Nottingham Trent University

School of Science and Technology

Domain Specific Language For Balancing Binary Search Trees

by

Joseph Woodward

in

2018

Project report in part fulfilment

of the requirements for the degree of

Bachelor of Science

in

Software Engineering

I hereby declare that I am the sole author of this report. I authorize the Nottingham Trent University to lend this report to other institutions or individuals for the purpose of scholarly research.

I also authorize the Nottingham Trent University to reproduce this report by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature

Joseph L. Woodward

Abstract

A Binary Search Tree (BST) is an effective data storage technique used within programming languages, allowing one to store, organize and retrieve data in a—potentially—performant way.

There is, however, one bottleneck that may nullify the performance of BSTs, and that is the inefficient, non-hierarchical organization of the data stored within the BST itself: if data is stored linearly, then retrieving data from the BST will be linear, providing no positive usage over many other inherently linear structures. To combat this, one may use a BST Balancing Algorithm, which is used to ensure that all data is stored in a non-linear, hierarchical structure, thus increasing the efficiency of the retrieval of said data.

This work introduces the general topic of BSTs, including how data is stored and retrieved within them, how balancing algorithms can be used to maximize the efficiency of the structure itself. The primary bulk of this work, however, will be introducing a new Domain Specific Language (DSL), a DSL that will be used to allow one to easily experiment with one’s own design and implement of balancing algorithms, providing the capability to benchmark said algorithms with performant and common algorithms used within both the industry and within academia today.

Acknowledgements

Firstly, I would like to thank my supervisor Dr Neil Sculthorpe, who has supported me greatly with both personal and academic queries and issues. His critique and guidance of my work and interest within the domain of programming languages—academic and beyond—has vastly increased both the quality of my work and my interest in programming language theory.

A special thanks to my love, Katarzyna, who not only took on the task of completing my extra-curricular activities, allowing me to solely focus my attention on this project—at least, when not working—but whos love and kindness infuses within me the ability to stay calm, to stay peaceful, to stay thoughtful.

To my Mother, Mandy, whose sheer determination and ability to endure has motivated me more than She will ever know. I love you, Mum.

Lastly, to the Earth and all of Her inhabitants, whose interconnectivity has and continues to allow me to be.

Table of Contents

[Introduction 1](#__RefHeading___Toc1485_386191692)

[1.1 General Problem Overview 1](#__RefHeading___Toc1923_1488950273)

[1.2 Topic Introduction 2](#__RefHeading___Toc1925_1488950273)

[1.1.1 What is a Binary Search Tree? 2](#__RefHeading___Toc1927_1488950273)

[1.1.2 What is the Difference Between Balanced and Unbalanced BSTs? 3](#__RefHeading___Toc1929_1488950273)

[1.3 Project Introduction 3](#__RefHeading___Toc1931_1488950273)

[CONTEXT 6](#__RefHeading___Toc1487_386191692)

[1.1 Introduction 6](#__RefHeading___Toc501_2103244707)

[1.1.1 What Exactly is a Domain Specific Language? 6](#__RefHeading___Toc1138_2103244707)

[1.1.2 What Are The Benefits Of A DSL? 7](#__RefHeading___Toc1140_2103244707)

[1.2 Case Study of DSLs 8](#__RefHeading___Toc1142_2103244707)

[1.2.1 LaTeX 8](#__RefHeading___Toc1144_2103244707)

[1.2.2 Cascading Stylesheets (CSS) 11](#__RefHeading___Toc1148_2103244707)

[New Ideas 13](#__RefHeading___Toc1489_386191692)

[1.4 Introduction 13](#__RefHeading___Toc1035_1941171729)

[1.5 Aims and Objectives 13](#__RefHeading___Toc1146_2103244707)

[1.2.3 Overview 13](#__RefHeading___Toc1903_2103244707)

[1.5.1 How will the DSL be implemented? 14](#__RefHeading___Toc1905_2103244707)

[1.3 Project Scope, Milestones, Main Tasks & Deliverables 14](#__RefHeading___Toc1907_2103244707)

[1.3.1 Overview 14](#__RefHeading___Toc1909_2103244707)

[1.3.2 In-scope 15](#__RefHeading___Toc1911_2103244707)

[1.3.3 Out-of-scope 16](#__RefHeading___Toc1913_2103244707)

[1.4 Project Risks 16](#__RefHeading___Toc1915_2103244707)

[1.4.1 Overiew 16](#__RefHeading___Toc1917_2103244707)

[1.4.2 Task Overflow 17](#__RefHeading___Toc1919_2103244707)

[1.4.3 Implementation Language Knowledge 17](#__RefHeading___Toc1037_1941171729)

[1.4.4 Data Destruction 18](#__RefHeading___Toc1921_2103244707)

[1.4.5 Project Member Propensities 18](#__RefHeading___Toc1923_2103244707)

[1.5 Gantt Chart 18](#__RefHeading___Toc1925_2103244707)

[IMPLEMENTATION 20](#__RefHeading___Toc1491_386191692)

[22Software Development Life-Cycle 20](#__RefHeading___Toc2568_1488950273)

[22Language Design 21](#__RefHeading___Toc1039_1941171729)

[1.6 Syntax 21](#__RefHeading___Toc1041_1941171729)

[33Language Features 23](#__RefHeading___Toc1043_1941171729)

[1.1 Overview 23](#__RefHeading___Toc1045_1941171729)

[1.2 Feature Implementation 25](#__RefHeading___Toc1047_1941171729)

[1.2.1 Console Output 25](#__RefHeading___Toc1049_1941171729)

[1.2.2 Functions 26](#__RefHeading___Toc1051_1941171729)

[1.2.3 Variables 28](#__RefHeading___Toc1053_1941171729)

[1.2.4 Loops 30](#__RefHeading___Toc1374_1941171729)

[1.2.5 If / Else 31](#__RefHeading___Toc1376_1941171729)

[1.2.6 Arithmetic Expressions 33](#__RefHeading___Toc1378_1941171729)

[1.2.7 String Manipulation 34](#__RefHeading___Toc1380_1941171729)

[RESULTS / DISCUSSION 37](#__RefHeading___Toc1493_386191692)

[33Introduction 37](#__RefHeading___Toc2570_1488950273)

[CONCLUSIONS / FUTURE WORK 38](#__RefHeading___Toc1495_386191692)

Insert > Reference > Index and Tables … > Table of Contents

List of FigureS

List of Tables

dwqddwqdwqdwq

CHAPTER 1

Introduction

## General Problem Overview

A Data Structure is a specific way that data is stored and organized. The importance of choosing precisely what data structure to use varies greatly and is dependent on myriad variables: necessary capacity, frequency of both storing and searching, the need to be sortable, the stored data type, and many more.

