Quantum

The Parallel Programming Language

Project Report

By SW4-04-F16

Aalborg University

AALBORG UNIVERSITY

STUDENT REPORT

Computer Science
Aalborg University
Cassiopeia
Selma lagerløfs vej 300
9220 Aalborg Øst
http://www.aau.dk

Title:
Quantum

Theme:
Design, Definition and implementation of a programming-language

Project Period:
Spring Semester 2016

Project Group:
DS4-04-F16

Participants:
Daniel Thiemer Sørensen
Jakob Koch
Mads Fich Engelbreth
Nikolaj Lepka
Patrick Lynnerup

Supervisor:
Søren Kejser Jensen

Copies: 1

Page Numbers: 62

Date of Completion:
March 15, 2018

Abstract:

This report aims at covering the subject of designing, implementing, and formally describing a new programming language. The aim of this language is to ease parallel programming in an object oriented programming context; in doing so, it explores message-passing systems and ways to manage these. The language must aim towards avoiding race conditions, starvation, and deadlocks; which are major problems that can occur in parallel computing. The report explains parallelism and some of the key structures of designing a programming language.

Contents

Chapter 1

The Initial Problem

New processors today rarely have a significantly higher clock frequency compared the previous processor generation, instead the processors often gets physically smaller, and the amount of transistors in the processors increase[1].
The processors clock frequency is directly linked to its heat dissipation which results in several problems trying to keep the processor cold, the higher the clock frequency is[2; 3]. Instead of increasing the clock frequency of the processor, the newer generations of processors often have more cores clocked at a lower frequency[4], which makes it possible to perform several computations in parallel, with a more manageable heat dissipation, than at a higher clock frequency.
To utilise the many cores in a multi-core processing unit, it requires that the programs perform several computations in parallel. This can be difficult for the programmer to manage, as a program thus no longer is a sequential sequence of computations and effects[5], as the order of when the computations are performed can vary for each execution of a parallel program[6; 7; 8].

This project will look at how developing software, designed for modern processors, by modern meaning processors with multiple cores produced after 2009. Creating a programming language which can be simpler for the programmer to learn and use; and how many of the current problems characterising parallel programming can be avoided with a new programming language.

When designing a new programming language that is easy to read and write for developers; it is important to know what programming paradigms most developers already use. According to the Tiobe index[9], an index showing the current popularity of programming languages, four of the top five languages are Object-Oriented Programming (OOP) languages[9; 10]. The combined usage percentage of the four languages account for a third of the programming languages[9]. This makes the OOP paradigm

a good choice for a new programming language. In the top right corner of [Figure 1.1][1] is the languages with most GitHub repositories and tags on StackOverflow, among the top 10 languages 8 of them are primarily Object-Oriented (OO).

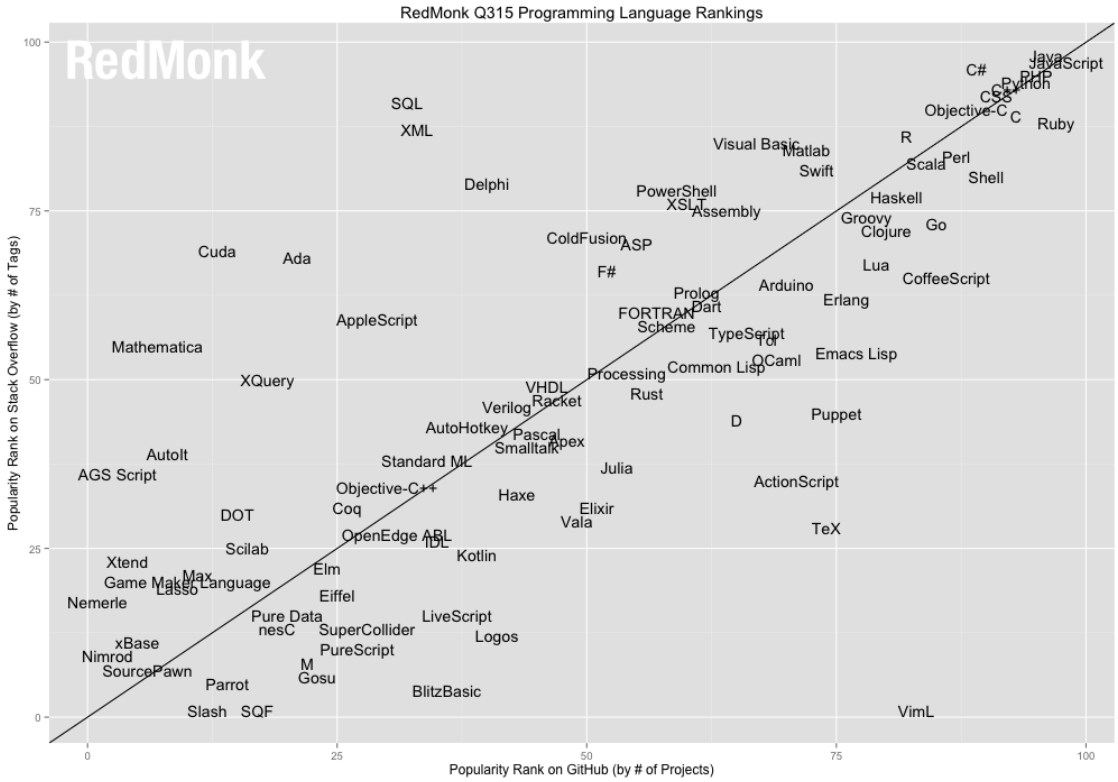


Figure 1.1: RedMonk programming rankings from 2015[11]

Designing a new language within this paradigm, means that developers would be able to pick up on the fundamentals of the language quickly, without needing an introduction to the basic principles of the language. Because of this, the programming language that will be designed and implemented in conjunction with this paper will be OO.

This leads to the following initial problem statement:

How can developing programs for a multi-core processing unit be optimised in regards to the programmer in an OO language?

[1] For a full size picture see appendix B

Chapter 2

Parallelism

In this chapter parallelism and parallel models and methods will be described.

2.1 Parallelism Definitions

This chapter will describe and define a classification of computer architectures based on Flynn's taxonomy and Parallel Random Access Machine[12]. Afterwards a description and a definition for each of the terms *Parallel Processing* and *Parallel Programming* so that the rest of the paper can work with these definitions. This is done to make sure that the reader of this paper understands these terms and use the definitions as references if needed.

2.1.1 Flynn's Taxonomy

Flynn's taxonomy is a categorisation of forms of parallel computer architectures[13]. This categorisation is based on the notion of a stream of information[12]

Single Instruction Single Data

Conventional single processor, uniprocessor, computers are classified as Single Input Single Data (SISD) systems[14]. These processors compute a single instruction at a time with a single piece of data as seen in [Figure 2.1]. SISD cannot be parallelised as it is a single instruction that is being executed on a single data, which means it

cannot be divided[14].

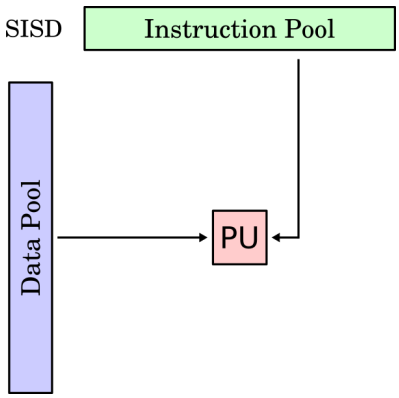


Figure 2.1: A visual representation of SISD[15]

Single Instruction Multiple Data

Single Input Multiple Data (SIMD) machines have one instruction processing unit, sometimes called a controller, and several processing units as seen in [Figure 2.2][16]. One advantage of this style of parallel machine organisation is a saving in the amount of logic, anywhere from 20 to 50 percent of the logic on a typical processor chip is dedicated to control, decoding and scheduling instructions[16]. The rest is used for on-chip storage and the logic required to implement the data processing[16].

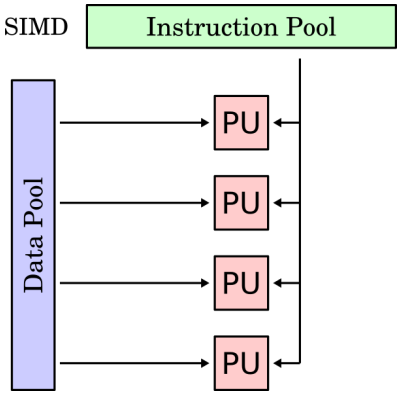


Figure 2.2: A visual representation of SIMD[15]

Multiple Instruction Single Data

Multiple Input Single Data (MISD) is a type of parallel computing where many functional units perform different operations on the same data as seen in [Figure 2.3][17]. In other words, all instructions have shared access to the same data on which the execute their own separate instructions. Fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors, in a manner known as task replication, may be considered to belong to this type[17]. Not many instances of this architecture exist compared to Multiple Input Multiple Data (MIMD) and SIMD, as they are more appropriate for common data parallel techniques[17]. SIMD and MIMD allow for better scaling and use of computational resources compared to MISD, however, one prominent example of MISD computing are the Space Shuttle flight control computers[17].

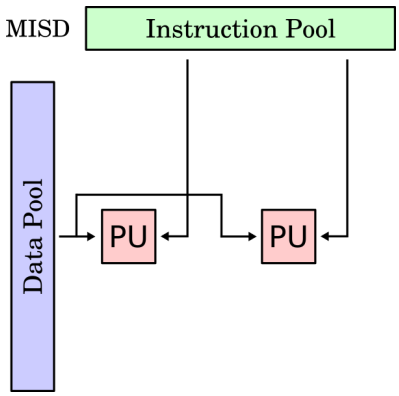


Figure 2.3: A visual representation of MISD[15]

Multiple Instruction Multiple Data

In computing MIMD is a technique used to achieve parallelism[17]. Machines using MIMD have a number of processors that function asynchronously and independently of each other[17]. Different processors may be executing different instructions on different pieces of data, at any point in time[17] as can be seen in [Figure 2.4]. This architecture is the most complex of the four mentioned, but also the most versatile in terms of the amount and types of operations that it can perform and is as such also the most used[17]. This architecture may be used in various application areas such as simulation, modeling, and as communication switches[17]. Machines using MIMD can be of two categories, either shared memory or distributed memory[17].

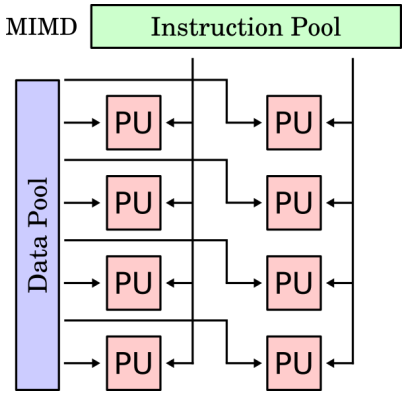


Figure 2.4: A visual representation of MIMD[15]

2.1.2 Parallel Processing and programing

The definitions of *Parallel Programming* and *Parallel Processing* will be:

- **Parallel Processing** — When multiple cores work on different tasks at the same time, with an intended performance gain.
- **Parallel Programming** — When programmers write programs that utilise multiple cores by dividing tasks so they can be executed simultaneously.

Based on their descriptions and definitions in their sections later in the chapter.

Programs that utilise multiple cores to solve multiple tasks at the same time use *parallel processing* to achieve this. This means that the program delegates tasks to the different cores of the available Central Processing Unit (CPU). This speeds up the process of solving these tasks, since the computer does not have to do them sequentially, but can work on them simultaneously.

Much software nowadays require more computing power than a conventional sequential CPU can offer[18]. A cost-efficient way to get around this problem is by using another CPU with more cores in the multi-core system and implement an efficient way for the different modules to communicate[18]. This is done so the work-load can be shared efficiently, and to harness the full potential of the available CPU cores[18]. If this is done effectively, better performance can be achieved than what could otherwise be achieved with a single processor[18].

Development of multi-core CPU is highly being influenced by many factors. Some of the most prominent are:

- The sequential architectures of computers are reaching their physical limitations due to constraints such as the laws of thermodynamics. Because of this the clock frequency at which sequential CPUs can operate at is reaching it's limit and other ways of increasing processing power have to be used, such as connecting multiple cores[18].
- Computational requirements are increasing in both scientific and business areas of computing. Being able to utilise multiple cores in these fields are important to be able to do them efficiently[18].

When programmers develop software for multi-core CPU they use parallel programming to divide tasks into subtasks. Subtasks are essentially just smaller components of a task that can be worked on, individually. These subtasks are then delegated to different cores so they can be worked on simultaneously. By doing so, the amount of time a task takes to complete is essential divided by the number of subtasks able to run simultaneously, in the best case. In other cases some of the subtasks will have to wait for other tasks to complete in order to finish and thus the gain is not linear.

2.1.3 Parallel Random Access Machine

Parallel Random Access Machine (PRAM) is a model for parallel computer architecture, that uses a shared memory, which is shared among all CPU cores in the system. Every CPU is said to be a Random Access Machine with a private local memory[19]. The more shared memory in the system, the more information can be shared between the CPU cores. In turn, private memory is faster for the CPU to read from.

PRAM instructions comes in three different variations[19; 20]:

- Reading from the shared memory.
- Local computations in the private memory.
- Writing to the shared memory.

Furthermore, the CPU's perform these instructions synchronous. To avoid conflicts between the many CPU cores, it is important to specify how the PRAM-system accesses the shared memory. There are numerous models for how this should be done: If only a single CPU core has read- and write-access to a memory cell at a time, the system is called *Exclusive Read, Exclusive Write* (EREW); if several CPU cores can read from the same cell, but only one can write to it at a time, it is called *Concurrent Read, Exclusive Write* (CREW). Some variations of these two models gives each

CPU core ownership of a single cell, which only they can write to; if this is the case the models are called *Exclusive Read, Owner Write* (EROW), and *Concurrent Read, Owner Write* (CROW) respectively. It is also possible to use a model, that allows simultaneous writes to memory cell, this is called *Concurrent Read, Concurrent Write* (CRCW). CRCW comes in three different variations:

- **Priority CRCW** - CPU cores are assigned priorities, the one with the highest priority is allowed to write.
- **Arbitrary CRCW** - A randomly selected CPU core gets writing permissions.
- **Common CRCW** - All CPU cores are allowed to write, but only if the value they write is same.

The benefit of using PRAM, is that it allows for the possibility to share data-structures in the memory upwards, so they can be calculated in parallel in their own core[21]. If a *Graphics Processing Unit* (GPU), which typically has several thousands of cores[1], is being used. Compared to a regular high-end desktop CPU which only has four cores, it is possible to parallelise large amounts of data, which is all contained in the same memory.

2.1.4 Data Parallelism

Data Parallelism is when a single operation is performed on a set of data at the same time, which means that a single operation e.g. the subtract operation can be performed on multiple sets of data, without a multi-core CPU.

