

UiO • **Department of Informatics**
University of Oslo

A Programming Language for the Internet of Things

Magnus Åsrud
Master's Thesis Spring 2017



A Programming Language for the Internet of Things

Magnus Åsrud

2nd May 2017

Abstract

The Internet of Things (IoT) is becoming more integrated with our daily lives for each passing day. Some households are already equipped with smart devices which perform tasks such as monitoring the indoor climate to mowing the lawn. There exists open source frameworks designed for IoT which can be used by hobbyist and professionals for their own projects. However, many of those tools require the user to know programming languages such as C, Python and JavaScript. Users who do not know much programming and who would like to tinker with IoT may find it challenging to work with those tools.

In this thesis we'll look at how to design a programming language targeted specifically at IoT. The language is targeted at amateur programmers with the goal to make it easy to program any IoT device. We'll discuss the current limitations of the IoT hardware and how the language should work around them. Finally, we'll look at how to implement an interpreter for this language on a Raspberry Pi 3 Model B.

Contents

I	Background	13
1	Introduction	15
1.1	Impulse SmartCity	15
1.2	The Current Project	16
1.3	Acknowledgements	16
2	The Internet of Things	17
2.1	Internet of Things	17
2.1.1	What is the Internet of things?	17
2.2	Sensors	18
2.2.1	Energy efficiency	20
2.3	Communication	20
2.3.1	Bluetooth	20
2.3.2	WiFi	22
2.3.3	WiFi HaLow	23
3	Related Projects	25
3.1	Alternatives	25
3.1.1	Scriptr	25
3.1.2	Libelium Waspote Plug & Sense	25
3.1.3	Johnny-Five	26
3.1.4	Tessel 2	26
3.2	Evaluation	26
II	Design	27
4	Requirements for an IoT Programming Language	29
4.1	The Target Platform	29
4.2	The Target Audience	29
4.3	The Core Requirements	30
4.4	Hot Swap Scripts	30
4.5	Interpreting vs Compiling	30
4.5.1	Compiling	30
4.5.2	Interpretation	32
4.5.3	Evaluation	32
4.6	The Type System	32
4.6.1	Static Typing	32

4.6.2	Dynamic Typing	33
4.6.3	Inferred Typing	33
4.6.4	Evaluation	33
4.7	Event Oriented Programming	34
4.7.1	Handling Events	34
4.8	Security	34
4.9	Errors and Exceptions	35
4.10	Handling a Software Crash	35
4.10.1	Alternative 1: Reset	35
4.10.2	Alternative 2: Remove the script	36
4.10.3	Alternative 3: Ignore it	36
4.10.4	Alternative 4: Terminate	36
4.10.5	Alternative 5: Inform the user	37
4.10.6	Watchdog Daemon	37
4.11	Sensors and Actuators	37
4.12	The Standard Library	38
5	Designing Daspel	39
5.1	What is Daspel?	39
5.2	The syntax	39
5.3	Data types	39
5.4	Integer	40
5.4.1	Integer Syntax	40
5.5	Real	40
5.5.1	Real Syntax	41
5.6	Boolean	41
5.6.1	Why include boolean?	41
5.7	String	42
5.7.1	Unicode	42
5.7.2	Dynamic Size	42
5.7.3	Single Character	42
5.7.4	String Syntax	42
5.8	List	43
5.8.1	List Syntax	43
5.9	Nil the Error Type	43
5.10	Variables and Variable Declaration	43
5.10.1	Variable Declaration Ambiguity	44
5.10.2	Variable Declaration Part	44
5.10.3	Variables and Scopes	45
5.11	Scoping	45
5.11.1	Variable Scope	46
5.11.2	Library Scope	47
5.11.3	Global Scope	47
5.11.4	Function Scope	47
5.12	Operators	47
5.13	Statements	48
5.13.1	For Loop	49
5.13.2	While Loop	49

5.13.3	Conditionals aka. If-Else	49
5.13.4	Functions	49
5.14	Things That Did Not Make It To The Language	50
5.14.1	Variadic Function	50
5.14.2	Tuple	50
5.14.3	Type Annotation For Function Parameters	50
5.14.4	Data Type Methods	51
5.15	Unicode Syntax	51
 III Implementation		59
 6 Proof of Concept		61
6.1	The Goal of the Implementation	61
6.2	The Hardware	61
6.3	The Interpreter	61
6.4	Source Code	62
6.5	Disclaimer	62
 7 The Raspberry Pi and the Sense HAT		63
7.1	Rasperry Pi 3 Model B	63
7.1.1	Software	64
7.2	Sense HAT	64
7.2.1	Sensor Communication	65
7.2.2	Byte Encoding	65
7.2.3	The Sensors	65
7.2.4	The LED Matrix	66
7.2.5	RTIMULib	67
7.2.6	Errors In Reading Temperatures	67
 8 Implementation		69
8.1	Criteria	69
8.2	Implementation Language	69
8.2.1	Python	70
8.2.2	C	70
8.2.3	C++	71
8.2.4	Rust	72
8.2.5	Conclusion	73
8.3	Work setup	73
8.3.1	Required Software	73
8.4	Porting the Sense HAT library to Rust	74
8.4.1	Writing the C wrapper	74
8.4.2	The Framebuffer	75
8.4.3	Reimplementing the Sense HAT module from C to Rust	76
8.5	The Lexical Analysis and Syntax Analysis	76
8.5.1	The Scanner	76
8.5.2	The Parser	77
8.6	Representing the Data Type Real in Rust	78

8.6.1	Arithmetic Operations	78
IV	Summary	81
9	Conclusion	83
9.1	Analysis	83
9.2	Future Work	83

List of Figures

1.1	A program for controlling a fan using graphical notation . . .	16
2.1	Range representation of the WiFi bands	23
4.1	Flowchart for the program execution	31
5.1	The bit pattern of a fixed point number	41
5.2	An example of how variables can be declared in a language with a static type system	46
7.1	The Raspberry Pi 3 Model B. Source [39]	63
7.2	The Sense HAT. Source [41]	65
7.3	The orientation. Source [43]	66
7.4	The RGB 565 bit field after a RGB 888 value has been encoded by the Python library.	66
8.1	A tagged union in C	71

List of Tables

5.1	Arithmetic operators	48
5.2	Logical operators	48
5.3	Emojis for nil	52
5.4	Emojis for send	52
5.5	Emojis for power	53
5.6	Emojis for timer	53
5.7	Emojis for microphone	53
5.8	Emojis for speaker	54
5.9	Emojis for GPS	54
5.10	Emojis for temperature	54
5.11	Emojis for humidity	55
5.12	Emojis for pressure	55
5.13	Emojis for LED	55
5.14	Emojis for Light	56
5.15	Emojis for monitors	56
5.16	Emojis for On/Off action	56
5.17	Emojis for loop	57
5.18	Emojis for If-Else	57
5.19	Emojis for function	57
7.1	The Raspberry Pi 3 Model B specification [36]	64
7.2	The Sense HAT sensor modules [42]	66
7.3	The Sense HAT specification [40, 48]	67

Part I

Background

Chapter 1

Introduction

1.1 Impulse SmartCity

This project was initially going to be a contribution to another project called Pulse SmartCity. Pulse SmartCity is, as the name implies, a smart city project lead by Jon Bøhmer. The idea was to use hundreds of IoT devices scattered accords a city to gather information about weather, temperature, movement of vehicles and to monitor the electrical consumption. The system would consist of a gateway, nodes and tags. The nodes would communicate with a gateway to transmit data. The tags would be used as identifiers and nodes would be able to recognise them.

The contribution to Pulse SmartCity was to design a script language for this system. Part of the nodes design was that the user should be able to program them through a Web UI. To make the programming easier, the UI would use graphical elements to construct a script. Similar concepts can be found in programming languages such as scratch [1]. The graphical notation would be converted to a textual form behind the scenes. The language had to be simple so that anyone could use it.

The second part of the contribution would be to design an interpreter for the language. The scripts would be interpreted on the IoT devices. The devices also had to be able to switch out running scripts with new ones at any time. The goal was to make the nodes as independent as possible, even if they were placed in hard to reach areas. This means that the user can simply send a new script to a node through radio waves.

There were a few restrictions that had to be addressed when designing both the programming language and the interpreter. The nodes have limited storage space, RAM and processing power. They also have a small battery, so the goal was to minimise the power consumption as much as possible. The scripts had to be as small and compact as possible in terms of file size. This is to reduce the amount of data which has to be transmitted to a node as sending and receiving to radio signals consumes a lot of power.

The data collected by the nodes would be sent to a cloud service, such as the Amazon AWS, where it would be processed. The nodes themselves would be able to do some of the processing themselves. As a result, less data will be transmitted over the network which will free up some

bandwidth. The nodes would be able to conserve more energy by not having to use generate radio waves as often.



Figure 1.1: If the temperature is above 30 units, turn on the fan at 65% power. The unit was not specified by the creator. Credit: Jon Bøhmer.

1.2 The Current Project

As time went by, the goal of this thesis changed. The focus moved away from the idea of creating a programming language for one specific platform. Instead, the programming language would be designed to work for all IoT devices, regardless of architecture.

Most of the original goals remain the same. The main objective is to create a small scripting language which is easy to use. The language is designed to resemble a traditional imperative programming language. The target audience are people who possess basic knowledge about programming. The functionality of the interpreter remains the same.

1.3 Acknowledgements

I'd like to express my gratitude to my supervisor, Dag Langmyhr, for his excellent help and guidance. His cheerful mood and optimism kept me going throughout the duration of the project.

I would also like to thank two students, Vetle Volden-Freberg and Olav Wegner Eide, who kept me company throughout the last couple of months.

Chapter 2

The Internet of Things

2.1 Internet of Things

Technology has come a long way since the early days of computing. The first digital computers were big, expensive and consumed a lot of energy. Writing a program for solving a particular task could take more time a few decades back than it would today. As time passed, computers became smaller, faster, more energy efficient and most importantly: cheaper. The power of a supercomputer from back then can now be found in modern smartphones. This technological advance has come with many benefits. Communication and exchange of information over the whole globe and even to space has been possible for a few decades now and is part of our every day life. All kinds of information from various media is now available at the tip of our fingers.

Technology has in recent years been taking a step in a new and unexplored direction and a new phrase for this phenomena emerged: The Internet of Things (IoT).

2.1.1 What is the Internet of things?

The Internet of Things (IoT) is a concept of connecting devices through the Internet with the ability to gather and exchange data. These devices or gadgets are usually embedded with micro-controllers, software, sensors, actuators and Internet connectivity[2]. Such gadgets may include regular household items like washing machines, fridges, sound systems, coffee makers, alarm clocks and much more. There are also IoT applications used in cities like sensors which monitor traffic, air and water pollution and electrical energy consumption. In the future, self driving cars will be transporting people to their destinations and these cars will use sensors and wireless technology to communicate with each other in the traffic. It is estimated that there will be 6.4 billion ($6.4 \cdot 10^9$) devices connected and in use by the end of 2016 and that this number will increase to 20.8 billion in 2020 [3].

2.2 Sensors

"A sensor is a device that detects and responds to some type of input from the physical environment. The specific input could be light, heat, motion, moisture, pressure, or any one of a great number of other environmental phenomena. The output is generally a signal that is converted to human-readable display at the sensor location or transmitted electronically over a network for reading or further processing." [4]

IoT devices are usually equipped with one or more sensors which they use to gather information. This information is then processed internally on the device or sent back to a server (ex. Amazon Cloud Services or a local computer).

Position, placement and displacement

This sensor is able to detect how the distance to another object. In other words, distance. A different type of this kind of sensor is able to determine how far it has travelled from a fixed point by measuring angular movement, or by rotation. For example, a robot can know how far it has travelled based on how many times its wheels have rotated.

Presence and motion

Sensors which sense presence have the ability to detect if an object is located nearby. For example, a lamp can be equipped with a presence sensor. It can light up if its sensor detects that a human has entered the room. The sensor can detect movement, body heat or both. Another way of sensing a specific presence would be to use a tracking device. This device could for example be placed on a pet. Then the sensor can detect when the pet with the device moves in to or out of its range.

Speed

These sensors can measure how fast an object is moving with the help of magnets or light. Example of use can be found in Anti-spin Break Systems (ABS) in cars which measure how fast the wheels are spinning.

Temperature

A type of sensor commonly found in households. A well known example is a digital thermometer for measuring temperature inside and outside a house. A more advanced system would involve could be able to measure temperature in all the rooms inside a house and control temperature regulating devices. The system would detect whenever it has to turn on the heating panel, the heat pump or the air conditioner if it is too hot or cold based on temperature readings.

Humidity and moisture

The ability to measure the amount of water in air. This is also a very common sensor which can be found in devices which also measure temperature. Equipment installed with humidity sensors can be found in homes, bundles together with a thermometer. They can also be found in greenhouses, air conditioning systems, offices and cars[5].

Sound, acoustic and vibration

This sound detecting sensor is more commonly known as a microphone. The microphone can be designed in various ways. It can be designed to only pick up specific types of noises like a door closing shut or a pair of hand clapping.

Light

These are sensors which can detect human visible light. Much like proximity sensors, these sensors use visible light to detect changes and movement in an environment. For example, a lamp can turn itself on automatically if it detects that there is too little light present in a room and it can turn itself off if there is too much light.

Motion and rotation

These are more commonly known as gyroscopes and they sense rotational motion and changes in orientation. Cameras can use gyroscopes to detect shaking and irregular movement which helps them stay in position. This results in less shaky photos and video capture.

GPS

Global Positioning System (GPS) is a system which allows a device to measure its position and velocity anywhere in the world[6]. GPS receivers use satellites orbiting the Earth to determine their position. At least four satellites are required to get an accurate position. The margin of error is usually within a few meters with a good signal. The satellites broadcast their positions and time via radio waves and these signals are picked up by a receiver. The receiver knows how much time has passed after the signal was sent and it is able to determine its location on Earth.

