A Functional Language *for* Programming Small Robots

Adam D. Barwell

MEng Computer Science

Supervisor
Dr Graham Roberts

Friday, 26 April 2013

Abstract

This project explores the value and role of functional programming in the field of robotics. We consider the nature of small robots, and how they are presently programmed. We seek to answer the question of whether the application of the functional style to the programming of these small robots under an educational context is beneficial, and if so, how it might be achieved.

As the field of robotics is relatively large, this project focusses on a specific target platform. We perform a review of the functional style, exploring the features and design of select functional languages.

To illustrate our findings and suggestions resulting from this, we create a prototypical functional language. This necessitates an exploration of what is expected and required of any language to program small robots, an exploration we undertake.

The project ends with an evaluation of the idea to apply functional languages to robotics, under the context of the developed language.

The project achieves moderate success. It produces a simple programming language in the functional style that connects with the target robots' emulator. The language is effective as a prototype, and we explore how to further mature the language, and other methods that might be adopted.

Finally, we conclude the functional style is both suited and beneficial for both robotics in general, and the teaching of software engineering skills.

Contents

Chapter 1

Introduction

This project attempts to create a functional language that can be used to program small robots. We explore why functional languages are suited to such an endeavour, and how such a system might be implemented.

1.1 The Problem

As computers and their constituent parts become cheaper and easier to access, the manner in which they are used leads to many possibilities. These take the form of commercial and amateur projects alike. The creation, and programming, of small robots being but one example.

With the growing importance and changing shape of the computer science field, so too must the concepts taught, and the means with which we teach them, change. The use of the aforementioned small projects in teaching is one way to adapt to these changes.

One example of this is the use of small robots in education. Robotics requires and teaches a range of skills. Where any non-trivial robot requires the programmer to consider a range of factors, such as motor speed and sensor readings, for his program to be successful in carrying out its intended task.

Such considerations require an understanding of a program's structure. Furthermore, as these robots may be built with inexpensive, single-chip computers, the skills learnt during students' experiences in programming for them are eminently transferable to the 'real world'.

Aiming to teach first year undergraduate students these skills, and more, the Department of Computer Science at University College London has recently introduced a course module centred about the programming of small robots. These robots are developed within the department and remain a work-in-progress.

The UCL robots are movable, feature a number of sensors, and may be controlled by coding in C. No other language is currently supported. Although it has inspired many others, and remains generally important, C is not a modern language.

'Functional languages' provide a completely different approach to programming. More than an academic intellectual curiosity, they give greater freedom than 'impera-

tive languages' in modularity and structure. By having no side-effects, and freeing the programmer of the task of manipulating flow of control, they are suited to concurrent and parallel applications alike.

This, in addition to being effectively at a higher level of abstraction than imperative languages, make them suitable for the task of programming robots. Especially small robots, such as those developed by UCL.

Under the purview of teaching, functional languages can help the student consider problems in a more abstract manner. A feat that is often hindered by the low-level concerns inherent in imperative languages. Both paradigms have their advantages and weaknesses, meaning that C being the sole option is unfortunate.

In this project we use the UCL robots as a base for our investigation and implementation. We investigate the nature of functional languages, and how their application to UCL's *Robotics Programming* module proves advantageous. We define and develop a prototypical functional language to illustrate how the functional style can be applied to the programming of small robots, and to show these benefits in context.

1.2 Aims and Goals

The primary goals of this project focus on the evaluation of applying the functional paradigm to the programming of small robots, and may be found below.

1. To consider how the functional programming paradigm might be applied to the programming of small robots.
   (a) To understand the problem, and challenges faced, when programming small robots.
   (b) To understand the capabilities of currently available tools.
   (c) To research the functional programming paradigm and languages, considering how they might enhance current solutions.
2. To build a prototypical platform using functional programming to solve the problem.
   (a) To define the project's scope and requirements.
   (b) To design and implement a platform that highlights the benefits brought to the solution through functional programming.
   (c) To test the features of the implementation.
3. To evaluate the effectiveness of the functional paradigm when applied to the problem.
   (a) To assess the developed prototype based on the goals laid out.
   (b) To assess the methods and choices made of the developed prototype.
   (c) To evaluate the extent of the beneficial nature of the functional paradigm when applied to the problem.

1.3 Project Approach

In accordance with the above three goals, we approach the project in three stages. First, we aim to understand the problem space through a little research. Secondly, we build the prototypical platform using that which we learn during the first stage. Finally, we consider the success of the project and prototype.

1.4 Annotated Contents

Chapter 1 – Introduction
This chapter. Gives an overview of the project.

Chapter 2 – Background and Related Work
Pertaining to the first goal listed above, this chapter details the initial research undertaken at the genesis of this project. It explores the motivation behind the project, any related work, and the general means of implementation.

Chapter 3 – Requirements
Giving the requirements and scope of the project, this chapter lists what is expected of the prototype.

Chapter 4 – Design and Implementation
This chapter details the design and implementation choices made during the project's course. It includes a review of the elements that are most relevant to the functional programming paradigm.

Chapter 5 – Testing
Detailing the method and approach used in testing the implementation of the prototype platform.

Chapter 6 – Evaluation and Conclusions
Finishing this report, we assess the successfulness of the prototype and project. We consider how both might be extended into the future, and give our final thoughts on the project.

Chapter 2

Background Information and Related Work

In this chapter we explore the motivation for undertaking the project. First we ask why the project exists, looking at the current state of the project domain and how we have come to form the foundation of the ideas explored in this report.

Through this task we consider currently available solutions. Exploring their requirements, successes, and shortcomings. Upon which we discuss possibilities for improvement, and how these might be achieved.

2.1 The Problem

Whilst today computers continue to grow evermore ubiquitous; in the past, they were severely limited machines, requiring knowledge of complex and unwieldy languages. Conversely, the computers we use now are both powerful and significantly easier to operate.

This ubiquity has been driven by a number of simultaneous factors: from the shrinking size of components, through the continued growth in computational power, to the relatively low cost of materials. This, combined with the pursuit of new markets, has resulted in form factors no longer being restricted to the iconic beige box atop a desk.

Amongst the increasingly impressive tablets and smartphones however, are a breed of computers that appear to be taking a step backwards. These computers have all components on a single chip, have little processing power, and do not come with the myriad accessories to which we have become accustomed.

Yet these little computers are proving reasonably popular. Indeed, they have inspired a number of amateur projects. [58, 56] One such example being the creation and development of robots. [56]

Another consequence of this ubiquity is the similarly increased need to adjust the teaching of both how to use computers, and more importantly, how to program them. With the landscape in computer science rapidly changing, there have been calls to

improve the quality of this teaching, using a broader range of methods that reflect the current state of the discipline and market. [50, 53, 47, 45]

Inspired by both amateur projects, and motivated by the need for relevance, University College London (hereafter UCL) has recently introduced a course to teach their students a range of programming skills through practical exercises alone. The course, introduced in the 2011/12 academic year, divides the students into groups and gives them a series of tasks to complete. These tasks are designed to incrementally introduce the students to a range of ideas, thus teaching them various software engineering concepts. [60]

The course aims to be interesting, by instead of asking the students to complete trivial, relatively abstract tasks; it uses small, programmable robots. [60] These robots are primarily developed by Professor Mark Handley of UCL, and use a range of standard, inexpensive components. Hence, providing students with a movable robot that can sense the world around it. [62]

Each robot consists of a number of components; these include: a Raspberry Pi computer,[1] an Arduino computer,[2] two motors, a motor controller, a range of short- and long-range sensors, and batteries. Where the motor and sensor components are self-explanatory, we note that the two computers are tasked with distinct responsibilities. [35]

The Arduino interfaces with the low-level components themselves – passing them commands and returning their responses. Whereas the Raspberry Pi acts as the interface between user-given code and the Arduino. It is on the Raspberry Pi that students run their code. [35]

The Raspberry Pi used is a 700MHz ARM-based computer, with 256GB of RAM, and 4GB of storage space. It runs the Raspian Linux distribution, and can be connected to via SSH over a WiFi connection. We note that this interface can be changed, indeed it already has been changed. In the first year of the course's run, a smartphone running the Android mobile operating system was used.[3] [34] Hence, there is a reasonable possibility of the Raspberry Pi being changed in future developments.

The Raspberry Pi computer provides two programming interfaces, or APIs, titled: Synchronous ASCII, and Asynchronous ASCII. The former is the simpler of the two, and the most often used. It allows the programmer to control the robot's components and receive sensor readings through the transmission of formatted strings. [35]

Said strings have been kept small and simple to ensure commands are passed to the Arduino computer, and evaluated therein, with minimal cost. As such, they are case sensitive, and must rigidly follow the format:

$$command\ [options]$$

Where a *command* is a single upper-case letter as defined in [35], and *options* depend upon that command.

The programmer, in this instance the student, must then use *The C Programming Language* [46] to automate the actions of the robots in line with the tasks set. Using

[1] Website: `http://www.raspberrypi.org`
[2] Website: `http://www.arduino.cc`
[3] Website: `http://www.android.com`

C, with libraries written in support, provides a suitable base upon which the students can employ, and subsequently develop, their programming knowledge and skills. On top of which, new software engineering concepts may be introduced.

One of several limitations of the course is the number of supported programming languages, with C being the sole option. Whilst it remains an important language, C is not representative of all modern programming languages and paradigms. Whilst we concede that many of said modern programming languages are influenced by C, we nevertheless note that other programming paradigms exist for which it remains foreign.

Different programming paradigms, or styles, naturally have different advantages and weaknesses. Indeed, whilst imperative languages are most common, other styles remain in use. We observe that these styles need not be used exclusively of one another – the Make and Apache Ant[4] build tools use a combination of styles, for example.

Therefore, we might argue it advantageous to know and understand several programming languages belonging to several paradigms. Furthermore, the lack of support beyond the imperative C, makes this a limitation of the course and provided materials.

In considering the spectrum of programming languages and styles, we find that most languages today are imperative. Imperative languages are defined by their implicit state, which is manipulated by the programmer through assignment, and control flow operations. [39] Examples of imperative languages include: Java,[5] Python,[6] and Ruby.[7]

Declarative languages fundamentally differ from imperative languages as they have no implicit state, instead explicitly carrying it. They have been often described as tasking the programmer with the job of describing the desired program without needing to be concerned with the flow of control. [39] Two examples of declarative languages include: SQL, and SWI Prolog.[8]

Within the above two paradigms we find a number of other, smaller paradigms. These can either be sub-paradigms, such as 'logic programming' under declarative, or styles that aim to improve another, such as 'object-oriented programming'. Functional programming is a programming style under the declarative paradigm. It is this particular style, and the languages that implement it, that we are interested in.

2.2 Functional Programming

Functional programming provides a number of distinct changes from imperative languages that can be highly beneficial. Yet before we proceed to detail these benefits, and how they apply to the problem, we first ask the question: what is functional programming?

[4]Website: `http://ant.apache.org`
[5]Website: `http://www.oracle.com/technetwork/java/index.html`
[6]Website: `http://www.python.org`
[7]Website: `http://www.ruby-lang.org/en/`
[8]Website: `http://www.swi-prolog.org`

2.2.1 A Brief Introduction

The genesis of functional programming is often attributed to Church's work on the lambda calculus. Created with the intention to capture the intuition behind the behaviour of functions, it allows both their definition, and their application to values and other functions. [39] Furthermore, as a calculus, it can be shown mathematically consistent. [13]

Arguably, the lambda calculus could be considered the first functional language. Yet, and whilst there have been a number of elaborations to the calculus, it is not alone in shaping today's functional languages. McCarthy's Lisp contributed to the aforementioned elaborations. Iswim, APL, FP, and ML each influenced and developed the functional style further. This eventually lead to the style we know and use today. [39]

As the name implies, the core operation of functional languages is the application of functions to arguments. Where a program is defined as a function, whose body may contain calls to other functions, and whose statements are executed when called upon. [40]

From these ideas arise the defining traits of functional programming. Assignments, if supported by the language in question, are immutable – i.e. once a variable is assigned, it cannot be re-assigned. More generally, we note that functional programs exhibit no side-effects. Any function call can only return its result, affecting nothing else. It follows that functions may then be executed at any time, thus relieving the programmer of prescribing the flow of control. [40]

It is from this the oft used description of the functional programming style comes – i.e. one specifies *what* one wants the program to do, not *how* to do it.

Whilst the above properties might be considered beneficial in their own right, the different functional languages available hold other advantages and specialities. Indeed, several functional languages exist, and continue to be developed. These include:

- Erlang,[9]
- Haskell,[10]
- F#,[11]
- Miranda,[12]
- Amanda, and
- Scala.[13]

Having introduced the functional style, we now explore its relevance, and thus its suitability in its application to the problem. Is functional programming useful in the real world, or is it simply an academic curiosity?

[9] Website: `http://www.erlang.org`
[10] Website: `http://www.haskell.org`
[11] Website: `http://www.fsharp.org`
[12] Miranda is a trademark of Research Software Ltd. Website: `http://miranda.org.uk`
[13] Website: `http://www.scala-lang.org`

2.2.2 Relevance

Whilst functional programming is neither as well known, nor as often used as its imperative counterpart, functional languages remain suitable and beneficial under certain circumstances. Their general flexibility, and ability to handle infinite lists both contribute to their use in financial institutions, for instance. [27] They also remain an active area of research and development under academia, with research into how they might assist in the utilisation of true parallelism, and how they might successfully be applied to constrained systems. [33, 11, 10]

Their effects may also be felt when using modern imperative languages. Java, for example, automates memory management. Part of this management system is the use of 'garbage collection' – a technique derived from functional programming. [49, 43]

Therefore, functional languages may be considered interesting and useful in both academia and the real world. So why is it beneficial to apply them to the aforementioned problem?

2.2.3 Advantages

First, we remark that, as they *are* both interesting and useful in the real world, it is naturally beneficial that they might be taught to students. Functional programming is currently taught on two occasions at UCL: under *Principles of Programming* in the first year of study, and optionally under *Functional Programming* in the third. These two modules teach the general concept of functional programming using a relatively abstract approach. [55, 25]

The former contrasts the style with imperative programming; where the latter presents a comprehensive exploration of the style, concepts, and how such languages are evaluated. [55, 25] In contrast, the larger software engineering projects of the curriculum mostly use imperative languages. [65] Introducing a functional option for programming the robots under the *Robotics Programming* module would, therefore provide greater involvement for functional programming in the curriculum.

Secondly, and following from the previous point, we observe that the aim of the module is to teach students a range of skills. Amongst others, these include: effective code structuring, software engineering in teams, and problem solving. [60]

In using software to solve problems, one is influenced heavily by the language one uses. The available libraries, tools, and the syntax itself all contribute to how one chooses to solve a given problem. There might be a library that simplifies the problem in Java, but not in Ruby, for example.

Here, we again highlight the difference between functional programming's declarative style, and C's imperative nature. The latter teaches one to consider the problem from a lower level – in effect, teaching how to engineer a solution. Where the former encourages the student to consider the problem from a more generic, logical perspective – in effect, teaching how to conceptually consider a problem. Neither is always better than the other, yet both are useful under the computer science discipline for practical and theoretical purposes alike.

Finally, we consider current research. Two notable areas present themselves: that functional languages have been the subject of research into the introduction of paral-

lelism through refactoring [11, 10]; and that functional languages are a possible candidate as the base for provably bound languages. [33]

The former is considered beneficial because many modern processors no longer have a single core, but several. Functional programming is seen as especially suited for this task because programmers need not worry about the flow of control. Therefore, the programmer is not burdened with the need to consider locks and other aspects. This, in turn, provides a simplified interface for an otherwise relatively complex task. [11, 10]

Whilst true parallelism is concerned with multiple, physical cores; concurrency is similar in many respects. Understanding the robot's state, and issuing multiple commands is likely to require some means of concurrency. Concurrency uses similar constructs, such as locking and synchronisation, to true parallelism. Meaning, the functional paradigm is also useful when designing concurrent systems.

With the latter, it is argued that provably bound systems are increasingly necessary as the aforementioned diversity of computers leads to systems that are constrained in hardware, yet continue to be programmed with fully featured languages. Functional languages are chosen as a base for this task, as again, they are effectively automatic when compared with imperative languages. This automation, combined with the relative similarity to standard mathematical systems, allows the construction of a language that is suitably constrained such that it can be proven to execute within given bounds. [33]

From these four considerations, we might reasonably conclude that a functional language designed to program small robots is both feasible and, in some ways, beneficial. We must now consider how to best introduce the functional paradigm to the current system.