One such data structure used frequently is a Binary Search Tree (BST). Due to its inherent capability to be performant, BSTs are frequently used for a multitude of domains, from compression algorithms—jpeg, mp3—to syntax trees—used by compilers for expression parsing.

There is, however, one primary problem with BSTs: the performance of insertion into or extraction from them can be greatly decreased if no balancing algorithm exists. A BST balancing specifies how the structure will be balanced, to ensure that it’s data storage is non-linear, for if the structure is unbalanced, it may store data linearly—its benefits virtually entirely nullified.

Many balancing algorithms exist, such as Red Black Tree and AVL, but the ability to implement and experiment with such algorithms is limited. The limitation primarily exists because it requires one to know the intricacies of both the BST implementation within the implementation language used and the intricacies of the implementation language itself. Only with such knowledge could one even begin to implement a balancing algorithm on a BST.

Parallel to this, there are few—if any—languages or frameworks that sufficiently benchmark balancing algorithms, which raises the aforementioned issue of experimentation, and disallows one to accurately benchmark self-designed algorithms against those frequently used within the industry, such as the aforementioned AVL or Red Black.

## Topic Introduction

### What is a Binary Search Tree?

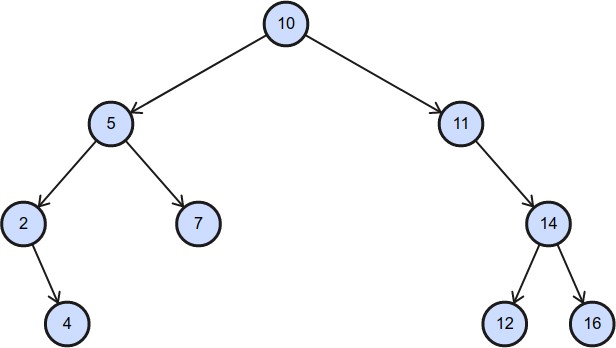
A Binary Seach Tree (BST) is method of organising data hierarchically, allowing its efficient manipulation in a reversed, tree-like structure, with the root sitting at the top of tree. BSTs are commonly used to implement other data structures, such as Hash Trees and Abstract Syntax Trees. It consists entirely of nodes, nodes that store a particularly type—either composite or primitive—and pointers to other nodes. There are essentially two types of nodes: an internal node and a leaf node. An internal node is any node which has pointers to other existing nodes

i.e. an internal node is the parent of its child nodes (the root node is the only internal node which has no parent); whereas a leaf node is any node whose pointers to its children are non- existent, and thus is not a parent.

BSTs typically provide three operations:

* **Search**: when given a value, traverses the BST until either:
  + the value is found; or
  + all nodes have been searched.
* **Insertion**: when given a value, inserts a node into the correct location.
* **Removal**: when given a particular value, searches the BST for the correct node; if found, the node is removed.

An example of a BST:



Within this example, the root node is 10, and its children 5 and 11. Node 11 has one child only, node 14, and nodes 4, 7, 12 and 16 have no children, and thus are leaves.

### What is the Difference Between Balanced and Unbalanced BSTs?

Given a BST with n nodes and a height h, a BST is balanced if and only if 2h-1 ≤ n ≤ 2h; ergo, the previous example is balanced. Balancing BSTs is essential for providing maximum performance, reducing the time complexity of its aforementioned operations. A balanced tree requires less time to performs its operations than an unbalanced tree—with the difference exponentially increasing with node count (an unbalanced tree takes O(log n) for its basic operations; an unbalanced tree, however, is O(n)).

## Project Introduction

This project aims to provide a solution to combat the difficulty of easily implementing and benchmarking BST balancing algorithms. One may ask: “Would not a simple framework for a currently existing language suffice?” To which the answer would be a resounding no. And for what reason the arrival at this answer is that, as aforementioned, this would still require one to understand the intricacies of language that the framework would be created for, which in and of itself causes an even greater problem: it restricts the experimentation with algorithms only to said persons who have knowledge of the implementation language.

Therefore, a fully operational Domain Specific Language (DSL) shall be presented here, which has all common features of basic programming language, such as loops, variable—both scoped and global—functions, conditional expressions, and more. It is hoped that the simplicity of the DSL provides the ease and capability to effectively allow one to implement balancing algorithms.

The language interpreter was originally written Haskell, due to its inherent capability to easily implementt DSLs. However, as this was a new technology learned solely for the purpose of this project—and exists as part of a programming paradigm that differs largely to what was, at that time, known—the interpreter was translated into Python, and thus Python was used to expand and finalise the DSL itself. However, the language choice for the DSL implementation matters very little for this particular project for one primary reason: the language implementation does not necessitate optimal performance.

The project outcome itself is not entirely dependent on the language developed; Instead, this report forms the foundation for the entire project. Sections existing within this report are:

* ***Introduction*:** including the section within which this text is written, the *Introduction* is a description of precisely what the project completion entails, including the problem domain, aims and objectives, and benefits of project outcome.
* ***Context:*** the context refers primarily to provide and explain the current State-of-the-Art work that currently exists within the field of the problem domain. This provides the elucidation of the gaps within the relevant field, and forms a foundation with which the problem solution is derived.
* ***New Ideas:*** As aforementioned, with the *Context* section detailing current State-of-the-Art—thus elucidating the gaps within the relevant field—this section extensively details the precise aspect that the outcome will focus on.
* ***Implementation or Investigation:*** This section largely discusses how the aims and objectives attempt to be achieved, methodologies used for software development, and the tools and resources used to ensure the correct, efficient implementation of the software.
* ***Results / Discussion:*** Detailed within this section is the oucome of the project itself—specifically, how the outcome relates to the proposed improvement within the field.
* ***Conclusions / Future Work:*** This section is a summary of the *Results / Discussion* section, including what has been developed relevant to the proposed aims, and what is uncompleted.

CHAPTER 2

CONTEXT

## Introduction

Discussed and presented within this section shall be the benefits of DSLs in general, supplimented by examples of previous work within the domain of DSLs and how they have benefited the subject area within which they exist.

Approaching this section in this particular way was discerned as suitable for one primary reason: there is very little—indeed, if any—previous work regarding the implementation of environments to ease the experimentation of BST Balancing Algorithms. Thus, when combining this with fact that this project itself focuses primarily on the design and implementation of a DSL, it is discerned as more suitable and illuminating that the benefits of DSLs within other areas will allow one to see the link between the decision to design and implement a DSL for this particular project, and the reasons chosen to implement DSLs within other domains.

