PROGRAMMING ABSTRACTION FOR IOT DEVICES

YAHUI SONG

(B.Sc. in Computer Engineering, Sun Yat-Sen University, China)

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2018

Declaration

I hereby declare that this thesis is my original work
and it has been written by me in its entirety.
I have duly acknowledged all the sources of information
which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

_____

Yahui Song

April 2018

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Chin Wei Ngan for the continuous support of my master study and related research, for his patience, motivation, and immense knowledge. His guidance helped me all the time of doing research and writing this thesis. I could not have imagined having a better advisor for my master study. Also Prof. Chin provided me an opportunity to join their team as a research assistant. His cheerful mood and optimism kept me going throughout the duration of this project. Without his precious support it would not be possible to conduct this research.

Besides my advisor, I would like to thank all the teachers who offered me lectures during this whole master programme: Prof. Tan Soon Huat Gary, Prof. Dong Jin Song, Dr. Yair Zick, Prof. Mauro Pezze and Dr. Li Xiaoli, for their excellent teaching, insightful comments and encouragement which incented me to widen my research from various perspectives.

My sincere thanks also goes to Dr. Mahmudul Faisal Al Ameen, for his active discussion upon my project, selfless academic sharing and guidance.

I thank all the lab mates, especially Andreea Costea, Ta Quang Trung, Xia Wei, for their generous helping and supporting. I thank all my group mates in different modules, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last one year. And also I thank all those friends around me who always encourage me and accompany me.

Last but not the least, I would like to thank my family: my parents and my sister for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

This thesis provides a solution of abstracting the programming process on IoT (Internet of things) devices. IoT is the network of physical devices, vehicles, home appliances and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these things to connect and exchange data, creating opportunities for more direct integration of the physical world into computer-based systems, resulting in efficiency improvements, economic benefits and reduced human intervention. That is necessary to find out a neat way to make sure the communications between those entities work smoothly and correctly, which can also provide a better user experience and a higher security IoT system. Having a good architecture of building such an IoT system might achieve this goal. Therefore, abstraction aims to get a more systematic and safer approach for constructing IoT systems.

There is a simple yet powerful architecture which is brought up along with the purely functional programming language called Elm which is designed for creating web browser-based graphical user interfaces. The basic architecture of Elm is "Model-View-Update", makes it easy to build high performance web applications. This thesis mainly discusses how to map this neat architecture to IoT programming.

On the other hand, in order to make sure our approach is applicable to real life devices, this thesis also analyses several IoT programs implemented by Node.js based on Raspberry Pi. These programs are used to manipulate several representative IoT devices such as lights, buttons, different kins of sensors and so on. Based on this, this thesis designs a compiler which can compile Elm programs into Node.js files which can be executed directly on Raspberry Pi. The name of this translator is "Compiler from Elm to JavaScript(CEJ)" which is implemented by Haskell. In this way, it is easier and more concise to produce complex and scalable IoT applications using Elm-like code.

From another stand of point, using Elm to accomplish the simulation goal of IoT systems is also useful. For example, in order to test the fire alarm, people do not need to set up

a real fire. Instead, they can use the web interface constructed by Elm to simulate a fire environment and send it to IoT systems. In this way, it becomes easy to test the different scenarios of IoT systems. To achieve this goal, this thesis takes advantage of Raspberry Pi, a powerful tiny Linux OS single-board computer.

All in all, the main contributions of this thesis are: abstraction of IoT programming, construction of the Elm to JavaScript compiler CEJ, implementation of a simulation platform for some IoT systems.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

"We want to do our best work, and we want the work we do to have meaning. And, all else being equal, we prefer to enjoy ourselves along the way."

—— *Sandi Metz*

1.1 Research Background

1.1.1 Programming Languages

The demand of inter-connectivity in this world is rapidly increasing. **And programming languages are the foundation of applications which can connect people together.** No matter you are using social media applications on your mobile phone or online collaborating with your team, the tasks all rely on programming languages. The first programming language was created over 100 years ago by Ada Lovelace, and her contribution on computing marked the beginning of the rich history of programming languages. If we outline the history and evolution of each programming language over these years, we can find tons of information on what kinds of vulnerabilities are most common in programs developed in each programming language and what kind of flaws are most seriously existing there. In the meanwhile, we also can find out lots of outstanding advantages of each language.

Why do programmers need to learn more computer programming languages? The answer is people should keep pace with the times and conform to the trend of the times. The future trend of programming languages is to let each of them stand up to their core application focuses. The designers should both learn and compete with each other, with

the goal of gradually finding more appropriate abstractions and methods which make code easier to write and maintain. A new language represents the understanding of the renewal of things and the better way of elaboration. The essence of languages between human beings and between human and computers are both to express ideas. Language is the tool of human thinking, and people rely on programming languages to command the computer. Different languages represent different ways of understanding the problem domain and the computer system.

It is the change of environmental that inspires these languages and promotes their development. The updated language often includes the advantages of the last generation of language, adds new ideas and methods to solve new problems. Some users may find the new language is more effective and convenient, and this language can take a firmer foothold in some problem domains. From the single-board application to the network application, and from the development of mobile devices to the integration with the network, there are a variety of hardware devices, with different interconnection needs and requirements. In existing programming languages, the original considerations may not be appropriate anymore and new language features may be needed. The so-called "balance point" means where the compromise point is of making design decisions for conflicting characteristics. Do we want a static compiling language or dynamic interpretive language? Is the focus of programming flexibility or running speed? How are security issues being considered? Do we want garbage collection? What mechanism do we use? For example, CPU evolves from single core to multi-core, how to make full use of such hardware during the process of designing a programming language.

Many changes in application requirements will inevitably catalyse the production of computer languages with different design ideas. It is impossible for a single language to solve problems in all fields. Forcing it may only makes the developers suffer, and may greatly reduce efficiency and quality. For example, rare people write the Web applications with C++. One has to take into account the environment, design ideas, development patterns

and idioms which are relied on, so as to make it easier to develop applications using the certain chosen programming language.

1.1.2 Internet of Things

The Internet of Things, abbreviated as IoT, is not a new concept, but it is still a hot topic in the industry. IoT refers generally to the digital interconnection of objects to the internet, and the ability to collect and send data between devices. There are lots of definitions of IoT, most of them focus on how computers, sensors, and objects interact with one another and process data. There are already tons of devices and products designed under the IoT spectrum, and the number of connected devices is increasing exponentially. By 2020, it is predicted that there will be around 50 billion connected smart devices in the world[7][18], as shown in Figure 1.1[10]. If we do a computation, everybody in this world will have 6.5 devices on average. It is universally known that the hype around IoT is huge, and new IoT enabled products are being implemented quickly. Those IoT devices seem impress users with its intelligence, and "Smart Home" stands out as the most prominent applications of IoT. Since IoT devices step into our life deeper and deeper, it is necessary to create a safer and more reliable IoT system.

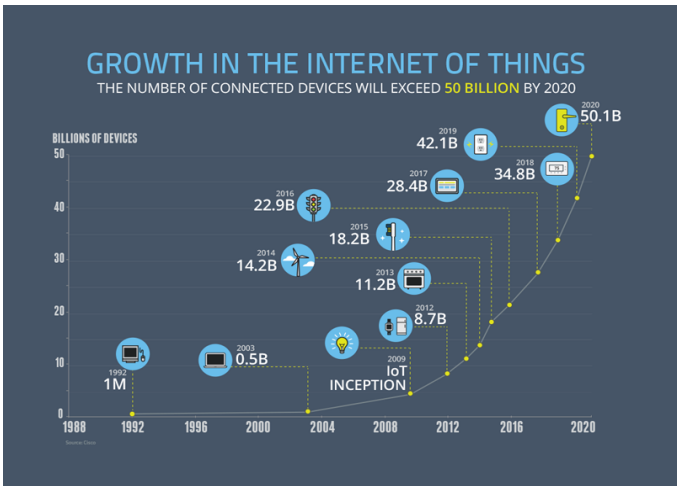

Figure 1.1: Growth of IoT devices

Three important elements contribute to the popularity of IoT nowadays are:

1. **Hardware is becoming cheaper.** Phones are mega sensors that are always with us, but we do not treat them as a true IoT sensor device, yet. In the IoT world, a phone is a collection of many sensors. The price of smart phones continue to drop rapidly.

2. **Connectivity is necessary.** Wherever we go, it is a basic expectation to have wifi connectivity these days. Even in emerging and developing countries.

3. **Development is becoming easier.** While the complexity increases, the tools, libraries, and methodology projects, are making it easier to interact and interface with hardware. The learning curve for development is shortening.

With the current situation of IoT, the vision should be an intelligent connectivity, precise sensing and efficient controlling. Besides, related systems can manage a wide range of physical "smart" devices. A meaningful use is to manage energy and power in buildings. Current industry examples of IoT include smart cars and intelligent building systems, so called "smart home" or "smart city", controlling lighting, heating ventilation air conditioning systems and security systems. Additionally, environmental monitoring of water quality, air pollution and soil conditions are important applications of IoT. Healthcare organisations are also starting to use smart beds, remote health and heart monitoring services and devices. Basically IoT devices could be anywhere, and there are six growth areas, namely:

1. **Household devices monitoring,** such as fridges, washing machines, light or door controllers.

2. **Power management and controlling,** to remotely control devices using a mobile phone.

3. **Entertainment machines,** such as a Bubblino, which is a just-for-fun device can blow bubbles when certain keywords appear on twitter.

4. **Tracking tools,** used for cars, pets, offenders, and other valuable asserts.

5. **Health monitoring devices,** associated with online monitoring tools.

6. **Environmental monitoring tools,** used by thermostats and weather stations for collecting meteorological data.

According to the survey conducted by Eclipse Foundation[9] , the top four languages for developing IoT systems are Java, C, JavaScript, Python. Though they look very similar when constructing either all sorts of applications or servers, there are some differences when it comes to different parts of things which make up IoT. In general, there are three main parts constructing an IoT architectural environment: sensors, to generate the data; gateways, to organise the data; servers, to collect all the data and give a timely respond to them. If it is a basic sensor, programmers typically use C since it works directly with the hardware. For other applications, programmers may choose some other languages which best suit the task at hand.

Currently, one of the most common features of popular languages for IoT is run-time polymorphism. This is an ability to have a set of heterogeneous objects which can all perform a common action. They are popular also because they do not require much power during processing like C. However, one of the biggest issues of IoT is any internet-enabled device is potentially vulnerable to attack from hackers, no need to say the risks when virtually every object and appliance we use is connected. Hackers have successfully hacked into real, on-the-market IoT systems. In contrast, purely functional programming is mainly used on mathematical computations. It is not as popular as imperative languages in IoT development because of lack of good libraries, and the lack of documents and communities. However, it is true that it makes mathematical computations easy to write and read, and makes it easy to verify safety properties because of its syntax. Therefore, nothing should be the barrier which can stop us from exploring more about what can be done better using functional programming languages in an IoT system[25].

1.2 The Current Work

The goal of this thesis is to abstract IoT programming and map it into the architecture of functional programming language Elm[26]. Elm is chosen because of its high level abstraction, and easy for mapping into real devices. The focus of this dissertation is to see how to map the abstraction of IoT into an existing Elm architecture.

Currently, this thesis focuses on one specific target platform Raspberry Pi 3 Mode B and one specific target language JavaScript. However the abstraction and the mapping would be designed to work with all IoT devices, regardless of platform or language.

The main work of this thesis consists of two parts:

1. Construct a compiler which can compile Elm into JavaScript following the abstraction and mapping rules we created previously.
2. Build a simulation platform using Elm for IoT systems based on Raspberry Pi.

The target audience are people who possess basic knowledge about programming and have the interest in IoT programming.

1.3 Thesis Overview

This thesis consists of four main parts:

Firstly, Chapter 1, 2, 3 briefly introduce the research background, current situation of programming languages and Internet of Things, explain the motivation of this thesis, list some related work in recent years.

Secondly, Chapter 4 discusses the preliminary of constructing a compiler and understanding basic IoT devices. Chapter 5 explains in detail on designing the compiler and the simulation system, and it mainly consists following contents: (a) The mapping strategies from IoT

programming to Elm architecture; (b) The abstract syntax tree(AST) of Elm and JavaScript; (c) The translation scheme; (d) The design of the simulation system. Chapter 6 talks about the concrete implementation of the compiler, and all the technologies used in this work.

Lastly, Chapter 7 evaluates the result of this thesis. Chapter 8 gives a summary of this work and raise some important future work.

The Haskell implementation for Elm AST and JavaScript AST are listed in the appendixes.

Chapter 2

Motivation

2.1 Existing Problems

As the Internet of Things is growing fast in the industry nowadays and become a commonplace in our lives, the technology used to support IoT must be chosen reasonably and scientifically. There are several programming languages can be used in the IoT programming, for example, assembly, C, C++, Java, JavaScript and Python. Even though they are popular in the industry, none of them has a considerate architecture for IoT systems. Some of them do not have any architecture at all like assembly and C. Some of them are objected oriented programming(OOP) languages like C++ and Java. Some of them are partially functional like JavaScript. Some of them do not have a type system like JavaScript and python. However, in IoT world, OOP is too complex to be managed due to the limitation both in memory capacity and in processing power. Partially functional is not helping because few programmers are actually taking advantages of it. Lack of type system even raises the risk on getting runtime errors.

