampleProgram()$}{
    \Return 1
  }
  \caption{Example program}
\end{algorithm}
\end{document}
```

Listing 4.10: Example program with Algorithm2e to show macros in action

```
Algorithm: Example program
   Input: Nothing
   Output: The number 1
 1 Procedure ExampleProgram()
 2     return 1
```

Figure 4.2: The result of compiling Listing 4.10

Another special macro is `\SetKwFunction{f}{f}`. This allows us to write e.g. `\f{arg1, arg2, arg3}`, which displays `f(`*arg1, arg2, arg3*`)` in the compiled version.

Since we are still in a LaTeX environment, and backslashes are used for macros in standard LaTeX too, we have to be careful. We are not allowed to rename internal macros. For instance, `\m` is already a macro in LaTeX, thus attempting to compile a file with `\SetKwFunction{m}{m}` will lead to multiple errors and ruin the final output.

4.4.2 Compatibility with Psnodig

As previously mentioned in Section 4.2.2.3, all Psnodig library functions taken into consideration by our TBP writer. The function call `length(list)` is transpiled with the cardinality symbols to $|list|$. The function call `append(x, xs)` is transpiled with natural language to `append x to xs`, to avoid ambiguity.

Mathematical expressions are also taken into account. For instance, the an expression of value `BinaryExp Division (Constant (Number 2)) (Constant (Number 1))` will show $\frac{2}{1}$, rather than something like `2/1`. Similarly, an expression with multiplication will be displayed as `m · n` rather than `m * n`. To work with mathematical symbols in LaTeX we use the packages **amsmath** and **commath**.

We also declare some macros to match the syntax of Psnodig. Before our programs are transpiled to TBP, we scan it for keywords, which are then imported in the LaTeX file. If our program contains code like `v := false`, the LaTeX file will include `\SetKw{False}{false}`, and we will apply `\KwFalse` rather than just "false".

4.4.3 Output

If we run `psnodig --tbp program` in our command line, and the program is syntactically correct, we receive a corresponding LaTeX file. If we add a flag `--pdf`, we also get a PDF of the compiled LaTeX file.

Because the LaTeX file is built from Psnodig's internal representation, and not directly from the source program, formatting is not taken into account. This means that a program like

```
func f(){v:=5 returnv}
```

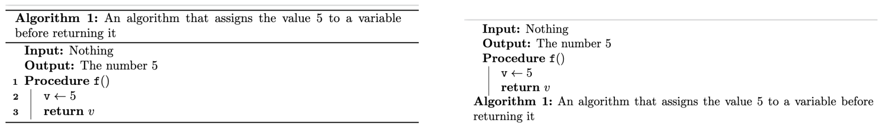
will not be transpiled to

```
\proc{$\f()$}{$\texttt{v}\gets5$\; \Return$v$\;}
```

but instead

```
\proc{$\f()$}{
    $\texttt{v} \gets 5$ \;
    \Return $v$ \;
}
```

even though the compiled versions produce the same PDF. The LaTeX files will also include `\SetKwProg{proc}{Procedure}{}{}` and `\SetKwFunction{f}{f}`. We include the `linesnumbered`- and `ruled` parameters with algorithm2e, purely for aesthetic reasons. Figure 4.3 (a) shows the program we just mentioned *with* these parameters, whilst Figure 4.3 (b) shows the same program *without* them.



```
Algorithm 1: An algorithm that assigns the value 5 to a variable
before returning it
  Input: Nothing
  Output: The number 5
1 Procedure f()
2   | v ← 5
3   | return v
```