Some IoT gadgets have a GPS sensor installed so they can log or signal their current position and speed. The GPS sensor can be used to keep track of the location of humans, animals and items. A household animal pet, such as a cat, can carry collar installed with a GPS tracker so that the owner can know where his or her pet is located.

2.2.1 Energy efficiency

Low energy consumption is an important key element for the IoT. Because the devices can be placed in remote or hard to get locations, it will usually be necessary to equip them with an internal power source. New technology with low power consumption in mind is being created for the use of IoT. For example, a new version of Bluetooth, called Bluetooth Low Energy [7] and the Wi-Fi HaLow [8].

The devices spend most of their time in a low power state where they lie dormant and only turn on if the sensors pick up a signal or the device has to send/receive information. This requires intelligent hardware and low energy consuming sleepmode compatibility [9].

3D RAM and 2.5D and 3D processors are technologies which are design for higher performance with lower power consumption as well as lower manufacturing costs [10][11]. The development of these technologies is partly driven by IoT like smartphones.

2.3 Communication

The IoT usually use wireless communication technology to transmit information. This information can be send to other IoT devices, terminals or servers on the Internet. In some instances, it may not be necessary to send data over the Internet. For example, a smartphone which has received a message via a social media application can send a notification over Bluetooth to the owner's smartwatch. The watch can then display the message on its screen. In other situations, a message is send over the Internet to notify a remote party over varying distances, from just a few meters to entire continents.

As the IoT phenomenon has grown, the need for better and more energy efficient technologies have emerged. Wireless communication is one of the technologies which have been modified and developed to fit the need of IoT. The IoT devices have very low battery capacity but they usually don't send a lot of information at any given time. This means that developers can create new or modify existing communication protocols to use less energy at the cost of less throughput. The next sections will cover two upcoming and popular wireless protocols aimed at improving the battery life and range of modern IoT.

2.3.1 Bluetooth

Bluetooth is a wireless technology designed for exchanging data over short distances. The technology allows for continuous streaming of data between two devices at a high transfer rate (25 Mbit/s theoretical bandwidth). Bluetooth uses radio signals and operates between 2,4 GHz and 2,485 GHz, meaning it operates within the frequency spectrum of other wireless technologies like WiFi[12]. Bluetooth devices can connect with each other by pairing. Bluetooth does not send signals in a general direction. Instead, it broadcasts a signal so other nearby devices can see it and pair up.

However, they need to be in close range, usually up to 10m. Bluetooth also comes with the benefit of backwards compatibility, meaning devices with newer versions of Bluetooth can still communicate with devices with older versions of Bluetooth.

Common usage

This technology is very popular and easy to use which is why it is used in many common electronic devices like smartphones. For example, a user can pair up a phone to a wireless headset and stream music to it. Bluetooth can also be used to exchange data from one device to another, making it easy to transfer images from a phone to a computer for backup, or to synchronise data on two devices. In recent years there has been a development in the use of Bluetooth beacons in retail stores. The beacons can inform a person through his or her phone about the location and information about the store. Both Apple and Google, the two major smartphone operating system developers, provide support for Bluetooth beacons in iOS and Android[13][14]. This technology works by having the phone constantly listening to information sent from such beacons while consuming as little battery power as possible. When it receives a signal from a beacon, it can then notify the user by displaying a message or opening a designated application.

Bluetooth Low Energy (Bluetooth Smart)

Standard Bluetooth is excellent for transferring data continuously but this feature may not be a necessity for IoT devices, which does not stream data, but rather transmit it via short bursts and otherwise stay in a low energy (sleep) mode. Therefore, a new version of Bluetooth was created: Bluetooth 4.0, also known as Bluetooth Low Energy (BLE)[15]. This version was designed with IoT in mind, meaning it offers low energy consumption by staying in a low energy state when not communicating with other devices and by sending small amounts of data at a high transfer rate (1 Mbit/s). BLE also spends little time pairing with devices which draws less power. While classic Bluetooth can spend around 100ms to connect, BLE needs only around 6ms. Battery-wise, this technology can make a small device last a couple of years with only a small coin sized battery as a power source. In terms of range, BLE has around the same range as standard Bluetooth at around 10m indoors and up to 100m outdoors with no obstacles.

Bluetooth version 4.1 and 4.2

Bluetooth 4.0 was previously known as Wibree and was developed by Nokia from 2001 to 2006 before changing name[16]. The technology was first implemented in the iPhone 4S in later 2011 and has since been added in other devices from other manufacturers. At the time of writing, two new versions of the Bluetooth 4.x specification have been released: version 4.1 and 4.2. Version 4.1 was a software update to the existing specification

which added new features which would improve both user and developer usability. One feature fixed a commonly know issue where Bluetooth 4.0 and 4G (LTE) signals would interfere with each other, resulting in worse performance and more power consumption. (version 4.2)

Bluetooth 5

The next version of Bluetooth was announced in June 2016 by the Bluetooth Special Interest Group (SIG), which is the caretaker and innovator of Bluetooth technology[17]. The new version promises increased range, transfer speed and broadcasting messaging capacity. The range is quadrupled from a theoretical maximum of 100m to 400m outdoors and 10m to 40m indoors. The transfer speed is doubled from 1Mb/s to 2Mb/s. Bluetooth 4.2 allows devices to send packets of 31 bytes in size but Bluetooth 5 increases this amount to 255 bytes, resulting in an 800% increase in capacity. These major improvements allow for devices to stay connected even at long distances and possibly reducing the amount of repeaters needed in a network of Bluetooth devices. Data transfer is higher, meaning more information can be transferred in less time. As a result, less power is consumed due to less time being used to send data. Since version 4.2, devices can connect to each other without pairing up like one usually does when pairing up a phone to a wireless headset. Bluetooth beacons use this type of pair-less connection to send information and location data to smartphones. Bluetooth 5 allows such beacons to send a lot more information per message than previous versions.

2.3.2 WiFi

WiFi is a technology which allows devices to connect to a wireless LAN (WLAN) network. This allows the devices to wirelessly connect to other devices or to the Internet. WiFi operates in the 2,4 GHz and 5 GHz frequency spectrum. The Institute of Electronics and Electronics Engineers (IEEE) is an association which defines standards and protocols for communication technologies in industries like telecommunication. The IEEE uses unique numbers for each standard. The 802 is the prefix used by any protocol or amendment that entails area networking. For instance, Bluetooth personal area networks are designated by 802.15 while WLAN is designated by 802.11 [18]. Today, most devices and gadgets use 802.11n or 802.11ac, also known as WiFi N and WiFi AC respectively. WiFi N, which was released in 2007, has a transfer rate around 300-450 Mb/s depending on the number of antennas used and the range while WiFi AC from 2013 can reach a speed of 1 Gb/s. However, WiFi AC only uses the 5 GHz frequency band. While the high frequency screens the signal from the popular 2,4 GHz (resulting in less interference) it loses a lot of range. WiFi N can operate in both bands. Both of these standards are backwards compatible with the older A, B and G standards.

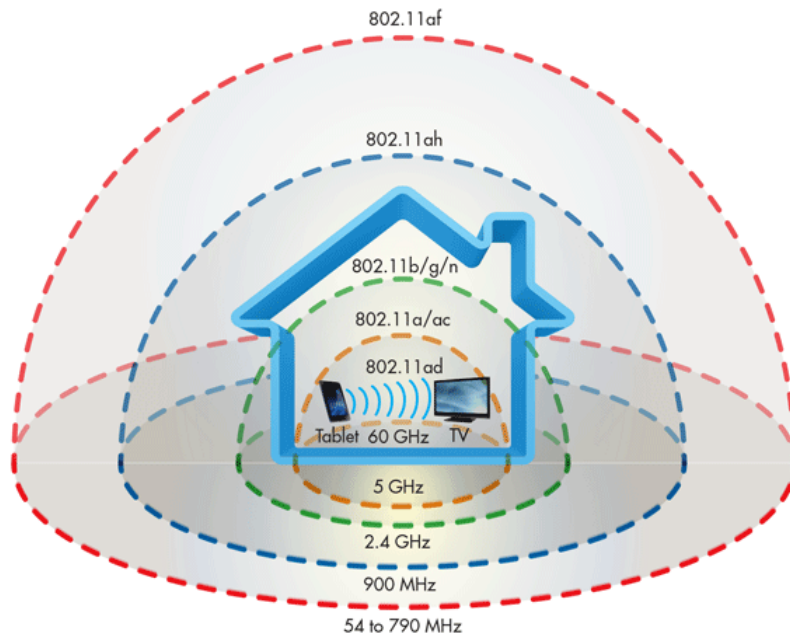


Figure 2.1: The image is not to scale with actual range. Source [19]

2.3.3 WiFi HaLow

While standard WiFi allows for high transfer rates, it uses a lot of energy to send data. For example, the Amazon IoT Button is a device which registers button pressed and sends a message over the Internet to the Amazon AWS service. While the device itself stays in sleep mode most of the time, when pressed it wakes up and connects to the WiFi. Connecting to a WiFi network consumes a lot of energy. The device ends up only lasting around a 1000 presses before its battery runs out[20]. This is highly inefficient.

A new generation of WiFi is currently being developed by the WiFi alliance[8]. The next generation WiFi protocol goes under the name of WiFi HaLow and its specifications are described by the standard IEEE 802.11ah. The WiFi HaLow protocol is, like the Bluetooth LE, designed with Internet of Things in mind. It is a competitor to the BLE standard, but the two protocols differ in that WiFi HaLow allows devices to directly connect to the Internet.

Specification

WiFi a/b/g/n/ac works in the 2,4 GHz band while WiFi HaLow operates in the 900 MHz Band. The 900 MHz band is a lower frequency meaning it takes less power to send data. WiFi HaLow signals can travel further than normal WiFi signals. WiFi HaLow is expected to have a range of 1 km[21]. The lower bandwidth also means that the signals are better at penetrating walls, meaning a building can do with fewer WiFi repeaters for their IoT devices. This strengthens the ability to place IoT devices in remote areas as they can operate over longer distances. WiFi HaLow has

a minimum throughput of 100-Kbps which may not seem like a lot, but it should suffice for the short bursts of data transfer IoT devices employ. Devices which support WiFi HaLow are supposed to use the 2,4 GHZ, 5 GHZ and 900 MHz bands, meaning they can communicate with both new and old technology, making WiFi HaLow more accessible and easier to integrate.

Chapter 3

Related Projects

In this chapter we'll look at projects which incorporate scripting languages for the IoT platform.

3.1 Alternatives

3.1.1 Scriptr

Scriptr.io, or just script, is a cloud based platform for IoT, mobile and web applications. It offers developers to write and deploy server-side scripts which serve as custom back-end APIs for their IoT solutions [22]. The scripts can be written in either JavaScript or in Blockly. Blockly is a visual programming language where the user can drag and drop code "blocks" to create programs and it is similar to the programming language Scratch. Scriptr offers a web hosted user interface and provides secure connections for devices connecting to the service by the means of authentication tokens. In addition to existing functionality in JavaScript, Scriptr also offers a variety of modules such as managing devices and connection to social media platforms.

3.1.2 Libelium Wasmote Plug & Sense

Libelium is a wireless sensor network platform provider for Smart City solutions and is the developer of the Wasmote Plug & Sense (WPS) [23]. The WPS is an encapsulated wireless sensor device which allows system integrators to implement scalable and modular wireless sensor networks [24]. It can be powered through solar power through a solar panel, it has options for many types of radio communication technologies such as WiFi, Zigbee, 3G, and its eight modules can be integrated with more than 60 sensors [24]. Most importantly, its program can be swapped wirelessly, meaning it can be placed in out of reach areas while allowing the maintainer to effortlessly change the currently running program. The programs run by the WPS devices are written in C/C++. Similarly to Arduino code, the WPS C/C++ code also require setup and loop

functions. The WPS can also be accessed through a graphical programming interface.

3.1.3 Johnny-Five

Johnny-Five is an Open Source JavaScript platform for Robotics and IoT. The framework supports a wide range of microcontrollers and single board computers such as Arduino and Raspberry Pi and many others. The framework relies on node.js to interpret the JavaScript programs. Johnny-Five has support for a wide range of sensors and actuators used by IoT devices, such as LEDs, servos, GPS, motors and environmental sensors.

3.1.4 Tessel 2

Tessel 2 is a robust IoT and robotics development platform [25]. More specifically, the Tessel 2 is an open source development board with on-board WiFi capabilities that allows you to build scripts in Node.js [25, 26]. It is also capable of running Rust programs. The platform offers official and community-created modules for various sensors and actuators [25].

3.2 Evaluation

JavaScript is a popular programming language. It is used on the web and in desktop applications [27] and IoT [28]. JavaScript is able to run on IoT devices thanks to Node.js. Node.js is an open-source, cross platform JavaScript run time environment [29]. It uses the Google V8 execution engine to compile JavaScript code to native machine code for improved speed and performance [29, 30]. V8 can be run on multiple systems, including the ARM processors. With Node.js and JavaScript it becomes easy to write applications which can connect to the Internet and use web services directly on the IoT device.

JavaScript can also be used on the server side instead. The physical devices can use programming languages which operate on a lower level, such as C. This approach raises the required knowledge of the user as low level C is harder to get right than working with a language with high level abstractions.

The reason why Node.js and JavaScript is not used for this project is because JavaScript is big and complex. One of the goals of this project is to create a small and simple programming language.

Part II
Design

Chapter 4

Requirements for an IoT Programming Language

In this chapter we will discuss the requirements for implementing a scripting language for the IoT. We will also look at the IoT platform, the interpreter and its requirements.

4.1 The Target Platform

The language is targeted at IoT devices. These devices are not only very power efficient and usually small in size, but they are also limited in terms of processing power, space and ram capacity. IoT devices do not perform a whole lot of data processing and in most cases only gather data from their surroundings and transfer it to an external device, such as a remote server.

