Final Year Project

An Extension to Dijkstras Guarded Command Language

Edward O'Neill

Student ID: 17342691

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Professor Fintan Costello

UCD School of Computer Science
University College Dublin

May 6, 2021

Table of Contents

Details

The link to the code repository can be found here : https://csgitlab.ucd.ie/EdONeill/Olivia

Installation requires the installation of the Haskell compiler and its package management system, Cabal.

The Windows and MacOS installations has been tested and work however I'm not sure if the Linux versions work as I do not have access to an Linux machine. If there are any issues in installation, please contact me at edward.oneill@ucdconnect.ie

Abstract

Language plays an important role in how one constructs correct programs in both natural languages and programming languages. If one wishes to construct such programs, it is necessary to be able to express oneself such that correctness follows naturally from the language one uses. Dijkstras creation of the Guarded Command Language facilitates this through its use of guards; Boolean constructs which only when evaluated to True permit the execution of commands or statements. My aim for this project is to show that the Guarded Command Language possesses qualities that should be considered for all languages by implementing a language which is inspired by its non-determinism and mission of correctness. I will extend it however to include function and object definitions to show that this idea of correctness can be used to construct meaningful and worthwhile programs.

Chapter 1: **Introduction**

The purpose of this Final Year Project and paper overall is to attempt to address the problem regarding how one constructs a correct program. It is seldom spoken about because the act of writing a program is often regarded as a trivial exercise to what one is actually attempting to do such as implementing a feature within a web application or solve a programming assignment. The consequences of this has left programming as some quasi-mystical art. Those that can "program" are deified within freshman undergraduate Computer Science programs. Industry filters candidates based on problem sets not too dissimilar to those prescribed in the same undergraduate classes. There is such an emphasis placed on the ability to program and yet it's trivialised. I believe that Dijkstras work attempted to show why it should not be trivialised. His Guarded Command Language shows that the foundations of Computer Science, Mathematics can be used rigorously to construct elegant solutions to otherwise complicated problems. This is not to suggest that elegant solutions cannot be derived using preexisting languages. The language instead shows that the syntax of the language can aid one in reasoning about constructing a program.

The language designed and presented for this project is heavily inspired by Dijkstra and his work regarding program correctness. Constructing a correct program is something that should be striven for within programs of all nature and complexity. In his paper *EWD 361 Programming As A Discipline Of Mathematical Nature*[1], Dijkstra notes that a crisis emerged based around the question of how can we rely on our algorithms because after all, the program that passes n unit tests could fail on the $n + 1$ unit test. A language designed to produce correct programs however circumnavigates this issue. A program that we know to be correct is one that we know to be reliable. The language designed thusly is one that uses a minimal set of features set by Dijkstra to aid in the construction of correct programs.

Chapter 2: **Project Specification**

It is imperative to explain what a guarded command is before I delve into the notation that will be used. The Guarded Command Language operates upon the idea of pre and post conditions. The only way to move to the post condition is by executing a guarded command. These commands are bounded to a Boolean expression which begets the statement execution if and only if it is evaluated to True. These guarded commands are extremely useful. Structured programming and the emergence of non-determinism within the Guarded Command Language is such an interesting feature that lets us think about computation in a more natural way.

To explain the power of the language and guarded commands, consider the problem wherein we wish to find the sum of the even elements of array *f*.

$$f = 1, 2, 3, 4, 5$$

A program is constructed through the *Eindhoven Quantified Notation*. It is a very powerful notation that does two things. The first is that it uses the idea of pre and post conditions within the construction of programs. A pre condition P is operated upon by an action/statement S to give the post condition Q in that current state. This change is denoted by an expression, the string "= []", and another expression. The second thing this notation achieves is the ability to reduce a problem by recognising that the problem could be simplified but still retain the same meaning. Therefore if we wanted to find the even number sum the problem may look like the following:

$$r = isEven?(f[0]) + ... + isEven?(f[n])$$

Well if the array consisted of a million elements, it wouldn't make too much sense to write out the summation this way therefore we can change it to the following expression:

$$< +j : 0 \leq j < n : g(f.j) >$$

This looks a bit esoteric to what we had before but if you bear with me, I will show that its much more expressive and powerful. The first section refers to the operator which is being applied; in our case its addition because we're just finding the sum. The single letter *j* exists to inform us that we are only concerning ourselves with a single index of *f* at a time. The middle refers to the range that we're concerning ourselves and usually the problem forces us to consider the zeroth index to some arbitrary point *n*, in this case five. The last section is very interesting. In this section one can apply a predicate to the index of *f* like in our case, if the number at the index is even or not.

Every construction begins a model in which we can work from. First, consider a function *g* which performs a case analysis on an integer to check if it is even or odd.

Model

* (0) g.x = x ← even.x

* (1) g.x = 0 ← odd.x

As I will go into later, correctness guiding the proof is one of the qualities that the language possesses and it can be seen here. There's only two cases that we need to be concerned with : is a number even or odd? Therefore the function *g* only needs two entries. The next is to include our *Eindhoven Quantifed Notation Expression*.

* (2) C.n = < + j : 0 ≤ j < n : g(f.j) > , $0 \leq n \leq 5$

We have to consider now where we could go from here. A reasonable first step to take is to see what value is given when *n* is instantiated to zero.

$$< +j : 0 \leq j < 0 : g(f.j) >$$

= { Empty Range }

$$0$$

Since the range is empty, what we obtain is zero.

. (3) C.0 = 0

Right now that we know what value is given by C.0 but there is an issue on how to proceed from here. We could test C.1 but then we would have to test C.2 and so on which definitely misses the mark. Look at (2) : This essentially tells us what value we have at any given time. We used this to see what value we had when *n* was zero so why couldn't we say lets check the value after the current value that we know? If we think of it this way, we are extrapolating the logic of observing the change of C.0 to C.1 and so on to the more general C.n to C.(n + 1).

$$< +j : 0 \leq j < n+1 : g(f.j) >$$

= { We can't go directly to n so we split off this term }

$$< +j : 0 \leq j < n : g(f.j) > +g(f.n)$$

= { From observing the model, the left hand side of the expression is (2)}

$$C.n + g(f.n)$$

If we look at the fact that the function *g* returns two different values, we infer that there will be two cases in which C.n is added to f.n when f.n is even and zero when odd.

. (4) C.(n + 1) = C.n + n ← (0), $0 \leq n < 5$

. (5) C.(n + 1) = C.n + 0 $\leftarrow$ (1), $0 \leq n < 5$

The model is now completed. The next is to create an invariant. An invariant is something which doesn't change throughout the program and is maintained as such. The best thing to use is C.n as that gives us the sum of the program at the end and is the easiest to maintain.

Invariants

P0 : r = C.n

P1 : $0 \leq n \leq 5$

After creating invariants they need to be established and the model tells us exactly how. If we look back to the model, we know the value of C.0 is the initial value that C.n has and if this is also our variant then we can say at this point C.0, the invariant is still maintained and True.

After establishment we need to look to what the guard and variant could be. Since the guard begets the loop, it has to be a True statement otherwise we wouldn't be able to execute the program. Considering this, we know that *n* cannot be at the end of the list otherwise we couldn't sum the list therefore we can propose the guard to be $n \neq 5$ because when it reaches five then we know to exit the loop otherwise we can continue executing the body of the loop.

The variant of the program tells us how much work is left to do within the program and we can look to the guard for help in deriving this. If we end at five then we know the work that is left to do is the amount of places between this point and whatever point we're on or in other words, *5 - n*.

The next step is the loop body. In order to derive the loop body we can simply consider the fact that the function *g* gives us two things to consider by its own definitions. We know that the variable *r* is bound to *C.n* and thus stores the current sum therefore it either adds the value at the index of *f* if it is even or zero if it is odd. We also know that *n* increases by one regardless if the number is even or odd then we can conclude that there exists two guards within the loop body : if the number is even add the value to the current sum and if its odd add zero.

This gives us the final program

Listing 2.1: Reduction - Finding The Sum Of Even Integers

```
n, r := 0, 0
;Do n != 5 ->

    if even.n -> n, r := n + 1, r + f.n
    [] odd.n  -> n, r := n + 1, r + 0
    fi
Od
```

2.1 Core Goals

The Guarded Command Language possesses only five operations[1]: *Assignment, Repetition, Skip, Concatenation,* and *Selection*. These are core to the Guarded Command Language and will thus will be core to my language. These operations are implied through the proof methods of the *Eindhoven Quantified Notation*. In general, *Assignment* is implied by the proprieties and need to change state in a program, *Repetition* comes from the fact that one operates upon something repeatedly within a program, *Skip* comes from the idea that some *Selection* branches could be skipped, and *Selection* is implied by the use of functions found within the third field of a Quantified Expression.

Assignment is a basic feature that exists in every language and without it, the language would be meaningless. It is a key building block in programming languages much like how verbs are key building blocks in natural language. A collection of registers in a computer or a collection of nouns in a language are meaningless if they can't be used. It's notation is denoted by the string ':='.

Repetitions notation is clean and unambiguous. It is denoted by the *Do..Od* notation from the program above.

Where there exists ";", the expression before and after it are concatenated together. For example in the program above, the *Do* block is concatenated with the instance of *n* and *r*.

Skip is the most interesting operation of the Guarded Command Language as it representative of its more obvious mathematical roots. The function of *Skip* could be thought of as the identity of a program. *Skip* takes the current state of the program and gives you back the current state of the program.

$$Skip(Program) \equiv Program$$

In a program, *Skip* would arise whenever there exists Selection. Its uses are numerous but one use is to avoid the issue of side-effects. Side-effects could arise in an If-Else block in all languages if one is not exhaustive and explicit in their conditional branches therefore *Skip* allows you to "skip" over conditionals that don't need to be evaluated but would otherwise cause side effects and still continue the program.

The notation of Selection is indicative of how it behaves. It lays out a series of one to many guards. These guards are interpreted such that when a guard is evaluated to True then the statement(s) bounded to it are taken. Non-determinism of course arises because if multiple guards are True then the True branches are taken until one completes and thus completes the program.