### What Exactly is a Domain Specific Language?

Unlike a general-purpose programming language, which allows one to complete and solve a multitude of different programming tasks, a DSL is a (usually small), purpose-driven programming language that exists to allow one to complete a very specific task—or a very specific small set of sub-tasks. DSLs have been created to ease the completion of myriad tasks, from typesetting, to the creation and formatting of web pages, to database technologies, to spreadsheet manipulation and many, many more.

The following is a partial list of commonly used DSLs:

|  |  |
| --- | --- |
| **DSL** | **Usage** |
| HTML (HyperText Markup Language) | Web-based document markup |
| CSS (Cascading Style Sheets) | Web-based document styling |
| LaTeX | Document styling and layout |
| SQL (Structured Query Language) | Managing and manipulating database data |
| UML (Unified Modelling Language) | Visual Modelling |
| CUDA (Compute Unified Device Architecture) | Parallel Computing Platform |
| Makefile | Code Compilation Management |

*(Table 1, Partial List of Commonly Used DSLs)*

### What Are The Benefits Of A DSL?

When considering using a DSL, one must ask oneself: *does using a DSL provides benefits over using a general purpose programming language?* To answer this question, one must understand the benefits of DSLs themselves.

Three primary benefits of DSLs are:

* they are quicker to write;
* they are simpler to understand; thus
* they can be used to complete tasks by non-programmers

***They are quicker to write.*** Due to the precise nature of DSLs, the functionality that the DSL provides is limited: it aims only to allow the completion of a specific task—or small set of subtasks—and thus greatly reduces the complexity of the language itself—only features that aid this completion usually exist.

***They are simpler to understand.*** As aforementioned, DSLs typically provide only the capability to complete a small—or small subset of—task(s). As a result, they are usually devoid of unncessary features, and their syntax greatly reduced in complexity. Take this very small example of LaTeX—a document layout and styling DSL:

**\begin{abstract}**

Just a small example of some abstract text.

**\end{abstract}**

This code automatically formats the abstract section of a report or document—it is not ambiguous; It is precise, assertive and extremely simple.

***They can be used by non-programmers.*** Evident from both previous points, DSLs provide a monumental advantage over general-purpose programming languages: non-programmers (typically) find it must easier and logical to complete tasks within them. Paul Hudak, former and late Professor of Computer Science at Yale University, states this as one the greatest advantages of DSLs (Hudak, 1997) when he says that it

helps bridge the gap (often a chasm) between developer and user, a potentially major hidden cost in software development. It also raises an important point about DSL design: a user immersed in a domain already knows the domain semantics. All the DSL designer needs to do is provide a notation to express that semantics.

## Case Study of DSLs

### LaTeX

As aforementioned, LaTeX is a popular DSL used for the formatting of documents. Primarily used for scientifiec document, due to it’s ability to elegantly format mathetmatical formulae and symbols, it can also be used for all documents of all types, due to its inherent ability to format all possible elements of documents.

LaTeX’s motto is indicative of that which it aims to achieve: “LaTeX is not a word processor! Instead, LaTeX encourages authors not to worry too much about the appearance of their documents but to concentrate on getting the right content” (LaTeX, 2018). And so beautifully it does.

Take the following code, which is a very basic example of LaTeX code:

**\maketitle**

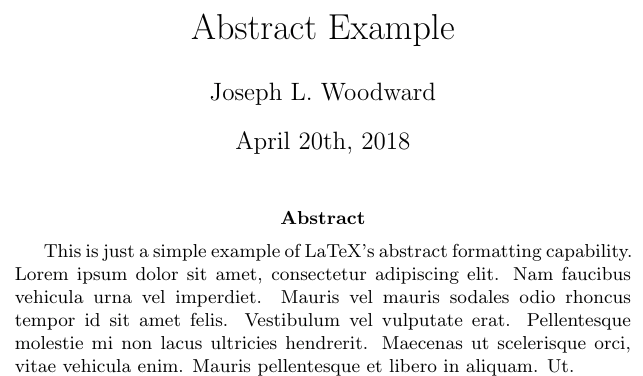
**\begin{abstract}**

This is just a simple example of LaTeX's abstract formatting capability.\**\**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam faucibus vehicula urna vel mperdiet. Mauris vel mauris sodales odio rhoncus tempor id sit amet felis. Vestibulum vel vulputate erat. Pellentesque molestie mi non lacus ultricies hendrerit. Maecenas ut scelerisque orci, vitae vehicula enim. Mauris pellentesque et libero in aliquam. Ut.

**\end{abstract}**

This code produces an elegant front page, which states the document’s title, author, date, and abstract. This code example is simple to use: the syntax is very simple and the keywords exist as simple, elucidatory English-language words. Indeed, it is very easy to discern precisely what this code (aims) to achieve, and, even with very little—even non-existent—knowledge of LaTeX, one can quite simply discern the proposed outcome, which is:



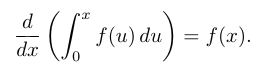
As aforementioned, LaTeX can format mathematical formulae and symbols extremely elegantly. For example, given the code

**\[**

**\frac{d}{dx}\left( \int\_{0}^{x} f(u)\,du\right)=f(x).**

**\]**

the formatted document text will be provided through LaTeX as



Granted, to achieve the first example in Microsoft Word would required little effort, but indeed it does assume that graphical word-processing applications format text documents precisely the same—indeed, this is not the case when considering the two primary graphical word-processing application, Microsoft Word and LibreOffice Writer.

The second—mathematical—example, however, would be extremely difficult in both Microsoft Word and LibreOffice Writer, if not entirely impossible. This would require the usage of another tool, and thus then inserting the formula into the document as an image, which requires great effort and time on the part of the user.

LaTeX achieves the solving of it’s specific domain problem—that of elegantly formating documents—so well, that one particular study shows that over 95% of the mathematical documents inspected were written using LaTeX (Brischoux and Legagneux, 2009).

### Cascading Stylesheets (CSS)

Simply put, CSS files are used to assert styles and layout decisions for HTML elements. CSS stylesheet are simple, easy-to-understand and—at least, when the accompanying HTML is well-formatted—vastly ease the styling of HTML elements.

W3Schools, the premier resource for learning the basics of web-based development, states, simply, that “CSS saves a lot of work. It can control the layout of multiple web pages all at once” (W3Schools, 2018). And this is extremely true.