IoT devices are not necessarily meant to be used as a general purpose computer or micro-controller, like the Raspberry Pi or the Arduino, but are instead carry out small and particular tasks. The Raspberry Pi is more suited for general purpose applications. That is not to say that device such as a Raspberry Pi cannot be used in IoT, but they may not be as efficient at doing a particular task as a custom device would be.

In some cases, programs for embedded devices are written in C as the language allows the programmer to write fast programs with a small memory footprint and file-size. C gives the programmer a lot of control over the behaviour of the program, but it requires a good understanding of the hardware it's running on. On the other hand, there already exists scripting solutions for IoT which utilise JavaScript for the scripting part. These applications may trade away some the speed brought by a custom, low level solution for a the benefit of easy setup and deployment.

4.2 The Target Audience

The target audience are students or people who know a little about programming aspects and constructs. Furthermore, the users are people who are interested in programming and IoT. The language should provide

the means to write simple IoT project in a short amount of time. As such, the language must offer the basic functionality required to quickly deploy simple IoT solutions.

4.3 The Core Requirements

The language must be simple and easy to use. The target demographic are amateur programmers, so the language should be restricted in terms of complexity. This should ease out the learning curve and make the users more productive.

4.4 Hot Swap Scripts

The programs are scripts. The IoT devices will have software which can run those scripts. Part of the design is the ability to swap out scripts with new ones. This means that a device which is currently running one script can receive a new script. When this happens, the software will have to check if the new script is usable (ie. it does not contain any syntax errors). If the new script is OK, the running script is stopped and deleted along with its data (variables etc.). Then the interpreter starts running the new script. On the other hand, the new script is deleted if the interpreter detects that it contains an error. The execution of the old script is resumed.

The user must be able to upload a new script to the desired device regardless of where it is located. This means that scripts can be uploaded over wireless connections such as WiFi and Bluetooth. These mechanics makes the platform more flexible and easier to use.

The flowchart in figure 4.1 on page 31 shows the various states the interpreter can be in.

4.5 Interpreting vs Compiling

In this section we'll look at benefits and downsides of program compilation and program interpretation with regard to this project.

4.5.1 Compiling

A compiler is able to verify the semantics of a program before compiling it to a executable format. The compiler can also create executable which are tailored to a specific architecture for improved speed and memory consumption. As a result, programs which are compiles tend to be fast. Depending on the language, the analysis of the program can be used to detect bugs which would occur during run time. Thus, compilers can add additional safety guaranties. For example, the compiler can detect type mismatches in a statically typed language.

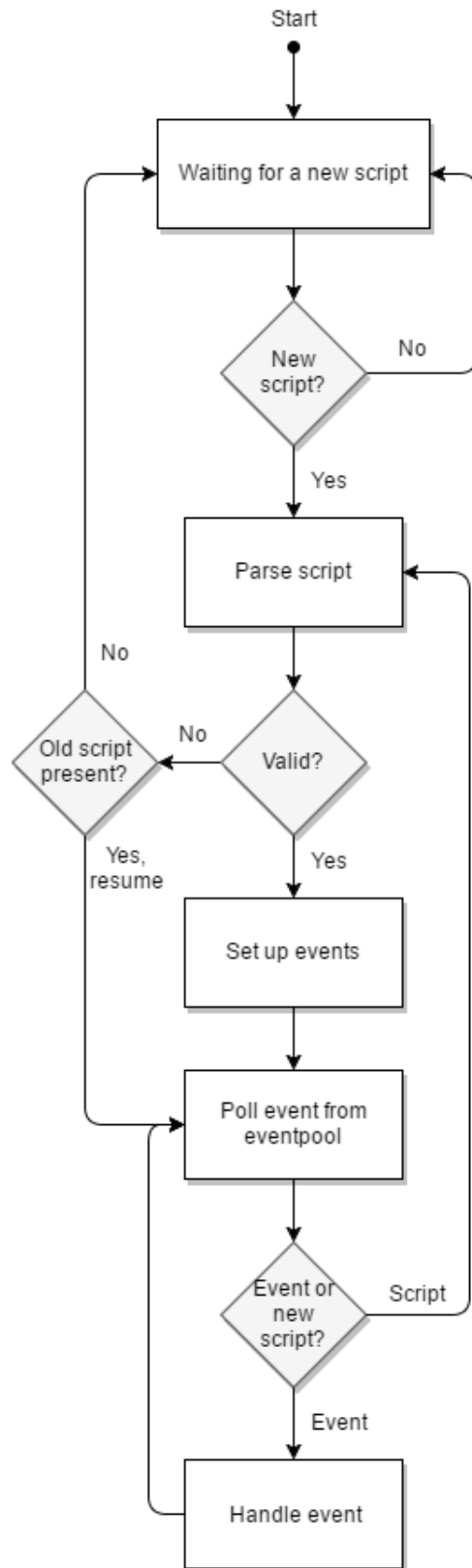


Figure 4.1: Flowchart for the program execution

The downside is that the generated executable can (usually) only be used on one target architecture. The compilation time can also be slow, at the benefit of fast executable.

4.5.2 Interpretation

A language that is interpreted is run by an interpreter. This means that the code is not compiled to a hardware specific language. Instead, it is interpreted either directly by an interpreter or compiled to bytecode first. For example, Java compiles to a Java specific bytecode which can be interpreted by the Java virtual machine (JVM). A benefit of interpreted programs is that they can be deployed fast. Interpreted languages are also portable and can run on any machine as long as that machines architecture is supported.

Interpreted programs are usually slower than compiled programs. A compiler can generate can generate optimised binaries for a hardware platform, while an interpreter has to keep interpreting the program. That said, it is possible to optimise interpreted programs by generating optimised bytecode. The interpreter can detect segments of code which are used often and spend some time to optimise those. Eventually, the interpretation can become faster for each optimisation.

4.5.3 Evaluation

The language is more suited to be an interpreted language as it will be used as a scripting language. The issue with compiled programs is that it would be hard to verify them on the IoT device. There is always a possibility that something were to go wrong with the program as it is sent to the device. For example, it is not impossible that bits get flipped or that some packets with the data are lost. The compiler would have to support a wide range of different architectures. That said, this argument may be mute as an interpreter must be able to run on different architectures.

Part of the original design was to use an interpreter for a script language. The reason was, and still is, the benefit of code verification in multiple stages of deployment. This allows both the user and the receiving device to verify the scripts. This should make the platform more robust.

4.6 The Type System

In this section we'll discuss static typing and dynamic typing. We'll also discuss which type system is suited based on the given requirements.

4.6.1 Static Typing

With static typing the user must specify the type of a variable when a variable is declared. The type of a variable cannot be changed once assigned. This means that all variables have their types known during compile time. The compiler can just check the type of each variable and

tell the user if there is a type miss-match. It can also prevent bugs which would occur during run time. Static typing can lead to less bugs as errors are caught early on during development.

As an example, let's look at variable declarations in C. An integer variable can be declared without an assignment (`int x;`) or with one (`int y = 42;`), but it must always have a specified type.

4.6.2 Dynamic Typing

Dynamic typing allows variables to be declared without having to explicitly declare their type. A variable is not bound to any type. Therefore it can be reassigned to another value of a different type. Usually interpreted languages use dynamic typing as there is no compiler to perform a type checking. Any type errors are caught during run time. Below is a trivial example on how dynamic typing can lead to subtle bugs. In a larger code base, finding this type of bug can take up a lot of time.

```
1 def add(a, b):
2     return a + b
3
4 add(1, 2) # OK
5 add(3, "hello") # Run time error
6 add("abc", "def") # OK
```

4.6.3 Inferred Typing

Inferred typing allows the compiler to infer the type of a variable based on how it is used. This allows users to skip writing out the type of a variable, but they retain the benefit of type checking during compile time. Whenever the compiler cannot infer the type of a variable, it will terminate with a compiler error. This forces the user to specify the data type so the program can compile. Inferred typing falls under the category of static typing as all types are known during compile time.

As an example: In Rust we can write the following code.

```
1 let x = 128;
2 let y: u16 = 20 + x;
```

Here, `x` initially has no type, but the compiler knows it must be an integer. The `y` is declared with the type `u16`, so the expression on the right-hand side of the assign statement must return a value with type `u16`. The compiler can then infer that `x` must have the type `u16`.

Inferred typing is more complex than normal static typing and it is harder to implement. It also offers no practical benefit, but it can make code look a little less verbose.

4.6.4 Evaluation

Dynamic typing means the user does not have to worry about types declarations. Dynamic typing should also help lowering the learning curve.

The script file sizes should be as small as possible. We can potentially save a few bytes by not having to write out the type of each variable. The less data the IoT devices have to send, the less power they have to consume on wireless communication. This also means we can potentially free up some space on disk and RAM.

A counter argument for dynamic types is that they increase the chance of run time errors due to type mismatches. One of the goals of the language is that it should never cause a crash, so choosing dynamic typing over static typing is counter intuitive as the latter can prevent type errors.

4.7 Event Oriented Programming

Most of the programming languages that are used for IoT are imperative languages [28]. It is probably the most used paradigm and it is easy to understand as it is easy to follow the flow of the program execution when reading imperative code. However, there is another programming paradigm which is more suited for IoT and for the language of this project: event oriented programming.

IoT devices are units which wait for their sensors and actuators to react to changes. When an event is triggered, the software on the device must handle the event. Event oriented programming allows the user to write programs which model resemble the behaviour of the device. This should make programming IoT feel more natural to the user.

It is possible to write event oriented programs in imperative languages, but they can be clunky, verbose and not very intuitive. Instead of making event handling an addition to the language, the language will be built around handling events directly.

4.7.1 Handling Events

All events will be inserted into an event pool. The events will be sorted based on their priority and when they are scheduled to run. The interpreter will poll one event at a time and handle the event. When an event handler is running, it will not be timed out and it will be allowed to run until it is complete.

The interpreter will be single threaded. This means that no more than one event can be handled at any time. This simplifies the implementation and it eliminates the possibility of race conditions for shared data between event handlers.

The type of event handlers range from user defined to input and output from sensors, actuators and IO.

4.8 Security

Security is an important factor when it comes to IoT. The devices are rather vulnerable as they are connected to the Internet. As such, they can be

accessed by a malicious third party. These device tend to have little to no protection from outside attacks.

Regrettably, the security mechanisms of the language and the interpreter were not a primary concern for this thesis.

4.9 Errors and Exceptions

The interpreter should always be running and should never terminate unless the device is turned off. Ideally, the interpreter should be able to handle errors caused by user scripts, in the sense that it should never crash when it encounters one. On the other hand, compilers and interpreters can print out a warning if they detect an something unsound within the code they are parsing. A program which causes an error due to an exception or a logical error can terminate safely thanks to built in error handling. Of course, some errors are beyond our control, while others are due to implementation oversights or bugs. For example, the program may run out of memory, it tries to access data which is out of bounds, it runs into the infamous null pointer exception or simply causes a segmentation fault. What ever the cause may be, many of these errors cannot be handled by the interpreter itself and the underlying operating system will most likely terminate the interpreter. We will later look at how we can try to maintain a near 100% up-time by using a background process to monitor the status of the interpreter.

4.10 Handling a Software Crash

When the interpreter detects an error in the user script it must now choose an appropriate course of action. An error can be caused by various means, depending on the implementation and design of the language. We will now look at the several different ways of handling a crash.

4.10.1 Alternative 1: Reset

The interpreter restarts the interpretation of the script. This means that it starts to interpret the script from the beginning. There are downsides, however.

- All temporary data is lost.
- The program may encounter the error again, causing a never ending cycle of resets.
- The user will be spammed with error reports if enabled.

There does not seem to be many advantages to this solutions. However, the user does not need to perform any additional maintenance as the scripts just restarts. It could be that the error was caused by a one time error which means it could have been expected to happen.

4.10.2 Alternative 2: Remove the script

The interpreter removes the script altogether from the storage medium, meaning the interpreter no longer has a script to interpret. This scenario may also result in loss of intermediate data. The interpreter enters its default state, the standby state. In this state, the interpreter must now wait for the user to provide it a script it can run.

The advantage of this approach is that the user knows the interpreter will safely stop interpreting a script just like a normal program which has encountered an error would on a desktop computer. Some exceptions can be caused by severe errors and it might be safer to simply stop instead of rerunning the same script again.

4.10.3 Alternative 3: Ignore it

The interpreter can choose to ignore the error and continue to interpret the program. In JavaScript, there is a null value, which represents the intentional absence of an object value. The user can use null in arithmetic operations with other data types which can yield confusing results.

```
1 > 1 + null
2 1
3 > 1 * null
4 0
5 > {} + null
6 0
7 > {} * null
8 Syntax Error
9 > null * {}
10 NaN
```

Nevertheless, it is there by design and it may cause a program crash in certain situations, which may force the programmer to check for null values.

Returning to the interpreter, we could implement a similar mechanic if the language implements a null type for representing an error or lack of data. Now, if the interpreter were to detect the use of a null value, it could choose to ignore it and allow data to become corrupted. However, some exceptions are caused by more severe problems which cannot be ignored. This solution would have to be implemented together with one or both of the solutions discussed in the sections above.

4.10.4 Alternative 4: Terminate

The interpreter can terminate completely and never restart automatically. What this does is that the user has to restart the program manually. This can be done in various ways, depending on the hosting hardware and software. For example, if the device is a Raspberry Pi running a Linux distribution, the user can restart the program, either remotely (through ex. ssh) or locally. For other devices with simpler operating systems, the user might have to physically approach the device and manually restart it,

by unplugging and replugging the power source or by hitting the power switch.

Since we'd like our IoT devices to always be up and running. Terminating the running process and the script would probably do more harm than good.