If we're purely speaking about writing a language, I find this notation to be much cleaner than *if...elif...else* blocks. Laying out the guards and their consequences one after each other

is much easier to read. The benefit of this is that since Selection causes non-determinism to arise within the Guarded Command Language, there's less ambiguity seeing how it comes about. Non-determinism is something that we want in the implementation language. It represents a very natural way to think about computation where one doesn't have to rewrite the problem to suit the determinism of architecture.

2.2 Advanced Goals

There are a few extensions that I would like to incorporate into the language. The first is function definitions. Function definitions don't really exist within the Guarded Command Language since the language to a degree just focuses on distinct problems at a time rather than including separate functions into the program. Function definitions let one mediate this issue because it is an issue. One is limited in their language if they can't use function definitions because it allows for an expression of richer programs. The benefit of function definitions however is that one obtains the ability to perform recursion for free upon its implementation. If it's possible to define functions then one obtains the ability to define functions in terms of themselves which is fundamental to numerous algorithms within Computer Science. Some algorithms are so inherently recursive and its important to account for this.

A slight deviation from the "Dijkstrian" principles set out in the Guarded Command Language is the inclusion of objects. The goal is to allow for the implemented language to be object based such that one can construct programs using structures which are more complex the Guarded Command Languages arrays/lists. It's by no means an attempt to formalise the idea of Object Oriented Programming to the Eindhoven Quantified Notation that Dijkstra uses but is instead a complementary aspect to add to the idea of correctness. There are many problems that exist within the Object Orientated Paradigm such as modelling inheritance or polymorphism which is something that I don't have to concern myself with if the language is object based rather than orientated.

Chapter 3: Background Work and Research

3.1 Language and the Role of Notation

There exists an issue within language, both natural and programming in how does one translate their ideas, their thoughts into speech or writing or what this language concerns itself with, code. Learning to program is much like learning a second language, a hypothesis claimed by the authors of the study *Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages*[2] which does make sense to a degree : both have a grammar and both can be used to represent ideas. Of course programming languages are more formal in their expressiveness than natural languages. Coming from mathematics, they can really only express ideas of the such which seems like a rather banal statement to make but it's important. In the study, it was found that there was a strong correlation between the characteristics of second language acquisition and the participants experience of learning Python. It's an interesting result because anecdotally, from the beginning of my studies until now, the culture surrounding being able to "understand" a programming language is predicated upon ones mathematical ability and yet it seems that mathematical ability is only part of the puzzle.

Through the many layers of abstraction from the early days of computing languages to now show that there is at least some truth to the hypothesis of the study. Python is one of the most popular languages today and it's easy to see why. It forgoes constructs like "==" and "!" for "is" and "not" which reduces the level of ambiguity of a program. The indentation of the language is a fascinating stylistic choice. It forces the programmer to write in a paragraph like style that does help in understanding. The body of a for-loop must be indented for example which allows one to quite clearly see where the loop ends. But why is this all important? All of these facets are based upon notation.

Notation is fundamental to language. Another banal statement perhaps, but the difference between good and bad notation is that a language like Java (for all its verbosity) has remained popular for twenty-five years whereas Ruby thankfully fell by the wayside. I believe that this was one of Dijkstras aims in his creation of the Guarded Command Language. I would put the notation of it into the good category because although there does exist a learning curve, it is powerful in its expressiveness and as we will see later, it possesses all of the hallmarks of good notation.

3.1.1 Natural Language Principles in the Guarded Command Language

In Walls piece *Natural Language Principles in Perl*[3], he gives examples of some linguistic principles he used in the creation of Perl, one of the most important being Topicalization.

A topicalizer is a word which introduces the subject of which one is speaking about. An example in Perl is the keyword *foreach*. Many languages use a keyword like this : for in the family of C languages for example. The primary topicalizer in the Guarded Command Language is the keyword *Do*. It introduces the subject of which the loop refers to. A single topicalizer could help the language quite a lot. If there exists many ways to do the same thing then there is redundancy. This is not something that should exist in a language because it's not economical to it. The notation should be small but rich enough to allow for one to express a large number of ideas.

3.1.2 The Relationship Between Syntax and Notation

The set of rules which governs what is valid is quite an important thing to achieve in the design of a language. The goal is to reduce ambiguity. To look at natural language for example, the English phrase "*Hello my friend, how are you?*" can be translated to French quite easily : "*Salut mon ami, ça va bien?*". Syntactically, meaning isn't really lost. If we were to translate this phrase to Mandarin however what we would find is that all meaning is lost for Western people as the Mandarin alphabet is completely different to ours. One cannot make any inferences regarding syntax since the representation of the language is unknown.

The relationship of syntax and notation are thus connected. Familiar notation breeds familiar syntax and this is certainly apparent between C and C++. It's definitely part of the reason why we don't program in any strange language because we're technologically at a stage wherein much of the syntax between programming languages is the same. There are a few languages which deviate from this syntactic consistency though such as Scheme.

Scheme is a powerful language but there do exist a few differences between its syntax and other languages. To perform an addition, one writes the operator addition and then two to many integers or floats that are to be added : (+ 1 2 3 4 5)

It is a good way to perform addition over a list of integers because there's less redundancy. One doesn't have type $1 + 2 + ...$ when it's clear what is being performed. However it's not actually as good as just writing operators as binary operations rather than as functions. Binary operations allow you to nest sub-expressions much more quickly and easily than in the alternative.

Consider the two identical expressions which are syntactically correct given their language

and perform the same operation but whose notations are different.

$$(\text{define x 1}) \equiv \text{x} := 1$$

The Guarded Command Languages notation of instantiating a number is much nicer as it isn't as superfluous as Schemes as a common complaint about Lisp like languages is the number of brackets one has write.

A language shouldn't add new concepts or syntax to help defining constructs which could be otherwise shown using preexisting constructs. It just makes the language unwieldy because less is in fact more. Part of the richness of the Guarded Command Language is the fact that there only exists five operations to it and the richness of its notations allows for one to construct some really nice programs.

3.1.3 The Notation of the Guarded Command Language and its Extensions

> "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race" - Whitehead

A quote which should have been on Dijkstras *A Discipline of Programming*. It was an aim to facilitate the ability to construct formally verified programs using the notation of the Guarded Command Language and the consequence of this is that many programs are just slight variations of the same program. The best example of this is the Partitioned Reduction Theorem from the beginning regarding finding the sum of all of the even numbers in an array.

3.1.4 The Characteristics of Good Notation

The majority of examples I have stated regarding the Guarded Command Language can be seen in the characteristics that Iverson stated in *Notation as a Tool of Thought*[4]:

- Ease of expressing constructs arising in problems
- Suggestivity
- Ability to subordinate detail
- Economy
- Amenability to formal proofs

The Guarded Command Language is both economical and permits the ease of expressing constructs as one can write two different programs by making small alterations since the language exploits the fact that many programs and algorithms are really just the same program. Suggestivity can be seen in the topicalization of the language but one could go a bit deeper in showing this characteristic.

Consider the following, if one wanted to see if the integer two existed in a list of *n* integers then what type of program would one write to solve this problem? Say now we're tasked with finding if a French person, student or staff exists within the School of Computer Science. How would one write a program to solve this problem? Well its the same problem! They're both examples of a Bounded Linear Search. Much like the Reduction examples, the wording of the problem in natural languages gives some guidance in what program one needs to ultimately write.

The third characteristic is a little more tricky to show. What I think will help in this task is looking to the last characteristic for help. Proof methods are foundational to the construction of a program. The benefit of this is that when one carries out a program construction in the Guarded Command Language, the program itself is almost an unwanted guest to the proof of it. The proof however needs to be minimal in its construction otherwise its not a very good proof. To construct a program, one needs to be able to abstract the necessary details of the problem to permit the program construction.

Notation facilitates the goal of constructing a correct program which is ultimately the desired outcome of writing any program. Many other languages arguably have easier notation to understand such as Python but simplicity can hinder expressiveness at times. It does take more time to familiarize oneself with the Guarded Command Language but its rich notation and constructs allows one to write programs which are simply much more beautiful but importantly more powerful. The Eindhoven Quantified Notation was created with the purpose of exploiting mathematics and logic which is why it is the gold standard when one speaks about correctness of programs.

3.2 Non-Determinism

3.2.1 Introduction

Non-determinism is a really interesting concept which arises in the Theory of Computation and in the Guarded Command Language. It isn't really spoken about too much in general because it's quite unimportant in society today. Modern technology is limited in its expressiveness compared to the Guarded Command Language. For example, an integer x in C can only exist within the range of $-2147483647 \leq x \leq 2147483647$[5] whereas the Guarded Command Language on the other hand operates upon the set of all integers. It

seems like an unimportant difference but the lack of expression regarding modern machines forces them to be deterministic rather than non-deterministic. But why is non-determinism useful? Well it allows for some very nice solutions that would otherwise be more of a headache to implement in a deterministic way. Consider the following problem, *"Alex has 75 red tokens and 75 blue tokens. There is a booth where Alex can give two red tokens and receive in return a silver token and a blue token, and another booth where Alex can give three blue tokens and receive in return a silver token and a red token. Alex continues to exchange tokens until no more exchanges are possible. How many silver tokens will Alex have at the end?"*

In this problem, non-determinism arises as it doesn't matter which token is exchanged, all that matters is that there exists a solution.

Listing 3.1: Pseudo Guarded Command Language example of Nondeterminism

```
red, blue, silver := 75, 75, 0
;Do (red != 0 v blue != 0) ->
    red, blue, silver := (red - 2, blue + 1, silver + 1)
                       v (red + 1, blue - 3, silver + 1)

 Od
```

All that matters in a non-deterministic program is that there exists a solution. If there doesn't exist a solution then it's not a particularly worthwhile program. In the problem above, at some point Alex will have either no more red tokens or no more blue tokens to exchange. The non-determinism arises because it doesn't matter if the program ends when there's no more red tokens or no more blue tokens. This may seem simple but it's extremely important. The goal of the problem is to obtain a number of silver tokens not an analysis of how many red or blue tokens we have at the end along with the silver tokens. Exchanging any colour is valid so why would we force ourselves to rewrite the problem when non-determinism offers such a nice and clean solution? This can be illustrate in another problem.