To do this operation a CPU would do each set sequentially, using cycles on each operation, instead of doing it concurrently[23]. Which was seen in section 2.1.1.

2.1.5 Task Parallelism

Task Parallelism is the simultaneous execution, on multiple cores, of many different instructions across the same, or different datasets. Task parallelism is achieved in a multi-core processor environment where each processor executes a different thread or process on the same or different data. Task parallelism focuses on distributing tasks across different parallel computing cores. Task parallelism differs from data

[1] An NVIDIA GTX 980Ti as an example has 2816 cores[22]

parallelism in that data parallelism works on shared data sets; an example of this could be converting a character array to uppercase: Each character would get its own thread or process that then all in parallel perform the conversion. By contrast, since task parallelism is more suited for running independent instructions on the same dataset without modifying the originals; this could for instance be getting the average, the sum, the product, and the max-value of an array of numbers in parallel; since each instruction is independent of the other, and does not modify the original array, they can all run in parallel.[24]

2.2 Parallelism Method

In this chapter message-passing will be described, as both parallelism models described in the sections 2.4.1 & 2.4.2 are based on this model, and message-passing works well with the OO paradigm, and the functional paradigm.

2.2.1 Multi-Threading

A single, or multiple processes with a single thread makes it irrelevant to talk about threading as there is only one thread to work on[25]. However, when working with more than one thread per process things get somewhat more interesting. As seen in [Figure 2.5], multi-threading is several concurrent paths or execution within a single process that can be run concurrently on one CPU core, or in parallel on multiple cores depending on the Operating System (OS)[25].

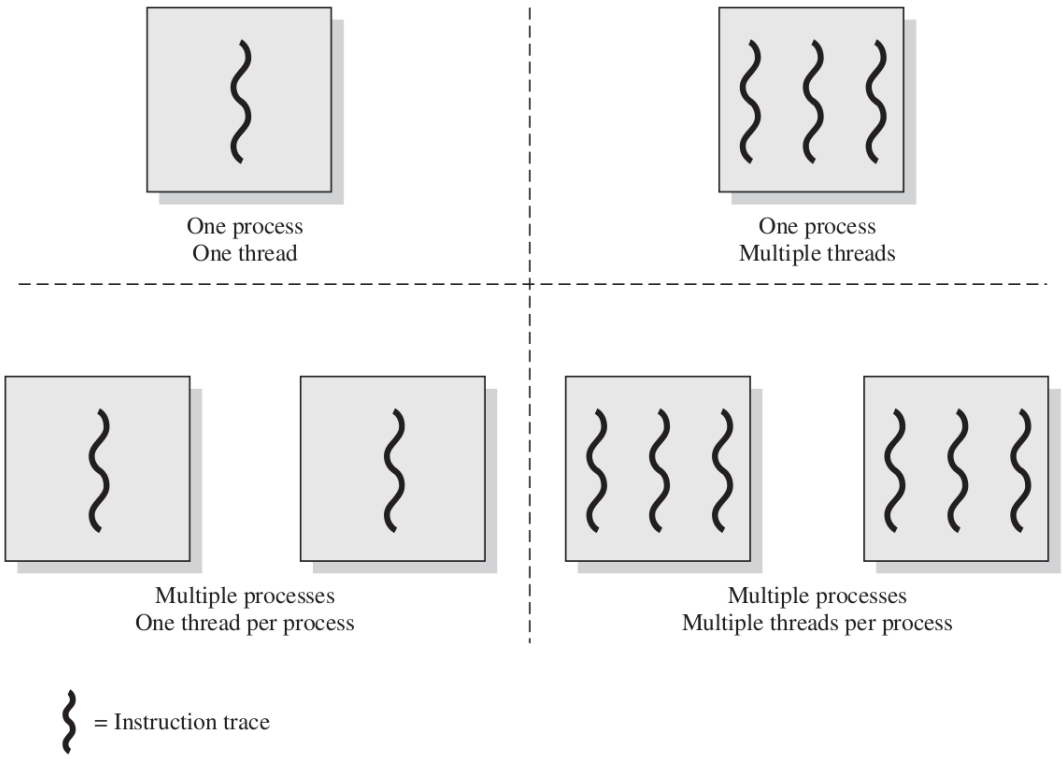


Figure 2.5: A representation of threading in processes found in figure 4.1 of [25]

A thread contains the following properties[25]:

- Execution State.
- Saved context when not running.
- Execution stack.
- Static storage for variables.
- Access to the resources of the process the thread exists in.

These properties are for a single thread and each individual thread must have them[25]. Some of the benefits of programming using threads is the fact that it is much faster to create threads than it is processes. It is also faster terminating and switching between threads. Another advantage of multi-threading is that the threads have the same access to all shared data. Because they exist within the same process they can communicate freely without having to involve the kernel[25].

When implementing threading there are two main methods, it is either implemented as user-level threads or as kernel-level threads. These have different advantages and disadvantages. With only user-level threads the kernel is not at all aware of the threads existence. This means that there are some specific advantages, one of which is that scheduling can be customised to the pattern that best fits the application. Another big advantage is that it is not necessary to have kernel access in order to switch from thread to thread, thus avoiding the overhead of having to switch from user-mode to kernel and back again[25].
The disadvantages of the user-level threads are that if one thread is blocked all threads from that one process are blocked as well. This causes threads, that could still continue, to wait for the system calls to finish. Once the system calls have been finished, the process can again be activated by the kernel which allows other threads to run. It also limits the ability of utilising multiple cores because the kernel will only assign one core to one process as seen in a in [Figure 2.6][25].

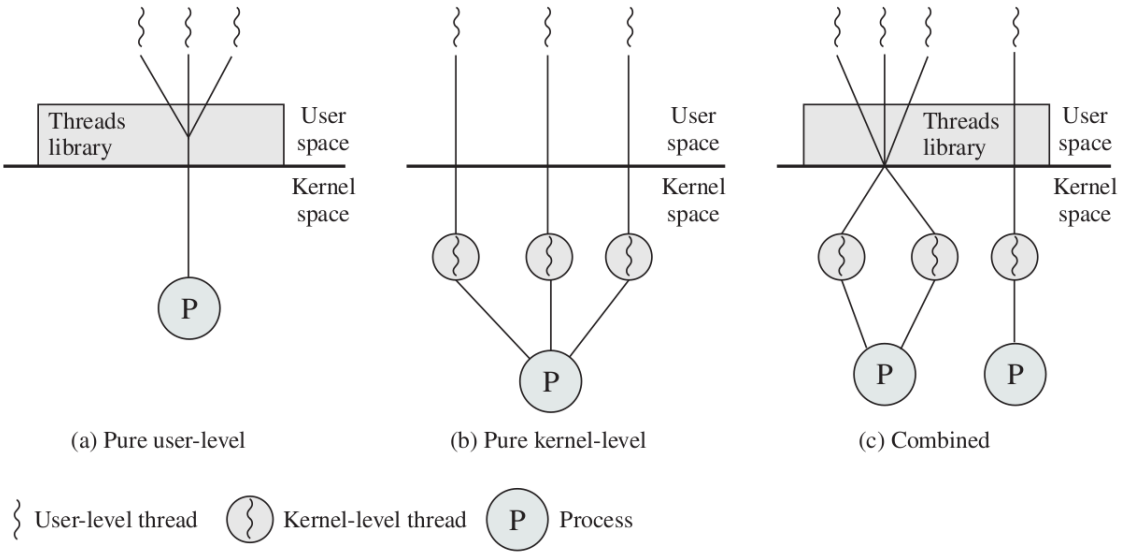


Figure 2.6: Visual representation of thread types found in figure 4.5 of [25].

For the kernel-level approach, the two disadvantages of user-level threads are taken care of; meaning that if one thread is blocked, another thread from the same process can continue running. It also opens up the opportunity to run threads from the same process on different cores at the same time. This means that there is a trade-off between the extra overhead and restrictions from kernel-level, and the blocking system calls and lacking multiprocessing of user-level as seen in b in [Figure 2.6][25]. These differences can be visualised in [Figure 2.6], for a better comparison between them.

However, there is also the possibility of combining the two approaches. If combined properly, it is possible to take advantage of the positives from both types while limiting the drawbacks. This means that the programmer can choose to use one type or the other based on which one suits the program part the best. If for instance the program sequence contains multiple system or Input/Output (I/O) calls a kernel-level thread might be of better use than a user-level thread to avoid interruptions for all threads in the process. Likewise, if the sequence calculates already loaded data, user-level threads can be of use to limit the overhead as seen in C in [Figure 2.6][25].

2.2.2 Message-Passing

In programming, a method for managing communication between multiple separate processes called *message-passing* can be used. The way message-passing works is by sending the processes a message and letting the infrastructure deal with running the correct code. What distinguishes message-passing from other types of programming is, that for each process exists separate data[26]. This means that if data needs to be transmitted from one process to another, it is necessary to make a copy of the data, to keep the processes internal memory separate. The reason for this is to ensure encapsulation of the processes and their data, which makes data that is necessary for a process immutable by another process[27].

In message-passing there are two main types, Multiple Program Multiple Data (MPMD) and Single Program Multiple Data (SPMD), which are an extension to Flynn's Taxonomy. In this project it will be the message-passing type SPMD that will be used, to simplify the work related to writing or maintaining code. It is individual programs, that is the focal point of this project and for that reason the SPMD type of parallel programming is the most relevant to continue with based on the problem definition.

Within message-passing there are two main categories. Those are Asynchronous-Message-Passing (AMP) and Synchronous-Message-Passing (SMP). In SMP the object that sends a message can continue to be locked while it awaits an answer from the object that receives the message[28]. This means, that data can be send with the messages as a *call-by-reference*, as long as the system is not a distributed system, a reference is a pointer to the data. Another advantage of SMP is, that it introduces a guarantee that there cannot be done computations using *old* data that should have been replaced before a given process was executed.[28].

This means that deadlocks can occur. A deadlock is a problem where several actions are waiting for each other to finish, before they themselves can finish or continue. This results in none of the actions ever being completed[29]. Another form of deadlock is livelock, where two different processes switch states constantly in respect to each other,

and hereby effectively never get any closer to completing their process[29].

In AMP after a request is sent, an answer will be given in return; through this, it can be avoided that the object that has sent the request gets locked in a state where it awaits an answer from the recipient. This means that, in contrast to SMP it needs to create a new copy of the data that is being sent with the message to the receiving object. The copying of data results in a bigger need for local storage, which can be hard for the system to manage[30; 31]. AMP alleviates the deadlock problem, in comparison to SMP. This is done with the cost of a more heavy data transmission compared to just sending references to the data. This does however open up to a new problem, the ability to do computations on *old* data. [28] Due to the increased computational capacity and the safety behind avoiding deadlocks, more work will be done regarding the AMP-type in this report. Some methods that implements and uses message-passing is described in section 2.4.

2.3 Processing Units

This section will describe and discuss both CPU and Graphics Processing Unit (GPU) to get an understanding of the fundamentals of these. Then there will be a comparison between them highlighting which features from each make them the better choice, when aiming for parallel computing. This will result in a choice of either the CPU or the GPU, with a reasoning behind the choice.

2.3.1 Central Processing Unit

The CPU is the component responsible for interpreting and executing most of the commands from the system. Principal components of a CPU include the Arithmetic Logic Unit (ALU) that performs arithmetic and logic operations. Processor registers that supply operands to the ALU and store the result of ALU operations, and a control unit that fetches instructions from memory and *executes* them by directing the coordinated operations of the ALU, registers and other components. CPUs from the 80's and later however are microprocessors, meaning that they are contained on a single Integrated Circuit (IC) chip[32]. This interaction within the CPU can be seen in [Figure 2.7].

An IC chip that contains a CPU may also contain memory, peripheral interfaces and other components of a computer, such integrated devices are variously called micro-controllers or System on a Chip (SoC)[34]. Modern computers employ a multi-core CPU, which is a single chip containing two or more CPUs called *cores*.

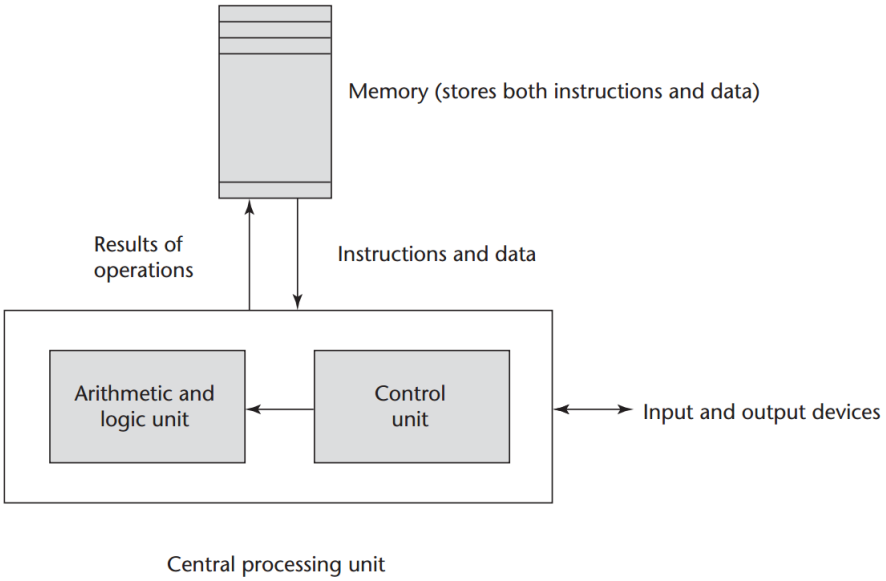


Figure 2.7: Central Processing Unit[33]

There are generally two types of microprocessors: General-Purpose microprocessors and dedicated microprocessors. General-Purpose microprocessors, such as the Pentium CPU can perform different tasks under the control of software instruction[35]. General-Purpose microprocessors are used in all personal computers[35]. Dedicated microprocessors, also known as Application-Specific Integrated Circuit (ASIC)s, on the other hand, are designed to perform just one specific task, though not for the x86 architecture[35]. An example could be the embedded microprocessor in a cell phone which does nothing besides controlling the operation of the phone[35].

2.3.2 Graphics Processing Unit

A GPU is a dedicated parallel processor optimised for accelerating graphical computations[36]. The GPU is designed to perform the many floating-point calculations essential to 3D graphics rendering[36]. Modern GPUs are massively parallel, and are fully programmable[36]. The parallel floating-point computing power found in a modern GPU is several times higher than a CPU[37]. The individual processors of the GPU are grouped into multiprocessors that share instruction decoder, memory access and a small amount of on-chip memory[38].