4.10.5 Alternative 5: Inform the user

It would be beneficial to inform the user that an error has occurred. In the case where the interpreter crashes or terminates, it will not be possible for it to send a diagnostic message to a user, unless there is a watcher program (daemon) which detects the crash and is capable of reporting errors. On the other hand, if only the script crashes or causes an error, the interpreter itself can notify the user of when and what went wrong. The interpreter must therefore know how and to whom it should send the message. The diagnostic should contain information such as when the error occurred, what caused it and the position in the script where it happened.

4.10.6 Watchdog Daemon

A daemon is a program which runs in the background and is not under direct control of the user. On Unix-like systems, there is a daemon called Watchdog which checks whether the operating system is running correctly and can cause the kernel to perform a reset if an error has occurred.

In the case where the interpreter terminates, for whatever reason, a daemon process could be responsible for starting a new instance of the interpreter program. The daemon must be able to determine whether the interpreter is running or not, or if its simply sleeping.

The daemon itself must be simple, small in size and perform as few operations as possible while still being able to correctly assess the state of the interpreter its watching. This is to reduce the power consumption and to conserve as many system resources as possible as they may already be limited due to the current nature of IoT devices.

Adding a daemon process may require the IoT device to be capable of running an operating system which is capable of running multiple processes, either in parallel or concurrently. In the case where the interpreter is running as a integrated part of the system, aka there is no underlying OS with a scheduler and resource manager, it may become impossible to implement the desired daemon. Though, perhaps it would be possible to mimic a daemons behaviour through other means.

4.11 Sensors and Actuators

The language will provide an interface to the sensors, actuators and IO. This means that the user will never have to worry about implementing low level functionality. This puts a lot of responsibility on the one responsible for implementing the interpreter as he or she must implement the interface

for sensors and actuators. However, the interface becomes more efficient as it is implemented in a lower level language and can be accessed directly by the interpreter.

One of the goals of this project is to make a language you can use to swiftly deploy IoT applications. By offering a built in interface to sensors and Actuators we are able to fulfil this goal.

4.12 The Standard Library

The standard library (STD) will contain functions which allows the user to access sensors, actuators and IO. In addition, the STD will contain functions which allows the user to manipulate and interact with the built in data types.

Chapter 5

Designing Daspel

5.1 What is Daspel?

Daspel is an event-oriented scripting language which is targeted at Internet of Things devices. Daspel provides high level abstractions over the device, including sensors, actuators, IO and network communication.

Naming the language

‘There are only two hard things in Computer Science: cache invalidation and naming things.’ – Phil Karlton[31]

With the help of Vetle Volden-Freberg, I settled on a name for the script language: Daspel. The name is derived from Event-Driven Actuator & Sensor Programming Language, or EDASPL for short. However, EDASPL isn’t easy to pronounce for English speaking users, so the letter E was moved from the front to between P and L.

5.2 The syntax

The syntax is inspired by C, JavaScript and Rust syntax. Daspel is written to resemble imperative programming languages in terms of function and block structures.

5.3 Data types

In Daspel there are 5 fundamental data types.

- `int`
- `real`
- `bool`
- `string`

- `list`

There is also the `nil` type which is used for error values. Out of the five types, three of them can be considered to be primitive types: `int`, `real` and `bool`. The `string` and `list` are data structures and have a more complex implementation.

The following sections, the data types will be referred to by using a mono-spaced font. This is just to visually separate the data types and concepts with the same name.

5.4 Integer

In Daspel, the `int` is a signed two's-complement 32-bit integer. As such, it should cover the range of numbers required for simple calculations an IoT device will perform. Granted, it's not given that every IoT device has a 32-bit processor as they can also be 16-bit or 8-bit. However, a 16-bit processor is fully capable of handling 32-bit and 64-bit data at the cost of spending more time on additional instructions as opposed to 16-bit data. 32-bits is a commonly used size for integers and is often the default size.

But what if the user would like to use smaller integers to save space? The dynamic type system would make it a bit to specify a more specific bit size for an integer as opposed to a static type system. In other languages, there is usually no way to specify the integer size without resorting to creating a new variable or casting. However, Daspel could borrow the integer annotation used in Rust. The syntax consists of appending the data type at the end of the number. For example, `100i32` is a signed 32-bit integer with value 100 while `9u8` is an unsigned 8 bit integer with value 9. This feature could have been added, but for the sake of simplicity and due to time constraints it was not added in this version of Daspel.

5.4.1 Integer Syntax

The syntax for the `int` type is exactly the same as in other languages.

```
1 let a = 1;  
2 let b = 2;  
3 let c = a + b;
```

5.5 Real

The term `real` is used to represent the decimal numbers. There are two sub types of reals: fixed point numbers and floating point numbers. Daspel uses the fixed point notation. The data type which represents decimal numbers is called `real`. Visually, there does not seem to be any difference between the two types. However, fixed point numbers are stored as integers. Fixed point numbers use integer arithmetic as opposed to floating point arithmetic. This gives them an edge when it comes to computation speed. On devices which do not have a floating point processor [32] (FPU),

floating point arithmetic is very slow. For this project it is assumed that IoT devices do not have an FPU ¹. Floating point numbers are also have a higher precision than fixed point numbers due to the number bits used to represent the fractions being higher for floats. However, the assumption is that user of Daspel will not require the full precision of a float. It is much more convenient to use the faster fixed point.

`real` in Daspel is a signed two's-complement 32-bit fixed point number. It uses 16 bits to represent the integer part and 16 bits to represent the fractional part. We refer to this format as the Q format [33] which is a format used for fixed point numbers. In other words, the `real` has the format Q16.16. More specifically, it is a Q15.16 because one bit is used for the sign. 16-bit fractions are able to represent decimal numbers with a precision up to at least 0.9999. Figure 5.1 shows how the integer and fractional bits are stored.

Just like with floating point numbers, precision can be lost during multiplication and division. This is because there is not always enough space to represent the full range of the fractional bits.

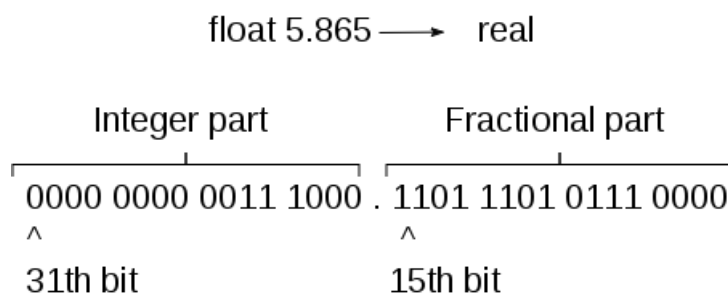


Figure 5.1: This particular pattern is the result of converting a float with value 5,865 to a real.

5.5.1 Real Syntax

```
1 let pi = 3.14
2 let r = 2;
3 let area = pi * r * r;
```

5.6 Boolean

A Boolean is a binary value and can be either `true` or `false`.

5.6.1 Why include boolean?

Boolean is a clear representation of a condition which is used in conditional structures. They make the programmers intentions clearer and can make the code more readable. Other languages, such as C and JS, allow integers to represent boolean values. Any non-zero integers is `true` and 0 is `false`.

¹This is usually due to the high cost of FPUs.

This approach can be considered to be a form of implicit casting between types and allowing one type to represent multiple forms of data at the same time. There is nothing necessarily wrong with this approach, but in Daspel we chose to separate the duality of C integers and instead separate the it into two distinct types.

If Daspel did not feature conditional structures, it would no longer need to have a boolean type.

5.7 String

Strings in Daspel are dynamically sized containers for Unicode encoded text.

5.7.1 Unicode

The `string` type uses Unicode encoding instead of plain ASCII. It is to make it more convenient for non-English speakers to send and receive text in a language they speak from their IoT device.

At one point during the design process, emojis and other Unicode symbols were considered to be part of the syntax. It therefore made sense to also make `string` support Unicode text.

5.7.2 Dynamic Size

`string` is dynamically sized, meaning it can grow and shrink. The size will be handled automatically by the interpreter.

Daspel could have used fixed sized strings, but then the user has to make sure that the string is large enough when inserting new data into it. Daspel is supposed to be as simple as possible and by using dynamically sized strings the user has one less thing to worry about.

5.7.3 Single Character

Daspel does not feature a separate data type for single characters. This is because a character can be expressed as a `string` and it reduces the complexity of the language. Of course, this means that Daspel is slightly less efficient as it has to allocate single character strings on the heap instead of on the stack.

5.7.4 String Syntax

`string` uses the same syntax as most other programming languages. No Unicode characters are shown in this example because \LaTeX does not support Unicode symbols.

```
1 let s = "this is a string";  
2 let empty_string = "";
```

5.8 List

List is a collection which can store any of the Daspel data types. Just like `string`, `list` allocates data on the heap. This allows it to grow and shrink as elements are added and removed from the `list`. The allocation is handled automatically by the interpreter.

The issue with fixed size lists is that the user have to be careful when appending new elements. The user must ensure that the list is always large enough to fit new elements. In the case that it isn't, the user must allocate a bigger list. Instead, `list` does this work for you behind the scenes. This is of course backed up by the goal to make Daspel as simple as possible.

Due to time constraints, the syntax for adding and removing elements to a `list` were not added. However, it would most likely be similar to the ones found in the Python implantation.

5.8.1 List Syntax

`list` has the same syntax as lists in Python. Each element in the list is separated by a comma.

```
1 let positive = [1, 2, 3, 4, 5];
2 let any = ["hello", 13, true];
3 let nested = [1, [2], [[3]]];
4 let empty_list = [];
```

5.9 Nil the Error Type

`nil` is an error type in Daspel. It represent the absence of a value. It is also used to represent an error. This type was included so it would be possible to return some kind of error from functions which read sensor and actuator data. For example, if the temperature sensor is not connected and the program tries to read the temperature, the function will simply return `nil` instead of a `real`. Instead of `nil`, default values could be used instead. So in the example above, instead of returning a `nil`, the function could return 0.0. This is problematic as 0.0 is also a legal value. Therefore, having a dedicated error value makes errors more explicit and easier to catch.

5.10 Variables and Variable Declaration

Like many other languages, Daspel has variables. A variable is a named container which stores a value. In this section we'll discuss how variables are declared. In the scoping section we'll look at where variables can be declared.

The keyword for variable declaration in Daspel is `let`.

5.10.1 Variable Declaration Ambiguity

We'll look at two ways to declare variables. In JavaScript, we declare a new variable by using the `var`, `let` or `const` keyword (`const` creates an immutable variable).

```
1 let x = 32;  
2 x = x + 20 / 3;
```

In Python we can declare a variable without using a keyword.

```
1 x = 32  
2 x = x + 20 / 3
```

However, there is an issue with the Python implementation. Because there is no distinction between a variable declaration and an assignment, we get semantic ambiguity. Consider the following code written in Python.

```
1 x = 1  
2 if condition:  
3     x = 2
```

The user may have intended to declare a new variable `x` in the if-statement, but instead ends up mutating the `x` above. This may seem like a trivial issue for experienced Python programmers, but it can lead to confusion among beginners and can lead to run-time bugs. With explicit variable declaration, we can avoid these kinds of bugs.

```
1 let x = 1;  
2 if (condition) {  
3     x = 3; // Mutates the x above  
4     let x = 2; // Shadows x  
5 }
```

To avoid this kind of ambiguity, Daspel requires explicit variable declarations. This gives us a clear distinction between declaration and assignment.

5.10.2 Variable Declaration Part

In the scoping we'll discuss where variables can be declared. For now just know that variables can only be declared at the beginning of each scope. That section called the variable declaration part. As variable declarations can only occur once place per scope, it becomes easy to check for misplaced variable declarations.

A variable declaration part consists of three variants. If there are no variables to declare, the declaration part is either empty or omitted entirely. The second variant consists of a single declaration. The final variant consists of multiple declarations.

The question is: how can we represent the variable declaration part and it's three states in a simple and elegant way?

One solution is to require each variable to be declared separately as demonstrated below.

```

1 let a = 1;
2 let b = "heat";
3 let c = [4, 3, 2, 1];

```

The next solution is to group all declarations in one block. The block can be empty or omitted if there are no local variables.

```

1 # empty
2 let {}
3
4 # single
5 let { x = true }
6
7 # multiple
8 let {
9     a = 1,
10    b = "heat",
11    c = [4, 3, 2, 1],
12 }

```

A third alternative is to chain the declaration. Something similar can be done in C-like languages.

```

1 # single
2 let x = true;
3
4 # empty
5 let a = 1,
6     b = "heat",
7     c = [4, 3, 2, 1];

```

The last proposal is the simplest alternative which also happens to require the least amount of characters. Thus it is the most suited syntax for the variable declaration part.

5.10.3 Variables and Scopes

This section mostly applies to variables defined in the function scope.

When a user defined function is called it may declare some variables. When the function call is over, the allocated data needs to be cleaned up. In some languages this is done by a garbage collector. In Daspel there is no manual memory management, but there is automatic memory management. The `string` and `list` data types do allocated data on the heap. To avoid filling up the heap to the point where it runs out of space, Daspel needs to clean up allocated data. So when variables pointing to `string` and `list` data go out of scope, the interpreter will automatically free the data on the heap.

5.11 Scoping

In this section we'll look at the scoping rules in Daspel. There are a total of three scopes: the library scope, the global scope and the function scope. The library scope is the outmost scope, followed by the global scope. The

function scope can only exist inside the global scope. There is only one library and global scope, but the user can define many functions and thus many function scopes. However, the function scope cannot be nested. In other words, you cannot define functions inside functions.

5.11.1 Variable Scope

The variable scope determines where variables can be declared and where they can be accessed from. Two designs for variable declaration were considered during the design.

The first design is borrowed from imperative languages and it allows variables to be declared anywhere in a scope. In figure 5.2 we see how variables in a statically typed language are declared and how long they live based on the scope they are defined in. This design makes certain things a bit more complicated as the interpreter has to know exactly where a variable is declared in a scope. This is to ensure that expressions and statements don't use variables which are technically not yet defined.