Consider the following program which finds a value y such that $1 \leq y \leq n$ where n is the integer ten. Forcing sequencing of this problem is a ghastly error in judgement because if the purpose of the program is find this value of y, does it particularly matter what value it is? If a value is forced to be considered then the algorithm itself changes because the algorithm is supposed to find a value in the range between 1 and n, not a specific value in of itself.

Listing 3.2: Finding a y between 1 and n

```
x, y, n := 1, 1, 10

;Do (x != n)->

        if x mod 2 <= y mod 2 -> x := x + 1
```

```
            [] x mod 2 >= y mod 2 -> y, x := x, x + 1
            fi
    Od
```

3.2.2 Guarded Command Language and Probabilistic Programming

There exist two approaches to non-determinism. One approach is using Probabilistic Programming to reach the end state through different paths. The other approach, the one taken by Dijkstra in his creation of the Guarded Command Language is to consider a single correct answer being reached from many to one possible solutions. Probabilistic Programming (PP) is concerned with the former approach but they're both equally valid views of what "non-determinism" constitutes (though this seems more like a Wittgensteinian language game). What does it precisely mean if there exists two equally valid approaches? I aim to discuss the similarities and differences between these two because they are worth mentioning and it should offer an explanation as to why I'm taking Dijkstras approach more than just the reason of that's how it was implemented within the Guarded Command Language already.

Similarities

The main similarity between the two is this thing called randomness. Non-determinism appears to be random and Probabilistic Programming allows us to model what appears to be random. Going further into the similarities between the two, in the Guarded Command Language, the branch that is taken is picked at random from *n* possible branches and in PP, methods like inference can be used to make a choice that success is probable which in effect is random because probable doesn't equate to certain.

Differences

Probabilistic Programming draws values from random from probability distributions[6]. This is an important distinction to make because within the Guarded Command Language, a Selection branch isn't taken based on the probability that it will be taken. Probability doesn't come into the implementation at all. Instead, a solution is arrived at from one to many different branches whose guards were evaluated to True. One is chosen at random from a list of possible branches but one branch isn't more likely to be chosen that another. This directly opposes how a branch is taken regarding probabilistic methods.

Probabilistic Programming provides statistical modelling. *"The goal in PP is analysis, not execution."*[7]. This is different in the case of the Guarded Command Language. We're not concerned with the analysis of the current state of the execution, all we're concerned with is the output of the program. The program is a figurative black box. We don't know

what goes on inside it and to a degree, we don't actually care. All we care is that a result that is produced.

For the purpose of this project, the type of non-determinism which will be considered is Dijkstras interpretation. His Guarded Command Language is a bounded non-deterministic system. The bounded non-determinism is actually quite helpful to reason about the implementation of a program as a whole.

In an arbitrary program, the initial guard breaks the loop once it is evaluated to False. This is very important because it means that there is a well defined starting state and a well defined finishing state. These states are predicated upon the guard, not the body of the loop. This is great because it goes back to the black box example from before. We know that if one places an input into this box then it will give one an output. What's in the box? Well, what occurs inside it is dependent on the current state of the program at that instance because each new state may give rise to different branches that could be taken. Non-determinism arises here but since the number possible branches are finite and since the guard must be broken at some point, then the program must end[8]. If the program must end then a lot of interesting approaches open up in implementing non-determinism. This approach allows us to treat the program more like how Dijkstra intended rather than exploiting probability to find the quickest solution.

3.2.3 Implementing Non-Determinism

McCarthy initially proposed the operator *amb* in his paper *A Basis For a Mathematical Theory Of Computation*[9] for Lisp. He proposed the operator as a way to describe non-determinism. The operator takes two or more expressions and yields a number of "futures" and the future that leads to a successful computation is the one which is chosen. Non-determinism can be seen in Haskell in the following way:

Listing 3.3: Non-determinism Using Lists

```
concat [[a, b] | a <- [1, 2], b <- [1, 2], a < b]
```

It doesn't do much good to consider each Haskell case as its own special implementation of non-determinism, *amb* or something like it needs to be implemented within Haskell to permit non-determinism within the proposed language.

Suppose that in a list of possible options, the first one that is possible to be evaluated is the branch that is taken. If then we go through this branch until the code is executed successfully then we would have a valid program but if it reaches an impasse then it won't be able to finish executing. This is an obvious problem. One option is to use backtracking to solve this. If an impasse is reached, then we could just go back to the point in which this branch was taken or where there didn't exist an impasse. This would be very similar to Lisps

amb. In fact, we could go further and implement *amb* using the following implementation from the official Haskell website[10] if we wanted IO but we may not really need it therefore its quite possible to simply use the List Monad as a way to implement non-determinism.

Listing 3.4: Haskell and Amb

```
example :: Amb r (Integer, Integer)
example = do x <- amb [1,2,3]
             y <- amb [4,5,6]
             if x*y == 8
             then return (x,y)
             else amb []
```

But if *amb* was removed, we would achieve the same result.

Listing 3.5: Haskell Nondeterminism Using The List Monad

```
example = do x <- [1,2,3]
             y <- [4,5,6]
             if x * y == 8
             then return (x,y)
             else []
```

That's not a very interesting example however. Perhaps applying this to find two numbers whose sum is prime would be more interesting[1].

Listing 3.6: Two Integers Whose Sum Is Prime

```
factors :: Integer -> [Integer]
factors n = [ x | x <- [1..n], n `mod` x == 0 ]

prime :: Integer -> Bool
prime n = factors n == [1, n]

findPrimes = do
        x <- [1..]
        y <- [1..]
        if prime (x + y) && prime x && prime y
           then return (x + y)
           else []
```

The approach of using the List Monad could be the best way in implementing non-determinism. Since non-determinism only arises in the instances that Selection is used within a program, then we could consider the following as a pseudo-way to implement non-determinism:

[1] Using Haskells lazy evaluation, we would use the function *take* to obtain *n* primes

```
f = do
    x <- [branch_0 .. branch_n]
    if canBranch? x
       then branchOff x
       else return ERROR
```

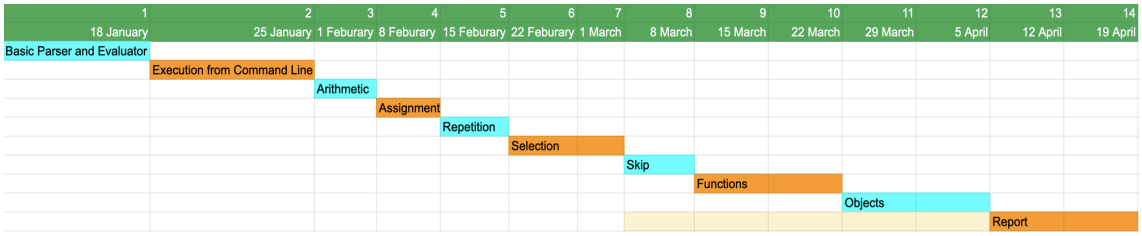
There is also the option of setting up the structure of a program like a graph. We could start with a single execution thread from the starting state. The starting state would be the guard of the loop. It makes sense to set it as the starting state as it would be the easiest way to break the loop as this state leads into the non-determinism rather than being a part of it. After this initial thread is created, whenever one or more branches are possible to take then an execution thread is created for each option. These are stored in a queue and are iterated upon in a Round Robin way until the guard breaks.

Or we could turn to pre-existing Haskell packages to see if they're of any use such as *Control.Monad.Logic*[11], adapted from the paper *Backtracking, Interleaving, and Terminating Monad Transformers*[12] or *Control.Effect.NonDet*[13].

I think the best course of action to implement non-determinism would be to exploit the fact that it is essentially the List Monad. The advantage of this is that it's part of the Haskell environment. Often the case that arises when using peoples libraries is that they may be deprecated or in some cases not portable. Moreover debugging will be easier if the non-determinism is implemented through the List Monad and not someone else code as the former is held to a higher standard.

Chapter 4: **Implementation Plan**

My implementation plan is quite simple. Since there only exists five operations to the Guarded Command Language and since I'm only extending it to include objects, function definitions, and print functionality, I can breakdown the implementation into the following list and its Gantt Chart representation.



1. Implement a Basic Parser and Evaluator
2. Implement Ability to Execute Program from Command Line
3. Implement Arithmetic
4. Implement Assignment
5. Implement Repetition
6. Implement Selection and its Non-Determinism
7. Implement Skip
8. Implement Function Definitions and Print Functionality
9. Implement Objects
10. Write Report

My aim is to implement the basic features as soon as possible to allow me to dedicate more time to the more complicated aspects of the language such as making it object based and non-determinism. I allocated a week for numerous features but this is just due to having to do other modules as well. A basic parser shouldn't be too complicated for example but with the time I allocated for each feature, between a week to two weeks, I am confident that I should be able to complete the project implementation by the end of the semester on the 22nd of April 2021. Since the language is small, and the features are distinct, I will be able to mark my progress by being able to use the features within a program as they are implemented. The more features that are implemented the more complex programs I will be able to write. By the time I implement the extensions, I hope to be able to construct a

program that finds the longest common subsequence of two arrays because that has been the most complicated Guarded Command Language problem I have come across to date. Regarding my extensions, I will be able to verify the veracity of the function definitions by being able to perform recursion because that is the most complicated use of function definitions. Implementing objects will be a little more tricky but since I am not modelling object orientated behaviour and instead effectively just creating a linear data structure that allows for one to instantiate what is essentially just instantiations of sets and call them, then they will be verified as the language will only be able to concern itself with linear data structures.

Regarding my choice of implementation language however, I have made the decision of choosing Haskell. Since speed isn't really a concern for me, I can forgo the difficulty of writing a tremendous amount of code that I don't have to otherwise when using the efficient *Text.Parsec*[14] library in Haskell. Furthermore part of the motivation in choosing Haskell is that its also enjoyable programming in it. It's not so much a slog like in other languages and the community is the most enthusiastic one I have encountered which helps when I will be focusing on this project for a long time. And perhaps being functional is in the spirit of the proof methods used to construct a Guarded Command Language program!

