UNIVERSITY
OF OSLO

Master's thesis

Psnodig: Converting Source Code to Pseudocode

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: Converting Source Code to Pseudocode

A tool for converting source code to presentation targets

Abstract

Pseudocode is commonly used to provide a description of an algorithm at a suitable level of abstraction. Pseudocode is technically void of any syntax rules, allowing us to omit or de-emphasise certain aspects of our programs. However, with this amount of freedom, it is easy to accidentally misplace parantheses, fail to consistently rename variables, or even demonstrate wrong formulas. Additionally, we are now tasked with maintaining both versions.

In this thesis, we present **Psnodig**, a transpiler which converts source code to presentation-only based formats pseudocode and flowcharts. This way, we are able to write our programs in an imperative, high-level language, test them, and when satisfied, change their abstraction levels for presentation. This way, we are only maintaining one version, and the burden of manually ensuring consistency and searching for the appropriate level of abstraction is lifted off our shoulders.

Psnodig aims to be **extensible** (adding parsers at will), **executable** (running the source code we write), and **presentable** (automatically converting the source code to a presentation-only version). To evaluate these properties, selected algorithms from Algorithm Design and Applications by Michael T. Goodrich and Roberto Tamassia were chosen as a case study. The results show us that Psnodig does indeed posess all these traits, being the first of its kind to our knowledge.

Acknowledgements

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 natural language description of a problem solution [37].

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-emphasise certain aspects of their algorithms. Consequently, pseudocode is first and foremost aimed to serve as a tool for presentation.

Consequently, as pseudocode is not executable, there is no omniscient way to verify its correctness. This can, in turn, lead to accidental inclusion of critical inaccuracies. When working within an integrated development environment, most popular languages have support for static analysis tools that detect anti-patterns and warn about bad practices [44, 46]. When writing pseudocode, on the other hand, we are left to our own devices, as identification of anti-patterns and bad practices generally relies on the existence of established standards.

1.1 Motivation

Correct presentations are important in education where the goal is to teach students concepts they were previously unfamiliar with. At university level, concepts within algorithms and data structures have traditionally proved challenging for undergraduates [11, 30, 57]. If their first impression of an algorithm is an incorrect presentation, their path is already hampered.

In this thesis we present a tool called **Psnodig** (pronounced snoo-dee[1]), which allows us to convert source code to other, less technical presentations. The presentation targets in this thesis are mainly pseudocode and flowcharts. The

[1]Imagine the **sn** in ***snow***, the **oo** in ***cool***, and the **dee** in ***deep***. The silent **p** is an ode to the silent p in the word ***pseudocode***, naturally.

target audience is people who study or work with algorithms.

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 [56, 19, 60]. However, tools for direct conversion from source code to flowcharts does not seem to be widespread.

We spend much more time *reading* code than we do writing it [43, 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 Psnodig can be a valuable tool for authors who want to present their algorithms, as well as students who wish to get a better understanding of them.

We aim to promote algorithmic thinking through various forms of representation. We believe that this can aid us to better understand and modify our 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.

Another key point is that the software that *do* let us convert source code to other representations, are completely independent of each other. Most of them also operate with their own domain-specific language to write source code, which makes them even more unavailable. Psnodig, on the other hand, attempts to centralise all these resources, making it a powerful all-in-one tool for any conversion we might wish for.

1.2 Goals

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

- **Extensible**, users can add parsers and code generators to Psnodig.
- **Executable**, users can run the source code they have written.
- **Presentable**, users can use Psnodig to convert their source code to a different level of abstraction.

We will use selected algorithms from [22] to evaluate to which degree the tool posesses these properties in Chapter 6. We will then discuss them more in depth in Chapter 7.

1.3 Contributions

The main contribution of this thesis is the Psnodig tool, which facilitates the conversion of executable source code to a number of different formats. The tool gives people an easy and accessible way to look at their code from a different perspective.

By using the Psnodig tool, we aim to help people to spend more time writing meaningful code and less time mastering LaTeX libraries, writing boilerplate code, and worrying about maintaining multiple sources.

We are able to add parsers and writers to Psnodig at will, making it a flexible tool. Psnodig is materialised through the following implementations:

- A parser for an imperative, C-like programming language we have coined **Gourmet**.
- Two writers converting Psnodig ASTs to pseudocode and flowcharts.
- Two writers converting Psnodig ASTs to C-like programs and Python-like programs.

To the best of our knowledge, Psnodig is the first tool of its kind for transpiling source code to these formats, in additional to being openly available and extensible.

1.4 Project Source Code

All the source code from the master thesis can be found on Github.[2]
`NOTE: Må huske å gjøre repoet public!`

[2]More specifically, the entire thesis work can be found at `https://github.com/sergiosja/Master`. It is actually divided into two folders: ***psnodig***, containing the source code for the Psnodig tool, and ***thesis***, containing the LaTeX source code for this thesis.

Chapter 2

Background

This chapter will cover key concepts one should be familiar with in order to fully understand the rest of this thesis. We start by providing a definition for pseudocode, before discussing how transpiling works. Lastly, we discuss Haskell as an implementation language. We will round off by looking at Pandoc, a transpiler written in Haskell.

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 instead put their focus 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 [37].

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. As Donald Knuth famously put it, after presenting some algorithm implementations in pseudocode [62]:

```
Beware of bugs in the above code; I have only proved
it correct, not tried it.
```

When explaining a solution to a non-technical audience, it makes more sense to use pseudocode than source code. More specifically, pseudocode that encapsulates the program's core functionality, to provide clarity on its essential aspects. This enables even individuals without a programming background to provide feedback, based on their understanding of both the problem and now also the proposed solution.

Now, since pseudocode has many faces, we must define what we really mean when we refer to pseudocode in the coming sections. In the context of this thesis, we believe that pseudocode can work as an umbrella term for both traditional pseudocode flowcharts.

2.1.1 Traditional Pseudocode

The most conventional form of pseudocode, commonly found in text books on algorithms, published papers, as well as design documents discussing problems [10, 22, 39, 20]. 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 [66, p. 247].

Since there is no formal set of rules commanding how pseudocode should look, we are prone to viewing different variations of the same algorithms across different literatures. A frequently presented algorithm is **Binary search**, 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 [10].

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 [48]. 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]. Roughly 17 years later, Lewis et al. present their own version in Data Structures and Their Algorithms [36, p. 182]. The underlying logic stays much the same, though the approaches and syntaxes are different, as seen in Listing 2.1 and Listing 2.2, respectively.

```
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)
```

Listing 2.1: Binary Search by Aho et al.

```
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
```

Listing 2.2: Binary Search by Lewis et al.

The wish for automatic generation of pseudocode has been desired for some time, with the intention of presenting ideas without having to worry about the syntax of a particular programming language [68]. Traditional pseudocode allows us to draft ideas in an imperative fashion, the same way we write baking recipes and instructions for 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 [10, 51].

As previously mentioned, pseudocode has a well-established history in university curricula. When learning algorithms, data structures, and programming concepts in general, the focus is really on the underlying ideas. The concepts are generally more important than the specifics of how they are implemented in a particular programming language. Thus, the approach of learning with pseudocode prioritises concept comprehension over language-specific knowledge.

The usefulness of a tool that converts source code to pseudocode can be backed by a number of factors. For one, existing online translators like Code Kindle.[1] There also exists research on the topic, which we discuss further in Section 3.3.1. Lastly, there are still being created new styles for dislaying pseudocode [34].

2.1.2 Flowcharts

Flowcharts - in the context of computer science - are a way to model computer programs. Also called flow diagrams, they contain nodes (shapes with text) explaining computational steps, and arrows showing us how the programs proceed. This gives a step-by-step overview of computer programs.

Imperative programming languages, like Python, 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.

On the other hand, there are languages with different execution flows. For instance, all processes in a VHDL description are executed concurrently [31]. 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.

The imperative 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 shapes, arrows and colours. When code stretches over enough lines, it often becomes uniform in appearance and more challenging to comprehend. By contrast, flowcharts explicitly capture the control flow of the program, and makes it possible to direct our focus.

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

[1] Works with C++ and Python, can be found at `https://devpost.com/software/code-kindle`. Its source code is also available at `https://github.com/Open-Sourced-Olaf/Code-Kindle`.

betwee deterministic and non-deterministic FSAs, although that 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 [26].

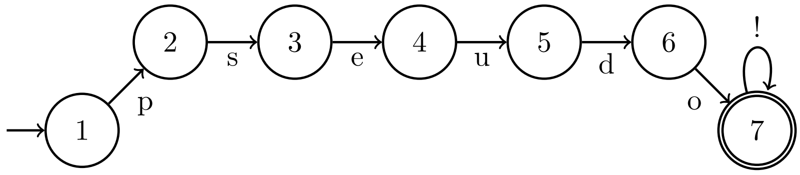

Figure 2.1: A finite state automata.

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 7 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. State 7 has an additional ring within its circle, which means that it is an accepting state. If a combination of symbols have not been rejected at this point, and is finished, it is accepted.

State 7 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 7, 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, back in 1943 [45]. Their paper presents a simplified computational model of biological neurons.

The first flowcharts in computer science - to the best of our knowledge - were introduced a few years later, by Herman H. Goldstine and John Von Neumann. They used them to depict the flow of programs in "electronic computing instruments" [21].

Also to the best of our knowledge, there does not exist a flourishing amount of research on converting source code to corresponding flowcharts. For instance, when Zhang et al. converted source code to flowcharts for the purpose of plagiarism detection, they used the paid software Visustin [70]. There *does* exist software to convert source code to flowcharts on the web, like Code2Flow and Mermaid.[2]

There have been multiple attempts to create flowchart *editors*, most notably by Carlisle et al. and Charntaweekhun et al. [7, 8]. These allow us to build flowcharts using the drag-and-drop approach, rather than keeping it all in text form. Benefits of learning with help from visual aid is well documented, and

[2]These software are discussed in more detail in Section 3.3.2.

when it comes to computer science, visualisations are especially prominent in the context of machine learning [50, 17, 69].

Given the imperative nature of flowcharts, the way they walk through problems step-by-step, it should be no surprise that people have attempted converting flowcharts to pseudocode. Wu et al. has proposed a structure identification algorithm, which can take an identified flowchart as input and automatically generate code in return [67]. 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 1980s by Scanlan et al. He recorded 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 [56].

More recently, Nita et al. 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 [49].

2.1.3 LaTeX

LaTeX is a document preparation system that is widely used for the production of scientific documents.[3] It is an open-source typesetting system recognised for its capabilities in creating visually appealing documents that meet typographic 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 to produce a formatted document, typically in PDF format.[4]

This is a contrast to more ubiquitous word processors like Microsoft Word or Google Docs, which abide to WYSIWYG principles.[5] 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 has an additional collection of macros that simplifies the use of TeX, and makes it more accessible to non-technical users [32, p. 7].

[3] In fact, this thesis is written in LaTeX.
[4] The concept of compilation is discussed more in-depth in Section 2.3
[5] WYSIWYG is an acronym for ***What You See Is What You Get***.

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.

As mentioned in Section 1.3, two LaTeX packages are central to our contributions: `Algorithm2e` and `TikZ`. The former is a package to typeset computer programs, whilst the latter, on the other hand, is arguably the most complex and powerful tool to create graphic elements in LaTeX [16, 61].[6]

Algorithm2e

After importing the Algorithm2e package, we are free to typeset computer programs within an **algorithm** environment.

There are many macros we can use to typeset our programs. Common ones include **caption**, to caption our programs, and **LinesNumbered**, to number the lines of our programs. Each line ends with a semicolon by default, but we can remove them by including **DontPrintSemicolon**. If we want to add a description of desired input and output above the program, we can do so using **KwIn{}** and **KwOut{}**, respectively.

We can define our own keywords by using the `\SetKw{}{}` macro. If we define `\SetKw{KwBreak}{break}`, we can add `\KwBreak` to our LaTeX, which is then displayed as **break** in the compiled version.

There are also more specific macros, like `\SetKwProg{Prog}{Title}{is}{end}`. `Prog` is what we refer to in our LaTeX, `Title` is what is displayed (like **break** in the previous paragraph), `is` directly follows `Title`, and `end` is the last line of the program. We can write multiple programs within the same **algorithm** environment. Another common macro is `\SetKwFunction{f}{f}`. This allows us to write `\f{arg1, .., arg n}` in LaTeX, to display `f(`*arg1, .., arg n*`)` in the compiled version.

Algorithm2e comes with multiple predefined English keywords that are commonly found in computer programs, like **uIf**, **uElseIf** and **uElse** for conditionals, and **While**, **For** and **Repeat** for loops.[7]

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

---

[6] TikZ was actually used to create the FSA example in Figure 2.1.

[7] The entire documentation can be found here: `https://ctan.mirror.garr.it/mirrors/ctan/macros/latex/contrib/algorithm2e/doc/algorithm2e.pdf`.

Algorithm: A program to demonstrate Algorithm2e

```
Procedure f() is
    First statement;
    Another statement;
    if conditional then
        Statement 1;
    else
        Statement 2;
    Last statement;
end
```

Figure 2.2: An example program utilising the Algorithm2e package in LaTeX.

TikZ

As previously stated, TikZ is a powerful tool, and can be used to create just about any graphic element in LaTeX. One of them is flowcharts, which can be constructed using three macros: `\tikzstyle`, `\node`, and `\edge`. `tikzstyle` lets us define the appearance of nodes and edges, whilst `node` and `edge` are used to draw nodes and edges, respectively.

Listing 2.3 gives a rough estimate of how a tikzstyle can be constructed.

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

Listing 2.3: Rough description of tikzstyle attributes.

Each tikzstyle has a unique name, so that it can be applied to nodes later on. The standard shapes are rectangles, but other shapes like diamonds and ellipses can also be imported through libraries. The three colours refer to a node's border-, text- and background colours, respectively.

Listing 2.4 gives a rough estimate of how nodes are written.

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

Listing 2.4: How nodes are written with TikZ.

All nodes should have a unique name, for the purpose of being referenced 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.

Listing 2.5 shows how most edges are written. It is important to note that nodes must already be defined prior to being referenced by an edge.

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

Listing 2.5: How edges are written with TikZ.

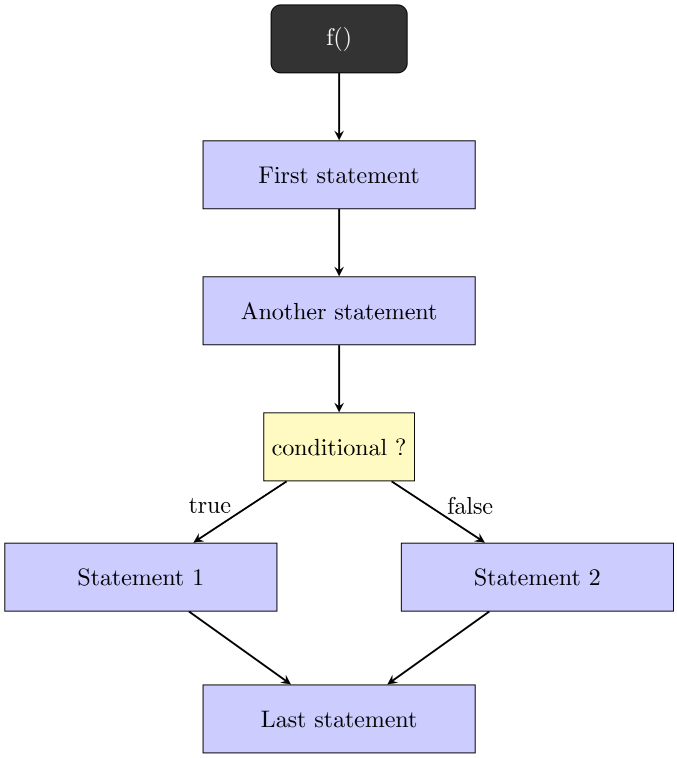

Figure 2.3: An example flowchart utilising the TikZ package in LaTeX.

2.2 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. It consists of a frontend and a backend. The frontend is often referred to as the analysis part (analysing the source program before constructing an intermediate representation), whilst the backend is referred to as the synthesis part (patches together the desired target program from that intermediate representation) [2].