2.3 Related Work

Few topics in any academic discipline are novel. We therefore, seek to understand whether a similar project has already been undertaken. If this is the case, we explore how the two projects are similar, or otherwise.

Indeed, in programming robots there are examples of the application of functional languages. We take a brief look at three such examples.

The first work, [54], uses the purely functional language Haskell to demonstrate the power of declarative languages in robot control. The authors perform a very similar task to that which this project is centred about. One considerable difference however, is the use of Haskell to control the low-level aspects of the target robot. This effectively makes their target the Arduino computer.

Similarly to [54], we find the authors of [12] attempting to control robots through the use of Erlang. Again, we find the aim is to introduce control at the Arduino level. Yet, the authors here intend to introduce concurrency through Erlang's support of the concept.

Finally, in [63] we find the authors attempting to add yet more complexity. The small robots they aim to program are not simply small robots with the standard array of motors and sensors, but are self-reconfigurable. Such robots require yet more

commands, with each command having a stronger coupling with the others. This is far beyond the level of complexity of UCL's robots.

For all three examples, we observe that these languages focus at a much lower level than the intention behind this project. We also observe that these projects are not concerned with robots in an educational capacity.

2.4 Applying Functional Programming to the Problem

Thus far, we have explored the problem domain, what functional languages are, and how they have already been applied to robotics. In this section, we consider a feasible method of applying the functional style to the problem.

Generally speaking, we find there two means of achieving this. The first is to use an existing functional language; and the second, the creation of a custom domain specific language (hereafter DSL).

2.4.1 Existing Languages

We have already seen in Section 2.2.1 that a number of modern functional languages exist. Some of these are purely functional, where others include non-functional elements. Each have their own motivation behind their development, and accordingly, each have advantages and weaknesses.

The use of an existing functional language comes with the inherent advantages of mature languages. They commonly provide a rich feature set, sufficient support, and the assurance that the language is robust. It also provides the possibility of libraries and tools to make things easier.

Yet, for education purposes the question of their suitability is raised. Whilst a mature tool set is desirable, the shift in thinking required to transition from imperative languages could possibly make the task excessively difficult for first year students.

This could, in part, be due to the range of features being greater than that which is required for the relatively limited task of controlling robots. In addition to the required shift in thinking, this may arguably make the task too difficult.

We are then subjected with the question of which language to use. The chosen language must be representative of the functional style. Should we therefore, choose a purely functional language? Whilst indisputably representative of the functional style, these languages also contribute to the difficulty of the task. Conversely, were we to use a non-purely functional language, we must then consider whether this is representational of the style and so beneficial in its use.

2.4.2 Domain Specific Languages

A domain specific language is inherently limited, which is beneficial in this instance. Furthermore, as such languages are built with their target platform or task as their focus, the domain specific language will naturally be suited to the programming of small robots – thus eliminating the concerns regarding complexity.

There are a number of possible means to create a DSL, which we shall now consider. Within these, there are two overarching possibilities: to build a language from nothing, or to modify an existing language. We consider the latter first.

Altering an AST

One exemplar language whose AST can be altered is Groovy.[14] A dynamic scripting language based on Java, it has a number of advantages for its use in the creation of domain specific languages.

Firstly, it is based on Java. The Java Virtual Machine helps to ensure cross-platform compatibility. Furthermore, it offers the strength and maturity of the Java language, whilst offering the power and flexibility of other scripting languages.

Secondly, and perhaps more importantly, Groovy supports the creation of domain specific languages. [67] Whilst it is possible to overload core Groovy methods and functions, one can also alter the Groovy AST itself. Achieved through the use of special methods, one may rewrite and define how tokens and syntax rules are used and interpreted. [31]

However, we note that we intend to build a *declarative* domain specific language. As such, this requires the overriding of much of Groovy's core AST, so leading to questions of suitability.

Indeed, as the creation of such a domain specific language would require the rewriting of the majority of Groovy's AST, we therefore consider such a course of action to be ill-advised and inefficient. Instead we suggest building a language ourselves to be the better course of action.

Building a DSL

The creation of a language is a relatively mature process in computer science. As such, a pattern may be followed to its completion, and various tools might be used. In this process we first consider the language we use to build the compiler as this will affect the choice of tools used later on in the process.

To adequately select a language, we consider the currently existing tools and the target robots. Upon inspection, we find the code that runs on the Raspberry Pi, and the surrounding tools, is largely written in Python. Hence, and as the robots continue to be developed, it is believed Python to be the best choice of language to create our DSL.

Having chosen Python, we now consider the disparate elements required for the creation of a language. In most cases, three things are needed:

1. a lexical analyser,
2. a syntactic analyser, and
3. an interpreter.

[14] Website: `http://groovy.codehaus.org`

We explore each of the above in order.

Lexical analysers, otherwise known as 'lexers', match their input to a list of regular expressions, producing a stream of 'tokens'. Through the execution of this task they perform two services: firstly, they ensure that the given input is lexically correct for that language specification, i.e. they ensure no invalid characters are present; and secondly, they normalise the raw input data so that it can be fed into the syntactic analyser.

The syntactic analyser, or 'parser', then analyses this token stream according to a given grammar. Ultimately this results in an AST, or similar representation, of that particular input.

These two elements are commonly generated, given a specification, through selected tools, of which many exist. Perhaps the most commonly known are Lex and Yacc, both of which are designed to generate lexical and syntactic analysers in C. In Python, there are three such prominent tools.

The first tool is ANTLR.[15] Supporting C, C#, Java, and Python, ANTLR is a parser generator that also generates a lexical analyser. Its syntax is relatively simple and is used in a number of projects. Yet, it should be noted that Python, whilst supported, is not the primary target language. [3]

Second is another top-down parser, PyParsing.[16] The module aims to simplify the creation of parsers through a library of classes that the programmer uses to define the grammar directly.

Third is PLY, or Python Lex-Yacc.[17] As the name implies, it is a bottom-up lexer and parser generator that is based on the aforementioned Lex and Yacc. In two parts, PLY's lexer and parser generators may be used independently or together. It requires the user to define both regular expressions for tokens, and rules in a particular format to define the grammar. Compared with the other two, PLY is generally faster. [52] This project uses PLY for its speed, Python focus, and the power and versatility offered with BNF and regular expressions.

2.4.3 Interpretation

The DSL will invariably have an interpreter component, whereby the representation of the given program is executed. It is possible to construct these interpreters using standard Python classes and software engineering techniques, hence we shall detail its construction in a later chapter.

However, as the DSL is to command small robots, we consider how it will interact with its targets. One means of doing this is to include the interface within the compiler and interpreter itself.

Yet, this introduces greater coupling. With any change to the robots' interface, as has already been established likely, the interpreter itself must be changed. Indeed, for any and every possible DSL that interfaces with the UCL robots the same changes must be made in each instance.

[15] Website: `http://www.antlr.org`
[16] Website: `http://pyparsing.wikispaces.com`
[17] Website: `http://www.dabeaz.com/ply/`

An alternative to this method is the creation of an API that interfaces with both the target robots, and any and all languages that might wish to control the robots.

As the latter option is not only more generic, but allows greater flexibility in development, we create both a DSL and an intermediate API. Where the API sits between the DSL and the target robots' own interface.

Having considered the problem space, similar projects, and the basis for undertaking our proposed solution, we must now explicitly define what is required of the project and of the proposed language.

Chapter 3

Requirements

In this chapter we define the requirements of the project and language. First, we give the full problem statement. Secondly, we list the requirements for the project. Finally, we consider the results of our analysis of the aforementioned requirements.

3.1 Problem Statement

In this project we propose a functional language to program small robots. Said robots are those used by the UCL first year undergraduate course module, *Robotics Programming*.

We use the 2011/12 academic year version of the robots. At the start of the project, the eventual changes had not yet been completed. This resulted in the need to use the older version, but also to consider the changes being made to the robots throughout the project's duration.

The youthfulness of the course module and development stage of the robots themselves leads to capable but limited robots, and only one supported programming language. Additionally, we note the students' benefit of having increased knowledge and experienced gained through the practical application of a functional language in programming robots. Thus, providing the basis for our motivation behind this project.

A prototype of this language is to be built to illustrate the concept and idea behind the project. To build the DSL, we use the Python programming language with the Python PLY module. We also develop and use an interface between our DSL's compiler and the robots. At this stage we focus on connection to the target robots' emulator, as developed by Professor Handley of UCL.

The DSL and API shall be built and tested over the course of two academic terms during the 2012/13 academic year, and a report written to document the project.

3.2 Requirements

We list the requirements for the project and language in the below table. In listing these requirements we use the MoSCow Analysis approach to highlight what the implementation must achieve, what it should achieve, and what it could achieve given sufficient time. [41] We note that the requirements featured here are changeable due to the research orientation of the project.

For clarity, we split the table in two; where we focus on the DSL requirements, followed by the API requirements. The former pertains to what is expected of the proposed, prototypical functional language. Whereas, the latter focusses on what is expected of the interface between the language and the target environment.

	Requirements for the proposed DSL
ID	**Requirement**
L01	The DSL must follow the functional programming paradigm.
L02	The DSL must connect to the developed API.
L03	The DSL must be capable of issuing commands to the API.
L04	The DSL must be capable of relaying messages to the user from the API.
L05	The DSL must have basic functionality for a language.
L06	The DSL must be extensible.
L07	The DSL must be usable by first year undergraduate students.
L08	The DSL should follow existing conventions and style for functional languages.
L09	The DSL could include a library with exemplar functions.
L10	The DSL could include a library of exemplar programs.
L11	The DSL could be computationally complete.
L12	The DSL could support concurrent operations.
L13	The DSL could produce intermediate code to run on the target robots directly.
	Requirements for the proposed API
A01	The API must connect to the target robot emulator.
A02	The API must provide a standard interface to which the language may connect.
A03	The API must translate commands from the connected language and pass them to the target.

Continued overleaf.

ID (CTD.)	Requirement
A04	The API must receive and pass messages back to the connected language.
A05	The API should be extensible.
A06	The API should be standardised such that it may be used with different interfaces and languages.
A07	The API could connect to the target robots directly.

3.3 Results of Requirements Analysis

In this section we give the results of requirements analysis. After looking at the developed requirements, a number of necessary elements became apparent.

These necessary elements primarily include the objects that define the DSL to be created. The listed requirements and the decision to use the Python module PLY necessitates the definition of both lexical tokens and a grammar.

The definition of these alone is not sufficient for the creation of a DSL, however. Indeed, a data structure will be required to represent a given program once parsing has completed successfully. This data structure will also need to be traversable to allow the execution of the program it represents.

Execution, in turn, shall require approximately two data structures to store information about that program. These are: a symbol table to maintain references to variable and parameter values; and a function table that maintains information regarding user-defined functions.

Yet we note that the DSL must communicate with the target emulator. The requirements state that this should be done via an API. This API requires two interfaces: one to communicate with the DSL, and another to communicate with the emulator. The former must be designed and agreed upon, and should be suitably generic to allow for different languages or interfaces to similarly connect. We remark that the latter must agree with the design of the emulator.

Chapter 4

Design and Implementation

In this chapter we explore the design of the developed system, and detail the methods by which it was implemented. We do this by considering the DSL, and the API that sits between said DSL and the emulator, separately.

For this chapter we assume a reasonable knowledge of software engineering practices. Hence, we primarily focus on the implementation details that are most pertinent under the application of the functional style to the programming of small robots.

4.1 Overview of the Approach

In this section we give a brief overview of the approach used in the implementation of both API and DSL. We include a brief listing of what tools were used.

Both parts were developed using a number of computers, featuring a combination of a MacBook Air (Late 2010), a MacBook Pro (2.4GHz Intel Core 2 Duo), and an iMac (Mid 2007). Each computer uses the Mac OS X operating system, specifically version 10.8 and above.

On each development machine, we use the default installation of Python – i.e. version 2.7.2. A number of text editors were used to develop the DSL and API, chosen for their language support features, and lack of satisfactory IDE or IDE plug-in for the Python language. These include: Sublime Text 2,[1] Chocolat,[2] and Vim.[3]

A version control system was used to simplify the development across all aforementioned machines. Such systems also allow the means to track changes made to the system during development, and revert to an earlier version if necessary. Git was chosen over other version control tools for its simplicity, modernity, and the additional effective backups created through each repository clone.

The repository was hosted using Atlassian's BitBucket service.[4] BitBucket offers free, private repositories under their academic user licence. It is for these private repositories that this service was chosen over similar repository hosts, such as GitHub.

[1] Website: `http://www.sublimetext.com/2`
[2] Website: `http://chocolatapp.com`
[3] Website: `http://www.vim.org`
[4] Website: `https://bitbucket.org`

Code	Command Class
`M`	Motor Commands
`I`	Infrared sensor positioning
`S`	Sensor readout
`C`	Robot mode configuration

Table 4.1: Emulator command codes and classes.
(These commands are quoted from [34].)

Regarding development, an iterative approach was employed. As this project is in part research focussed, this approach allows for changes to both the requirements and design to be made swiftly and easily.

4.2 The API

The primary duty of the API is twofold. It must provide an interface such that the DSL, another language, or program, can interact with it. Secondly, it must transmit commands to the target emulator, and receive any response.

We first consider the latter task as it will help define the former. The target emulator in this instance has the version number 121, and it is this interface that our API shall interact with.

As this is a core component, we spend a little time looking at the available commands and how they are formatted to ensure future sections are clear and understandable. The following information may be found in [34], and is quoted from there.

4.2.1 Emulator Commands

The target emulator is built in Python, and accepts the same API as the robots themselves. Once running, the emulator listens for commands on port 55443. A TCP connection can be made using this port, following which string-based commands are sent to the emulator. [36]

There are a finite number of commands, each following a strict pattern. According to the format in Section 2.1, we can define four classes of command. These are typified by a single uppercase character given first; these, and their classes of command, may be found in Table 4.1.

We now consider each in turn, looking at their respective formats and possible commands.

Motor Commands

The first we consider are motor commands; these provide the means to set the voltage of both motors. This is done in the following format:

`M` *motor* *speed*

Here, *motor* allows the programmer to specify which of the motors he wishes to set the voltage of. This is done via one of three commands using, once again, a single uppercase letter.

To set the left motor's voltage, one uses the command `L`. To set the right motor's voltage, one uses the command `R`. Finally, to set both motors' voltage in a single command, one may use `LR`. The concept of these commands will be used again in other command classes.

Finally, *speed* is defined as a decimal integer in the range $[-127, 127]$ – where a negative value indicates reverse motion. When setting both motors' voltages, two such values are required. These are separated by a single space.

Infrared Sensor Positioning

The infrared sensor positioning class pertains to the two infrared rangefinders located atop the robot. The commands for which are very similar to those just described under the Motor Commands.

Again, we find a similar format:

`I` *servo* *angle*

The *servo* variable is used in the same way as the *motor* variable above. Yet we note that *angle* must be a decimal integer in the range of $[-90, 90]$. Again, if *servo* is defined as `LR` then two values separated by a space must define *angle*.

Sensor Readout

The robots represented by the emulator have a number of sensors, which can be read from using a series commands. Again, we find a similar basic command structure:

`S` *sensorname*

We note that, unlike the previous two command structures, no argument is needed. Instead, *sensorname* is constructed to give the full range of available sensors. We find the full list of these names in Table 4.2

Robot Mode Configuration

Under the current system and documentation, the mode commands are unused.

4.2.2 Architecture

As stated previously, knowing the interface to which we connect greatly influences the design of our API. Indeed, knowing how we might connect to the emulator, and the commands available to us, define much of our API. The former has necessitated the use of the `socket` module, for instance.

Hence, we now turn our attention to the overarching structure of the API – our architecture, if you will.