2.3.3 Central Processing Unit versus Graphics Processing Unit

To decide between using a standalone CPU or a GPU, it is necessary to understand which features stand out between the two. CPUs are low latency low throughput processors and GPUs are high latency high throughput processors[38]. GPUs can have more ALUs for the same sized chip, compared to a CPU, and therefore run more threads of computation[38]. When managing threads on a GPU some things to consider are how to[38]:

- Avoid synchronisation issues between so many threads.
- Dispatch schedule, cache, and context switch 10.000s of threads.
- Program 10.000s of threads.

GPUs use stream processing to achieve high throughput and the high latency tolerance results in a lower cache requirement[38]. Having less transistor area for cache, by having a higher latency tolerance, results in more area for more computing units[38].

2.3.4 Why Central Processing Unit?

The CPU is capable of running many types of software and has in more recent years provided multiple cores to process data in parallel[37]. On top of this the CPU is a low latency low throughput processor[38]. Architectural advances for the CPU such as branch prediction, out-of-order execution, and super-scalar have been responsible for a performance improvement[37].

2.4 Parallelism Models

After delimiting to task parallelism it is necessary to understand how parallelism functions on machine-level, hence there is a need to understand multi-threading. To understanding the further abstractions for the machine-level parallelism, it is necessary to take a look at the implementations and problems of parallelism models, two of which will be described in this chapter.
The two models will be the actor model and Communicating Sequential Processes (CSP) and based on these models the project will be delimited to one of them and the problems of that chosen model. These models have been chosen as the actor model works well with the OOP paradigm, as actors can be viewed as objects and CSP with concurrency. These models are base for/based on message-passing respectively.

2.4.1 Actor Model

The actor model is a mathematical model that treats actors as the universal primitives of digital computation. One of the advantages of the actor model is that an actor encapsulates data and behaves like objects in the OOP paradigm. This means that there is a separation of the interface and the representation of an actor. When an actor receives a message, it can concurrently make three types of local decisions[39]:

- Send messages to (unforgeable) addresses of actors that it has.
- Create new actors.
- Designate how to handle the next message it receives.

The actor model can be used as a framework for modelling, understanding, and reasoning about a wide range of concurrent systems[39]. Actors interact with each other through one actor sending a message, the messenger, to another actor, the target[40]. An actor has a name, an address that is unique and a behaviour which determines its actions. It is worth noting that an actor's behaviour can change depending on its local state. Each actor in a system carries out its own actions concurrently and asynchronously with other actors. In order to send a message to an actor, the actor's name must be used. This name cannot be guessed, but may be communicated in a message. The name of the actor can be implemented in a variety of ways[39]:

- Memory or disk addresses.
- Network addresses.
- Email addresses.

In the actor model a message's path from one actor to another or delays it may encounter are not specified. Therefore the order in which messages arrive is indeterminate. The actor model supports indeterminacy because the reception order of messages can affect future behaviour of the actors.

Some semantic properties of the actor model are[27]:

- Encapsulation of state.
- Atomic execution of a method in response to messages received.
- Fairness in the scheduling of actors and the delivery of messages between these.

- Location transparency which enables mobility and distributed execution.

The actors in the actor model behaves as objects sending and receiving messages, and performing actions based on the messages. The actors can only send messages to other actors they already know.

2.4.2 Communicating Sequential Processes

CSP is an in-process algebra, and is based on the notion of message passing through channels. Channels are a model for inter-process communication and synchronisation through message passing; due to their synchronous nature, messages sent through channels will block, and the sender waits, until the receiver is ready to accept the message. This is sometimes also referred to as *rendezvous behaviour*[41].
Processes in CSP are inherently anonymous, as the only means of communicating with processes is through named channels[41]. The communication can happen as in the following examples

One to One
A single process sends data onto a channel while another process pulls that data off the channel[41].

One to Many
One process puts data onto a channel that multiple processes listen on; they can then pull it off the channel. The first process that pulls the data is the only one that can work with it. After the data has been pulled no other process can pull that data from the channel, which comes down to who pulls the fastest from the channel. A low-tech example of this could be workers pulling items off a shelf; once a certain item has been pulled off, the other workers will never see it[41].

Many to One
Many different processes can feed data into the channel and then a single process will accesses this data[41].

Many to Many
Many different processes feed data into the channel and many processes that can access this data. The access works the same way as for the one to many. Once the data has been pulled it cannot be accessed by other processes[41].

The amount of data the *sender*-process can put upon the channel is determined by a buffer. Before the process can put data onto the channel it has to check whether or not there is room for it. As an example; if the amount of data that can be on a channel is determined by n'. Then the sender-process will check whether or not there

is n' amount of data in the channel[41].
In the case that there is equal or more data on the channel the sender-process will simply wait with putting more data onto it. One could see the buffer as a sort of shelf. The sender-process puts boxes of data onto this shelf for the worker-processes to take off and handle. In case the shelf is full, the sender-process will go into *wait mode* before putting new boxes onto the shelf[41].

In the previous two subsections two models of parallelism in computing was looked into the actor model and CSP. The outcome of this research and analysis was two methods of message passing that both fit the target of this project. The actor model had properties that made it the better fit for the Quantum language. Namely that actors are objects them selves which went along with the premise that everything in Quantum would be objects. Where as the processes in CSP were inherently anonymous which would be a difficulty to be overcome in the later process.
OOP languages revolve around modelling real things as objects, the actor model in section 2.4.1 is a good way of utilising parallel computing. This is due to the fact that each actor in an actor model in itself is an object. Therefore this report will work with implementing the actor model into a new language.

2.5 Problem Statement

Previously in the chapter some choices regarding the outcome of the project and language was made leading to a concrete problem statement:

Could multi-core processing become a more streamlined experience by implementing the actor model in a manner invisible to the developer?

Chapter 3

Language Design

In this chapter, various possible and considered design elements for Quantum will be described and chosen based on the description of the language design elements.

3.1 Readablility & Writability

In this section readability and writability will be described; followed by an explanation of how these are prioritised in relation to the language design of Quantum. This is done to give a better understanding of why some compromises between read- and writability were made with some design elements. The following paragraphs are based on source [33].

Readability

Readability refers to the ease with which programs can be understood. One can write hard-to-understand code in any language; but a language's characteristics can make it either easier, or more difficult, to write easy-to-understand code. If there are too many basic constructs or features, the program will likely be harder to read, as the reader might know a different subset of the language than the programmer. On the contrary, if there are very few, as in the Assembly language, the code can be hard to read because what may be a single operation, could require several instructions to represent it. Other ways of making it more, or less readable is feature multiplicity or, operator overloading; depending on how it is used. Feature multiplicity is the ability to do the same operation in multiple ways, an example could be the looping

constructs *while*, *do-while*, and *for*. Operator overloading can aid in readability if it is used with discretion, but can lessen readability if used unwisely, e.g. by using `+` as a comparison operator. Adequate facilities for defining data types and structures can also aid readability. Standard data types should be adequate too, an example would be the early versions of C in which there were no *boolean* type, forcing the programmer to use an *integer* to represent `true` and `false`. An example of using special keywords to increase readability could be the beginning/end of a compound statement, in this case a loop, where using curly brackets or an *end loop*. A good choice of special words helps in regards to the form and meaning of the program, e.g. using *if* and not *glorp*.

Writability

Writability is a measure of how easily a language can be used to develop programs for a chosen problem domain. The support for abstraction allows the programmer to define and use complicated structures/operations in ways that allow implementation details to be ignored, which is a key concept in modern language design. The expressivity is enhanced by the presence of powerful operators that make it possible to accomplish a lot in a few lines of code. The classic example is APL, which includes so many operators that it is based upon a character set larger than the one found on a typical keyboard. By contrast, Assembly languages typically lack expressivity in that each operation does something relatively simple, which is why a single instruction in a high-level language could translate into several instructions in assembly language.

Readability & Writability in Quantum

In Quantum it has been chosen to focus more on a mix of readability and writability, without the two clashing, so that both new and experienced programmers will find it understandable and useful. A summation of the language evaluation criteria in regards to readability and writability can be viewed in [Figure 3.1].

To accommodate this there have been some cases where readability has been down-prioritised for an increase in writability due to the readability suffering a minor loss, while writability instead gained a major increase. This has also been applied the other way around.

	CRITERIA	
Characteristic	READABILITY	WRITABILITY
Simplicity	•	•
Orthogonality	•	•
Data types	•	•
Syntax design	•	•
Support for abstraction		•
Expressivity		•
Type checking		

Figure 3.1: Language Evaluation Criteria[33]

3.2 Standard Data Types

This section will describe the standard data types that was considered to be included in Quantum. There will be arguments for and against the types as they are being described, along with an explanation of decisions as to what will be in- or excluded in Quantum.

Numbers

Natural numbers could be represented either by a fixed size integer type or arbitrary large numbers, with the fixed size integer having greater performance. When thinking about integers it is worth considering using or at least having an unsigned version to be able to work with even greater positive numbers, unless *bigints* are used. Decimal numbers are also necessary as they allow the programmer to work with real numbers, for this both floating point and double-precision floating point numbers (floats and doubles for short) could be used. As floats and doubles have precision limitations when working with long decimal numbers, another consideration for representing more precise numbers could be a type such as decimal[42]. With a precision of 28-29 significant digits in decimal compared to the 7 or 15-16 of floats and doubles[42; 43].

In Quantum there are no explicit different number types, such as integer and floats. Instead there is only the type *Number*, though behind the scenes it covers both integers and floating point numbers. The integer value starts off being the default word size of the machine, either 32-bit or 64-bit, while the floating point numbers are 64-bit, hereby being double precision floating point numbers. In Quantum an integer scales as needed, meaning that if a 64-bit integer isn't big enough to represent the entire

number an even bigger integer value will be assigned by use of a bigint system, which is an arbitrary-precision integer. Quantum uses implicit number conversion, so one can add an integer and floating point value together and the resulting number will be a floating point. Taking this into consideration in regards to bigint, if the programmer tries to add a sufficiently large bigint, and a floating point number a BadArithmetic error, called badarith in Erlang, will be thrown, as the floating point cannot be larger than 64-bit[44].

Characters

Allowing the programmer to work with text is necessary if they intend to communicate with humans, or other systems in a useful way. To do this, strings were considered to represent text in the form of alphabetic characters and numbers. When referring to a character, it is necessary to consider if there should be more than one type to represent text in Quantum. A single character could be represented as either a character or a single-character string, the latter resulting in having one less type in the language at the cost of a slightly larger overhead.

In Quantum it has been decided not to have several different ways to represent alphanumerical characters, such as text and single characters. To increase the writability, while losing some readability when trying to represent single-character values, it has instead been decided to only have strings, meaning single characters are represented as single-character strings. This leads to having a higher orthogonality due to the lower amount alphanumeric representations.

Boolean

Boolean were considered as it would make it more intuitive to work with logic operations such as if-statements. The Boolean expressions *true* and *false* were considered along with other similar keywords such as *yes* and *no* and whether or not there should be more than one keyword pair. It was however chosen to only have one keyword pair, to increase the writability and the orthogonality, while only losing a minor part of the readability in a very few special cases. The keyword pair chosen for Quantum will be *true* and *false*.

Types Without a Value

In Quantum it has been decided to include *Unit* as a type. Unit is special, in that it's a type without a value, and unlike null it is not a valid value in any other type. Unit is orthogonally useful as it allows writing functions without a return value without having to both define functions that don't return, and functions that do return; instead the function always returns, and can instead return a "nothing" value called Unit.

3.3 Immutability

This section is uses source [45].

Immutability is one of the key concepts in functional programming. In functional languages immutability is present in two ways; variables and data structures. Variables in functional languages are replaced by symbols and values that cannot be changed. This decreases writability as programmers cannot reuse variables, in turn forcing them to either use more variables or find a workaround. Immutability does however increase readability as the variables are never changed after they have been declared. As immutability influences the read- and writability of the language, it is also an important language design choice to consider in regards to order design choice. Having immutability is important in regards to parallelism as the programmer can avoid race conditions and deadlocks, as the data is never changed after being declared. Having immutable data will change the way in which the chosen data structures will be used. For example, one cannot change a record and would instead return a new record with the modified data.

In Quantum immutable data has been chosen to avoid race conditions, deadlocks and starvation, as the language is focused on parallelism. In parallelism these are some of the worse problems, and it is therefore important to avoid these.

3.4 Data Structures

The data structures that have been considered to be included in Quantum will be described in this section. There will be arguments for and against the structures as they are being described. This is done to illustrate the thought process behind the decisions of what structures to include in Quantum.

Array

The strengths of an array lies in its access speed and use of memory. The programmer is not able to change the size of an array after it has been defined, but if the programmer is aware of how many indices are needed before defining it, this limitation will not be a significant problem[46]. The reason for the fast accessing speed is that the array is placed sequentially in the memory[46]. But due to the array's static size, it cannot be used for data sets with varying sizes.

List

The list allows adding or removing of elements and thereby allowing for a more flexible collection type than an array[47]. The programmer can access elements by their integer index, the position in the list, and search for elements in the list[47]. A benefit with the list is that the programmer will not have to take the size of the list into consideration and they can ignore the underlying implementation because of this.

Matrix

The matrix data structure is either made up of an array of arrays, or a two-dimensional array. The array of arrays can be jagged, meaning that not all of the arrays must have the same length, while the two dimensional array does. The matrix data structure is commonly used to structure large datasets when working with scientific calculations and matrices. This would give the programmer a common generic structure to use instead of forcing them to implement their own version of a matrix.

Record

Records, usually called *struct* in the C family of programming languages, were considered for the language as they allow the programmer to group different kinds of data or create new data structures[48]. The array, list, and matrix only allow for one type grouped together for each assigned collection, i.e. it is not possible to have a list containing two lists of integers and strings.

Data Structures in Quantum

In Quantum it has been decided to use lists, as they can be built immutably by creating a new list from the old one using modified data. An obvious implementation of a list is the singly linked list, which lends itself well to recursive algorithms for easy manipulation.

3.5 Scope

One of the main reasons for have a scope is to keep variables in different parts of the program distinct from each other. Since there are only a small number of short variable names, and programmers share naming conventions of the variables. When using static scoping a variable always refers to its top-level environment. Static scoping is standard in most modern programming languages. Static scoping makes it more simple for the programmer to make modular code and reason about it, since the variable is defined in the top-level environment. On the contrary, using dynamic scoping, each identifier has a global stack of bindings. Dynamic scoping cannot be predicted at compile time, unlike static scope where it is possible just by looking at the program, which is why it is called dynamic scoping.

Static scoping is used in Quantum to ensure better one-to-one code translation between Quantum and Elixir, as Elixir uses static scoping. If dynamic scoping had been used, a way of translating the dynamic scope to a static scope would be needed, one way to do this would be by declaring variables globally.