The second design simply requires all variables to be declared at the beginning of each scope. While not enforced, this practice is not unusual in C programs where some variables are reused for different purposes, such as counters in for-loops.

Forcing variable declarations to the top of each scope makes it a bit more simpler to design the parser. It also makes it clear of how many variables are used per scope. For these reasons, it was decided that variables must be declared in a variable declaration part.

```
1 {
2   let x = 1, y = 2;
3   # Some code...
4   let z = 3; # <- Illegal! Must be declared at the top
5 }
```

```
1 var global; // Lives in the global scope.
2           // Goes "out of scope" when the
3           // program terminates.
4 {
5   var x;
6   ... // Some code...
7   var y;
8   {
9     ...
10    var x; // Shadows variable x.
11           // x in the scope above is untouched.
12    var z;
13  } // x and z go out of scope here.
14 } // x and y go out of scope here.
```

Figure 5.2: An example of how variables can be declared in a language with a static type system

5.11.2 Library Scope

The library scope is the outmost scope and it contains functions found in the standard library. This scope can be accessed from the other two scopes. Note that this functions and variables defined in this scope cannot be changed by the user through a Daspel script as it is defined in the implementation of the interpreter.

5.11.3 Global Scope

This is the scope the user sees when he or she is writing Daspel program. In this scope, variables and functions can be declared. Variables must be declared at the top of the scope in the variable declaration part. Any function defined in this scope can be accessed from other functions, regardless of the order in which they were defined. In other words, Daspel avoids the problem C has where a function cannot access another functions defined below it unless the user declares the function prototypes at the top of the file or in a header file.

5.11.4 Function Scope

The function scope is the scope which exists in functions. This scope is only accessible to the owning function, meaning no external functions can access its contents.

```
1 # The global scope
2 # Cannot see what's inside
3 # foo or bar
4 let glob = 10;
5
6 fn foo() {
7     # Can see glob and bar,
8     # but cannot see bar's zap
9 }
10
11 fn bar() {
12     # Can see glob and bar
13     # Can see zap
14     let zap = [];
15 }
```

5.12 Operators

Daspel includes the standard arithmetic and logical operators. Data types in Daspel cannot be used together with these operators. The exception is that `int` and `real` can be used together in arithmetic operations. When mixing the two types, the resulting value will always be a `real`. This is a similar to the how other languages do it, including Python.

The reason why types cannot be mixed and used together by operators is because it's not always clear what the answer should be. For example,

- plus +
- minus -
- multiplication *
- division /

Table 5.1: Arithmetic operators

- equal ==
- not equal !=
- not !
- greater than >
- greater or equal >=
- less than <
- less or equal <=

Table 5.2: Logical operators

what should the result of `true + 3 - "hello"` be? It's ambiguous and it can potentially create weird values which can just confuse the user and lead to bugs.

There is a special operator used for checking `nil` values. It is called the question mark operator `?` and it is similar to the `nil?` method found in Ruby. By appending the `?` to a value or variable, Daspel will perform a `nil` check on the value. If it is `nil` it returns `true`, else it returns `false`.

```
1 let x = nil;
2 if x? {
3   # ...
4 }
```

5.13 Statements

```
1 let sum = 0;
2 for i in 1..4 {
3   # i will be 1, 2 then 3
4   sum += i;
5 }
```

Daspel has types of statement: while-loop, for-loop, if-else and functions. Only functions can be defined in the global scope. while-loop, for-loop and if-else statements can be defined in function scoped and they can be nested.

Functions cannot be nested inside functions or other statements.

5.13.1 For Loop

The for-loop is similar to the loop found in the Rust programming language. The structure defines an accumulator value and a range to iterate over. The range is denoted by two ints separated by two dots (. .). The for-loop uses curly braces to capture the body of the loop.

The range notation expresses an exclusive range. More specifically: `<start>..<exclusive-stop>`. The number on the left is the first value assigned to the accumulator. The for loop will increment the accumulator by one for each pass. It stops when the accumulator becomes equal to the value on the right. For example, `0..5` yields 1,2,3,4. By using three dots (`...`), the range becomes inclusive, meaning it takes one additional step before it stops. For example, `0...5` yields 1,2,3,4,5

At this point, there is no syntax for specifying the step for the iteration.

5.13.2 While Loop

The while loop is the same as in other programming languages. It consists of a conditional expression and a body surrounded by curly braces. A conditional expression must return a boolean value.

```
1 while t > 3 {  
2     t /= 2;  
3 }
```

5.13.3 Conditionals aka. If-Else

If-Else works exactly the same as in other languages. Just like the while loop, it requires a conditional expression, except for the else case.

```
1 if x > 10 {  
2     # ...  
3 } else if x == 2 {  
4     # ...  
5 } else {  
6     # ...  
7 }
```

5.13.4 Functions

Function in Daspel are declared by using the `fn` keyword, followed by the name of the function and the parameters. The function statement does not declare any return type due to the fact that Daspel is dynamically typed. As such, the function statement is reminiscent of the one found in Python.

To return a value from a function, the `return` statement must be used. By default, functions return the `nil` value.

```
1 fn blink_led(pin, time, colour) {  
2     # ...  
3 }
```

Functions are used to construct event handlers. By prepending an annotation in front of the function, Daspel will understand that this is an event handler. Annotations are used to specify which event the function should handle or how often the function should be called. The annotations start with the @ symbol followed by a name and an optional condition. In the annotations, the user can specify the threshold for the event trigger. For example, to create a handler which is only called when the temperature is above 20 degrees Celsius we write the following:

```
1 @Temperature > 30 C;  
2 fn handler() {  
3     # handle event  
4 }
```

5.14 Things That Did Not Make It To The Language

In this section we'll briefly look at things which could have been a part of Daspel, but were not included because of time constraints.

5.14.1 Variadic Function

A variadic function is a function which accepts a variable number of arguments [34]. One can argue that variadic functions are not necessary in a dynamically typed language as lists can hold any types and such lists can imitate the variadic functionality.

```
1 fn variadic(*args) {  
2     # args is a list  
3 }  
4  
5 variadic(1, 2, 3);  
6 variadic(2);  
7  
8  
9 fn normal(list) {  
10     # takes a single argument  
11 }  
12  
13 normal([1, 2, 3]);  
14 normal([2]);
```

5.14.2 Tuple

A tuple is a finite list of elements [35]. Tuples could be used when returning multiple values from functions. However, lists can be used for the same purpose, so tuples would not add anything of value to Daspel.

5.14.3 Type Annotation For Function Parameters

Allowing the user to add type to function parameters would make it easier to see what the arguments are supposed to be. They would also reduce the

amount of type checking errors during run time as they would be caught during type type checking That said, Daspel scripts are supposed to be very small, so it would be easy to figure out the parameter types anyway. The user can also just use comments to document the type of the parameters.

5.14.4 Data Type Methods

In the programming language Rust, primitive data types implement certain methods. This also applies for complex data structures such as strings and vectors. This can also be applied to Daspel. Standard library functions for interacting and manipulating Daspel data types could instead be implemented as methods. This would make the language feel more object oriented as opposed to the current mix of imperative, event oriented and functional.

```
1 # std function
2 let x = [2, 3, 1, 4];
3 sort(x);
4 # x is now [1, 2, 3, 4];
5
6 # built in method
7 x.sort();
```

5.15 Unicode Syntax

My supervisor suggested to use Unicode symbols for keywords and function names. The idea was to have a Unicode symbol as an alternative way to refer to keywords, functions, IO, sensors and actuators. Long keywords use more bytes, but a Unicode symbol uses 1-4 bytes, depending on the symbol. Scripts which use Unicode symbols can in most cases be smaller and therefore reduce the amount of data which needs to be transferred to a IoT device. It also means they take up less space in RAM and on disk on the IoT device.

Below you'll find tables of Unicode symbols. Note that only one symbols would be used for each concept. Each table just shows the alternatives. Unicode as keywords never made it into the language specification, at least not at this stage. All symbols were found at <https://unicode-table.com/en/> and <http://getemoji.com/>.

Nil		
∅ U+2205	∅ U+2300	∅ U+00D8
∅ U+2298	⊗ U+2297	⊗ U+2BBE
⊖ U+29B8	⊖ U+1F6AB	✕ U+274C
✕ U+2717	✕ U+2613	× U+1F5F4
× U+1F5F6	⊠ U+1F5F5	⊠ U+1F5F7
× U+00D7	✕ U+1F5D9	× U+2A2F
☠ U+1F480	☠ U+2620	☠ U+1F571
⚠ U+26A0	⊖ U+26D4	🙄 U+1F635
🗨 U+1F44E	🌐 U+1F4A3	🚫 U+1F6A7
🗑 U+1F4A9	⚡ U+2757	

Table 5.3: Before settling on dynamic typing, types had to be written out. This table shows the symbols for the value `nil`. `nil` represents an error or an empty value, so Unicode depicting negative symbols were preferred.

Send	
✉ U+1F4E7	✉ U+2709
📧 U+1F583	✉ U+1F582
📄 U+1F4EE	💬 U+1F4AC

Table 5.4: Send means to send a message. A letter or speech bubble is a good visual representation for sending something.





Power	
 U+1F50B	 U+1F5F2
 U+26A1	 U+1F50C

Table 5.5: The power keyword or function would return a value representing how much battery is left.








Time/Timer	
 U+23F0	 U+23F1
 U+23F2	 U+1F570
 U+231B	 U+23F3
 U+231A	

Table 5.6: A clock represent time and the time function would put the device to sleep for a specified duration-












Sound (microphone)		
 U+1F3A4	 U+1F508	 U+1F509
 U+1F50A	 U+1F3B5	 U+266A
 U+266B	 U+1F3B6	 U+1F39C
 U+1F3A7	 U+1F442	

Table 5.7: The microphone, ear and headphones gives a clear indication on which sensor to use.

Sound (speaker)












 U+1F3A4	 U+1F508	 U+1F509
 U+1F50A	 U+1F3B5	 U+266A
 U+266B	 U+1F3B6	 U+1F39C
 U+1F399	 U+1F444	

Table 5.8: Play sound.

GPS









 U+1F310	 U+1F30E
 U+1F30D	 U+1F30F
 U+1F5FA	 U+1F6F0
 U+1F6F0	 U+1F4E1

Table 5.9: Planet Earth and satellites were the best fit for GPS.

Temperature


 U+1F321

Table 5.10: The simplest and most elegant representation for temperature is a classic thermometer.

Humidity


 U+1F4A7  U+1F4A6

Table 5.11: It was hard to find a good symbol for humidity. Water droplets were the most accurate as humidity represent the amount of water vapour in the air.

Pressure

















 U+1F32C	 U+1F300	 U+2601
 U+1F327	 U+26C5	 U+1F326
 U+1F32A	 U+1F32B	 U+1F301
 U+1F38F	 U+2193	 U+21E9
 U+2B07	 U+1F83B	 U+1F847
 U+1F89B		

Table 5.12: Pressure was also hard to represent. In this case, pressure represent air pressure, so clouds and downwards arrows were were the best thing I could find.

LED (single or display)









 U+1F6A6	 U+1F4A1
 U+2600	 U+263C
 U+1F308	 U+1F526

Table 5.13: The lamp is probably the best alternative to represent an LED light.

Light (sensor)

 U+2600

 U+263C





 U+1F308

Table 5.14: The light sensor measures light. It does not emit light, but senses it, so finding an accurate representation was tricky.

Monitor

 U+1F4FA

 U+1F5B5

 U+1F4BB





 U+1F5B3




Table 5.15: A monitor is rather straight forward.

On/Off

✓ U+1F5F8 ✓ U+2713 ✓ U+2714

✗ U+2716 ✗ U+2717 ✗ U+2718

 U+1F197  U+270B  U+1F44D

 U+1F44E  U+1F592  U+1F593


 U+1F44C

Table 5.16: On and Off represent the action of turning a device on or off. They are also meant to symbolise the action accept and reject.

Loop	
♻ U+21BA	♻ U+21BB
♻ U+27F2	♻ U+27F3
♻ U+2940	♻ U+2941
♻ U+267B	

Table 5.17: Instead of writing `for` or `while`, the user would use a Unicode symbol instead.

IF-ELSE
🤔 U+1F914

Table 5.18: A thinking face symbolises a choice. The problem was that no good alternatives for `else if` and `else` were found.

Function	
λ U+03BB	λ U+1D6CC
λ U+1D77A	f U+1D453
f U+1D487	

Table 5.19: The function symbol represents the function structure. Lambda is often used for functions and closures.

Part III

Implementation

Chapter 6

Proof of Concept

6.1 The Goal of the Implementation

The goal for the implementation is create a proof of concept for the Daspel interpreter.

In the following chapters we'll look at the hardware the interpreter will run. We'll also look at the provided sensors and changing the pixels on the display. Finally, we'll go through the code implementation.

6.2 The Hardware

This thesis uses a Raspberry Pi 3 Model B (RPi) as a platform for developing and running the interpreter. The Sense HAT add-on board sensors are used for testing environment readings by the scripts.

6.3 The Interpreter

The interpreter consists of several parts.

- Scanning, parsing and analysing Daspel scripts.
- Generating an Abstract Syntax Tree.
- Setting up an event queue.
- Poll an event from the queue and interpret it.

Basically, when the device receives a new script, it has to parse it first. The interpreter scans the new script and checks for syntactical errors. After this is done, an Abstract Syntax Tree (AST) is generated. This tree is a data structure which represents the script. The interpreter uses the AST to reason about the program. From here, the interpreter has to verify that the scrip (AST) does not contain any illegal actions, such as calling an undefined function or using an undefined variable. Then an event pool is initialised. The pool contains events which the interpreter can run. Events from the sensors, the script and IO is put into the event pool.