In the meanwhile, the large scale of data produced by IoT devices leads to great complexity during designing systems. Due to manufactures tend to release new products to the market as faster as possible, they often ignore these complexity concerns. This phenomenon leads to the fact that some of those products contain severe security holes and are not scalable.

It is urgent to have a considerate programming language with a well designed architecture which is suitable for constructing an IoT system, which may solve not only the safety problem but also the high complexity problem.

2.2 Existing Solutions

Facing the existing problems, there are two kinds of approaches can be considered as a solution, building good architectures and making use of functional programming languages. In this thesis, a new solution is proposed combining these two approaches. Let us have a look at them one by one.

2.2.1 Build Architectures

In IoT world, the distinction between protocol architectures and system architectures is not very clear. Often the protocol and the system architectures are co-designed. Here we take two basic architectures as examples:

Server-Client

Most architectures proposed for the IoT obey the "Server-Client" pattern. They have a server side. The server connects to all the interconnected components, composes the services, and acts as a single point of service for users.

In the server side, there are typically three layers. The first layer is a database that stores information of all the devices, their attributes and their relationships. The second layer logically controls the interaction between the server and devices, queries their status from the database, and uses the query result to effect a service. The topmost layer is the application layer, which provides services to the users.

In the client side, there have two layers. The first is the object layer, which allows a device to connect to other devices, talk to each other, and exchange information. The object layer passes information to the social layer. The social layer manages the execution of users' applications, executes queries, and interacts with server on the application layer.

Peer-to-Peer

In the IoT era, connected devices will spread sensitive personal data to centralised companies, which represents a serious risk for user's privacy. With the purpose of overcoming this problem, constructing decentralised private-by-design IoT systems could be a solution. The basic idea is that data produced by personal IoT devices are safely stored in a distributed system whose design guarantees privacy. To achieve this goal, Peer-to-Peer architecture need to be considered.

Peer-to-Peer network (P2P)[19] is a topology and has a permanent link between two end-points. One example of P2P would be the connection in the game, "paper cup-and-string telephones", where two nodes have a dedicated channel for communication. Using switching technologies, P2P can be set up dynamically. P2P technology let nearby users directly exchange information with one another to improve the data exchange throughout.

2.2.2 Use Functional Programming

Internet of Things appliances, such like light, thermostats or other kinds of sensors, are sensitive to software errors. Software bugs might lead to security problems, which are not acceptable. The increasing complexity of IoT makes it even harder to avoid potential bugs of the system. Functional programming is well-suited to solve this concern[22]. The features of functional programming, preference for immutability, function composition, avoiding side-effects, less code, etc. may help to avoid many of the pitfalls in the IOT world for following considers: (a) Using immutable data helps solve the concurrency issue as locks can be avoided; (b) Real-time communication is also better supported by functional programming; (c) It is of great use to program with side-effect free functions when developing IoT applications as it makes scaling easier while making the code easier to reason about; (d) With functional programming, there is less code to write, which leads to less bugs and a better programs.

2.3 Proposal Solution

As mentioned previously, having a good architecture and taking advantages of functional programming languages both help on improving the performance of IoT systems. The solution proposed in this thesis is to combine these two approaches together, using a purely functional programming language with a suitable architecture for IoT programming. Here is a programming language meets the requirements called Elm. Elm is originally designed for building high performance web applications with a "Model-View-Update" architecture. It is chosen because of its high level abstraction, and feasible for mapping into real devices. This proposal is motivated by the urgency to construct a scalable, safe IoT system.

Chapter 3

Related Work

3.1 Elm-Related Architectures

3.1.1 MVC

The Model-View-Controller(MVC)[30] design pattern is a common way to structure systems which need to react to events. It divides an application into three interconnected parts: Model, View and Controller. The MVC pattern decoupled these main components allowing for efficient code reuse and parallel development. This architecture has become popular in not only designing web applications but also mobile apps. MVC is usually implemented in an object-oriented style. Java, C#, Ruby, PHP and other programming languages are current popularly being used in web application development.

The model manages the data of the application and receives user input from the controller. The view is a presentation of the model in a particular format which could be customised by programmers. The controller is used to respond to the user input and perform interactions on the data model objects. The controller receives the input, validates it and then passes the input to the model[14].

However, the MVC architecture increases the complexity of the structure and implementation of a system, especially for some simple interfaces. Strictly following MVC, separating models and views from controllers, will increase the complexity[24] of the structure and reduce the performance of the system. For example, if we do not use this hierarchical structure, many operations can directly visit the database to get the corresponding data, but now they must be completed through the middle layer.

3.1.2 FRP

"This is the Hollywood principle in action: do not call us, we will call you.
And it is great for loosely coupling code, allowing you to encapsulate your components."

Functional Reactive Programming(FRP)[31] design pattern was proposed in 1997. It is a programming paradigm for reactive programming, asynchronous data flow programming, using functional programming. There are many interpretations of it based on different scenarios[21]. The main differences between FRP and MVC is when using the MVC design pattern, the Model and the View never directly communicate while FRP allows the Model and the View have a direct connection.

As it discussed in the last section, MVC is not that practical in more and more occasions due to the redundant life circle. For example, in a web application, every single change of the Model will push the application to render the UI view again. However, most times the UI is not affected by the change, which means the UI may care about several specific changes of the Model, but not all of them. That is why we need "Reactive programming" here. Back to the example, if we implement the web application using reactive programming architecture, the UI will subscribe some changes of the Model, only when those specific changes happen, the View would be rendered again. This highly increases the efficiency of applications comparing to MVC architecture.

Actually Elm is a "Event-Driven" concurrent FRP language focused on easily creating responsive GUIs[17]. Traditional FRP used sequential updates, so only one event could be processed at a time. In Concurrent FRP, many updates can be processed at the same time. In Elm, synchronisation is enabled by default, strictly maintaining the global order of events. But by its very nature, synchronisation incurs a delay. It requires faster results to wait for slower results before moving on, but in certain cases, it is acceptable to ignore the global order of events.

3.2 Build new Programming Languages for IoT

There are lots of options on choosing a programming language for an IoT system, and as mentioned before, the top four languages for developing IoT systems are Java, C, JavaScript, Python. However none of them is designed specially for IoT programming. Due to all kinds of problems caused by these popular programming languages in IoT systems, it is indispensable for programmers to design a language especially for IoT programming.

In the meanwhile, even though there exists open source frameworks designed for IoT which can be used by hobbyist and professionals for the tasks on their hand, many of those tools require the programmers to know programming languages such as C, Python and JavaScript. Those who do not know much about programming but would like to tinker with IoT may find it challenging to work with those tools.

There is a ongoing work[13] which focus on 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. And another example, existing technologies that aim at making embedded devices easier to be programmed, e.g. Arduino, work well for prototyping. However, limitations are existed when trying to scale these prototypes towards real-world deployment. Also the level of controlling over timing, memory, and behaviour that such existing technologies provide, often do not meet the requirements of commercial IoT deployments. Eclipse Mita[6] is a new programming language for the embedded IoT. It aims to close the gap between cloud and embedded development, brings these two communities closer.

3.3 Projects of Raspberry Pi

Since the first Raspberry Pi was released in 2012, It has captured the imaginations of enthusiasts and hobbyists. There are tons of outstanding Raspberry Pi projects we can find in this community: $https://www.raspberrypi.org/community/$. Here are two examples of them:

A Weather Station

This project uses a Raspberry Pi and a Sense HAT to measure the temperature, barometric pressure, and humidity. It then uploads all that data to Weather Underground. On the display screen, users can see how the temperature has changed since the last reading.

A Home Surveillance System

Generally speaking, home surveillance systems are expensive. Some people build their own small-scale system using a Raspberry Pi. For example, there is a system uses the Raspberry Pi camera module and a few other IoT devices. After setting up, this system can detect motion, broadcast a live stream and can provide other services. It is a surprisingly powerful system and it's cheap enough that everyone can try to set it up.

3.4 Security for IoT

Along with the enormous benefits, the IoT also brings up plenty of risks. In a world which is already vulnerable to the threat of cyber attacks and data breaches[15], more connected devices means billions of new entry points for hackers. And in many cases, these devices are a much softer target than normal traditional computers. For example, IoT devices know all about you and many smart devices are constantly collecting data on your movements and habits as you spending your daily life. This is useful in one perspective because the more the technology knows about you, the better it can respond to and work around your needs. However, it could be used by other businesses, like selling your private information. It could get into the hands of criminals, allowing them to steal your identity, or target your property or belongings based on what they know about you. To avoid becoming a victim, or inadvertently helping to cause an attack elsewhere, IoT security should be taken as seriously as with any other computer or device. Currently, there are several technologies may help

to prevent those security issues. In the meanwhile, to achieve those technologies is also a challenge on preventing IoT systems from attacking:

1. **IoT network security.** Unlike traditional networks, it is much more challenging to secure an IoT network. The reason is that there are wide range of communication protocols, standards and devices involved, which makes things more complex.

2. **API security.** Securing API is also critical to ensure that the data transmitted through end-point to back end system is only done by authorised persons. On the one hand, it helps users to make sure that only authorised devices, developers and applications can access to the APIs. On the other hand, it aids in detecting threats and attacks upon these APIs.

3. **Security analytics.** From collecting to aggregating data, from monitoring to normalising data from IoT devices, we need a monitoring that provides us options for reporting as well. Security analytics act as a brilliant way to alert organisations about any malicious activities that might be taking place in the background.

Chapter 4

Preliminaries

This chapter summarises some of the relevant technicalities used in this thesis. In particular, it discusses the grammar and theoretical foundation of programming languages, introduces the language Elm, talks about Raspberry Pi single-board and several common IoT devices. In the end of this chapter, it brings up the main research problems of this thesis.

4.1 Programming Language

4.1.1 Abstract Syntax

Programming languages express a set of instructions in a form which is comprehensible to both human beings and machines. The syntax of a language summaries all the possible kinds of expressions, declarations and commands might be combined to make up a program. We know that in human language, no matter what language is, there will be a "subject", "verb", "object", "punctuation" to describe a real world event. In computer programming language, no matter what language is, "type", "operator", "flow statement", "function", "object" and other concepts may be needed to express the sequence of 0 and 1 in the computer memory, and the operation and logic behind it.

In computer science, the abstract syntax tree (abbreviated AST), or the syntax tree, is a tree representation of the abstract syntax structure of the source code, which refers specifically to the source code of the programming language. It is an order tree whose leaves are variables and whose interior are operators whose arguments are its children. Each node in the tree represents a structure of this programming language. The reason why syntax tree

is “abstract” is that the grammar here does not represent every detail that appears in real grammar. For example, nested brackets are implied in the structure of the tree and do not appear in the form of nodes. To show the grammar details of one programming language, we use concrete syntax tree (CST). AST and CST are two important ways to present the syntax of one language.

An example of abstract syntax tree

Here is an expression: 1 + 3 * (4 - 1) + 2. Listing 4.1 is the data type for the AST of this expression implemented by Haskell. The graphical representation of the abstract syntax tree of it is shown in Figure 4.1.

```
1 data Expr = Number Integer
2           | Add Expr Expr
3           | Minus Expr Expr
4           | Times Expr Expr
5           | Divide Expr Expr
6           | Parens Expr
7             deriving (Eq,Show)
```

Listing 4.1: AST of a simple expression

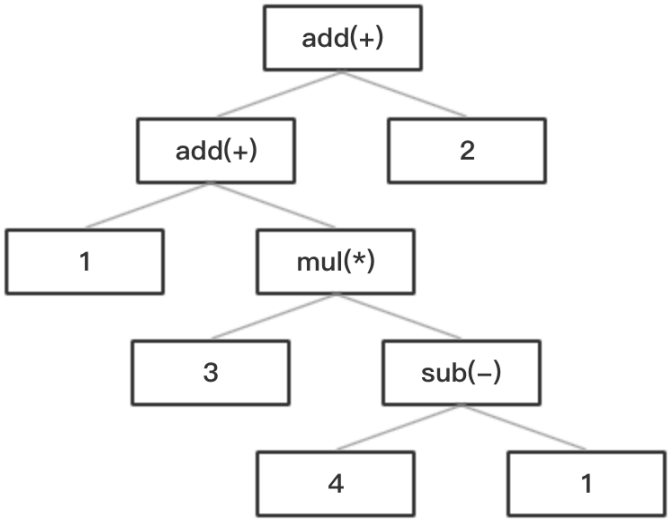

Figure 4.1: A graphical representation of AST

Abstract syntax tree is very useful and widely used in many fields such as browsers, intelligent editors, compilers. There are many tools used to simplify the programming process and help developers to program or debug in a easier way. In fact, the principle of these tools is to compile the code into an abstract syntax tree first, which defines the structure of the code. Then by manipulating the tree, we can accurately locate statements, assignments, operations and so on. In the end, those tools can implement the analysis, optimisation, modification and other operations of the code.