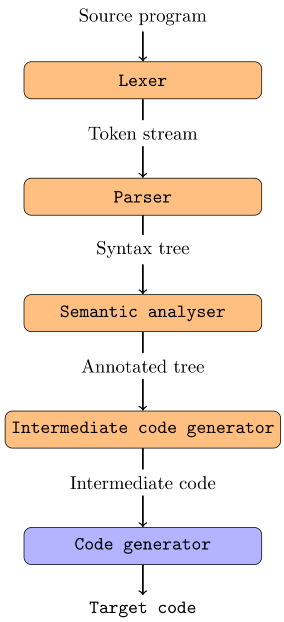

Figure 2.4: The phases of a typical compiler.

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.

The typical phases of a compiler is demonstrated in Figure 2.4. The orange nodes demonstrate the frontend, whilst the purple node demonstrates the backend. Some compilers also do more work, like optimising the intermediate- or target code, or both.

2.2.1 Source-to-Source Compilers

A transpiler, formally **source-to-source compiler**, is a tool that converts input source code to output source code, whilst maintaining a similar abstraction level [42]. The first transpiler to our knowledge was developed in 1978 by Intel, with the aim of translating Assembly source code from an 8080/8085 processor to an 8086 processor [28].

JavaScript, the world's most commonly used programming language [54], has a rich history of transpilation. As a language in constant development, it faces an issue where not all browsers are always compatible with its newest features. Babel is a transpiler that converts modern JavaScript into a backwards compatible version.[8] According to Nicolini et al., without a transpiler almost 14% of web users risk facing a JavaScript bug when accessing a website with new JavaScript features [47].

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 could likely be a thesis topic of its own.[9] However, we can bring forward some notable exambles.

Contrary to JavaScript, TypeScript is structurally typed. TypeScript is syntactically a superset of JavaScript, adding a layer of static typing. The primary purpose of these types is to enhance the development experience by catching potential errors during compilation, as well as making the code more maintainable. However, before the code is run, TypeScript is transpiled into plain JavaScript, and the types are stripped away [6].

AlaSQL is an open-source SQL database for JavaScript [18]. It lets us write SQL within a JavaScript context, i.e. creating tables and performing CRUD operations. Listing 2.6 shows how we can create a table, populate it with a few Italian football teams, before selecting the ones who meet our criterion.

```
alasql("CREATE TABLE
        teams (name string, points number)");

alasql("INSERT INTO teams
        VALUES ('SSC Napoli', 49),
               ('Salernitana', 15),
               ('Atalanta', 51)");

const topHalf = alasql("SELECT *
                    FROM teams
                    WHERE points < 40
                    ORDER BY points
                    DESC");
```

Listing 2.6: JavaScript code to create, populate and select a table with AlaSQL.

Despite all the commands being written in SQL, the selected table will have a JavaScript value, as seen in Listing 2.7

[8] Babel can be accessed on https://babeljs.io

[9] This overview https://gist.github.com/matthiasak/c3c9c40d0f98ca91def1 provides a list of 320 languages and tools that compile to JavaScript.

```json
[
    {
        "name": "Atalanta",
        "points": 51
    },
    {
        "name": "SSC Napoli",
        "points": 49
    }
]
```
Listing 2.7: The value of topHalf from Listing 2.6

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 convert programs in their favourite languages to JavaScript. This 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.

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, the Haskell compiler GHC (Glasgow Haskell Compiler) used to convert the code to C rather than direct native code generation. This enabled Haskell to run on any platform with a C compiler. It also benefitted directly from others' improvements in C code generation [55].

Another example is a transpiler presented by Lunnikiv et. al, where Python is converted to Rust as an intermediate source code step [41]. The paper shows how pre-existing Python implementations that depend on optimised libraries can be semi-automatically transpiled to Rust. This way, the user can keep writing Python whilst simultaneously basking in the glow of Rust's performance optimisation.

2.2.2 Parsers and Code Generators

We mentioned that the frontend of a compiler is tasked with reading source code, and — given that it is syntactically correct — building 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 key points of this thesis. Nevertheless, two important aspects of a compiler that *will* be central focus points, are parsers and code generators. Parsers and code generators play a vital role also in a transpiler. In fact, they are all we really need to build a simple transpiler. When the parser has converted the source code to an intermediate representation, the code generator can convert that intermediate representation into a target program.

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 or 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 in practice is the programming language **Derw**, an ML language mainly inspired by Elm [23]. It comes with a single parser (for the Derw language), but multiple code generators (referred to as just generators) which target JavaScript, TypeScript, Elm as well as English and Derw itself. Since it is open source, anyone can fork the repository and add their own code generator, if they so desire.

Listing 2.8 shows how expressions like `6 <= 8` are converted from Derw to English. 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`.[10]

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

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

Listing 2.8: The function that converts a `Less than or equal`-expression in Derw to English.

2.3 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, which accounts for a large part of the thesis.

2.3.1 Data Types

Haskell lets us create new data types with the `data` keyword. In reality, types and data types are the same thing, but the `type` keyword is reserved to make type aliases. For the remainder of the thesis, type and data type will be used

[10] The entire English code generator is located at `https://github.com/eeue56/derw/blob/main/src/generators/English.derw`

interchangeably.

A common type in any programming language is the boolean, shown in Listing 2.9. 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 a disjunction [38, p. 109].

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

Listing 2.9: Recreating the Boolean type with Haskell.

With Haskell, it is straightforward to create our own data types, which can then be used to model ASTs. For instance, we can create a simple calculator language in just a few lines of code, as shown in Listing 2.10. From this, we can construct an AST as the one we see in Listing 2.11, intended to model the mathematical expression `1 + 2 - 3`.

```
data Program = Program Expression

data Expression =
      CompoundExpression Integer Operator Expression
    | IntExpression Integer

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

Listing 2.10: Data types for a calculator language in Haskell.

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

Listing 2.11: An AST constructed with data types presented in Listing 2.10

The calculator language is endlessly expressive within the limits of integers and the operation values we defined. If we wish to expand our operator data type, we only have to add a pipe and the new value name, as seen in Listing 2.12.

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

Listing 2.12: An extended version of the Operator data type presented in Listing 2.10

2.3.2 Pattern Matching

Another integral part of Haskell is its strong type system, which allows 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.

An example of pattern matching is demonstrated in Listing 2.13, with a function converting each value of the Operator type (introduced in Listing 2.10) to its string equivalent. The function `convert` takes a value of type Operator as input, and returns a corresponding value of type String.

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

Listing 2.13: Haskell function converting values of one type to another.

If the function shown in Listing 2.13 was to work on the extended Operator data type from Listing 2.12, the compiler would let us know that our pattern matching is non-exhaustive, 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 2.14 shows how the underscore symbol can be used to capture all remaining values of a type.

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

Listing 2.14: A rather lazy Haskell function attempting to convert values of type Integer to its string equivalent.

2.4 Pandoc

Pandoc is an example of a transpiler written in Haskell, which converts between different markdown formats [14]. It includes a Haskell library, as well as a command-line program. It is able to convert documents from 45 source formats to 63 target formats.[11] Additionally, Pandoc is able to convert documents in LaTeX, Groff ms and HTML into PDFs.

At its core, Pandoc is really just a specification of Haskell data types. A Pandoc document has the type `Pandoc Meta [Block]`. The first attribute is `Meta`, a document's metadata 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

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

type, shown in its entirety in Listing 2.15.

```
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 2.15: The `Block` data type of Pandocs native representation.

The modular design of Pandoc means that adding an input format only requires the program to convert *to* this native representation, whilst an output format only needs to convert *from* it. 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`. Throughout this thesis, the words `writer` and `code generator` will be used interchangeably.[12]

Though it must be said, Pandoc cannot always ensure a completely lossless conversion. As the native representation is less expressive than many of the formats it converts between, details are bound to get lost on the way. For instance, LaTeX is a notoriously rich format, much more so than many of its companions in the Pandoc dynasty. However, Pandoc is an excellent interpreter of lightweight markup languages like Markdown, which are more *neutural* by design [14].

To demonstrate an example of how Pandoc works, we provide an example of how a Markdown program can be transpiled to LaTeX. Listing 2.16 is the Markdown program in question, and Listing 2.17 partly shows the final LaTeX file.[13] Listing 2.18 also shows its 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
```

[12] In reality, traditional code generators do much more than directly translating data types to a target program. In the context of this thesis, however, code generators will not be asked to do much more. This is also a way to acknowledge that the writers we ultimately construct are not fully fledged code generators like the ones we often see in compilers.

[13] The final LaTeX document actually comes with 50 more lines of boilerplate generated by Pandoc. However, we can safely remove them, and leave only what we see in Listing 2.17. Compiling this and compiling the original result with boilerplate yields the exact same PDF.

```
This is a paragraph about programming languages
```

Listing 2.16: A program written in Markdown.

```
\documentclass[]{article}
\title{Programming languages}
\author{}
\date{2023-02-01}

\begin{document}
\maketitle

\section{Introduction}\label{introduction}

This is a paragraph about programming languages

\end{document}
```

Listing 2.17: The result of transpiling Listing 2.16 to LaTeX.

```
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"
           ]
    ]
```

Listing 2.18: The internal Pandoc AST of Listing 2.16.

Chapter 3

Analysis

In this chapter, we will be doing problem analysis. We will dive deeper into the problem we are addressing, and why we believe it to be a problem in the first place. We also compare source code with our definition of pseudocode, to get a better feel for their differences. Lastly, we will discuss some of the existing approaches, which already solve parts of the problem.

3.1 Problem Definition

Let us outline the problem at hand more concretely. Simply put, we aim to present computer programs at different abstraction levels, whilst maintaining their underlying ideas. The concept is nothing new, but has traditionally been carried out manually by the programmer. We believe there is a benefit in centralising resources and automating the conversion part.

There are several use cases where it is useful to display source code in a different light. Mainly in computer science education, where the goal often is to teach students concepts in a programming language agonstic way. It could also be used by researchers to exchange ideas across preferred programming languages.

When teaching computed programming at e.g. university level, it is important to distinguish between a student's grasp of computer science topics and their mastering of a particular programming language. Even though a student struggles with the syntax of, say, Go, it does not mean they have an inability to understand the broader concepts. Thus, the teacher can level the playing field by opting for pseudocode when introducing new concepts.

Shackelford and LeBlanc Jr. go as far as to say that using pseudocode can "free students from the myriad of programming language details that annoy and distract attention from the core issue of algorithmic problem solving" [58]. This could also apply to researchers who might wish to exchange ideas across different programming languages.

On the other hand, flowcharts can offer a visual approach to understanding programming concepts. If a teacher presents their ideas in the form of flowcharts, the classroom can focus on understanding its underlying ideas. If the ideas are presented in source code, students might be distracted by syntactic nuances of the teacher's preferred programming language.

In the context of our problem, and the remainder of this thesis, we treat both traditional pseudocode and flowcharts as pseudocode. To make a distinction, we introduce the terms **Text based pseudocode** (TBP) and **Image based pseudocode** (IBP).

3.2 Comparing Source Code with Pseudocode

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 is a solution to the `FizzBuzz problem`, commonly presented in entry level interview settings and beginner programming exercises. The input is an integer `n`, and the output depends on the integer's value: if `n` is divisible by 3 and 5 we return `FizzBuzz`, if `n` is divisible by 3 but not 5 we return `Fizz`, and if `n` is divisible by 5 but not 3 we return `Buzz`. If `n` is not dividible by either, we simply return `n`.

Listing 3.1 shows the program written in the Go programming language. Figure 3.1 and Figure 3.2 show the same program, but with TBP and IBP, respectively.

The biggest difference between the Go program and the pseudocodes, is that the former contains boilerplate code. For instance, we have to declare which package the file belongs to. In our case, we simply call the package `fizzbuzz`. We also have to import a package for converting integers to strings, to convert `n` before returning it in the last case. This is because functions can only have one return type in Go.

```go
package fizzbuzz

import "strconv"

func FizzBuzz(n int) string {
    if n%3 == 0 && n%5 == 0 {
        return "FizzBuzz"
    } else if n%3 == 0 {
        return "Fizz"
    } else if n%5 == 0 {
        return "Fizz"
    }
    return strconv.Itoa(n)
}
```

Listing 3.1: A Go program solving the FizzBuzz problem.

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 common program language syntax like `else if` and `return`.

One of the differences is how the TBP version has a specific place for describing input and output. With Go, we would either have to them with comments, or leave the users to guess the desired output.

Additionally, since TBP is intentionally not executable, we given total control over what we wish to include and exclude. We are free to emphasise particular parts, and also omit non-essential details. This allows us to increase the understanding of our programs more easily.

Algorithm: A solution to the FizzBuzz problem.

Input: An integer n.
Output: $FizzBuzz$, $Fizz$, $Buzz$ or n, depending on the value of n.

Procedure `FizzBuzz` *(n)*
  if $n \ \% \ 3 = 0 \wedge n \ \% \ 5 = 0$ **then**
    return FizzBuzz
  else if $n \ \% \ 3 = 0$ **then**
    return Fizz
  else if $n \ \% \ 5 = 0$ **then**
    return Buzz
  else
    return n

Figure 3.1: Pseudocode of a program solving the FizzBuzz problem.

All 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 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 flowcharts are not executable, we can again opt for more precise mathematical notation to describe expressions, and exclude static properties like types.

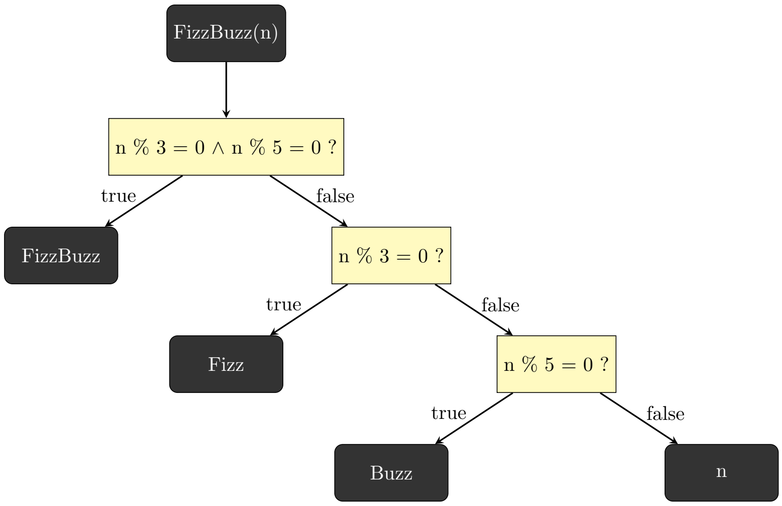

Figure 3.2: Flowchart of a program solving the FizzBuzz problem.

3.3 Previous work

This section covers selected works and software that have already solved parts of the problem, in their own right. The first section shows different approaches to convert source code to pseudocode, by applying machine learning.

The second section shows interplay between source code and flowcharts, converting one to the other. We also discuss the most basic way of converting between these formats: by doing it manually.

3.3.1 Pseudocode

Despite TBP being used in so many textbooks, online courses and published papers, there are not an overwhelming amount of source code-to-pseudocode editors currently available online. Majority of research on this topic, to the best of our knowledge, seems to be centered around machine learning approaches.

Manual Approach

The benefits of manually pseudocode manually, is that we are void of any restrictions. We can select the desired abstraction level, we can emphasise what we wish, and the visual formatting is entirely up to us.

The biggest downside is the extra effort it requires to maintain two versions of our computer programs. We might change our source code but forget to update the pseudocode. Sometimes we might fail to consistently rename variables, or accidentally misplace parentheses and demonstrate a wrong formula.

Machine Learning Approach

In 2015, Oda et al. introduced **Pseudogen**, a tool for converting Python source code to pseudocode using a statistical machine translation (SMT) approach [52]. The pseudocode produced is really a line-for-line description of the Python program provided as input.

SMT is a technique to train machine learning models on samples of translations supplied by humans [40]. It is most commonly used for translating between two natural languages, for instance from English to Italian. In the context of Pseudogen, SMT is used to translate Python to English and Japanese.

Despite being a programming language notoriously known for sticking to natural language where many others use more technical notation,[1] Python still bears the mark of being a programming language. People unfamiliar with programming might still struggle to understand some of the more technical aspects of its syntax, like decorators and closures.

Figure 3.3 shows an excerpt taken from the 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, that we also looked at in Section 3.2.