However there does exists a Guarded Command library[15] within Haskell. My project in particular differs from this in that this library acts much like an API which is complementary to Haskell than its own language. What I am proposing is a language which uses Haskell internally but externally, one writes their program how they would have written a Guarded Command Language program on paper. Furthermore this library at the time of writing is only mostly portable and its stability is experimental. The language I am proposing will be entirely portable and stable as I plan to use third party libraries like *Text.Parsec* minimally.

Chapter 5: **Implementation**

Before carrying out this project, I had some experience in language design working with Tang's *Write Yourself a Scheme in 48 Hours*[16]. It was a beneficial project to take on because coming to an implementation of my own language, much of the lessons and general structure of a Haskell project that I learned afforded me a clear outline of how to build a language.

Normally in language design one would construct a Lexer then a Parser and then build up an evaluation of the parsed program. Instead I intertwined the Lexer and Parser as the Text.Parsec[14] library more or less facilitates an interconnection between the two.

5.1 Language Representation

How data such as integers and Booleans and then more complicated constructs like loops were to be represented within the program was something that had to be thought about before beginning the project. It's relatively simple to represent data like how you would represent objects in Object Orientated languages with classes in Haskell. It's simply a case of naming your type and having it equate one of Haskells predefined types.[17]. For instance, regarding the use of integers they were defined in the following way:

Listing 5.1: Integer Representation

```
data HVal
  = HInteger Integer
```

Each data type is coupled with the letter 'H' to reduce confusion with Haskell's data of the same type. At the moment, the only primitive data types that are used within the language are integers, Booleans, and strings to start as implementation of further data types such as floats can be done at a later time. The only data structure with the Guarded Command Language is lists/arrays therefore it was only data structure regarded for the language.

At the point of evaluating the code, a problem arose regarding the evaluation of statements such as Repetition wherein a return value had to be given if it was successful. The solution to this was to consider the differences between values and statements/expressions. An integer has a definite value but a statement doesn't have one. Repetition and Selection don't have any values associated with them because their function is to allow for one to manipulate data rather than being associated with an inherent value. This meant that

there was a necessity to split the data from the statements/expressions to have HVals and HStatements. The necessity of this will become apparent in the evaluation section.

5.2 Parsing

Parsing code was reserved to a file : Parser.hs

5.2.1 Main

Main, the program that reads and runs the programs went through a few versions. In the beginning when everything was quite simple and there was no functionality really, all that the Main function would do would be to read a string from command line arguments, parse it and return what value it is i.e inputting the number 4 would return HInteger 4. HStatements were the only real problem regarding evaluation.

In order to evaluate a HStatement, there were two different approaches that were available:

1. Constructing a Read-Eval-Print-Loop (REPL).
2. Reading from a file.

During the summer of last year when I first began to explore language development, I took the REPL approach but the problem with this approach is that it's really just tedious to implement and then to debug programs because one would have to write out the program each time. Instead, programs are now read from a file.

Reading from a file proved to be more difficult than expected due to Haskells type system. Due to it being a pure language and so IO actions like reading from a file are impure, Haskell bounds the type of variable that is returned from an IO action to IO.

For example the function *getLine* returns IO String and in the case of reading a collection of HStatements / a program from a file returns IO [HStatement]. This wouldn't seem like much of an issue but any variable bound to IO can't interact with functions of the type of the variable. My first approach was to alter the Parser and Evaluation entirely to account for IO but this proved to be too difficult. The solution was to exploit Monads.

There's a nice notation within Haskell called do-notation[18]. It affords a somewhat imperative approach to writing monadic code yet it still retains the functional purity of Haskell. This imperative-esque style was quite useful in both reasoning about the code and debugging later on. Within the do-notation in Haskell, assignment can be done with operator $\leftarrow$. What this operator allows one to do is to essentially unwrap that which is bound thus IO can be completely ignored and the program can be parsed properly.

5.2.2 HVal

It's impossible to take any other approach to the design of a language but a bottom up approach if complex behaviour such as Repetition or Selection is desired.

The Text.Parsec library[14] facilitates parsing in a straightforward way. For instance, it supports the use of Monads which saw a tremendous amount of use in the project overall. In simpler instances in parsing (and evaluation), Haskell's bind operator '»='[18]allowed a "binding" of the result of parsing "many digit" or "many letter" (both functions which exist in the parsing library) to return a HInteger or HString.

In the parsing of Booleans, the use of Haskells fmap[19] was needed. It's quite an interesting and useful function. It's somewhat similar to map but can be used to map function of a certain type over an array or list of the same type that is bound to another type. For instance to map a function *f* to a variable *x* of type *g* that actually prohibits function *f*, what *fmap* allows you to do is the following:

Listing 5.2: fmap

```
fmap f (g x) = g (f x)
```

So in the case of Booleans which are inputted as strings, fmap allowed me to map the function *classifyBool* to the input. This then allowed me to take the input and pass it to the classification function and return a HBool rather than have it return a HString. The notation used was the operator $< \$ >$ which is analogous to fmap. A similar strategy was used in the parsing of mathematical operators.

Using the do-notation, reasoning about parsing expressions such as arithmetic wasn't as difficult as it first appeared before beginning the project. For example, using Text.Parsecs spaces function with the do-notation, reasoning about an expression such as "2 + 2" became essentially a task of parsing each character in sequence i.e

Listing 5.3: Basic Arithmetic Parser

```
parseArithmeticExpression = do
                            var_1    <- parseInteger
                            spaces
                            operator <- parseOp
                            spaces
                            var_2    <- parseInteger
```

One thing that had to be considered when implementing the arithmetic parser however was the order of precedence. One possibility was implementing a Pratt Parser[20] which introduces the order of precedence but I instead opted for something arguably a bit more simple. The benefit of developing in Haskell is enormous as it's easy to exploit recursion. It was extremely easy to force the order of precedence with recursion by allowing the second

variable in the arithmetic expression to be it's own arithmetic expression which begins and ends with a bracket such that precedence is forced by definition of the arithmetic parser. More complex expressions could now be evaluated in the following way:

- 1 * (2 + 3)
- (1 + 1) * (2 + 2)
- (1 + (2 div 1)) mod (3 + 4)

It is important to note however that at the time of writing, expressions like $1 + 2 * 3$ by themselves won't evaluate.

5.2.3 HStatement

Eval

Since what is really being evaluated in the program is a collection of statements, a function was needed to bridge the gap between HVals and HStatements. This function was just a call to a general HVal parser.

Repetition

Repetition was interesting to implement because representation is an important facet to these features in the Guarded Command Language. The structure of a loop can inform of what its representation should be. Each loop is bounded by a guarded expression which is then followed by one to many statements. This can be represented by a HVal expression since the HVal type contains Booleans and arithmetic expressions which can evaluate to Boolean values. If it's followed by one to many statements then the natural data structure to represent these varying amount of statements is to use a list consisting of HStatements.

The challenge now was to figure out how to parse many statements. My first thought was to figure out a derivation of some recurrence relation but the solution was much more simple. The parsing library's *many* and *many1* functions allows for one to parse many statements in one go. This could have only worked however by ending the loop in parser with the expectation of parsing the string "Od" otherwise the many parser would have parsed for an hyperbolic eternity.

Listing 5.4: Repetiton Parser

```
parseDo :: Parser HStatement
parseDo = do
    _ <- string "Do"
    spaces
    _ <- string "("
```

```
p <- parseArith
_ <- string ")->"
spaces
q <- many1 ( spaces *> parseStatements )
_ <- string "Od"
return
```

Selection

Selection was the most difficult statement to implement. It went through many different iterations, the first being to have a single Selection parser then sub-Selection parsers if there was a requirement for multiple selections in the program. That did work but when working on the evaluation of statements, the program became quite long and frankly confusing as it would only evaluate properly part of the time.

It took a while to come up with an appropriate representation. Eventually I recognised that a single selection is a constrained form of a loop. Both are bounded to a guarded expression. This guarded expression when True allows for the evaluation of one to many statements. Once these statements are evaluated, the program exits from the Selection block. This behaviour is very much like a loop. This realisation essentially means the parsing and evaluation code Selection would be similar to Repetition.

5.3 Evaluation

5.3.1 HVal Evaluation

Evaluation of the primitive types was relatively simple. It's just a case of returning the value bounded to the primitive type.

Listing 5.5: Evaluating An Integer

```
evalVal env (HInteger val) = return (val)
```

Going back to what constitutes an arithmetic expression, two variables and a binary operator, what is the best way to evaluate it? It was laborious but the initial step was to create a function which does exactly that. I made an expression applicable for each operator and each type of HVal that is being operated upon. Once that was finished the execution of binary arithmetic expressions was really simple.

What about more complicated expressions such as $1 + (2 * 3)$. It has to be evaluated in accordance to the bracket distribution i.e $(2 * 3)$ is evaluated first and then this is added to 1. There were two ways that approach this problem which came to mind. The first was

to place an arithmetic evaluation statement inside the second variable. This did work but it wasn't clean and was more ambiguous to the solution taken, binding the evaluation to a lambda function.

Consider an arithmetic evaluation function *f*, evaluating the most complicated section first and binding this to a lambda function leads to a nice function that can be applied in many different cases.

Listing 5.6: Nested Arithmetic Evaluation Using Lambda

```
  f var1 op (arithmetic expr) =
f (arithmetic expr) >>= \var2 -> f var1 op var2
```

Lambdas offer the ability to use the monadic do-notation which was quite useful in the evaluation of some of the HStatement's and the project overall. One difficulty that I faced before the evaluation of these statements though was the ability to define variables that could be used in arithmetic and other expressions.

Assignment

Write Yourself a Scheme in 48 Hours[16] book helped tremendously with this section. The best way to describe the process in this section is to consider conceptually how a variable is defined.

Consider the Guarded Command Language expression $x := 1$. Since the goal of the language is to create some sort of persistence in which a variable can be defined at the beginning and altered throughout the program, a context in which to frame the assignment is needed. To achieve this, an environment is created which takes advantage of the Data.IORef library which allows for one to use state. The next thing to consider is what could occur in an assignment. For example an expression might want to add an integer or another variable to the current instance of the variable. Other than that it's simply a case of assigning a value to the variable.