3.6 Control Structures

In a programming language a control structure determines the order in which statements are executed[49]. There are three major control structures: Sequential execution, selection statement and iterative statement, these are the most common structures[49]. The sequential execution is when statements are executed one after another in order.

Selection Statement

A selection statement allows for execution path selection, most commonly used for two-way decision making, which is typically performed using an *if-then-else* structure[49]. Execution path selection is not forced to be only two-way decisions, but can be multiple decisions depending on the programming language. A way to allow two-way statements to express multiple selection statements could be by nesting the two-way statements, an example of this would be a nested if-statement. These two-way statements can be generalised to be multiple-selection statements which allow one of any number of statements to be selected. An example of a multiple-selection statement would be a switch-case statement where it is possible to to choose one of many different cases.

Iterative Statement

Iterative statements, often referred to as *loops*, are the ability to execute statements zero to many times. There are several different types of iterative statements, such as, *for*, *while* and *do-while*. Each has their own strengths and weaknesses depending on how the programmer needs to iterate. Using a *do-while* would be used to ensure that the enclosed code would be run at least once, compared to the *while* whereas it might not execute the enclosed code, if the statement is not true.

Control Structures in Quantum

In Quantum both *if-then-else* and *match* are used for multiple decisions. *Match* corresponds to a *switch-case* with added pattern matching. In terms of iterative statements, only one way of creating loops has been used, this being *for*. The reasoning for only having a single iterative statement is that it is not strictly necessary to have loops in Quantum as immutable data would make updating looping conditions difficult.

3.7 Function Structure

Functions are an important part of Quantum, as they introduce ways of solving problems regarding how code can be modularised, as well as how values are moved between these modules. It is also considered whether or not the ability to have multiple returns from functions should be possible.

Pass-By-Reference

Pass-by-reference copies the address of the function's parameter into the scope of the function, rather than the parameter itself. This results in an efficient method of value-passing, which lets the programmer access its value without having to copy the data as would be the case in pass-by-value[50].

Higher Order Functions

Higher order functions, allow the programmer to pass functions as input to other functions, this allows for the creation of more generic functions. A simple example in ECMAScript as follows:

```
function apply2(f, a, b) {
  return f(a, b);
}

var result = apply2(Math.pow, 2, 4); // same as Math.pow(2, 4);
```

This function *apply2* takes a function and two arguments, and returns the application of said function on those two arguments. This can be expanded to more complex use-cases.

Another use of higher order functions, is the ability to create and return new functions. A simple ECMAScript example:

```
function printerMaker(name) {
  return function() {
    console.log("Hello, " + name + "!");
  };
}

var mePrinter = printerMaker("Niko");
mePrinter(); // prints "Hello, Niko!"
```

Combining these features can lead to interesting results, where the output function from one function is the input of another.

Multiple Return

Having multiple return can enhance the readability as the programmer can return the result as soon as it is known. An example of multiple return from Go could be:

```
package main
import ‘‘fmt’’

func vals() (int, int) {
  return 1, 2
}

func main() {
  a, b := vals()
  fmt.Println(a)
fmt.Println(b)
}
```

Function Structure in Quantum

Pass-by-reference is used as there is no need to modify the data, due to how Quantum uses immutable data to combat some of the problems related to parallelism.

3.8 Static/Dynamic Types

Dynamically typed languages perform type checking at run-time, while statically typed language perform type checking at compile-time[51], This means code written in a dynamically typed language can compile even if it contains type errors that will prevent the code from running properly[51]. If statically typed code contains errors at compile-time, it will keep failing until these errors have been fixed[51]. Additionally, variables with a previously assigned type, can be overwritten with a value with a completely different type; this can create issues, as it can lead to unforeseen results.

Static types have been chosen for Quantum to ensure type checking at compile time. This is to ensure that no type changes take place, and if it does it alerts the user at compile time.

Chapter 4

Syntax & Semantics

4.1 Grammar

In this section the grammar for our language will be described, for the raw version of the grammar see appendix C.

4.1.1 Context-Free Grammars

A grammar consists of a collection of substitution rules, also called productions. Each of these rules appear as a line in the grammar, comprising a symbol and a string separated by an arrow. In [Figure 4.1] an example of a Context-Free Grammar (CFG) grammar is shown to give a better understanding of how a CFG is made[52]. The notation of Grammars in this report will be described here:

Notation

In this grammar description the following notation is how the symbols and terms are shown in the grammar.

- Non-terminal: <non-terminal-name>

Non-terminals are a part of the grammar that does not terminate.

- Terminal: 'terminal name'

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#1111$$

Terminals are a part of the grammar that terminates.

- Special Characters:

  $\boldsymbol{\epsilon}$ | ::=

$\boldsymbol{\epsilon}$ is a symbol that allows the non-terminals to be empty.
The | symbol is representing an alternative of a description for the previous non-terminal.
The ::= symbol represents that the non-terminals/terminals on the right side of the symbol, can be abbreviated to the non-terminal on the left side of the symbol.

$\langle A \rangle$ ::= '0' $\langle A \rangle$ '1'
| $\langle B \rangle$

$\langle B \rangle$ ::= '#'

Figure 4.1: Context-Free Grammar[52]

The non-terminal symbols are often represented by uppercase letters. The terminals are often lowercase letters, numbers, or special characters. One non-terminal is designated as the start variable and usually occurs on the left-hand side of the topmost rule. As seen in figure [Figure 4.1], A is the designated start variable. As an example, the grammar in [Figure 4.1] generates the string 000#111. The sequence of substitutions to obtain a string is called a derivation and a derivation of string 000#111 in the grammar in [Figure 4.1] is[52]:

This is achieved by following the rules of the grammar in [Figure 4.1], substituting the non-terminals to achieve the string. Another way to represent the same information pictorially can be done using a parse tree. An example of a parse tree is shown in figure [Figure 4.2][52].

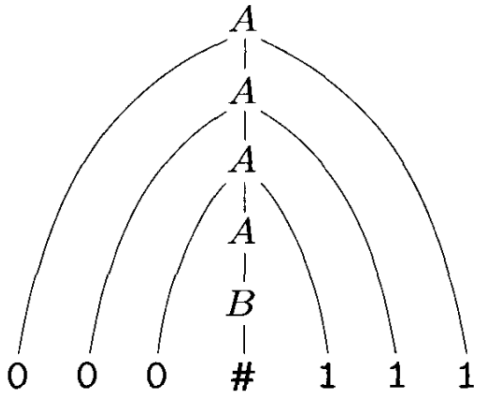


Figure 4.2: Parse tree for 000#111 in the grammar from [Figure 4.1][52]

This can be done to visualise CFGs, and verify that the string is achievable using the rules of the grammar[52].

4.1.2 Ambiguity

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and will thus have different meaning, as the order in which the substitutions are performed will vary. This result may be undesirable for certain applications, such as programming languages, where a given program should have a unique interpretation. If a grammar is called ambiguous if it generates the same string in several different ways. As an example, consider the grammar as shown in [Figure 4.3]:

$\langle EXPR \rangle$ ::= $\langle EXPR \rangle + \langle EXPR \rangle$
| $\langle EXPR \rangle \times \langle EXPR \rangle$
| '('$\langle EXPR \rangle$')'
| 'a'

Figure 4.3

This grammar generates the string $a + a \; x \; a$ ambiguously, where two different parse trees can be observed in figure [Figure 4.4] below[52].

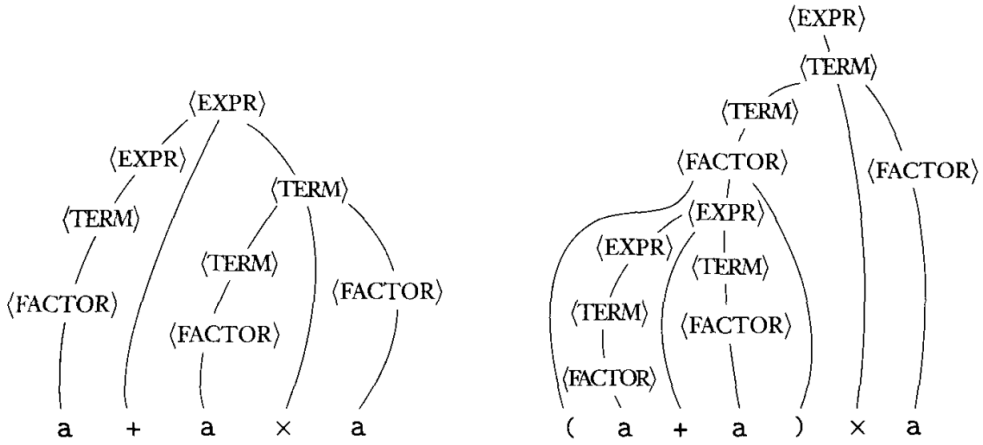


Figure 4.4: The two parse trees for the string $a + a \; x \; a$ in the grammar from [Figure 4.3][52]

This grammar does not capture the usual precedence relations and may group the + before the x or vice versa. When saying that a grammar generates a string ambiguously, it means that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace non-terminals yet not in their overall structure[52].

4.1.3 Backus-Naur Form

Backus-Naur Form (BNF) is a formal mathematical way to describe a language, developed by John Backus and Peter Naur, to describe the syntax of the Algol 60 programming language[53]. Formally,BNF grammars consists of a set of tokens that are divided into terminals, which are items that can appear in the language, and non-terminals, which can be expanded into one or more terminals or non-terminals[54]. Furthermore, there is a set of production rules mapping a non-terminal to a sequence of tokens. This grouping can be expressed using parentheses and choices in a group are separated by a vertical bar. A parse tree can then be generated by starting with an initial start-token and applying any rules until only terminals are left[55].

4.1.4 Extended Backus-Naur Form

Extended Backus-Naur Form (EBNF) is, as BNF, a notation for formally describing syntax.BNF has since its creation in the 1950's been improved into what is commonly

referred to as ExtendedBNF, of which there are several kinds. Among such extension, Augmented Backus-Naur Form (ABNF) is an Internet RFC and ISO/IEC defines a common uniform precise EBNF syntax[55]. All of these common extension are for practical purposes, as the various EBNF grammars can also be represented asBNF grammars[55].

4.1.5 Comparison

SinceBNF and EBNF both use terminals and non-terminals to set up the grammar for the language, it will not be used as a logical delimitation for the project, as there is no distinct difference in that feature. The EBNF allows for a non-terminal to call a non-terminal from a higher abstraction-level, and theBNF does not allow for this feature. The compilation time taken for the EBNF will increase as the is a higher number of calculations needed, caused by the multiple jumps to higher abstraction-levels, this has lead the group to choseBNF to keep compile-time to a minimum[52].

4.1.6 Backus-Naur Form Grammar

Regular Expression Representation

Regular expressions for accepted characters in Quantum.

⟨*string-lit*⟩ ::= '`"(\.|[^"])*"`'

⟨*identifier*⟩ ::= '`[a-zA-Z_][a-zA-Z0-9_'-]*`'

⟨*atom-def*⟩ ::= '`#[a-zA-Z_][a-zA-Z0-9_'-]*`'

⟨*num-lit*⟩ ::= '`[0-9]+`'

⟨*hex*⟩ ::= '`0x[0-9A-Fa-f]+`'

⟨*bin*⟩ ::= '`0b[01]+`'

The regular expressions is the character-symbols allowed in the terminals mentioned above, the regular expressions are for direct implementation of the language.

Top-Level Definitions

The top level rules for an acceptable code, the compiler will accept.

⟨*program*⟩ ::= '`module`' ⟨*module-name*⟩ '`;`' ⟨*top-level-cons*⟩

⟨*module-name*⟩ ::= ⟨*identifier*⟩ '`.`' ⟨*module-name*⟩
| ⟨*identifier*⟩ '`;`'

⟨*top-level-cons*⟩ ::= ⟨*module-import*⟩ ⟨*top-level-cons*⟩
| ⟨*actor-def*⟩ ⟨*top-level-cons*⟩
| ⟨*data-struct-def*⟩ ⟨*top-level-cons*⟩
| ϵ

⟨*module-import*⟩ ::= '`import`' ⟨*module-name*⟩ '`;`'

⟨*actor-def*⟩ ::= '`class`' ⟨*type-def*⟩ ⟨*actor-body-block*⟩
| '`class`' ⟨*type-def*⟩ '`<-`' ⟨*type-defs*⟩ ⟨*actor-body-block*⟩
| '`object`' ⟨*type-def*⟩ ⟨*actor-body-block*⟩
| '`object`' ⟨*type-def*⟩ '`<-`' ⟨*type-defs*⟩ ⟨*actor-body-block*⟩

In the top-level definition the instantiation of the program. This is followed by the class and object definitions, which is done through the actor definition.

Type Name Definitions

The abbreviation of types from the Top-Level Definitions, setting the acceptable type definitions for the language the compiler will accept.

⟨*type-defs*⟩ ::= ⟨*type-def*⟩ '`,`' ⟨*type-defs*⟩
| ⟨*type-def*⟩

⟨*type-def*⟩ ::= ⟨*identifier*⟩ '`of`' ⟨*type-params*⟩
| ⟨*identifier*⟩

⟨*type-params*⟩ ::= ⟨*type-param*⟩ '`,`' ⟨*type-params*⟩
| ⟨*type-param*⟩

⟨*type-param*⟩ ::= '`(`' ⟨*type-def*⟩ '`)`'
| ⟨*identifier*⟩

⟨*actor-body-block*⟩ ::= '`{`' ⟨*actor-body-def*⟩ '`}`'
| ϵ

⟨*actor-body-def*⟩ ::= ⟨*message-def*⟩ ⟨*actor-body-def*⟩
| ⟨*message-def*⟩
| ϵ

⟨*message-def*⟩ ::= define ⟨*type-def*⟩ ⟨*pattern-def*⟩ '`=`' ⟨*block*⟩

⟨*pattern-def*⟩ ::= ⟨*literal*⟩
| '`(`' ⟨*typedVal*⟩ '`)`'

⟨*typedVal*⟩ ::= ⟨*type-def*⟩ ⟨*identifier*⟩

Data Structure

Defines the data structures in the language and what it should contain.

⟨*data-struct-def*⟩ ::= '`data`' ⟨*type-def*⟩ ⟨*data-body-block*⟩
| '`data`' ⟨*type-def*⟩ ⟨*data-body-block*⟩ '`<-`' ⟨*type-defs*⟩

⟨*data-body-block*⟩ ::= '`{`' ⟨*field-defs*⟩ '`}`'
| ϵ

⟨*field-defs*⟩ ::= ⟨*typedVal*⟩ '`;`' ⟨*field-defs*⟩
| ϵ

As seen above the data structure definition uses the keyword *data* followed by a type definition and a data-body-block that is either empty or contains one or more field definitions.

Module

This describes what acceptable code blocks in the code can contain. An example would be a function definition, which would contain the *func* keyword followed by an optional identifier with function arguments enclosed in parentheses. A block, which contains one or more statements, is assigned to the function definition.