```
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.
```

Source code (Python) | Pseudo-code (English)

Figure 3.3: Python source code translated to pseudocode with Pseudogen. This example is not generated by us, but taken directly from [52].

Since Pseudogen will translate each line in a servile manner, all error handling is translated too. Listing 3.2 shows some error handling in Python, and Listing 3.3 shows the result from transpiling it with Pseudogen. The result is considerably more verbose, which defeats some of the point with Python, whose syntax tends to be elegant and succinct, already closely resembling English.

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

Listing 3.2: Error handling in Python.

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

[1] The best example might be **and** over **&&**, and **or** over **||**, which we find in more or less all other programming languages.

Listing 3.3: Pseudocode version of Listing 3.2 generated with Pseudogen. This example is not generated by us, but copied from a video on their website.

Another example is visible in Listing 3.4 and Listing 3.5, where list comprehension has been translated quite 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 [4, p. 11].

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

Listing 3.4: 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 3.5: Pseudocode version of Listing 3.4 generated with Pseudogen. This example is not generated by us, but copied from a video on their website.

In 2018, Alhefdhi et al. introduced **Code2Pseudocode**, a machine learning model which also utilises machine translation techniques. The output follows the format of Pseudogen, but rather than sticking with SMT, they have opted for Neural Machine Translation (NMT).

NMT is another variant of machine translation, inspired by neural networks in the human brain. The approach has become the standard for large-scale machine translation, according to a 2019 review and survey carried out by Stahlberg [59].

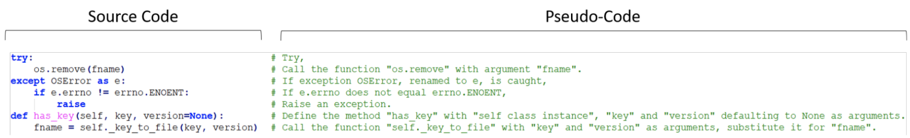

Figure 3.4: Python source code translated to pseudocode with Code2Pseudo -code. This example is not generated by us, but taken directly from [3].

Figure 3.4 shows an excerpt taken from the 2018 paper where Code2Pseudocode is presented. It is similar to Figure 3.3 in nature: Python source code on the left, and generated psudocode on the right. The program tries to remove a file from the computer, and raises an error if it fails.

Since neither Pseudogen nor Code2Pseudocode have knowledge of Python itself, the descriptions they provide are very general. For instance, the pseudocode in Figure 3.3 does not explain what an "exception" is, or what it really means to "throw an exception". Likewise, the pseudocode in Figure 3.4 does not tell us that `os.remove(fname)` will actually remove `fname` from our computer, only that it calls `os.remove()` with `fname` as argument.

Truthfully, recent advancements in artificial intelligence are able to do much of this for us now. For instance, the freely available prompt-based tool ChatGPT is now able to do this job entirely on its own.[2] Additionally, rather than giving a line-for-line translation, it is able to give a concise summary of what the program actually does. Figure 3.5 and Figure 3.6 are examples of asking it to summarise the Python programs we saw in Figure 3.3 and Figure 3.4, respectively.

ChatGPT

This Python program defines a function called "fizzbuzz" that takes an integer "n" as input. Here's what the program does in natural language:

1. Check if the input "n" is an integer. If not, raise a TypeError indicating that "n" is not an integer.
2. If "n" is divisible by 3 without a remainder, check if it's also divisible by 5 without a remainder:
   - If yes, return "fizzbuzz".
   - If not, return "fizz".
3. If "n" is not divisible by 3 without a remainder, but it's divisible by 5 without a remainder, return "buzz".
4. If "n" is neither divisible by 3 nor by 5 without a remainder, return "n" converted to a string.

Figure 3.5: ChatGPT giving a summary of the Python code from Figure 3.3.

ChatGPT

This Python code can be translated into natural language as follows:

1. Attempt to remove a file specified by the variable "fname" using the "os.remove" function.
2. If an "OSError" occurs during the attempt:
   - Check if the error code is not equal to "errno.ENOENT" (which typically means the file doesn't exist).
   - If it's a different error code, raise the same error.
3. Define a method named "has_key" that takes three arguments: "self" (which refers to the instance of the class), "key", and "version" (with a default value of "None").
4. Inside the method, assign the result of calling the "_key_to_file" method of the instance "self" with arguments "key" and "version" to the variable "fname".

Figure 3.6: ChatGPT giving a summary of the Python code from Figure 3.4.

[2] It is currently available here: `https://chat.openai.com/`.

3.3.2 Flowcharts

There is a good deal of research advocating for using flowcharts as an alternative to traditional code when demonstrating computer science concepts [56, 19, 60]. Research on ways to construct corresponding flowcharts from source code, however, is not plentiful. In this section, we will look at a 2015 paper, as well as online editors providing our desired transpilation.

Manual Approach

Like with TBP, the main benefit of translating our source code to flowchart manually is the lack of restrictions. We are free to use our preferred drawing tool, and can choose to emphasise parts of our program just the way we like - with colours, shapes, arrows, and more.

However, if we were to change the main ideas of our code, we would have to draw the flowchart again from scratch. This means that the time spent on making the original IBP version is - to a certain extent - wasted.

Another burden is that we are entirely responsible for how the corresponding flowchart should look. By doing this process manually, we have to spend time on the design, which is actually unrelated to the actual program logic.

Research-driven Approach

In 2015, Kosower et al. introduced **Flowgen**, a flowchart-based documentation framework for C++. The tool converts annotated C++ programs to high-level UML activity diagrams, providing a description of the code's dynamic behaviour [33].

Flowgen extracts control flow parts of a program, as well as annotations initiated by `//$`, to build flowcharts for each function or method. Understanding what C++ programs do is outside the scope of this thesis, with the relevant part being how Flogen represents a program written in executable source code as a flowchart.

Listing 3.6 shows a C++ program with Flowgen annotations, and Figure 3.7 shows the corresponding flowchart created by Flogen. We have not generated this flowchart ourselves, instead it is taken directly from [33].

```
#include \\aux.h’’
#include <iostream>
int main()
{
    int control flag=0;
    //$ ask user whether to proceed
    std::cin >> control flag;
    if (control flag==1){
        //$ call shower
        // pointer to the object VINCIA
```

```
        VINCIA* vinciaOBJ = new VINCIA();
        vinciaOBJ->shower(); //$
    }
    return 0;
}
```

Listing 3.6: A C++ program.

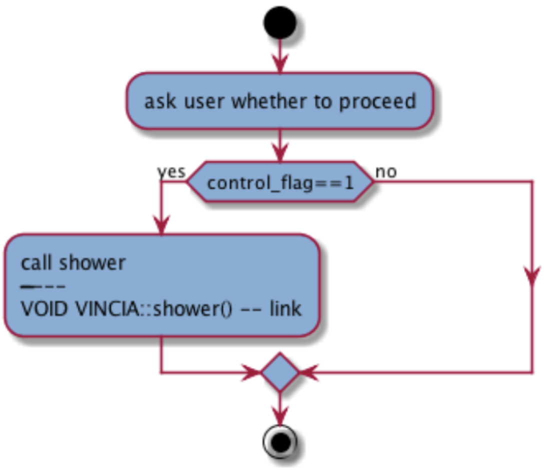

Figure 3.7: Listing 3.6 converted to a flowchart with Flowgen. This example is not generated by us, but taken directly from [33].

Online Tools

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 statement;
Another statement;
if (conditional) {
  True statement;
} else {
  False statement; // Random comment
}
```

```
Last statement;
```

Listing 3.7: A Code2Flow program.

Listing 3.7 shows a program written with Code2Flow, and Figure 3.8 presents the corresponding flowchart. As we can see, syntactically correct expressions are any combination of UTF-8 characters. Code2Flow will never warn us about syntactic errors, and will *always* try to construct whatever flowchart it can. In turn, there is no clear way to test a Code2Flow program.

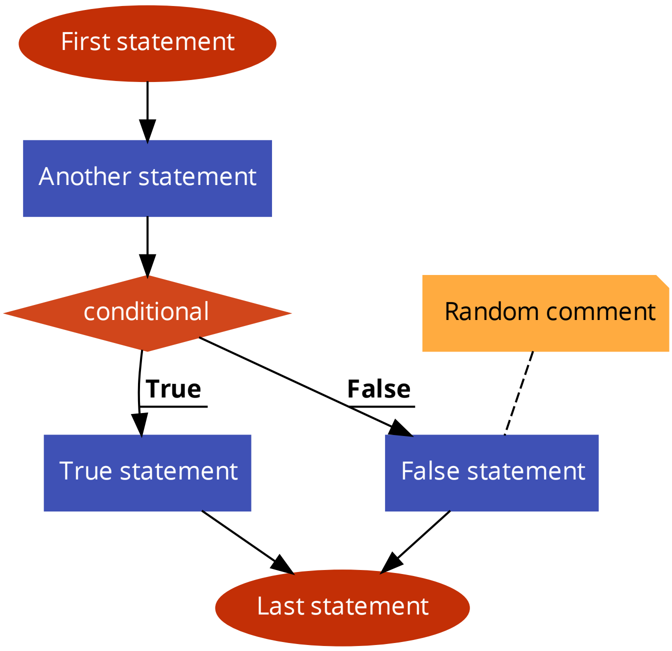

Figure 3.8: The resulting flowchart from transpiling the Code2Flow code in Listing 3.7

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

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

To construct a Mermaid.js flowchart, our source 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 dotted arrow, whilst − − −

will display a simple link.

We can also add text to our nodes, by placing it within the brackets, like `Node[text]`. Arrows can also include text by breaking them up into two parts, like `-- text -->`.[3]

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

Listing 3.8: A mermaid.js program.

Just like with Code2Flow, the text inside these nodes can be anything. Listing 3.8 shows a program written with Mermaid.js, and Figure 3.9 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. It also 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 our code.

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

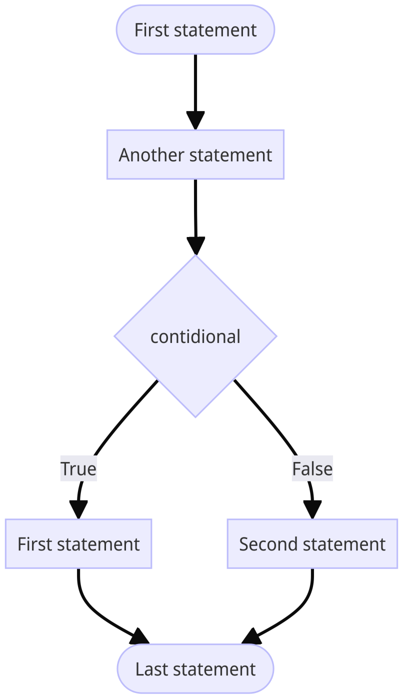

Figure 3.9: The resulting flowchart from transpiling the Mermaid.js code in Listing 3.8

Chapter 4

Design

In this chapter, we properly introduce Psnodig and relevant design decisions behind the tool. We also describe the design behind the Gourmet programming language, as well the desicions behind the writers we landed on.

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, first and foremost intended to convert executable programs to equivalent presentation-only programs.

We wish to alleviate common frustrations associated with pseudocode, like unbalanced parentheses, forgetting to consistently rename variables, issues related to performance and more. At the same time, we aim to preserve the good things that pseudocode let us have, like flexibility and abstraction, and using more precise mathematical notation.

The software discussed in Section 3.3 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 is ultimately too lossy.

Additionally, we believe these writers actually 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. Lastly, we see value in users adding their own parsers and code generators, to further increase the tool's usefulness.

As the programs discussed in Section 3.3.2 specialise in flowchart generation, the DSLs are not actually 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 these exact devices does not currently exist.

4.2 Psnodig

At its core, Psnodig is really just an intermediate representation through an AST. 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. Currently, it boasts one parser and four writers, as portrayed in Figure 4.1, depicting the parsers on the left and writers on the right.

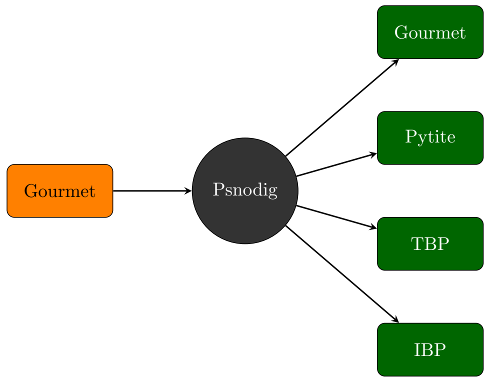

Figure 4.1: The parsers and writers of Psnodig.

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 internal, intermediate representation. Later, this representation is used to convert the original program further to a new target program. This means that a source program written in different languages can produce the same target program, given that their ASTs are identical.

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.

All of this leads to a feeling of predictability: We write our source code in an available language. Then we use the interpreter to test it, until we are satisfied with the results. Now we are able to transpile it to TBP and IBP through the command line, rather than having to reconstruct it all manually.

Currently, Psnodig offers multiple command line arguments. By running `psnodig help` in the command line, we are presented with Listing 4.1, which discloses our remaining options.

```
Psnodig - a general transpiler with options for pseudocode and
          flowcharts.

Available options:

<psnodig program>           : Parses the program and runs it
                              through the interpreter.

<psnodig ast program>       : Prints the program's AST to the
                              terminal.

<psnodig tbp program>       : Transpiles the program to TBP in
                              LaTeX.
<psnodig tbp pdf program>   : The same as above, but also
                              compiles the LaTeX file to a PDF.

<psnodig ibp program>       : Transpiles the program to IBP in
                              LaTeX.
<psnodig ibp pdf program>   : The same as above, but also
                              compiles the LaTeX file to a PDF.

<psnodig gourmet program>   : Transpiles the program to the
                              equivalent in Gourmet.
<psnodig Python program>    : Transpiles the program to the
                              equivalent in Python.

<psnodig help>              : Brings you back here!
```

Listing 4.1: The command line options for Psnodig, presented by running `psnodig help` in the terminal.

4.2.1 Syntax

Psnodig's syntax is located at Listing 5.1, in the form of Haskell data types.

The entry point of a Psnodig program is `Program`. Since all of its symbols are optional, 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 resembles the syntax of common programming languages. However, there are two statements that have been introduced specifically for Psnodig's real use case: `Hash statements` and `Annotation statements`.

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 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.

Figure 4.2 shows an AST of a hash statement. An example of a hash statement in action can be seen in Listing 4.12.

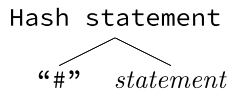

Figure 4.2: An AST of a hash statement.

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`.

We can also leave the first value of an Annotation statement to be an empty string. This means that the statement list will be interpreted, but nothing will show in the TBP or IBP versions. This also means that Annotation statements to some extend are supersets of Hash statements, offering the same functionality, but allowing more than statement at the time.

Likewise, the second value of an Annotation statement can be left empty. This allows us to explain things solely with natural language. Figure 4.3 shows an AST of an annotation statement. An example of an annotation statement in action can be seen in Listing 4.13.

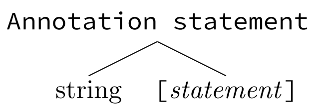

Figure 4.3: An AST of an annotation statement.

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.

4.2.2 Interpreter

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

As seen from the syntax in Listing 5.1, Psnodig programs consist of four parts: An optional program description, 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 is the interpreter's entry point, much like a `main` function in compiler languages like Go, Kotlin and Java.

Scope

Psnodig works with both a global and a local scope. All structs and functions are global, and can be accessed from within any function. This means that, among other things, 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 a runtime error, because `n` is not available in the scope of `g`. The latter program will run smoothly, as we pass the variable as an argument to `g`, which excitedly returns the variable straight back.

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

func g() {
    return n
}

f()
```

Listing 4.3: Gourmet program which provokes a runtime error.

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

func g(n int) {
    return n
}

f()
```

Listing 4.4: Gourmet program that will run uninterrupted.

Psnodig supports nested scopes as indicated by the `For String Expression Expression [Statement]` value within the `Statement` type in Listing 5.1. 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

Listing 5.1 shows that Psnodig prohibits two types of `For`-loops. `For String Expression [Statement]` is discused in the previous subsection, but we also entertain `ForEach String Expression [Statement]`. This is intended to mimic standard `For each`-loops found in languages like Go 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.