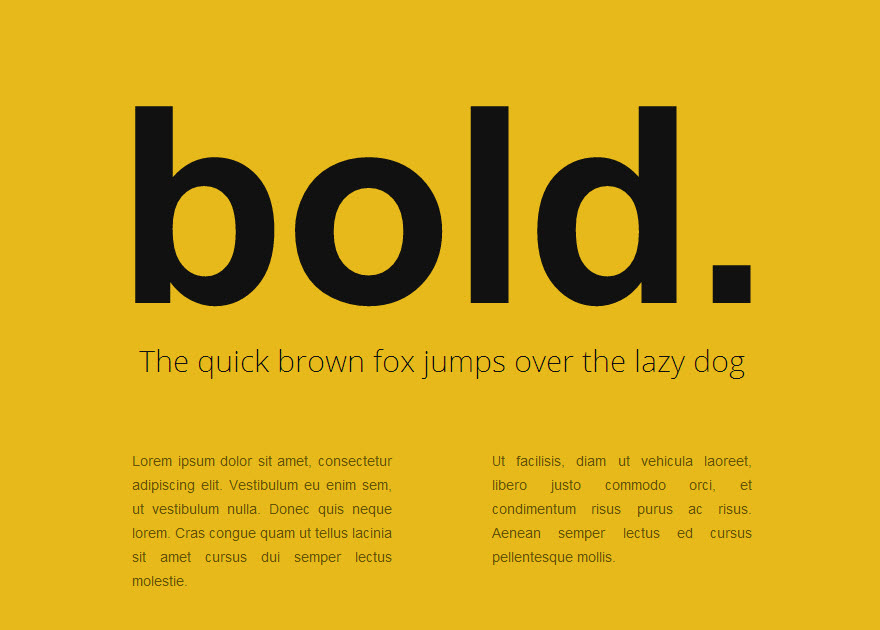
Take the following code:

**h1** { **color**: #111; **font-family**: 'Helvetica Neue', **sans-serif**; **font-size**: 275px; **font-weight**: bold; **letter-spacing**: -1px; **line-height:** 1; **text-align**: center; }

**h2** { **color**: #111; **font-family**: 'Open Sans', sans-serif; **font-size**: 30px; **font-weight**: 300; **line-height**: 32px; **margin**: 0 0 72px; **text-align**: center; }

**p** { **color**: #685206; **font-family:** 'Helvetica Neue', sans-serif; **font-size**: 14px; **line-height**: 24px; **margin**: 0 0 24px; **text-align**: justify; **text-justify**: inter-word; }

Granted, this is a little more complex than the LaTeX example, but, given a correct HTML document, this code produces the following output:



***(Both the Image and Code used within this example come from (WDExplorer, 2018)).***

This is just a small example of what CSS can achieve; Indeed, much more complex, elegant and pragmatic usage is everywhere, on all webpages, on myriad documents and, with the creation Electron—a desktop-based development suite that allows one to use HTML and CSS to design desktop-based application—is becoming increasingly popular with the development of desktop-based application, such as Discord.

CSS is so useful, so necessary, in fact, that it is supported by all web browsers, is used for over 95% of all current websites (W3Techs, 2018), and is the the fourth most popular language used for projects that exist as repositories at GitHub: an astonishing 10% of all GitHub repositories use CSS (GitHub, 2019).

CHAPTER 3

New Ideas

## Introduction

WRITE HERE ABOUT ACTUAL BST BALANCING.

## Aims and Objectives

### Overview

As aforementioned, the importance of BST balancing algorithms is paramount, and this project shall aim to reduce the programming complexity for algorithmic experimentation relative to this. This project aims to implement a Domain-Specific Language (DSL) which will provide high-level abstractions for BSTs, BST operations and a multitude of functionality for the self- development of balancing algorithms.

With this project being aimed at both new and seasoned students of Data Structures & Algorithms, the DSL will allow one to experiment with balancing algorithms without having to implement a BST oneself, which could be a daunting task for a new or seasoned student when one considers that the BST need be polymorphic, efficient, and provide all necessary functionalities for managing its balancing algorithm. Not only will this potentially allow an expansion of interest in BST balancing algorithms for beginning programming enthusiasts, but it decreases the complexity of algorithmic balancing for seasoned students who wish to measure the performance of algorithms in a more performant language, although without the initial overhead of learning the language complexities.

### How will the DSL be implemented?

A major part of this project itself will be the development and implementation of an interpreter, which will be responsible for interpreting the user-defined DSL source code and providing accurate, relative output regarding its operations. The interpreter will interpret the code through a defined semantics which, once the source code is broken down into individual components, commonly known as tokenization, will ensure that a statement is valid if and only if it adheres to the grammar rules.

## Project Scope, Milestones, Main Tasks & Deliverables

### Overview

Ass aforementioned, the primary project objective is to implement a fully-functional DSL, which will provide the capability of allowing BST balancing algorithm exploration for both novice and seasoned students. However, this provides only a broad view of the tasks to be undertaken, and focuses entirely on the language itself, disregarding the project entirety. The project stages (although overlapping) can be viewed as threefold:

* **Research**: the necessary research needed to undertake the project will need to be conducted.
* **Language Implementation**: the language itself—the project outcome—need be fully- functional.
* **Project Thesis:** as documentation of analyses, design, justification and criticial evaluation, a project report will elucidate the rationales for how, why and what exactly the project entails.

### In-scope

First and foremost, the language must work. Given a source code file written in the DSL, the interpreter must produce the correct output; this is the ultimate task. It must initially begin with research, research relative to both BST and programming languages; only then will

enough knowledge be amassed to understand precisely what capabilities the language will need to provide, and how it need be implemented.

Upon completion of primary research, one must implement the BST backend, which will be used by the interpreter to manipulate BSTs declared within the DSL. This backend BST will not only provide typical BST operations, but will also provide complex functionality to carefully manipulate BST balancing algorithms.

Next, the interpreter. This sits as a very large bulk of what allows the DSL to exist. Firstly, a Lexical Analyser need be developed, which, when given a source file, will tokenize the source file, with each token stating precisely what it is: TYPE, IDENTIFIER, KEYWORD, etc. Lexical Analysis methods have existed for ostensibly computing-time immemorial, and are a key component not only of interpreter development, but of compiler development also.

Once the source code is tokenized, it will then need to be parsed. There are opportunities to use a third-party parser, but for both the sake of learning and to ensure sufficient complexity, this project will implement a self-built parser. The parser is responsible for transforming the lexically analysed tokens into a context-free grammar (CFG). Once rules are generated, the parser will use the lexically analysed tokens to generate strings that exist within the grammar, called derviations.

Upon successful completion of both the lexical analyser and parser, the entire interpreter will need to be put together. Initially and to reduce complexity, they will be developed seperately, and will later be amalgamated to form the interpreter entirety. This is a monumental milestone, due to, as aforementioned, the interpreter being the primary bulk of the project, and, once completed, will require only testing.