Name	Sensor Type and Location
IFL	Infrared rangefinder, Front Left
IFR	Infrared rangefinder, Front Right
IFLR	Infrared rangefinders, Front Left and Right
ISL	Infrared rangefinder, Side Left
ISR	Infrared rangefinder, Side Right
ISLR	Infrared rangefinders, Side Left and Right
US	Ultrasound, Front
BFL	Bumper, Front Left
BFR	Bumper, Front Right
BFLR	Bumper, Front Left and Right
V	Battery Voltage
MEL	Motor Encoder, Left
MER	Motor Encoder, Right
MELR	Motor Encoders, Left and Right
MCL	Motor Current, Left
MCR	Motor Current, Right
MCLR	Motor Current, Left and Right
IBL	Infrared reflectometer, Bottom Left
IBC	Infrared reflectometer, Bottom Centre
IBLC	Infrared reflectometer, Bottom Left and Centre
IBR	Infrared reflectometer, Bottom Right
IBCR	Infrared reflectometer, Bottom Centre and Right
IBLR	Infrared reflectometer, Bottom Left and Right
IBLCR	Infrared reflectometer, Bottom Left, Centre and Right

Table 4.2: Sensor Names for use in sensor readout commands.
(These commands are quoted from [34].)

Overview

The Python language is very flexible compared to languages such as Java. This leads to comparatively looser standards and general concepts. Python is dynamically typed, and its fundamental unit is the *Module*. The language can be used as either a scripting language, or applied to the object-oriented style.

For this project, we use the object-oriented style as the ability to generate objects that may be ordered is desirable in both the API and the DSL itself. By using objects in the API we have the flexibility to generate numerous instances of a variable or command, and introduce some encapsulation to a language that is relatively open.

Hence, we split our API into a number of modules, and further into classes. Where modules are used to separate different concepts. In the context of this project, and with regard to the emulator interface, we might divide our API into three different modules. Conceptually speaking, these consist of:

1. a module of predefined strings for use in generating commands;
2. a module containing functions pertaining to the emulator interface; and
3. a module containing functions that represent all possible commands.

These may then be split further into classes. In creating the classes we determine, for each module, which functions and variables are relatively similar. We define two similar functions or variables as elements that fall under the same command class, as defined in Section 4.2.1.

In the case of modules 1 and 3, we find this an easy task. It is possible to translate the string commands into functions, using predefined strings. These are then split into corresponding classes.

However, the module dedicated to interfacing with the emulator remains alone in its functionality. By design it does not belong to any of the aforementioned command classes. Hence, we simply define a single class that represents the emulator within the API.

We take the above concept and enhance it, making use of inheritance where useful. A detailed illustration of the classes, their functions, and variables may be found in Figure 4.1. As much of the implementation process consists of the manipulation of standard Python code, we shall not go into further detail here, but leave the UML diagram in its place.

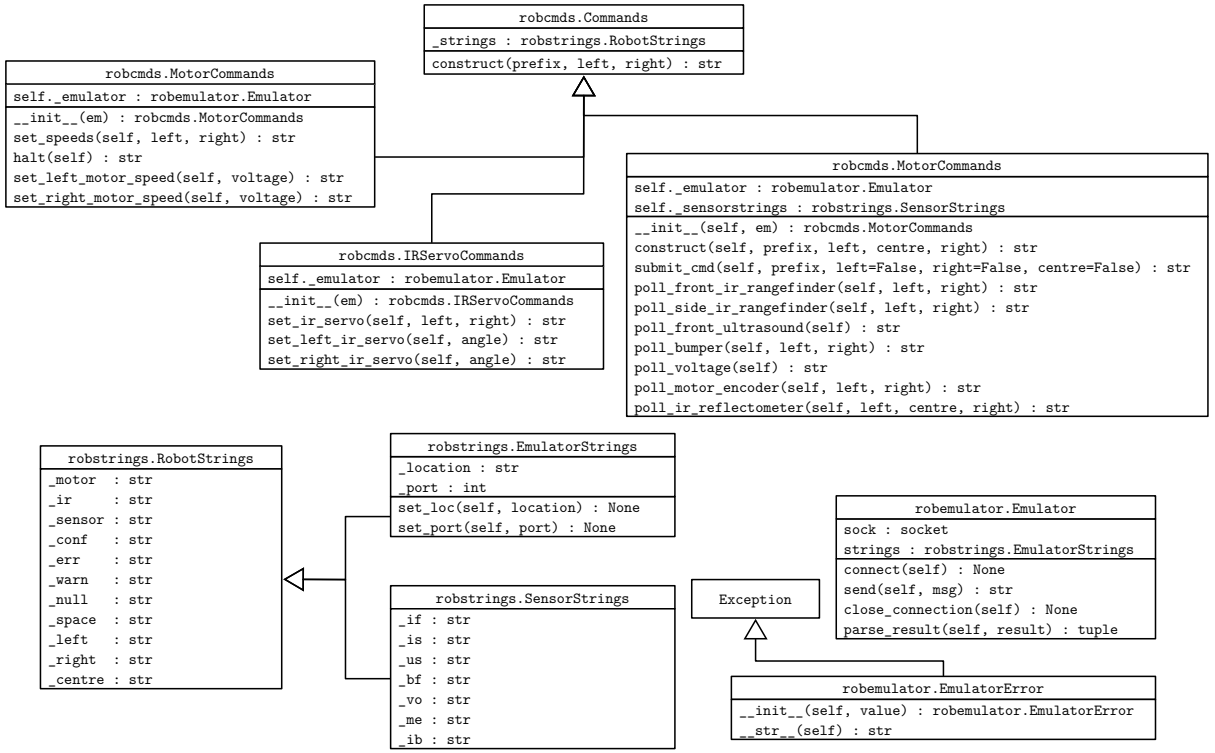

Figure 4.1: A UML Diagram Illustrating the Classes, Functions, and variables of the API.

Response and Error Handling

In defining the architecture of the API, we have primarily focussed on the connection to the emulator, and the subsequent transmission of commands. We now highlight how the API receives a response from the emulator and how that response is handled.

Again we must first know the types of response we are capable of receiving. These fall under four categories, listed below.

1. The null response.
2. The sensor response.
3. The warning response.
4. The error response.

The null response is received in response to movement and infrared servo commands alike. It is an indication of a successful command that has no further information to convey. From the emulator, a null response is in the format of a single full stop (`.`).

The second category of response, is also the second standard response. This occurs in the instance where the programmer requests some information from the robot in the form of a sensor command. Under this scenario, the command is returned along with the value, or values, requested.

Under the warning response category, currently three warnings exist. As the emulator interface is not entirely synchronous, these are sent in lieu of a standard response when a warning is raised. A warning has the format:

`W` *message*

Where *message* may be: `RUN`, `STOP`, or `TEST`.

Finally, is the more severe error response. Identified by the `ERR` prefix, they indicate that an error has occurred. This could be the programmer's mistake, or an error otherwise encountered.

With this, we may now determine how each of the four response classes are to be handled by the API. The aforementioned `Emulator` class is used to transmit the formatted string commands to the emulator; hence, we might also use the class to handle the responses.

As for every transmission a response is returned, we need only manipulate the string we receive. We use pattern matching techniques – including both Python's standard regular expression class (`re`), and string equality function – to determine the class of the response string.

For null responses we return Python's respective `None` object, and for both warnings and standard responses we return a `tuple` object. Whilst placing the burden of handling the response on the programmer might be considered bad practice, we note that the API and robots continue to be developed. In taking this approach, we allow for changes to the response format without needing to change the API.

Akin to the null response, we take advantage of Python's exception functionality for error responses. As seen in Figure 4.1, we create a new class that inherits from Python's `Exception` class. In using the exception functionality, we need not create our own error handling system. Furthermore, we benefit from the maturity and tools already available.

Thus, we conclude the design and implementation details for the API. We now consider the language itself.

4.3 The Language

In creating the DSL we again note a number of necessary elements. These elements include:

1. the definition of lexical tokens, thus the creation of a lexical analyser;
2. the definition of the language's grammar, and so the creation of a syntactic analyser; and
3. the creation and definition of an interpreter.

In this section, we look at each of these three elements. We explore the steps taken to design them, and the approach taken to implement them.

Regarding the architecture of the DSL compiler, we note that these three elements suitably define individual Python modules. At the end of development we total six such modules:

- `astlang`
- `astinterp`
- `astlex`
- `astparse`
- `astvisitor`
- `astsymtab`

In terms of composition, we examine the majority of these modules in the following subsections, under their respective categories. For `astlang`, which does not fall under any of the aforementioned categories, we report this the module one calls to run the compilation process.

Following invocation, `astlang` will check for a file to parse. If found, it will then open the document, analyse the contents, and execute the program found therein. It consists of two if-statements, and no classes.

Whilst strict adherence to the object-oriented style was celebrated for use under the creation of the API, we find a looser approach more beneficial under the creation of the DSL. This is, in part, due to the library modules we use and the requirements of the compiler. Again, we discuss the finer details of these in their respective subsections.

For clarity, we look at the elements of the compiler using the order under which they are invoked. We first detail the lexical analyser, swiftly followed by the syntactic analyser, finishing with an exploration of the interpreter. A review of the core, functional, language features is also present. The review details those elements of the DSL that are main features of functional languages.

4.3.1 The Lexical Analyser

The creation of a lexical analyser is relatively simple when using Python PLY. To do so, one defines a number of lexical tokens using a series of regular expression statements. The function `lex.lex()` is then used to generate the actual analyser.

Designing lexical tokens (hereafter simply 'tokens') is a relatively simple task. The same, or similar, tokens are used in multiple languages – it is their syntactic ordering that differs. Hence, the tokens used in our DSL are heavily inspired by a number of languages. The language with the most influence is Python itself.

Whilst many of the tokens and definitions are trivial, for those that were less trivial, other existing token lists were referenced. These included both the Java grammar and the examples found within the PLY module. [8, 48] One of these non-trivial rules includes both strings and character strings, differentiated by the surrounding quotation marks: '`"`' and '`'`', respectively. These rules may be formed in a number of ways.

We note that, as with other lexical analyser generators, we are capable of changing lexical state. Whilst Java uses this in its definition of strings and character strings alike, we elect to use the C definition of strings primarily due to the simplicity of their execution – especially in consideration of the aim to create a *prototypical* functional language for programming small robots. [8, 48]

In our final definition, strings return a token whose value consists of the string itself. Indeed, all tokens consist of their type, value, line number, and column number. This format allows us to report where in the text an illegal character is found, raising an exception to do so. [8]

Similar to strings, another non-trivial token is the comment. Comments are allowed via the full C notation once more. This allows single-line and multi-line comments alike. Both are, of course, ignored during execution. [8]

The `astlex` module was developed iteratively and reflected the needs of the grammar, always including the smallest set of possible tokens at any one stage in its growth. Token lists can grow very large, very quickly; yet the method used ensured that the token list was never too large or unintelligible.

Ultimately, this means the current token list is limited when compared with other languages. Yet at present, this does not worry us greatly. The full list of lexical tokens, including their respective definitions, may be found in Appendix C.

4.3.2 The Syntactic Analyser

Akin to the the lexical analyser, we use PLY to generate our syntactic analyser from a set of rules. These rules are adapted from the DSL's grammar. It follows, therefore, that we must first design our grammar.

In this section we focus on the steps taken to define our grammar, and ultimately build the syntactic analyser specification found in `astparse`. First, we consider the grammar of other functional languages. Secondly, we detail the approach taken to define the grammar. Finally, we highlight the extra steps taken to adapt the grammar to the format required by the PLY module.

Language Review

By looking at other functional languages' syntax we can develop an understanding of their general format, and thus produce a DSL that reflects the basic conventions and style of functional languages.

We compare three such languages: Miranda, Erlang, and Haskell. All are relatively prominent, with Miranda helping to develop the field itself. [51] Miranda and Haskell are lazy, purely functional languages; with Haskell being influenced by Miranda. [51, 42] Erlang is slightly different from the other two; it is a strictly evaluated, dynamically typed language with support for concurrent programming. [24]

Whilst the three languages differ from one another, each supports pattern matching, functions, higher-order functions, and lists. We remark that whilst Erlang supports assignment, Miranda and Haskell do not. Assignment statements in Erlang are immutable and might be defined in the following manner:

⟨*assignment*⟩ ::= ⟨*variable*⟩ '=' ⟨*expression*⟩

Where a ⟨*variable*⟩ starts with an uppercase letter or underscore, and contains alphanumeric characters, underscores, and the commercial at (@). Whilst a single underscore alone denotes an anonymous variable, it cannot be used in assignment statements. Variables may be bound to values or functions – denoted here by the non-terminal, ⟨*expression*⟩. [2]

The definition of lists is similar for all three languages, where they are denoted by comma separated values enclosed in square brackets. Lists can be represented and manipulated through the use of pattern matching, allowing access to the first element and the rest of the list.

Under Miranda and Haskell, functions are defined in a similar manner. We might define these using the following:

⟨*function*⟩ ::= ⟨*identifier*⟩ ⟨*patterns*⟩ '=' ⟨*body*⟩
| ⟨*identifier*⟩ '(' ⟨*patterns*⟩ ')' '=' ⟨*body*⟩

Here we note that two styles may be used. The first definition is for a 'curried' function, named for Haskell Curry who introduced the concept; the second uses a tuple. For clarity, we define ⟨*patterns*⟩ as a space- or comma-separated list of identifiers. [1]

Multiple function definitions using the same ⟨*identifier*⟩ may be defined. This functionality is heavily used when defining recursive functions, using pattern matching to distinguish between the generic and terminating cases. [1]

Curried functions allow a certain measure of flexibility when calling functions; this works well with Miranda and Haskell's lazy evaluation. Erlang lacks curried functions, instead opting for a more structured format to its function definitions – indeed, we suggest Erlang's definition more heavily inspired by imperative methods.

Erlang's functions may be defined in the following format:

⟨*function*⟩ ::= ⟨*variable*⟩ '(' ⟨*patterns*⟩ ')' '->' ⟨*body*⟩ '.'
| ⟨*variable*⟩ '(' ⟨*patterns*⟩ ')' 'when' ⟨*guard*⟩ '->' ⟨*body*⟩ '.'

Function definitions may also be overridden in a manner similar to that which is found in Miranda and Haskell. To construct this format under Erlang, one replaces the above full stop with a semi-colon to define the end of the body. Successive rules are then added using the same definition above, with only the final rule using the aforementioned full stop. [23]

Prototyping in Erlang

How can these definitions help us? We must first consider the (effective) two styles presented here in relation to the task our DSL is expected to undertake. Hence, we must look at their differences.

Two of the main differences between Erlang, and the Haskell and Miranda pair, are the support for assignment and concurrency. Indeed, we might suggest that all significant differences stem from the choice of lazy or strict evaluation and the purity of the adherence to the functional paradigm.

We must, therefore, decide whether it possible to implement a lazily evaluated functional language in Python, and whether it is a good idea. As the existence of the `lazypy` module [6] suggests, it is possible to make Python itself perform lazy evaluation. Thus, we may safely conclude it possible to build a language that features lazy evaluation.

Yet one question remains: is it a good idea? Lazy evaluation effectively separates data from control. This further frees the programmer from flow of control commands, and provides support for objects such as infinite lists. [39]

The efficient implementation of lazy evaluation is, however, a relatively complex task – initially taking ten years for a solution to be found. [39] Hence, we propose that lazy evaluation should not be given a high priority for our prototypical language. This will allow us to concentrate on the implementation of other features that are considered more pressing.

The desire to create a strictly evaluated language does not necessarily eliminate the choice of using Haskell or Miranda as our primary inspiration, however. Instead we look to Erlang's other, notable feature: concurrency.

In the programming of robots of any kind, one must be aware of the robot's state and possibly command several components at once. For this purpose, the ability to run multiple threads may be preferred. Indeed, whilst the target robots and emulator are relatively simple in this instance, the ability to manipulate threads is desirable from a teaching perspective.

Hence, we use Erlang to model how our language might be structured. To do this, we take the core desired functionality and implement that functionality in an Erlang module. Module 4.1 lists the result.