Built-in Functions

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, e.g. due to wrong number of arguments provided, or arguments having the wrong type, the program will stop and the user will receive an explanatory error message.

print(x_1, .., x_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` (presented in Listing 5.1). 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), which takes one number $x \in \mathbb{Q}$ as argument. The return value will be rounded up to the nearest $n \in \mathbb{N}$.

floor(x), which takes one number $x \in \mathbb{Q}$ as argument. The return value will be rounded down to the nearest $n \in \mathbb{N}$.

max(x_1**,** ..**,** x_n), which takes $n \in \mathbb{N}_{>0}$ arguments. The arguments themselves must be an $x \in \mathbb{Q}$. The return value will be the largest value amongst the arguments.

min(x_1**,** ..**,** x_n), which takes $n \in \mathbb{N}_{>0}$ arguments. The arguments themselves must be an $x \in \mathbb{Q}$. The return value will be the smallest value amongst the arguments.

append(x**,** xs), 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 1 upon success.

add(x**,** hs), which takes two arguments: an `Expression` x and a `HashSet` hs. It adds the former to the latter. The function returns 1 upon success.

add(k**,** v**,** hm) is an overloaded version of **add**, this time taking three arguments: An `Expression` k, an `Expression` v, and a `HashMap` hm. It maps k to v, and adds the mapping to hm. This version of the function also returns 1 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 returns the value v that k maps to in hm.

in(x**,** xs), which takes two arguments, an `Expression` x and either a `List`, a `HashSet`, or a `HashMap`. The function will check if x exists in xs, and return the appropriate boolean.

pop(x), which takes one argument, a non-empty `List` x. If the list is not empty, the function will modify x by removing v, before subsequently returning v.

toString(x), which takes one argument, an `Expression` x. It returns the string version of x. Base expressions and lists will look identical, whilst hashset, hashmap and struct expressions vary.

Hashsets are printed on the form $\texttt{(expr}_1\texttt{, .., expr}_n\texttt{)}$, hashmaps are printed on the form $\texttt{\{key}_1\texttt{:value}_1\texttt{, .., key}_n\texttt{:value}_n\texttt{\}}$, and structs are printed on the form $\texttt{field}_1\texttt{:value}_1\texttt{, field}_n\texttt{:value}_n$. Nested structs are enclosed in parentheses.

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 have 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 a new input target for Psnodig. This can motivate others to add their own language, or even write parsers for existing languages.

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 [63], whilst the largest university in northern Norway prefers C [64]. The largest university in Greece sticks to Java [5], and Harvard's renowned CS50 course introduces students to both JavaScript and SQL [24].

What the languages of computer science introductory courses do have in common, is that they tend to be within the imperative paradigm of computer programming. Therefore, we aimed to stay in the same sphere, 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

Before we reveal Gourmet's grammar, we believe it is valuable to walk thorugh important lexical aspects like keywords, comments and whitespace. These are handled by the lexer, which is presented in its entirety in Listing 5.5.

Identifiers and Keywords

An identifier in Gourmet is used to reference either a struct, function or variable. Identifiers are 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 cannot shadow already existing keywords.

In total, there are 14 keywords in Gourmet: `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 identifiers like `while` and `func`.

Gourmet allows us to define functions that shadow the built-in functions of Psnodig, however the interpreter will always prefer the built-in ones. While we cannot write and run a function called `print`, we can always define a function `print'` or `print1`, which will be handled just like any other function we define ourselves.

Comments

As there are no data types for comments in Psnodig, they must be handled individually by each lexer. Gourmet 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. A practical example of how they work can be seen in Listing 4.5.

```
func f() {
    // This is a single-line comment.

    /* This is a

       multi-line
       comment.
       */

    /* However, our lexer
    does not allow
        /* nested
        multi-line
    */  comments.
    */
}
```

Listing 4.5: Example of legal and illegal comments in a Gourmet program.

Whitespace

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

Listing 4.6, Listing 4.7 and Listing 4.8 show three identical programs, besides the use of whitespace. We have a function `f`, which defines a variable `a` and returns it. All three programs will be parsed successfully, and have the same internal representation in Psnodig.

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

Listing 4.6: A Gourmet program with a standard amount of whitespace.

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

Listing 4.7: A Gourmet program with a lot of whitespace.

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

Listing 4.8: A Gourmet program with no whitespace.

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 *let* us. 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.[1]

However, any value evaluated at runtime will inherently have one of five base types: `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 UTF-8 characters within a pair of double quotes. `Nil` indicates the absence of a value, and often serves as a placeholder.

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

[1]This way of employing type hints is inspired by other dynamically typed languages like Python and PHP.

Listing 4.9 shows how we can create a struct for modelling a Tree data structure, and Listing 4.10 shows how we can later initialise a variable to have a Tree value.

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

Listing 4.9: 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 4.10: A function to initialise three Tree structs and return the last one.

4.3.3 Evaluation Strategy

The Psnodig interpreter operates with a strict call-by-value evaluation strategy, just like the Pascal programming language.[2] 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.

4.3.4 Syntax

Grammar

We present Gourmet's EBNF grammar in Listing 4.11. EBNF, short for Extended Backus-Naur form, is a simple, precise and sufficiently powerful notation to formally describe the syntax of a programming language [15]. We could have opted for the original BNF notation, but EBNF allows us to present it more suffinctly.

We use a few meta symbols to help describe the grammar:

- $\Leftarrow$ indicates the application of a rule. For instance, `X` $\Leftarrow$ `Y` means that `X` shall be substituted with `Y`.
- `{ ... }` indicates the repetition of 0 or more times (much like the reflexive arrow in Figure 2.1).
- `[ ... ]` indicates binary presentness. The content inside the square brackets is either present or it is not.

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

- | indicates disjunction, like we saw in Section 2.2.1.
- `" ... "` indicates a terminal: a Gourmet keyword without production rules.

Non-terminals are symbols that can be substituted. All the symbols on the left side of the arrows are non-terminals, and they often appear on the right side too. We write them in uppercase. Additionally, there are seven terminals written in uppercase, appearing solely on the right side of the arrows. They are `NAME`, `STRING`, `TEXT`, `INTEGER`, `DECIMAL`, `BOOLEAN`, and `NIL`.

`NAME` works as a placeholder for an identifier, e.g. the name of a function. `TEXT` is any combination of UTF-8 characters. It is used to provide program descriptions, and to serve as the first value of an Annotation statement. `STRING` is similar to `TEXT`, but requires the character sequence to be wrapped in double quotes. `INTEGER` is any $n \in \mathbb{N}$. `DECIMAL` is any $q \in \mathbb{Q}$. `BOOLEAN` is a $b \in \{\top, \bot\}$. Lastly, `NIL` is meant to indicate the absence of a value.

Since the first argument of `STRUCTFIELD` is `EXPRESSION`, and the former also appears on the right hand side of the latter, we risk running into a *left-recursive cycle*. A cycle of this nature occurs when the parser attempts to match the same rule again and again, without actually consuming any input.

Therefore, we have an additional non-terminal `EXPRESSION1`, identical to `EXPRESSION` with the exception of not having a `STRUCTFIELD` non-terminal on its right hand side. This allows us to change the right hand side of `STRUCTFIELD` to `EXPRESSION1 "." EXPRESSION 2`.

```
PROGRAM            ⇐ [ PROGRAMDESCRIPTION ] { STRUCTDECL }
                     { FUNCTIONDECL } [ FUNCTIONCALL ]

PROGRAMDESCRIPTION ⇐ "?" TEXT "?" "!" TEXT "!"

STRUCTDECL         ⇐ "struct" NAME "{" { ARGUMENT } "}"

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

ARGUMENT           ⇐ NAME NAME

STATEMENT          ⇐ "#" STATEMENT | ANNOTATIONSTMT | "break"
                     | "continue" | ASSIGNMENT | WHILE
                     | FOREACH | FOR | IF
                     | "return" EXPRESSION | FUNCTIONCALL

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         ⇐ BINARYEXP | FUNCTIONCALL | LISTINDEX
                     | STRUCTFIELD | "not" EXPRESSION
                     | STRUCT | VALUE | NAME

EXPRESSION1        ⇐ BINARYEXP | FUNCTIONCALL | LISTINDEX
                     | "not" EXPRESSION | STRUCT | VALUE
                     | NAME

BINARYEXP          ⇐ OPERATOR EXPRESSION EXPRESSION

LISTINDEX          ⇐ NAME "[" EXPRESSION "]" { "["
                     EXPRESSION "]" }

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

STRUCT             ⇐ NAME "(" EXPLIST ")"

STRUCTFIELD        ⇐ EXPRESSION1 "." EXPRESSION

VALUE              ⇐ "nil"
                     | "map" "{" [ PAIR { "," PAIR } ] "}"
                     | "set" "{" EXPLIST "}" | BOOL | DECIMAL
                     | NUMBER | STRING | "[" EXPLIST "]"

BOOL               ⇐ "true" | "false"

PAIR               ⇐ EXPRESSION ":" EXPRESSION

EXPLIST            ⇐ [ EXPRESSION { "," EXPRESSION } ]
```

Listing 4.11: Gourmet's grammar in EBNF notation.

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. .
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 parantheses to make `(tree.value) + (tree’.value) + (tree”.value)`.

All operations are left-associative.

Compatibility with Psnodig

An important part of being compatible with Psnodig, is handling hash- and annotation statements. Since Gourmet is developed with Psnodig in mind, it was natural to fit them in. Listing 4.12 shows a program with a hash statement, we use it to assign the length of a list to a variable. Listing 4.13 shows a program with an annotation statement, where we use it to abstract swapping to variables in a list.

```
func f(a: list) {
    # n := length(a)
    for i := 0, n {
        print(a[i])
    }
}
```

Listing 4.12: A Gourmet program with a hash statement

```
func f(a: list) {
    if length(a) > 1 {
        @{swap a[0] and a[1]}{
            tmp := a[0]
            a[0] := a[1]
            a[1] := tmp
        }
    }
}
```

Listing 4.13: A Gourmet program with an annotation statement

4.3.5 Gourmet Writer

In addition to having an input language, we need an output language to show that Psnodig works as intended. The output language could be just about anything, but since we designed a parser for Gourmet, we figured it would be interesting to also have a writer.

By having both a parser and a writer for the same language, we are able to test Psnodig's consistency. Since the languages are the same, we should be able to have a lossless conversion back and forth. Psnodig does not take formatting into consideration, so we might experience a difference in whitespace. However, as discussed in Section 4.3.1.3, whitespace cannot alter a Gourmet program's semantics, so this is not a problem.

Additionally, since Gourmet boasts most of the characteristics of other dynamically typed languages, the writer serves as a pointer to how much effort it might take to write a parser for another dynamically typed programming language, like Python or Ruby.

4.4 Python Writer

As mentioned in Section 4.3, Gourmet's syntax is mainly inspired by Go and Python. As they are far richer languages, and Gourmet almost resembles a subset of Go, we decided to create an additional writer which converts ASTs to Python-like syntax.

The Gourmet and Python writers are very similar, only with a few notable differences, like `=` over `:=` for assignment, and colons over curly brackets. The biggest difference, however, is handling structs, as Python does not have them. Instead, they have *classes*. Listing 4.14 shows how we translate an empty struct `struct Struct{}` into a class.

```python
class Struct:
    def __init__(self):
        pass
```

Listing 4.14: The Python equivalent of an empty struct in Gourmet.

An important difference between Gourmet and Python, is that Python does not allow variable names to end with quotes (like the ones we introduced in Section 4.3.1.1). This means that, if we transpile a program containing variable names like `var'`, the Python interpreter will raise a syntax error during execution.

Additionally, while Gourmet allows all type hints, Python only allows a few (without importing external libraries). Therefore, if a type hint is not `int`, `float`, `double`, `str`, `boolean`, or `nil`, they are simply removed from the final Python version.

Most of Psnodig's built-in functions have equivalents in Python, but not all. For instance, to use `min` and `max`, we would have to import the `math` package. Other than those, we translate `length` to `len`, `get(key, hashmap)` to `hashmap[key]` etc., given that the correct number of arguments is provided.

Python does not inherently have hash- and annotation statements. However, while these are statements that are intended to be hidden during presentation, they are essential to a program's runtime. Therefore, we have decided to convert them to regular statements in Python, also ignoring the first argument of annotation statements.

Lastly, for descriptions of input and output, we have decided to put them at the start of the file as block comments. Listing 4.15 shows a program in Gourmet, and Listing 4.16 shows the transpiled version in Python.

```
? An integer x ?
! The integer x squared !

func f(x int) {
    return x * x
}
```

Listing 4.15: A Gourmet function where an input variable gets squared and returned.

```
# Input: An integer x
# Output: The integer x squared

def f(x: int):
    return x * x
```

Listing 4.16: The result of transpiling Listing 4.15 to Python with Psnodig

4.5 Pseudocode Writer

An important part of Psnodig is the ability to convert source code to equivalents on a different abstraction level. One of our writers transpile internal representations TBP in LaTeX.[3] As previously stated, we are not attempting to create a ground truth for pseudocode. Therefore, we have chosen to work with an already-established package for writing pseudocode like Algorithm2e.

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 anyway, in a way that adds particular value.

4.5.1 Compatibility with Psnodig

As already mentioned in Section 4.2.2.3, all built-in Psnodig functions are taken into consideration by the TBP writer. The function call `ceil(decimal)` is

[3]The writer can be found here: link

transpiled with mathematical symbols to $\lceil decimal \rceil$. The function call `append(x, xs)` is transpiled to the natural language description `append x to xs`.

Mathematical expressions are also taken into account. For instance, the 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`. They are always included, regardless of the rest of the program.

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 reflected 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".

4.5.2 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 the `pdf` flag we also get a PDF of the compiled LaTeX file.

Just like the Gourmet writer, the TBP writer is not able to take the original source code's formatting into account. This means that a program like the one shown in Listing 4.17 will be transpiled to the one shown in Listing 4.18 rather than the one shown in Listing 4.19.

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

Listing 4.17: A Gourmet function which declares a variable and returns it.

```
\proc{f()}{
    v ← 5 \;
    \Return v \;
}
```

Listing 4.18: How Psnodig transpiles the program from Listing 4.17.

```
\proc{f()}{v ← 5\;\Returnv;}
```

Listing 4.19: Alternative version of f in LaTeX with Algorithm2e.

Even though the compiled versions produce the same PDF, we believe the curated amount of whitespace increases the maintainability of the LaTeX document, in case we wish to tweak it manually after transpilation. All transpiled documents will include `\SetKwProg{proc}{Procedure}{}{}`, and the one transpiling Listing 4.17 will also have `\SetKwFunction{f}{f}`.

We also import the `linesnumbered` and `ruled` options from Algorithm2e, purely for aesthetic reasons. Figure 4.4 shows TBP *with* these options included, while Figure 4.5 shows the same program *without* the options.

```
Algorithm: A function that squares its output and returns it.
Input: An integer x.
Output: x squared.
Procedure Square (x)
| return x · x
```

Figure 4.4: An Algorithm2e program, with `ruled` and `linesnumbered`.

```
Input: An integer x.
Output: x squared.
Procedure Square (x)
| return x · x
  Algorithm: A function that squares its output and returns it.
```

Figure 4.5: An Algorithm2e program, without `ruled` and `linesnumbered`.

Lastly, all documents transpiled to TBP include `\DontPrintSemicolon` to prevent lines ending with semicolons, and `\renewcommand{\thealgocf}{}` to avoid numbering the compiled result. All TBP examples so far in this thesis shows the result of including the latter. Since we only transpile the topmost function in a file, we feel like excluding the number looks cleaner. However, if we were to transpile multiple functions, it might make sense to remove it.

4.6 Flowchart Writer

Our third and last writer converts internal representations to LaTeX programs using the TikZ package. Compiling these files produce flowcharts. We believe there is value to working with such a different abstraction level (compared to Gourmet and TBP), and pretty flowcharts helps showcase Psnodig's powerfulness.

Just like the TBP writer, the IBP writer only works on topmost function in a file, ignoring the program description, structs and function call. Our reasoning is the same as earlier.

4.6.1 Compatibility with Psnodig

To make the corresponding flowcharts clear as well as pleasing, we have opted for three different node themes. This will make it easier to differentiate significant contexts.

The start node and all return nodes have the same theme of rectangles with white text on a dark background. Statements containing a list of statements have the same theme of ellipses with black text on a yellow background. Lastly, all remaining statements have the theme of rectangles with black text on a light purple background. Figure 4.6 shows the result of transpiling a simple function, showcasing all the aforementioned node styles.

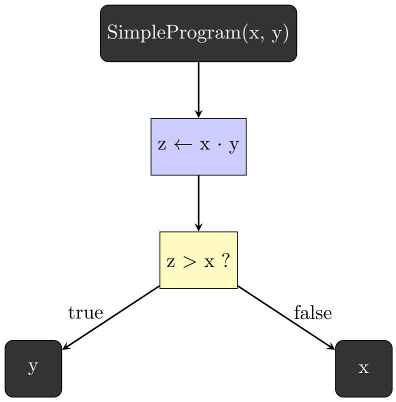

Figure 4.6: A program showcasing all the different node types the Flowchart generator produces.