Testing, however, will be both complex and necessiate pedantry; it will be divided into two sections: interpreter implementation testing and small DSL program testing. The interpreter implementation testing will test a varity of cases, from correct cases to slightly minor failed cases, to major failed cases. The DSL program testing itself will, as an extension of interpreter testing, test to ensure that, given correct code, it produces the correct output; given incorrect code, the program terminates.

The project thesis, which is also a major component of the project itself but differs greatly from the DSL development, is the final—aside from demonstration—milestone to be completed. Although almost last, the project report will be partially completed even before the interpreter has been developed and tested. The final sections necessitate the analysis of the DSLs implementation, results and conclusions, and therefore must start only once the implementation is complete.

### Out-of-scope

Due to time constraints, there are a vast number of functionalities that is hoped to be implemented, but cannot be guaranteed. The primary out-of-scope functionality is a detailed, assertive and informative error-handling system—not too complex, but providing the user with a message that aids understanding.

## Project Risks

### Overiew

As is the case when undertaking any project—perhaps especially one as complex as developing a programming language—there are myriad risks. These risks can range from project member illness to unpredictability of equipment breakage to data destruction. With this particular project being completed individually, devoid of necessitating the usage of external equipment (exempting computer access and a hard drive), the primary risks involved exist within one of two categories:

* **Task Overflow:** Tasks taking longer than expected.
* **Implementation Language Knowledge**: Lack of knowledge regarding the implementation language.
* **Data Destruction**: data becomes inaccessible.
* **Project Member Propensities**: behavioural tendencies of project members that negatively impact on the completion of tasks.

### Task Overflow

Task overflow is extremely common within software development. This overflow usually occurs due one—or both—of two reasons: Unrealistic time assignment to individual tasks; And inability to overcome sub-tasks.

To combat these, tasks shall be allocated time that far surpasses the approximate task completion time, ensuring that if a task takes longer than expected or problems occur within sub-tasks—as is the case quite often within software development—then said time allocation will provide a time-period to complete them.

### Implementation Language Knowledge

Due to the writing of the DSL interpreter in Haskell, there is a possibility that a lack of Haskell- and functional programming-specific may cause some issues. To combat this, there will be frequent communication with the project supervisor—an individual who has vast knowledge of functional programming and specifically Haskell—whereby guidance should provide the ability to overcome such problems.

However, if this becomes a big problem early on within the implementation phase, Haskell could possibly be replaced by something simpler, something quicker to develop in, something that the developer has more knowledge about.

### Data Destruction

Data destruction exists in many forms, such as corruption, failure to save files, hardware fault, accidental power-off, and more. Within this project, the primary potential data desctruction refers to a hardware fault. To combat this (and other potential descruction), the project updates will daily be backed-up to two places: the cloud, and an external hard drive. Ergo, if a hardware fault occurs, there will be the ability to restore data that is at most one day old.

### Project Member Propensities

Undertaking arduous endeavours requires discipline, motivation, passion, intelligence, justification, planning, and much more. These are but a small sample of necessary behavioural components that one must ensure strict adherence to when undertaking a large, individually- based project. An inability to abide by these components must be remedied near- instantaneously—what seems like a small setback initially may transform into an ostensibly insurmountable setback, strengthening one’s self-perceived inability to continue. To combat this, there shall have weekly meetings with the project supervisor and project member, and, if negative propensities persist, it shall need to be discussed.

## Gantt Chart

The following Gantt Chart provides an illuminating view of how tasks have been assigned—both the order and the time allocation:

***(For a more graphical view, please see Appendix A)***

CHAPTER 4

IMPLEMENTATION

# Software Development Life-Cycle

A SDLC is a framework [that defines] tasks performed at each step [of a] software development process, . . . a structure followed by a development team” consisting “of a detailed plan describing how to develop, maintain and replace specific software” (Techopedia, 2018).

Having never previously worked with defining and implementing a programming language, the decision was made to use an Iterative SDLC model. Joseph Woodward (Woodward, 2017), describes the Iterative model as

a cyclical process that requires only a minor planning phase, a phase which details just enough requirements to allow a rapid prototyping of an application. Once this prototype is developed, then a small handful of stages are cyclically repeated, adding extra features and elimnating bugs with each cycle.

This was discerned as a suitable for several reasons. Firstly, this particular model would allow the rapid development a prototypal language, one that would provide basic language capabilities, such as variable decaration and definition, and console output. Secondly and depedent on the aforementioned point, this would allow the language features to be implemented iteratively: once one feature had been designed and implemented, the next feature could then be worked on.

# Language Design

## Syntax

Intentionally, the DSL syntax style correlated quite closely with that of the already existing, niche and functional language Lisp. Beaten only by Fortran, Lisp is the second-oldest high-level programming language whose use is still fairly vast, even today: the Tiobe Index—a popular website used to detail the popularity of programming languages—ranks Lisp as #21, beating the increasingly popular D Language and Scala.

Lisp pioneered, amongst other ideas, tree data structures, and thus it was this reason plus two more that it was discerned a suitable syntactic style for this particular DSL: Firstly, assuming code is written with illuminating spaces and carriage returns, the syntax is very readable; And secondly, the syntax itself is inherently and aesthetically beautiful. The parenthesized prefix notation—which enforces the pseudo-rule that everything (statement, expressions) are lists—is elegant, and minimizes the necessity to learn other starting and termination symbols for expressions and statements themselves.

When thinking of language grammar, it is best to formalise a Context-Free Grammar (CFG), which is, essentially, a set of rules such that all possible code written within the language can be “reduced independent of the symbols to its left and right in the text” (McKeeman, 2007).

Backus-Naur Form (BNF), which is a mathematical way to formally define a programming language, uses CFG to negate ambiguity of the programming language that adheres to it. BNF states that, when given the rule:

**left-hand-side ::= right-hand-side**

what is meant precisely is that the left-hand-side can be replaced by the right-hand-side.

program ::= ‘(begin ’ expr\* ‘)’

expr ::= exprArithmetic

| exprConditional

| exprFunction

| exprLoop

| exprPrint

| exprConcat

| exprDefine

| exprLen

| value

exprArithmetic ::= ‘(‘ operator expr expr ‘)’

exprConditional ::= ‘(if ‘ exprArithmetic expr\* expr\* ‘))’

exprFunction ::= '(func ' atom ‘(‘ atom\* ‘)’ expr\* ‘)’

exprLoop ::= '(loop ' expr expr atom expr\* ‘)’

| '(while ' expr atom exprArithmetic expr\* ‘)’

exprPrint ::= ‘(‘ ('put' | 'putln') expr+ ‘)’

exprConcat ::= '(‘ concat expr+ ’)'

exprDefine ::= '(var ' atom expr+ ‘)’

exprLen :: ‘(len ’ string ‘)’

operator ::= < | > | <= | >= | && | \* | / | ^

| + | - | % | == | != | “neg”

value :: string | number | atom | bool

string ::= “\”” (.\*?) “\””

number ::= (\d+)

atom ::= (^[A-Za-z][A-Za-z0-9\_]\*)