⟨*block*⟩ ::= '`{`' ⟨*stmts*⟩ '`}`'
| ⟨*stmt*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ⟨*stmts*⟩
| ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*expr*⟩ ‘`;`’
| ⟨*valDef*⟩ ‘`;`’
| ⟨*binaryOperation*⟩ ‘`;`’

⟨*valDef*⟩ ::= ‘`val`’ ⟨*identifier*⟩ ‘`=`’ ⟨*expr*⟩
| ‘`val`’ ⟨*typedVal*⟩ ‘`=`’ ⟨*expr*⟩

⟨*funDef*⟩ ::= ‘`func`’ ⟨*identifier*⟩ ‘`(`’ ⟨*funArgs*⟩ ‘`)`’ ‘`=`’ ⟨*block*⟩
| ‘`func`’ ‘`(`’ ⟨*funArgs*⟩ ‘`)`’ ‘`=`’ ⟨*block*⟩

⟨*funArgs*⟩ ::= ⟨*typedVal*⟩ ‘`,`’ ⟨*typedVal*⟩
| ⟨*typedVal*⟩

⟨*expr*⟩ ::= ⟨*if-expr*⟩
| ⟨*for-compr*⟩
| ⟨*match-expr*⟩
| ⟨*ask-stmt*⟩
| ⟨*identifier*⟩
| ⟨*funCall*⟩
| ⟨*fieldCall*⟩
| ⟨*metodCall*⟩
| ⟨*literal*⟩

Message-passing to actors happens through the tell and ask commands. Tell simply sends a message to an actor without expecting anything in return, while when ask sends a message to an actor it expects some kind of a result or something in return.

⟨*tell-stmt*⟩ ::= ‘`tell`’ ⟨*ne-args*⟩ ‘`about`’ ⟨*ne-args*⟩ ‘`;`’

⟨*ask-stmt*⟩ ::= ‘`ask`’ ⟨*ne-args*⟩ ‘`about`’ ⟨*ne-args*⟩ ‘`;`’

As is inferred by the way the tell and ask statements are written, the way in which their actions are performed and what they expect in return is naturally understood. Making it more intuitively for the user to use it.

Control Structures

The general style across control structures in Quantum is made to be as similar to each other as possible, to improve the read/write-ability.

An if-expression works much like C# or Java if-statement, except it does not have a traditional *else* branch, instead all possible if-branches are written in full within the if-block

⟨*if-expr*⟩ ::= '`if`' ⟨*if-block*⟩

⟨*if-block*⟩ ::= '`{`' ⟨*if-stmts*⟩ '`}`'
| ⟨*if-stmt*⟩

⟨*if-stmts*⟩ ::= ⟨*if-stmt*⟩ ⟨*if-stmts*⟩
| ⟨*if-stmt*⟩

⟨*if-stmt*⟩ ::= ⟨*stmt*⟩ '`then`' ⟨*expr*⟩

A match-expression is like the switch from C#, except it matches the structure of data, unlike the if-expression, the match-expression works on things that are not just booleans.

⟨*match-expr*⟩ ::= '`match`' '`(`' ⟨*expr*⟩ '`)`' ⟨*match-block*⟩

⟨*match-block*⟩ ::= '`{`' ⟨*match-stmts*⟩ '`}`'
| ⟨*match-stmt*⟩

⟨*match-stmts*⟩ ::= ⟨*match-stmt*⟩ ⟨*match-stmts*⟩
| ⟨*match-stmt*⟩

⟨*match-stmt*⟩ ::= ⟨*patternDef*⟩ '`then`' ⟨*expr*⟩

A for-comprehension works much like the for-each loops in c#, except it allows multiple collections to iterate over the nested loop.

⟨*for-compr*⟩ ::= '`for`' ⟨*for-block*⟩ '`do`' ⟨*block*⟩
| '`for`' ⟨*for-block*⟩ '`yield`' ⟨*block*⟩

⟨*for-block*⟩ ::= '`{`' ⟨*for-stmts*⟩ '`}`'
| ⟨*for-stmt*⟩ '`;`'

⟨*for-stmts*⟩ ::= ⟨*for-stmt*⟩ ⟨*for-stmts*⟩
| ⟨*for-stmt*⟩

⟨*for-stmt*⟩ ::= ⟨*identifier*⟩ '`in`' ⟨*expr*⟩

Literals

Defines all the literals in the language.

⟨*list*⟩ ::= '[' ⟨*args*⟩ ']'

⟨*ne-args*⟩ ::= ⟨*expr*⟩ ',' ⟨*ne-args*⟩
| ⟨*expr*⟩

⟨*args*⟩ ::= ⟨*ne-args*⟩
| ϵ

⟨*dec-lit*⟩ ::= ⟨*num-lit*⟩ '.' ⟨*num-lit*⟩

⟨*literal*⟩ ::= ⟨*string-lit*⟩
| ⟨*num-lit*⟩
| ⟨*dec-lit*⟩
| ⟨*atom-def*⟩
| ⟨*atom-def*⟩ '(' ⟨*args*⟩ ')'
| ⟨*list*⟩

4.2 Semantics

In this chapter the general semantic methods used in the semantic analysis will be defined.

4.2.1 Semantic Method

For this project Structural Operational Semantics have been used to formally define the meaning of a writing in the language. When describing a language using structural operational semantics there are two methods known as small- and big-step semantics that can be used, where small-step is the best to describe the problems concerned with parallelism, as in big-step semantics there is no way of describing deadlocks and other known problems in parallel programming [56].

The uninteresting transitions are similar to the ones from the *Transition and Trees* book [56], consisting of only one major difference. These semantics, formation rules, and transition rules can be viewed in *Transition and Trees*[56], where the statements are described.

For one of the more interesting transitions in Quantum is the if-statements. the CFG for If-statements in Quantum can be seen in [Figure 4.5]:

The *if-expr* iterates through all the results of all the *if-stmts* in the *if-block*. The

⟨*if-expr*⟩ ::= '`if`' ⟨*if-block*⟩

⟨*if-block*⟩ ::= '`{`' ⟨*if-stmts*⟩ '`}`'
| ⟨*if-stmt*⟩

⟨*if-stmts*⟩ ::= ⟨*if-stmt*⟩ ⟨*if-stmts*⟩
| ⟨*if-stmt*⟩

⟨*if-stmt*⟩ ::= ⟨*stmt*⟩ '`then`' ⟨*expr*⟩

⟨*stmt*⟩ ::= ⟨*expr*⟩ '`;`'

⟨*expr*⟩ ::= ⟨*if-expr*⟩
| ...
| ⟨*literal*⟩

Figure 4.5: Grammer for If-statements

if-stmts is recursively calling itself until it meets an end where it is only a single *if-stmt*.

This semantics will be checked in an evaluation context. The environment-store model will contain the environments for variables and procedures denoted env_v for the variable environment store-model and env_p for the procedure environment store-model. On a run-time stack we need to keep track of the bindings that are in effect, this can be achieved by introducing the run-time stack env_l, which is a list of pairs of (env_v, env_p), the list of pairs denotes all lists whose elements belong to the set $env_v \times env_p$. The set of stores is denoted *sto* The transition for this system will in all cases be a bit alike to what is know from the *C*-family. The semantics for these if-statements configuration will be:

- $\langle < if - stmt >, sto, envl \rangle$, which is an intermediate configuration, where the *if-stmt* may contain a boolean value.
- $\langle sto, env_l$, which is a terminal configuration

The transition rules for a skip statement can be seen in [Figure 4.6] The transition rules for the if-statements can be seen in [Figure 4.7].

$$[\text{skip}]\langle skip, sto, env_l\rangle \Rightarrow (sto, envl)$$

Figure 4.6: Skip Statement

$$\text{If-True}\frac{\langle if < stmt > then < expr >, sto, env_l\rangle \Rightarrow \langle < stmt >, sto, env_l\rangle}{if\ env_v, sto \vdash < stmt > \rightarrow_{<stmt>} tt\ where\ env_l = (env_v, env_p) : env_l'}$$

$$\text{If-False}\frac{\langle if < stmt > then < expr >, sto, env_l\rangle \Rightarrow \langle skip, sto, env_l\rangle}{if\ env_v, sto \vdash < stmt > \rightarrow_{<stmt>} ff\ where\ env_l = (env_v, env_p) : env_l'}$$

Figure 4.7: If-Statement

Chapter 5

Parser

The parser is responsible for creating a syntax tree from a set of rules given an input of tokens typically generated by a lexer[1]. The syntax tree is then used to determine various things about the program, such as scope rules, the layout of various data structures, etc.[57].

5.1 Recursive Descent Parser

A recursive descent parser consists of several mutually recursive procedures, which work together to parse an input string[57].

5.1.1 Parser Generator

This is often done with tools that directly translate a CFG to a corresponding recursive descent parser. Such a tool is usually called a *parser generator* or *compiler-compiler*. The downside of a parser generator is the need for an external tool removed from the language in which the bulk of the work on the parse tree will be performed. The upshot on the other hand is the flexibility. An external tool, if constructed correctly, could generate parsers for many different languages, giving the user ample flexibility[57].

[1] also called a scanner or tokeniser

5.1.2 Parser Combinator

The parser combinators allow creating recursive descent parsers by way of combining the recursive procedures, letting the user create larger parsers by piecing smaller ones together. Unlike parser generators, they are also typically made as a library for the language in which it's used; as a result it's then possible to utilise all the strengths and utilities of the language in which it's used, such as putting parsers into collection, or specifying procedures that output different parsers depending on their input. This of course also comes with the downside of having to adhere to the given language's limitations when it comes to doing various things[58].

5.2 Implementation

5.2.1 Scala's Parser Combinator

The Scala programming language has a community maintained parser combinator library available for download. It makes use of Scala's rich and scalable syntax to allow a Domain-Specific Language (DSL) capable of representing an EBNF of the CFG within Scala itself, without any need of external tools or software. The parser combinator library is flexible enough to allow for anything as the output type, whether it be Scala's native types, plain text, or nodes in a parse tree.

A Brief Introduction to the Syntax The following operators are necessary to know to understand the code:

$\sim$ means *followed by*. So while in standard BNF you might have something like

```
1 Operand + Operand
```

2 In Scala you'd have to write

```
1 Operand ~ "+" ~ Operand
```

The $\sim$ denoting the separation between the terminals and nonterminals. $\sim>$ and $<\sim$ are variations of $\sim$, and have to do with simplifying pattern matching later.

$\wedge\wedge$ means *if the pattern holds, do the following*. So say one needs to match a number, and then return it as an integer. That would be done as so:

```
1 "[0-9]+".r ^^ {str => str.toInt}
```

Where `".r"` means *turn string into regular expression*. The above example states, that if the pattern `[0-9]+` matches, the string is then parsed to and returned as an integer. Note that `Foo ^^ Bar` is shorthand for `Foo ^^ {x =>Bar(x)}`.

$\wedge\wedge\wedge$ is similar to $\wedge\wedge$, except rather than doing something, it simply returns a value

```
"Hola" ^^^ "Hello"
```

If "Hola" occurs it'll be parsed to "Hello". In other words, $\wedge\wedge$ expects a function, $\wedge\wedge\wedge$ does not.

`rep()` matches a sequence repeated 0 or more times, it's equivalent to the regular expression `*`.

`rep1()` matches a sequence 1 or more times, and is equivalent to `+`. Both `rep()` and `rep1()` return a list.

`opt()` matches 0 or 1 instance of the input sequence, and is equivalent to the regex `?`.

`|` is the alternative, it works just like in BNF.

`positioned()` takes care of recording line and column numbers, saving them in a *pos* field.

If a parser[2] lacks a $\wedge\wedge\wedge$ or $\wedge\wedge$, it simply means it doesn't do anything in particular, this is useful for cases where something is defined as many other things. Compare the following BNF to its equivalent code:

```
Literal ::= String | Number | Atom | List
```

```
def literal = string | number | atom | list
```

It doesn't need to return anything in particular, as it's just a nonterminal that groups other terminals and nonterminals without performing any operations on them.

5.2.2 Parser Implementation

Nonterminals, and a few terminals in the code are represented with what Scala calls *case classes*. Case classes, are a special subset of classes that behave like functions, are able to be pattern-matched on, and come with a free string representation of themselves. This makes them ideal for creating data structures that can easily be iterated over, manipulated, and printed. The following section makes heavy use of these case classes; `IfExpression`, `MatchExpression`, and `ForComprehension` are examples of

[2]Parser here means parser method in the context of a parser combinator

these. The following section also takes a look at how selected bits of the parser are implemented, in this case, the control structures *if*, *match*, and *for*.

Listing 5.1: If Expression

```
def ifExpr: Parser[IfExpression] =
  positioned("if" ~> ifBlock ^^ IfExpression)

def ifBlock: Parser[List[IfStatement]] =
  "{" ~> rep1(ifStmt <~ ";") <~ "}" | ifStmt ^^ {
    stmt => List(stmt)
  }

def ifStmt: Parser[IfStatement] = stmt ~ "then" ~ expr ^^ {
  case boolExpr ~ "then" ~ action =>
    IfStatement(boolExpr, action)
}
```

Following the BNF closely, the if-expression is split up into three constituent parts, the *ifExpr*, the *ifBlock*, and the *ifStmt*. An if-expression only contains an if-block, but the if-block comes in two different forms; one capable of expressing an arbitrary number of if-statements surrounded by curly braces, and one only capable of expressing a single one not surrounded by anything. This is done to allow simpler one-liner if-statements without any need for extra typing.
Compare for example `if a < b then io.format("Hello");`
with `if { a < b then io.format("Hello"); };`. This creates an interesting problem: because of the fact that an if-block needs to return a *list* of if-statements, and not just an if-statement, the case without curly braces therefore needs to be wrapped in a `List` instance; this wrapping happens automatically when using `rep1` in the other case. The if-statement is an interesting case. It showcases Scala's ability to treat any lambda expression as a pattern match, allowing the user to operate on just the patterns necessary.

Listing 5.2: Match Expression

```
def matchExpr: Parser[MatchExpression] =
  positioned("match" ~ "(" ~> expr ~ ")" ~ matchBlock ^^ {
    case expr ~ ")" ~ block =>
      MatchExpression(expr, block)
  })

def matchBlock: Parser[List[MatchStatement]] =
  "{" ~> rep1(matchStmt <~ ";") <~ "}" | matchStmt ^^ {
    statement => List(statement)
  }
```

```
11
12 def matchStmt: Parser[MatchStatement] =
13   patternDef ~ "then" ~ expr ^^ {
14     case pattern ~ "then" ~ expression =>
15       MatchStatement(pattern, expression)
16   }
```

The match-expression differs from the if-expression in that it also requires an input parameter; it also gives us our first real example of how ~ and ~> differ. ~> ignores everything left of it in the pattern match, which simplifies the matching, without it matchExpr would have needed to be written like this:

Listing 5.3: matchExpr with full pattern

```
1 def matchExpr: Parser[MatchExpression] =
2   positioned("match" ~ "(" ~ expr ~ ")" ~ matchBlock ^^ {
3     case "match" ~ "(" ~ expr ~ ")" ~ block =>
4       MatchExpression(expr, block)
5   })
```

This includes two unused terminals in the case expression that could just as easily have been avoided. This isn't without its downfalls though, as the ")" terminal needs to remain, since adding a <~ would also exclude the required *matchBlock*.