In it we provide functions for connecting to the emulator and several simple commands. The module may be loaded into the standard Erlang virtual machine, or exe-

Module 4.1: An Erlang Prototype Modelling Core Desired Functionality

```
-module(mockup).
-export([new_emulator/2, start_motors/3, start_motor/3, halt_motors/1,
   poll_sensor/2, start/0]).

new_emulator(Location, Port) ->
   {ok, Emulator} = gen_tcp:connect(Location, Port, [{active, false},
      {packet,0 }]),
   Emulator.

start_motors(Emulator, N, M) ->
   gen_tcp:send(Emulator, "M LR " ++ N ++ " " ++ M).

start_motor(Emulator, left, N) ->
   gen_tcp:send(Emulator, string:concat("M L ", N));
start_motor(Emulator, right, N) ->
   gen_tcp:send(Emulator, string:concat("M R ", N)).

halt_motors(Emulator) ->
   gen_tcp:send(Emulator, "M LR 0 0").

get_response(Emulator) ->
   gen_tcp:recv(Emulator, 0).

poll_sensor(ifl, Emulator) ->
   gen_tcp:send(Emulator, "S IFL"),
   get_response(Emulator);
poll_sensor(ifr, Emulator) ->
   gen_tcp:send(Emulator, "S IFR"),
   get_response(Emulator);
poll_sensor(isl, Emulator) ->
   gen_tcp:send(Emulator, "S ISL"),
   get_response(Emulator);
poll_sensor(isr, Emulator) ->
   gen_tcp:send(Emulator, "S ISR"),
   get_response(Emulator);
poll_sensor(bfl, Emulator) ->
   gen_tcp:send(Emulator, "S BFL"),
   get_response(Emulator);
poll_sensor(bfr, Emulator) ->
   gen_tcp:send(Emulator, "S BFR"),
   get_response(Emulator).

start() ->
   Emulator = new_emulator({127,0,0,1}, 55443),
   start_motor(Emulator, left, "20").
```

cuted as a script. We might further elaborate these commands and program by chaining the individual commands together, creating increasingly complex functions calling those already listed.

In regard to the DSL, we first note that the immediate connection to the emulator, and the creation of the commands is already under the purview of the API. Hence, these will not be necessary. However, some means of connecting to these functions will be. Thus, we conclude that built in functions that connect to the API shall be necessary. We also remark that the style of the function definitions used in Erlang are suitable for our purposes.

We further note that the definitions in lines 0 and 1 will not be required in our bespoke language. Regarding statements, we conclude that the standard binary operations will be beneficial in manipulating the core components. Also, that assignment statements will likely be useful when reusing voltage values and sensor angles.

Regarding types, and although Erlang and Python both allow a wide range, the target emulator, so the API also, requires only strings and decimal integers. This suggests an initial focus on these two types is advisable. Of course, boolean types are also required.

Other language elements, including higher-order functions, are likely to be desirable only at a later stage in development as they are not immediately necessary.

Forming the Grammar

Now we have an understanding of the general conventions and style of functional languages, as well as an understanding of how the language might be used, we can begin designing the grammar. The grammar was designed and implemented in an iterative manner.

First, a simple calculator was made. This was then made generic, and support for different features added in successive stages. This approach was taken as grammars can become very complex, very quickly. By splitting the development into stages, we are able to address any errors in the grammar quickly and efficiently. The developed grammar may be found, in full, in Appendix D.

As noted at the bottom of Appendix D, the grammar as it stands is not conflict free. We also comment that the grammar is not in, nor developed in, Backus-Naur Form. Instead, as with `astlex`, we define a series of functions that define the rules listed in the grammar which are then used to generate the syntactic parser.

Python PLY offers the ability to define the relative precedences for each token in a conflicting rule. It also allows the ability to define the token's associativity. We make use of this functionality to avoid the grammar's conflicts, and to avoid the arduous task of translating the grammar to one with none.

4.3.3 The Interpreter

Both a lexical token list and grammar have been defined, yet we are still not executing the given program. The lexical analyser checks for invalid characters, and the syntactic analyser ensures that it is also syntactically correct, but nothing more.

To actually execute the given program, four more elements are required. First, we require a representation of the program. Secondly, we need a means of traversing that representation. Thirdly, some means of storing values is necessary. Finally, so too, must function information be recorded. In this section, we will look at each in turn.

Representing the Program

There are numerous means of representing a given program. We could use a method similar to BASIC, where a program is a list of statements. Alternatively, we could use the well-established method of building an abstract syntax tree (hereafter, AST).

An AST is used here because of its relative simplicity and flexibility in representing a program's structure. Many of the benefits are felt in the tree's traversal and in the storage of information. In traversing the tree, we are able to use well defined tools, and in storing both values and function information we may choose our level of abstraction – do we evaluate first, or do we evaluate when needed?

To construct an AST we return to the module containing our grammar, `astparse`. Under the PLY module's definition, we are given the opportunity to execute and return the result of the last line in each rule-defining function. In the simple calculator examples, this was used to directly execute and return the result of the binary operations. In place of this, we shall create a series of objects to build the tree.

To build the tree, we first define the objects that act as nodes. In essence, all nodes are similar, a fact that we can make use of. Thus, we define the `Node` class that contains two instance variables: `_left`, and `_right`. To these variables we assign other `Node` instances; thus, in effect, making these two variables the edges of the tree.

This does not allow us to distinguish between the different possible constructs, however. Again there are a number of ways to do this. We choose to create a subclass for each type of node; chosen as they inherently, and succinctly contain the information we wish to store.

However, we concede that one complication arises with their use under Python. Unlike Java, Python does not allow the overloading of functions, and it does not demand that all types be declared before compilation. This gives rise to a problem when traversing the tree, which we shall now discuss.

Traversing the Representation

To traverse the tree, so that we might execute the program, we turn to design patterns. The particular pattern that we are interested in, is the Visitor pattern. This allows for methods or functionality to be added to a series of slightly different objects, without needing to extend or affect the object itself. [28]

Thus, we add the requisite accept method to each subclass of `Node`. Then a new module is created, titled `astvisitor`, enabling a number of visitors to be written. It is in the writing of these visitors that we encounter the aforementioned problem.

This problem is solved in a simple manner. In lieu of multiple, overriding visit functions, as with Java, we define one visit function. This function then contains an if-statement that checks the type of `Node` passed it, using the `isinstance()` function.

With one check for each `Node` subclass, we may then place the custom code for that visit function in the body of that if-statement.

The Symbol Table

Now halfway through our list of necessary interpreter elements, we consider the final two. Our first concern is storing the assigned variables and their values.

As functions feature prominently in the language, the scope of these variables is also a concern. Hence, we define the `SymbolTable` class in the same module. Each instance of `SymbolTable` represents a scope, where that scope may have: a parent scope, given by the instance variable `parent`; a list of children scopes, stored in the list variable `children`; the table itself; and a name.

The table itself is an adapted form of Python's inherent `dict` data type. We choose to implement our own dictionary data type to avoid reaching any limitations on dictionary depth during recursion. This may be found in the `astsymtab` module, which defines the class `Table`.

An instance of `SymbolTable` is created at the start of our `ExecutionVisitor`, used to execute the program, and represents the topmost scope of that program. Defined functions, when called, create new `SymbolTable` instances which are stored in the children list. A number of `SymbolTable` instances may be nested in this fashion.

Storing Function Information

A similar approach is taken when storing function definitions. The `FunctionTable` class is defined in the `astvisitor` module to store the definitions of the functions, such that they may be executed when called. Three variables are used to store this information: `names`, `params`, and `bodies`.

The first two store the name and parameter names, respectively, of the defined functions. The body stores the sub-tree representation of the function. This is traversed when the function is called.

In both the `FunctionTable` and `SymbolTable` classes we store the sub-tree representation of the assigned expression or body in lieu of its evaluated content. This is done to better support the possible implementation of lazy evaluation in the future.

With these four elements now in place, we are capable of generating a tree representation of the given program during syntactic analysis, and subsequently execute that given program according to our visitor class. This visitor connects to the API, which enables the programmer to send commands to the target emulator, affect that emulator, and receive a response. Thus, and though it may be developed further, we conclude our compiler complete at this stage.

4.3.4 Language Feature Review

Having completed the overarching description of the implementation of the DSL, we now take a more detailed look at the key features that make our DSL a functional language. In this section, we shall look at three such features:

1. assignment,
2. recursion, and
3. lists.

Before we explore these, we offer a short observation on the typing used in the DSL. All three of the languages we considered in Section 4.3.2, are dynamically typed. Indeed, in our aforementioned interpreter we do not explicitly check for type. For this functionality we rely upon Python's typing system, effectively making the developed DSL dynamically typed.

Whilst we recognise that this might cause problems in the future, we consider this an acceptable risk for our prototypical DSL. Our own static typing system, or dynamic safeguards, can be implemented during further development.

Assignment

Unlike Miranda and Haskell, we allow assignment. We clarify that this is *single assignment* as in Erlang, ensuring no side-effects can arise and eliminate one of the advantages of the functional programming style.

We do mention that singleton types are not supported in the DSL, as in all three of the listed functional languages. This is primarily because we rely upon Python's dynamic typing system. We also remark that singleton types are not a necessity in programming small robots; they are however possible in case of future development.

Recursion

Without the programmer's ability to specify the flow of control, recursion is the primary tool to create multiple, but slightly different, successive calls to a function in functional languages. In effect, they are the functional style's answer to the loop. Hence, recursion is an essential part of the DSL.

Both the `SymbolTable` and `FunctionTable` classes have been designed in order to ensure that recursion is possible. `SymbolTable` is capable of containing a reference to other instances of itself under the `children` instance variable. Hence, when the body of a newly called function is entered, a new `SymbolTable` is created and a reference to it is added to the children of the current scope. This occurs regardless of whether the function called is different or the same as the current function. Furthermore, and as mentioned previously, to avoid depth limits on Python's dictionary data type, the `SymbolTable` class uses a custom implementation of that data type.

`FunctionTable` is not intended to hold copies of itself, however. Instead, we hold the sub-tree that represents the body of the function in the table. This allows

us to reference, and visit, that sub-tree as many times as is necessary. The idea behind this is inspired by similar implementations in other functional languages, where conceptually, copies of sub-trees are created with every function call.

In combination, both tables allow for recursive functions to be defined and executed.

One subtle implementation detail that could cause problems was discovered during testing. Python is not optimised for tail recursion, and so sets an artificial limit to how deep one may recurse. The repercussions of this limitation are explored in Chapter 6, but for now this problem has effectively been solved by increasing the limit millionfold.

Lists

Akin to recursion, lists are the primary, non-trivial data type in functional languages and so should be included in the DSL. Rather, it is not the lists themselves that are important, for their implementation is relatively simple, it is the way they are manipulated.

Functional lists are not treated as sequential objects, whose elements are individually accessible by index, but are recursively defined. The terminal case for lists being the empty list. This leads to a list of length three being represented as a list, applied to a list, applied to a list, applied to an empty list.

Hence, list manipulations are carried out recursively by pattern matching for the top element of the current list, along with the rest of the list. As the DSL is inspired by Erlang's function definition format, pattern matching has thus far been an implicit feature of the DSL.

To enable such list manipulations we therefore define a special identifier format. This format allows the programmer to give two standard identifiers – separated by the conventional bar (|) – which then assigns the top element of the list to the first, and the rest of the list to the second. The latter maintains its representation as a sub-tree until evaluated, in accordance with how `SymbolTable` variable values are stored.

This gives basic list manipulation functionality to the DSL. We insist upon the caveat of *basic* functionality, as whilst the grammatical definition of lists theoretically support infinite lists, these have not been tested and are possibly limited by Python's recursion limit. This functionality has not been explored further due to its relatively low priority.

Chapter 5

Testing

Testing allows a developer to have confidence in his program, and to demonstrate that his program works as intended. Testing also ensures that the program is more robust than were no testing undertaken. Hence, a number of tests were implemented during the development of the DSL.

As with development, testing was approached in two stages. First, the API; second, the language itself. In this chapter, we look at the testing undertaken for both.

5.1 Testing the API

As the API is relatively simplistic, object-oriented Python code, a number of unit tests were created. Our test suite contains a unit test for each function in the API.

We use Python's standard `unittest` module, placing our test suite in the module `robtestsuite`. It is divided into sections, i.e. classes, according to the modules tested.

As all tests require an emulator instance, we define the imaginatively named class `Test` for the generic `setUp()` and `tearDown()` functions. All successive classes, barring one, are then sub-classed to `Test`.

First, we test the `robemulator` module. Under the class `TestEmulator`, we test each and every function in the original `Emulator` class. Starting with connection tests, we check for both connection and disconnection under standard situations, as well as were these actions to be attempted more than once.

Also tested here are the `send()` and `parse_result()` functions. To test these, we create a number of exemplar string inputs and responses, checking what is returned.

Next, the `robcmds` module is tested. Once again, we test each class and function therein. We note that the test case, `TestCommands` is the exception, where it is not a subclass of `Test`. It is the exception because it does not require the emulator to work.

The rest of the test cases are relatively uninteresting, as they simply test for most inputs for each function, in each command class. One interesting test case is defined under the `test_poll_ir_reflectometer` function. According to the emulator this test should succeed, yet it fails. Upon investigation using different methods – such as connecting to the emulator directly – we find this is an error in the emulator itself. These tests are skipped.

Snippet 5.1: Added Functionality to Allow Unit Testing of `astlex.py` module

```
if __name__ == '__main__':
    lex.runmain()
```

The test suite does not contain any tests pertaining to the `robstrings` module, as this contains string variables alone. All tests, barring those skipped, succeed.

5.2 Testing the Language

In this section, we describe the method used to test the DSL. Unlike with the API in the previous section, and as the PLY module is used, we are unable to simply test each and every function used to create the DSL's compiler.

Firstly, we explore the method used to test the lexical analyser as defined by the module `astlex`. We add Snippet 5.1 to allow us to generate the lexical analyser, pass it input, and receive a printed list of tokens in return.

A number of text files are created to be passed as input. These were created and expanded according to the size and content of the token list.

This methodology is continued when testing the syntactic analyser and interpreter. For each feature of the DSL, we create a text file containing a program that uses the feature. These files were created and expanded at each iteration where a feature was added.

This method proved most effective when implementing recursion. Given a simple recursive function, our initial tests failed, as our immutability of assignment was too strict. Once this was changed to allow multiple copies of the same parameters, another problem presented itself.

As mentioned in Section 4.3.4, Python limits recursion to avoid stack overflows. By default, this set to 1,000. When testing recursion, we noticed that, whilst shallow recursion allowed the given program to run and finish without fault, functions that required deeper recursion caused the execution to halt without either warning or error.

The language test suite was also required to test the built-in functions designed to communicate with the target emulator via the API. For these functions, as we have created a separate test suite for the API, we assumed said API to be correct.

Chapter 6

Evaluation and Conclusions

In this, our final chapter, we look back at the work completed in relation to the project goals laid out at the very beginning. We do so by splitting the exploration of our results into several sections.

First, for context, a summary is given of what has been achieved during the project's run. Secondly, we evaluate this work with respect to the project aims and goals. Thirdly, we suggest possible future work were the project to continue; and lastly, we give our final thoughts on the project as a whole.

6.1 A Brief Summary

Throughout the project's duration, we have developed a domain specific language that follows the principles of the functional style to program small robots. The language connects to the target robot's emulator and is capable of controlling that emulator's basic functions.

The development has included the creation of an interface, written in Python, that is capable of connecting, and transmitting commands to the target emulator, as well as receiving responses from said emulator. It has also included: the definition of a lexical token list; the definition of a context-free grammar; and the creation of an interpreter to execute a given program written in the language defined by the aforementioned context-free grammar.

Whilst the domain specific language is not purely functional, it has been designed such that it is extensible. The language is dynamically typed.

The domain specific language allows the assignment of variables under the principle of single assignment. It allows the definition of functions, and the ability to send basic commands to the target emulator. Both recursion and lists, including simple, functional list manipulations, are possible under the language. Simple binary operations are also supported.

6.2 Evaluation

In this section we consider how well our results have met our original goals. In Chapter 1 we outlined three goals; these follow.

1. To consider how the functional programming paradigm might be applied to the programming of small robots.
2. To build a prototypical platform using functional programming to solve the problem.
3. To evaluate the effectiveness of the functional paradigm when applied to the problem.

The third goal shall be addressed in this section and chapter.

6.2.1 First Goal

In Chapter 2 we considered why the programming of small robots as an exercise should be desirable. This lead to the consideration of possible target robots, and the discovery of University College London's development and use of small robots in teaching.

Under the purview of their use, we considered the background and benefits that functional languages provide. Highlighting their use in business and industry, beyond the confines of academia and intellectual curiosity. We then explored how they might be applied to the situation, and what benefits they might bring with their application.

Thus, we might conclude that:

1. functional languages may be applied to the programming of small robots;
2. that they are beneficial in educational contexts; and
3. that our first goal has been achieved.

6.2.2 Second Goal

Following this, and over the course of Chapters 3 and 4, we explored how a prototypical platform might be built. This included the development of a domain specific language – whose details are summarised above – that follows the functional style and allows programmers and students to program for the target emulator of the UCL robots.