4.1.2 Inductive Definitions

Inductive definitions play a central role in the study of programming languages. They specify the following 4 aspects of a language: concrete syntax, abstract syntax, static semantics, dynamic semantics. An inductive definition in computer science is used to define the elements in a set in terms of other elements in the set. Often a set is described in the following way. Some clauses stipulate that certain basic elements are to be in the set; then other clauses are given to stipulate further elements of the set in terms of elements already included. Implicitly, only elements produced in these stipulated ways are to be included in the set. Sets described in this way are called inductively defined. So an inductive definition consists of:

1. One or more judgments, i.e., assertions.
2. A set of rules for deriving these judgments.

Inference Rule Notation

Inference rules are normally written as:

$$\frac{J_1 \quad ... \quad J_n}{J}$$

where J and $J_1, ..., J_n$ are judgements.

Examples

The set Num is defined by following rules:

1. Zero is a numeral.
2. If n is a numeral, then Succ(n) is also a numeral.

This inductive definition can be represented by notations as following:

$$\frac{}{zero\ numeral} \qquad \frac{zero\ numeral}{Succ(n)\ numeral}$$

Similarly, the set of binary trees defined by following rules:

1. The empty tree, Empty is a binary tree.
2. If t_l and t_r are binary trees, then Node(t_l, t_r) is a barary tree.

This inductive definition can be represented by notations as following:

$$\frac{}{Empty \in Tree} \qquad \frac{t_l \in Tree \quad t_r \in Tree}{Node(t_l,t_r) \in Tree}$$

4.1.3 Denotational semantics

Denotational semantics, initially known as mathematical semantics, is an approach of formalising the meanings of programming languages by constructing mathematical objects, called denotations, which describe the meanings of expressions of a certain language. Other approaches to providing formal semantics of programming languages include axiomatic semantics and operational semantics.

Denotations of data types

Many programming languages allow users to define recursive data types. The type of lists of numbers can be specified by:

```
1 datatype list = Cons of nat * list | Empty
```

4.1.4 Statics and Dynamics

Most programming languages can be divided into two phases, static and dynamic. A static type language refers to a language that the type of data can be determined when it is compiled. Most static type languages require that the data type must be declared before the variable is used, and some modern languages with type derivation may be able to partially mitigate this requirement. Dynamic type languages are languages that the type of data will be determined while it is executed. Variables do not need type declarations before they are used. The type of variables is usually the type of value assigned.

For the distinction between dynamic language and static language, there is a popular saying: "Static typing when possible, dynamic typing when needed."

Besides static and dynamic, there is another pair of terms called "strongly typed" and "weakly typed". In strongly typed language, the type of variables cannot be transformed once determined. Conversely, in weakly typed language, the type of a variable is determined by its application context. For example, weakly typed languages allow strings and integers to do plus operation directly.

Many people state that static strongly typed languages are suitable for developing complex, large-scale systems while dynamic weakly typed languages are not suitable for developing too complex or large projects. However, there are lots of big projects developed in Google are implemented by Python which is a weakly typed language. Actually dynamic languages give programmers more freedom and save their energy to concentrate more on implementing the crucial logical computations of the program. The greatest advantage of static typed languages is that it provides static type security, and the compiler can check whether the name of every function call is correct or not, and whether type of parameters are correct or not. Such a system allows many errors to be detected and located at compile time which can save the unit testing time.

4.1.5 Type Safety

If a well formed program behaves well when it is executed, it can be called safe. Most strongly typed programming languages are safe. Informally, this means that certain kinds of mismatches cannot arise during execution. For example, it will never happened in a safe languages that a number and a string to be added together or two numbers are concatenated, which is not meaningful.

Type safety of the language E is stated precisely as follows[23]:

1. If e : τ and $e \mapsto e'$, then $e' : \tau$
2. If e : τ, then either e val, or there exists e' such that e $\mapsto$ e'

The first rule, called preservation, saying that the type should be preserved with the execution process. The second rule called progress, ensures that well-typed expressions are either a value or can be further executed. The type safety is the conjunction of this two rules.

4.1.6 Elm

Elm is a purely functional domain-specific programming language. Elm was published by Evan Czaplicki as his thesis in 2012, this language was initially designed two years before that, around 2010. The first release of Elm came along with several examples and an online editor[2], which made it easy and fast for new learners to try it out in a web browser. We mainly use Elm for creating websites and web apps. Elm's compiler compilers it down into optimised JavaScript. It solves tons of problems which Web programming are facing in day-to-day work flow. Some programmers even think that Elm could be the future of front end development. There are several important features of Elm.

Architecture

The concepts around the Elm architecture are starting to be more and more used. It essentially boils down to three parts: Model, View, and Update. The entire application can be viewed as one loop that runs in perpetuity. It takes an initial model, presents it to the users as a view in a certain way, lets them issue messages from the view, updates the model based on those messages, and presents the updated model back to the users with a new view again. The work flow of Elm is specified as Figure 4.2, and the interactions between the Elm runtime and various components in an application is shown in Figure 4.3:

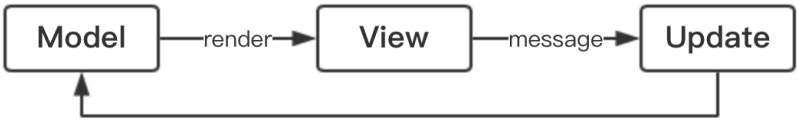

Figure 4.2: Work flow of Elm architecture

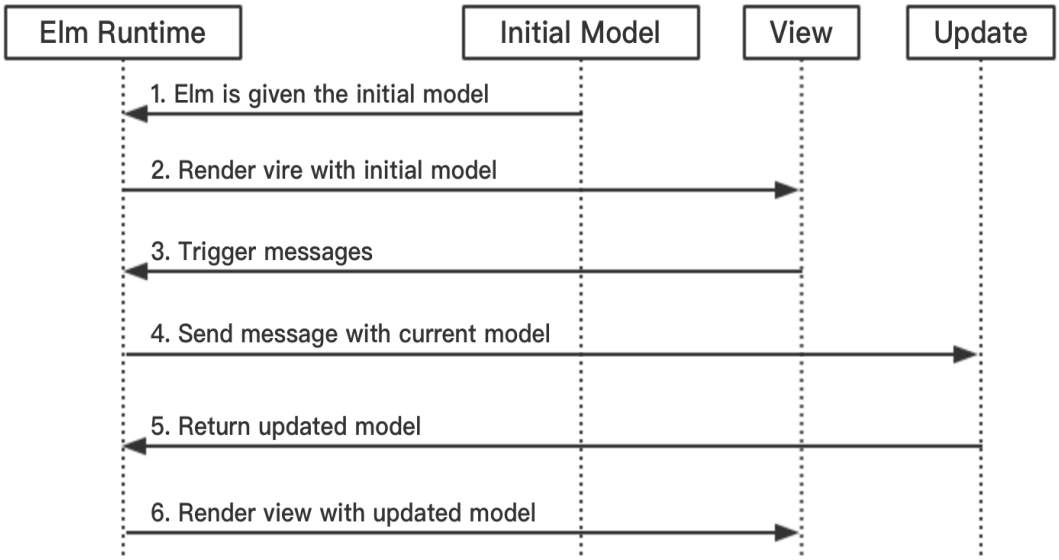

Figure 4.3: Interactions during Elm runtime

1. **Model:** a model represents the state of the application. It does not necessarily have to be complicated, all depends on how complex the application is and how many different

things it needs to track.

2. **Update:** a way to update the application's state and do some certain operations based on different messages.

3. **View:** a way to view the state as a html page. It takes a model as input and outputs HTML and CSS code.

This pattern is so reliable that programmers can always start with this following skeleton shown in Listing 4.2 and fill in details with increasingly interesting logic for their own particular aim.

```
1  import Html exposing (..)
2
3  -- MODEL
4  type alias Model = { ... }
5
6  -- UPDATE
7  type Msg = Reset | ...
8
9  update : Msg -> Model -> Model
10 update msg model =
11   case msg of
12     Reset -> ...
13     ...
14
15 -- VIEW
16 view : Model -> Html Msg
17 view model = ...
```

Listing 4.2: Skeleton of Elm architecture

That is the essence of the Elm Architecture. One of the other benefit we can get from this architecture is that it is great for modularity, code reuse, and testing.

Commands and Subscriptions

Since Elm is a pure functional programming language, it does not have side effect. Most scenarios in which an Elm app needs to interact with the outside world tend to fall into two categories[1]:

1. Tell the Elm runtime to do something. Here are some examples:

    (a) Send and receive data from a remote HTTP server.
    (b) Save data to a local storage.
    (c) Generate random numbers.
    (d) Request a JavaScript library to perform an operation

2. Get notified when something happens. Here are some examples:

    (a) Listen for web socket messages.
    (b) Listen for location changes.
    (c) Listen for clock ticks.
    (d) Listen for an output generated by a JavaScript library

Elm offers commands to deal with the scenarios in the first category and subscriptions to deal with the scenarios in the second category.

Compiler

Elm is a statically-typed language with type inference which means the type of a variable is known at compile time and programmers no need to specific what type each variable is. This strong Elm compiler is one of the most important reason why programs written by Elm get no run-time exceptions. Based on this compiler, there is a biggest benefit which is reliability. Firstly it almost never crash. Secondly it will give a very friendly assistance when we are

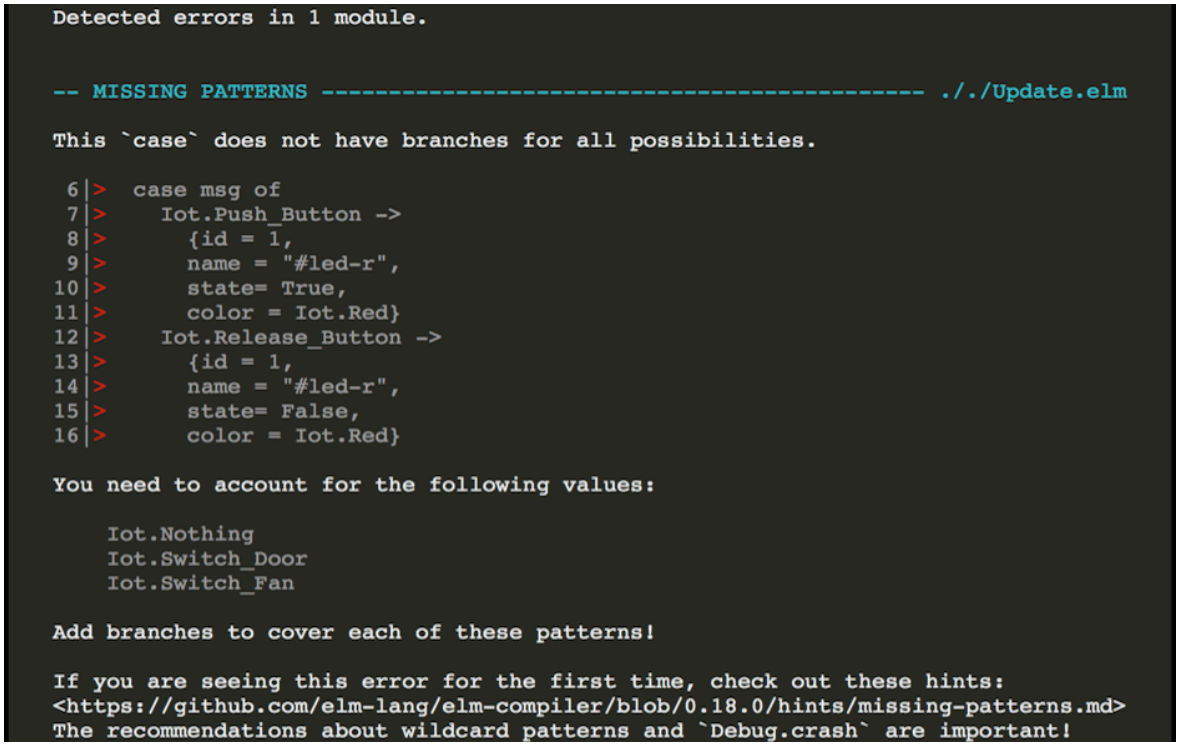

Figure 4.4: Syntax error messages of Elm

debugging. Here is an example on detecting cases which programmer may missed shown in Figure 4.4.

This is a pattern mating example, the message says that there is a situation branch which is not covered in the program. As we all know, pattern matching in imperative languages is implemented using "If-Else", in most cases, programmers are not required to fill up all the situations, it might be fine and easy to implement, but sometimes it could cause some serious consequence because of forgetting those small cases. However, in Elm, the compiler will detect all the cases would happen, and strictly requires you to fill all of them. Besides this helpful feature here, it gives us a fast and specific feedback which makes it quite clear and easy to debug. Basically, the compiler saves us from things that we don't usually think to write unit tests for.