Listing 5.4: For Comprehension

```
1 def forCompr: Parser[ForComprehension] =
2   positioned("for" ~> forBlock ~ ("do" | "yield") ~ block ^^ {
3     case forBlock ~ "do" ~ block =>
4       ForComprehension(forBlock, Left(Do), block)
5     case forBlock ~ "yield" ~ block =>
6       ForComprehension(forBlock, Right(Yield), block)
7   })
8
9 def forBlock: Parser[List[ForStatement]] =
10   "{" ~>  rep1(forStmt <~ ";") <~ "}" | forStmt ^^ {
11     statement => List(statement)
12   }
13
14 def forStmt: Parser[ForStatement] =
15   identifier ~ "in" ~ expr  ^^ {
16     case id ~ "in" ~ expr => ForStatement(id, expr)
17   }
```

The for-comprehensions exist in two different flavours, the *do*, and the *yield* variant,

here represented as a single parser with an enclosed alternative[3]. Because of this alternative, the expression following ∧∧ now has two different cases, one for *do*, and one for *yield*, this creates slightly different nodes in the resulting parse tree, where one holds a `Left(Do)`, and the other holds a `Right(Yield)`[4].

5.3 Parser Test

In order to test the parser for Quantum a series of smaller programs has been run through. The purpose of this being validating the Abstract Syntax Tree (AST) being generated during the parsing process. An example of these can be found in the following listing 5.5

Listing 5.5: One of the ran programs generating the for block AST

```
for {
  i in K;
  j in F;
} yield {foo;};
```

The piece of code in 5.5 generated the AST in [Figure 5.2] which was then validated to ensure the parser followed the wanted pattern.

Below in the figure there is an example of how a generalised AST for any program in Quantum can contain. There will always be a ModuleName, and a list containing a series of modules, ReceiverDefinitions, and ActorDefinitions. The example below in [Figure 5.1] is what was generated by the test program contanign the for comprehension.

A specification of the "Block" node follows in [Figure 5.2] of cause still referring to the test program for for comprehensions.

Aside from running these test cases, a black box unit test was also done for the parser. In listing 5.3 an example of these tests can be found for parsing Number Literals.

```
  it should "parse a string as a valid numberliteral" in {
    val input = "1234"
    val expectation = NumberLiteral("1234")
```

[3]Note that everything between ∼ and | are parsers, meaning (“do”|“yield”) itself is a parser made up of two other parsers. Hence the term parser *combinator*

[4]Right and Left are part of a built-in type called *Either*, which lets a variable be one of two different types, the one caveat is that they need to be wrapped in a Left or a Right to be considered valid. For example, Either[String, Int] can be either Left("some string"), or Right(42) (or any other number).

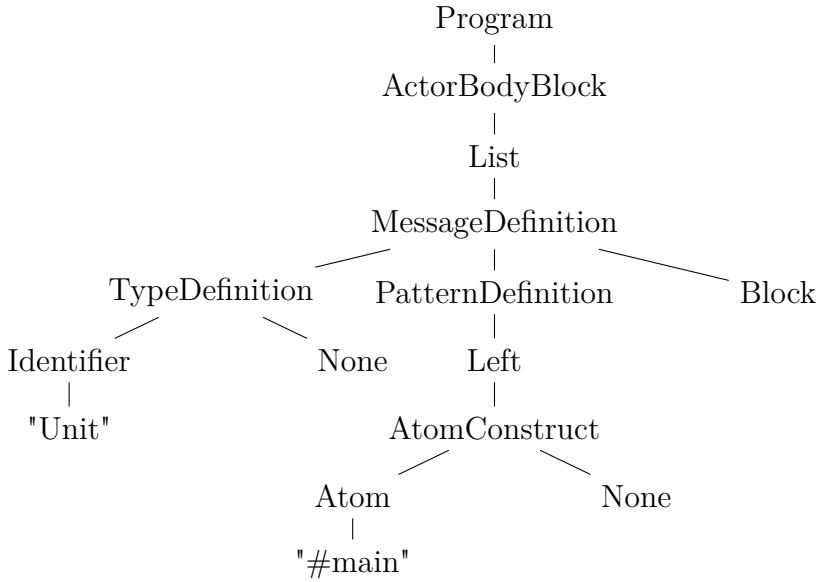


Figure 5.1: The generel AST in the beginning of a Quantum program

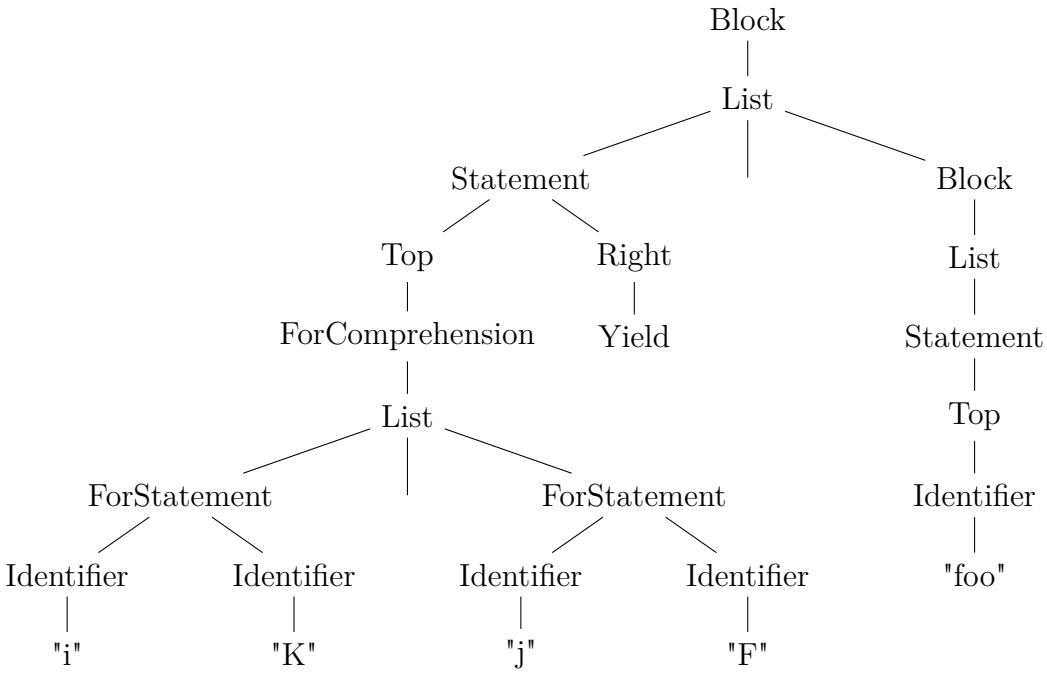


Figure 5.2: The For Block AST genereted by the sample program in 5.5

```
4
5     val result = NParser.parse(NParser.numberLiteral, input)
```

```
6       result should be (expectation)
7     }
```

Besides the parser, some black box testing was also preformed for the tokenizer to verify the tokens generated when keywords were met were correct.

Testing the correctness of the compiler in general is of course also something that has been done. In order to do this the output code from the code generator was validated, and run through a compiler for the intermediary language thus validating that the program written in Quantum acts as expected.

Chapter 6

Discussion

During the duration of this project some choices were made that impacted the outcome of the language. In this chapter some of these choices will be discussed and argued for.

6.1 Parallelism

In order to start the project a very important question had to be answered. What should it be about and why should that be the subject. The group decided on a focus point of parallel programming and that lead to Quantums creation. The reasoning behind choosing this subject as the focus of the project were twofold.
For one the subject interested the group members. The notion of being able to compute several pieces of code simultaneously in different cores would mean a significant upgrade til computational speeds of many algorithms if some of the challenges behind the idea could be avoided.
Secondly the group members found it to be a niche many native languages cannot support. This made for further research into parallelism and concurrency. While researching the group found other languages that could handle some of the problems regarding parallelism. Some of these are as previously mentioned deadlocks as well as race conditions. One of these languages was Bogdan/Björn's Erlang Abstract Machine (BEAM) which the group decided would be the target for the compiler.

6.2 Bogdan/Björn's Erlang Abstract Machine

BEAM allows mixing thread code emulation with compiling into C[59].
By reading the Erlang BEAM virtual machine specification[59], it is visible that BEAM guarantees that each process has its own heap and stack. The BEAM virtual machine has its own scheduler to keep track of active and inactive processes[59].
In BEAM each process has its own message queue, placed locally for the process[59]. The scheduler swaps processes between active and inactive through messages received[59], making this ideal for Message-Passing systems. Elixir uses the Erlang VM BEAM, which is known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain[60]

6.3 Elixir

Elixir is a dynamic, functional concurrent programming language designed for building scalable and maintainable applications, and was created with the OO mindset and based on the functional paradigm language Erlang, making it an multi-paradigm language. This makes it a good compilation step before compiling to BEAM byte code, as it can be complicated to compile to byte code directly.
The downside of choosing a target language like Elixir it the fact that there was no available help for the group to be had. This meant that understanding the language well enough to make the Quantum compiler generate code that could then be compiled by the Elixir compiler was up to the group and the group alone.
In the end the group found that the positives of Elixir and BEAM were greater than the difficulties there could arise from having to learn the language.

6.4 Parser Combinator

When the time came to look into creating a parser for the Quantum compiler the choice fell on using a parser combinator. As previously mentioned this meant that the group ended up being forced into one language, and writing the whole parser manually. The Reasoning behind this choice, though it seems a bad idea, was the modularity of the parser and the fact that most things the group needed to create the parser was available in one library.
This choice also removed the opportunity of using a parser generator to help create the parser in a more swiftly manor.
In the end the group found that Scala's library for parser combinators were the better

choice for the compiler as it would both help further the understanding of the parser due to the fact that it was manually written. And also the built in features made it easier to generate the needed grammars directly in the programming language without having to use any additional tools.

6.5 Syntactic Choices

Regarding the syntax of Quantum an abundance of choices were made in order to ensure readability, and writability. These decisions, to mention some range from data types and data structures to punctuation when ending a statement. The overall outcome of the choices was decided that it should be focused more on the readability aspect instead of the writeability. This decision was made due to the fact that in todays world of computer science it is not the time it takes to write a program that is most expensive, but rather the time it takes to maintain. However the writability cannot be sacrificed completely as the language would become very much a niche that not many programmers would like to use.

6.6 Reliability

When evaluating the compiler looking at the output code is an essential step. Therefore it is necessary to compile the intermediate language with its own compilers. This lead the group to the conclusion, that the Quantum compiler is not particularly reliable in its current state. It has very specific requirements regarding how to write the code in Quantum or it will give errors when trying to compile either the Quantum code or the Elixir output.

Chapter 7

Conclusion

As the project is coming to an end it is time to look at the degree of completion and in what manor the problem statement has been fulfilled.

To evaluate whether or not the problem statement has been fulfilled is a hard process as there has been conducted no user tests. This means that an objective assessment is impossible for the group to produce. The problem statement wrote:

Could multi-core processing become a more streamlined experience by implementing the actor model in a manner invisible to the developer?

In Quantum the actor model has been hidden behind regular classes meaning that in order to create actors one would simply create and instantiate a new object from a class which lies very close to the general idea behind many of the OOP languages. This means that making the actor model invisible to the developer has been completed. Whether or not this has made the experience more streamlined is, as stated, impossible to say.

Chapter 8

Reflection

During the process of project some problems has been encountered. To mention some of them user testing would, seen in hind sight, have been a great idea in order to give the group a basis for assessing the problem statements completion. The management of time on hand was not satisfying. In general a time table was completely absent which would probably been a good idea to add in order to manage time.
In the future using time tables or backlogs is a thing that the group can agree on is a necessary step to manage the work and avoiding falling far behind on the work. Being firmer when enacting the group contract is also something that has to be done regarding missing work and, or members.
Beginning the project the group did great work using peer writing to make headway regarding the report. A lot of work were completed quickly. However the peer writing died out as members fell out of the group due to various reasons.
The dividing of work and roles in the group is another thing that needs adjusting in the future. This is another area where the missing group members impacted the planed approach. Starting out the plan was to give on member a role that has to be kept throughout the duration of the project. The roles began shifting leading to a bit of chaos and confusion.
The workload was also planned quite differently from how it turned out. The plan was to have the report lay dormant as the group focused on the compiler and then added the implementation passages of the report as parts of the compiler were finished.

Bibliography

[1] Pär Persson Mattsson. Why haven't cpu clock speeds increased in the last few years, 2014. URL https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years/.

[2] Gigabyte. Overclocking world records broken at gigabyte ces 2014 extreme oc event, 2014. URL http://www.gigabyte.com/press-center/news-page.aspx?nid=1267.

[3] Amir Ashraf Kamil. *Single Program, Multiple Data Programming for Hierarchical Computations*. PhD thesis, University of California, Berkeley, 2012. URL http://web.eecs.umich.edu/~akamil/papers/thesis.pdf.

[4] Compare intel products, 2016. URL http://ark.intel.com/compare/84679,84678,84677,84676,84688,84686,84685,84684,84683,84682,84681,84680.

[5] Michael Garland Kevin Skadron John Nickolls, Ian Buck. Scalable parallel programming with cuda, 2008.

[6] Blaise Barney. Introduction to parallel computing, 2015. URL https://computing.llnl.gov/tutorials/parallel_comp/#Whatis.

[7] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley, March 2010.

[8] Wen mei Hwu. Three challenges in parallel programming, 2012. URL http://parallel.illinois.edu/blog/three-challenges-parallel-programming.

[9] Tiobe index, March 2016. URL http://www.tiobe.com/tiobe_index.

[10] M. F. Sanner. Python: A programming langauge for software integration and development, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.6459&rep=rep1&type=pdf.

[11] RedMonk. Redmonk diagnistics. Website, 2015. URL http://redmonk.com.

[12] Computational Science Education Project. Flynn's taxonomy. Website, 1994. URL https://www.phy.ornl.gov/csep/ca/node11.html.

[13] Michael Flynn. Flynn's taxonomy. Website, 2011. URL http://link.springer.com/referenceworkentry/10.1007%2F978-0-387-09766-4_2#CR2_2.