This grammar allows the assertion that the source code written using the DSL is either legal or illegal. As an example, given the code snippet:

**(var var\_name “test var”)**

we can formally check, through derivation, whether the snippet adheres to the formalised CFG:

→ expr ::= ‘(‘ exprDefine ‘)’

→ expr ::= ‘(var ‘ atom expr+ ‘)’

→ expr::= ‘(‘ var var\_name expr+ ‘)’

→ expr::= ‘(‘ var var\_name value ‘)’

→ expr::= ‘(‘ var var\_name string ‘)’

→ **expr ::= ‘(‘ var var\_name “test var” ‘)’**

And from this, we have asserted that the code snippet is legal. However, if we modify the code snippet given slightly, such that we replace var\_name with a string as follows:

**(var “invalid atom” “test var”)**

we can clearly verify that the syntax is illegal:

→ expr ::= ‘(‘ exprDefine ‘)’

→ expr ::= ‘(var ‘ atom expr+ ‘)’

**→ error: “invalid atom” is not a valid atom; instead, it is a string**

# Language Features

## Overview

The features that exist within the DSL were partially determined when considering the formal grammar, although only a small subset. Initially, what the language needed were:

* Conditional statements; and
* Operator expressions.

Initially, the intention was to treat the entire DSL as a BST itself, and allow operations on the BST from within itself. For example, calling something like *update\_parent* would update the node that is currently being balanced—remember: the language was intended to define BST balancing algorithms; the actual BST balancing would be done within the backend.

However, there were some complications with this: the inability to create variables, loops, etc—in essence, the inability for the DSL to function as an actual language—was difficult to accept, to acknowledge, conceptually. Thus, it was discerned relative early on that instead of approaching the DSL as though the source code existed as a BST itself, a fully-functional, basic programming language would be much better suited, and thus, a multitude of extra features were adopted.

The full list of language features are:

* **Console Output:** outputting values to console.
* **Functions:** creating, defining and calling functions.
* **Variables:** creating, defining, modifying, and using declared variables.
* **Loops:** creating and defining the behaviour of two distinct loops.
* **If / Else:** execute expressions based on satisfying a condition.
* **Arithemetic Expressions:** mathematical expressions, such as multiplication, power, modulus, division, etc.
* **String Manipulation:** obtaining a string length, and concatenating string

## Feature Implementation

### Console Output

The DSL provides two kinds of console outputs:

* **put:** output the expression following the put, without postfix carriage return; And
* **putln:** output the expression following the putln, with postfix carriage return.

These features seem simple, but there is one greatly benefit of the way that console output is implemented within this DSL, illuminated by the language used above—‘output the ***expression***following . . .-’-and that is the ability to output not only values, but to also output a specific value based off of an expression.

For example, consider the code snippet:

**(putln (if (< 5 10) “if condition hit” “else condition hit”))**

the output will be, as discernible from the condition, ***if condition hit.*** This behaviour mimics the behaviour of ternary operators in many imperative and declarative languages. The following juxtaposition illuminates this:

**DSL: (putln (if (< 5 10) “if condition hit” “else condition hit”))**

**C#: Console.WriteLine(5 < 10 ? “if condition hit” : “else condition hit”);**

This expression output—as opposed to a simple value output—also allows the concatenation of strings, which is discussed shortly.

### Functions

The ability to create, define and call functions within the DSL is, by some margin, the most impressive feature. Functions can typically be seperated into one of two categories:

* **Action Functions:** functions that perform a specific task; And
* **Functional Functions:** functions that return a value.

Regardless of whether a function is Action or Functional, the function definition format is:

**(func FUNC\_NAME (FUNC\_ARGS) (**

**FUNC\_DEFINTION**

**)**

That is, where FUNC\_ARGS are the function’s argument, seperated only by whitespace, e.g. (arg1 arg2). All function argument used within the function definition are replaced with the passed values, due to the DSL’s stack-pushing function implementation.

#### Action Functions

Action functions are, as aforementioned, used to perform a specific task. The specific task can be anything from output to variable redefinition. An example of a simple action function could be:

**(func action\_func (arg1 arg2) (**

**(putln “Testing action func”)**

**)**

#### Functional Functions

A functional function, however, is, as aforementioned, a function that performs a caluclation, and thus implicit returns the value. An example of a simple functional function could be:

**(func cube (x) (**

**(^ x 3)**

**))**

The function definition states that, when calling the function like so: ***(putln (cube 5))*,** the result will be a console output of the result of ***(^ x 3)***, or x to the power of 3.

#### Recursive Functions

Recursion is a beautiful concept that is allows one to call a function from within the function definition itself. This concept is also largely used for the three primary capabilities of BSTs: search, removal and insertion. Recursion is greatly beneficial to it being “clearer, simpler, shorter, and easier to understand than [it’s] non-recirsuve counterparts [and] directly [reflecting] the abstract solution strategy (algorithm)” (Ualberta, 2018).

Due to the implementation of functions within the DSL—a function call is added to the stack, and garbage collected (inconsistently, due to Python being a managed language) once the calling expression terminate—recursion is possible. As an example, to recursively calculate the factorial of a given value within the DSL, one could create a function such as:

**(func my\_factorial (x) (**

**(if (<= x 0) (1)**

**(\* x (my\_factorial (- x 1)))**

**)**

**))**

Which, when calling the function like so: ***(putln (my\_factorial 5))*** will result in a console output of 120, the correct value for !5.

#### Higher-Order Functions

Higher-Order Functions are function that allow one to pass a function as an argument to another function. This is greatly beneficial due to the fact that it “not only shortens programs, but also produce clearer programs because the intended meaning of the program is explicitly rather than implicitly stated” (Ericsson, 1999).

Such a feature is prominent within many modern programming languages, with the following highlighting how it is used within several of them—including this DSL itself:

**Haskell: apply f x = f x**

**JavaScript: function apply(f x) { return f x; }**

**Python: def apply(f: "function", x: int) -> int: return f(x)**

**C#: public static int apply(Func<int,int> f, int x) { return f(x); }**

**DSL: (func higher\_order (f x) ( (f x) ))**

These examples elucidate the stylistic similarities between the DSL’s usage of higher-order functions, and thus render the usage of them within the DSL simple, minimal, elegant.