6.4 Source Code

The code for the implementation can be found on my GitHub page.

- <https://github.com/CodeGradox/Master-interpreter>
- <https://github.com/CodeGradox/daspel-nom>
- <https://github.com/CodeGradox/sensehat-rs>
- <https://github.com/CodeGradox/Rust-Sense-Hat>
- <https://github.com/CodeGradox/IoT-Interpreter>

6.5 Disclaimer

The implementation is not finished because I ran out of time. Nonetheless, I will describe what I managed to implement and how I did it.

Chapter 7

The Raspberry Pi and the Sense HAT

7.1 Raspberry Pi 3 Model B

The Raspberry Pi 3 is the third generation Raspberry Pi which is part of a series of small single-board computers, about the size of a credit card [36, 37]. The RPi is a powerful device fully capable of running an operating system such as a Linux distribution. The device provided for this thesis came with an SD-card with the Raspbian operating system installed. Raspbian is based on the Debian operating system and is optimised for the Raspberry Pi hardware [38]. It comes with a set of software tools which can be used for education, programming and general use [36]. The full list of the hardware specification for the Raspberry Pi can be found in figure 7.1.

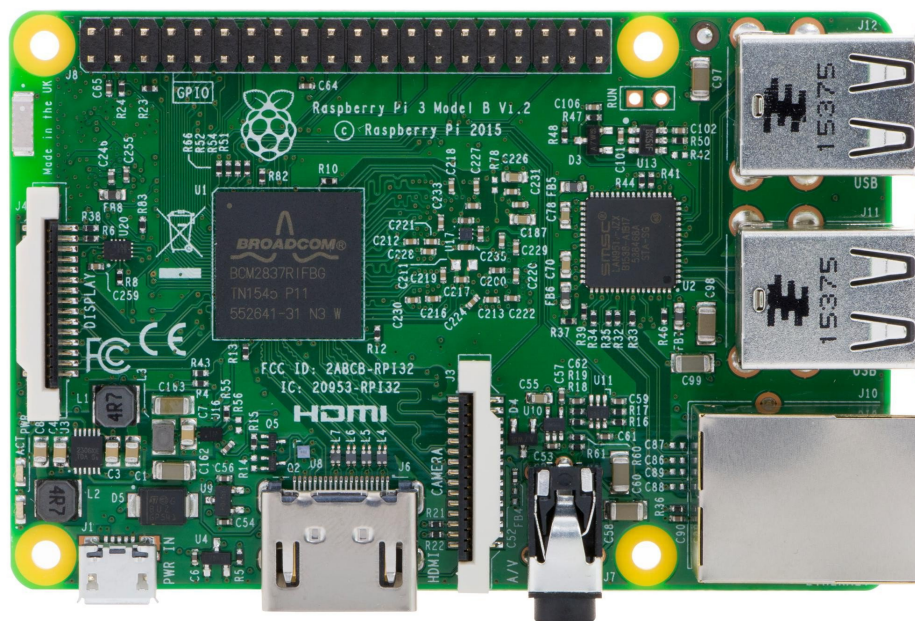


Figure 7.1: The Raspberry Pi 3 Model B. Source [39]

- A 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 1GB RAM
- 4 USB ports
- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot
- VideoCore IV 3D graphics core

Table 7.1: The Raspberry Pi 3 Model B specification [36]

7.1.1 Software

For this project we had to download and install some additional software to be able to implement the Daspel interpreter. The two most important pieces of software were the Sense HAT Python library, which comes with the RTIMULib C++ library, and the Rust compiler. The Rust compiler was installed using rustup which is a tool-chain installer for Rust. Rustup also provides Cargo, the package manager for Rust which manages external dependencies for a code project. Other tools which were necessary for developing on the RPi were SSH, GIT and Vim, but only Vim needed to be installed as the former two were already pre-installed by Raspbian.

7.2 Sense HAT

The Sense Hat is an add-on board for the Raspberry Pi, made specifically for the Astro Pi mission [40]. The board features a LED display, a joystick and six sensors which can be accessed through a Python library. A full list of features can be found in figure 7.3.

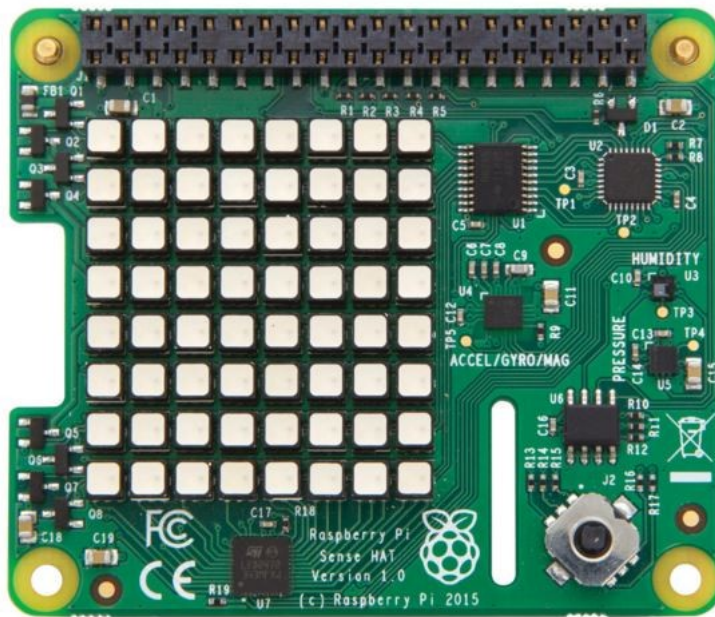


Figure 7.2: The Sense HAT. Source [41]

7.2.1 Sensor Communication

Both the sensors and the 8x8 RGB LED matrix uses the I²C protocol for communication. On the RPi, the I²C has to be enabled before the Sense HAT sensors can be used. The Linux kernel provides C functions for I²C communication, making it relatively easy to write a custom read and write functionality to the Sense HAT.

7.2.2 Byte Encoding

All data on the Sense HAT is encoded in little-endian.

7.2.3 The Sensors

Although the Sense HAT website specifies six different sensors, the board only has three sensor chips. The Pressure sensors is a LPS25H sensors, the relative humidity sensor is a HTS221 sensors and the 9 Degree of Freedom (DoF) sensors is a LSM9DS1 sensor [42]. Both the LPS25H and the HTS221 sensors are capable of reading temperature from pressure and relative humidity respectively. The LSM9DS1 is an IMU, which stands for Inertial Measurement Unit [43]. The IMU is actually three sensors: a 3D accelerometer, 3D gyroscope and a 3D magnetometer. The user can use the data from the IMU to assert the movement the Sense HAT is experiencing [43]. The picture in figure 7.3 displays the three axes the Sense HAT can detect.

- LPS25H (Pressure)
- HTS221 (Humidity)
- LSM9DS1 (Accelerometer, gyroscope and magnetometer)

Table 7.2: The Sense HAT sensor modules [42]

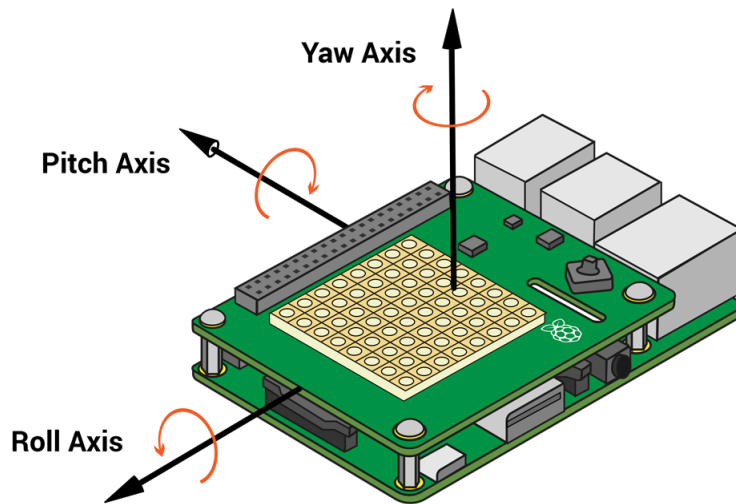


Figure 7.3: The orientation. Source [43]

7.2.4 The LED Matrix

The 8x8 RGB LED matrix is controlled LED2472G connected to an Atmel ATTINY88 communicating via I²C with the RPi [42]. The LED matrix has a 60 fps refresh rate and 15-bit colour resolution [44]. To use the LED matrix, one does not need to use the I²C protocol directly. Instead, the framebuffer of the display can be read from file or be mapped to memory with mmap. A framebuffer is a portion of RAM containing a bitmap that is used to refresh video display from a memory buffer containing the complete frame of data [45]. In the Sense HAT Python library the framebuffer is read from file first. Then new data is written to the file, which in turn updates the pixels on the LED matrix.

High Byte							Low Byte								
R7	R6	R5	R4	R3	G7	G6	G5	G4	G3	G2	B7	B6	B5	B4	B3

Figure 7.4: The RGB 565 bit field after a RGB 888 value has been encoded by the Python library.

RGB 565

The pixels on the LED display use the colour format RGB 565. Each pixel requires 16 bits of data where 5 bits are used for red and blue values while 6 bits are used for green values. In figure 7.4 we see how each pixel is formatted by the Python Sense HAT library. More specifically, the library accepts colours formatted as RGB 888, which is the standard colour format used on the web [46], and compresses them to the RGB 565 format. Because RGB 565 is 1 byte smaller than RGB 888 (24-bit), some data is lost during the conversion. Red and blue values lose their 3 least significant bits, while green values only lose 2.

7.2.5 RTIMULib

The Python Sense HAT API provides an easy to use interface for the Sense HAT board. However, the sensor data is not accessed by the API directly, but through a C++ library called RTIMULib [47], which is maintained by the Raspberry Foundation. The library is written in object-oriented C++ and it uses Linux specific C libraries for communication over I²C. RTIMULib is capable of detecting several kinds of environmental sensors and IMUs, including those found on the Sense HAT board. RTIMULib also creates a settings file which allows the user to change the settings for each sensor.

- An 8x8 RGB LED matrix
- A five button joystick
- Gyroscope
- Accelerometer
- Magnetometer
- Temperature
- Barometric pressure
- Humidity

Table 7.3: The Sense HAT specification [40, 48]

7.2.6 Errors In Reading Temperatures

Due to the design of the Sense HAT board, it is impossible to read the exact temperature from the Sense HAT when mounted directly above the RPi. The temperature readings will show values which are much higher than what they are supposed to be. This is caused by the heat generated from the RPi CPU and the hot air around the RPi ends up affecting the readings of the sensors. To counter this phenomenon, the Sense HAT

would have to be separated from the RPi or the readings would have to be calibrated. However, accurate readings were not a primary concern for the implementation, so the issue was ignored.

Chapter 8

Implementation

8.1 Criteria

The criteria for the Daspel interpreter are as follows:

- The binary size should be as small as possible.
- It should have a small memory footprint.
- It must use as little power as possible.
- It should be able to run on any IoT device.
- It must be able to switch running scripts.
- It must be robust, meaning it should never crash.

8.2 Implementation Language

The first step of writing the Daspel interpreter is to choose a programming language. Based on the criteria in the section above, my supervisor and I decided to not use languages which utilises garbage collection for memory management. This excludes languages such as Python and Java or interpreters such as Node.js. One of the reasons is that these languages have a larger memory footprint than languages with manual memory management. The second reason is that garbage collection takes time and can in some implementations cause the program to pause during runtime. IoT devices use systems operating in real time, so having a pause to reclaim memory is not a scenario we want to deal with. That said, Java is the most popular programming language for IoT development, followed by JavaScript in 3rd place [28]. It is possible that the IoT devices which are running Java are on par with the Raspberry Pi, so memory is not that big of an issue.

In other cases Java and JavaScript are used on the server and the gateway, meaning the IoT device don't use these languages directly. On another note, it would be possible to communicate with the Sense HAT

device through the I²C protocol with Java, JS and Python through external dependencies. This makes it possible

Given the time limit for the thesis, it's preferable that the implementation language makes it easy to represent dynamical sized strings and lists. The language or its ecosystem (third party libraries) should provide support for Unicode text. The less time is spent on dealing with those the more time can be spent on the implementation.

8.2.1 Python

Python was briefly considered even though the language is interpreted and uses automatic memory management. A reason to choose Python is because the Sense HAT API is written in Python. This would mean that only the scanner, parser and interpreter would have to be implemented.

8.2.2 C

C is the second most used programming language in the world [49]. C is also the second most used programming language used in IoT [28]. The C language is small and simple. It has a small memory footprint and performs very well and it can generate small binaries. The language allows the developer to interact closely with the hardware. As we can see, C is a very promising choice. It fits the requirements perfectly. It is also portable and it will most likely support more platforms than its contenders. However, it was not chosen due to the following reasons.

Originally, C was planned to be used as the implementation language for the Daspel interpreter. However, I am not a skilled C programmer, nor am I too fond of C. This is admittedly one of the reasons why I did not want to use C. I also did not want to spend time on implementing dynamically sized lists and strings in C. Furthermore, I would also have to deal with manual memory management. By this I mean I would have to keep track of `malloc` allocations and I would have to ensure that I'm using `free` in the right places. I have mostly been using Java during my time at the university and therefore have very little experience with manual memory management. To be clear, I am aware of tools which can check for memory leaks, use after free and undefined behaviour in C programs. These tools would definitely have been a great help.

Lists may be easy to implement by using linked lists. Strings, on the other hand, must support Unicode characters. As a result, the parser must also have to be able to parse Unicode symbols.