```
  Input: Nothing
  Output: The number 5
  Procedure f()
    | v ← 5
    | return v
Algorithm 1: An algorithm that assigns the value 5 to a variable before
returning it
```

(a) f with linesnumbered and ruled   (b) f without plain Algorithm2e

Figure 4.3: TBP of an algorithm f, with and without including linesnumbered and ruled from the Algorithm2e package

4.5 IBP Writer

Our IBP writer works much like the TBP writer: An internal representation of Psnodig is transpiled to a LaTeX file. The flowcharts are created with the TikZ package.

Just like the TBP writer, our IBP writer only transpiles the topmost function, ignoring structs and the initial function call. Our reasoning is the same as earlier.

4.5.1 TikZ

TikZ is an enormous package, which can be used for just about anything related to drawing in LaTeX. In fact, we actually used TikZ to create the FSA example in Figure 2.1.

`Bør jeg skrive en intro om denne pakken også?`

Our flowcharts mainly consist of three macros: `\tikzstyle`, `\node`, and `\edge`. `tikzstyle` lets us choose what our nodes look like, `ndode` lets us draw the nodes, and `edge` lets us add edges between nodes.

Each node in our flowcharts represents a statement, except the top one, which is the function name. The visual design is simple: the top node and all nodes of return statements are dark rectangles with rounded edges. Decision nodes, where the program diverges, are yellow rhombuses. This includes while-, for- and if-statements. The remaining statements are purple rectangles. Edges are presented as thin arrows.

Tikzstyles are written more or less on the form

```
\tikzstyle {style name} =
    [ <shape>
    , minimum width = <x> cm
    , minimum height = <y> cm
    , text <position>
    , draw = <colour>
    , text = <colour'>
    , fill = <colour''>
    ]
```

The style name will be referenced later by nodes. The standard shapes are rectangle, circle or coordinate, however we can import more libraries to get shapes like e.g. diamonds. The three colours refer to the node's border-, text- and background colours, respectively. We have also opted to center all text.

Nodes are written on the form

```
\node (unique name)
      [metadata]
      {text displayed on node}
```

All nodes should have a unique name, so that they can be referenced correctly later. The square brackets denote metadata, like what the node should look like (by referencing a tikzstyle), or the node's positioning relative to other nodes. The curly brackets is the text displayed within the node body.

Edges are written on the form

```
\draw [edge] (node) -- (node')
```

The biggest issue with the TikZ package, is that we first have to write all of our nodes before adding edges between them. `This means that we must save all statements in a graph before printing them.`

When working with large control flow statements, we have to deal with multiple else-branches. Rather than following each other vertically, they are placed horizontally. `We calculate an even distance, and place the nodes adjacent to each other with this distance`
. An issue is that nodes with a lot of text might interfere with each other.

`Eksempler?`

Additionally, having multiple straight edges from the same source might interfere with edge labels. Thus, we might also have to change the way we draw edges. An option is to change `--` to `|-` like

```
\draw [edge] (node) |- (node');
```

This will curve the edge between `node` and `node'`, making the flowchart clearer. However, this is a difficult task to carry out since we never know how programs turn out beforehand. Therefore it seems natural to opt for the easiest choice available and instead, unfortunately, force the authors to tweak the resulting LaTeX on their own if they are dissatisfied.

4.5.2 Compatibility with Psnodig

`Jeg lar denne stå litt WIP, siden jeg ikke har kommet så langt`

4.5.3 Output

Working with IBP in the command line works precisely like it does with TBP. We run `psnodig --ibp program` to transpile the program and receive a corresponding LaTeX file, given that the program is syntactically correct. If we also wish for the compiled result, we add a `--pdf` flag.

Listing 4.11 shows the (most important parts of the) LaTeX file we receive by transpiling the program from Section 4.4.3. We see how every node has a unique ID, reference a tikzstyle, and how they are explicitly placed beneath each other.

```
    \node (0) [startstop] {f()};
    \node (1) [statement, below of=0] {v = 5};
    \node (2) [startstop, below of=1] {v};