As a complete solution, one might successfully argue the developed language insufficient to solve the defined problem. However, the second goal suggests a more exploratory approach – requiring only a *prototypical* solution – that illustrates that a solution *does* exist. Thus, one might equally argue that this goal has also been fulfilled.

It is, in part, the discussion of this second goal that we focus on to complete our final goal. To explore the effectiveness of our developed language, and by extension the application of the functional paradigm to the the problem, we ask a series of questions. First of which is whether the language could be used.

Usability

To answer this, we consider the functionality of the language as it presently stands. We note that the language supports the minimum set of commands that allow the manipulation of the target robots when using the emulator. Hence, we might conclude the language capable of being used.

However, we also note that the language as it stands is incomplete in comparison to its C counterpart. Indeed, and whilst a domain specific language, it is heavily limited in expressiveness.

Concerning Implementation

Aside from concerns regarding expressiveness, we might also look to flaws in its implementation. Thus, we ask our second, and third question: was Python a suitable language for the task at hand, and indeed in hindsight, was the overall approach the best method?

Regarding the choice of Python, and in light of continued development of the robots, it arguably remains a good language to use. Within reason, it ensures compatibility with the target robots and emulator. However, as noted before, Python is limited in its recursion depth – for a functional language, this is problematic.

Furthermore, Python is an imperative language. Meaning that any functional language built using it requires a greater 'translation' effort than building another imperative language. Naturally, this will require more time for each adjustment or expansion.

Therefore, we might suggest using a functional language to build the DSL. This appears possible in Haskell. [9] This decision forces yet more questions.

First we must reconsider the role of the API. Should this be developed with the intent to make it as generic as possible, or should it be incorporated in to the language? If the latter option is chosen, we must recreate the means to access both target robots and emulator in each language we wish to build for the problem. Furthermore, we must consider that each of these must be changed if and when the robot interface changes.

Conversely, if the API is kept, we might use or develop modules that translate sufficient instructions from the given program to Python. This allows any and all languages to make use of the API, and any changes need only be made once.

We might, therefore, conclude that the decision to divide duties between an API and a language is a good one. Hence, we shall continue to assume its existence.

It follows that we next reconsider the decision to create a DSL. Assuming the existence of such an API, we need not create a DSL, instead using one or more mature functional languages.

Again, using these languages would allow the programmer to make full use of the existing language. Whilst, unlike the developed DSL, a connection would need to be explicitly made, the programmer would gain practical experience in a real world functional language. This is arguably more beneficial than using a specially devised one for a specific purpose.

Might we, therefore, conclude the prototypical system, the second goal, the project itself an effective failure? Is the idea of building a functional language for education purposes as well as to program specific, small robots unsound?

Whilst an alternative method might seem desirable at this stage, the idea remains sound. Indeed, the developed DSL is a functional language, and does work. It provides a number of shortcuts in programming for those robots, such as making the connection to said robots automatically. With more time, effort, and perhaps a slightly different approach, the DSL created here would grow to become more functional, and a language truly usable in the undergraduate module.

This leads us to finally conclude that, whilst we cannot ignore the caveats expressed above, the developed DSL and API are successful under the terms of the second goal. They demonstrate it eminently possible to create a domain specific functional language for the programming of small robots.

6.3 Future Work

With the above evaluation in mind, it is obvious that there is much room for future work. In this section we consider a few possible routes from here, using the evaluation as a source of inspiration.

Firstly, the DSL and API must be updated to be compatible with the latest version of the emulator. A task not undertaken earlier due to time constraints and available materials.

Secondly, and with similar reasons to the previous point, the API must be tested on the robots themselves. As the robots now use a Raspberry Pi as their interface, it is possible that the API may run directly on the Raspberry Pi itself. Hence, a full exploration of how best to make the API interface with the robots is desirable.

Thirdly, we should like to suggest the continuation of improvements to the API. As explored in the evaluation above, the API is the most promising and versatile element of the developed system in considering its practical application.

Therefore, we suggest the API to be advanced in such a way that it is made more robust, and offers more options when interfacing. An example of this might be to allow different languages through the development of a layer that accepts translation modules.

Alongside the development of the API, the options for advancing the language itself should be explored. Perhaps a combination of the mentioned approaches should be taken. Indeed, it is possible that a formal comparison of the methods might be made. Thus, providing evidence for which approach is most useful in the development of functional languages for programming small robots.

One final point of advancement, should the DSL be continued, is the addition of support for both higher-order functions and concurrency. The former is a notable and useful aspect of most functional languages. Where the latter will likely prove useful under the robotics context.

6.4 Final Thoughts

In this section, we offer a few final thoughts on the project in general. Throughout the duration of the project, and this academic year, there have been setbacks, there have been delays, but there have also been little epiphanies.

The project has renewed our interest and curiosity in functional languages and the paradigm itself. Indeed, the differences between language styles have been a source of fascination throughout the project's run. Akin to translating between natural languages, albeit in a more controlled setting.

In closing, we suggest the project to be a moderate success. With more work necessary to fully realise the potential of functional programming in the domain of robotics programming, we believe this project has served to highlight the possibility and benefits of their use.

Chapter 7

Bibliography

[1] *A Gentle Introduction to Haskell*. URL: http://www.haskell.org/tutorial/functions.html.

[2] Ericsson AB. *Erlang/OTP System Documentation*. Section 5.7.3. Feb. 25, 2013.

[3] *About the ANTLR Parser Generator*. URL: http://www.antlr.org/about.html.

[4] *Android App for UCL Robots*. Android Application used to run students' code on UCL's robots. Unpublished. Received from Dr Mohammedally of UCL. 2012.

[5] Kenneth Barclay and John Savage. *Groovy Programming*. San Francisco, California, United States of America: Morgan Kaufmann Publishers, 2007, p. 476. ISBN: 9780123725073.

[6] Georg Bauer. *lazypy 0.5 : Python Package Index*. Sept. 27, 2011.

[7] David M. Beazley. *PLY (Python Lex-Yacc)*. URL: http://www.dabeaz.com/ply/.

[8] David M. Beazley. *PLY (Python Lex-Yacc)*. URL: http://www.dabeaz.com/ply/ply.html.

[9] Bryan Bell. *Haskell Compiler Series*. URL: http://bloggingmath.wordpress.com/haskell-compiler-series/.

[10] Christopher Brown, HW Loidl, and Kevin Hammond. "ParaForming: forming parallel haskell programs using novel refactoring techniques". In: *Twelth Symposium on Trends in Functional Programming*. Ed. by Ricardo Pena. Brown12ParaForming, 2012.

[11] Christopher Brown et al. "Cost-Directed Refactoring for Parallel Erlang Programs". In: *Proc. International Symposium on High-level Parallel Programming and Applications*. Springer, 2013, pp. 1–15.

[12] TP Carlson. "Robotlang-A Concurrent Functional Programming Language For Robots". In: *Integers* 1 (), p. 42.

[13] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941, p. 77.

[14] Christopher Clack, Colin Myers, and Ellen Poon. *Programming with Miranda*. 2011, p. 315.

[15] *COMP1010 Lab Tasks 3*. 2012.

[16] *COMP1010 Lab Tasks 4*. 2012.

[17] *COMP1010 Lab Tasks Week 1-2*. 2012.

[18] *COMP1010 Lab Tasks Week 3*. 2012.

[19] *Compilers – COMP2010*. Slides and examples used as reference; from Dr Capra's resources as provided in second year. URL: https://moodle.ucl.ac.uk/enrol/index.php?id=10783.

[20] *Computer Science Undergraduate Final Year Individual Projects – Project Information and Guidelines*. URL: https://moodle.ucl.ac.uk/course/view.php?id=11525&topic=1.

[21] *CTAN: Comprehensive TEXArchive Network*. URL: http://www.ctan.org.

[22] *CUP User's Manual*. URL: http://www2.cs.tum.edu/projects/cup/manual.html.

[23] *Erlang – Functions*. URL: http://www.erlang.org/doc/reference_manual/functions.html.

[24] *Erlang Programming Language*. URL: http://www.erlang.org.

[25] *Functional Programming*. URL: http://www.cs.ucl.ac.uk/students/syllabus/undergrad/3011_functional_programming/.

[26] *Functional Programming (GC16 / 3011)*. 2012. URL: http://www0.cs.ucl.ac.uk/teaching/3C11/.

[27] *Functional Programming in the Real World*. URL: http://homepages.inf.ed.ac.uk/wadler/realworld/.

[28] Erich Gamma et al. *Design Patterns*. Ed. by Addison-Wesley Pub Co. Vol. 47. Addison Wesley Professional Computing Series February. Addison Wesley, 1995, pp. 1–429. ISBN: 0201633612.

[29] *GHC – HaskellWiki*. URL: http://www.haskell.org/haskellwiki/GHC.

[30] Stephen A. Goss. *delicious robots blog. A Simple Compiler, Part 1: Parsing With Python and Ply*. URL: http://blog.deliciousrobots.com/2010/3/24/a-simple-compiler-with-python-and-ply-step-1-parsing/.

[31] *Grooy – Compile-time Metaprogramming - AST Transformations*. URL: http://groovy.codehaus.org/Compile-time+Metaprogramming+-+AST+Transformations.

[32] Barry Warsaw Guido van Rossum. *PEP 8 – Style Guide for Python Code*. July 5, 2001. URL: http://www.python.org/dev/peps/pep-0008/.

[33] Kevin Hammond. "Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour : the Hume Approach". In: *In Central European Summer School on Functional Programming*. Springer-Verlag, 2005, pp. 4–15.

[34] Mark Handley. *1. About Robots*. 2012.

[35] Mark Handley. *1. About Robots*. 2013.

[36] Mark Handley. *2. Introducing the Simulator*. 2012.

[37] Mark Handley. *2. Introducing the Simulator*. 2013.

[38] Mark Handley. *4. Compiling Code for the Robots*. 2012.

[39] Paul Hudak. "Conception, evolution, and application of functional programming languages". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300.

[40] John Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32 (1984), pp. 98–107.

[41] Iba and Kevin Brennan. *A Guide to the Business Analysis (BABOK ® Guide)*. Section 6.1.5.2. IIBA, 2009, p. 272. ISBN: 9780981129211.

[42] *Introduction - HaskellWiki*. URL: http://www.haskell.org/haskellwiki/Introduction.

[43] *Java HotSpot Garbage Collection*. URL: http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html.

[44] *Java SE 6 Documentation*. URL: http://docs.oracle.com/javase/6/docs/.

[45] Tom Kenyon. "Computing in schools: teaching the next generation of computer scientists". In: *The Guardian* (Feb. 13, 2013).

[46] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[47] Alison Kershaw. "Michael Gove brands ICT curriculum 'a mess'". In: *The Independent* (Nov. 1, 2012).

[48] Gerwin Klein. *JFlex User's Manual*. URL: http://jflex.de/manual.html.

[49] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782.

[50] *Michael Gove speech at the BETT Show 2012*. URL: http://www.education.gov.uk/inthenews/speeches/a00201868/michael-gove-speech-at-the-bett-show-2012.

[51] *Miranda Homepage*. URL: http://miranda.org.uk.

[52] *more ANTLR – Java, and comparisons to PLY and PyParsing*. URL: http://www.dalkescientific.com/writings/diary/archive/2007/11/03/antlr_java.html.

[53] *National curriculum review launched*. URL: http://www.education.gov.uk/inthenews/inthenews/a0073149/national-curriculum-review-launched.

[54] John Peterson, Paul Hudak, and Conal Elliott. "Lambda in motion: Controlling robots with Haskell". In: *Practical Aspects of Declarative ...* (1998).

[55] *Principles of Programming*. URL: http://www.cs.ucl.ac.uk/students/syllabus/undergrad/1007_principles_of_programming/.

[56] *Project Exhibition – Arduino Playground*. URL: http://playground.arduino.cc/projects/arduinoUsers.

[57] *Python Documentation Index*. URL: http://www.python.org/doc/.

[58] *Raspberry Pi Projects – Raspberry Pi Forum*. URL: http://www.raspberrypi.org/phpBB3/viewforum.php?f=15.

[59] R. M Ritter, ed. *Oxford Style Manual*. Great Clarendon Street, Oxford, OX2 6DP: Oxford University Press, 2003, p. 1033. ISBN: 9780198605645.

[60] *Robotics Programming*. URL: http://www.cs.ucl.ac.uk/students/syllabus/undergrad/1010_robotics_programming/.

[61] *Robotics Programming – First Lecture*. 2012.

[62] *Robotics Programming – Moodle*. Login necessary, login as guest if not registered. URL: https://moodle.ucl.ac.uk/course/view.php?id=15171.

[63] Ulrik P Schultz et al. "A Functional Language for Programming Self-Reconfigurable Robots 1". In: *Trends in Functional Programming*. Nijmegen, The Netherlands, 2008, pp. 1–13.

[64] *The Java Language Specification*. Feb. 28, 2013. URL: http://docs.oracle.com/javase/specs/jls/se7/html/index.html.

[65] *Undergraduate Syllabus Index 2012-2013*. URL: http://www.cs.ucl.ac.uk/students/syllabus/ug/.

[66] *LaTeX– Wikibooks*. URL: http://en.wikibooks.org/wiki/LaTeX.

[67] *Writing Domain-Specific Langauges – Groovy – Codehaus*. URL: http://docs.codehaus.org/display/GROOVY/Writing+Domain-Specific+Languages.

Appendix A

System Manual

In this appendix, we detail each element of the developed system. We do this so that were the development of the system to be continued by a third party, said third party can read this appendix and continue with little confusion.

Firstly, we look at the developed system as a unit. We describe how the modules connect together, and so how to compile the system. Secondly, we explore the function of each module. Here we detail how one might start the expansion of that module.

The system requires both Python and the target robots' emulator. The latter also requires the former. We assume the developer has both a terminal window[1] open and is navigated to the directory containing the project files.

In addition to the terminal window, we assume the developer has a Finder window – or its Linux equivalent – also open to the same directory. Finally, we expect the developer to use the text editor with which he feels comfortable.

Within the project directory, one finds: two subdirectories, one makefile, one readme markdown file, and ten Python files. The first directory, `robo-sim`, is included to speed the running of the language. The contents of this directory allow the target emulator to be run, and are the product of Professor Handley's work. We have in no way changed these files.

As one might expect, the `testfiles` subdirectory contains the files used to test the language. These are text files whose contents are written in the DSL itself.

We now consider the loose files. The readme file may be ignored. The makefile contains five targets.

The first rule, `run`, simply compiles the python code. The second, `test`, runs the interpreter, passing the `defaulttest.txt` testfile as input. The fifth, `clean`, cleans the project directory. Finally, the third rule, `ctest`, runs `clean`, followed by `test`. Finally, the fourth runs the API unit tests. This may be modified and expanded in the normal manner for makefiles.

Regarding the python files, these might be split into two conceptual categories.

[1] Alternatively, a command prompt if you are running Windows. We note that this program has neither been run nor tested on a Windows machine. So if you do insist upon this route, we wish you the very best of luck.

Those concerned with the DSL have the filename prefix, 'ast'. Likewise, those concerned with the API have the file name prefix, 'rob'. We will look at each of these in detail in the following sections.

To run the compiler and interpreter, one must first have the emulator running. Instructions for this task may be found under the emulator's documentation. We do, however, advise that this be run in a separate tab or window of one's terminal, lest one receive unexpected output in said terminal window.

A file written in the DSL is also necessary, such as those found in the `testfiles` subdirectory. For the purposes of explanation, we shall call this file: `file.txt`. Once these elements are in place, one simply enters the following command:

```
$ python astlang.py file.txt
```

This results in the system compiling and executing `file.txt`. The results of which are printed to `stdout`. If any emulator commands were included in `file.txt`, the results of these will appear in the emulator window.

We now consider the individual modules.

A.1 robstrings

The intent behind the `robstrings` module is to store information required for the running of the emulator. It includes three classes in an an attempt to separate these strings into categories.

`EmulatorStrings` stands apart from the other two classes, and contains the most basic information required to connect to the target emulator. The default values for both `_location`, and `_port` are, respectively, `localhost` and 55443.

These defaults were chosen, as they are the current location and port where the emulator may be found. These may either be changed by editing `robstrings` itself, or by calling the necessary function. The latter method only changes the values for as long as the `EmulatorStrings` object persists.