4.1.7 JavaScript and Node.js

JavaScript is a language aimed for web development, in order to execute it, all browsers have JavaScript engines which execute the JavaScript code of web pages, like: Firefox has an engine called Spider-monkey, Safari has JavaScriptCore, and Chrome has an engine called V8.

Node.js is not a language or a special dialect of JavaScript. Node.js is simply the V8 engine bundled with some libraries to do I/O and networking, so that programmers can use JavaScript outside of the browser, to create shell scripts, backend services or execute it on hardware.

The original compiler of Elm compiles it into JavaScript and aims to be executed in web browsers. The compiler CEJ constructed in this thesis is used to compile Elm into node.js which can be executed on somewhere outside of browsers, like normal x64 system or other hardware.

4.2 The Internet of Things

4.2.1 Raspberry Pi

The Raspberry Pi is a series of small single-board computers developed in the United Kingdom to promote the teaching of basic computer science in schools. There are lots of developers and applications that are leveraging the Raspberry Pi for home automation[11]. These programmers are trying to change the Raspberry Pi into a cost-affordable solution in energy monitoring and power consumption. Because of the low cost of the Raspberry Pi, it has become a popular and economical solution to the expensive commercial alternatives.

The Raspberry Pi 3 is the third generation Raspberry Pi, about the size of a credit card. The Raspberry Pi is a powerful device fully capable of running an Linux operating system. The

device provided for this thesis came with an SD-card with the Raspbian operating system installed. The Raspberry Pi main board is shown in Figure 4.5.

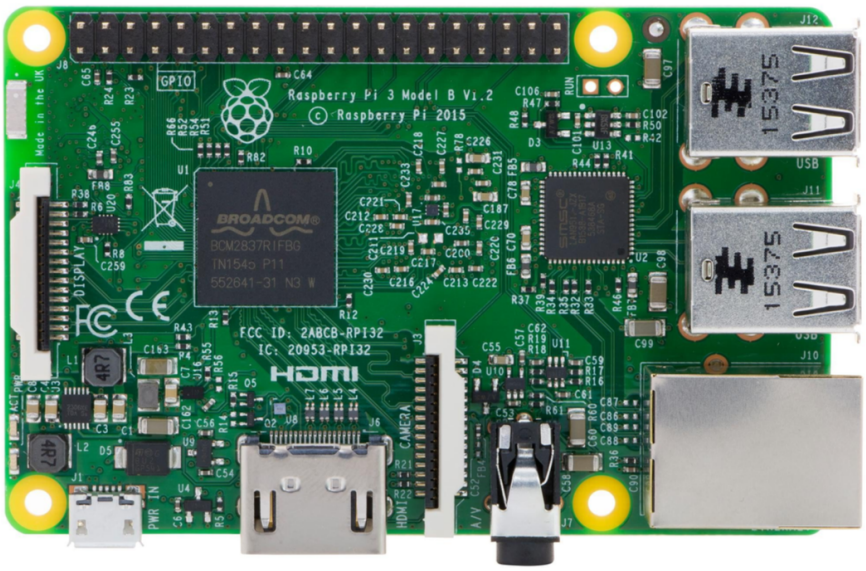

Figure 4.5: The Raspberry Pi 3 Model B

The hardware specification for the Raspberry Pi 3 is listed as follows:

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

GPIO

An important feature of the Raspberry Pi is the rows of GPIO (general-purpose input/output) pins along the top edge of the board[8]. A 40-pin GPIO header is found on most current Raspberry Pi boards. Any of the GPIO pins can be designated in softwares as an input or output pin and used for a wide range of purposes. It is possible to control GPIO pins using a number of programming languages and tools, and programmers just need to include corresponding libraries and call the function with a pin index as one of the parameters. Note that the numbering of the GPIO pins is not in numerical order As shown in Figure 4.6.

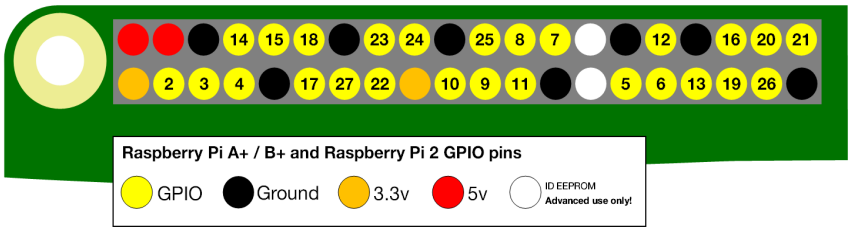

Figure 4.6: GPIO pins

I2C

Except for GPIO, The other serial interface is the Inter-Integrated-Circuit bus(I2C)[5]. It potentially allows many devices, as long as their addresses do not conflict.

I2C is a communication standard in the computing world for sensors, micro controllers, port expanders and more. Since the Raspberry Pi can talk using I2C protocol, we can connect it to a variety of I2C capable chips and modules. Besides, The I2C bus allows multiple devices to be connected to our Raspberry Pi, each with a unique address. It is very useful to be able to see which devices are connected to your Pi as a way of making sure everything is working. The i2cdetect program will probe all the addresses on a bus, and report whether there are any devices. To check it out, only need to type “i2cdetect -y 1” in the command line, As shown in Listing 4.3

```
pi@raspberrypi:~/$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: 60 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Listing 4.3: Addresses table of I2C bus

This map indicates that there is a peripheral at address 0x60.

NPM Support

Library "Raspi-sensor" is a Node.js C++ plugin, allowing to easily read data from Raspberry Pi's sensors. There are two kinds of ways to plug in a sensor in Raspberry Pi, I2C and GPIO. If we wish to use I2C sensors, the I2C driver should be loaded. If we wish to use GPIO sensors, an existing installation of wiringPi is required. For now, those sensors are supported by npm:

- DHT22(or DHT21) (GPIO) : temperature and humidity sensor
- DHT11 (GPIO) : temperature and humiditysSensor
- PIR (GPIO) : motion sensor
- BMP180 (I2C) : pressure, temperature and altitude sensor
- TLS2561 (I2C) : adafruit digital light sensor

Since the first two items are both temperature and humidity sensors, this thesis only focus on one of them, DHT11. In total we will discover 4 sensors: DHT11, PIR, BMP180, TLS2561.

4.2.2 Sensors

DHT11

DHT11 digital temperature and humidity sensor is a composite sensor contains a calibrated digital signal output of the temperature and humidity. It applies a dedicated digital modules collection technology and the temperature and humidity sensing technology which ensure that the product has high reliability and excellent long-term stability. The sensor includes a resistive sense of wet components and an NTC temperature measurement devices, and connected with a high-performance 8-bit microcontroller.

PIR

A passive infrared sensor (PIR sensor) is an electronic sensor that measures infrared light radiating from objects in its field of view. They are most often used in PIR-based motion detectors.

BMP180

BMP180 combines barometric pressure, temperature and altitude. The I2C allows easy interface with any microcontroller. On board 3.3V LDO regulator makes this board fully 5V supply compatible. In advance resolution mode, BMP180 can measure pressure range from 300 to 1100hPa which is +9000m to -500m relating to sea level with an accuracy down to 0.02hPa, around 0.17m.

TLS2561

The TSL2561 luminosity sensor is an advanced digital light sensor, ideal for use in a wide range of light situations. This sensor is precise, allowing for exact lux calculations ranges to detect light ranges from up to 0.1 - 40,000+ Lux on the fly.

4.3 Research Problems

This thesis focuses on these following three problems.

Firstly, implementing IoT applications using Elm-like code. Through the original compiler, Elm can be compiled into JavaScript and used to create web applications. In the meanwhile, node.js is used to control IoT devices in Raspberry Pi. Based on this, there is a creative idea and can be the first research problem here: if we can produce those IoT applications directly implemented using Elm. However, the JavaScript file generated by Elm compiler can not be executed in Linux OS. Apparently, directly using current Elm compiler is not doable, then this research problem becomes: if we can generate a compiler of Elm which can generate concise and valid node.js files can be executed on Raspberry Pi directly.

Secondly, semantic analyse: The description of a language can be split into two components: syntax and semantics. Syntax refers to the grammatical structure of a program and semantics refers to its meaning. There are many scenarios where Elm can throw a syntactic error such as a misplaced keyword, two operators in a row, unbalanced parentheses, etc. After a program's syntactic validity has been established, the next step is to look for semantic errors. For example, the '++' is a valid operator in Elm, but if it used to combine a string and a number, now is semantically incorrect. For a program to be valid in Elm, it has to be both syntactically and semantically correct. The second research problem could be: is there any change of the semantic of Elm after we link it to IoT applications.

Thirdly, Hardware-in-the-loop (HIL) simulation. This is a kind of technique which can be used both in the development and testing of complex real-time embedded systems. Widely usage of HIL enhances the quality of the testing by increasing the scope of the it. Ideally, an embedded system would be tested against the real plant, but most of the time the real plant itself imposes limitations in terms of the scope of the testing. In this case, HIL provides the efficient control and safe environment where test or application engineer can focus on the functionality of the controller. As we mentioned before, since Elm is designed for creating

web applications, testers can use the web interface constructed using Elm to simulate a real world model. In this way, it becomes easy to test abilities of IoT system. So, the last research problem is if we can use Elm as a simulation tool of some specific hardware applications.

We will discuss those three research problems deeper in following chapters.

Chapter 5

Design

5.1 Abstraction and Mapping

In order to abstract IoT programming and map it into Elm architecture, we can start with a node.js example of a piece of IoT program. After analysing this example, the mapping strategies of each part, namely Model, View and Update, will be displayed separately and explained in detail.

5.1.1 Start with an Example

As shown in Listing 5.1. This program mainly manipulates DHT11 sensor (temperature sensor), and a fan, accomplishes the "Smart Home" service shown in Figure 5.1. Simply speaking, in this tiny IoT system, if the temperature is above 27 degree, it will turn on the fan automatically. This example is chosen because it represents the typical IoT programs, getting data from sensors and change the state of peripherals according to some logical operations.

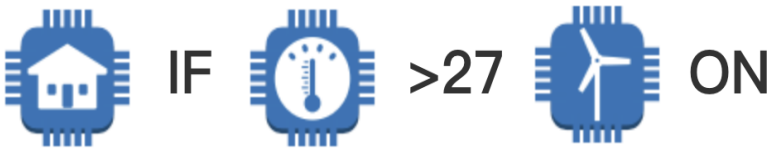

Figure 5.1: A smart home example

```
1  var sleep = require("system-sleep");
2  var dht_sensor = require("node-dht-sensor");
3  var Gpio = require("onoff").Gpio;
4  var fan = new Gpio(18, "out");
5
6  while (true) {
7      dht_sensor.read(11, 4, function(err, temperature, humidity) {
8          if (err) {
9              process.exit(1)
10         }
11
12         console.log("temp:" + temperature.toFixed(1) + "degree");
13
14         if ((temperature.toFixed(1) > 27)){
15             fan.writeSync(1);
16         }
17         else {
18             fan.writeSync(0);
19         }
20     });
21     sleep(5000); // 5 seconds
22 }
```

Listing 5.1: An IoT program of DHT11

Before diving into the details of this piece of program, it should be specified that the DHT11 sensor is connected with the pin 4 and the fan is connected with pin 18 in the GPIO of the Raspberry Pi. The first 4 lines are declaring all the variables necessarily used later on. From line 6 to 22, there is a while loop will be executed repeatedly before the termination of this program. Inside of this loop, firstly, it reads the value of current temperature and humidity in line 7. Then if there is no error invoked form the reading process, it generates a log record. Next, it comes to an if-else statement from line 14 to 19, deciding what value should be

assigned to the fan, namely 1 represents to turn it on while 0 to turn it off. In the end it sets a 5 seconds interval before enter into next loop. (Note that in line 7, the first parameter "11" in the "read" function represents the type of the DHT11 sensor)

From this node.js example shown above, what can be tried is to abstract this real example correspond to the Model-View-Update architecture first which is shown in Table 5.1. In order to make it clearer, a web app also showed there as a comparison. Basically, that web application allows logged in users to create blog posts.

Table 5.1: One instance of the mapping between a Web app and an IoT app

	Abstraction	**Web App**	**IoT App**
Model	the state of the application	isLoggedIn = False numberOfPosts = 10	temperature = 20
View	a way to show the state as a HTML page	an HTML page showing posts	write LOW to the pin connected with the fan
Update	a way to update the state of Model	add a post	update the value of current temperature

It seems that the architecture works well in at least this example. Next step, a formal prescription will be given on how to abstract any IoT program correspond to Model-View-Update architecture. What need to be stressed here is due to the diversity of programming in general, there might be different ways to abstract and map. The solution proposed in the following sections is one of the prototypes of abstracting IoT programming.

5.1.2 Essential Library

To build a complete IoT system using Elm, libraries are indispensable which defines basic types and functions will be used in the programs, such as the data type of the sensor and device and the function used to listen to the changes from the sensors.

The partial code of the library is shown in Listing 5.2, and the complete code of this essential library is listed in Appendix C.