    \draw [edge] (1) -- (2);
    \draw [edge] (0) -- (1);
```

Listing 4.11: The LaTeX from transpiling a program to IBP

Chapter 5

Implementation

This chapter will present concrete implementations of Psnodig and its parts. As previously mentioned, Psnodig is really just a syntax. The remaining contributions of this thesis are Haskell programs, written to work together with Psnodig. We will look at how each of these individual programs are created, as well as roughly explaining Monads, which are essential to all of our main programs.

`Rekkefølgen her stemmer ikke.  Er det dumt?`

5.1 Monads

First introduced in a programming context by Philip Wadler back in the 90's, monads allow us to structure computations within programs [40, 20]. They are a fundamental pattern that harmonise the needs of imperative and functional programming. They preserve the benefits of functional purity, whilst elevating the expressiveness of our code.

Haskell is a purely functional programming language [18]. In this paradigm, functions' output values are solely determined by their input values. *Side effects* are operations that occur outside a function's scope, somehow affecting or depending on an external state. Monads are a way of keeping the immutable and predictible nature of our functions, whilst simultaneously working with the "outside world".

Monads play a vital role in all of our main implementations. For instance, the interpreter uses monads to modify a global state, and our TBP writer uses monads to read from an external environment.

We also utilise monads for more effective error handling, mainly in our interpreter. We use the **Maybe** monad, which returns either `Just value` or `Nothing`, and the **Either** monad, which returns either `Left value` or `Right value`.

`Maybe` is used for lookups, for instance when trying to access the value of a variable. Thus, we either return the value, or `Nothing`, if it is not already bound. `Either` is used when we expect one of two values, though the left side is often associated with an error. For instance, running a Psnodig program will yield either an error, or the state of the finished program.

The remaining implementations in this chapter will for the most part revolve around the monad we opted for.

5.2 Interpreter

5.2.1 Monad

The interpreter is built around our monad of choice, **StateT ExecutionState (ExceptT RuntimeError IO)**. It arguably looks a bit heavy, so let us break it down. The main point of our monad is `StateT`, whose signature is `StateT s m a`.[1] As you can see, it takes three arguments: some state `s`, an inner monad `m`, and lastly `a`, the type of the result of applying the monad.

ExecutionState

The state we apply is `ExecutionState`, fleshed out in Listing 5.1. It is a compound type with named fields, and closely resembles a struct in programming languages like Go and C. Applying `StateT` allows us to maintain and operate on this state throughout computations.

```
type StructDecls = Map.Map String [String]
type FuncEnv = Map.Map String Function
type Scope = [[(String, Value)]]
type ScopeStack = [Scope]

data ExecutionState = ExecutionState
    { structDecls :: StructDecls
    , funcEnv     :: FuncEnv
    , scopeStack  :: ScopeStack
    , output      :: [String]
    }
```

Listing 5.1: The Psnodig interpreter's state

It contains `structDecls`, which is a declaration of structs. This is a mapping from struct names to their corresponding fields. For instance, the value `StructDecl "Person" [Argument "name" "txt", Argument "age" "int"]` is saved as a mapping from `"Person"` to `["name", "age"]`.

We also have `funcEnv`, which is a declaration of functions. It is a mapping from function names to the corresponding function values. If we call a function, we can look it up, bind the arguments to temporary values, and run the statements within the function body. The use of `Map` was primarily chosen due to its O(log

[1]More documentation can be found at `https://hackage.haskell.org/package/transformers-0.6.1.1/docs/Control-Monad-Trans-State-Lazy.html#g:2`

n) lookups, [2] which beats the O(n) lookups of its `List` counterpart.[3]

The third value in our state is the `scopeStack`. This is a triple nested list of `(String, Value)` pairs. A pair of this type is a binding, so writing `x := 23` will create the binding `("x", Number 23)`. The initial scope list is intended to be the base level scope of a function.

`Når jeg tenker meg om er det ingen kjempegod grunn til å ikke bruke maps her og. Er det verdt å refaktorere? Ja, maps må resize, men vi aksesserer antakeligvis variabler oftere enn vi legger til nye, og O(log n) slår fortsatt O(n).`

The nesting accounts for loops and if-statements. Upon entering e.g. a while-loop, we push a new scope to the stack, and all variables created within this loop will die when we exit the loop. The last nesting introduced by `ScopeStack` accounts for functions. Each function has their own scope, much like any other programming language.

Lastly, our state takes care of `output`, which is a list of strings. This stores whatever we pass to the `print` function. If a program crashes, only the print-calls up until the crash will be presented in the terminal, which makes it a great tool for debugging. If a program runs uninterrupted, all print-calls will be presented in the terminal.

ExceptT

The inner monad that StateT operates over is `ExceptT RuntimeError IO`. This allows us to augment the `IO` monad with a value of type `RuntimeError`. The latter is several types of custom error messages, to locate the source of error easier. The entire list is shown in Listing 5.2.