The class also contains the strings that partially make up the emulator responses. These include the null response (`.`), the warning, (`W`), and error prefixes (`ERR`).

`RobotStrings` contains the strings required to send the most basic commands. These include: the different prefixes for the classes of command provided by the emulator; the uppercase character representations for indicating left, right, and centre operations; and finally, a shortcut for a string space character.

`SensorStrings` is a subclass of `RobotStrings`, and contains the additional sensor codes required when constructing the commands to request sensor readings.

A.2 robemulator

Two classes exist within `robemulator`:

- `EmulatorError`, and
- `Emulator`.

The names of which adequately describe the respective class' task.

The former being a subclass of Python's `Exception` class, and raised when an error response is received from the emulator. Where the latter acts as a representation of the emulator within the API.

A.3 robcmds

The final module under the purview of the API, `robcmds` contains four classes. The first, `Commands`, contains the function used to construct a command, and is a superclass to the other three. Said remaining three, contain the functions that represent all possible commands.

We suggest these represent *all possible* commands, as the structure allows one to send instructions to the emulator that are not represented by the current list of functions. This may be achieved via the use of the `construct` function directly, or by sending whichever string one might choose to create using the `send` function within one's currently held `robemulator` instance.

A.4 astlex

This module is laid out in the standard format expected by PLY. We first include a list of keywords, followed by a list of tokens. These tokens are then defined using regular expressions.

For relatively simple tokens, these are defined using assignment statements. Those that remain are defined using functions. Although the state functionality of PLY is not used, all tokens are defined in any and all states.

This is mainly a remnant of an experiment during implementation to define strings and comments, and thus may be removed. If states are again used, however, it is necessary to specify under which states the tokens are defined.

`astlex` may be run either as a stand alone module, or imported and used in PLY's syntactic analyser generator.

A.5 astparse

This module may be split conceptually in two. We start with the definition of the AST node classes. Akin to `astlex`, we finish with the standard layout for syntactic analyser generation under PLY.

The tokens, as defined in `astlex`, are imported first. We then define precedence and association rules. This is followed by the functions that define the grammar.

In each function we use a string to define the left-hand and right-hand side of the grammar rule. These are then followed by small snippets of code dedicated to the creation of the AST. Here we note that only the last statement in each function is returned.

At the very end of the module, we define the parser, and subsequently a function that uses the parser to process any data given it. If at any point during this process an error occurs, we print the error and halt the execution.

A.6 astsymtab

A simple module, containing a sole class. This class is a custom implementation of Python's dictionary data type, created to avoid any depth restrictions that might prematurely halt the execution of a given program during recursion.

It offers the ability to add objects via a `put` function; the ability to get a stored value from a key; and return a list of the keys contained within that instance.

A.7 astvisitor

This module contains the definition of both the `SymbolTable`, and `FunctionTable` classes. Both of which are used during the execution of the given program.

The former is used to store identifiers and their values, and the latter is used to store function information. Both have a number of functions defined to aid their use, such as the ability to store and retrieve information.

Next, we define the basic `Visitor` class and `ExecutionError` class. The latter is similar to the aforementioned `EmulatorError` class, and is raised whenever an error occurs during execution. The basic Visitor class is designed such that it defines the basic shape of any visitors subsequently defined. In this respect, we are following the Visitor design pattern.

Lastly, we define the `ExecutionVisitor` class. This, as its name might imply, is used to actually execute the given program. It is a subclass of the aforementioned `Visitor` class.

Within it, we first set Python's recursion limit to one milliard (1,000,000,000). We also define a function to allow us to flatten lists, as Python lacks this functionality.

Following this, we define the `visit` function. Within this function, each node of the AST is tested for, and a means of execution defined. This follows the general pattern of expanding each node until the bottom-most node is reached. At which point, we return upwards through the tree, taking each step as is necessary.

If a newly defined grammar rule requires the definiton of a new type of AST node, a statement that checks for this type must also be added.

A.8 astinterp

The `astinterp` module is used to begin the execution of the given program and to handle any execption errors. It is in this module that we start any and all visitors on the input. If the functionality of the interpreter requires more than one visitor, this module shall be used to coordinate these visitors and the data held between.

A.9 astlang

The final module in our list, is that of the module we call to run the compiler and interpreter. It checks the number of arguments passed it; it loads `file.txt` into memory; and subsequently passes that file to the lexical analyser, then to the syntactic analyser as a stream of tokens, and finally as an AST to the interpreter.

Barring significant refactoring, or the additon of an interactive environment, this module should remain relatively unchanged.

Appendix B

User Manual

We have created a domain specific language designed to program small robots. Hence, in this appendix we describe the syntax of the language, and how one might create a simple program through its use.

We first describe what is necessary to run the program. We then explore the syntax of the language, building our exemplar program as we proceed.

B.1 Pre-conditions

As the language is intended to eventually be used to educate first year computer science undergraduate students, we expect you to know how to use the terminal. More specifically, we expect you to know how to run Python code. Naturally, it follows that we shall assume you have Python installed on your computer.

We also require you to have the emulator, be able to run it, and have an understanding of what the robots are capable of and how to get them to do it using the ASCII Synchronous interface.

Finally, we expect you to have the `robotlang` directory on your (UNIX based) computer.

Now we have this set up, we can start.

B.2 Building a Simple Program

Before we begin, we must first navigate our terminal's present working directory to that of the aforementioned `robotlang`. We must also have both the emulator and a text editor of your choice open.

B.2.1 Hello World

As is traditional in computer science let us begin with the most simple example. We shall create a program that prints 'Hello World' to the console. In a new file in your text editor we type the following:

Snippet B.1: Hello World

```
print "Hello World"
```

Once saved under a name of your choice, at a location similarly of your choice, we run this program by using the following code in our terminal window. For the purposes of this manual, we shall use `file.txt` as our exemplar program's filename.

```
$ python astlang.py file.txt
```

This will print the words, 'Hello World', in your terminal window.

We can print other data types also. Currently supported data types are strings, integers, boolean values, and lists. Expanding Snippet B.1 to the following:

Snippet B.2: Extended Hello World

```
print "Hello World"
print 42
print True
```

Produces the output:

```
Hello World
42
True
```

Simple binary operations such as plus, minus, times, divide, greater than, and, or, and not, are also possible. We might print the results of some simple arithmetic operations thus:

Snippet B.3: Binary Operations

```
print 40 + 2
print 5 * 2
print True or False
```

Produces the following:

```
42
10
True
```

B.3 Assignment

Currently we have three print statements. These are presently printing three raw values or arithmetic operations. Let us now introduce assignment statements.

An assignment statement consists of both a value and an identifier. Under the DSL we define an identifier as any word that begins with a letter or underscore, followed by more alphanumeric characters and underscores.

Hence, we might assign the variable `ident` to the integer value 10. We print it to show that it works:

Snippet B.4: Introducing Assignment

```
0 def ident = 10
1 print ident
```

Producing:

```
0 10
```

Lists may be assigned in two ways. They may be assigned in the way we have just assigned 10 to `ident`, or they may be assigned using pattern matching. We may get the first element from a list, for example, by using the previous method:

Snippet B.5: Assigning Lists

```
0 def [First | Last] = [10, 20, 42]
1 print First
```

Which produces:

```
0 10
```

B.3.1 Functions

Thus far, all of our programs have been lists of statements. Being a functional language, we should probably introduce functions.

Under the DSL, functions are relatively simple. All functions are defined before any global statements occur. Containing lists of statements, functions are only executed when called.

In the below Snippet we introduce a simple function that takes a single parameter and prints it:

Snippet B.6: Functions

```
def foo(def ident)
    print ident
end

foo(10)
```

Which, as one might expect, produces:

```
10
```

In lieu of while- and for-statements we rely upon recursion. We might recursively define and call a function in the following manner. To do this, we also introduce if-statements that work as you might imagine:

Snippet B.7: Recursive Functions

```
def foo(def a, def b)
    if a < 5:
        print b
    else:
        print False
        foo(a - 1, b)
    end
end

foo(10, True)
```

This produces the slightly more exciting:

```
False
False
False
False
False
False
True
```

B.3.2 Controlling the Robot

Yet, whilst the DSL is designed to control robots, this functionality is conspicuously absent. Hence, we will take a look at the three classes of commands that we might choose to send to the emulator.

Movement

First we introduce movement. We do this using the `mv` function, which takes two parameters. These parameters correspond to the voltage one wishes to apply to the respective motor. Hence, we might write the following:

Snippet B.8: Movement

```
def foo(def a, def b)
    if a < 0:
        print b
    else:
        mv(50, 50)
        foo(a - 1, b)
    end
end

foo(10, "Done")
```

This makes the robot in the emulator move forward slightly, and we receive the following in the terminal:

```
Done
```

Infrared Servos

Next we add the ability to adjust the infrared servos located on the top of the robot. This is achieved using the `ir` command. Akin to `mv` mentioned above, it takes two parameters. This time the left and right parameters refer to the angle at which the servos are set. We give an example below:

Snippet B.9: Infrared

```
def foo(def a, def b)
    if a < 0:
        print b
    else:
        mv(50, 50)
        foo(a - 1, b)
    end
end

ir(45, -45)
foo(10, "Done")
```

Building upon the previous example, we not only move the robot forward, but we first adjust the infrared servos. The should now be parallel with the front of the robot.

Sensor

For our final component, we add to our working example the ability to request values from the robot's sensors. This is achieved via the `sn` function.

Unlike the above two commands, this takes four parameters. The first parameter is the base string of the sensor whose value you are requesting. The next three parameters take boolean values and depend upon the options available for that function.

In the below example we request the ultrasound value after each movement.

Snippet B.10: Sensors

```
def foo(def a, def b)
    if a < 0:
        print b
    else:
        mv(50, 50)
        sn("US", False, False, False)
        foo(a - 1, b)
    end
end

ir(45, -45)
foo(10, "Done")
```

In addition to setting the infrared rangefinders parallel with the front of the robot, and moving the robot forward, we receive multiple responses from the ultrasound sensors. From the default starting position, this prints:

```
('US', '83')
('US', '82')
('US', '81')
('US', '79')
('US', '78')
('US', '77')
('US', '76')
('US', '74')
('US', '73')
('US', '72')
('US', '70')
Done
```

Hence, we now have a simple program written in the DSL that controls the robot within the emulator.

Appendix C

List of Lexical Tokens

Below is the list of lexical tokens with their definitions, as defined in the lexical analyser class `astlex.py`.

Token	Definition (Python Regular Expression Format)
`EQUALS`	`=`
`PLUS`	`\+`
`MINUS`	`-`
`TIMES`	`\*`
`DIVIDE`	`/`
`MOD`	`%`
`EQEQ`	`==`
`GT`	`>`
`GTE`	`>=`
`LT`	`<`
`LTE`	`<=`
`LPAREN`	`\(`
`RPAREN`	`\)`
`LBRACK`	`\[`
`RBRACK`	`\]`
`COLON`	`\:`
`COMMA`	`\,`
`BAR`	`|`
`INTEGER`	`\d+`

Continued overleaf.

Token ctd.	Definition (Python Regular Expression Format) ctd.
STRING	\"([^\\\n]|(\\.))*?\"
CHARSTR	\'([^\\\n]|(\\.))*?\'
ID	[a-zA-Z][a-zA-Z0-9_]*
DEF	def
END	end
PRINT	print
IF	if
ELSE	else
MOVE	mv
IR	ir
SENSOR	sn
NULL	null
TRUE	True
FALSE	False
AND	and
NOT	not
OR	or

Appendix D

The Context-free Grammar for the Language

Below we list the context-free grammar that defines the developed language, and used in `astparse` to generate the syntactic analyser. The grammar uses the tokens listed in Appendix C, and is given in Backus–Naur Form.

⟨*program*⟩ ::= ⟨*functionlist*⟩ ⟨*statementlist*⟩
| ⟨*statementlist*⟩
|

⟨*functionlist*⟩ ::= ⟨*functionlist*⟩ ⟨*function*⟩
| ⟨*function*⟩

⟨*function*⟩ ::= '`def`' ⟨*ID*⟩ ⟨*paramlist*⟩ ⟨*body*⟩
| '`def`' ⟨*ID*⟩ ⟨*paramlist*⟩ '`end`'

⟨*paramlist*⟩ ::= '`(`' ⟨*shortdeclaration*⟩ ⟨*restparams*⟩
| '`(`' ⟨*shortdeclaration*⟩ '`)`'
| '`(`' '`)`'

⟨*restparams*⟩ ::= '`,`' ⟨*shortdeclaration*⟩ ⟨*restparams*⟩
| '`,`' ⟨*shortdeclaration*⟩ '`)`'

⟨*shortdeclaration*⟩ ::= '`def`' ⟨*ID*⟩

⟨*body*⟩ ::= ⟨*statement*⟩ ⟨*body*⟩
| ⟨*statement*⟩ '`end`'

⟨*statementlist*⟩ ::= ⟨*statement*⟩ ⟨*statementlist*⟩
| ⟨*statement*⟩ '`end`'

⟨*statement*⟩ ::= 'def' ⟨*ID*⟩ '=' ⟨*expression*⟩
| 'def' '[' ⟨*ID*⟩ '|' ⟨*ID*⟩ ']' '=' ⟨*expression*⟩
| 'if' ⟨*expression*⟩ ':' ⟨*statementlist*⟩ 'end'
| 'if' ⟨*expression*⟩ ':' ⟨*statementlist*⟩
'else' ':' ⟨*statementlist*⟩ 'end'
| 'mv' '(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'
| 'ir' '(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'
| 'sn' '(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ','
⟨*expression*⟩ ',' ⟨*expression*⟩ ')'
| 'print' ⟨*expression*⟩
| ⟨*expression*⟩

⟨*params*⟩ ::= ⟨*expression*⟩ ⟨*restparamexprs*⟩
| ⟨*expression*⟩ ')'
| ')'

⟨*restparamexprs*⟩ ::= ',' ⟨*expression*⟩ ⟨*restparamexprs*⟩
| ',' ⟨*expression*⟩ ')'

⟨*expression*⟩ ::= ⟨*ID*⟩
| ⟨*ID*⟩ '(' ⟨*params*⟩
| '[' ⟨*expression*⟩ ⟨*restlist*⟩
| '[' ⟨*expression*⟩ ']'
| '[' ']'
| ⟨*expression*⟩ '+' ⟨*expression*⟩
| ⟨*expression*⟩ '-' ⟨*expression*⟩
| ⟨*expression*⟩ '*' ⟨*expression*⟩
| ⟨*expression*⟩ '/' ⟨*expression*⟩
| ⟨*expression*⟩ '%' ⟨*expression*⟩
| ⟨*expression*⟩ '==' ⟨*expression*⟩
| ⟨*expression*⟩ '>' ⟨*expression*⟩
| ⟨*expression*⟩ '>=' ⟨*expression*⟩
| ⟨*expression*⟩ '<' ⟨*expression*⟩
| ⟨*expression*⟩ '<=' ⟨*expression*⟩
| ⟨*expression*⟩ 'and' ⟨*expression*⟩
| ⟨*expression*⟩ 'or' ⟨*expression*⟩
| '-' ⟨*expression*⟩
| 'not' ⟨*expression*⟩
| '(' ⟨*expression*⟩ ')'
| ⟨*term*⟩

⟨*restlist*⟩ ::= ',' ⟨*expression*⟩ ⟨*restlist*⟩
| ',' ⟨*expression*⟩ ']'

⟨*term*⟩ ::= ⟨*STRING*⟩
| ⟨*CHARSTR*⟩
| ⟨*INTEGER*⟩
| ‘`True`’
| ‘`False`’
| ‘`null`’

Where possible Backus-Naur Form has been adhered to, yet we note that four non-terminals are not expanded here as their expansion does not help to clarify the grammar. In these instances – i.e. ⟨*ID*⟩, ⟨*STRING*⟩, ⟨*CHARSTR*⟩, and ⟨*INTEGER*⟩ – the non-terminals correspond to their token declarations given in Appendix C.

We also note that this grammar is not conflict free. Details of these conflicts, including how they are resolved under the context of the DSL may be found in Subsection 4.3.2.

Appendix E

Project Plan

Please find overleaf an unadulterated reproduction of the Project Plan, as submitted on Wednesday 14th November, 2012.