```
type SensorInput msg = SI msg Sensor
type IoTSystem   a b = IS a b
type IOSensors   msg = IOS msg Sensor
type IODevices   msg = IOD msg Device
type IOSignal        = SetHigh | SetLow
type alias Sensor =  {s_type: String, s_address: Int}
type alias Device =  {d_pin : Int, d_lib : Maybe String,
                      d_func: Maybe String, d_dir : Maybe String}

bmp180 : Sensor
bmp180 = {s_type = "BMP180", s_address = 77}

fan_1 : Device
fan_1 = {d_pin = 16, d_lib = Just "onoff"
        ,d_func = Just "Gpio", d_dir = Just "out"}

temperature : a -> Sensor -> IOSensors a
temperature a b = IOS a b

fan : IOSignal -> Device-> IODevices IOSignal
fan a b = IOD a b

iot : List (IOSensors a) -> List (IODevices b) ->
      IoTSystem (List (IOSensors a)) (List (IODevices b))
iot a b = IS a b

onTemperatureChange : (Int->Msg) -> Sensor ->  SensorInput Msg
onTemperatureChange f s = let m = f 1 in SI m s
```

Listing 5.2: Partial essential library of constructing an IoT system using Elm

5.1.3 Model

Model is basically a set of data which expected to be tracked during the whole process of executing the application. Besides, every Model should be initialised properly because it is related to what dose View present at the first time.

In one IoT program, the data should be tracked all the time is the values returned by the sensors. In the example shown in Figure 5.1, the data we care about is the value of the temperature returned by DHT11, because only when temperature being tracked can program decide which value to be assigned to peripheral devices, like a fan.

Based on these thinkings, **the Model should be simply constructed with a record which consists of all the values returned by existing sensors in the IoT system.** One example of the Model data structure design is shown in following Listing 5.3.

```
1 type TemperatureTyp    = HIGH|MEDIUM|LOW        -- union type of temperature
2 type LightTyp          = DAY|EVENING|NIGHT      -- union type of light
3
4 -- MODEL -- returned from BMP180 sensor and TSL2561 sensor
5 model : (TemperatureTyp, LightTyp)
6 model = (HIGH, DAY)
```

Listing 5.3: A model design of IoT program

In this example, there are two sensors are included, BMP180 (temperature sensor) and TSL2561(light sensor). In the line 1 and 2, for each of them, there is an union type designed to describe the data from the sensor. As we can see, the type of the model defined in line 5 is a tuple of union types, aggregated the value of temperature and light. The initialisation of Model is shown in line 6.

You may ask that what if the concrete value of the temperature and light need to be precisely saved. In this case, setting the type of the Model as a tuple of integer can solve this problem.

5.1.4 View

Intuitively, the view of a web application simply refers to the html page which users can see directly, it could be an input text area, a button, a paragraph of text, an image and so on. Based on the current view, users may make some operations like pressing the keyboard, clicking the mouse or entering some text in input area. Reacting to those operations, the html view page may change to another view, so on and so forth. View needs to trigger Update using a concrete message. In this sense, View is also an interface to get information from outside because if we enter a word in one input area, the View will send a message along with this entered word to Update. In this perspective, View is a component which not only collects data from outside but also present the current Model in a certain way. **In the IoT world, the View must have two parts, one of them collects data from the sensors, one of them shows the current state of devices.** The view of a device could be a light, on or off, red or blue; It could be a fan, on or off, strong or weak; It also could be a buzzer, alarming or not. As shown in Figure 5.2 and Figure 5.3.

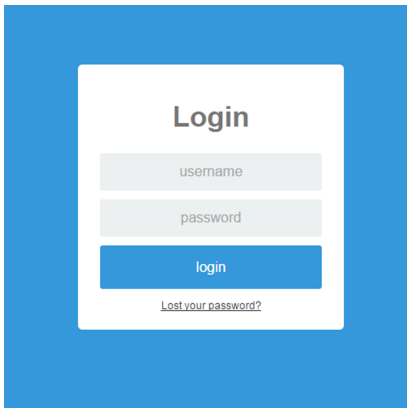

Figure 5.2: View of a Web app

Figure 5.3: View of an IoT app

One example of the View function design is shown in Listing 5.4. As we can see, the model function takes the current model as input. Inside of the View, it calls a function named “iot” which takes two listS as parameters. The first list is a collection of all the sensors while the

second one collects all the devices would be controlled in this system. There are two functions defined in line 10 and 16 used to change the state of the fan and the buzzer based on the current state of the Model.

```
1  -- VIEW
2  view model =
3      iot [ light (onLightChange Light) tsl2561
4           ,temperature (onTemperatureChange Temperature) bmp180
5          ]
6          [ fan (control_fan model) fan_1
7           ,buzzer (control_buzzer model) buzzer_1
8          ]
9
10 control_fan model =              -- A function to control the fan
11     case model of
12         (HIGH, DAY)     -> 1 -- SetHigh
13         (HIGH, EVENING) -> 1 -- SetHigh
14         otherwise       -> 0 -- SetLow
15
16 control_buzzer model =           -- A function to control the buzzer
17     case model of
18         (LOW, NIGHT)    -> 1 -- SetHigh
19         otherwise       -> 0 -- SetLow
```

Listing 5.4: A view design of IoT program

5.1.5 Update

The update is the vital part in the "Model-View-Update" abstraction, not only it connects the other two parts together, but also contains all the logical computations inside of the application. Basically, the update relies on how many messages the app can handle in total. No matter it is a web application or an IoT application, the job of Update is quite clear,

that is to get the current Model and a specific message as inputs, then return a new Model correspondingly. The complexity of Update depends on the complexity of Model and the complexity of messages. In IoT world, **the Update should be constructed along with the message data type and has a logical pattern matching used to return a new state of Model.** One example of the Update function design is shown in Listing 5.5.