```
    data RuntimeError =
          VariableNotFound String
        | FunctionNotFound String
        | StructNotFound String
        | ArithmeticError String
        | BadArgument String
        | WrongNumberOfArguments String
        | NoReturnError String
        | Error String
        | RuntimeErrorWithOutput [String] RuntimeError
        deriving (Show)
```

Listing 5.2: All error messages to be encountered in Psnodig

[2]The entire Map documentation can be found at `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html`
[3]The entire List documentation can be found at `https://hackage.haskell.org/package/base-4.19.1.0/docs/Data-List.html`

`Skal rydde opp i denne på toget til Roma, akkurat nå bruker jeg nesten bare 'BadArgument'.`

What this really means is that we have a computation that does IO (prints to the terminal). If we encounter an error condition, we do not proceed with further computations, and instead return the error, which is also displayed in the terminal. This is also what allows us to print errors, by joining the current print calls to `RuntimeErrorWithOutput`, and followingly displaying them to the terminal before the error message.

5.2.2 Evaluation strategy

The Psnodig interpreter operates with a strict call-by-value evaluation strategy, just like the Pascal programming language. [4] When we pass lists or structs as arguments to a function, it has the capability to modify indexes and fields. However, these changes remain local to the function's scope and do not affect the caller's original data structure.

5.2.3 Deviations from the Psnodig syntax

Even though Psnodig's syntax opens up for it, the interpreter does not currently support Break- and Continue statements. This would require a significant refactoring of our monad, and potentially a larger refactoring of the entire interpreter.

`Bør jeg forklare hvordan standard library er implementert?`

5.3 Gourmet

There are two tools related to Gourmet: a parser and a writer. `Jeg nevner at vi har en Gourmet writer i Introduction, men ikke i Design. Bør den være med der og?`

5.3.1 Parser

We used the Haskell library Parsec to write the parser for Gourmet. Parsec is an industrial-strength parser library, being simple, safe, and well documented.[5] It is monadic parser combinator library, which roughly means that it combines multiple smaller parsers using monadic abstraction. This allows us to write a parser very succinctly, describing each rule in more or less natural language.

Parsec also lets us write our lexer and parser in one. The entire lexer is presented in Listing 5.3, which takes up just 20 lines. It could have been presented with even fewer lines, but we have focused on readability instead of showboating.

[4] `https://www.freepascal.org/docs-html/ref/refsu68.html`

[5] The entire Parsec documentation can be found at `https://hackage.haskell.org/package/parsec`

The lexer mainly defines the form of identifiers, comments and reserved names.

```
lexer :: Token.TokenParser ()
lexer = Token.makeTokenParser emptyDef {
    Token.identStart = letter,
    Token.identLetter = alphaNum <|> char '\'',
    Token.reservedOpNames =
        [ ":=", "+", "-", "*", "/", "<", ">", "=="
        , "!=", "{", "}", "(", ")", ">=", "<="
        , "[", "]", "&&", "||", "!", ",", ":", "#"
        , "@", "%", "."
        ],
    Token.reservedNames =
        [ "while", "if", "func", "true", "false"
        , "return", "else", "for", "break", "set"
        , "map", "not", "struct", "continue"
        ],
    Token.commentStart = "/*",
    Token.commentEnd = "*/",
    Token.commentLine = "//",
    Token.nestedComments = False
}
```

Listing 5.3: The Gourmet lexer

Listing 5.4 shows the entry function, and how programs are parsed. We parse all struct declarations, function declarations, and potentially a function call, before the program returns the resulting AST in a Parser context. `Many` means 0 or more, and `optionMaybe` means 0 or 1. This also reveals that any text written after the function call is simply ignored.

```
parseGourmet :: Parser Program
parseGourmet = do
    whiteSpace
    structs <- many parseStructDecl
    funcs <- many parseFunction
    functioncall <- optionMaybe parseFunctionCall
    return $ Program structs funcs functioncall
```

Listing 5.4: Parsing Gourmet programs

We have opted for an applicative programming style when developing the parser, which means that we use applicative functors. The four central ones are <$>, <*>, *>, and <*. An example where all are applied can be seen in Listing 5.5, where we parse function calls. Remember, the Psnodig FunctionCall data type is `FunctionCall String [Expression]`.

<$> allows us to put everything on the right as arguments to `FunctionCall`. <* means that we parse both the left side and the right side, but only keep the left side. *> means the opposite. <*> means that we keep both.