Project Plan

Adam D. Barwell

November 2012

Project Title: *A Functional Language for Programming Small Robots*
Supervisor: *Dr Graham Roberts*

1 Aims and Objectives

The aims and objectives for this project are as follows.

Aim: To build and develop a system to pass instructions to small robots devised for, and used by, students taking the COMP1010 module.
Objectives: 1. Review the current functionality of the robots, how that functionality is expressed, and the types of tasks required of the robots. 2. Create a robust means of connecting to the emulator and passing commands to it. 3. Implement the desired and required functionality, with suitable abstraction, and tailored for the common tasks found in the previous part. 4. Test the effectiveness of the system through a series of exemplary tasks that, in part, reflect the expected usage of the system by students and other end users.

Aim: The construction of an interface layer for the aforementioned command system; the layer will allow different programming languages to use the devised system in a similar manner.
Objectives: 1. Look again at the functionality of the command system, and consider how it can best be used by connecting languages. 2. To prototype and establish a means of communication between the two layers, command system and higher language; an intermediary language, for example. 3. Implementation and testing of the method.

Aim: To define a functional language and implement a compiler for that language. Intended to be used both for programming the robots in earnest, and as an illustration of what can be done with the underlying system.
Objectives: 1. Confirm the syntax and grammar of the domain specific functional language to be built. 2. Consider the output format required as specified in the interface layer. 3. Write the grammar and syntax definitions using JFlex and CUP as tools to facilitate the implementation of the compiler, testing the language as new features are added incrementally.

Aim: To design and conduct a user survey to establish the usability and successfulness of the new language.

Objectives: 1. Review the functionality of the language and systems created, in conjunction with the original requirements. 2. Devise a set of questions and/or tasks to be proposed to survey participants that will suitably assess the functional language and system. 3. Carry out the survey. 4. Assess and formalise the results of the survey, to make them useful when considering the successfulness of the functional language.

2 Deliverables

The project aims to produce the following deliverables:

- A Project Plan outlining the aims of the proposed project. (This document.)
- The Interim Report; a document to be presented no later than midday on Wednesday, 23 January 2013, detailing the work completed up to this point. It should also highlight any changes made to the original aims, and the work needed to complete the project.
- A fully documented, modular API that can interpret instructions from a number of higher level languages, and pass those instructions to small robots, as built for the COMP1010 module by the UCL Computer Science department.
- A feature rich domain specific functional language, designed in part to facilitate the teaching of programming, interfacing with the aforementioned API. It will be fully documented.
- Analysed and suitably presented results of the survey assessing the system's effectiveness in both teaching and general use.
- The Final Report, a bound document to be presented no later than midday on Friday, 26 April 2013. It will document the methodology and results of this project.

3 Work Plan

As the majority of this project is concerned with development, an iterative approach was considered most appropriate. Hence, the following Work Plan is proposed:

Project start to end October (4 Weeks)
Literature search and review.
Consideration of specific project aims and requirements.

Mid-October to mid-November (4 Weeks)
Refinement of project aims and requirements.
Minor prototyping of possible solutions to assist with decisions made.

End October to 14 November (2 Weeks)
Drafting and submitting the Project Plan.

Mid-November to end November (2 Weeks)
Initial iteration for both API and language.

December to mid-January (6 Weeks)
Work through iterations.
Add functionality to both the API and language in order of usefulness and import. Higher level functionality, such as searching and mapping, should be left until the later weeks.

January to 23 January (3 Weeks)
Compile the Interim Report.

Mid-January to mid-February (4 Weeks)
Devise and undertake user survey.
Appropriate adjustments to the system should be made according to these results.

Mid-February to March (6 Weeks)
Work on Final Report.

4 Signatures

Supervisor's Signature:

Appendix F

Interim Report

Please find overleaf an unadulterated reproduction of the Interim Report, as submitted on Wednesday 23rd January, 2013.

MEng Individual Project

Interim Report

Adam D. Barwell

Supervisor
Dr Graham Roberts

Wednesday, 23 January 2013

Project Title: *A Functional Language for Programming Small Robots*

1 Current Progress

At time of writing, both the proposed communications API, i.e. the layer that passes set commands to the robots or their emulator, and the functional language that sits atop it are being developed.

The target robots have been studied; access to the robots and their emulator for testing, and the means of communicating with them, have both been acquired.

Initially, the communications API and the rudimentary language had been written in Java. However, due to recent developments in the robots' design, and as a result of discussions with their developer, it was decided that a system written in the Python language would better facilitate the usefulness of the project's resulting deliverables. Hence, the communications API has been rewritten to the same standard using the Python language. Following this, the core compiler for the functional language has also been rewritten using Python, and is currently being improved upon.

In creating the functional language, a model was first constructed using Erlang. Erlang is a functional language that supports multiple threads, making it a useful language with which to model the proposed system, and a useful source for inspiration.

The decision was made to create a simple compiler that would serve the needs of the domain specific language being developed. A core library of commands would then be included with the code, written by the user, when compiled and run. This was deemed a better approach over writing the library into the compiler itself, as its increased modularity makes it easier to test and modify for future extensibility.

As JFlex and CUP are predominantly Java tools, the Python Lex-Yacc module is being used in their place. The compiler is being written to accommodate both files passed as arguments, and a shell mode, where the user may input commands from his terminal window.

2 Remaining Work

The remaining work required for the project first includes the completion of the compiler. The creation of the core library should follow. Subsequent developments in the core library may require adjustments to the compiler, hence this should continue to be done iteratively with any such adjustments made where appropriate.

The basic compiler, core library, and communications API should be finished by the start of the Reading Week of the Spring Term at the latest.

Working towards a usable and professional system, with a suitable user interface being a significant concern, the language shall then be iteratively improved. Suitable features shall be added in order of their relevance and priority. Such changes shall also likely require the communications API to be developed at a similar rate.

A small user study should be devised and conducted during the latter half of the Spring Term. Work on the Main Report should begin during this time.

The user study should be completed before the end of the Spring Term. The main focus, however, should be the Main Report, which must be at least be partially completed by term's end.

3 Signatures

Supervisor's signature:

Appendix G

Source Code

Over the following pages we list the project's source code. Whilst originally intending to list the entire source code of the project, we find the end result excessive.

Hence, we have removed certain parts. First we do not list the `robstrings` module, as that adds little in terms of information. We also do not include `robtestsuite` for similar reasons. For the rest of the modules, we offer a condensed version. Where whitespace is reduced, and comments and documentation strings have been removed. Finally, we note that we have removed the AST node definitons from the `astparse`. Again, and whilst necessary, these add little more than pages to the code.

G.1 robemulator

```
import socket
import re
from robstrings import EmulatorStrings

class EmulatorError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)

class Emulator(object):
    def __init__(self):
        super(Emulator, self).__init__()
        self.sock = socket.socket()
        self.strings = EmulatorStrings()

    def connect(self):
        self.sock.connect((self.strings._location, self.strings._port))

    def send(self, msg):
```

```
        self.sock.send(msg)
        result = self.sock.recv(1080)
        return self.parse_result(result)

    def close_connection(self):
        self.sock.close()

    def parse_result(self, result):
        if result == ".\n":
            return None
        elif re.match(r'^S ', result):
            return tuple(re.sub(r'^S ', '', result).split())
        elif re.match(r'^W ', result):
            return tuple(result.split())
        elif re.match(r'^ERR.', result):
            raise EmulatorError(re.sub(r'^ERR ', '', result))
        else:
            return result
```

G.2 robcmds

```
from robstrings import RobotStrings, SensorStrings

class Commands(object):
    _strings = RobotStrings()

    def construct(self, prefix, left, right):
        _cmd = prefix
        if left is not None:
            if right is not None:
                _cmd = (_cmd + self._strings._left + self._strings._right
                       + self._strings._space + left +
                           self._strings._space + right)
            else:
                _cmd = _cmd + self._strings._left + self._strings._space +
                    left
        else:
            _cmd = _cmd + self._strings._right + self._strings._space +
                 right

        return _cmd

class MotorCommands(Commands):
    def __init__(self, em):
        super(MotorCommands, self).__init__()
        self._emulator = em

```

```
23      def set_speeds(self, left, right):
24          if not ((left is None) and (right is None)):
25              _cmd = self.construct(self._strings._motor, left, right)
26              return self._emulator.send(_cmd)
27          else:
28              return None
29
30      def halt(self):
31          return self.set_speeds("0", "0")
32
33      def set_left_motor_speed(self, voltage):
34          return self.set_speeds(voltage, None)
35
36      def set_right_motor_speed(self, voltage):
37          return self.set_speeds(None, voltage)
38
39  class IRServoCommands(Commands):
40      def __init__(self, em):
41          super(IRServoCommands, self).__init__()
42          self._emulator = em
43
44      def set_ir_servo(self, left, right):
45          if not ((left is None) and (right is None)):
46              _cmd = self.construct(self._strings._ir, left, right)
47              return self._emulator.send(_cmd)
48          else:
49              return None
50
51      def set_left_ir_servo(self, angle):
52          return self.set_ir_servo(angle, None)
53
54      def set_right_ir_servo(self, angle):
55          return self.set_ir_servo(None, angle)
56
57
58  class SensorCommands(Commands):
59      def __init__(self, em):
60          super(SensorCommands, self).__init__()
61          self._emulator = em
62          self._sensorstrings = SensorStrings()
63
64      def construct(self, prefix, left, centre, right):
65          _cmd = self._strings._sensor + prefix
66
67          if left:
68              _cmd = _cmd + self._strings._left
69          if centre:
70              _cmd = _cmd + self._strings._centre
71          if right:
```

```
            _cmd = _cmd + self._strings._right

        return _cmd

    def poll_front_ir_rangefinder(self, left, right):
        return self.submit_cmd(self._sensorstrings._if, left, right)

    def poll_side_ir_rangefinder(self, left, right):
        return self.submit_cmd(self._sensorstrings._is, left, right)

    def poll_front_ultrasound(self):
        return self.submit_cmd(self._sensorstrings._us)

    def poll_bumper(self, left, right):
        return self.submit_cmd(self._sensorstrings._bf, left, right)

    def poll_voltage(self):
        return self.submit_cmd(self._sensorstrings._vo)

    def poll_motor_encoder(self, left, right):
        return self.submit_cmd(self._sensorstrings._me, left, right)

    def poll_ir_reflectometer(self, left, centre, right):
        return self.submit_cmd(self._sensorstrings._ib, left=left,
            right=right, centre=centre)

    def submit_cmd(self, prefix, left=False, right=False, centre=False):
        _cmd = self.construct(prefix, left, centre, right)
        return self._emulator.send(_cmd)
```

G.3 astlex

```
from ply import *
import re

keywords = {
    'def': 'DEF',
    'end': 'END',
    'print': 'PRINT',
    'if': 'IF',
    'else': 'ELSE',
    'mv': 'MOVE',
    'ir': 'IR',
    'sn': 'SENSOR',
    'null': 'NULL',
    'True': 'TRUE',
    'False': 'FALSE',
```

```
    'and': 'AND',
    'not': 'NOT',
    'or': 'OR',
}

tokens = list(keywords.values()) + [
    'EQUALS',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'MOD',
    'EQEQ',
    'GT',
    'GTE',
    'LT',
    'LTE',
    #
    'LPAREN',
    'RPAREN',
    'LBRACK',
    'RBRACK',
    #
    'COLON',
    'COMMA',
    'BAR',
    #
    'INTEGER',
    'STRING',
    'CHARSTR',
    #
    'ID',
]

states = (
    ('func', 'exclusive'),
)

t_ANY_ignore = ' \t'

t_ANY_EQUALS = r'='
t_ANY_PLUS = r'\+'
t_ANY_MINUS = r'-'
t_ANY_TIMES = r'\*'
t_ANY_DIVIDE = r'/'
t_ANY_MOD = r'%'
t_ANY_EQEQ = r'=='
t_ANY_GT = r'>'
t_ANY_GTE = r'>='
```

```
t_ANY_LT = r'<'
t_ANY_LTE = r'<='

t_ANY_LPAREN = r'\('
t_ANY_RPAREN = r'\)'
t_ANY_LBRACK = r'\['
t_ANY_RBRACK = r'\]'

t_ANY_COLON = r'\:'
t_ANY_COMMA = r'\,'
t_ANY_BAR = r'\|'


def t_ANY_INTEGER(t):
    r'\d+'
    t.value = int(t.value)
    return t


def t_ANY_ID(t):
    r'[a-zA-Z][a-zA-Z0-9_]*'
    t.type = keywords.get(t.value, 'ID')
    return t


def t_ANY_STRING(t):
    r'\"([^\\\n]|(\\.))*?\"'
    t.value = re.sub(r'^\"|\"$', '', t.value)
    return t


def t_ANY_CHARSTR(t):
    r'\'([^\\\n]|(\\.))*?\''
    t.value = re.sub(r'^\'|\'$', '', t.value)
    return t


def t_ANY_newline(t):
    r'\n'
    t.lexer.lineno += 1


def t_ANY_comment(t):
    r'/\*(.|\n)*?\*/'
    t.lexer.lineno += t.value.count('\n')


def t_ANY_error(t):
    print("Illegal character %s" % t.value[0])
```

```
    t.lexer.skip(1)

lexer = lex.lex()

if __name__ == '__main__':
    lex.runmain()
```