Daspel is dynamically typed and the interpreter must therefore be able to represent dynamic types. The easiest way to do this is by using tagged unions as shown in figure 8.1. With tagged unions it becomes possible to represent multiple data-types through a single struct. It also makes it possible to change a dynamic variable to another type by changing the union value and swapping the enum tag. The issue is that I would have to be careful to always check the tag before changing the union value as the value can be a pointer.


```

1 #include "stdio.h"
2 #include "stdbool.h"
3
4 typedef enum TypeKind {
5     Integer, Boolean, String
6 } TypeKind;
7
8 typedef struct {
9     enum TypeKind kind;
10    union Value {
11        int num;
12        bool boolean;
13        char *str;
14    } value;
15 } Type;
16
17 Type type_new_int(int v) {
18     Type t;
19     t.kind = Int;
20     t.value.num = v;
21     return t;
22 }

```

Figure 8.1: A tagged union in C

Finally, the RTIMULib cannot be used by C directly as it is implemented in C++. C cannot call C++ functions. This is due to the name mangling of function and method names done by the C++ compiler. Using the extern "C" construct will prevent the C++ compiler from name mangling function names.

One solution is to rewrite the necessary parts of RTIMULib to C. A quicker solution is to write a C wrapper. The wrapper acts as an interface which allows C to make indirect calls on C++ functions. The wrapper is just a C header file which has its function implemented in C++. Both the C++ code and the wrapper are compiled together by a C++ compiler such as gcc or clang. A C program can then simply use the header file and call the header functions like normal C functions.

8.2.3 C++

Like C, C++ is a popular language used for IoT applications and it ranks 6th on the popularity list [28]. With C++ I'd be able to write the interpreter in pretty much the same fashion as I would do with C. The main benefit of using C++ is that the RTIMULib library can be used directly. C++ also has the `std::String` and `std::Vector` types defined in its standard library. Both of these types can allocate data on the heap and re-size themselves automatically. These types also implement a destructor method, meaning that when a variable which owns a `std::String` or `std::Vector` goes out of scope, the allocated data is freed automatically. This makes it easy to

implement Daspels string and list types. The issue is that it is not possible to use these types inside of a union as unions only support primitive data-types. To solve this, I'd have to use the Boost library and use the Any type. The Any type is capable of holding more complex data-types and provides a safe way to check the type of the value.

I ended up not choosing C++ because I did not want to depend on the Boost library for the implementation and I was not sure it would even compile on the Raspberry Pi. Furthermore, I have zero experience with C++ programming and I'm slightly biased against C++. That said, I believe C++ is a good candidate due to its portability, speed and memory footprint.

8.2.4 Rust

Rust is a systems programming language developed by Mozilla. It had its first stable release in 2015 [50]. Rust aims to compete against C++ in terms of performance, safety and memory consumption [51]. Safety in this context means that Rust guarantees type soundness, memory safety and prevents data-races. The Rust compiler prevents most, if not all unsafe operations during compile-time by using static analysis and keeping track of data ownership.

Rust has `String` and `Vec`, which are data structures similar to `std::String` and `std::Vector`. The difference is that Strings in Rust are UTF-8 encoded. Rust also has built in tagged unions which are called `Enum`. In contrast to C/C++ unions, Enums in Rust can store any type, be it a struct, pointer or a plain data type. These Enums work in a similar fashion as `datatype` does in Standard ML.

Rust can call functions in C code, but it cannot call C++ functions. To make a Rust implementation work with the RTIMULib, it must either use a C wrapper or re-implement the RTIMULib in Rust.

The Rust compiler relies on the LLVM compiler back-end to translate its intermediate representation (IR) to assembly [52]. The C/C++ compiler, Clang, also uses LLVM. As such, it's possible that Rust and C++ code can be compiled to similar binaries. The issue with the Rust compiler is that it uses static linking instead of dynamic linking which can result in larger binaries. This may be an issue for certain IoT platforms with very limited space capacity.

Other than that, Rust usually performs similarly to C++ code in terms of execution speed. Some benchmarks¹ comparing Rust, C++ and C can be found on The Computer Language Benchmarks Game².

!!!! BOX PLOT HERE! !!!!

I ended up choosing Rust due to the reasons mentioned above and because I am more familiar with Rust programming. Rust handles most, if not all memory management by itself and it enforces safe coding practises thanks to static analysis during compile-time.

¹Benchmarks are not the most optimal way of comparing language performance, so please take the results with a grain of salt.

²<https://benchmarksgame.alioth.debian.org/>

8.2.5 Conclusion

The implementation language is not too important for the interpreter as it's only a proof on concept which will run on a fairly powerful device. That said, if the goal was to create a more realistic implementation, I would most likely have chosen C over Rust. This is because C has support for vastly more CPU architectures, which makes it more portable.

8.3 Work setup

There are at least two way to interact with the Raspberry Pi. The first one is to connect a monitor, a keyboard and a computer mouse directly to the RPi. This allows us to use the RPi as a normal computer as the Raspbian operating system features a graphical user interface. The desktop environment can be disabled if the direct use of the command line interface is preferred.

The second approach is to connect to the Raspberry Pi with SSH through a remote computer. This is easy to set up at home where it is possible to set a static IP address on your personal network. On the other hand, the Raspberry Pi does not seem to be able to connect to the Eduroam WiFi on IFI. A solution is to use an Ethernet cable and connect the RPi to a computer and run SSH over the local connection. For Windows, there exists a program called MobaXterm which is a graphical user interface for SSH connections. It can also open remote files in a local editor. This allows you to use programming environments on Windows for writing to program files located on the Raspberry Pi if you don't want to use Vim or Emacs over normal SSH.

When I first started working on the Raspberry Pi I was using the first approach. After a while it too cluttered to have two keyboards, two computer mouse and two monitors all on the same desk, so I switched over to using SSH instead. I started to use MobaXterm after I started to work on the Rust implementation.

8.3.1 Required Software

The only important piece of software which must be downloaded is the Rust compiler, as it is not part of the Raspbian distribution. For testing and asserting that the Rust code is interacting properly with the Sense HAT device is by using the Python Sense HAT module. This module must also be downloaded, but unlike Rust it is available through the Raspbian package manager. The RTIMULib library is installed together with the Python module.

The Python module installs a few example programs which show how to use the Python module and the RTIMULib C++ library. These programs were useful when trying to understand how to work with the Sense HAT.

8.4 Porting the Sense HAT library to Rust

Initially the interpreter was supposed to be written in C. The work first started with figuring out how to use the C++ library RTIMULib from C.

The code in RTIMULib is programmed as object oriented C++ and it provides several classes which represent the sensors and IMUs. Through the provided example code we see that a Settings object is created first. The Settings class contains information about calibration, polling frequencies and I²C addresses for the supported sensors and IMUs. The Settings class initially tries to find and read a .ini file which contains the settings. If no such file is found, it will create a local RTIMULib.ini file and save the default settings in it. The Settings object is passed to a Pressure, Humidity and Imu objects, which use the Settings for calibration. Each sensor and IMU supported by the RTIMULib has its own class. When using the Sense HAT, RTIMULib will create an instance of the LPS25H class, the HTS221 class and the SM9DS1, which are subclasses of Pressure, Humidity and IMU respectively. All three classes contain methods with specific algorithms for reading and writing to their designated hardware component.

8.4.1 Writing the C wrapper

As stated earlier, it is not possible for C code to call C++ functions directly. To be able to use a C++ library from C a wrapper must be created first. Through the wrapper it becomes possible to convert C++ types to C types. The wrapper provides an interface which C code can see and use to interact with C++ code. Essentially, C simply calls the declared wrapper functions declared in a header file. The actual implementation of these functions resides in C++ files which follow the C++ conventions.

Since classes don't exist in C, the wrapper must use special pointers, called opaque pointers, which C can use instead. To make things easier, a single Wrapper class is created. This class contains all of the four objects required to use the Sense HAT, with methods for reading and writing data to the sensors and the IMU. Since C has no concept of classes, it also has no concept of methods. It is not possible to use the opaque pointer of the wrapper class as a C++ object directly in C. Instead, the wrapper declares C functions which takes an opaque Wrapper pointer as a parameter. The functions then cast the opaque pointer to the appropriate C++ class and call the appropriate method before finally returning.

There is an issue with calling C++ code from C: exceptions. C has no concept of exceptions, let alone C++ exceptions. Thus, if the C++ were to throw one, the C code would not be able to handle it. In the best case, the program would just crash. To prevent this, the wrapper functions should try to catch all exceptions, both when casting the opaque pointer and when calling the object methods. The implementation of the C wrapper never came far enough to provide a good mechanism to relay a message to C code about errors which may have occurred.

The methods for the Wrapper class were imitating the Python implementation of the Sense HAT module and the wrapper did in fact work. In retrospect it would have been possible to port the C++ implementation almost directly to C code instead of writing a wrapper. RTIMULib use C functions for reading and writing over the I²C protocol and all the calibration and calculations can easily be implemented in C. However, this could require some time to do as there is a lot of calibration and calculations required for reading the IMU. The pressure and humidity sensors are rather simple to interact with in comparison.

8.4.2 The Framebuffer

The Sense HAT module implements functionality to interact with the LED matrix. It can write one pixel at a time, fill the whole screen or clear it along with rotating the currently displayed image. It's also capable of displaying 8x8 pixel images and text.

We will look at two ways of accessing the LED matrix framebuffer. The first approach is use by the Python module and involves writing to the framebuffer as a file, while the C approach is to memory map the framebuffer.

The Python module first finds and saves the path to the framebuffer when it is initiated. It never writes new pixel data to the framebuffer directly. Instead, it writes all new data to a two dimensional numpy matrix which acts as the current frame. The module writes this frame to the framebuffer just like it would write data to a normal file. The Sense HAT devices updates the LED matrix with the new frame automatically. It's worth noting that the framebuffer file is reopened every time the program updates the frame. Frequent call on the drawing functions may lead to slower execution times because the program has to wait for IO operations to read the file.

With memory mapping, we're able to write to the framebuffer directly. An example on how to use the framebuffer in C can be found in the provided Snake game which is located in the Sense HAT example folder. The games uses the LED matrix to display the current game state. The approach is similar to the Python module as it has to locate the correct framebuffer first. It uses a struct with a 16 bit 8x8 array as a frame. The framebuffer is then memory mapped to the struct which is located on the stack. The program can then access any index of the frame and read or write its data directly. Any change done to the array results in a change in the framebuffer, which in turn changes the currently displayed image.

For the C implementation I choose to use permanent memory mapping. This means that the framebuffer is memory mapped for the whole duration of the program. This is because I intended access to the framebuffer to be as fast as possible at. However, this may not be the best solution as the target devices have very little memory available. It's also subpar solution if the display is never used by the script. A better solution would be to have the interpreter calculate how often the script draws to the screen. It can then determine if it should open the framebuffer permanently or not.

8.4.3 Reimplementing the Sense HAT module from C to Rust

A Rust implementation for interacting with the LED matrix was made in parallel with the C implementation. It did turn out that there already existed a Rust module for interacting with the framebuffer. This module uses a Rust wrapper for C functions, such as `mmap`, and provides an interface for writing to the memory mapped region. Rewriting the C functions from my C implementation of the Sense HAT was fairly trivial. Both implementations ended up having the same functionality.

At some point the implementation switched focus from C to Rust. By then, only the C code was able to interact with the sensors thanks to the `RTIMULib` wrapper. The Rust implementation was supposed to use the C wrapper as Rust does not have a Foreign Function Interface (FFI) to C++. This means Rust has to go through the C code to interact with C++, while the C implementation interacts with C++ directly. At the end of February a Rust module for the Sense HAT was released [53]. It only provides functionality for reading pressure and humidity in its current release. This module was forked and the framebuffer functionality was added. Work was started to add the ability to read IMU data, but at this point time was running short and the Rust Sense HAT module had to be left unfinished. It's worth mentioning that the third party Rust Sense Hat module does not use a wrapper for the `RTIMULib` library, but instead copies the C++ `RTIMULib` implementation directly. In other words, it uses the I²C protocol to communicate with the sensors and LED matrix on the Sense HAT board.

8.5 The Lexical Analysis and Syntax Analysis

The first phase of the interpreting starts with converting the program text into an Abstract Syntax Tree (AST). This is the scanners job. The first step is to convert sub-strings to tokens while skipping white-spaces, newlines and comments. It is the scanners job to perform scan and generate the tokens. The parser uses the tokens to generate a parse tree (the AST). It is responsible for verifying that the program is following the correct syntax. Any errors encountered during this phase returns an error. For example, the scanner can find an illegal symbol while the scanner can detect an illegal sequence of tokens.

8.5.1 The Scanner

No work was conducted on the scanner until the syntax of Daspel was starting to take shape. At this point, the implementation language had changed to Rust. Some example scanners written in Rust were found on the Internet and were used for inspiration. The core mechanic of the scanner is that it behaves like an iterator. The input program is read from file and is stored in a string. The scanner object takes a string reference of the input string and creates a peekable char iterator of the string. A char in Rust is a 32-bit UTF-8 encoded character. More specifically, a char is a Unicode scalar value [54]. The scanner has a method named `next_token` which

returns a Token. A token is an enum type which can represent keywords, data types and identifiers. When the `next_token` is called, the scanner first peeks at the next char in the file. It then performs a pattern match on the char to figure what action to perform. For example, if it peeks a `"` then it knows it is the start of the string literal. The scanner then calls the method responsible for scanning string literals which in turn return a string literal token. The scanner is finished when the internal iterator is depleted, which happens when the end of file is reached (the end of the input string). The scanner is very modular and it makes it easy to add or remove tokens to look for.