[14] Computational Science Education Project. Sisd computers. Website, 1994. URL https://www.phy.ornl.gov/csep/ca/node12.html.

[15] Flynn's taxonomy. Website, 2003. URL https://en.wikipedia.org/wiki/Flynn%27s_taxonomy.

[16] Computational Science Education Project. Simd computers. Website, 1994. URL https://www.phy.ornl.gov/csep/ca/node13.html.

[17] *Parallel processing in processor organization*, January 2014. URL http://www.ijarcce.com/upload/2014/january/IJARCCE6G__s_prabhudev_parallel.pdf.

[18] Rajkumar Buyya. *The Design of PARAS Microkernel*. millenium, June 2000. URL http://www.buyya.com/microkernel/.

[19] Faith E. Fich. The complexity of computation on the parallel random access machine. Master's thesis, University of Toronto, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.2932&rep=rep1&type=pdf.

[20] Arvind Krishnarmurthy. Pram algorithms, 2004. URL http://homes.cs.washington.edu/~arvind/cs424/notes/l2-6.pdf.

[21] Cyril Zeller. *CUDA C/C++ Basics Supercomputing 2011 Tutorial*. NVIDIA Corporation, 2011. URL http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.

[22] Nvidia gtx980ti specification sheet, 2015. URL http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications.

[23] Borko Furht, editor. *Encyclopedia of Multimedia*, chapter SIMD (Single Instruction Multiple Data Processing), pages 817–819. Springer US, Boston, MA, 2008. ISBN 978-0-387-78414-4. doi: 10.1007/978-0-387-78414-4_220. URL http://dx.doi.org/10.1007/978-0-387-78414-4_220.

[24] *Approaches for Integrating Task and Data Parallelism*, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.3811&rep=rep1&type=pdf.

[25] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice Hall, 7th edition edition, 2012.

[26] Rajesh Karmani. Actorfoundry. Website, 2012. URL http://osl.cs.illinois.edu/software/actor-foundry/.

[27] *Actors*, 2011. URL http://web.cs.ucla.edu/~palsberg/course/cs239/papers/karmani-agha.pdf.

[28] Prasun Dewan. Synchronous vs asynchronous. Website, February 2006. URL http://www.cs.unc.edu/~dewan/242/s07/notes/ipc/node9.html.

[29] *Comparison of Erlang Runtime System and Java Virtual Machine*, May 2015. URL http://ds.cs.ut.ee/courses/course-files/To303nis%20Pool%20.pdf.

[30] Queues and message-passing. Website, 2016. URL http://web.mit.edu/6.005/www/sp16/classes/22-queues/.

[31] Ray Toal. Message passing. Website, 2016. URL http://cs.lmu.edu/~ray/notes/messagepassing/.

[32] Intel. What is a microprocessor? Website, 2011. URL http://download.intel.com/newsroom/kits/40thanniversary/pdfs/What_is_a_Microprocessor.pdf.

[33] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 2012. URL https://mathematics.abu.edu.ng/staff/aliyu-salisu/cosc311/Concept%20of%20Programming%20Languages%2010th%20Edition.pdf.

[34] Roman Dementiev & Patrick Fay Thomas Willhalm. Intel performance counter monitor - a better way to measure cpu utilization. Website, April 2016. URL https://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization.

[35] Enoch O. Hwang. *Digital Logic and Microprocessor Design with VHDL*. Thomson, first edition, 2006. URL http://faculty.lasierra.edu/~ehwang/digitaldesign/.

[36] *History and Evoltuion of GPU Architecture*, 2010. URL http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf.

[37] Jatin Chhugani Michael Deisher Daehyun Kim Anthony D. Nguyen Nadathur Satish Mikhail Smelyanskiy Srinivas Chennupaty Per Hammarlund Ronak Singhal Victor W Lee, Changkyu Kim and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. Website, June 2010. URL http://www.hwsw.hu/kepek/hirek/2010/06/p451-lee.pdf.

[38] Ashu Rege. An introduction to modern gpu architecture. Website, 2008. URL http://http.download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf.

[39] Carl Hewitt. Actor model of computation: Scalable robust information systems, 2010. URL http://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf.

[40] *Actors and continuous funtionals*, December 1977. M.I.T. Artificial Intelligence Laboratory. URL http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-194.pdf.

[41] *Communicating Sequential Processes*, August 1978. URL http://spinroot.com/courses/summer/Papers/hoare_1978.pdf.

[42] Floating-point types table (c# reference. Website, 2016. URL https://msdn.microsoft.com/da-dk/library/9ahet949.aspx.

[43] decimal (c# reference. Website, 2016. URL https://msdn.microsoft.com/da-dk/library/364x0z75.aspx.

[44] Robert Virding. Twitter conversation, 2016. see pictures in appendix A.

[45] Thomas Petricek and Jon Skeet. *Real-World Functional Programming: With Examples in F# and C#*. Manning Publications, first edition, January 2010.

[46] Arrays. Website, 1995. URL https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

[47] Interface list<e>. Website, 1993. URL https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

[48] Microsoft. struct (c# reference). Website, 2016. URL https://msdn.microsoft.com/da-dk/library/ah19swz4.aspx.

[49] Bradly Kjell. Contorl structures. Website, 2006. URL https://chortle.ccsu.edu/QBasic/chapter15/bc15_2.html.

[50] Carol Zander. Function pass by value v. pass by reference. Website, 2006. URL http://courses.washington.edu/css342/zander/css332/passby.html.

[51] Oracle. Dynamic typing vs. static typing. Website, March 2015. URL https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html.

[52] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[53] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work? Website, August 2008. URL http://www.garshol.priv.no/download/text/bnf.html.

[54] Michael O'Neill & Conor Ryan. *Automatic Generation of Caching Algorithms*. PhD thesis, Dept. Of Computer Science And Information Systems, University of Limerick, Ireland, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.769&rep=rep1&type=pdf.

[55] *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*, March 2003. URL https://www.researchgate.net/profile/Ivan_Porres/publication/2896192_A_Relation_Between_Context-Free_Grammars_and_Meta_Object_Facility_Metamodels/links/55dc45fc08aed6a199ac9192.pdf.

[56] Hans Hüttel. *Transition and Trees - An Introduction to Structural Operational Semantics*. Cambridge University Press, first edition, 2010.

[57] Ronald Cryton Charles Fischer, Richard LeBlanc. *Crafting a Compiler*. Pearson, 1988.

[58] Graham Hutton. Higher-order functions for parsing, 1992. URL http://www.cs.nott.ac.uk/~pszgmh/parsing.pdf.

[59] Bogumil Hausman. The erlang beam virtual machine specification. Website, October 1997. URL http://www.cs-lab.org/historical_beam_instruction_set.html.

[60] Elixir. Website, 2012. URL http://elixir-lang.org.

Glossary

SIMD Single Instruction Multiple Data

CSP Communicating Sequential Processes

OS Operating System

MS-DOS Microsoft Disk Operating System

I/O Input/Output

CPU Central Processing Unit

OOP Object-Oriented Programming

OO Object-Oriented

AST Abstract Syntax Tree

MPMD Multiple Program Multiple Data

SPMD Single Program Multiple Data

SMP Synchronous-Message-Passing

AMP Asynchronous-Message-Passing

PRAM Parallel Random Access Machine

SISD Single Input Single Data

SIMD Single Input Multiple Data

MISD Multiple Input Single Data

MIMD Multiple Input Multiple Data

ALU Arithmetic Logic Unit

IC Integrated Circuit

SoC System on a Chip

ASIC Application-Specific Integrated Circuit

GPU Graphics Processing Unit

BEAM Bogdan/Björn's Erlang Abstract Machine

IC Integrated Circuit

ASIC Application-Specific Integrated Circuit

CFG Context-Free Grammar

DSL Domain-Specific Language

BNF Backus-Naur Form

EBNF Extended Backus-Naur Form

ABNF Augmented Backus-Naur Form

Chapter 9

Appendix

Appendix A

Robert Twitter Conversation

The chat conversation with Robert Virding, co-designer of Erlang.