So in essence this could be expressed in the following pseudo code

Listing 5.7: Pseudo-Assignment Check

```
if alreadyDefined(var)
    then setValue(var, val)
    else
         writeValue(var, val)
```

Then to evaluate the variable in which the value is stored in, a strategy like the nested arithmetic function was employed. If the string is evaluated in the following way:

Listing 5.8: Evaluation Of Variables

```
f env x op var = getVar env var >>= \val -> f env x op val
```

then all that is required is to check whether or not the string is bound to a value and pass to a lambda function to be evaluated using the bind operator.

With that issue solved, the next step is evaluating each HStatements.

5.3.2 HStatement Evaluation

Evaluation code was reserved to a file : Expr.hs

As stated before, the difference between HVals and HStatements is the fact that HVals have a value associated with them whereas HStatements don't have one by default because they're just manipulating data which is an important difference. It means that the return type has to actually not return anything but still cause an effect to occur throughout the evaluation i.e the evaluation of a loop shouldn't return a success value but instead return nothing and just cause the manipulation of data within its body. In more "popular" languages, the type that comes to mind is *void* and in Haskell the equivalent is the Unit type denoted by `()`. [21]

Now if the expression to evaluate was for example the statement 'Eval val', what occurs is yet another use of the do notation wherein the value is calculated and then the Unit type is returned. The Unit type allows for the consideration of the side effect to the function rather than the return type meaning that a consistency of return types can be kept across all of the HStatements.

5.3.3 Skip

If the only thing that constitutes Skip is that it does nothing, well the type that does nothing is the aforementioned Unit type. Returning the Unit type in every occurrence of Skip achieves the behaviour of it because whenever Skip is encountered, it shouldn't evaluate any value. It's evaluation is the evaluation of nothing thus the only possible type it can concern itself with is the Unit type.

5.4 Main, Repetition, Selection

The most surprising aspect to the implementation of the project was that the major issues with Main, and Repetition and Selection were all solved with more or the less the same

solution. All of the problems were solved with the function traverse_[22]. This function is similar to map in that it operates across a list but differs in that it composes the calculations into a single structure and forgoes returning the result. The use of this will come apparent in describing how it solved the problems within Main, Repetition, and Selection.

5.4.1 Main

When it came to evaluate a program, an issue that kept coming up again and again was that only a single line would evaluate. This then was solved by using traverse (the version which doesn't ignore the results) which allowed me to map the evaluation function across my program input. This traverse function that was used returns the results of these computation rather than ignoring them because in this case, the result is needed otherwise the program couldn't begin.

5.4.2 Repetition

The difficulty in evaluating a loop was figuring out how to precisely evaluate it. The first approach was an if-else statement. If the guard was True then continue to evaluate the loop otherwise return the Unit type. In the True branch do-notation was employed to evaluate the current instance of the guard then map was used to apply the HStatement evaluation function to the list containing the various HStatements. This didn't work. One problem which occurred was that the guard was evaluated one more time than it was supposed to. For example if the sum of the integers between one and ten inclusive were being evaluated, it would evaluate the body one extra time causing the sum to be between one and eleven inclusive. The other problem is that map constructs a list of evaluations which meant that at times, a loop would only update some of the values rather then composing all of the evaluations into one computation.

The solution to the first problem involved reasoning more about what is occurring. Upon each evaluation of the guard, what is actually being evaluated is the current instance of the guard but this comes from it's evaluation from within the body of the loop, not the evaluation of the guard when it's being checked. If the evaluation of the guard is what is being returned on each subsequent check after the first one then of course an infinite loop occurs because the Boolean True becomes the new guard, not what's supposed to be the guard.

Regarding the second problem, what is actually needed is to have the effect of map but compose the evaluations into a single one. The traversable_allows for a mapping of the evaluation function to each statement in such a way that it acts like a loop in of itself. The benefit of this is that when the traversable_function is ran over the list, what happens is that each entry in the list of statements becomes updated so all that is needed to continue the loop is to return the guard and list without any alteration.

5.4.3 Selection, Determinism, and Nondeterminism

The Unfortunate Association with If

Selection is not synonymous with If statements. Although they both present themselves as control flow, Selection works in such a way that all possible branches that could be taken in a block are taken and produce their own instances of the program and the branch that finishes first is taken as the result to the program. If on the other hand only takes the first branch which can be taken, evaluates the body of that branch, and exits the If-block. The distinction has to be made before I discuss the evaluation implementation of If because Dijkstra confusingly uses the strings "if" and "fi" to denote a Selection block but these aren't the same thing.

Implementation of Selection

This was the most difficult aspect of the language to implement thus far. It's initial approach was much like the initial approach to Repetition therefore all of the problems with Repetition became Selections. Since Selection is a constrained form of Repetition, its evaluation function mimicked Repetitions. However an amendment was made. Since Selection is represented as a collection of individual statements, two functions were needed in order to implement Selection in its entirety.

An implementation of an individual statement was quite simple given that Repetition was completed first. The only difference between the execution of a Selection statement and a loop is that the evaluation of the statement isn't called like the loop again but instead the body is "traversed" if the guard to it is True otherwise the Unit type is returned.

To an extent, the evaluation of the Selection block is similar to the evaluation of a loop in that it's traversing over the block. Whether or not a particular statement within the block should be executed is irrelevant because that is taken care off by the function which evaluates a single Selection statement.

But this didn't work exactly as intended. Evaluation of a program terminated whenever a guarded expression in a block was False even if the guard of the statement after was True. This occurred because the guarded expressions and their consequents weren't intrinsically bounded together. The optimal way to bound was placing them within a tuple and then everything worked properly.

When the guards and their statements were placed in a tuple and were encountered by the traverse_function, they would be strictly bounded together such that if the guard of one statement was False, it wouldn't just terminate the Selection block but instead just go to the next Selection statement within the block.

Evaluating Selection in this way was pertinent to the fact it is based upon the Guarded Command Language rather than traditional if-blocks. Within traditional languages, if-

blocks terminate once a single statement within the block is True and its body is evaluated.

This isn't fantastic behaviour to have because it forces longer sequencing based on the languages syntax rather than in the case of the Guarded Command Language. In this case, upon the True evaluation of each guard in the block, a new thread is created with its own instance of the program and the thread which finishes first at the very end of the program is taken as the result of the program.

The use of this might not be entirely obvious. In some scenarios it is a more natural way to think about computation such in the case of Alex and the tokens from the beginning of this paper. Non-determinism is useful in the case of exchanging tokens and then checking how many silver tokens are left due to the nature of the problem itself. The problem doesn't take into account how many red or blue tokens Alex has at a given time, all that it concerns itself with is that there exists a number of silver tokens. This is extremely important because if we were bound by determinism, we would be forced to consider the number of red and blue tokens. So generally, a use case of this non-deterministic behaviour aids us when the inputs to the problem aren't important to the output of the problem.

However, this implementation is neither a deterministic implementation nor a non-deterministic implementation.

The behaviour of the traverse_function means that the Selection block and all True blocks are calculated. This causes a problem with the non-deterministic approach as one branch can cause a later branches guard to evaluate to True but only one branch can be taken at a time and it also couldn't be considered for an If implementation because all of the True branches are taken.

To implement If, I changed Selections syntax to use 'select..end' instead and I defined a structure much like Selection denoted by 'if..fi'. They are parsed similarly but are evaluated much differently. Selection and its non-determinism will be spoken about in the next section.

Consider a list containing If-statements xs. This list can be broken into its head and tail by considering it as x : xs. The benefit of this is that since If works by evaluating the first branch whose guard is True then exits the If-block, then if the guard of x is True, its body can be evaluated and the return type can be returned otherwise the head of the tail and the tail of the tail can be called until a True branch is reached.

Non-Determinism

The problem with Dijkstras version of non-determinism is that although it is quite natural and freeing, it is also impractical on modern machines. Although Königs Lemma[8] shows that if a program is represented as a tree and has a finite number of branches, it still doesn't absolve the problem of approaching infinity or resources running out during the programs execution. This type of unbounded non-determinism is something that even Dijkstra himself argued that it would be impossible to implement which lead Tony Hoare to the conclusion

that an idea of fairness must be introduced to have any feasible hope of implementing non-determinism.

The unbounded problem arose during the first attempt at implementing non-determinism. Since Dijkstras non-determinism operates on the idea that each Selection statement spawns a new instance of the program and the spawned program that causes the Repetition guard to break is the result which is taken, then the idea that should come to mind is multithreading.

Multithreading can be achieved quite easily with Haskells Async library[23]. In fact there's an interesting function within this library that has the aforementioned behaviour of taking the result of the function that execute first : race. This function takes two functions bounded to IO and returns the result that is executed first and discards the other. Two issues arose from using this library however. The first was that I was unable to detach the function from IO. The curse and blessing of Haskell is that although correct code is incredibly fast and efficient, the type system does cause development to be slow at times. The second issues was the realisation that perhaps the unbounded non-determinism might not have been able to be achieved at all with the way I was trying to implement it.

Suppose race was used to achieve non-determinism. Amending it then to Selection, it would take n Selection statements and evaluate them all until the first one finishes and takes that. If n threads are ran concurrently however for a loop of m iterations for example, then the process becomes extremely resource intensive as $n * m$ threads are started but not necessarily finished given a program. Disregarding the fact that race only works on two IO actions at a time, the amount of threads that are started can become exponentially large.

Hoares idea of fairness was helpful in deriving another solution. If the result doesn't matter but the existence of a result does, then it doesn't really matter how the result was obtained through Dijkstrias non-determinism or another kind of non-determinism. For each Selection block, evaluate a random statement whose guard is True and continue the program otherwise generate a new random statement until a statement with a True guard is reached. This approach still retains the idea that the result doesn't matter just that there exists one but achieves this in less resource intensive manner.