```
1  -- UPDATE
2  type Msg = Temperature Int | Light Int
3
4  update msg model =
5    case msg of
6      Temperature num ->
7                 if num < 20
8                 then (LOW, second model)
9                 else if num < 30 then (MEDIUM, second model)
10                else (HIGH, second model)
11     Light num ->
12                if num > 500
13                then (first model, DAY)
14                else if num > 200 then (first model, EVENING)
15                else (first model, NIGHT)
```

Listing 5.5: An update design of IoT program

As we can see, there are two kinds of sources can invoke the Update function, the change of the temperature and the change of the light. Taking the temperature as an example, if the value of it less than 20 degree, the state of it will be changeD to LOW, and the state will be MEDIUM if the temperature greater than 20 and less than 30, otherwise it will be set to HIGH.

5.2 The CEJ

This thesis is also going to implement a compiler which compiles Elm into JavaScript (CEJ). Of course, the first step is to construct the AST of source code. The main working process of the compiler is roughly shown in Figure 5.4. Basically, the compiler takes a piece of source code as input and generates the corresponding AST of it. After a series of operations and transformations, a new AST of target code will be generated as well. In the end, the compiler will be able to generate the target code based on the new AST. The transformation process between two ASTs may contains semantical validation and translation and other necessary operations.

Figure 5.4: Translation process

5.2.1 What is CEJ?

CEJ is the compiler which compiles Elm into JavaScript. The original compiler of Elm compiles it into JavaScript which executed in web browsers. CEJ is used to compile Elm into node.js which can be executed on somewhere outside of brewers, like normal x64 system or other hardware. In this thesis, the target platform is Raspberry Pi 3.

The CEJ is implemented by Haskell. As shown in Figure 5.5. It takes an Elm file as input and outputs a corresponding JavaScript file. From the designing standpoint, there are four main stages inside of the executing process, namely lexical analysis, syntactic analysis, semantic analysis and code generating. In the real implementation process, the lexical and syntactic analysis are combined together to be a parser to generate the abstract syntax tree. The

technologies used in these stages will be explained in the following sections.

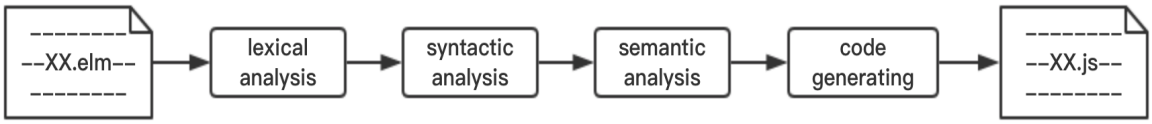

Figure 5.5: Main work flow of CEJ

5.2.2 Syntax Design

The CEJ aims to compile Elm files into JavaScript files. In this thesis, in order to simplify the the problem, a subset grammar of Elm language is supported by the CEJ. The following syntax tree defines the grammar of the supported scope of Elm. Since the main purpose of this thesis is not creating a new language, the syntax tree still follows the origin compiler of Elm, which is open-sourced in GitHub ($https://github.com/elm/compiler$).

The abstract syntax, specified by a collection of grammar rules. It provides a systematic and unambiguous account of the hierarchical and binding structure of the language and is considered as the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through all the trouble to set up fully precise grammar for it. As shown in Table 5.2. The following syntax chart summarises the abstract and concrete syntax of Elm which is used in this thesis.

Syntax Chart of Elm

Table 5.2: Syntax Chart of Elm

		AST	**CST**	**Description**
Decl	d ::=	Union$(str_1, [str_i], [(str_j, [t_j])])$	$type\ str_1\ [str_i]\ =\ [(str_j, [t_j])]$	union type
		Alias $(str_1, [str_i], t)$	$type\ alias\ str_1\ [str_i]\ =\ t$	alias of type
		Annotation (str, t)	$str\ :\ t$	type definition

	Definition $(str, [p], e)$	$str\ [p]\ =\ e$	function definition
Typ t ::=	Var(str)	str	variable
	TypeQual($str, [t_i]$)	$str\ [t_i]$	type definition
	Lambda(t_1, t_2)	$t_1\ \rightarrow\ t_2$	type composition
	Tuple($t_1, t_2, [t_i]$)	$(t_1,\ t_2,\ [t_i])$	tuple type
	Record[(str_i, t_i)]	$\{(str_i\ :\ t_i)\}$	record type
Pattern p ::=	Anything	_	anything
	Var(str)	str	variable
	Num(int)	int	numeral
	Str(str)	str	string
	Ctor($str, [p_i]$)	$str\ [p_i]$	pattern conposition
	Cons(p_1, p_2)	$p_1\ ::\ p_2$	constant pattern
	List$[p_i]$	$[p_i]$	list
	Record$[str_i]$	str_i	record
	Tuple($p_1, p_2, [p_i]$)	$(p_1,\ p_2,\ [p_i])$	tuple
	Unit	()	unit
Expr e ::=	Var(str)	str	variable
	Num(int)	int	numeral
	Str(str)	str	string
	List$[e_i]$	$[e_i]$	list
	Negate(e)	$-e$	negation
	Binops($e_1, \oplus, e_2$)	$e_1 \oplus e_2$	binary operations
	Lambda($[p_i], e$)	$\backslash[p_i] \rightarrow e$	lambda expression
	Call($str, [e_i]$)	$str\ [e_i]$	call expression

		If$([(e_i, e_j)], e)$	$if\ e_1\ then\ e_2\ else\ e$	if expression
		Let$([d], e)$	$let\ [d]\ in\ e$	let expression
		Case$(e, [(p_i, e_i)])$	$case\ e\ of\ p_i \rightarrow e_i$	case expression
		Tuple$(e_i, e_2, [e_i])$	$(e_1, e_2, [e_i])$	tuple expression
		Record$[(str_i, e_i)]$	$\{(str_i\ =\ e_i)\}$	records
		Update$(str, [(str_i, e_i)])$	$\{str \mid [str_i\ =\ e_i]\}$	update expression
		Tag$(str, [e_i])$	$str\ [e_i]$	tag expression
Def	d ::=	Define$(str, [p_i], e)$	$str\ [p_i]\ =\ e$	definition
$\oplus$	::=	$+ \mid - \mid \times \mid \div \mid == \mid != \mid > \mid < \mid \geqslant \mid \leqslant \mid \&\&$		

There are five main layers of this AST, namely declaration, type, pattern, expression and definition. The "declaration" is the topmost layer. Since some of them are overlapping, the most complex one will be explained in detail here, which is expression. For each of these key syntax, an example will be showed to help explain.

Operators

Binary operators consists of arithmetic operators and logical operators. There are four arithmetic operators: plus(+), minus(-), multiplication($\times$) and division($\div$). And seven logical operators: equal(==), not equal(!=), greater than(>), less then(<), greater or equal($\geqslant$), less or equal($\leqslant$) and and operation(&&).

Lambda Expression

A Lambda expression represents an anonymous function which is a function defined, and possibly called, without being bound to an identifier. The following example represents a

function takes two parameters as input and return the addition of them:

$$\backslash\ x\ y\ \rightarrow x + y$$

Call Expression

Any set of identifiers separated by spaces is a function call. For instance: " $a\ b\ c\ d\ e$ " is a call to function "a" with arguments "b", "c", "d", and "e".

If Expression

The "if then else" construct is common across many programming languages. Although the syntax varies from language to language, the basic structure is the same. In Elm, "else" cannot be omitted in an If expression. Here is an simple example:

$$\begin{aligned}&\text{if}\ \ x < 0\ \ \text{then}\ \ 1\\&\text{else if}\ \ x > 100\ \ \text{then}\ \ 2\\&\text{else}\ \ 3\end{aligned}$$

Let Expression

A let expression may be considered as a lambda abstraction applied to a value, may also be considered as a conjunction of expressions, within an existential quantifier which restricts the scope of the variable. Here is an example that the definition of " x = y " only valid in the scope of z.

$$\text{let}\ \ x = y\ \ \text{in}\ \ z$$

Case Expression

A case expression defines the pattern matching function. An expression is matched against the patterns. In case of run time errors, all the possible patterns should be defined.

$$
\begin{array}{lll}
\text{case } value \text{ of} & 1 \rightarrow 100 \\
& 2 \rightarrow 99 \\
& ... \\
& otherwise \rightarrow 0
\end{array}
$$

Record Expression

Consider a datatype whose purpose is to hold configuration settings. In order to distinguish each entry of the configuration, we simply give names to the fields in the datatype declaration, as follows:

$$
\begin{array}{lll}
\text{data } Server & = & \{\ username :: String \\
& & ,\ localHost :: String \\
& & \ \ ... \\
& & \}
\end{array}
$$

Update Expression

The update expression here is actually a syntax sugar which is used to update specific entries of the Model record. The reason why we need it is sometimes the Model need to record tons of items of data, but at each time, we may only wish to update one pr two of them. There is no need to define the full Model record again. One example of update expression is shown as following:

$$\{\ model \mid username = \ "Newname"\ \}$$

Union Type

In Elm, an Union Types declaration is used for many things as they are incredibly flexible. A union type has the following components:

$$\text{type } State = Pending \mid Done \mid Failed$$

In this example "State" is the type. And "Pending", "Done" and "Failed" are constructors. These are called constructors because you construct a new instance of this type using them.

5.2.3 Parser

After confirming the concrete syntax design of the Elm source file, it comes to the task of parsing a file which is a common one for programmers. Parsec is a useful parser combinator library, with which we combine small parsing functions to build more sophisticated parsers. Parsec provides some simple parsing functions, as well as functions to tie them all together. It's helpful to know where Parsec fits compared to the tools used for parsing in other languages. Parsing is sometimes divided into two stages: lexical analysis and parsing itself. In this thesis, Parsec is used to perform both lexical analysis and parsing.

5.2.4 Semantic Validation

Assume that we have completed construction of the AST, we are ready to analyze the semantics, or the actual meaning of a program, and not focus on simply its structure. Unlikely to other programming languages, due to the architecture of Elm itself, it has its own semantics.

Main function

Due to the special architecture of programming language Elm, One input file has to define the main function which has a specific way to present, as shown below:

$$\text{main} = \{\text{model} = model,\ \text{view} = view,\ \text{update}\ =\ update\ \}$$

In this declaration, "main", "model", "view" and "update" are reserved words. It does not matter which order you follow to define "model", "view" and "update", but all of them should be defined at the same time. And those italic names refer to the functions you want

to bind with each component of the architecture. Programmers can change those names to whatever they want as long as those functions will be defined as well in the input Elm file. In the CEJ, there are two kinds of semantic errors could be triggered here.

1. Incomplete declaration of "main" function.
2. Function names are not defined in the file.

Type Checking

In this thesis, the type checking is not implemented, we simply use the existing type checking system of Elm either by compiling the source code in the online editor or compiling it in the local installed Elm compiler. If there is no error message, the source code is valid on type matching.

Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. The symbol table serves the following purposes:

1. To store the names of all entities in a structured form at one place.
2. To determine the scope of a name (scope resolution).

As shown in Figure 5.6, in order to keep a symbol table, the CEJ keeps a stack to save all the symbols which are presented as forms. Each form illustrates the scope and the type of each symbol. There are two kinds of semantic errors could be triggered here.

1. The variable name is not in this scope
2. The type of the parameters are not matching.

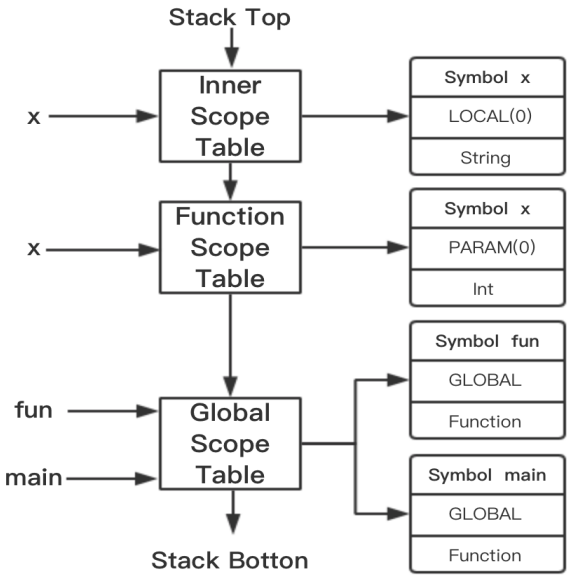

Figure 5.6: Symbol table construction

5.2.5 Code Generation

As shown in Figure 5.4, after constructed the AST of the source code of Elm, next step is to transform this Elm AST into the corresponding AST of JavaScript. A parser for JavaScript implemented using Haskell is founded in Hackage [4]. In this transforming part, I referenced the AST of their parser. As shown in Table 5.3. The following syntax chart summarises the abstract and concrete syntax of JavaScript which is used in this thesis.

Syntax Chart of JavaScipt

Table 5.3: Syntax Chart of JavaScipt

			AST	CST	Description
JSAST	p	::=	JSAstProgram[s]	[s]	program
JSState	s	::=	JSStateBlock[s_i]	s_1 s_2 ...	block statement

	JSVariable($[e_i], semi$)	$var\ e_1, e_2, ...;$	variable declaration
	JSFunction($str, [str_i], s$)	$function\ str\ (str_1, ...)\{s\}$	function
	JSIfElse(e, s_1, s_2)	$if(e)\ s_1\ else\ s_2$	if-else statement
	JSReturn($Maybee, semi$)	$return(e);$	return statement
	JSWhile(e, s)	$while(e)s$	while statement
	JSCallDot($e_1, e_2, semi$)	$e_1.e_2;$	call statement
	JSStateList$[e_i]$	$e_1\ e_2\ ...$	list of statements

$JSExpr$	e	::=	JSId *str*	*str*	identifier
			JSInt *str*	*str*	numeral
			JSBool *str*	*str*	boolean
			JSString *str*	"*str*"	string
			JSIndex (e_1, e_2)	$e_1[e_2]$	index
			JSList $[e_i]$	$[e_1,\ e_2,\ ...]$	list expr
			JSVarInitExpr(e_1, e_2)	$e_1 = e_2$	variable define
			JSRecord$[(str_i, e_i)]$	$\{(str_1 : e_2), ...\}$	record define
			JSCallExprDot(e_1, e_2)	$e_1.e_2$	call expression
			JSMemberExpr($e, [e_i]$)	$e\ (e_i)$	function call
			JSMemberNew($e, [e_i]$)	$new\ e(e_i)$	new function
			JSExprBinary(e_1, op, e_2)	$e_1\ op\ e_2$	binary operation
			JSFuncExpr($str, [str_i], s$)	$str\ [str_i]\ s$	function defination

---

$JSBinOp$ op ::= $+ \mid - \mid \times \mid \div \mid == \mid != \mid > \mid < \mid \geqslant \mid \leqslant \mid \&\&$

$JSSemi$ $semi$ ::= ; semicolon

Mappings between JavaScript and Elm

In order to transform between ASTs of different languages, the consistence of syntax and semantics should be strictly guaranteed. Fortunately, there are lots of common usages between Elm and JavaScript. The Table 5.4[3] shows side-by-side mappings between these two languages. Though this is just a subset of the syntax, it still can be observed that lot of syntaxs are very similar.

Table 5.4: Syntax Mappings between Elm and JavaScript

Elm	JavaScript
Literals	
3	3
3.1415	3.1415
“Hello world!”	“Hello world!”
True	true
[1,2,3]	[1,2,3]
Objects / Records	
{ x = 3, y = 4 }	{ x: 3, y: 4 }
point.x	point.x
{ point | x = 42 }	point.x = 42
Functions	
\x y ->x + y	function(x, y) { return x + y; }
max 3 4	Math.max(3, 4)
List.map sqrt numbers	numbers.map(Math.sqrt)
List.map .x points	points.map(function(p) { return p.x })
Control Flow	
if 3 >2 then “cat” else “dog”	3 >2 ? ‘cat’ : ‘dog’
let x = 42 in ...	var x = 42; ...
Strings	
“abc” ++ “123”	‘abc’ + ’123’
String.length “abc”	‘abc’.length
String.toUpper “abc”	‘abc’.toUpperCase()
“abc” ++ toString 123	‘abc’ + 123

For normal functions, it can be translated directly based on those mapping rules, but for the functions related to the Elm architecture, such as “model”, “view”, “update”, should be translated specially based on the special semantics.

5.3 The Simulation System

5.3.1 Problem Description

IoT is simultaneously driven by both software and hardware. They are equally important and rely on each other. Since hardware takes longer to develop, companies are forced to wait until the hardware is completed and then begin to truly test their entire IoT system. There is no doubt that this leads to a longer time for market, sub-optimal user experience, and more costly endeavours. It is true that changes to hardware take more time and money than changes to software, so the usage of simulation systems in IoT programming provides a way for companies to save tons of time, money, and energy. If we could simulate the hardware of an IoT solution while it's still being developed to test as if the hardware actually existed. Nothing would be physically connected to the software, but the tasks that the hardware performs such as measure, collect, or transfer data would be simulated and the software would not know any difference. The aims of the simulation system are:

1. To make sure the software is working as expected under normal conditions.

2. Use stress test to see how the software behaves at scale or as key values exceed thresholds.

3. Rapidly iterate before a product goes to market to ensure the highest quality solution and best user experience possible.

4. Allow IoT companies to actually put something in the hands of stakeholders or potential customers before the hardware is finished, providing critical feedback early in the process.

5.3.2 Simulation Model

This thesis uses language Elm to implement the simulation system. Essentially the simulation system is a web application, and the users can easily open it in a browser. The simulation

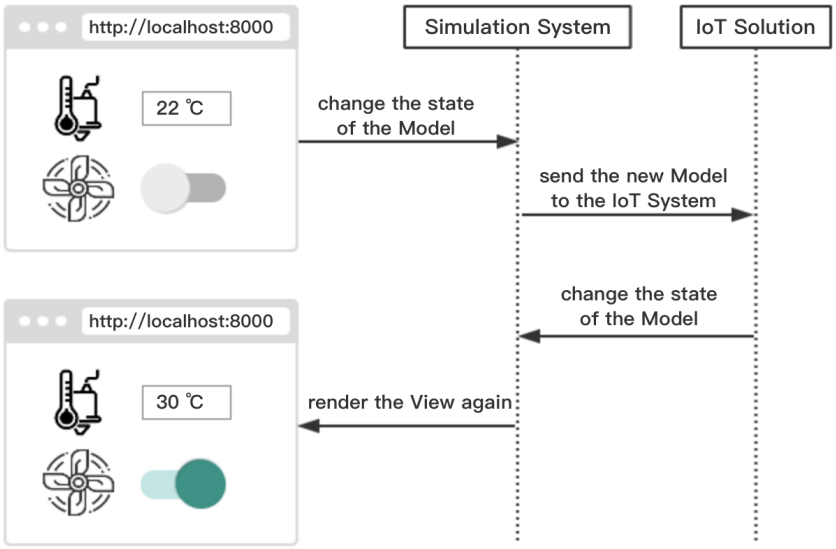

Figure 5.7: Work flow of simulation model

system consists of 2 parts, input area and output area. As the example shown in Figure 5.7, the view of the application gives users a interface to change the parameter of the sensor, and shows users the current state of the device. The web application will keep one Repository of the Model which contains the current information of the sensor and current state of the device. Once the user changed the parameter of the sensor, the simulation system will first update the state of the Model and then send it to the IoT solution through web socket. Receiving the data, the IoT solution will give a reaction based on the new state of the Model. In the end, the simulation system will render the View again after receiving the commands from IoT solution.

Chapter 6

Implementation

This chapter mainly uses some important code segments to explain how dose it implemented on each part. The full repository can be found on my GitHub page:
$https://github.com/songyahui/Elm-IoT-CEJ.git$.

6.1 Parser

```
1  declaration :: Parser Decl
2  declaration = try union_or_alias <|> define_or_annotation
3
4  declarations :: Parser [Decl]
5  declarations = spaces *> many (lexeme $ declaration)
6
7  build_AST :: SourceName -> String -> Either ParseError [Decl]
8  build_AST = runParser declarations ()
9
10 parse :: String -> String -> Either ParseError [Decl]
11 parse fileName inpStr =
12         let inpStr_no_comments = clearComments inpStr ""
13         in  build_AST fileName inpStr_no_comments
```

Listing 6.1: Implementation of the parser (1)

The haskell parser combinator library Parsec is used in this part. As shown in Listing 6.1, if one file needs to be parsed, the interface function *parse* should be called with the file's

name and the content string as input parameters. It will return us either an error message or a complete syntax tree. Before calling the *build_AST* function, all the comments should be cleared as a pre-process. Since the topmost level of Elm AST is declarations, it starts with trying to parse a possible declaration, as shown in line 2, the "*try*... < | > ..." expression provides a chance to consume a correctly matched declaration syntax rule. The combinator "∗ >" applies its first argument, throws away its result, then applies the second and returns its result. Function"*many*" is used to consume zero or more declarations.