Just like the TBP writer, the IBP writer uses mathematical notation where appropriate. For instance, `x && y` is transpiled to `x ∧ y`. Contrary to the TBP writer however, Psnodig's built-in functions are not considered. That means that code like `length(collection)` will look identical both in the source code and the corresponding flowchart.

All statements are displayed in a simple rectangular box, but `ForEach`-, `For`-, `Loop`-, and `If`-statements are, as previously mentioned, given their own node themes. This is because they break with the standard linear pattern, and contain statements of their own.

`ForEach` `identifier collection statements` starts with a light yellow rectangle asking if we have iterated `collection`. From here there are two arrows. The first is labeled `true` and points to the subsequent statement following the current **`ForEach`** (if there are any).

The other one is labeled `false` and points to a statement stating `identifier` ← `next element in collection`. Then, each statement in `statements` is presented. After the last of these, there is an arrow pointing back to the light yellow rectangle. A working example is presented in Figure 4.7.

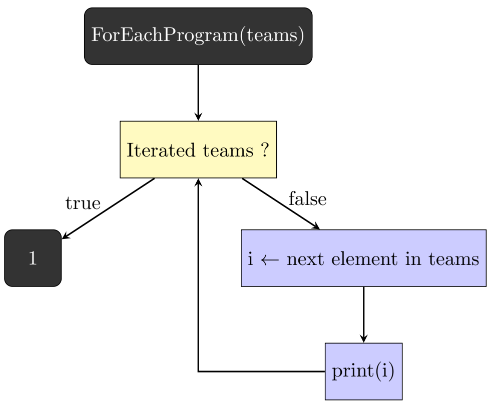

Figure 4.7: A `ForEach`-statement, iterating a variable `teams`.

`While` `condition statements` largely follows the pattern of **`ForEach`**. The statement starts with a light yellow rectangle displaying `condition` followed by a question mark.

There are again two edges: The first is labeled `true` and points to the subsequent statement following the current **`While`** (if there are any). The other is labeled `false` and points to the first statement in `statements`. The last of these points back to the light yellow rectangle. A working example is presented in Figure 4.8.

`For` `identifier from to statements` starts with assigning `identifier` to `fromRange`. The next node is a light yellow rectangle. If `from` and `to` are both numericals, the rectangles text will be either `from < to ?` or `from > to ?`, depending on their values. If they are not both numericals, the default is `from < to ?` since we do not evaluate expressions, and have no way of knowing their actual values.

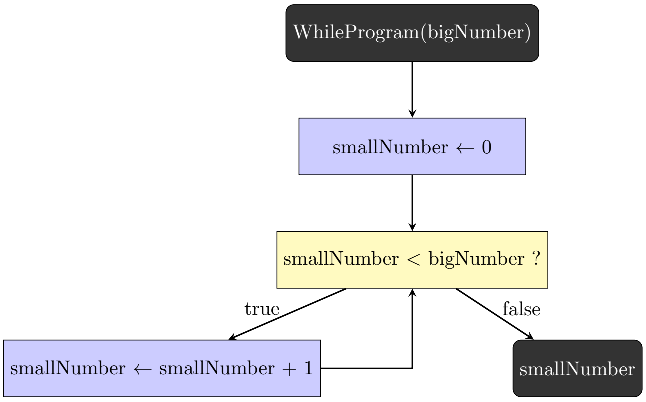

Figure 4.8: A `While`-statement, iterating until `i` reaches the value of `bigNumber`.

From this light yellow rectangle, we have an edge labeled `true` which points to the subsequent statement following the current **`For`** (if there are any). In the opposite way, an edge labeled `false` points to the first statement in `statements`. After the last of these, there is an assignment statement, either incrementing or decrementing `identifier`, before pointing back to the light yellow rectangle.

Figure 4.9, Figure 4.10, and Figure 4.11 shows the three different ways a **`For`**-statement can be transpiled to a flowchart.

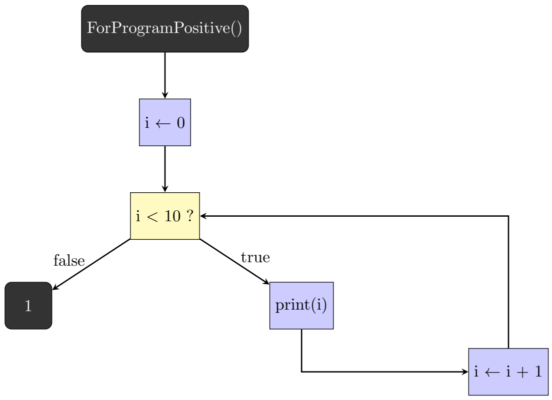

Figure 4.9: A `For`-statmenet where `from < to`.

`If` `condition statements else` is the most intricate statement of the four discussed in this section. Again we start with a yellow rectangle displaying `condition` followed by a question mark. An edge labeled `true` points to the first statement in `statements`. Another edge labeled `false` points to either `else`, the subsequent statement following the current **`If`**, or nothing, since both cases are actually optional.

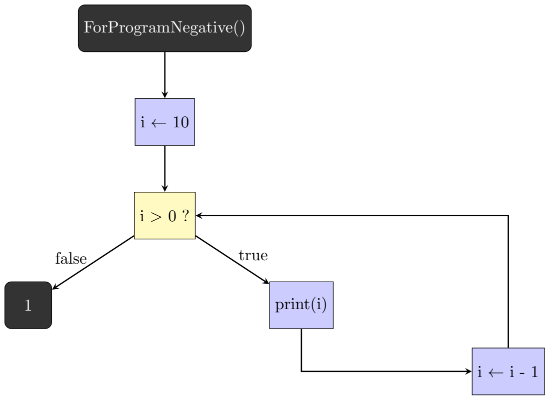

Figure 4.10: A `For`-statmenet where `from > to`.

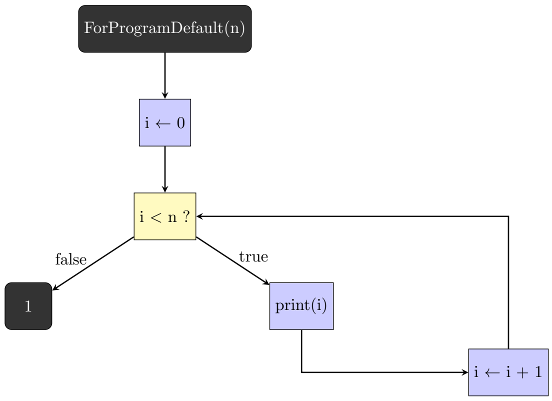

Figure 4.11: A `For`-statmenet where `from` and `to` are not both numericals.

The first example of this section, Figure 4.6, actually shows an If-statement.

If `statements` does not contain a `Return` statement, its last statement must point to the subsequent statement following the current **`If`** (if there is one). If `else` is present, the same applies to its list of statements.[4] Figure 4.12 depicts a program flow where the **`If`** has two `else` clauses. Since neither the statement list in the original if or the first clause contain a return statement, their final statements point to the subsequent statement following the current If. However, the last `else` clause does contain a return, and therefore does not point any further.

If the last statement in `statements` is a `Return`-statement, and the same is the case for all subsequent `else`s, the next statement after the current **`If`** is unreachable. If our code still has this structure, we are told in the command line, and the transpilation is interrupted.

[4] As we can see in both Listing 4.11 and Listing 5.1, the Else type carries its own list of statements.

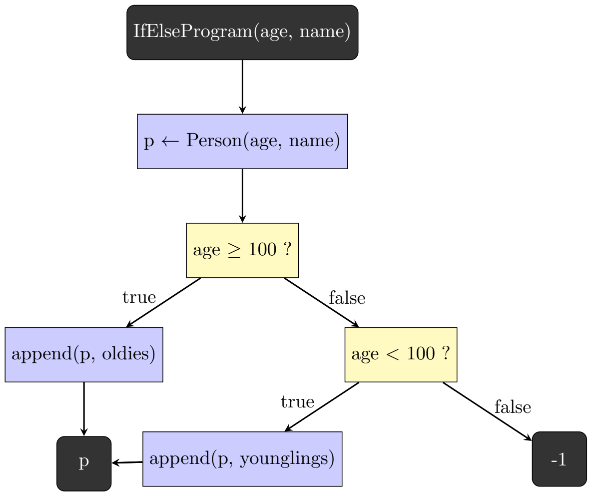

Figure 4.12: An `If`-statement with two clauses.

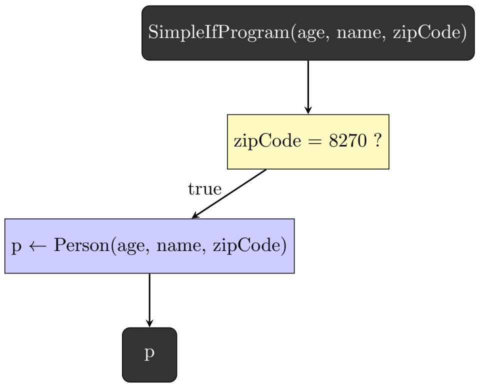

Figure 4.13: An `If`-statement without any else-clauses.

4.6.2 Output

Transpiling IBP with 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 4.20 shows the (most important parts of the) LaTeX file we receive by transpiling the program in Listing 4.17. We see how every node has a unique id, reference a tikzstyle, and how the metadata describes their placement. We also make a point out of drawing all the edges *after* drawing all the nodes, for the sake of readability. To take this one step further, we also sort the edges on the id of the source node.

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

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

Listing 4.20: Excerpt of the LaTeX from transpiling Listing 4.17 to IBP.

Chapter 5

Implementation

This chapter will present concrete implementations of **Psnodig** and its parts. As previously mentioned, Psnodig is really just an intermediate representation of a computer program. The tool is materialised through a parser and multiple writers, all written in Haskell.

First, we introduce Psnodig's data types in Haskell, before moving on to Monads, which essentially is the heart of all our implementations. Then, we delve deeper into how the interpreter, parser and each of our writers work. Lastly, we explain how we tested each part of the Psnodig tool.

5.1 Psnodig

Listing 5.1 shows the data types which define Psnodig's syntax.[1] The entry point of a Psnodig program is **Program**, which contains an optional **Program Description**, a list of **Struct** declarations, a list of **Function** declarations, and an optional **Function Call**. The latter is used as an entry point for execution. The program description is made up of two **String** values, tailored to produce descriptions for TBP.

The **Argument** type is just like the program description type, made up of two string values. It is used to provide type hints to developers, and is required for both struct- and function declarations. However, the second argument is not considered during execution, and is discarded when transpiling to TBP and IBP. Therefore, we are free to declare arguments like `x int, y string` if we wish to.

The **Function Call** can be both an expression and a statement. Even though if is not made explicit here, all functions that are executed have to return something. Therefore, doing `x := f()` will never lead to undefined behaviour, because the interpreter will make sure that `f` returns an appropriate value.

[1] The file containing these data types can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/src/Syntax.hs`.

```haskell
data Program = Program (Maybe ProgramDescription) [StructDecl]
                       [FunctionDecl] (Maybe FunctionCall)

data ProgramDescription = ProgramDescription String String

data StructDecl = StructDecl String [Argument]

data Struct = Struct String [Expression]

data StructField = StructField Expression Expression

data FunctionDecl = FunctionDecl 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 =
```

```haskell
        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 5.1: Psnodig's data types in Haskell.

5.2 Monads

First introduced in a programming context by Philip Wadler back in the 90's, monads allow us to structure computations within programs. 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 [65, 29].

Haskell is a purely *functional* programming language. In this paradigm, functions' output values are solely determined by their input values [27]. Operations that occur outside the scope of a function, affecting or depending on an external state, are referred to as *side effects*. 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. To create our parser we employed **Parsec**, which refers to itself as a *monad transformer* [35].

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

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

5.3 Interpreter

5.3.1 Monad

The interpreter is built on a **StateT ExecutionState (ExceptT RuntimeError IO)** monad. The main point is `StateT`, whose signature is `StateT s m a`.[2] As we 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 to our monad is `ExecutionState`, fleshed out in Listing 5.2. It is a compound type with named fields, and closely resembles a struct in programming languages like Go and C.[3] Using `StateT` allows us to maintain and operate on this state throughout computations.

```
type StructDecls = Map.Map String [String]
type FuncDecls = Map.Map String FunctionDecl
type Scope = [[(String, Value)]]
type ScopeStack = [Scope]

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

Listing 5.2: The Psnodig interpreter's state.

The state 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"]`. As previously mentioned, the second value of arguments types is simply discarded.

We also have `funcDecls`, which is a declaration of functions. It is a mapping from function names to that function's values. If we call a function in our code, the interpreter looks it up, binds the arguments to temporary values, and runs the statements in the function body. The use of `Map` was primarily chosen due to

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

[3]And Gourmet, for that matter.

its O(log n) lookups, which beats the O(n) lookups of its `List` counterpart [13, 12].

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.

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 its own scope.

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 a data type that encapsulates the various errors we can encounter at runtime, accompanied with a fitting error message. The entire list is presented in Listing 5.3.

```
data RuntimeError =
      VariableNotFound String
    | OutOfBounds String
    | FunctionNotFound String
    | StructNotFound String
    | ArithmeticError String
    | ListNotFound String
    | BadArgument String
    | InvalidStructField String
    | NoImplementationError String
    | WrongNumberOfArguments String
    | NoReturnError String
    | MissingEntryPoint String
    | RuntimeErrorWithOutput [String] RuntimeError
```

Listing 5.3: All error messages we can encounter in Psnodig.

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. Instead, we join the current print calls to `RuntimeErrorWithOutput`, and display them to the terminal before the error message. If we do not encounter any errors, we only output the results of the print calls.

5.3.2 Constraints

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.

The syntax also includes the statement value `ForEach String Expression [Statement]`. However, the interpreter only allows expressions that can be iterated: `Text`, `List`, `HashSet`, and `HashMap`.

As previously mentioned, every function invoked during execution must return a value. This is because function calls are also expressions, and the interpreter evaluated functions before applying them.

Take the function shown in Listing 4.12, which does not return anything. Assigning its result to a variable or using it in a conditional statement within a loop is illogical. However, the program presented in Listing 5.4 is syntactically correct, despite being semantically flawed.

```
func g() {
    x := f()
    if x == f() {
        return f()
    }
}
```

Listing 5.4: A syntactically correct Gourmet program, including three function call expressions.

5.4 Gourmet 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 [35]. 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 its entirety in Listing 5.5, taking up no more than 20 lines. It specifies the syntax for identifiers and comments, and establishes reserved terms for operators and keywords.

```
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 5.5: The Gourmet lexer.

Listing 5.6 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 accumulated 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
    programDescription <- optionMaybe parseProgramDescription
    structs <- many parseStructDecl
    funcs <- many parseFunctionDecl
    functioncall <- optionMaybe parseFunctionCall
    return $ Program programDescription structs funcs
                     functioncall
```

Listing 5.6: Parsing Gourmet programs.

We have opted for an applicative programming style when developing the parser, which means that we use applicative functors. The four we use are <$>, <*>, *>, and <*. An example where all are applied can be seen in Listing 5.7, where we parse function calls. Remember, the Psnodig FunctionCall data type is `FunctionCall String [Expression]`.

<$> allows us to populate the type on the left side with the values on the right side. <* means that we parse both the left side and the right side, but only keep the left side. *> means the opposite. <*> means that we parse and subsequently keep both.

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

Listing 5.7: Parsing Gourmet function calls.

5.5 Gourmet Writer

The Gourmet writer lets us transpile programs written in Gourmet back to themselves. It is created by the same principles as the Gourmet parser: where our parser converts `1 + 1` to `BinaryExp Plus (Constant 1) (Constant 1)`, our writer converts `BinaryExp Plus (Constant 1) (Constant 1)` to `1 + 1`.

However, since Psnodig does not store information about formatting, the AST has no way of letting us know what combinations of whitespace existed in the original program. Therefore, the programs we write and the transpiled results we receive by Psnodig might vary when it comes to indentation and newlines.

The writer adds a tab when entering new scopes in loops and if-statements, and the statements inside a function are indented with a tab by default. 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.

The writer is built on a `Writer String` monad. We traverse the AST and use the `tell` function (a key function within the Writer monad) to convert AST nodes to the Gourmet equivalents in string form. Essentially, the function takes a string value and adds it to the monad's maintained log. As such, the final result is just a concatenated string.