Chapter 6: Modus Operandi

On a high level, the language works in the following way. The name of a program is read in through command line arguments. The file is then opened and parsed. The Either Monad returns an error or returns a parsed program in terms of the language syntax. This is then sent to the evaluation function which traverses the parsed program, evaluating it then concatenating the evaluation up into a single item which is then returned.
The rest of this chapter will detail how the program is parsed and evaluated in more depth. To showcase how this works, I will use the following program:

Listing 6.1: Modus Operandi Code

```
x := 5
y := 10
z := x + (y * x)

print(z)
```

6.1 Parsing a program

Main is what handles the general logic of parsing and evaluating. The first command line argument is used as the parameter that holds the name of the file to be ran because as of time of writing, it is only possible to execute one program at a time. An environment is created in which to frame the execution of the program and the function which parses the program is called. This function takes the result of parsing the program and evaluates it.

The program above then is read first as a string and is placed in a variable called 'program'. This variable 'program' is then parsed according to a function parseProgram which parses one to many statements.

Listing 6.2: HStatement Parser

```
parseStatements :: Parser HStatement
parseStatements = try (parsePrint) <|> try (parseEvalHVal) <|>
                  try (parseDo)    <|> try (parseSelection)<|>
                  try (parseIfs)   <|> try (parseSkip)

parseProgram :: Parser [HStatement]
parseProgram = spaces *> many (parseStatements <* spaces)
```

The best way I found to reason about parsing was to consider how each parser is tried until it finds one which is successful. Take the first line which assigns the integer five to the

variable x. It will first try parsePrint but it cannot do this so it will move to parseEvalHVal. Since this function bridges the gap between HVals and HStatement, this parser begins to parse the assignment of x.

Listing 6.3: EvalHVal Parsing

```
p = try (parseAssign) <|> try (parseArith)

parseEvalHVal :: Parser HStatement
parseEvalHVal = do
x <- try p
spaces
return
```

The parseEvalHVal function begins parsing and uses a parser p to give either a parsed Assignment or a parsed Arithmetic expression. This result, in the case of the assignment and Assignment in general, is returned in the form of : Eval (Assign var val). The same logic is repeated for the variable y. The assignment of z however is slightly different. It is assigned an arithmetic expression.

In the case of z, the parser p parses an assignment and in parseAssign, a variable can be assigned a value given an arithmetic parser and the general HVal parser. Let's consider the arithmetic expression being parsed for the assignment coming from the HVal parser rather than its own parser to explore how the language works further.

Listing 6.4: HVal Parser

```
parseVals :: Parser HVal -- General parser for HVals
parseVals = try (parseValString) <|> try (parseCons)
          <|> try (parseCar)   <|> try (parseCdr)
          <|> try (parseListAccess) <|> try (parseLength)
          <|> try (parseAssign) <|> try (parseArith)
          <|> try (parseList) <|> try (parseBool)
          <|> try (parseString) <|> try (parseInteger)
```

The parseVals function then goes through each parser and tries each one until it finds one which is successful, in this case parseArith. With three assignments parsed correctly, the last thing which is parsed is the print statement. The HStatement parser begins again and tries first parser, parsePrint and parses the line successfully.

What is produced from this are four language parsed expressions : assigning five and ten to x and y, an arithmetic expression to z, and then to print the value of z. These parsed expressions are then fed back into the function which evaluates the program file.

6.2 Evaluating a program

Evaluation works similar to parsing. The general evaluation function, evalStatement_, consists of the operations of the language (Repetition, Selection, print, etc). Where the parsing is an exercise of trying each parser until succession, the evaluation function takes a parsed expression and applies the correct evaluation to it given its form.

The first parsed expression is fed into the evaluation statement function and is matched given it's structure, Eval (Assign var val). It evaluates the HVal expression and returns the Unit type. When it evaluates the expression, it is passed to the evaluation function of HVal values/expressions evalVal and matches the syntax of the expression with one of its possible values/expressions. 'Assign var val' is matched appropriately. The value that is being assigned is evaluated by the same function evalVal and is binded to the variable by use of the Monad operator '»=' because the value isn't necessarily a single value. The variable is then defined in terms of the value. This is repeated for the y assignment.

Regarding the z assignment, it is passed to evalVal again but it's value is evaluated given the function evalArithmetic. Within this function, it matches the first token in the expression with HString x as x is a string. The operator is matched with a placeholder called 'op'. The next token is matched as it's own expression such that evaluation works in the following way:

Listing 6.5: Evaluating Arithmetic

```
evalArithmetic env (HString  x) op   (Arith x' op' y') =
            evalArithmetic env x' op' y' >>=
                \y -> getVar env x >>=
                    \var -> evalArithmetic env var op y
```

The most complicated part is evaluated first and is binded to the the expression which obtains the value from the variable which is then binded to a call to evaluate the variable and the nested arithmetic expression in context of the operator that is matched in the final call. The parsed expressions have now been traversed, evaluated and concatenated together to compose the evaluations into a single structure which can be returned correctly to produce the correct output, fifty-five.

```
~/Desktop/fyp_branch/Olivia  master  cat Programs/Modus_Operandi.lsy
x := 5
y := 10
z := x + (y * x)

print(z)
~/Desktop/fyp_branch/Olivia  master  ./Olivia Programs/Modus_Operandi.lsy
55
```

6.3 Writing A Program

Assignment should be written in the following way : X := Y

Skip should be written in the following way : skip

Repetition should be written in the following way :

```
Do (P)–>
        Body
Od
```

Selection should be written in the following way :

```
select (S)–> R
       (P)–> Q
end
```

If should be written in the following way :

```
if     (S)–> R
       (P)–> Q
fi
```

Arithmetic expressions must be nested. For example to see if four modulo two is zero, $select((4\%\ 2) = 0) - > \ldots$. The following are supported operators:

Operator	Name	Use Case
up	Up	5 up 4 = 5
down	Down	5 down 4 = 4
and	And	True and False = False
or	Or	True or False = True
+	Addition	1 + 1 = 2
-	Subtraction	1 - 1 = 0
/	Division	4/2 = 2
%	Modulo	4 % 2 = 0
<	Less Than	5 < 10 = True
>	Greater Than	5 > 10 = False
leq	Less Than Or Equal To	5 leq 5 = True
geq	Greater Than Or Equal To	5 geq 10 = False
.	Dot	f := [0 1 2], f.0 = 0
=	Equal To	5 = 5
!=	Not Equal To	5 != 5

Chapter 7: Proofs and Program Construction

With a minimal language implemented, showing how it plays a role within constructing programs is imperative. There are a few motivations behind this language and the Guarded Command Language that play into this idea of correctness. The first is that the language itself only has a small set of features. I believe that the intent of this was in part an exercise of minimalism. If the goal of the language is to construct a correct program then a maximalist syntax creates a lot of difficulties and complexity because if there's many ways to do the same thing, no one of these things are the right thing. For instance although the Repetition syntax is more verbose than the C-style for-loops, it can still do the same thing as the for-loops. The C-style family have for-loops and while loops and do-while loops and although there are subtle differences to these, trying to prove that a C++ program is correct is much more difficult due to the amount of features present within the language itself.

The second intent is that the features of the Guarded Command Language are informed by the processes in which a program is derived from the Eindhoven Quantifed Notation. This section will delve into how these features can be used to construct a correct program and how they are quite natural extensions of the ideas presented by the Eindhoven Quantifed Notation.

Proof methods are of the utmost importance to the Guarded Command Language and in my opinion, should be for all programming languages. I believe that formalising the act of programming to involve the use of proof methods would prove to be an invaluable skill for those who wish to write correct programs. It doesn't particularly make much sense to deviate from the foundations of programming to something that trivialises it to the extent that it is alien to what it should be : a rigorous proof. It would certainly solve many problems that we have today regarding the reliance on hardware to mitigate poor algorithmic choices. A language which comes naturally from proof methods is one that we can be more sure about it's correctness. If the Eindhoven Quantified Notation gives rise to the features of the Guarded Command Language, then we can be sure that a program within the Guarded Command Language is one that is more correct than alternatives. To illustrate this, the following proofs will be carried out : The Linear Search Theorem, The Bounded Linear Search Theorem, and The Longest All Zero Segment. These proofs will detail how much of the complexity of programming can be transferred to the rote process of carrying out the proofs using the Eindhoven Quantified Notation in such a way that the programs produced are guided by the proof of the program which was the goal of Dijkstra in creation of the notation to give rise to a natural way of constructing programs.

7.1 The Linear Search Theorem

The Linear Search algorithm is a very simple searching algorithm. It starts at the beginning of an array and iterates over it until it reaches the value at the index in which it is searching for.

Consider the following array of integers :

$$f = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Applying the Linear Search algorithm requires one to know that the value exists within the list therefore consider the problem in which given this array of integers, find the location in which the number five appears.

First the pre and post conditions of the program have to be specified. This informs of what should be computed therefore if the goal is to sum the even numbers in the array, then the pre and post conditions of the program are:

$$\{< \exists j : 0 \leq j < 10 : f.j \neq 5 >\}$$

S

$$\{< \forall j : 0 \leq j < 10 : f.j \neq 5 >\}$$

If we know at the beginning that at least one value in the array is the integer five, then the post condition of the program could be considered in such a way that for all values at every index are not five which implies that once the index wherein five lies, the loop iterating over the list breaks. And this is an example of how the idea that the Eindhoven Quantified Notation informs the structure and syntax of the Guarded Command Language. The aforementioned behaviour of the post condition is very much like Repetition. In order to construct the model, what's first needed is to strengthen the post condition.

$$< \forall j : 0 \leq j < n : f.j \neq 5 > \wedge n = 10$$

The motivation behind strengthening the program is to further generalise the program.