### Variables

Variables are, at least in imperative and declarative programming languages, explicitly decalred and defined, and within this DSL it is no different. Variables follow the format:

**(var VAR\_NAME VAR\_EXPR)**

where VAR\_NAME is the variable name, and VAR\_EXPR is an expression whose outcome is bound to the name of the variable. It is worth noting that, due to how the expressions are handled—also elucidated within the section ***Console Output,*** one can use a conditional statement when binding a value to a variable name, such as:

**(var test\_var (if (< 5 10) “if condition hit” “else condition hit”))**

This particular expression will, as with a former example, result in the creation of a variable with name ***test\_var***, which will be—at least, at the time of declaration and definition—be a string variable with value “***if condition hit”.***

Variables declared within the DSL are scope-specific: that is, they exist within one of two scopes:

* **Global-Scope:** accessible in any scope; And
* **Function-Scope:** accessible only within the function that it is declared.

#### Global-Scope Variables

Global-scope variables are accessible to all expressions, within any scope: function, nested expressions or otherwise. Modifying the variable inside any of said scopes will thus modify the variable in all scopes. The only limitation on declaring global variables is that they cannot be declared within the function scope.

#### Function-Scope Variables

Function-scope variables are, however, much different. Function-declared variables are locally scoped, such that declaring and defining a variable within a function will, when the function is called, create the variable for usage inside of the function only. Once the function call exits, the variable will cease to exist, and any attempt to access it will causes a key error, causing the application to displaying a key error message and terminate.

### Loops

Once again, loops exist as a common feature within imperative and declarative programming languages—function programming languages such as Haskell, however, does not have loops and thus uses recursion for it’s necessary loop-like effects.

Loops within the DSL exist either:

* **For-Loop:** mimics the for loop of other languages; Or
* **While-Loop:** which, once again, mimics the while-loop of other languages.

#### For-Loops

For loops are extremely beneficial, as they allow one to specify a start number, an end number, and a value that represents the current number within the range START → END, that one can use to achieve a desired effect. The DSL’s for-loop follow the format:

**(loop START\_NUMBER END\_NUMBER ATOMIC\_VALUE (**

**LOOP\_EXPRs**

**))**

where ATOMIC\_VALUE is the identifier for the current number of each for-loop iteration; START\_NUMBER is the number that the for-loop will start at; And END\_NUMBER the number that the loop will terminate at.

As a clear example, consider the following code snippet, which will output all number from 5 to 15:

**(loop 5 15 index (**

**(put (concat index “ “))**

**))**

which will result in the console output “***5 6 7 8 9 10 11 12 13 14”****.*

#### While-Loops

While loops work differently than for-loops, as is the case with languages that implement the both of them also. Instead of specifying an end number, instead, one specifies a condition, and the loop will terminate only when the condition hits, with entire format being:

**(while START\_NUMBER ATOMIC\_VALUE conditionalExpr (**

**LOOP\_EXPRs**

**))**

where START\_NUMBER and ATOMIC\_VALUE are the same as for-loop, and conditionalExpr is the aforementioned expression, whereby the loop terminates only when the condition fails.

As a clear example of a while-loop within the DSL, we can mimic that of outcome of the previous for-loop example by writing:

**(while 5 index (< index 15) (**

**(put (concat index “ “))**

**))**

which will result in precisely the same output of the for-loop.

### If / Else

If / Else statements exist for the majority of commonly used programming languages. This feature is beneficial for allow one to execute a designated section of code if a specified condition is true; If it is not true, it will then execute the second section of designated code—if, indeed, the language requires that an if statement must be accompanied by an else statement, and in some languages, it does not.

In the DSL, however, the if statements must be accompanied by and else statement, albeit that the else statement may simply be an empty list, such as ***().*** The format used for if / else statement in the DSL differs from many explicit if / else languages—indeed, this is due to it being based on Lisp. The format is:

**(if CONDITIONAL\_EXPR PASS\_EXPR FAIL\_EXPR)**

where CONDITIONAL\_EXPR is the condition that must be passed in order to execute the ***if*** code section, PASS\_EXPR is the expression that is executed if CONDITIONAL\_EXPR is true, and FAIL\_EXPR is the expression to be execute if the condition fails.

For a real-world example of this, let’s assume one wishes to assign a number to a variable based off a condition, and that the required sqr—which squares a number—and cube—which cubes a number—has been defined. The example would look like so:

**(var sqr3\_gt\_cube2 (if (> (sqr 3) (cube 2))**

**(“sqr 3 gt cube 2”)**

**(“sqr 3 lt cube 2”)))**

Once again, due to the way that expressions are evaluated, this if statement acts like a ternary operator, which removes code redundancy, as, if it did not, then the code may look something like:

**(if (> (sqr 3) (cube 2))**

**(var sqr3\_gt\_cube2 "sqr 3 gt cube 2")**

**(var sqr3\_gt\_cube2 "sqr 3 lt cube 2"))**

which repeats the variable assignment.

### Arithmetic Expressions

Initially, the interpreter for the DSL was written in Haskell, and didn’t apply the same grammar rules as what it now does, namely that it was not Lisp-y, and thus expressions were not simply determined by the surrounding parentheses. This resulted in an initial need to implement a grammar that handles operator precedence, such that the expression ***12 + 3 \* 4 – 1*** correctly performs the calculations based on the priority of the operator.

In this example, without operator precedence, the derivation and consequent result would be:

→ (((12 + 3) \* 4) – 1)

→ (((15) \* 4) – 1)

→ ((60) – 1)

→ **59**

However, adhering to BODMAS—an acronym that describes ordering with mathematics (Brackets, Orders, Division, Multiplication, Addition, Subtraction)—this result is incorrect. Instead, the derivation would be:

→ 12 + (3 \* 4) – 1

→ 12 + (12) – 1

→ 24 – 1

→ **23**

As important as operator precedence is, the nature of the DSL syntax disregards the need to deal with it, due to, as aforementioned, it’s parenthesized expressions. To provide a clear example of this, the code to translate the given expression into a syntactically legal expression within the DSL, you would have to use:

**(- (+ 12 (\* 3 4)) 1)**

which, evidently, requires the user to specify the individual expressions themselves.

Now, the DSL provides myriad arithmetic expression operators, such as multiplication, division, power, addition, subtraction, modulus, etc. And, if one requires something a little more advanced—such as first adding two numbers and then diving them—one can define a function to handle this, such as:

**(func add\_then\_divide (lh\_add rh\_add div) (**

**(/ (+ lh\_add rh\_add) div)**

**))**

### String Manipulation

String manipulation tools are invaluable for cleverly altering or extracting a specific value from the contents of a string. This particular DSL provides two primary string manipulation tools:

* **concat:** which allows the concatenation values; And
* **at:** gets the string element at the index given
* **len:** which returns a string’s length

#### Concat

String concatenation is greatly beneficial for myriad reasons. However, the primary reason is that it allows the joining of values to be used for console output. String concatenation usually requires that all values to be concatenated also be of type string; However, this DSL allows the concatenation of multiple types: boolean, number (integer, decimal) or string.