```
1  union_or_alias  :: Parser Decl
2  union_or_alias = do
3      re <- try $ lexeme $ string "type"
4      fc <- lexeme_ret $ unionD <|> aliasD
5      return fc
6
7  union:: Parser Decl
8  union = do
9      (name, args) <- nameArgsEquals
10     ec <- sepBy unionDhelper (lexeme $ char "|")
11     return $ Union name args ec
12
13 alias:: Parser Decl
14 alias = do
15     re <- try $ lexeme $ string "alias"
16     (name, args) <- nameArgsEquals
17     ec <- type_
18     return $ Alias name args ec
```

Listing 6.2: Implementation of the parser (2)

As shown in Listing 6.2, this function *union_or_alias* is used to distinguish union type and an alias type. Since both of them start with a keyword *type*, once detected the keyword, it will keeping matching if it is a union type or a alias type in line 4. For each of them, there are some different rules needs to be matched before return a concrete declaration.

6.2 AST Transformer

The interface of AST transformation is $transformer$, as it shown in Listing 6.3, and it takes a list of Elm declaration as input, returns a JavaScript AST. For some special functions like iot_main, $model$, $view$, $update$, the transformation would be special designed due to the semantic reasons shown from line 7 to line 12. But for other declarations, they will be treated all the same, and uses function $trans\_def\_to\_JS$ to complete the transformation.

```
1  trans_def_to_JS :: Decl -> JSState
2  trans_def_to_JS (Definition c d e) =
3      JSFunction c (map trans_pattern_to_JS d) (trans_expr_to_JS_state e)
4
5  transformer :: [Decl] -> [JSState] -> JSAST
6  transformer [] temp  = JSAstProgram temp
7  transformer ((Annotation a ("Device")):(Definition c d e):xs) temp =  ...
8  transformer ((Annotation a ("Sensor")):(Definition c d e):xs) temp =  ...
9  transformer ((Definition "iot_main" a b):xs) temp =  ...
10 transformer ((Definition "model"    a b):xs) temp =  ...
11 transformer ((Definition "view"     a b):xs) temp =  ...
12 transformer ((Definition "update"   a b):xs) temp =  ...
13 transformer ((Definition c d e):xs) temp =
14         let body_expr = trans_state_to_JS (Definition c d e)
15         in  transformer xs (temp++[body_expr])
16 transformer (x:xs) temp = transformer xs temp
```

Listing 6.3: Implementation of the AST transformer (1)

For each syntax rule in Elm, a corresponding JavaScript rule should be mapped properly. Due to the difference of structures of two ASTs, some Elm expression need to be translated into JavaScript expression while some Elm expression need to be translated into JavaScript statement. As shown in Listing 6.4, function $trans\_expr\_to\_JS$ mainly translates Elm expressions into JavaScript expressions. For those terminal expressions like Str, Var, Int, it will

be translated directly into a JavaScript statement. But for those non-terminal expressions, it needs to call the function *trans_expr_to_JS* recursively.

```
1  trans_expr_to_JS :: Expr -> JSExpr
2  trans_expr_to_JS (Str p) =  JSString p
3  trans_expr_to_JS (Var p) =  JSId p
4  trans_expr_to_JS (Tag s e) =
5      let helper [] temp = temp
6          helper (x:xs) temp = helper xs (temp ++ [(trans_expr_to_JS x)])
7      in JSTagList ([(JSString s)] ++ (helper e []) )
8  trans_expr_to_JS (Record e) =
9      let helper [] temp = temp
10         helper (x:xs) temp =
11           case x of
12             (a, b) -> helper xs (temp ++ [(a,(trans_expr_to_JS b))])
13     in JSRecord (helper e [])
14
15 trans_expr_to_JS (Binops a b c) =
16     JSExprBinary (trans_expr_to_JS b) (trans_op a) (trans_expr_to_JS c)
17
18 trans_expr_to_JS (Call "second" a) =
19     JSIndex (trans_expr_to_JS (head a)) (JSInt "1")
20 trans_expr_to_JS (Call "first" a) =
21     JSIndex (trans_expr_to_JS (head a)) (JSInt "0")
22
23 trans_expr_to_JS (Tupple a b _) =
24     JSList [(trans_expr_to_JS a), (trans_expr_to_JS b)]
```

Listing 6.4: Implementation of the AST transformer (2)

6.2.1 Special Cases

For most cases, Elm and JavaScript have the same syntax, even though the writing style is not completely the same. But there are some other syntaxes existed in Elm while are not supported by JavaScript. Besides, out of the thinking that some functions related to the Elm architecture, they are not just a normal function, they have some other meanings. For those special cases, transform them directly will not be appropriate anymore. That is the reason why we need special considerations for those cases.

Tuple syntax

Elm supports the tuple expression. For example, the expression (1, "*b*") represents a tuple which consists two elements, the first one is a number 1, and the second one is a string "b". But there is no tuple expression in JavaScript. In this case, the tuple expression in Elm will be transformed into a List in JavaScript, due to the face that in JavaScript it is not necessary to keep all the elements in the List have the same type. So the expression (1, "*b*") will be transformed into [1, "*b*"]

Tag syntax

In Elm, *type Device = Light a b c* represents a constructor of type *Device*. There is a *Tag* expression written in this format which is a sequence of variables starts with a variable with an upper case as the first character. However, that is not required to declare the type in JavaScript. In this case, a type variable *Light* 1"*a*"*true* will be transformed to a List ["*Light*", 1, "*a*", *true*].

Declaration of device and sensor

In IoT programming, every sensor and every device should be declared at the beginning of the code. In Elm, the devices and the sensors will be declared using a function, and the body of the function is a record which contains all the necessary information of this device or sensor. In this case, the transformer will detect the type of each function, of the type is *Sensor* or *Device*, then this function will be transformed into a declaration statement in JavaScript.

View function

Since the view function is one of the elements of Elm architecture, it will be translated into a while loop, in which the sensor will keep fetching data from environment, and based on the changes of the data, the states of all the devices will also keep being update.

6.3 Code Generator

After last section, a JavaScript AST is generated, then the last step is to translate this AST into target code. The translation strategy is straight forward. What the generator need to do is to define the translation rules one by one for each expression ans statement. As shown in Listing 6.5, the function *generator* is the interface of the code generation, it takes a JavaScript AST as input and returns a string which is the target code.

```
1 generator :: JSAST -> String
2 generator JSAstProgram list = generator_all list ""
```

Listing 6.5: Implementation of the code generator (1)

The expression translation is partially shown here as an example in Listing 6.6. For those terminal expressions like $JSId$, $JSInt$, $JSBool$, it will be translated directly into a string.

But for those non-terminal expressions, it needs to call the function *gen_expr* recursively.

```
1  gen_expr :: JSExpr -> String
2  gen_expr (JSId p)    =  p
3  gen_expr (JSBool p)  =  p
4  gen_expr (JSString p)=  "\'" ++ p ++ "\'"
5  gen_expr (JSMemberDot a b) = (gen_expr a) ++ "." ++ (gen_expr b)
6  gen_expr (JSExprBinary a b c) =
7      let lhs = (gen_expr a)
8          rhs = (gen_expr c)
9          op = case b of
10                 Divide -> " /  "
11                 Ge     -> " >= "
12                 Gt     -> " >  "
13                 ...
14     in "(" ++ lhs ++ op ++ rhs ++  ")"
15 ...
```

Listing 6.6: Implementation of the code generator (2)

Chapter 7

Outcome from The CEJ

In this chapter, a test example will be used to show the outcome of implementing the CEJ. There are two sensors and two devices are manipulated based on the changes of the temperature and the light. The test source code of Elm is shown in the following Listing 7.1.

```
1  buzzer_1 : Device
2  buzzer_1 = {d_pin = 18, d_lib  = "onoff", d_func = "Gpio", d_dir  = "out"}
3  fan_1    : Device
4  fan_1    = {d_pin = 16, d_lib  = "onoff", d_func = "Gpio", d_dir  = "out"}
5
6  bmp180  : Sensor
7  bmp180  = {s_lib = "raspi-sensors", s_constFun = "Sensor", s_type = "BMP180"
8           , s_address = 0X77, s_desc = "Temperature_sensor"}
9  tsl2561 : Sensor
10 tsl2561 = {s_lib = "raspi-sensors", s_constFun = "Sensor", s_type = "TSL2561"
11          , s_address = 0X39, s_desc = "LIGHT_sensor"}
12
13 iot_main = { model = model, view = view, update = update }
14
15 type TemperatureTyp  = HIGH|MEDIUM|LOW    --temperature type
16 type LightTyp        = DAY|EVENING|NIGHT   --light type
17
18 model : (TemperatureTyp , LightTyp)
19 model = (HIGH, DAY)
20
21 type Msg = Temperature Int | Light Int
```

```
22
23 update msg model =
24   case msg of
25     Temperature num ->  if num < 20
26                         then (LOW, second model)
27                         else if num < 30 then (MEDIUM, second model)
28                         else (HIGH, second model)
29     Light num       ->  if num > 500
30                         then (first model, DAY)
31                         else if num > 200 then (first model, EVENING)
32                         else (first model, NIGHT)
33     otherwise      -> model
34
35 view model =
36     iot [ light (onLightChange Light) tsl2561
37          ,temperature (onTemperatureChange Temperature) bmp180
38         ]
39         [ fan (control_fan model) fan_1
40          ,buzzer (control_buzzer model) buzzer_1
41         ]
42
43 control_fan model =
44     case model of
45         (HIGH, DAY)     -> 1 --SetHigh
46         (HIGH, EVENING) -> 1 --SetHigh
47         otherwise       -> 0 --SetLow
48
49 control_buzzer model =
50     case model of
51         (LOW, NIGHT)    -> 1 --SetHigh
52         otherwise       -> 0 --SetLow
```

Listing 7.1: TEST- Elm source code

After generating and translating the AST of this source code, the target code is generated successfully, which is shown in Listing 7.2.

```
var buzzer_1_lib = require("onoff").Gpio;
var buzzer_1 = new buzzer_1_lib(18, "out");

var fan_1_lib = require("onoff").Gpio;
var fan_1 = new fan_1_lib(16, "out");

var bmp180_lib = require("raspi-sensors");
var bmp180 = new bmp180_lib.Sensor({
    type: "BMP180", address: 119
}, 'Temperature_sensor');

var tsl2561_lib = require("raspi-sensors");
var tsl2561 = new tsl2561_lib.Sensor({
    type: "TSL2561", address: 57
}, "LIGHT_sensor");

var model = ["HIGH", "DAY"];
function update(msg, model) {
    if (msg[0] == Temperature) {
        if (num < 20) {
            return ["LOW", model[1]];
        }
        else if (num < 30) {
            return ["MEDIUM", model[1]];
        }
        else return ["HIGH", model[1]];
    }
    else if (msg[0] == Light) {
        if (num > 500) {
            return [model[0], "DAY"];
        }
```