5.6 Python Writer

The Python writer works exactly like the Gourmet writer, but converts ASTs to Python rather than Gourmet, naturally.

5.7 Pseudocode Writer

The monad of choice for the pseudocode writer is similar to the one we used for the Gourmet- and Python writers. However, before converting the intermediate representation to pseudocode, another program skims the AST and extracts function names and relevant keywords. This information is stored in a tuple of sets, and the TBP writer uses the information to better take advantage of Algorithm2e formatting.

The pseudocode writer is built on a `ReaderT Environment (Writer String)` monad. 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 tuple that contains the information collected from the AST.

We have two separate functions for adding function names and keywords as macros, presented in Listing 5.8 and Listing 5.9, respectively. Calling `asks fst` gives us the set containing function declarations, and calling `asks snd` gives us the set containing keywords. The sets are then mapped over to create macros that utilise Algorithm2e's formatting.

The syntax `(\x -> ...)` is a way to write anonymous functions on the spot in Haskell. In this case, the function takes one argument: `x`. The code on the right hand side of the arrow is the function body. `++` is used to concatenate strings.

```
writeFunctionDecls :: Pseudocode ()
writeFunctionDecls = do
    funcs <- asks fst
    mapM_ (\x -> tell ("\\SetKwFunction{" ++ x ++
                       "}{" ++ x ++ "}\n"))
          funcs
```

Listing 5.8: Creating LaTeX macros for function declarations.

```
writeKeywords :: Pseudocode ()
writeKeywords = do
    keywords <- asks snd
    mapM_ (\x -> tell ("\\SetKw{Kw" ++ fstToUpper x ++
                       "}{" ++ x ++ "}\n"))
          keywords
```

Listing 5.9: Creating LaTeX macros for keywords.

5.8 Flowchart Writer

The flowchart writer is built on a `StateT Environment (Writer String)` monad. It is almost a combination of our interpreter and TBP writer: we maintain and operate on the state `Environment`, and the final result is a concatenated string. Listing 5.10 presents the state in question.

```
data Environment = Environment
    { edges :: [(Int, String)]
    , lastId :: Int
    , coreIds :: [Int]
    , activeBranches :: [String]
    }
```

Listing 5.10: The IBP writer's state.

Since we update node ids every time we enter a new statement, it can be challenging to keep track of which nodes should correspond to which statements. Because we wish to write out all the edges *after* writing out all the nodes, we have decided to save them in a list of integer-string pairs we call `edges`.

The integer is the id of the node with the outgoing edge. This is useful when we finally write the edges, because we can sort them, and make the output a little bit more tidy.

All nodes must have a unique id, to make them referenceable in the future. To know which ids we have already generated, we maintain an integer `lastId`. With each new node, the value is incremented by one.

Also, when working with the four control flow statements `Loop`, `ForEach`, `For`, and `If`, it is essential to track the original node. Figure 4.7 shows how the last inner statement of a `Loop` points back to the original condition, and `if exmpl` shows how we continue the program flow from the false-branch after writing the inner statements of the true-branch.

Since each statement will force `lastId` to be updated, we keep a stack of integers named `coreIds`. This allows us to preserve the reference to each starting node, even when we process nested control flow statements.

Lastly, we use `activeBranches` to deal with branches of if statements that do not end with a return statement. As mentioned in section 4.6.1, the last statement of an active branch must have an edge to the next statement outside of that branch. After processed the entire if-statement, we use these ids to create edges from them to the next generated id, before we put them in the `edges` list discussed earlier.

5.9 Testing

To feel more assured that the implementation correctly leads the design, we have adopted unit tests for all the tools that make up Psnodig. This includes the parser, all four writers, as well as the interpreter working on the internal representation of Psnodig.

Unit tests do well at isolating parts of our code into so-called **units**, and comparing them with expected results [53]. We have opted for HUnit, a unit testing framework specifically designed for Haskell. HUnit lets us easily create, name and execute tests, with the framework checking the results automatically [25].

By testing Psnodig, we are given a better insight into which parts work. We have kept the tests similar for each part, to be more certain that it is consistent. The tests makes the tool more robust as a whole. When adding a new parser or writer to Psnodig, we can add similar tests to make sure that they are compatible.

5.9.1 Gourmet parser

Since we only have one parser, it is important to know that it works as intended. Our goal is to see that we can successfully parse programs from Gourmet source code to a Psnodig AST. We have carried out tests from small values, like numbers and boolen values, to fully fledged programs.[4]

To make sure the smallest building blocks were functioning properly, we tested all of Psnodig's **values**. This includes small and big numbers, decimals and booleans. We have also tested the compound values lists, hashsets and hashmaps, both empty and populated.

We have tested all types of **declarations**, that is program declaration (describing a function's input and output - tailored for the pseudocode writer), struct declarations and function declarations.

Psnodig's statements are essential to programs, so we tested all of them. This includes all kinds of assignment, while- and for-loops, function calls, and also Psnodig's famous hash- and annotation-statements.

We have also tested all of Psnodig's **expressions**, which are essential to statements. This includes function calls, accessing on list indexes, constants, as well as negated expressions, and more.

And lastly, we have tested entire Psnodig programs. Both completely empty programs, but also ones containing all types of declarations and a triggering function call.

Passing all these tests make us trust our implementation more. We know that our parser is able to parse both small parts, but also entire programs into a well-defined Psnodig AST.

5.9.2 Writers

Now that we are certain that we can convert Gourmet programs to Psnodig ASTs, we are also interested to know that we can convert them further. We have created four test suites, one for each of our writers. They all look very similar, and purposely test the same AST tokens.

Another useful aspect of having unit tests for the writers, is to show how they coincide with the built-in functions of Psnodig.

Gourmet

An interesting aspect of the transpiler is to see if we can convert Gourmet programs "back to themselves". We tested the exact same attributes for the writer as we did for the parser, but reversed in the sense that we went from an AST to a program.

As previously mentioned, Psnodig does not take whitespace into consideration, so a program like `if x { return y}` is transpiled to `if x {\n\t return`

[4] The Gourmet parser test suite can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/test/parsers/Gourmet.hs`.

`y\n}`. Other than that, all values, declarations, expressions, statements and programs were losslessly transpiled.[5]

Python

Python is on a similar abstraction level to Gourmet, but still has a few differences. For instance, the conversion from structs to classes, and how we deal with hash- and annotation statements. Through unit tests we were able to make sure that the results are as intended.

We tested the same parts as we did with the Gourmet parser.[6]

Pseudocode

We have tested the conversion to TBP. When looking at smaller ASTs, the difference in TBP and Gourmet/Python is not that big. However, we lean heavily on the Algorithm2e when it comes to statements, and the test show that we utilise the package well. The tests also help us see that we balance the dollar signs (for math mode) well.

We have tested the same ASTs as before, except for the ones containing struct declarations, as they are not converted to TBP anyway.[7]

Flowcharts

Not done yet

5.9.3 Interpreter

The interpreter is a big part of Psnodig, thus it was important to make sure that it is able to interpret individual parts of a Psnodig AST. We have 10 test suites in total, ranging from positive and negative statement tests, to custom functions tailored specifically to Psnodig.

Binary expressions are tested thoroughly. This includes arithmetic operations with both single-type operands and multi-type operands. We have also tested all the relational operands, as well as the logical operands. This also includes negative tests, to make sure that we avoid things like argument mismatch, and that division by zero is handled by our interpreter rather than attempted executed.

Since binary expressions take expressions as arguments, we have also tested that all constant expressions are successfully interpreted and evaluated to their corresponding values, to make sure that the binary expression tests are indeed

[5] The Gourmet writer test suite can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/test/writers/Gourmet.hs`.

[6] The Python writer test suite can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/test/writers/Python.hs`.

[7] The Pseudocode writer test suite can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/test/writers/Pseudocode.hs`.

receiving expected input.

For the most part, the interpreter does not return anything when evaluating individual statements. Therefore, we have tested that providing legal arguments to statements types like assignment does not yield any errors. We have also tested compound statements, like For, ForEach, While, and If. Also here do we include negative tests, like trying to use undefined variables and structs.

Two integral parts of Psnodig that make its grammar stand out from other grammars, are the Hash- and Annotation statements. We have tested that they work as intended: Hash statements are interpreted and evaluated like any other statements. The first argument of an Annotation statement is ignored, whilst the statements in its second argument are interpreted and evaluated.

An integral part of the interpreter itself, are the built-in functions introduced in Section 4.2.2. They have all been tested to make sure they return values as expected.

Lastly, we have tested entire programs, showing the interplay between their different parts. This includes working with structs, variables, as well as making sure that various built-in functions actually perform expected side effects.[8]

[8] The Pseudocode writer test suite can be accessed here: `https://github.com/sergiosja/Master/blob/master/psnodig/test/InterpreterTest.hs`.

Chapter 6

Evaluation

In this chapter, we will evaluate the Psnodig tool against the goals we set in Section 1. We will look at the general effort to add a new parser, correctness and performance of our interpreter, and whether or not our writers are able to produce satisfactory output.

All the Psnodig examples in this chapter will be created entirely through the command line interface, to ensure that the examples are reproducable. To keep a common thread going through this chapter, as well as keeping it fair to all parts of the tool, there are four algorithms that we will use for evaluation: `Naive search`, `Bubble sort`, `Deletion in a Binary search tree` and `Depth-first search`.

Naive search

Naive search is a simple algorithm where we look for a particular element in a list. It is naive because it iterates the list in a simple start-to-end fashion, without applying any heuristics, contrary to the **Binary search** algorithm mentioned in Section 2.1.1.

Bubble sort

Bubble sort is one of the most straightforward sorting algorithms we have available. It is used to sort a list, making sure that for all elements, the element on the left is smaller, and the element on the right is bigger. Bubble sort is seldom applied to solve real problems, but is a popular algorithm for teaching students about sorting algorithms as a concept.

If the input list only contains one element or is empty, we return it as it is. Otherwise, we compare the elements on index 0 and 1, and if the first is larger than the second, we swap them. Then we continue to compare the elements on index 1 and 2, and so on. We do this **n** times, where **n** equals the length of the list minus one. In each passing, the unsorted largest element in the list is placed in its proper position towards the end of the list.

Deletion in a Binary search tree

Binary search trees are a common data structure in computer science. Each node has at most two child nodes, and all nodes are descendents of a single root node. Additionally, for all nodes **nd**, every node to the left of **nd** must have a value smaller than or equal to **nd**, and every node to the right must have a value greater than or equal to **nd**.

Deletion in a binary search tree involves finding a node with a specific value, and removing it whilst preserving the search tree's properties. There are three cases to consider: deletion of a node with no children, deletion of a node with one child, and deletion of a node with two children.

If the node has no children, we can simply remove it. If the node has one child, that child takes the place of its parent. If the node has two children, we substitute it with the right subtree's smallest-value node or the left subtree's largest-value node, before removing the original node.

Depth-first search

Depth-first search, also referred to as DFS, is an algorithm for traversing a graph data structure. We begin at an arbitrary node of the graph, and "visit" neighbouring nodes, following each path as far as possible.

It is commonly used to identify all nodes within a connected component. If the graph we are working with is also unweighted, we can use DFS to find its shortest path.

6.1 Extensible

The first goal of Psnodig is extensibility: we should be able to add parsers that are able to convert source code to an internal representation of Psnodig. We have successfully written a parser for an imperative, C-like programming language that we coined **Gourmet**.

6.1.1 Gourmet parser

What we aim to evaluate here, is whether or not the parser is compatible with Psnodig. The Psnodig grammar is a limitation of how rich a program can look before it is converted by a writer. This means that a parser can at most parse all data types, but at the very least a 'Program', as it is the entry point of all Psnodig programs.

A minimal example is an empty program. The Gourmet parser is capable of parsing empty files, producing an AST of a `Program Nothing [] [] Nothing` value. We cannot produce a maximal example, because `Program` contains a list of `FunctionDecl` values, which again contain a list of `Statement` values. These values can be compound, and in effect go on forever.[1]

[1]We could interpret a maximal example to be a program limited by our computer's memory, but theoretically Psnodig programs can be infinitely big.

To evaluate if we have successfully extended Psnodig with a parser, we will implement the four algorithms. We will then run them on a function that parses the programs, and produces either a Psnodig AST or a parse error.

Listing 6.1 shows an implementation of Naive search in Gourmet, and Listing 6.2 shows the Psnodig AST generated from parsing the program.

`gjør eksemplene pene! bytt ut bst greia, dessverre. men finn noe annet bst-relatert?`

```
? A list A and an integer x?
! True if x $\in$ A and false if not!

func NaiveSearch(A list, x int) {
    # n := length(A)
    for i := A {
        if i == x {
            return true
        }
    }
    return false
}
```

Listing 6.1: Naive search implementation in Gourmet.

```
Program
  (Just (ProgramDescription
    "A list A and an integer x"
    "True if x $\in$ A and false if not"))
  []
  [ FunctionDecl
      "NaiveSearch"
      [ Argument "A" "list", Argument "x" "int" ]
      [ HashStmt
          (Assignment
              (VariableTarget "n")
              (ExpressionValue
                (CallExp
                    (FunctionCall "length" [VariableExp "A"])))
          ),
        ForEach
          "i"
          (VariableExp "A")
          [ If
              (BinaryExp Equal
                (VariableExp "i")
                (VariableExp "x"))
              [Return (Constant (Boolean True))]
              Nothing
          ],
        Return (Constant (Boolean False))
```

```
        ]
    ]
    Nothing
```

Listing 6.2: The Psnodig AST generated by parsing Listing 6.1.

Listing 6.3 shows an implementation of Bubble sort in Gourmet, and Listing 6.4 shows the Psnodig AST generated from parsing the program.

```
? A list A?
! The list A, but ordered from smallest value to largest!

func BubbleSort(A list) {
    # n := length(A)
    for i := 0, n - 2 {
        for j := 0, n - i - 2 {
            if A[j] > A[j+1] {
                @{swap A[j] with A[j+1]}{
                    tmp := A[j]
                    A[j] := A[j+1]
                    A[j+1] := tmp
                }
            }
        }
    }
    return A
}
```

Listing 6.3: Bubble sort implementation in Gourmet.

```
Program
  ( Just
      ( ProgramDescription
          "A list A"
          "The list A, but ordered from smallest value to
    largest"
      )
  )
  []
  [ FunctionDecl
      "BubbleSort"
      [Argument "A" "list"]
      [ HashStmt
          ( Assignment
              (VariableTarget "n")
              ( ExpressionValue
                  ( CallExp
                      (FunctionCall "length" [VariableExp "A"])
                  )
              )
          ),
        For
          "i"
```

```
        (Constant (Number 0))
        ( BinaryExp
            Minus
            (VariableExp "n")
            (Constant (Number 1))
        )
        [ For
            "j"
            (Constant (Number 0))
            ( BinaryExp
                Minus
                ( BinaryExp
                    Minus
                    (VariableExp "n")
                    (VariableExp "i")
                )
                (Constant (Number 1))
            )
            [ If
                ( BinaryExp
                    GreaterThan
                    (ListIndex "A" [VariableExp "j"])
                    ( ListIndex
                        "A"
                        [ BinaryExp
                            Plus
                            (VariableExp "j")
                            (Constant (Number 1))
                        ]
                    )
                )
                [ AnnotationStmt
                    "swap A[j] with A[j+1]"
                    [ Assignment
                        (VariableTarget "tmp")
                        ( ExpressionValue
                            (ListIndex "A" [VariableExp "j"])
                        ),
                      Assignment
                        (ListIndexTarget "A" [VariableExp "j"
  ])
                        ( ExpressionValue
                            ( ListIndex
                                "A"
                                [ BinaryExp
                                    Plus
                                    (VariableExp "j")
                                    (Constant (Number 1))
                                ]
                            )
                        ),
                      Assignment
                        ( ListIndexTarget
                            "A"
```

```
                                [ BinaryExp
                                    Plus
                                    (VariableExp "j")
                                    (Constant (Number 1))
                                ]
                            )
                            (ExpressionValue (VariableExp "tmp"))
                        ]
                    ]
                    Nothing
                ]
            ],
          Return (VariableExp "A")
        ]
    ]
    Nothing
```

Listing 6.4: The Psnodig AST generated by parsing Listing 6.3.