The scanner reads the whole file in to a string instead of reading it line for line. The problem is that it has to allocate enough space to fit the whole input, instead of just allocating one line at a time. That said, if the whole script does not contain any newlines, the whole file would end up being read to RAM anyway. This potential issue is not a problem for the Raspberry Pi considering its specifications, but for a very small device it could become an issue if the scripts can become very large in size. The second issue is that the current implementation of the scanner does not allow it to rewind. This means that once some char or text has been read, it cannot be read again. This is due to how iterators work. A workaround would require changing how the scanner iterates over the string. Instead of creating a char iterator, the scanner can keep track of the current position and the last token position. It moves the `cur_pos` index based on a predicate and the chars between the two indexes is a sub-string. The generated tokens are based on the semantic meaning of the sub-strings. This is the approach the Go Language Lexer and the Rust Lexer use. This approach allows the scanner to backtrack and peek ahead more easily than our implementation, but it must also take Unicode symbols into considerations when scanning.

8.5.2 The Parser

The parser was never finished because there was not enough time left to implement it. I started working on two different implementations; a handwritten parser and a parser generator. The first parser was supposed to simply accept a string reference (of the input file) and internally create a scanner object. The parser would then use the scanner to generate tokens. It would then use tokens to determine the current syntax construct. For example, a token `Token::Function` would mean the parser is looking at the start of a function statement. The parser will then call the appropriate syntax handler method which will check the incoming sequence of tokens. If the sequence is syntactically correct, the method will return an AST object. This parser is a recursive decent parser. This means syntax methods use recursion to traverse the input. As such, the implementation will resemble the EBNF notation of the Daspel syntax.

The second parser variant is created by using the `nom crate`³ [55] which

³Crates are what libraries are called in the Rust ecosystem

is a parser combinator. Nom allows the user to create small parsers and combine them to make bigger parser, which is done with the help of Rust macros. When the macros are expanded during compilation, they generate a big state machine which can parse a u8 slice (a `char*` in C). Using nom makes the scanner redundant, as nom acts as both a scanner and parser. The parser macros end up resembling the EBNF notation they are representing.

The downside of using nom is that it expands to a very big state machine. More specifically, it generate many lines of pattern matching statements. As a result the binary file can become larger than the binary of a hand written scanner and parser. On the flip side it becomes relatively easy to add new parsers for each language construct. This parser is, in its current state, only capable of parsing expressions.

8.6 Representing the Data Type Real in Rust

A simple implementation for parsing reals was added when I was working on scanner. The `real` data type is represented as a struct in the implementation. The struct is called `Real` and it has one member of type `i32`. The `i32` is a signed two complement integer in Rust. Note that there is no built in type `real` in Rust, only floats. The struct has a method which takes a string as an argument and produces a `Real`. The string must be a textual representation of a decimal number.

8.6.1 Arithmetic Operations

Arithmetic operations with `Real` is rather trivial. Subtraction and addition is done in the same manner as you'd do with regular integers. No additional operations are required. On the other hand, Multiplication and division are a bit trickier. Fixed point numbers can be represented by the Q number format [33]. Our `real` is a fixed point number with the format Q15.16. When we multiply two `reals`, the products format becomes Q30.32 [56]. This means the product requires 62 bits, which is obviously way more than our `i32` can represent. To handle this, we need to cast both factors to a `i64`, multiply them, shift the product 16 bits to the right and finally cast it back to a `i32`. For division, if we just divide the two `reals`, we'd lose the fractional part. The process is almost identical to multiplication. We cast both to `i64`, shift the dividend 16 bits to the left, divide and finally cast the quotient to `i32`. Sadly, both multiplication and division can result in loss of fractional bits, aka loss of precision. As you may have noticed, we are only using integer operations. This means that the operations are fast, much faster than floating point arithmetic, especially if the device does not have an FPU.

The `Real` type can be used in arithmetic operations together with numbers of type `i32` and `f32`. This makes it easier to deal with integers and reals during the interpretation as Daspel allows the two data types to be used together in arithmetic operations. A variable `a` of type `Real` can

be used like this: `a + 3 - (2 * a)` or `3.14 + a + a - 1.2`. This works because Rust will translate an arithmetic operation to a method call. For example, `a + 3` becomes `a.add(3)`.

Part IV

Summary

Chapter 9

Conclusion

The aim for this thesis was to create a programming language suited for the Internet of Things. The language had to be small, simple and easy to use by people who are new to the field of programming and IoT.

I have looked at the Internet of Things in terms of both sensors and hardware. I have analysed the platform and established the constraints and needs for developing on IoT devices. Furthermore, I have looked at how to create a programming language for IoT which takes these restriction into account.

9.1 Analysis

For this project I definitely spent too much time on trying to write the interpreter instead of focusing on making a solid design for Daspel.

I also think that using a dynamic type system over a static type system was a minor mistake. This is because dynamic typing can lead to run time errors due to type mismatches which is something static typing would avoid. However, I also believe that dynamic typing is more suited for people who are new to programming, which are the target demographic for this project.

9.2 Future Work

There is much work left to be done. First and foremost, Daspel needs to be fleshed out in terms of core mechanics. It also needs a solid standard library. Furthermore, the exact mechanics of the of the interpreter must be fully designed and implemented.

There are quiet a few things which are lacking in Daspel, but it would be exciting to see the language fully implemented and used in the real world. I think Daspel has the potential to be used as teaching material for children to get them interested in programming and also to teach them concepts in programming. IoT is cheep and devices such as the Raspberry Pi is already being used as part of the education in other countries. By adding sensors the those single board computers along with Daspel, the children would

be able to create application which can interact with the real world. I think this type of interaction between hardware, software and education is the most creative way to teach programming to young minds.

Bibliography

- [1] MIT. *Scratch Programming Language*. URL: <https://scratch.mit.edu/> (visited on 04/27/2017).
- [2] *Internet of things*. 2016. URL: https://en.wikipedia.org/wiki/Internet_of_things (visited on 10/21/2016).
- [3] Rob van der Meulen. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. URL: <https://www.gartner.com/newsroom/id/3165317> (visited on 05/23/2016).
- [4] *sensor*. 2012. URL: <http://whatis.techtarget.com/definition/sensor> (visited on 10/07/2016).
- [5] *What is a Humidity / Dew Sensor?* URL: <http://www.futureelectronics.com/en/sensors/humidity-dew.aspx> (visited on 10/21/2016).
- [6] *What is the GPS - Global Positioning System ?* URL: <http://www.sensorland.com/> (visited on 10/21/2016).
- [7] *Low Energy*. URL: <https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy> (visited on 08/26/2016).
- [8] *Wi-Fi Alliance® introduces low power, long range Wi-Fi HaLow™*. URL: <https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-low-power-long-range-wi-fi-halow> (visited on 10/22/2016).
- [9] Rahul. *Internet of Things Wiki*. URL: <http://internetofthingswiki.com/requirements-internet-of-things/236/> (visited on 08/26/2016).
- [10] *What is 3D Integration?* URL: <http://www.3dincites.com/3d-incites-knowledge-portal/what-is-3d-integration/> (visited on 05/23/2016).
- [11] Harald Bauer, Mark Patel, and Jan Veira. *The Internet of Things Sizing up the opportunity*. 2014. URL: <http://www.mckinsey.com/industries/high-tech/our-insights/the-internet-of-things-sizing-up-the-opportunity> (visited on 05/23/2016).
- [12] *Bluetooth*. URL: <https://en.wikipedia.org/wiki/Bluetooth> (visited on 10/22/2016).
- [13] *Google Beacons*. URL: <https://developers.google.com/beacons/> (visited on 10/22/2016).
- [14] *iBeacon for Developers*. URL: <https://developer.apple.com/ibeacon/> (visited on 10/22/2016).

- [15] *Bluetooth Low Energy*. URL: <https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy> (visited on 10/22/2016).
- [16] *Wikipedia Bluetooth Low Energy*. URL: https://en.wikipedia.org/wiki/Bluetooth_low_energy (visited on 10/22/2016).
- [17] *Bluetooth® 5 quadruples range, doubles speed, increases data broadcasting capacity by 800%*. URL: <https://www.bluetooth.com/news/pressreleases/2016/06/16/-bluetooth5-quadruples-rangedoubles-speedincreases-data-broadcasting-capacity-by-800> (visited on 10/22/2016).
- [18] *Examining The Future Of WiFi: 802.11ah HaLow, 802.11ad (& Others)*. 2015. URL: <http://www.link-labs.com/future-of-wifi-802-11ah-802-11ad/> (visited on 10/07/2016).
- [19] *WiFi range*. 2015. URL: <http://mwrf.com/active-components/what-s-difference-between-ieee-80211af-and-80211ah> (visited on 10/07/2016).
- [20] *AWS IoT Button*. URL: <https://aws.amazon.com/iot/button/> (visited on 10/22/2016).
- [21] *What's the Difference Between IEEE 802.11af and 802.11ah?* 2015. URL: <http://mwrf.com/active-components/what-s-difference-between-ieee-80211af-and-80211ah> (visited on 10/22/2016).
- [22] *Documentation*. URL: <https://www.scriptr.io/documentation> (visited on 03/31/2017).
- [23] *Libelium launches Waspote Plug & Sense*. URL: <http://www.energyharvestingjournal.com/articles/4843/libelium-launches-waspote-plug-and-sense> (visited on 10/25/2012).
- [24] *Waspote Plug & Sense! Catalogue*. URL: http://liste.raspibo.org/www/d_read/meteo/libelium/waspote_plug_and_sense_catalogue.pdf (visited on 03/31/2017).
- [25] *TESSEL 2*. URL: <https://tessel.io/> (visited on 04/27/2017).
- [26] *Tessel 2*. URL: <https://www.sparkfun.com/products/13841?ref=tessel.io> (visited on 04/27/2017).
- [27] *Build cross platform desktop apps with JavaScript, HTML, and CSS*. 2017. URL: <https://electron.atom.io/> (visited on 04/27/2017).
- [28] *IoT Developer Survey 2016*. 2016. URL: <https://www.slideshare.net/lanSkerrett/iot-developer-survey-2016> (visited on 04/14/2017).
- [29] *Node.js*. 2017. URL: <https://en.wikipedia.org/wiki/Node.js#Threading> (visited on 04/27/2017).
- [30] Seth Thompson. *Introduction*. 2015. URL: <https://developers.google.com/v8/> (visited on 04/27/2017).
- [31] *TwoHardThings*. URL: <https://martinfowler.com/bliki/TwoHardThings.html> (visited on 03/29/2017).
- [32] *Floating-point unit*. URL: https://en.wikipedia.org/wiki/Floating-point_unit (visited on 01/29/2017).

- [33] *Q (number format)*. 2017. URL: [https://en.wikipedia.org/wiki/Q_\(number_format\)](https://en.wikipedia.org/wiki/Q_(number_format)) (visited on 05/01/2017).
- [34] *Variadic function*. 2017. URL: https://en.wikipedia.org/wiki/Variadic_function (visited on 05/01/2017).
- [35] *Tuple*. 2017. URL: <https://en.wikipedia.org/wiki/Tuple> (visited on 05/01/2017).
- [36] *RASPBERRY PI 3 MODEL B*. URL: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/> (visited on 01/10/2017).
- [37] *Raspberry Pi*. URL: https://en.wikipedia.org/wiki/Raspberry_Pi (visited on 03/28/2017).
- [38] *Welcome to Raspbian*. URL: <https://www.raspbian.org/> (visited on 01/10/2017).
- [39] URL: <https://d1dr2mxwsd2nqe.cloudfront.net/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/2/5/252522540.jpg> (visited on 03/30/2017).
- [40] *SENSE HAT*. URL: <https://www.raspberrypi.org/products/sense-hat/> (visited on 01/13/2017).
- [41] URL: <https://d1dr2mxwsd2nqe.cloudfront.net/media/catalog/product/cache/1/image/9df78eab33525d08d6e5fb8d27136e95/r/a/raspisensehat3.jpg> (visited on 03/30/2017).
- [42] *Sense HAT*. URL: https://pinout.xyz/pinout/sense_hat (visited on 03/29/2017).
- [43] *Movement*. URL: <https://www.raspberrypi.org/learning/astro-pi-guide/sensors/movement.md> (visited on 04/22/2017).
- [44] *ASTRO PI: FLIGHT HARDWARE TECH SPECS*. URL: <https://www.raspberrypi.org/blog/astro-pi-tech-specs/> (visited on 04/17/2017).
- [45] *Framebuffer*. 2017. URL: <https://en.wikipedia.org/wiki/Framebuffer> (visited on 04/17/2017).
- [46] Tantek Çelik et al. *CSS Color Module Level 3*. 2011. URL: <https://www.w3.org/TR/css3-color/#rgb-color> (visited on 03/30/2017).
- [47] *RTIMULib - a versatile C++ and Python 9-dof, 10-dof and 11-dof IMU library*. URL: <https://github.com/RPi-Distro/RTIMULib> (visited on 03/30/2017).
- [48] *SENSE HAT*. URL: <https://www.raspberrypi.org/documentation/hardware/sense-hat/> (visited on 03/29/2017).
- [49] *TIOBE Index for April 2017*. 2016. URL: http://www.tiobe.com/tiobe_index (visited on 04/14/2017).
- [50] *Version 1.0.0 (2015-05-15)*. URL: <https://github.com/rust-lang/rust/blob/master/RELEASES.md> (visited on 04/15/2017).
- [51] *Guaranteeing memory safety in Rust*. 2014. URL: <https://air.mozilla.org/guaranteeing-memory-safety-in-rust/> (visited on 04/17/2017).

- [52] *Projects built with LLVM*. URL: <http://llvm.org/ProjectsWithLLVM/> (visited on 04/15/2017).
- [53] Jonathan Pallant. *sensehat*. 2017. URL: <https://crates.io/crates/sensehat> (visited on 04/07/2017).
- [54] *Primitive Type char*. URL: <https://doc.rust-lang.org/std/primitive.char.html> (visited on 04/17/2017).
- [55] Geoffroy Couprie. *nom*. 2017. URL: <https://crates.io/crates/nom> (visited on 04/07/2017).
- [56] Gabriel Ivancescu. *Fixed Point Arithmetic and Tricks*. 2017. URL: <http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/> (visited on 05/01/2017).