To use a real-world example, let’s say that a debug logger needs to record whether a particular globally stored variable is greater than another. The DSL would write the code—assuming ***var\_one*** holds the value 10 and ***var\_two*** 20—like so:

**(putln (concat "var\_one is greater than var\_two? " (> var\_one var\_two)))**

The output? “***var\_one is greater than var\_two? False”.*** If such an ability to concatenate boolean values with strings did not exist, then the code would necessitate extra verbosity:

**(putln (concat "var\_one is greater than var\_two? "**

**(if (> var\_one var\_two) "True" "False")))**

or, even in the case of such a well-developed language as C#:

**Console.WriteLine("var\_one>var\_two? "+(var\_one>var\_two).ToString());**

#### At & Len

These two capabilities—at and len—are discussed together due to their intrinsic connection and usage, and their ability to—when combined—produce outstanding capabilities.

Quite simply, at returns the character of a string at the index given, if it exists. If it does not exist, an error message is thrown that illuminates an invalid index on the string, and the application terminates. The format of an at expression is:

(at STRING\_EXPR INT\_EXPR)

where STRING\_EXPR will be an expression whose outcome is a string literal; And INT\_EXPR will be an expression whose outcome is an integer literal. ***Note: indeces start at 0, as with array indeces in the majority of languages—indeed, a string is simply a char array.*** To use a real world example within the DSL, we could do:

**(putln (at “test string” 7))**

which will result in the console output of “r”.

The usage of len is reasonably axiomatic—len retrieves the length of a string. If the given expression results in an attempting to find the length of an integer, then an error message is thrown, which alerts the user of this, and the application terminates. The format of a len expression is:

**(len STRING\_EXPR)**

where STRING\_EXPR is an expression whose outcome is a string literal. To once again use a real-world example within the DSL, we could do:

**(putln (len “test string”))**

which would result in the console output of 11.

The power of these two features are magnified when used together, and specifically when used together with loops. Let’s say that an application written within the DSL needed to output individual characters of a particular string, seperated by a particular character—in this case, a “-”—then we could write the code:

**(func output\_with\_seperator (string sep) (**

**(var string\_len (len string))**

**(loop 0 string\_len index (**

**(put (concat (at string index) (if (< index (- string\_len 1)) sep "")))**

**))**

**))**

which could be called with ***(output\_with\_seperator "test string" "-"),*** and would result in the console output of “***t-e-s-t- -s-t-r-i-n-g”***.

CHAPTER 5

RESULTS / DISCUSSION

# Introduction

* What was achieved
* what is still left to be achieved (with the language)
* how the language fails to allow BST experimentation
  + and how would it do this?

CHAPTER 6

CONCLUSIONS / FUTURE WORK

Introduction

Whatever it was that your results showed should be summarised here. Hopefully the conclusion will be that your proposals proved to be brilliant and now the results bear this out. On the other hand your proposals may, in the light of the results obtained, prove to be less successful than you had hoped. In this case the conclusions should state why.

In either case there should be some reference to future work, either to forward and expand on the successful outcome or to test ways of overcoming the shortfall in your ideas that didn't work out quite as expected but there should be something that shows you can see further implications of what you have achieved.

This chapter should also include a discussion of the four PSEL issues (Professional, Social, Ethical and Legal) and the way in which you project has/will/could impact on each.

ReferenceS

<https://pdfs.semanticscholar.org/b06c/06de5335a0e53ad7122419886890c2cab2a4.pdf> - benefits of a DSL

<https://www.latex-project.org/about/> -- latex motto

<http://www.cebc.cnrs.fr/publipdf/2009/BTS24_2009.pdf> – latex usage

<https://www.w3schools.com/css/css_intro.asp> – css

<https://w3techs.com/technologies/details/ce-css/all/all> – css usage

<http://githut.info/> -- css usage again

<https://www.techopedia.com/definition/22193/software-development-life-cycle-sdlc> -- what is a SDLC

<http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/Grammar.html> – cfg

<https://webdocs.cs.ualberta.ca/~holte/T26/rec-prog-tech.html>– recursion

<http://erlang.org/documentation/doc-4.8.2/doc/extensions/funs.html>– higher order functions

Bibliography

Vogt, C. 1999. Creating Long Documents using Microsoft Word. Published on the Web at the Nottingham Trent University.

Coote, H., Dobbs, B. & Jones, C. (1996). Defining databases. Wiley: Melbourne.

Applications and Science in Soft Computing, Lotfi, Ahmad; Garibaldi, Jonathon M. (Eds.) 2004, X, 346 p. Springer, ISBN: 3-540-40856-8

**Note:** References are a list that includes the essential bibliographical details for each item to which you have referred in the body of your paper. It should ONLY include items to which you have made direct reference. A direct reference is where you have quoted/reproduced text or diagrams from another author or mentioned/referred to the work of another author in your report. That is quoted directly what they have said about something or mentioned their views or conclusions in your report. For details of citation and references see the information in the Project Guide.

A Bibliography is a list of published materials that you have read or consulted for general information in the preparation of your work, concerning the subject of your Project, but have not made any direct reference to in your report i.e. 'background reading'.

You should always provide a Reference List. **A Bibliography is optional but when provided it should include all items in your Reference List as well as any additional items consulted in preparation of your work.**

Appendix A

The content of these will differ with the different types of project. Any design and analysis charts/diagrams will be included here in full. In projects where software has been developed there will be an appendix for this. Our departmental requirement is that a CD, DVD or USB memory stick of all source code is submitted to your project supervisor. The appendix contained in the report will refer to this CD, DVD, or USB memory stick, provide a directory style listing of the files submitted and instructions for rebuilding and running the software. This might be source code of programs written in high level languages (C, C++, etc) together with any pertinent files ('make' files, non-standard libraries, etc). Alternatively, or in addition, you can place some or all of the source code in the appendix. In any case the source code needed to reconstruct any software you have developed must be submitted in its entirety in the CD, DVD, or USB memory stick. (Any code that has been used from a third party should reference the original developer).

Hardware designs will require schematics/circuit diagrams, PCB layouts, simulation tests and pin outs.

Most projects will require some form of user documentation to explain how to use the software/hardware produced. A researcher following up the work may wish to utilise the work of the original author and an appendix laying out the format of input files and how to interpret the output is required.