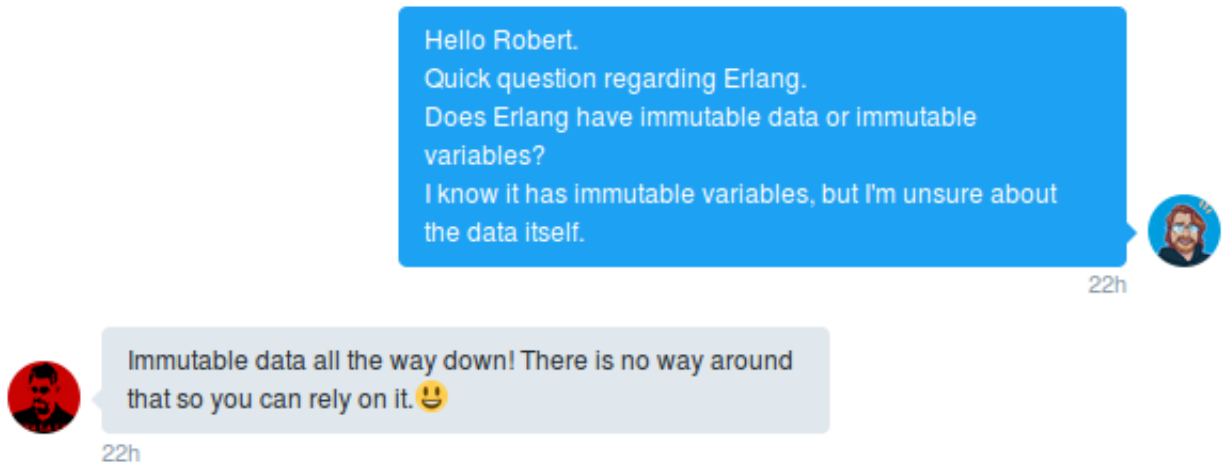


Figure A.1: First picture

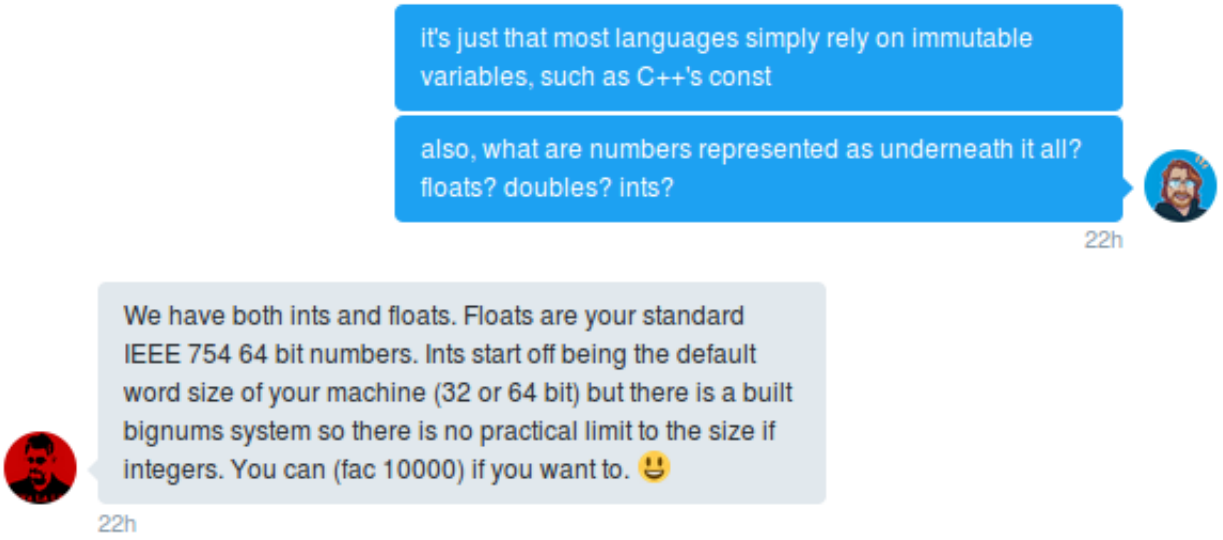


Figure A.2: Second picture

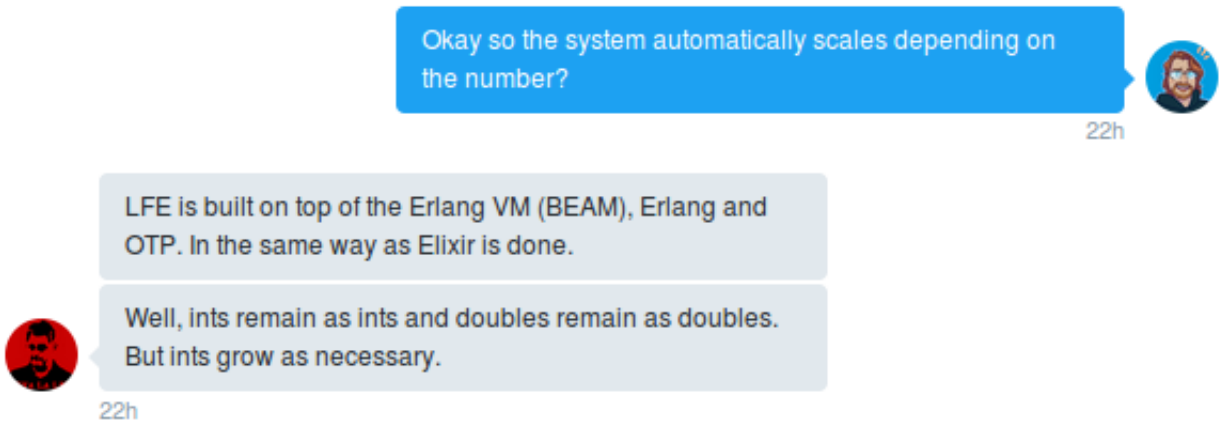


Figure A.3: Third picture

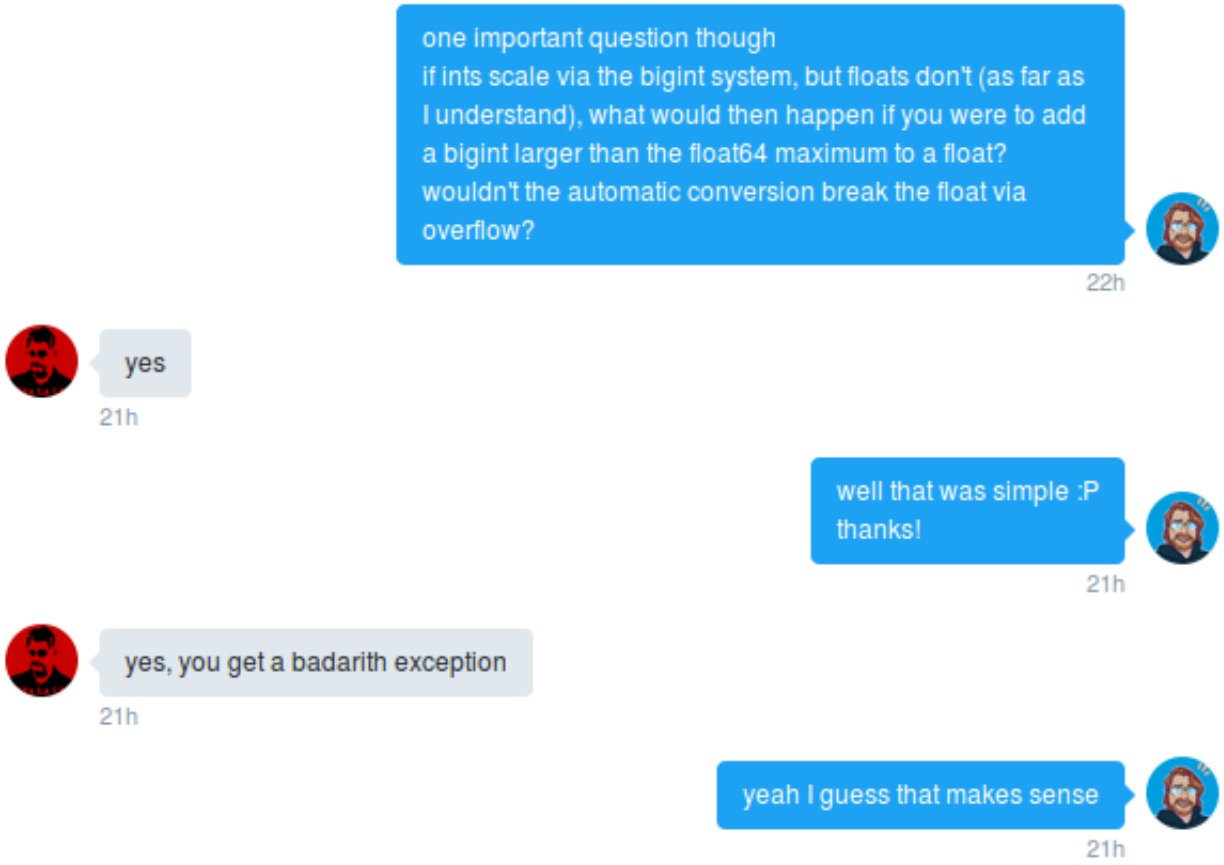
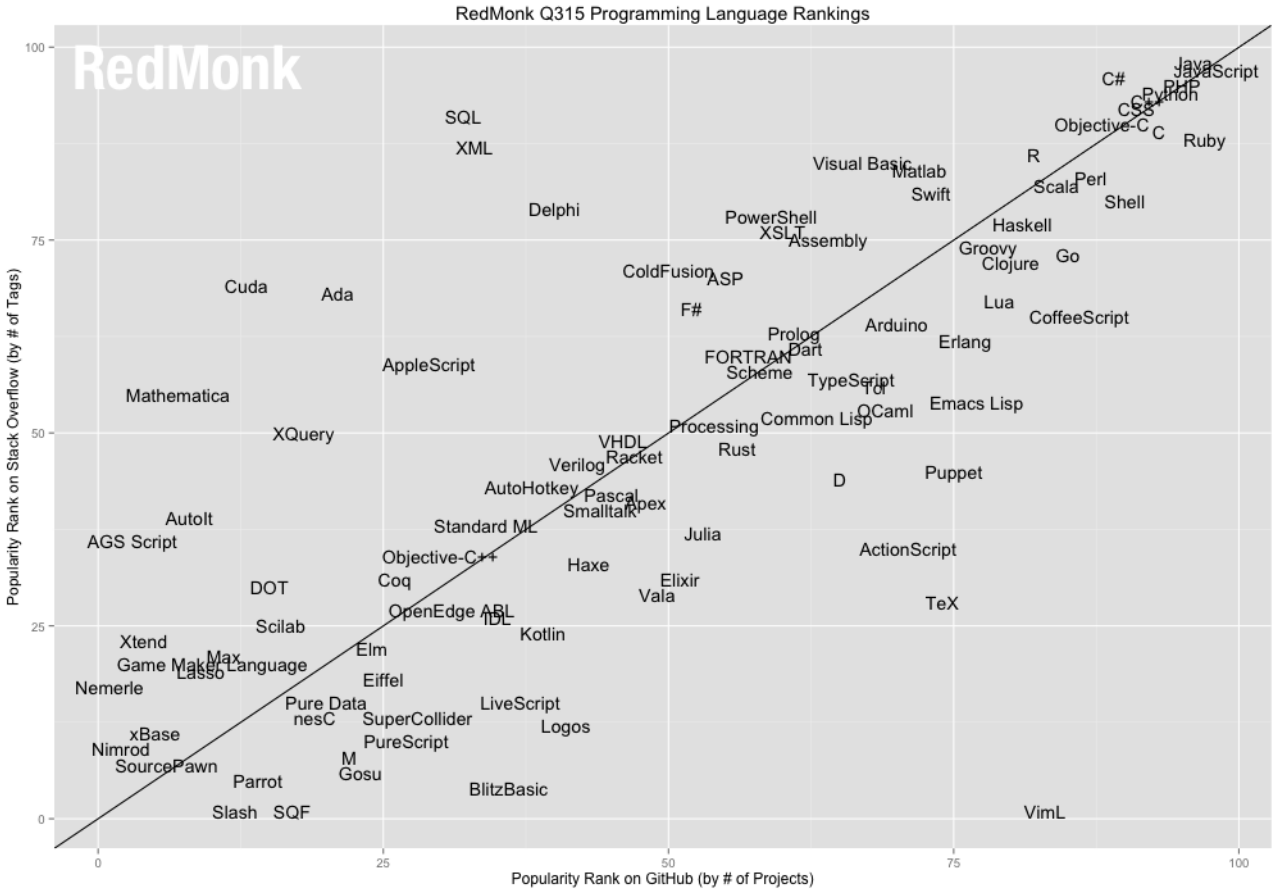


Figure A.4: Fourth picture

Appendix B

RedMonk popularity Graph


RedMonk Q315 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
100
SQL
XML
Delphi
Cuda
Ada
AppleScript
Mathematica
XQuery
AutoIt
AGS Script
DOT
Scilab
Xtend
Game Maker Language
Max
Lasso
Nemerle
xBase
Nimrod
SourcePawn
Parrot
Slash
SQF
Elm
Eiffel
Pure Data
nesC
SuperCollider
PureScript
M
Gosu
BlitzBasic
LiveScript
Logos
Kotlin
IDL
OpenEdge ABL
Objective-C++
Coq
Standard ML
AutoHotkey
Verilog
VHDL
Racket
Pascal
Apex
Smalltalk
Haxe
Julia
Elixir
Vala
Rust
Processing
Common Lisp
OCaml
Tcl
TypeScript
Scheme
FORTRAN
Dart
Prolog
F#
ASP
ColdFusion
Assembly
XSLT
PowerShell
Visual Basic
Matlab
Swift
R
Scala
Perl
Shell
Haskell
Groovy
Clojure
Go
Lua
CoffeeScript
Arduino
Erlang
Emacs Lisp
D
Puppet
ActionScript
TeX
VimL
Ruby
Objective-C
C
C#
JavaScript
Java
PHP
Python
CSS

Appendix C

Grammar in Backus Naur Form

⟨*program*⟩ ::= '`module`' ⟨*module-name*⟩ '`;`' ⟨*top-level-cons*⟩

⟨*module-name*⟩ ::= ⟨*identifier*⟩ '`.`' ⟨*module-name*⟩
| ⟨*identifier*⟩ '`;`'

⟨*top-level-cons*⟩ ::= ⟨*module-import*⟩ ⟨*top-level-cons*⟩
| ⟨*actor-def*⟩ ⟨*top-level-cons*⟩
| ⟨*data-struct-def*⟩ ⟨*top-level-cons*⟩
| ϵ

⟨*module-import*⟩ ::= '`import`' ⟨*module-name*⟩ '`;`'

⟨*actor-def*⟩ ::= '`class`' ⟨*type-def*⟩ ⟨*actor-body-block*⟩
| '`class`' ⟨*type-def*⟩ '`<-`' ⟨*type-defs*⟩ ⟨*actor-body-block*⟩
| '`object`' ⟨*type-def*⟩ ⟨*actor-body-block*⟩
| '`object`' ⟨*type-def*⟩ '`<-`' ⟨*type-defs*⟩ ⟨*actor-body-block*⟩

⟨*type-defs*⟩ ::= ⟨*type-def*⟩ '`,`' ⟨*type-defs*⟩
| ⟨*type-def*⟩

⟨*type-def*⟩ ::= ⟨*identifier*⟩ '`of`' ⟨*type-params*⟩
| ⟨*identifier*⟩

⟨*type-params*⟩ ::= ⟨*type-param*⟩ '`,`' ⟨*type-params*⟩
| ⟨*type-param*⟩

⟨*type-param*⟩ ::= '`(`' ⟨*type-def*⟩ '`)`'
| ⟨*identifier*⟩

⟨*string-lit*⟩ ::= '`"(\.|[^"])*"`'

⟨*identifier*⟩ ::= '`[a-zA-Z_][a-zA-Z0-9_'-]*`'

⟨*atom-def*⟩ ::= '`#[a-zA-Z_][a-zA-Z0-9_'-]*`'

⟨*num-lit*⟩ ::= '`[0-9]+`'

⟨*hex*⟩ ::= '`0x[0-9A-Fa-f]+`'

⟨*bin*⟩ ::= '`0b[01]+`'

⟨*actor-body-block*⟩ ::= '`{`' ⟨*actor-body-def*⟩ '`}`'
| ϵ

⟨*actor-body-def*⟩ ::= ⟨*message-def*⟩ ⟨*actor-body-def*⟩
| ⟨*message-def*⟩
| ϵ

⟨*message-def*⟩ ::= define ⟨*type-def*⟩ ⟨*pattern-def*⟩ '`=`' ⟨*block*⟩

⟨*pattern-def*⟩ ::= ⟨*literal*⟩
| '`(`' ⟨*typedVal*⟩ '`)`'

⟨*typedVal*⟩ ::= ⟨*type-def*⟩ ⟨*identifier*⟩

⟨*data-struct-def*⟩ ::= '`data`' ⟨*type-def*⟩ ⟨*data-body-block*⟩
| '`data`' ⟨*type-def*⟩ ⟨*data-body-block*⟩ '`<-`' ⟨*type-defs*⟩

⟨*data-body-block*⟩ ::= '`{`' ⟨*field-defs*⟩ '`}`'
| ϵ

⟨*field-defs*⟩ ::= ⟨*typedVal*⟩ '`;`' ⟨*field-defs*⟩
| ϵ

⟨*block*⟩ ::= '`{`' ⟨*stmts*⟩ '`}`'
| ⟨*stmt*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ⟨*stmts*⟩
| ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*expr*⟩ '`;`'
| ⟨*valDef*⟩ '`;`'
| ⟨*binaryOperation*⟩ '`;`'

⟨*valDef*⟩ ::= '`val`' ⟨*identifier*⟩ '`=`' ⟨*expr*⟩
| '`val`' ⟨*typedVal*⟩ '`=`' ⟨*expr*⟩

⟨*funDef*⟩ ::= '`func`' ⟨*identifier*⟩ '`(`' ⟨*funArgs*⟩ '`)`' '`=`' ⟨*block*⟩
| '`func`' '`(`' ⟨*funArgs*⟩ '`)`' '`=`' ⟨*block*⟩

⟨*funArgs*⟩ ::= ⟨*typedVal*⟩ '`,`' ⟨*typedVal*⟩
| ⟨*typedVal*⟩

⟨*expr*⟩ ::= ⟨*if-expr*⟩
| ⟨*for-compr*⟩
| ⟨*match-expr*⟩
| ⟨*ask-stmt*⟩
| ⟨*identifier*⟩
| ⟨*funCall*⟩
| ⟨*fieldCall*⟩
| ⟨*metodCall*⟩
| ⟨*literal*⟩

⟨*tell-stmt*⟩ ::= '`tell`' ⟨*ne-args*⟩ '`about`' ⟨*ne-args*⟩ '`;`'

⟨*ask-stmt*⟩ ::= '`ask`' ⟨*ne-args*⟩ '`about`' ⟨*ne-args*⟩ '`;`'

⟨*if-expr*⟩ ::= '`if`' ⟨*if-block*⟩

⟨*if-block*⟩ ::= '`{`' ⟨*if-stmts*⟩ '`}`'
| ⟨*if-stmt*⟩

⟨*if-stmts*⟩ ::= ⟨*if-stmt*⟩ ⟨*if-stmts*⟩
| ⟨*if-stmt*⟩

⟨*if-stmt*⟩ ::= ⟨*stmt*⟩ '`then`' ⟨*expr*⟩

⟨*match-expr*⟩ ::= '`match`' '`(`' ⟨*expr*⟩ '`)`' ⟨*match-block*⟩

⟨*match-block*⟩ ::= '`{`' ⟨*match-stmts*⟩ '`}`'
| ⟨*match-stmt*⟩

⟨*match-stmts*⟩ ::= ⟨*match-stmt*⟩ ⟨*match-stmts*⟩
| ⟨*match-stmt*⟩

⟨*match-stmt*⟩ ::= ⟨*patternDef*⟩ '`then`' ⟨*expr*⟩

⟨*for-compr*⟩ ::= '`for`' ⟨*for-block*⟩ '`do`' ⟨*block*⟩
| '`for`' ⟨*for-block*⟩ '`yield`' ⟨*block*⟩

⟨*for-block*⟩ ::= '`{`' ⟨*for-stmts*⟩ '`}`'
| ⟨*for-stmt*⟩ '`;`'

⟨*for-stmts*⟩ ::= ⟨*for-stmt*⟩ ⟨*for-stmts*⟩
| ⟨*for-stmt*⟩

⟨*for-stmt*⟩ ::= ⟨*identifier*⟩ '`in`' ⟨*expr*⟩

⟨*list*⟩ ::= '`[`' ⟨*args*⟩ '`]`'

⟨*ne-args*⟩ ::= ⟨*expr*⟩ '`,`' ⟨*ne-args*⟩
| ⟨*expr*⟩

⟨*args*⟩ ::= ⟨*ne-args*⟩
| ϵ

⟨*dec-lit*⟩ ::= ⟨*num-lit*⟩ '`.`' ⟨*num-lit*⟩

⟨*literal*⟩ ::= ⟨*string-lit*⟩
| ⟨*num-lit*⟩
| ⟨*dec-lit*⟩
| ⟨*atom-def*⟩
| ⟨*atom-def*⟩ '`(`' ⟨*args*⟩ '`)`'
| ⟨*list*⟩