```
32          else if (num > 200) {
33              return [model[0], "EVENING"];
34          }
35          else return [model[0], "NIGHT"];
36      }
37      else return model;
38  }
39  while (true) {
40      tsl2561.fetch(function (err, num) {
41          update(["Light", num], model);
42      });
43      bmp180.fetch(function (err, num) {
44          update(["Temperature", num], model);
45      });
46      fan_1.writeSync(control_fan(model));
47      buzzer_1.writeSync(control_buzzer(model));
48  }
49  function control_fan(model) {
50      if ((model[0] == "HIGH") && (model[1] == "DAY")) {
51          return 1;
52      }
53      else if ((model[0] == "HIGH") && (model[1] == "EVENING")) {
54          return 1;
55      }
56      else return 0;
57  }
58  function control_buzzer(model) {
59      if ((model[0] == "LOW") && (model[1] == "NIGHT")) {
60          return 1;
61      }
62      else return 0;
63  }
```

Listing 7.2: TEST- JavaScript Target code

Chapter 8

Conclusion

8.1 Research Process

This thesis started from around October 2017. From the beginning of 2018, we narrowed down the research direction into Elm programming language and Raspberry Pi. Before that, the research work was in a wide range. It took me around 2 months to get familiar with Haskell and functional programming in general. In the meanwhile, I was also learning how to develop a hardware application as well, and tried several different hardware programming languages such as Verilog and VHDL. When I first learnt about Elm, I was attracted to it because it can be nicely compiled into JavaScript. As we all know, JavaScript is a popular and robust programming language which can be used in many occasions. Next, I had to find some hardware platforms which can be supported by JavaScript. In the end, I chose Raspberry Pi. I tried several examples of Elm applications and IoT applications build in Raspberry Pi separately and independently. These experience allow me to create a new compiler which is specifically designed to convert Elm into Node.js. I did not have any hardware programming experience before. This may have contributed to the difficulty on pushing the project forward.

8.2 Evaluation

The solution provided in this thesis is mainly trying to tackle the high complexity problem of IoT programming in real life. The impact of this work can be summarised into following

five main points:

1. The most direct result of this work is programmers can use Elm-like code and its well-constructed architecture to manipulate IoT devices directly.

2. This work greatly changes the common way on IoT programming as well as changes the design mode of IoT programs.

3. The thesis also tends to combine IoT with purely functional programming languages which is not only improving the security of IoT systems[27] but also widening the usage of functional programming languages in real life.

4. The process on creating CEJ also gives an example in practice on creating a new programming language related to IoT programming.

5. In the end, to study more features of programming language Elm, and to expand the usage of it, this thesis also constructed a simulation system using Elm aims to provide a solution on testing IoT systems which increases the diversity of creating a simulation system.

8.3 Limitation

There are two current limitations of CEJ compiler.

Firstly, diversity. This thesis focuses on translating Elm to JavaScript, and because of the limitation of the sensors supported by npm, the sensor's diversity scope of CEJ is also limited.

Secondly, connectivity. Current test cases are mainly dealing with sensors and devices connected in one single platform. This is not enough. As we all know, smart homes control different devices which are connected in different places using network. It is important to extend the CEJ to supporting network programming as well.

8.4 Future Work

Firstly, based on the limitation pointed out in last section, one further study should be how to built a more robust system with high extensibility and connectedness.

Secondly, type checking. In the static analysis, the type checking should be an important component. Type checking has already implemented in the original compiler of Elm. We should systematically execute it and apply it to CEJ.

Thirdly more advanced technologies. It is of great importance to combine technologies such as data mining and AI to improve the performance of IoT applications.

Bibliography

[1] Beginning Elm. http://elmprogramming.com/subscriptions.html.

[2] Elm-Lang.Org. http://elm-lang.org/try.

[3] Elm Syntax vs JS. http://elm-lang.org/docs/from-javascript.

[4] Hackage. http://hackage.haskell.org/package/language-javascript.

[5] I2C: Inter-Integrated Circuit Communication. http://radiostud.io/howto-i2c-communication-rpi/.

[6] Introducing Eclipse Mita - A Language for Embedded IoT. https://www.eclipse.org/community/eclipse_newsletter/2018/march/mita.php.

[7] Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated. https://spectrum.ieee.org/tech-talk/telecom/internet/.

[8] Raspberry Pi + e-Paper module + Node.js. https://dbushell.com/2017/09/20/raspberry-pi-epaper/.

[9] The Eclipse Foundation. https://www.eclipse.org.

[10] The Growth of the Internet of Things. http://coolinfographics.com/blog/2016/5/11/the-growth-of-the-internet-of-things.html.

[11] P. Uma Maheshwari Dr. N. Sathish Kumar N. Geraldine Sherley A. Santha Priya, S. Saranya. Automatic detection and notification of potholes and humps on road and to measure pressure of the tire of the vehicle using raspberry pi. *raspberry pi IEEE PAPER 2016*, 2, 2016.

[12] M. Aazam and E.-N. Huh. Fog computing and smart gateway based communication for cloud of things. *Proceedings of the 2nd IEEE International Conference on Future Internet of Things and Cloud*, pages 464—-470.

[13] Magnus Asrud. A programming language for the internet of things. 2017.

[14] Frank Buschmann. Pattern-oriented software architecture. 1996.

[15] ChakibBekara. Security issues and challenges for the iot-based smart grid. *Procedia Computer Science*, 34:532–537, 2014.

[16] Wei Ping Shih Ting En Lin Rui-Jun Yeh Iru Wang Che Min Chung, Cai Cing Chen. Automated machine learning for Internet of Things. *IEEE International Conference on Consumer Electronics, Taiwan*, 2017.

[17] Evan Czaplicki. Elm: Concurrent frp for functional guis. *PHD thesis, Harvard*, Apr 2012.

[18] M.A. Razzaque et al. Middleware for internet of things: A survey. *IEEE Internet of Things, New York*, 3(1):70, 2016.

[19] Analoui M. Pathan M.; Buyya Garmehi, M. An economic replica placement mechanism for streaming content distribution. *Hybrid CDN-P2P networks, Comput, Commun*, pages 60—-70, 2014.

[20] Michael W. Engle Bobbi J. Young Jim Conallen-Kelli A. Houston Grady Booch, Robert A. Maksimchuk. *Object-Oriented Analysis and Design with Applications*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[21] J Peterson H Nilsson, A Courtney. Functional reactive programming, continued. *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51—-64, 2002.

[22] Till Haenisch. A case study on using functional programming for internet of things applications. *Athens Journal of Technology and Engineering*, March 2016.

[23] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[24] Morrison M Heidke N, Morrison J. Assessing the effectiveness of the model view controller architecture for creating web applications. *Midwest instruction and computing symposium, Rapid City, SD*, 2008.

[25] John Hughes. Why functional programming matters. *"Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley*, pages 17–42, 1990.

[26] J. Hund J. Heuer and O. Pfaff. Toward the web of things: Applying web technologies to the physical world. *Computer*, 48(5):34—-42, 2015.

[27] J. Tevis Jay-Evan. Secure programming using a functional paradigm. *Proceedings of the Illinois State Academy of Science Conference*, 2006.

[28] Sameh Sorour Mehdi Mohammadi, Ala Al-Fuqaha and Mohsen Guizani. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Communications Surveys and Tutorials*, 2017.

[29] Partha Pratim Ray. A survey on visual programming languages in internet of things. *Scientific Programming*, pages 1231430:1—1231430:6, 2017.

[30] T. Reenskaug. The original mvc reports. *Oslo: T. Reenskaug*, 1979.

[31] Hudak P. Wan, Z. Functional reactive programming from first principles. *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, British Columbia, Canada*, 2000.

Appendix A

Haskell Implementation of the Elm AST

```
1  -- EXPRESSIONS
2  data Expr
3    = Str String
4    | Int Int
5    | Float Double
6    | Boolean Bool
7    | Var String
8    | List [Expr]
9    | Negate Expr
10   | Binops String Expr Expr
11   | Lambda [Pattern] Expr
12   | Call String [Expr]
13   | If [(Expr, Expr)] Expr
14   | Let [Def] Expr
15   | Case Expr [(Pattern, Expr)]
16   | Tupple Expr Expr [Expr]
17   | Record [(String, Expr)]
18   | Update String [(String, Expr)]
19   | Tag String [Expr]
20   | Block [Expr]
21   deriving (Show, Eq)
22
23 -- DEFINITIONS
24 data Def = Define (String) [Pattern] Expr
25   deriving (Show, Eq)
26
```

```
27 -- PATTERN
28 data Pattern
29   = PAnything
30   | PStr String
31   | PInt Int
32   | PVar String
33   | PCtor String [Pattern]
34   | PCons Pattern Pattern
35   | PList [Pattern]
36   | PRecord [String]
37   | PTuple Pattern Pattern [Pattern]
38   | PUnit
39   deriving (Show, Eq)
40
41 -- TYPE
42 data Type
43   = TLambda Type Type
44   | TVar String
45   | TTuple Type Type [Type]
46   | TRecord [(String, Type)]
47   | TTypeQual String [Type]
48   deriving (Show, Eq)
49
50 -- DECLARATIONS
51 data Decl
52   = Union String [String] [(String, [Type])]
53   | Alias String [String] Type
54   | Annotation String Type
55   | Definition String [Pattern] Expr
56   deriving (Show, Eq)
```

Appendix B

Haskell Implementation of the JavaScript AST

```
1  module Generator.JSAST where
2  import Text.ParserCombinators.Parsec
3  import Control.Applicative ((<*), (*>), (<$>), (<*>))
4
5  data JSAST = JSAstProgram [JSState]
6      deriving (Eq, Show)
7
8  data JSState
9      = JSStateBlock [JSState]      -- ^{stmts};
10     | JSVariable [JSExpr] JSSemi  -- var exprs;
11     | JSFunction String [String] JSState  -- ^fn,name,(parameters)block
12     | JSIfElse JSExpr JSState JSState -- ^if,(,expr,),stmt,else,rest
13     | JSReturn (Maybe JSExpr) JSSemi
14     | JSWhile JSExpr JSState -- ^while,lb,expr,rb,stmt
15     | JSCallDot JSExpr JSExpr JSSemi
16     | JSStateList [JSExpr]
17       deriving ( Eq, Show)
18
19 data JSExpr
20     = JSId String
21     | JSInt String
22     | JSBool String
23     | JSString String
24     | JSIndex JSExpr JSExpr --model[0]
25     ------------------------------------
26     | JSTagList [JSExpr]
```

```
27      | JSList [JSExpr]
28      | JSVarInitExpr JSExpr JSExpr -- id = initializer
29      | JSRecord [(String, JSExpr)]
30      | JSMemberDot JSExpr JSExpr -- firstpart.name
31      | JSMemberExpr JSExpr [JSExpr] -- expr(args)
32      | JSMemberNew JSExpr [JSExpr] -- new, name(args)
33      | JSExprBinary JSExpr JSBinOp JSExpr -- lhs, op, rhs
34      | JSFunctionExpression String [String] [JSExpr] -- ^fn,name,lb, parameter
            list,rb,block`
35        deriving ( Eq, Show)
36
37  data JSBinOp
38      = Divide  -- /
39      | Eq      -- =
40      | Ge      -- >=
41      | Gt      -- >
42      | Le      -- <=
43      | Lt      -- <
44      | Minus   -- -
45      | Neq     -- =
46      | Plus    -- +
47      | Times   -- *
48      | Andand  -- &&
49      | EqEq    -- ==
50      deriving ( Eq, Show)
51
52  data JSSemi = Semi
53      deriving ( Eq, Show)
```

Appendix C

Essential Library of Constructing an IoT System using Elm

```
type SensorInput msg = SI msg Sensor
type IoTSystem   a b = IS a b
type IOSensors   msg = IOS msg Sensor
type IODevices   msg = IOD msg Device
type IOSignal        = SetHigh | SetLow

type alias Sensor =
    { s_type: String
    , s_address: Int
    }
type alias Device =
    { d_pin : Int
    , d_lib : Maybe String
    , d_func: Maybe String
    , d_dir : Maybe String
    }

bmp180 : Sensor
bmp180 = {s_type = "BMP180", s_address = 77}

tsl2561 : Sensor
tsl2561 = {s_type = "TSL2561", s_address = 39}

light : a -> Sensor -> IOSensors a
light a b =  IOS a b

```

```
27 temperature : a -> Sensor -> IOSensors a
28 temperature a b = IOS a b
29
30 buzzer : IOSignal -> Device-> IODevices IOSignal
31 buzzer a b = IOD a b
32
33 led : IOSignal -> Device-> IODevices IOSignal
34 led a b = IOD a b
35
36 fan : IOSignal -> Device-> IODevices IOSignal
37 fan a b = IOD a b
38
39 iot : List (IOSensors a) -> List (IODevices b) -> IoTSystem (List (IOSensors
       a))  (List (IODevices b) )
40 iot a b = IS a b
41
42
43 buzzer_1 : Device
44 buzzer_1 = {
45      d_pin   = 18
46     ,d_lib  = Just "onoff"
47     ,d_func = Just "Gpio"
48     ,d_dir  = Just "out"
49     }
50
51 led_1 : Device
52 led_1 = {
53      d_pin  = 12
54     ,d_lib  = Just "onoff"
55     ,d_func = Just "Gpio"
56     ,d_dir  = Just "out"
57     }
58
```

```
59 fan_1 : Device
60 fan_1 = {
61      d_pin  = 16
62     ,d_lib  = Just "onoff"
63     ,d_func = Just "Gpio"
64     ,d_dir  = Just "out"
65     }
66
67 onLightChange : (Int -> Msg) -> Sensor ->  SensorInput Msg
68 onLightChange f s = let m = f 1 in SI m s
69
70 onTemperatureChange : (Int->Msg) -> Sensor ->  SensorInput Msg
71 onTemperatureChange f s = let m = f 1 in SI m s
72
73 first (a, b) = a
74 second (a,b) = b
```