Listing 6.5 shows an implementation of Deletion in a binary tree in Gourmet, and Listing 6.6 shows the Psnodig AST generated from parsing the program.

```
? The root node in a tree and the value of a node to be deleted?
! The updated tree with one less node!

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

func Delete(node Tree, val int) {
    if node == nil {
        return nil
    }
    else if (node.value) < val {
        node.right := Delete(node.right, val)
        return node
    }
    else if (node.value) > val {
        node.left := Delete(node.left, val)
        return node
    }

    if (node.left) == nil {
        return node.right
    }
    else if (node.right) == nil {
        return node.left
    }

    node' := FindMax(node.left)
    node.value := node'.value
    node.left := Delete(node.left, node.value)
```

```
    return node
}

func FindMax(root Tree) {
    if (root.right) == nil {
        return (root.value)
    }

    return FindMax(root.right)
}
```

Listing 6.5: Deletion in a binary search tree implementation in Gourmet.

```
Program (Just (ProgramDescription "The root node in a tree and
    the value of a node to be deleted" "The updated tree with
    one less node")) [StructDecl "Tree" [Argument "value" "int",
    Argument "left" "Tree",Argument "right" "Tree"]] [
    FunctionDecl "Delete" [Argument "node" "Tree",Argument "val"
     "int"] [If (BinaryExp Equal (VariableExp "node") (Constant
    Nil)) [Return (Constant Nil)] (Just (ElseIf (BinaryExp
    LessThan (StructFieldExp (StructField (VariableExp "node") (
    VariableExp "value"))) (VariableExp "val")) [Assignment (
    StructFieldTarget (StructField (VariableExp "node") (
    VariableExp "right"))) (ExpressionValue (CallExp (
    FunctionCall "Delete" [StructFieldExp (StructField (
    VariableExp "node") (VariableExp "right")),VariableExp "val"
    ]))),Return (VariableExp "node")] (Just (ElseIf (BinaryExp
    GreaterThan (StructFieldExp (StructField (VariableExp "node"
    ) (VariableExp "value"))) (VariableExp "val")) [Assignment (
    StructFieldTarget (StructField (VariableExp "node") (
    VariableExp "left"))) (ExpressionValue (CallExp (
    FunctionCall "Delete" [StructFieldExp (StructField (
    VariableExp "node") (VariableExp "left")),VariableExp "val"
    ]))),Return (VariableExp "node")] Nothing)))),If (BinaryExp
    Equal (StructFieldExp (StructField (VariableExp "node") (
    VariableExp "left"))) (Constant Nil)) [Return (
    StructFieldExp (StructField (VariableExp "node") (
    VariableExp "right")))] (Just (ElseIf (BinaryExp Equal (
    StructFieldExp (StructField (VariableExp "node") (
    VariableExp "right"))) (Constant Nil)) [Return (
    StructFieldExp (StructField (VariableExp "node") (
    VariableExp "left")))] Nothing)),Assignment (VariableTarget
    "node'") (ExpressionValue (CallExp (FunctionCall "FindMax" [
    StructFieldExp (StructField (VariableExp "node") (
    VariableExp "left"))]))),Assignment (StructFieldTarget (
    StructField (VariableExp "node") (VariableExp "value"))) (
    ExpressionValue (StructFieldExp (StructField (VariableExp "
    node'") (VariableExp "value")))),Assignment (
    StructFieldTarget (StructField (VariableExp "node") (
    VariableExp "left"))) (ExpressionValue (CallExp (
    FunctionCall "Delete" [StructFieldExp (StructField (
    VariableExp "node") (VariableExp "left")),StructFieldExp (
    StructField (VariableExp "node") (VariableExp "value"))]))),
    Return (VariableExp "node")],FunctionDecl "FindMax" [
```

```
    Argument "root" "Tree"] [If (BinaryExp Equal (StructFieldExp
     (StructField (VariableExp "root") (VariableExp "right"))) (
    Constant Nil)) [Return (StructFieldExp (StructField (
    VariableExp "root") (VariableExp "value")))] Nothing,Return
    (CallExp (FunctionCall "FindMax" [StructFieldExp (
    StructField (VariableExp "root") (VariableExp "right"))]))]]
     Nothing
```

Listing 6.6: The Psnodig AST generated by parsing Listing 6.5.

Listing 6.7 shows an implementation of Naive search in Gourmet, and Listing 6.8 shows the Psnodig AST generated from parsing the program.

```
? A list of booleans, a stack of nodes, a graph and a starting
    node?
! The list of booleans!

func dfs(visited list, stack list, graph list, node int) {
    add(node, visited)
    append(node, stack)

    while length(stack) > 0 {
        m := pop(stack)

        for nb := get(m, graph) {
            if not in(nb, visited) {
                add(nb, visited)
                append(nb, stack)
            }
        }
    }
    return visited
}
```

Listing 6.7: DFS implementation in Gourmet.

```
Program
    (Just (ProgramDescription "A list of booleans, a stack of
    nodes, a graph and a starting node" "The list of booleans"))
    []
    [FunctionDecl "dfs" [Argument "visited" "list", Argument "
    stack" "list", Argument "graph" "list", Argument "node" "int
    "] [CallStmt (FunctionCall "add" [VariableExp "node",
    VariableExp "visited"]), CallStmt (FunctionCall "append" [
    VariableExp "node", VariableExp "stack"]), Loop (BinaryExp
    GreaterThan (CallExp (FunctionCall "length" [VariableExp "
    stack"])) (Constant (Number 0))) [Assignment (VariableTarget
     "m") (ExpressionValue (CallExp (FunctionCall "pop" [
    VariableExp "stack"]))), ForEach "nb" (CallExp (FunctionCall
     "get" [VariableExp "m", VariableExp "graph"])) [If (Not (
    CallExp (FunctionCall "in" [VariableExp "nb", VariableExp "
    visited"]))) [CallStmt (FunctionCall "add" [VariableExp "nb"
    , VariableExp "visited"]), CallStmt (FunctionCall "append" [
```

```
    VariableExp "nb", VariableExp "stack"])] Nothing]], Return (
    VariableExp "visited")]]
    Nothing
```

Listing 6.8: The Psnodig AST generated by parsing Listing 6.7.

6.2 Executable

The second goal from Section 1 was that our tool should be executable. Therefore, Psnodig comes with an interpreter working directly on the internal representation. When a parser has successfully converted a source program to Psnodig, we want to be able to run that code. This way, no matter what langauge we use to write our program, it is the converted version that is ultimately executed.

Naturally, we would like all programs to be executable. However, the interpreter currently stands without an implementation for the statement types **Break** and **Continue**. If a program flow runs into either of them, the program will terminate with a warning. Except for these, all Psnodig data types are handled by the interpreter.

Now we can try to execute the programs we mentioned earlier. We use the Gourmet programs from earlier to give us the internal representation, rather than constructing them ourselves through variables, because we believed them to be satisfactory.

6.3 Presentable

se på goals igjen. gourmet og python er faktisk deler av exteisble, ikke executable. presentable handler utelukkende om tbp og ibp. python writer brukes kinda som benchmark for å se hvor mye effort det er å lage python parser (kan sammenligne gourmet parser og writer her)

6.3.1 Gourmet writer

The last part of our evaluation is the set of writers we currently have available. We will evaluate all four individually, and we start with the Gourmet writer. This is the one where flaws are easiest to locate, as the output programs should be more or less 1-to-1 with the source programs.

Listing 6.9, Listing 6.10, Listing 6.11, and Listing 6.12 show the results of transpiling Gourmet back to Gourmet. The only real difference is the amount of whitespace in some areas, and by executing the transpiled results we get the same results as before.

```
? A list A and an integer x ?
! True if x ∈ A and false if not !

func NaiveSearch(A list, x int) {
  # n := length(A)
  for i := A {
    if i == x {
      return true
    }
  }
  return false
}
```

Listing 6.9: The result of transpiling Listing 6.1 back to Gourmet.

```
? A list A ?
! The list A, but ordered from smallest value to largest !

func BubbleSort(A list) {
  # n := length(A)
  for i := 0, n - 1 {
    for j := 0, n - i - 1 {
      if A[j] > A[j + 1] {
        @{swap A[j] with A[j+1]}{
          tmp := A[j]
          A[j] := A[j + 1]
          A[j + 1] := tmp
        }
      }
    }
  }
  return A
}
```

Listing 6.10: The result of transpiling Listing 6.3 back to Gourmet.

```
? The root node in a tree and the value of a node to be deleted
    ?
! The updated tree with one less node !

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

func Delete(node Tree, val int) {
  if node == nil {
    return nil
  } else if node.value < val {
    node.right := Delete(node.right, val)
    return node
  } else if node.value > val {
```

```
    node.left := Delete(node.left, val)
    return node
  }
  if node.left == nil {
    return node.right
  } else if node.right == nil {
    return node.left
  }
  node' := FindMax(node.left)
  node.value := node'.value
  node.left := Delete(node.left, node.value)
  return node
}

func FindMax(root Tree) {
  if root.right == nil {
    return root.value
  }
  return FindMax(root.right)
}
```

Listing 6.11: The result of transpiling Listing 6.5 back to Gourmet.

```
? A list of booleans, a stack of nodes, a graph and a starting
    node?
! The list of booleans!

func dfs(visited list, stack list, graph list, node int) {
    visited[node] := true
    append(node, stack)

    while length(stack) > 0 {
        m := pop(stack)
        print(m)

        for nb := get(m, graph) {
            if visited[nb] == false {
                visited[nb] := true
                append(nb, stack)
            }
        }
    }
    return visited
}
```

Listing 6.12: The result of transpiling Listing 6.7 back to Gourmet.

6.3.2 Python writer

The next writer we evaluate is the Python writer. The output will be relatively similar to Gourmet, since the latter also employs indentation. However, Listing 6.13, Listing 6.14, Listing 6.15, and Listing 6.16 show the results of transpiling Gourmet to Python. There are syntactic differences, like using colons over curly braces, and writing hash- and annotation statements like regular statements. We also see structs being replaced by classes, and Psnodig functions being replaced by Python ones.

```
# Input: A list A and an integer x
# Output: True if x ∈ A and false if not

def NaiveSearch(A, x: int):
    n = len(A)
    for i in A:
        if i == x:
            return True


    return False
```

Listing 6.13: The result of transpiling Listing 6.1 to Python.

```
# Input: A list A
# Output: The list A, but ordered from smallest value to
    largest

def BubbleSort(A):
    n = len(A)
    for i in range(0, n - 1):
        for j in range(0, n - i - 1):
            if A[j] > A[j + 1]:
                tmp = A[j]
                A[j] = A[j + 1]
                A[j + 1] = tmp




    return A
```

Listing 6.14: The result of transpiling Listing 6.3 to Python.

```
# Input: The root node in a tree and the value of a node to be
    deleted
# Output: The updated tree with one less node

class Tree:
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right


def Delete(node, val: int):
    if node == None:
```

```
        return None
    elif node.value < val:
        node.right = Delete(node.right, val)
        return node
    elif node.value > val:
        node.left = Delete(node.left, val)
        return node

    if node.left == None:
        return node.right
    elif node.right == None:
        return node.left

    node’ = FindMax(node.left)
    node.value = node’.value
    node.left = Delete(node.left, node.value)
    return node


def FindMax(root):
    if root.right == None:
        return root.value

    return FindMax(root.right)
```

Listing 6.15: The result of transpiling Listing 6.5 to Python.

```
# Input: A list of booleans, a stack of nodes, a graph and a
    starting node
# Output: The list of booleans

def dfs(visited, stack, graph, node: int):
    visited[node] = True
    stack.append(node)
    while len(stack) > 0:
        m = stack.pop()
        print(m)
        for nb in graph[m]:
            if visited[nb] == False:
                visited[nb] = True
                stack.append(nb)



    return visited
```

Listing 6.16: The result of transpiling Listing 6.7 to Python.

As an extra test to see if we really are able to transpile programs to syntactically correct Python equivalents, we will run the final transpiled code with the Python interpreter, and compare the results with those of the Psnodig interpreter.

men hvordan presentere det??

6.3.3 Pseudocode writer

The pseudocode writer converts the intermediate representation to a much different abstraction level than we saw in the two previous writers. Now, the code is no longer executable.

This writer gives us a LaTeX file, accompanied by the compiled result in a PDF if we wish. In this section, however, we will ignore the LaTeX file, and solely view the compiled versions.

Figure 6.1, Figure 6.2, Figure 6.3, and Figure 6.4 show the results of transpiling Gourmet to TBP.

```
Algorithm: NaiveSearch
  Input: A list A and an integer x
  Output: True if x ∈ A and false if not
Procedure NaiveSearch(A, x)
    for i ∈ A do
        if i = x then
            return true
    end
    return false
```

Figure 6.1: The result of transpiling Listing 6.1 to pseudocode.

```
Algorithm: BubbleSort
  Input: A list A
  Output: The list A, but ordered from smallest value to largest
Procedure BubbleSort(A)
    for i ← 0 to n − 1 do
        for j ← 0 to n − i − 1 do
            if A[j] > A[j + 1] then
                swap A[j] with A[j+1]
        end
    end
    return A
```

Figure 6.2: The result of transpiling Listing 6.3 to pseudocode.

6.3.4 Flowchart writer

The flowchart writer is the most advanced of the four, working on yet another level of abstraction. While the pseudocode writer produces TBP, which at

```
Algorithm: Delete
  Input: The root node in a tree and the value of a node to be deleted
  Output: The updated tree with one less node
Procedure Delete(node, val)
    if node = Nil then
        return Nil
    else if node_value < val then
        node_right ← Delete (node_right, val)
        return node
    else if node_value > val then
        node_left ← Delete (node_left, val)
        return node
    if node_left = Nil then
        return node_right
    else if node_right = Nil then
        return node_left
    node' ← FindMax (node_left)
    node_value ← node'_value
    node_left ← Delete (node_left, node_value)
    return node
```

Figure 6.3: The result of transpiling Listing 6.5 to pseudocode.

```
Algorithm: dfs
  Input: A list of booleans, a stack of nodes, a graph and a starting node
  Output: The list of booleans
Procedure dfs(visited, stack, graph, node)
    visited[node] ← true
    append node to stack
    while |stack| > 0 do
        m ← pop (stack)
        for nb ∈ get(m, graph) do
            if visited[nb] = false then
                visited[nb] ← true
                append nb to stack
        end
    end
    return visited
```

Figure 6.4: The result of transpiling Listing 6.7 to pseudocode.

least resembles executable code, the IBP we produce here does not. Whilst expressions are fitted into boxes, control flow statemenets like loops and ifs risk colliding when nested.

Figure 6.5, Figure 6.6, Figure 6.7, and Figure 6.8 show the results of transpiling Gourmet to flowcharts.

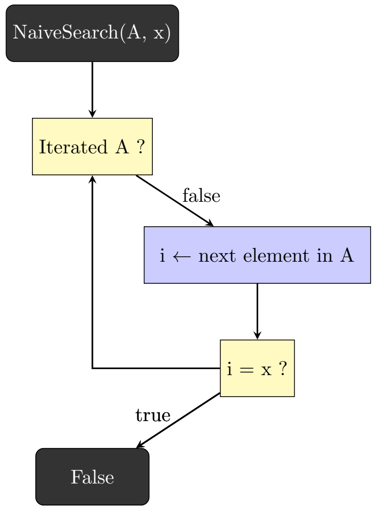

Figure 6.5: The result of transpiling Listing 6.1 to a flowchart.

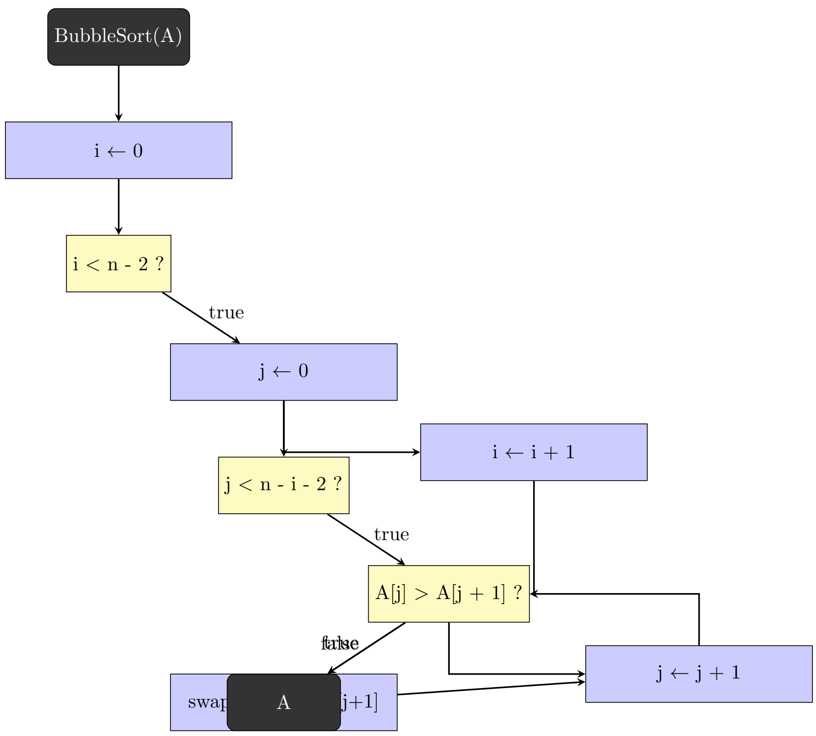

Figure 6.6: The result of transpiling Listing 6.3 to a flowchart.