```
parseFunctionCall :: Parser FunctionCall
parseFunctionCall =
    FunctionCall
        <$> identifier
        <* reserved "("
        <*> parseExpr `sepBy` comma
        <* reserved ")"
```

Listing 5.5: Parsing Gourmet function calls

5.3.2 Writer

Creating a writer when you already have a parser is manageable. In fact, the writer is more or less just a reverse engineering of the parser. The only potential difference between a Gourmet program we write, and the corresponding Gourmet program Psnodig can give us, is the formatting.

Since Psnodig does not take formatting into consideration, it has no way of knowing the combination of whitespace we used in our program. Therefore, we use the function in Listing 5.6 to add indentation. This is used when entering new scopes in loops and if-statements, to add readability to our programs. Thus, in a way Psnodig works like a linter: By transpiling our Gourmet programs, we receive semantically identical programs, but with the intended amount of whitespace.

```
addIndents :: Int -> String
addIndents n = replicate n '\t'
```

Listing 5.6: A function for adding indentation

The monad we opted for here is a simple `Writer String`. This allows computations to produce a String as the final result. We traverse the AST and use the `tell` function to write the corresponding Gourmet syntax. The final result is therefore just a combination of strings added together.

5.4 TBP Writer

The monad we opted for was similar to the one of the Gourmet writer. However, before we write our pseudocode, another program skims the AST and grabs some valuable information. This informations, like function names and whether or not there are return statements, is then stored in a tuple of sets. The first set contains function names, and the second set contains keywords. The TBP writer then uses this information to utilise Algorithm2e better.

The monad is `ReaderT Environment (Writer String)`. The last part is already covered in the previous section. However, we are now also working with `ReaderT Environment`. `ReaderT` adds the ability to read from a fixed environment. In our case, `Environment` is the set tuple that contains all the information collected from the AST.

5.5 IBP Writer

Sparer denne til den er mer ferdig.

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.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[3] A. Alhefdhi et al. "Generating Pseudo-Code from Source Code Using Deep Learning". In: *25th Australasian Software Engineering Conference, ASWEC 2018, Adelaide, Australia, November 26-30, 2018.* IEEE Computer Society, 2018, pp. 21–25.

[4] G. M. Bierman, M. Abadi, and M. Torgersen. "Understanding TypeScript". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* Ed. by R. E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 257–281.

[5] D. Bjorner and M. Henson. *Logics of Specification Languages.* Springer, 2007.

[6] M. Carlisle et al. "RAPTOR: Introducing programming to non-majors with flowcharts". In: *Journal of Computing Sciences in Colleges* 19 (Jan. 2004), pp. 52–60.

[7] K. Charntaweekhun and S. Wangsiripitak. "Visual Programming using Flowchart". In: *2006 International Symposium on Communications and Information Technologies.* 2006, pp. 1062–1065.

[8] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* Ed. by M. Odersky and P. Wadler. ACM, 2000, pp. 268–279.

[9] M. Clavel et al. "Principles of Maude". In: *First International Workshop on Rewriting Logic and its Applications, RWLW 1996, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996.* Ed. by J. Meseguer. Vol. 4. Electronic Notes in Theoretical Computer Science. Elsevier, 1996, pp. 65–89.

[10] D. H. Clements and D. F. Gullo. "Effects of computer programming on young children's cognition." In: *Journal of educational psychology* 76.6 (1984), p. 1051.

[11] T. H. Cormen et al. *Introduction to Algorithms.* 3rd ed. Cambridge, MA: MIT press, 2009.

[12] 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.

[13] F. Domini. *An overview of Pandoc.* 2014. URL: `https://www.tug.org/TUGboat/tb35-1/tb109dominici.pdf` (visited on 02/13/2024).

[14] R. Feynman and C. Objectives. “Ebnf: A notation to describe syntax”. In: *Cited on* (2016), p. 10.

[15] K. Gauen et al. “Comparison of Visual Datasets for Machine Learning”. In: *2017 IEEE International Conference on Information Reuse and Integration, IRI 2017, San Diego, CA, USA, August 4-6, 2017.* Ed. by C. Zhang et al. IEEE Computer Society, 2017, pp. 346–355.