After generalising the program, the next step is model the program. To model the program, the first consideration is what possible Boolean value could be calculated at any given point *n* within the program. The natural formulation of this idea is the strengthened post condition because it could be used to figure out what every value is at every point between the first index in the array and the last. Therefore consider C.n which is able to calculate each value.

$$*(0) \quad C.n =< \forall j : 0 \leq j < n : f.j \neq 5 >, 0 \leq n \leq 10$$

The next thing to consider is the beginning of the eventual loop.

$$C.n =< \forall j : 0 \leq j < n : f.j \neq 5 >$$

$=$ {n := 0}

$$C.0 =< \forall j : 0 \leq j < 0 : f.j \neq 5 >$$

$=$ {Empty Range}

$$True$$

$$.(1) \quad C.0 = True$$

Now it's necessary to look at what the next value after C.0 could be. The natural step is to calculate C.1 but it's necessary to generalise the program because the more abstract it becomes, the more control of the program is offered. As a consequence of this, consider the next value to look at being C.(n + 1). The motivation behind this is reasoning about how C.n represents the most current point in the program thus C.(n + 1) naturally represents the state of the program logically and'ed to the value at the next index not equal to five. If this construct is False then the entire chain of logical ands breaks and the index is known.

$$.(2) \quad C.(n+1) = C.n \wedge n \neq 5, 0 \leq n < 10$$

With the model derived, the next thing to look to is establishing invariants. An invariant is something which doesn't change throughout the program and is maintained as such.

One natural invariant to pick is the range of the array as that doesn't change. The other invariant that can be chosen is C.n itself because this is what is maintained throughout the program.

Invariants

P0: C.n

P1: $0 \leq n \leq 10$

To prove the validity of the invariants, they have to be established. The model informs of what this should be. To establish an invariant, one method is to see what value it is at the beginning therefore in this case look to rule two in the model. This implies that to establish the invariant, set it to zero.

The next step to the program construction is to consider the loop guard. The loop guard is informed by what the task is : until the index in which the value that is being searched for is reached, iterate over the program. This implies the loop guard has to be : $f.n \neq 5$.

The next thing to calculate is the variant. The variant is what actually changes throughout the program. It can't be C.n because that is the invariant. Consider it to be some index K. If K represents the unknown index where the value lies, the invariant could be represented as $K - n$.

The only thing that's left to derive is the loop body. Consider increasing the invariant by one.

$$(n := n + 1).P0$$

$=$ {Text Substitution}

$$C.(n + 1)$$

$=$ {(2)}

$$C.n \wedge f.n \neq 5$$

$=$ $\{f.n \neq 5 \quad means \quad loop \quad is \quad entered\}$

$$C.n$$

$=$

$$\{P0\}$$

$$True$$

Everything from the above contains all that is needed to construct the program. The model of the program guided the work of constructing the invariants, variants, and loop body which is what Dijkstra intended in his creation of the Guarded Command Language. Establishment informs on what to do before constructing the loop, the loop guard is informed by the task at hand, and the invariants come quite naturally from the post condition.

Listing 7.1: The Linear Search Theorem

```
n := 0
;Do n != 5 ->
            n := n + 1
 Od
```

And then an example program from the implemented language.

```
~/Desktop/fyp_branch/Olivia  master  cat Docs/Linear_Search.lsy
print("The following programs performs a Linear Search to find the location of the integer five.")

f := [0 1 2 3 4 5 6 7 8 9]
n := 0
N := len(f)

x := 5
a := f.n

Do (a != x)->

        a := f.n
        n := n + 1
Od

print(n)
~/Desktop/fyp_branch/Olivia  master  ./Olivia Docs/Linear_Search.lsy
"The following programs performs a Linear Search to find the location of the integer five."
6
```

7.2 The Bounded Linear Search Theorem

The motivation behind the Bounded Linear Search algorithm is to search within a list for a value that could possibly exist. It is obviously much more powerful than the Linear Search Theorem as a result and due to it's relation with it, it isn't too much of an extension to the Linear Search Theorem. The only difference to it is to consider the possibility of not being able to find what one is searching for. This is represented in the loop guard of the program. Either the value is found or the end of the list is reached.

Listing 7.2: The Bounded Linear Search Theorem

```
n := 0
;Do f.n != X ^ n != N - 1 ->
            n := n + 1
 Od
```

```
~/Desktop/fyp_branch/Olivia  master ±  cat Docs/Bounded_Linear_Search.lsy
print("The following program performs a Bounded Linear Search")
print("to find if the integer five exists within the given array")
print("and returns the index of which it is stored if it is there.")

f := [0 1 2 3 4 5 6 7 8 9]
n := 0
N := len(f)
X := 5

a := f.n

Do ((a != X) and (n != (N - 1)))->

                a := f.n
                n := n + 1

Od

select (a = X)-> print(n)
       ((n = (N - 1)) and (a != X))-> print("Element not in list")
end
~/Desktop/fyp_branch/Olivia  master ±  ./Olivia Docs/Bounded_Linear_Search.lsy
"The following program performs a Bounded Linear Search"
"to find if the integer five exists within the given array"
"and returns the index of which it is stored if it is there."
6
```

7.3 The Longest All Zero Segment

To show how the style of the Eindhoven Quantified Notation leads to correctness and informs of the syntax of the Guarded Command Language, consider a more complicated program : calculate the length of the longest all zero segment. Although the actual program itself isn't too complicated, the proof of the program is much more complicated than the Linear Searches and will therefore serve as a good marker of the power of the Eindhoven Quantified Notation such as how modelling this program is much like the Linear Search Theorem but also the power of the simplicity of the Guarded Command Language.

Consider the pre and post condition.

$$\{f = [1, 2, 3, 0, 0, 0]\}$$

S

$$\{r =<\uparrow i, j : 0 \leq i \leq j \leq N \wedge Az.i.j : j - i >\}$$

An array of integers of a general length N (in this case five) are acted upon to produce the post condition. This operator ↑ (pronounced "up") acts much like the max operator. For all indices in the array i,j, the up operator finds the longest all zero segment given the value at these indices being applied to the the function Az.

Az is a function which takes a sub-array and calculates whether or not it contains all zeros.

$$*(0)Az.i.j \quad = \quad < \forall k : i \leq k < j : f.k = 0 >, 0 \leq i \leq j \leq N$$

Like in the case of the Linear Search, the next step is to look at the beginning and appeal to Empty Range.

$$.(1)Az.i.i \quad = \quad True, \quad 0 \leq i \leq N$$

This is exactly the same result as in the Linear Search. What's fascinating about this calculational style of constructing programs is that it requires less thinking than it may appear. Empty Range always gives the identity of the operator which is being applied. Similarly, the C.(n + 1) cases and Az.i.(j + 1) exhibit the same behaviour too. Normally what is returned is the current instance of C.n or Az.i.j and the next instance of it being applied upon by the operator.

$$.(2)Az.i.(j+1) \quad = \quad Az.i.j \wedge f.j = 0, \quad 0 \leq i \leq j < N$$

Correctness guides the proof of the program. With Az totally derived, focus can now be put upon achieving the post condition.

$$*(3)C.n = <\uparrow i, j : 0 \leq i \leq j \leq n \wedge Az.i.j : j - i >, 0 \leq n \leq N$$

Calculating C.0 is a bit different than before due to how the last index can be reached rather than reaching the index before it. The One-Point Rule can be employed in this case. This rule is defined in the following way:

$$< \oplus j : j = i : f.j >$$

= {One-Point}

$$f.i$$

Applying this to C.n in the case of C.0 gives:

$$.(4)C.0 \quad = \quad 0$$

For C.(n+1) :

$$C.(n+1)$$

= {(3)}

$$<\uparrow i, j : 0 \leq i \leq j \leq n+1 \wedge Az.i.j : j - i >$$

= {Split j = n + 1 term}

$$<\uparrow i,j : 0 \leq i \leq j \leq n \wedge Az.i.j : j-i > \uparrow <\uparrow i : 0 \leq i \leq n+1 \wedge Az.i.(n+1) : (n+1)-i >$$

= {(3)}

$$C.n \uparrow <\uparrow i : 0 \leq i \leq n+1 \wedge Az.i.(n+1) : (n+1)-i >$$

= {Consider the new expression as D.n}

$$C.n \uparrow D.(n+1)$$

$$.(5) C.(n+1) \quad = \quad C.n \uparrow D.(n+1), 0 \leq n < N$$

A new expression being produced isn't too surprising. There exists two complicated expressions (C.n,Az) therefore it makes sense that a new expression would be produced to consider the zero segments while the other allows for the iteration across the array.

$$*(6) D.n \quad = \quad <\uparrow i : 0 \leq i \leq n \wedge Az.i.n : n-i >, 0 \leq n \leq N$$

Appealing to the One-Point Rule again gives

$$.(7) D.0 = 0$$

Regarding D.(n+1), there are two considerations : if f.n is zero or not.

$$D.(n+1)$$

= {(6)}

$$<\uparrow i : 0 \leq i \leq n+1 \wedge Az.i.(n+1) : (n+1)-i >$$

= {Split i = n + 1 term}

$$<\uparrow i : 0 \leq i \leq n \wedge Az.i.(n+1) : (n+1)-i > \uparrow (n+1)-(n+1)$$

=

$$<\uparrow i : 0 \leq i \leq n \wedge Az.i.(n+1) : (n+1)-i > \uparrow 0$$

= {(2)}

$$<\uparrow i : 0 \leq i \leq n \wedge Az.i.n \wedge f.n = 0 : (n+1)-i > \uparrow 0$$

= {f.n = 0 $\equiv True \rightarrow True \equiv ID \wedge$}

$$<\uparrow i : 0 \leq i \leq n \wedge Az.i.n : (n+1) - i >\uparrow 0$$

= {Addition and Up for non-empty ranges}

$$(1 + <\uparrow i : 0 \leq i \leq n \wedge Az.i.n : n - i >) \uparrow 0$$

= {(6)}

$$(1 + Dn) \uparrow 0$$

To calculate the other case, take the line just before the case analysis.

$$<\uparrow i : 0 \leq i \leq n \wedge Az.i.n \wedge f.n = 0 : (n+1) - i >\uparrow 0$$

= {f.n $\neq 0, P \wedge False \equiv False$}

$$<\uparrow i : False : (n+1) - i >\uparrow 0$$

= {Empty Range}

$$ID \uparrow\uparrow 0$$

=

$$0$$

These two cases provide the following derivations:

- $.(8)D.(n+1) = (1 + D.n) \uparrow 0 \leftarrow f.n = 0, 0 \leq n < N$
- $.(9)D.(n+1) = 0 \leftarrow f.n \neq, 0 \leq n < N$

The model of the program is completed. The creation of the invariants follows the same logic as in the Linear Search case but in this case, since there exists another expression D.n, then this too must be considered to be invariant.