G.4 astparse

```
from ply import *
import astlex
import sys

tokens = astlex.tokens

precedence = (
    ('nonassoc', 'GT', 'GTE', 'LT', 'LTE', 'EQEQ'),
    ('left', 'AND', 'OR'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('left', 'MOD'),
    ('right', 'UMINUS', 'NOT'),
)

def p_program(p):
    '''program : functionlist statementlist
               | statementlist
               | '''
    if len(p) == 1:
        p[0] = ProgramNode(None, None)
    elif len(p) == 2:
        p[0] = ProgramNode(None, p[1])
    else:
        p[0] = ProgramNode(p[1], p[2])

def p_functionlist(p):
    '''functionlist : functionlist function
                    | function '''
    if len(p) == 2:
        p[0] = ListNode(p[1], None)
    else:
        p[0] = ListNode(p[1], p[2])

def p_function(p):
    '''function : DEF ID paramlist body
                | DEF ID paramlist END'''
    if isinstance(p[4], str):
```

```
38          p[0] = FunctionNode(p[2], p[3], None)
39      else:
40          p[0] = FunctionNode(p[2], p[3], p[4])
41
42  def p_paramlist(p):
43      '''paramlist : LPAREN shortdeclaration restparams
44                   | LPAREN shortdeclaration RPAREN
45                   | LPAREN RPAREN '''
46      if len(p) == 3:
47          p[0] = None
48      else:
49          if isinstance(p[3], str):
50              p[0] = ParamListNode(p[2], None)
51          else:
52              p[0] = ParamListNode(p[2], p[3])
53
54  def p_restparams(p):
55      '''restparams : COMMA shortdeclaration restparams
56                    | COMMA shortdeclaration RPAREN '''
57      if isinstance(p[3], str):
58          p[0] = ParamListNode(p[2], None)
59      else:
60          p[0] = ParamListNode(p[2], p[3])
61
62  def p_shortdeclaration(p):
63      '''shortdeclaration : DEF ID '''
64      p[0] = ParamDeclarationStatement(p[1], p[2])
65
66  def p_body(p):
67      '''body : statement body
68              | statement END '''
69      if isinstance(p[2], str):
70          p[0] = ListNode(p[1], None)
71      else:
72          p[0] = ListNode(p[1], p[2])
73
74  def p_statementlist(p):
75      '''statementlist : statement statementlist
76                       | statement '''
77      if len(p) == 2:
78          p[0] = ListNode(p[1], None)
79      else:
80          p[0] = ListNode(p[1], p[2])
81
82  def p_statement_declaration(p):
83      '''statement : DEF ID EQUALS expression
84                   | DEF LBRACK ID BAR ID RBRACK EQUALS expression '''
85      if len(p) == 5:
86          p[0] = DeclarationStatement(p[2], p[4])
```

```
87      else:
88          p[0] = ListDeclarationStatement(p[3], p[5], p[8])
89
90  def p_statement_if(p):
91      '''statement : IF expression COLON statementlist END '''
92      p[0] = IfStatementNode(p[2], p[4], None)
93
94  def p_statement_if_else(p):
95      '''statement : IF expression COLON statementlist ELSE COLON
            statementlist END '''
96      p[0] = IfStatementNode(p[2], p[4], p[7])
97
98  def p_statement_cmd_mv(p):
99      '''statement : MOVE LPAREN expression COMMA expression RPAREN '''
100     p[0] = MotorCommandNode(p[3], p[5])
101
102 def p_statement_cmd_ir(p):
103     '''statement : IR LPAREN expression COMMA expression RPAREN '''
104     p[0] = ServoCommandNode(p[3], p[5])
105
106 def p_statement_cmd_sn(p):
107     '''statement : SENSOR LPAREN expression COMMA expression COMMA
            expression COMMA expression RPAREN '''
108     p[0] = SensorCommandNode(p[3], p[5], p[7], p[9])
109
110 def p_statement_print(p):
111     '''statement : PRINT expression '''
112     p[0] = PrintStatement(p[2])
113
114 def p_statement(p):
115     '''statement : expression '''
116     if len(p) == 2:
117         p[0] = StatementNode(p[1])
118
119 def p_expression_var(p):
120     '''expression : ID
121                   | ID LPAREN params '''
122     if len(p) == 2:
123         p[0] = IdentifierExpression(p[1])
124     else:
125         p[0] = FunctionIdentifierExpression(p[1], p[3])
126
127 def p_params(p):
128     '''params : expression restparamexprs
129               | expression RPAREN
130               | RPAREN '''
131     if len(p) == 2:
132         p[0] = ParamValListNode(None, None)
133     else:
```

```
134         if isinstance(p[2], str):
135             p[0] = ParamValListNode(p[1], None)
136         else:
137             p[0] = ParamValListNode(p[1], p[2])
138 
139 def p_restparamexprs(p):
140     '''restparamexprs : COMMA expression restparamexprs
141                       | COMMA expression RPAREN '''
142     if isinstance(p[3], str):
143         p[0] = ParamValListNode(p[2], None)
144     else:
145         p[0] = ParamValListNode(p[2], p[3])
146 
147 def p_expression_list(p):
148     '''expression : LBRACK expression restlist
149                   | LBRACK expression RBRACK
150                   | LBRACK RBRACK '''
151     if len(p) == 3:
152         p[0] = StdListNode(None, None)
153     else:
154         if isinstance(p[3], str):
155             p[0] = StdListNode(p[2], None)
156         else:
157             p[0] = StdListNode(p[2], p[3])
158 
159 def p_expression_list_rest(p):
160     '''restlist : COMMA expression restlist
161                 | COMMA expression RBRACK '''
162     if isinstance(p[3], str):
163         p[0] = StdListNode(p[2], None)
164     else:
165         p[0] = StdListNode(p[2], p[3])
166 
167 def p_expression_binop(p):
168     '''expression : expression PLUS expression
169                   | expression MINUS expression
170                   | expression TIMES expression
171                   | expression DIVIDE expression
172                   | expression MOD expression
173                   | expression EQEQ expression
174                   | expression GT expression
175                   | expression GTE expression
176                   | expression LT expression
177                   | expression LTE expression
178                   | MINUS expression %prec UMINUS '''
179     if len(p) == 3:
180         p[0] = BinOp(p[1], Integer(0), p[2])
181     else:
182         p[0] = BinOp(p[2], p[1], p[3])
```

```
183
184 def p_expression_logical_binop(p):
185     '''expression : expression AND expression
186                   | expression OR expression
187                   | NOT expression '''
188     if len(p) == 3:
189         p[0] = BinOp(p[1], None, p[2])
190     else:
191         p[0] = BinOp(p[2], p[1], p[3])
192
193 def p_expression_brackets(p):
194     '''expression : LPAREN expression RPAREN
195                   | term '''
196     if len(p) == 2:
197         p[0] = p[1]
198     else:
199         p[0] = p[2]
200
201 def p_term_string(p):
202     '''term : STRING
203             | CHARSTR '''
204     p[0] = String(p[1])
205
206 def p_term_int(p):
207     '''term : INTEGER '''
208     p[0] = Integer(p[1])
209
210 def p_term_bool_t(p):
211     '''term : TRUE '''
212     p[0] = Boolean(True)
213
214 def p_term_bool_f(p):
215     '''term : FALSE '''
216     p[0] = Boolean(False)
217
218 def p_term_null(p):
219     '''term : NULL '''
220     p[0] = None
221
222 # Error rule for syntax errors
223 def p_error(p):
224     print "Syntax error in input, p: " + str(p)
225     sys.exit(1)
226
227 parser = yacc.yacc(debug=True)
228
229 def parse(data):
230     parser.error = 0
231     result = parser.parse(data, tracking=True)
```

```
    if parser.error:
        return None
    return result
```

G.5 astsymtab

```
class Table(object):
    def __init__(self):
        super(Table, self).__init__()
        self.tuples = list()

    def put(self, key, value):
        self.tuples.append((key, value))

    def get(self, key):
        for tup in self.tuples:
            if tup[0] == key:
                return tup[1]

    def keys(self):
        keys = list()
        for tup in self.tuples:
            keys.append(tup[0])
        return keys
```

G.6 astvisitor

```
import sys
from astparse import *
from robemulator import Emulator, EmulatorError
from robcmds import MotorCommands, IRServoCommands, SensorCommands
from astsymtab import Table
from collections import Iterable

class SymbolTable:
    def __init__(self, parent):
        self.parent = parent
        self.children = list()
        self.table = Table()
        self.name = None

    def put(self, key, value, functions):
        if not self.contains(key):
            self.table.put(key, value)
```

```
17          elif key in functions:
18              self.table.put(key, value)
19          else:
20              raise ExecutionError("Error: key '" + str(key) + "' already
                    exists")
21
22      def contains(self, key):
23          if key in self.table.keys():
24              return True
25          elif self.parent is not None:
26              if self.parent.contains(key):
27                  return True
28              else:
29                  return False
30          else:
31              return False
32
33      def beginScope(self, name):
34          if name in self.table.keys():
35              for child in self.children:
36                  if child.name == name:
37                      return child
38          else:
39              newScope = SymbolTable(self)
40              newScope.name = name
41              self.children.append(newScope)
42              return newScope
43
44      def endScope(self):
45          return self.parent
46
47      def get(self, key):
48          if self.contains(key):
49              if key in self.table.keys():
50                  return self.table.get(key)
51              else:
52                  return self.parent.get(key)
53
54      def getValues(self):
55          if self.table is not None:
56              values = dict()
57              for tup in self.table.tuples:
58                  values[tup[0]] = tup[1]
59              for child in self.children:
60                  values[child] = child.getValues()
61              return values
62          else:
63              return None
64
```

```
65  class FunctionTable:
66      def __init__(self):
67          self.names = list()
68          self.params = dict()
69          self.bodies = dict()
70
71      def put(self, name, body):
72          self.names.append(name)
73          self.params[name] = list()
74          self.bodies[name] = body
75
76      def putParams(self, name, param):
77          self.params[name].append(param)
78
79      def getAllParams(self):
80          allparams = list()
81          for key in self.params.keys():
82              for param in self.params.get(key, None):
83                  allparams.append(param)
84          return allparams
85
86      def getParams(self, name):
87          return self.params.get(name, None)
88
89      def getBody(self, name):
90          return self.bodies.get(name, None)
91
92  class Visitor(object):
93      """Abstract Visitor class"""
94      def __init__(self):
95          super(Visitor, self).__init__()
96
97      def visit(self, node):
98          pass
99
100 class ExecutionError(Exception):
101     def __init__(self, value):
102         self.value = value
103
104     def __str__(self):
105         return repr(self.value)
106
107 class ExecutionVisitor(Visitor):
108     def __init__(self, values):
109         super(ExecutionVisitor, self).__init__()
110         self.root = self.current = SymbolTable(None)
111         self.functions = FunctionTable()
112         self.em = Emulator()
113         sys.setrecursionlimit(1000000000)
```

```
114
115     def flatten(self, _list):
116         flatlist = list()
117         for element in _list:
118             if isinstance(element, Iterable) and not isinstance(element,
                    basestring):
119                 for sublist in self.flatten(element):
120                     flatlist.append(sublist)
121             else:
122                 flatlist.append(element)
123
124         return flatlist
125
126     def visit(self, node):
127         if isinstance(node, ProgramNode):
128             self.em.connect()
129
130             print "Begin\n"
131             if node._left is not None:
132                 node._left.accept(self)
133             if node._right is not None:
134                 node._right.accept(self)
135             print "\nEnd\n\n"
136
137             print self.root.getValues()
138             print "\nnames:"
139             print self.functions.names
140             print "params:"
141             print self.functions.params
142             print "bodies:"
143             print self.functions.bodies
144
145             self.em.close_connection()
146
147         elif isinstance(node, ListNode):
148             if node._left is not None:
149                 node._left.accept(self)
150             if node._right is not None:
151                 node._right.accept(self)
152
153         elif isinstance(node, FunctionNode):
154             if node._left not in self.functions.names:
155                 self.functions.put(node._left, node._right)
156                 if node._params is not None:
157                     node._params.accept(self)
158             else:
159                 raise ExecutionError(str(node._left) + " already defined.")
160
161         elif isinstance(node, StatementNode):
```

```
162             if node._left is not None:
163                 node._left.accept(self)
164 
165         elif isinstance(node, MotorCommandNode):
166             if node._left is not None:
167                 left = node._left.accept(self)
168             if node._right is not None:
169                 right = node._right.accept(self)
170 
171             _cmd = MotorCommands(self.em)
172 
173             try:
174                 result = _cmd.set_speeds(str(left), str(right))
175                 if result is not None:
176                     print result
177             except EmulatorError, e:
178                 print e
179                 sys.exit(1)
180 
181         elif isinstance(node, ServoCommandNode):
182             if node._left is not None:
183                 left = node._left.accept(self)
184             if node._right is not None:
185                 right = node._right.accept(self)
186 
187             _cmd = IRServoCommands(self.em)
188 
189             try:
190                 result = _cmd.set_ir_servo(str(left), str(right))
191                 if result is not None:
192                     print result
193             except EmulatorError, e:
194                 print e
195                 sys.exit(1)
196 
197         elif isinstance(node, SensorCommandNode):
198             if node._left is not None:
199                 left = node._left.accept(self)
200             else:
201                 left = False
202             if node._right is not None:
203                 right = node._right.accept(self)
204             else:
205                 right = False
206             if node._centre is not None:
207                 centre = node._centre.accept(self)
208             else:
209                 centre = False
210             if node._prefix is not None:
```

```
211             prefix = node._prefix.accept(self)
212
213         _cmd = SensorCommands(self.em)
214
215         try:
216             result = _cmd.submit_cmd(prefix, left, right, centre)
217             if result is not None:
218                 print result
219         except EmulatorError, e:
220             print e
221             sys.exit(1)
222
223     elif isinstance(node, IfStatementNode):
224         if node.cond is not None:
225             if node.cond.accept(self):
226                 if node._left is not None:
227                     node._left.accept(self)
228             else:
229                 if node._right is not None:
230                     node._right.accept(self)
231
232     elif isinstance(node, DeclarationStatement):
233         if node._left is not None:
234             if node._right is not None:
235                 self.current.put(node._left, node._right,
                        self.functions.getAllParams())
236
237     elif isinstance(node, ListDeclarationStatement):
238         if node._right is not None:
239             body = node._right
240             first = node._first
241             rest = node._rest
242
243             if isinstance(body, StdListNode):
244                 pass
245             elif isinstance(body, IdentifierExpression):
246                 body = body.accept(self)
247             else:
248                 raise ExecutionError("Cannot assign non-list using
                        list declaration.")
249
250             if body._left is not None:
251                 self.current.put(first, body._left,
                        self.functions.getAllParams())
252             if body._right is not None:
253                 self.current.put(rest, body._right,
                        self.functions.getAllParams())
254
255     elif isinstance(node, ParamListNode):
```

```
256             if node._left is not None:
257                 node._left.accept(self)
258             if node._right is not None:
259                 node._right.accept(self)
260 
261         elif isinstance(node, ParamValListNode):
262             _params = list()
263             if node._left is not None:
264                 _params.append(node._left.accept(self))
265             if node._right is not None:
266                 _params.append(node._right.accept(self))
267 
268             # print _params
269             return self.flatten(_params)
270 
271         elif isinstance(node, ParamDeclarationStatement):
272             if node._right is not None:
273                 self.functions.putParams(self.functions.names[-1],
                        node._right)
274 
275         elif isinstance(node, PrintStatement):
276             if node._left is not None:
277                 expr = node._left
278                 # needs to be made recursive
279                 if (isinstance(expr, StdListNode)):
280                     print str(expr)
281                 else:
282                     print expr.accept(self)
283             else:
284                 print str(None)
285 
286         elif isinstance(node, StdListNode):
287             if node._right is not None:
288                 node._right.accept(self)
289             return node._left
290 
291         elif isinstance(node, BinOp):
292             if node._left is not None:
293                 _left = node._left.accept(self)
294             if node._right is not None:
295                 _right = node._right.accept(self)
296             if node._op == '+':
297                 return _left + _right
298             elif node._op == '-':
299                 return _left - _right
300             elif node._op == '*':
301                 return _left * _right
302             elif node._op == '/':
303                 return _left / _right
```

```
304             elif node._op == '%':
305                 return _left % _right
306             elif node._op == '==':
307                 return _left == _right
308             elif node._op == '>':
309                 return _left > _right
310             elif node._op == '>=':
311                 return _left >= _right
312             elif node._op == '<':
313                 return _left < _right
314             elif node._op == '<=':
315                 return _left <= _right
316             elif node._op == 'and':
317                 return _left and _right
318             elif node._op == 'or':
319                 return _left or _right
320             elif node._op == 'not':
321                 return not _right
322 
323         elif isinstance(node, IdentifierExpression):
324             if node._left is not None:
325                 value = self.current.get(node._left)
326                 if value is not None:
327                     if isinstance(value, StdListNode):
328                         return value
329                     else:
330                         return value.accept(self)
331                 else:
332                     return str(None)
333 
334         elif isinstance(node, FunctionIdentifierExpression):
335             if node._left is not None:
336                 if node._left in self.functions.names:
337                     num_params = len(self.functions.params[node._left])
338                     self.current = self.current.beginScope(node._left)
339                     if node._right is None:
340                         if num_params > 0:
341                             raise ExecutionError(
342                                 str(node._left) +
343                                 " requires " +
344                                 str(num_params) +
345                                 " parameters, none given"
346                             )
347                         else:
348                             pass
349                     else:
350                         given_params = node._right.accept(self)
351                         if not (num_params == len(given_params)):
352                             raise ExecutionError(
```

```
                            str(node._left) +
                            " requires " +
                            str(num_params) +
                            " parameters, " +
                            str(len(given_params)) +
                            " given"
                        )
                    else:
                        param_labels =
                             self.functions.getParams(node._left)
                        for index in range(0, len(given_params)):
                            self.current.put(param_labels[index],
                                Integer(given_params[index]),
                                self.functions.getAllParams())

                if self.functions.getBody(node._left) is not None:
                    self.functions.getBody(node._left).accept(self)
                self.current = self.current.endScope()
            else:
                raise ExecutionError(str(node._left) + " not defined")

        elif isinstance(node, ParamExpression):
            if node._left is not None:
                return node._left.accept(self)

        elif isinstance(node, String):
            return node._value

        elif isinstance(node, Integer):
            return node._value

        elif isinstance(node, Boolean):
            return node._value
```

G.7 astinterp

```
from astvisitor import *

class SimpleInterpreter:
    def __init__(self, prog):
        self.prog = prog

    def run(self):
        self.vars = {}
        self.lists = {}
        self.error = False

```

```
        self.printSeparator()

        try:
            self.prog.accept(ExecutionVisitor(None))
        except Exception, e:
            print "Error: " + str(e)
            self.error = True

        self.printSeparator()

    def printSeparator(self):
        print "\n *** \n"
```

G.8 astlang

```
import sys
from astparse import parse
from astinterp import SimpleInterpreter

sys.path.insert(0, "../..")

if sys.version_info[0] >= 3:
    raw_input = input

if len(sys.argv) == 2:
    print "***Beginning Parse***"
    data = open(sys.argv[1]).read()
    prog = parse(data)
    if prog is None:
        print "***End***"
        raise SystemExit
    b = SimpleInterpreter(prog)
    try:
        b.run()
        print "***End***"
        raise SystemExit
    except RuntimeError, e:
        print e
        sys.exit(0)
```