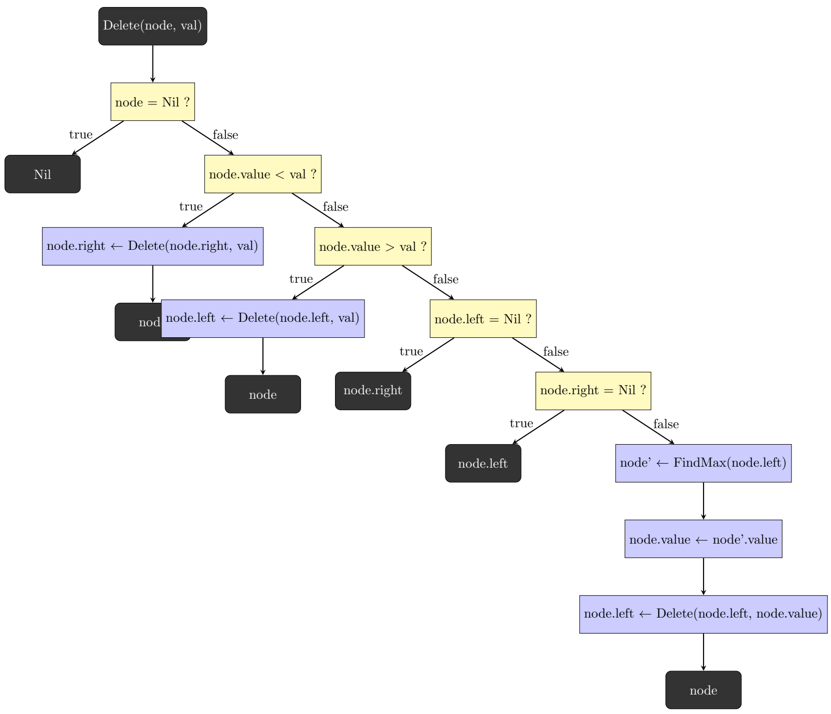

Figure 6.7: The result of transpiling Listing 6.5 to a flowchart.

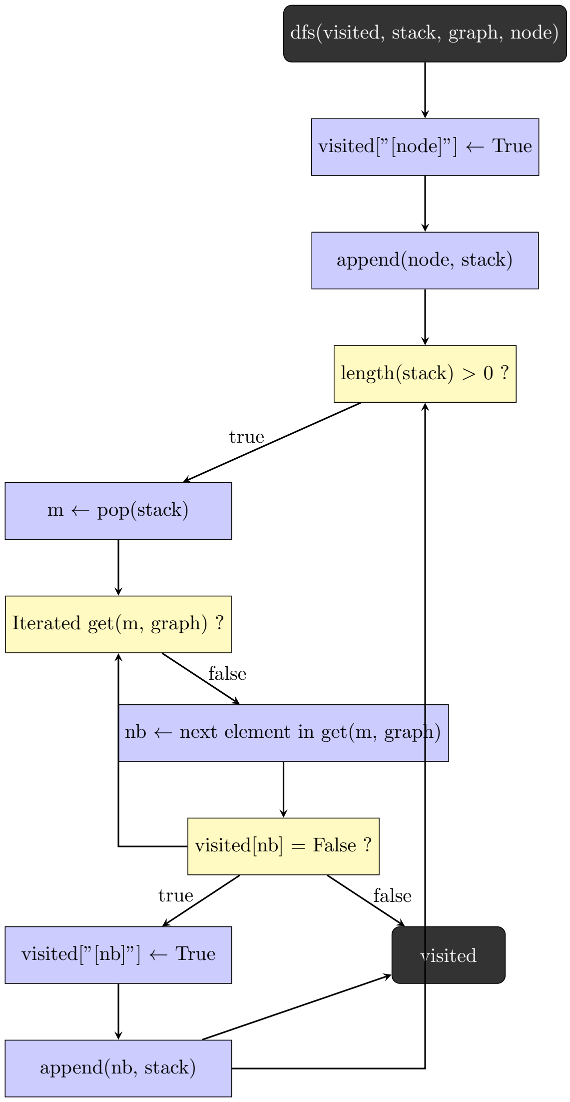

Figure 6.8: The result of transpiling Listing 6.7 to a flowchart.

Chapter 7

Discussion

In this chapter, we discuss the results we obtained in Section 6.

7.1 Reflections

7.1.1 Notation

The TBP- and IBP writers work on a different abstraction level than those of Gourmet and Python. Where the two last ones are restricted to ASCII-based keywords like `!=` and `floor(x)`, the two first ones allow us to use more precise mathematical notation like $\neq$ and $\lfloor \mathtt{x} \rfloor$.

7.1.2 Extensible

We currently only have one parser, for the Gourmet programming language. However, we do believe that similar parsers could be added, without much extra work. The parser is currently 336 lines, and we have also preferred readability to conciseness, so we believe it could be done in even fewer.

Oppdater dette tallet 14. mai!

An additional point is that the parser clearly works, as discussed in Section 5.8.1, but it is also able to produce the computer programs we wanted to evaluate with ease. Given the general nature of Psnodig, we believe one could follow more or less the same steps that we have, to add a new parser.

Adding writers

nei, ikke snakk om dette. men kanskje prøv å trekk en linje mellom str på gourmet parser og writer, og python writer, og si at det muligens kan være en liknende mengde jobb å legge til en parser (og parsere for liknende språk).

husk, gourmet- og python writer er egt ikke målet med thesisen. det er FRA soruce code TIL pseudokode og flytdiagram.

Currently, Psnodig boasts four writers. Two of them are for programming languages, whilst the other two are presentation-only targets.

The Gourmet writer works, as we are able to losslessly transpile Gourmet code back to itself. We saw it in Section 6.1, and we were also able to run the transpiled versions and obtain the same results as we did with the original ones. The Gourmet writer spans 224 lines of code, showing the general effort of adding a writer.

The Python writer also works, as we are able to run the transpiled versions on the Python interpreter, and receive the same output as we did with Gourmet. The Python writer spans exactly 300 lines of code, showing the general effort of adding a writer for a subset of an existing language.

The two presentation-only writers both target LaTeX. Both are able to convert a Psnodig AST to LaTeX, despite the formats being somewhat different. Particularly the IBP writer works on a very different abstraction level, primarily consisting of nodes and edges. However, in the end it worked well, and the writers span 323 and 538 lines of code, respectively.

7.1.3 Executable

It is clearly executable.

7.1.4 Does it hold up against the problem as a whole?

We believe that Psnodig *does* hold up against the problem definition introduced in Section 3.1, and that it successfully achieves the goals we set in Section 1.2. It is the first of its kind, a publically available tool that transpiles source code to TBP and IBP, as well as being executable, and is easy to add parsers and writers to.

7.2 Limitations

7.2.1 Presentable

We are able to successfully transpile all Psnodig data types to flowcharts on their own, and also construct more complex flowcharts, combining statements, expressions and more.

However, a limitation is that of nested control flow statements, being `While`, `For`, `ForEach`, and `If`. The biggest problem is that each edge has a fixed size, and when nesting these types of statements, we risk them crashing into each other. This can be seen in Figure 6.7, where one of the statements shadow parts of another. We also see, in Figure 6.8, that edges sometime interfere with nodes, making it a bit difficult to know where they actually point.

7.2.2 Extensible

Psnodig is primarily built to work with imperative languages, like Go and Python. This made it comfortable to write a parser for Gourmet, as well as the writers for Gourmet and Python. However, if the languages come from different paradigms, it might make it more challenging.

We believe it might be more challenging to add a parser for logic languages like Prolog, formal specification languages like TLA+, query languages like SQL etc. This is because they work in a different, and more specific way than general, imperative languages do.

It might also prove challenging to add a writer for these languages, especially if we are very dependent on structs or loops.
`Implementation language`

To add a parser or writer to Psnodig in the first place, they must be written in Haskell.[1]

7.2.3 Executable

Doesn't capture infinite loops!

[1]Technically, they can be written in a different language and *transpiled* to Haskell.

Chapter 8

Conclusion

`Important: Remind the reader of all the good stuff!`

This section will give a summary of our contributions and discuss future work, before concluding the thesis.

8.1 Future Work

make it more customisable. ref

```
    -- hadde vært kult om man kunne sende med egne sånne? feks gjennom en fil, også
-- istedenfor å være bundet til startstop, io osv. å legge inn en egen \tikzstyle{s
```

fra flowchart writeren

8.1.1 Psnodig

The syntax of Psnodig contains all the necessary building blocks to write complex computer programs, but as previously mentioned, the syntax shows the maximum available. Parsers and writers can easily utilise as little of the syntax as they want, for instance if we wish to create a simple calculator language.

Therefore, it could be interesting to expand Psnodig's syntax to include more things we commonly see today in programming, like lambda functions and more data structures. This would make the language even more flexible, and potentially make it more tempting for people to use.
`add more parsers!!!`

8.1.2 Interpreter

The interpreter has its primary focus on correctness, and is not particularly optimised for speed. There are parts of the interpreter that could be optimised, utilising Haskell's features to a greater extent. For one, variable scopes are lists of `(String, Value)`-pairs, but the innermost layer could have been a mapping, which would shrink the ammortised runtime of lookups from $O(n)$ to $O(1)$.

Another flaw is that the interpreter will only stop on error or success. For instance, We do not have a way of identifying infinite loops by print statements like we can in many other language. This is because the print statements appear on the screen only when the program has terminated.

8.1.3 Writers

The writers all work in their own right, but there are always improvements to be made. This section covers what we believe to be the ones to make.

Python Writer

```
Currently, Python is a subset of Python, but we are not sure to
what extent we cover the standard Python language.  There are
things we do not touch upon, like list comprehension, though there
are patterns in the Psnodig syntax that could be recognised and
converted accordingly.

no we dont!!  we dont gaf.  psnodig is first and foremost made
4 tbp and ibp :)
```

Pseudocode Writer

We have utilised the Algorithm2e package, but there might be a different package we could use.

Flowchart Writer

As discussed in Section 7.2.1, the IBP writer struggles with complex flowcharts where we nest the statements types `While`, `For`, `ForEach`, and `If`. This could be improved by making the edges longer in these cases, or choosing different placement rather than always building flowcharts downwards.

8.2 Summary of Contributions

We introduced an imperative C-like programming language Gourmet.

We designed a DSL Psnodig, and created an interpreter that runs on its internal representation.

We have added four writers to Psnodig, two of them being executable programming languages, and two of them being presentation-only targets.

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] R. Antonsen. *Logical Methods - The Art of Thinking Abstractly and Mathematically*. Springer, 2021.

[5] Athens University of Economics and Business. *Introduction to Computer Science*. Course webpage. 2023. URL: `http://www.dept.aueb.gr/en/dmst/content/introduction-computer-science` (visited on 04/18/2024).

[6] 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.

[7] 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.

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

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

[11] 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.

[12] *Data.List*. Maintained by libraries@haskell.org. n.d.

[13] *Data.Map*. Maintained by libraries@haskell.org. n.d.

[14] F. Domini. *An overview of Pandoc*. 2014. URL: `https://www.tug.org/TUGboat/tb35-1/tb109dominici.pdf` (visited on 04/18/2024).

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

[16] C. Fiorio. *algorithm2e - Floating algorithm environment with algorithmic keywords*. Version 5.2. 2017.

[17] 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.

[18] A. Gershun and M. R. Wulff. *AlaSQL*. GitHub repository. 2023.

[19] 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.

[20] S. Gold and A. Rangarajan. "A graduated assignment algorithm for graph matching". In: *IEEE Transactions on pattern analysis and machine intelligence* 18.4 (1996), pp. 377–388.

[21] H. H. Goldstine and J. Von Neumann. "Planning and coding of problems for an electronic computing instrument". In: (1947).

[22] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications*. en. Ed. by B. Golub. Wiley, Oct. 2014.

[23] N. Hall. *derw*. GitHub repository. 2023.

[24] Harvard University. *CS50: Introduction to Computer Science*. Course webpage. 2023. URL: `www.pll.harvard.edu/course/cs50-introduction-computer-science` (visited on 04/18/2024).

[25] D. Herington. *HUnit: A unit testing framework for Haskell*. Version 1.6.2.0. Maintained by Simon Hengel. 2021.

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

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

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

[29] L. H. Karlsen. "A Study In Monads". MA thesis. 2013.

[30] K. Karpierz and S. A. Wolfman. "Misconceptions and concept inventory questions for binary search trees and hash tables". In: *Proceedings of the 45th ACM technical symposium on Computer science education*. 2014, pp. 109–114.

[31] R. H. Klenke. *VHDL Processes*. Slide 8. June 1999.

[32] H. Kopka and P. W. Daly. *Guide to LATEX*. Pearson Education, 2003.

[33] D. A. Kosower, J. J. Lopez-Villarejo, and S. A. Roubtsov. "Flowgen: Flowchart-Based Documentation Framework for C++". In: *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 2014, pp. 59–64.

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

[35] D. Leijen, P. Martini, and A. Latter. *parsec: Monadic parser combinators*. Version 3.1.17.0. Maintained by Oleg Grenrus and Herbert Valerio Riedel. 2023.

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

[37] LINFO. *Algorithms: A Very Brief Introduction*. 2007. URL: `http://www.linfo.org/algorithm.html` (visited on 04/17/2024).

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

[39] 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.

[40] A. Lopez. "Statistical machine translation". In: *ACM Computing Surveys (CSUR)* 40.3 (2008), pp. 1–49.

[41] 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.

[42] 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.

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

[44] C. Martins. *Awesome Linters*. 2023.

[45] 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.

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

[47] 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.

[48] N. Nishimura. *Pseudocode*. University of Waterloo. URL: https://student.cs.uwaterloo.ca/~cs231/resources/pseudocode.pdf (visited on 04/17/2024).

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] M. Olan. "Unit testing: test early, test often". In: *Journal of Computing Sciences in Colleges* 19.2 (2003), pp. 319–328.

[54] S. Overflow. *Developer Survey 2023 - Most Popular Technologies: Language*. 2023.

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

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

[57] O. Seppälä, L. Malmi, and A. Korhonen. "Observations on student misconceptions—A case study of the Build–Heap Algorithm". In: *Computer Science Education* 16.3 (2006), pp. 241–255.

[58] R. L. Shackelford and R. LeBlanc. "Introducing computer science fundamentals before programming". In: *Proceedings frontiers in education 1997 27th annual conference. teaching and learning in an era of change*. Vol. 1. IEEE. 1997, pp. 285–289.

[59] F. Stahlberg. "Neural Machine Translation: A Review and Survey". In: *arXiv preprint arXiv:1912.02047* (2019).

[60] 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.

[61] T. Tantau. *TikZ & PGF Manual for version VERSION*. Version 3.1.10. 2023.

[62] *The correspondence between Donald E. Knuth and Peter van Emde Boas on priority deques during the spring of 1977*. Science Park, Universiteit van Amsterdam. URL: https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf (visited on 04/17/2024).

[63] University of Oslo. *IN1000 - Introduction to Object-oriented Programming*. Course webpage. 2023. URL: https://www.uio.no/studier/emner/matnat/ifi/IN1000/index-eng.html (visited on 04/18/2024).

[64] University of Tromsø. *INF-1100 - Introduction to programming with scientific applications*. Course webpage. 2023. URL: https://en.uit.no/utdanning/emner/emne/806701/inf-1100 (visited on 04/18/2024).

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

[70] F. Zhang et al. "Flowchart-based cross-language source code similarity detection". In: *Scientific Programming* 2020 (2020), pp. 1–15.