[16] 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.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[18] P. Hudak and J. H. Fasel. “A gentle introduction to Haskell”. In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52.

[19] Intel Corporation. *MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users.* 1979.

[20] L. H. Karlsen. “A Study In Monads”. MA thesis. 2013.

[21] D. Kreher and D. Stinson. “Pseudocode: A LATEX Style File for Displaying Algorithms”. In: (Oct. 1999).

[22] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms.* HarperCollins, 1991.

[23] LINFO. *Algorithms: A Very Brief Introduction.* 2007. URL: `http://www.linfo.org/algorithm.html`.

[24] M. Lipovaca. *Learn you a haskell for great good!: a beginner’s guide.* no starch press, 2011.

[25] H. Liu et al. “Pay Attention to MLPs”. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual.* Ed. by M. Ranzato et al. 2021, pp. 9204–9215.

[26] H. Lunnikivi, K. Jylkkä, and T. Hämäläinen. “Transpiling Python to Rust for Optimized Performance”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation - 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings.* Ed. by A. Orailoglu, M. Jung, and M. Reichenbach. Vol. 12471. Lecture Notes in Computer Science. Springer, 2020, pp. 127–138.

[27] M. Marcelino and A. M. Leitão. "Transpiling Python to Julia using PyJL". In: *Proceedings of the 15th European Lisp Symposium, ELS 2022, Porto, Portugal, April 21-22, 2022.* Ed. by D. Verna. ELSAA, 2022, pp. 40–47.

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

[29] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[30] Y. Megdiche, F. Huch, and L. Stevens. "A Linter for Isabelle: Implementation and Evaluation". In: *CoRR* abs/2207.10424 (2022). arXiv: `2207.10424`.

[31] T. Nicolini, A. C. Hora, and E. Figueiredo. "On the Usage of New JavaScript Features Through Transpilers: The Babel Case". In: *IEEE Softw.* 41.1 (2024), pp. 105–112.

[32] S. Nita and S. Kartikawati. "Analysis Of The Impact Narrative Algorithm Method, Pseudocode And Flowchart Towards Students Understanding Of The Programming Algorithm Courses". In: *IOP Conference Series: Materials Science and Engineering* 835.1 (2020), p. 012044.

[33] H. P. Nordaunet et al. "Competitive Reinforcement Learning Agents with Adaptive Networks". In: *11th International Conference on Control, Mechatronics and Automation, ICCMA 2023, Grimstad, Norway, November 1-3, 2023.* IEEE, 2023, pp. 314–319.

[34] B. Ó. Nualláin. "Executable Pseudocode for Graph Algorithms". In: *Proceedings of the 8th European Lisp Symposium (ELS 2015), London, England, April 20-21, 2015.* Ed. by J. A. Padget. ELSAA, 2015, pp. 47–54.

[35] 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.

[36] S. Peyton Jones et al. "The Glasgow Haskell compiler: a technical overview". In: (1998).

[37] B. C. Pierce. *Types and programming languages.* MIT press, 2002.

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

[39] M. Stefanowicz and A. Sasak-Okon. "AlgoPoint as an Original Didactic Tool for Introductory Programming Using Flowcharts". In: *Proceedings of the 15th International Conference on Computer Supported Education, CSEDU 2023, Volume 1, Prague - Czech Republic, April 21-23, 2023.* Ed. by J. Jovanovic et al. SCITEPRESS, 2023, pp. 162–170.

[40] P. Wadler. "Monads for Functional Programming". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text.* Ed. by J. Jeuring and E. Meijer. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 24–52.

[41] M. B. Wells and B. L. Kurtz. “Teaching multiple programming paradigms: a proposal for a paradigm general pseudocode”. In: *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1989, Louisville, Kentucky, USA, February 23-24, 1989*. Ed. by R. A. Barrett and M. J. Mansfield. ACM, 1989, pp. 246–251.

[42] X. Wu et al. “Research and Application of Code Automatic Generation Algorithm Based on Structured Flowchart”. In: *J. Softw. Eng. Appl.* 4.9 (2011), pp. 534–545.

[43] S. Xu and Y. Xiong. “Automatic Generation of Pseudocode with Attention Seq2seq Model”. In: *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. IEEE, 2018, pp. 711–712.

[44] J. Yuan et al. “A survey of visual analytics techniques for machine learning”. In: *Comput. Vis. Media* 7.1 (2021), pp. 3–36.