Invariants

P0: r = C.n ^ d = D.n

P1: $0 \leq n \leq N$

To establish the invariants:

$$n, r, d := 0, 0, 0$$

The guard logically has to be $n \neq N$ otherwise the loop couldn't begin. If the guard is that then the variant has to be n as it's the only thing that can change.

Since there's two cases to consider (whether or not f.n is 0), then in the program, Selection must exist because of the necessity of the case analyses.

$$(n, r, d := n + 1, E, E').P0$$

$=$ {Text Substitution}

$$E = C.(n + 1) \wedge E' = D.(n + 1)$$

$=$ {(5)}

$$E = C.n \uparrow D.(n + 1) \wedge E' = D.(n + 1)$$

$=$ {f.n=0, (8) twice}

$$E = C.n \uparrow (1 + D.n) \uparrow 0 \wedge E' = (1 + D.n) \uparrow 0$$

$=$ {P0}

$$E = r \uparrow (1 + d) \uparrow 0 \wedge d = (1 + d) \uparrow 0$$

This gives : $f.n = 0 \rightarrow n, r, d := n + 1, r \uparrow (1 + d) \uparrow 0, (1 + d) \uparrow 0$

$$(n, r, d := n + 1, E, E').P0$$

$=$ {Text Substitution}

$$E = C.(n + 1) \wedge E' = D.(n + 1)$$

$=$ {(5)}

$$E = C.n \uparrow D.(n + 1) \wedge E' = D.(n + 1)$$

$=$ {f.n$\neq 0, (9) twice$}

$$E = C.n \uparrow 0 \uparrow 0 \wedge E' = 0 \uparrow 0$$

$=$ {P0, $\uparrow$ *idempotent*}

$$E = r \uparrow 0 \uparrow 0 \wedge E' = 0$$

This gives : $f.n \neq 0 \rightarrow n, r, d := n + 1, r \uparrow 0, 0$

There is nothing left to do bar writing the program itself. As before, establishment informs on what to achieve before beginning the loop, the loop guard informs the guard to the Repetition block and the fact that there's a case analysis in the construction implies the existence of Selection. In the following Guarded Command Language, $\uparrow$ is denoted by "up".

Listing 7.3: The Longest All Zero Segment

```
n, r, d := 0, 0, 0
;Do n != N ->
        if f.n = 0 ->  n,r,d := n + 1, r up (1+d) up 0, (1+d) up 0
           f.n != 0 -> n,r,d := n + 1, r up 0, 0
        fi
 Od
```

This program can be simplified further as C.n and D.n cannot be negative therefore within this program, anything up zero will just be that thing giving :

Listing 7.4: The Longest All Zero Segment

```
n, r, d := 0, 0, 0
;Do n != N ->
        if f.n = 0 ->  n,r,d := n + 1, r up (1+d), (1+d)
           f.n != 0 -> n,r,d := n + 1, r, 0
        fi
 Od
```

```
~/Desktop/fyp_branch/Olivia  master  cat Docs/The_Longest_All_Zero_Segment.lsy
print("The following program calculates the longest all zero segement.")
f := [1 2 3 0 0 0]
N := len(f)
n := 0
r := 0
d := 0

Do (n != N)->
           a := f.n
           select (a = 0)->  n := n + 1
                             r := r up (1 + d)
                             d := d + 1
                  (a != 0)-> n := n + 1
                             r := r
                             d := 0
           end
Od

print(r)
~/Desktop/fyp_branch/Olivia  master  ./Olivia Docs/The_Longest_All_Zero_Segment.lsy
"The following program calculates the longest all zero segement."
3
```

7.4 Construction as a way of Programming

There is an elegance to languages that focus on the syntax of the language as well as the semantics. It's not enough to just have the ability to do something within a language. Take Python for example. Although functional constructs like lambda, map, reduce are possible, they are somewhat awkward. Mapping a function over a list of integers in Python returns a map object that has to be converted back into a list whereas in Haskell, the list is just returned. In Haskell and other functional languages however these concepts are much more natural. A language can reinvent the wheel if it wishes but the language itself should inform the structure of its programs to give rise to correctness and the only way to do this is if the syntax is minimal. Correctness only arises within the Guarded Command Language because the properties and ideas within the Eindhoven Quantified Notation such as post conditions, establishment, case analysis give rise to the features of the Guarded Command Language and thus the implemented language presented.

Chapter 8: Future Work

This section will delve into future work on the language itself and extensions to it that weren't able to be completed due to time constraints.

8.1 Further Work On The Language

The language itself is extremely minimal. The only real data type that can be worked with are integers therefore future work would require the inclusion of floats and expansion on strings and characters and Booleans. Furthermore there exists a bug when working with lists/arrays that I was unable to solve.

Consider a list f : f := [1]. If one wishes to perform arithmetic using this, it must be on the right hand side of an expression such that :

- 5 + f.0 is valid
- f.0 + 5 is invalid

This behaviour persists to guards of Selection and Repetition. The best solution is to instantiate a running variable with the current value of the list and operate as normal like in the program of the Longest All Zero Segment.

Although developing the language with Haskell was extremely rewarding, it was also at times frustrating. Part of this is due to the package management system. There are two types that can be used : Stack and Cabal. These essentially do the same things but they became a necessity in the project when I tried to explore other libraries. Part of my goal was to complete this project primarily using just the base library of Haskell and this was achieved but installing the parsing library for example took a few days due to the dependency problem within Haskell projects. If this package requires a package of version X then you are only told this upon the installation of the first package. What Cabal allows is the ability to create a Cabal file which one can specify the version of the package they want which fixes this dependency problem but it isn't really that good of a solution.

The issue of portability still remains. Although the code is more portable than other Haskell projects in comparison due to the focus of using the base library, Haskell itself isn't that portable as it doesn't have that much of a wide spread use. As a result of this there are two options to take : the first is to rewrite the whole language is a lower language like C or Rust which is great for these kinds of projects or transpile the project to another language

like JavaScript.[1] Developing the language in this way would also afford the opportunity to solve the issue of having strange representation for parsers.

8.2 Functions

Due to time constraints, functions were not implemented. This feature would be the first thing that would be implemented in future work of the project.

A possible strategy to implement them would be to follow the logic of assignment. They work in the similar way. Assignment binds a value to a variable and a function binds a collection of statements to a keyword.

8.3 Sigma Calculus

Cardellis and Abadis *A Theory of Objects*[24] serves as what could be an interesting foundation to the inclusion of objects within the Guarded Command Language. The goal of the book is to construct an understanding of object orientated languages in the same vein to how procedural languages are understood. The authors present an interesting calculi in which they develop their theory of objects. This theory concerns itself with not just the semantics of objects but also concepts such as classes and inheritance which aren't able to be modelled by the Eindhoven Quantified and thus the Guarded Command Language. Discovering a way to meld the two would lead to a very interesting language in which it can handle the complexity of objects and more complicated structures than arrays but is also provably correct.

[1] https://github.com/commandodev/ghcjs

Chapter 9: Acknowledgments

I would first like to thank my supervisor Fintan Costello who provided me with a lot of guidance and help over the course of the year in the completion of this project.

The theorems presented within this paper and those included within the repository which accompanies this project have come from the modules Program Construction One and Two by the permission of Henry McLoughlin.
The code for printing strings, error checking, and Assignment was taken from Tang's *Write Yourself a Scheme in 48 Hours*[16].

Bibliography

1. Dijkstra, E. W. Programming as a discipline of mathematical nature. *The American Mathematical Monthly* **81,** 608–612 (1974).
2. Prat, C. S., Madhyastha, T. M., Mottarella, M. J. & Kuo, C.-H. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports* **10,** 1–10 (2020).
3. Wall, L. Natural Language Principles in Perl. *Published web-only http://www. wall. org/~ larry/natural. html.* http://wall.org/~larry/natural.html (2007).
4. Iverson, K. E. in *ACM Turing award lectures* 1979 (2007).
5. *C and C++ Integer Limits* https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-160.
6. Olmedo, F. *et al.* Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **40,** 1–50 (2018).
7. Sampson, A. *Probabilistic Programming* http://adriansampson.net/doc/ppl.html.
8. König. *König's Lemma* https://mathworld.wolfram.com/KoenigsLemma.html.
9. McCarthy, J. in *Studies in Logic and the Foundations of Mathematics* 33–70 (Elsevier, 1959).
10. *Haskell Implementation of Lisps Amb* https://wiki.haskell.org/Amb.
11. *Control.Monad.Logic* https://hackage.haskell.org/package/logict/docs/Control-Monad-Logic.html.
12. Kiselyov, O., Shan, C.-c., Friedman, D. P. & Sabry, A. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *ACM SIGPLAN Notices* **40,** 192–203 (2005).
13. *Control.Effect.NonDet* http://hackage.haskell.org/package/fused-effects-0.4.0.0/docs/Control-Effect-NonDet.html.
14. *Text.Parsec* https://hackage.haskell.org/package/parsec-3.1.11/docs/Text-Parsec.html.
15. *Guarded Commands Haskell Library* https://hackage.haskell.org/package/language-gcl-0.2/docs/Language-GuardedCommands.html.
16. Tang, J. *Write Yourself a Scheme in 48 Hours* https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours.
17. *Type - HaskellWiki* https://wiki.haskell.org/Type.
18. *Haskell Tutorial - Monads* https://www.haskell.org/tutorial/monads.html.
19. *Haskells Prelude Library* https://hackage.haskell.org/package/base-4.14.1.0/docs/Prelude.html#v:fmap.
20. Pratt, V. R. *Top down operator precedence* in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1973), 41–51.
21. Pierce, B. C. & Benjamin, C. *Types and programming languages* (MIT press, 2002).
22. *Data.Foldable* https://hackage.haskell.org/package/base-4.14.1.0/docs/Data-Foldable.html.
23. *Async / Control.Concurrent.Async* https://hackage.haskell.org/package/async.
24. Abadi, M. & Cardelli, L. *A theory of objects* (Springer Science & Business Media, 2012).