Summary

The purpose of this project was to build a system whereby Lego Mindstorms robots could be given the ability to perform planned behaviour. By “planned behaviour”, it is meant that the robot will, in attempting to achieve a goal, identify pre-specified actions which will lead it to this goal, and execute them.
The actual implementation of this project involved the design of a suitable planning system, and the development of the necessary tools to enact this design. This involved, principally, the coding of a planning compiler – which converted human-written action code into a structure which the robot could attribute semantics to; and the development of a level of “artificial intelligence” on the robot which could read in this code and utilize it in pursuit of a number of goals.
Due to both time constraints, and the technical constraints of the Lego Mindstorms technology; not all of the theoretical design was expected to be practically implemented.

The minimum requirements are as follows:

- To build a robot capable of autonomous navigation around a given space.
- To enhance the robot so that it can perform goal-oriented tasks.
- To test this ability by making it achieve certain goals (e.g. to navigate a "maze" etc.)
- To consider methodologies for adding primitive learning, in order to improve results.

Table of Contents

1 Introduction

1.1 Aim

To produce a robot, building on existing projects, that can perform goal-oriented behaviour. This aim can be divided into two distinct sections – the theoretical aims and the practical aims:

Theoretical

This system will allow the user to plan two distinct areas of robot behaviour: firstly, they will be able to give the robot a list of goals which it must complete; secondly, they will be able to instruct the robot as to how to complete those goals. Note that the project will be interested in the theory of how plans are implemented, rather than the theory behind their creation.

Practical

The focus of the practical portion of the project will be on testing and validating the theoretical work. I will produce some test plans and make the robot complete some simple tasks that demonstrate the planning system in action.

As the theoretical aims are hard to strictly define, the success of the project will be measured by how well the robot performs in the practical tests.

1.2 Objectives

Again, the objectives can be divided into theoretical and practical.

Theoretical

To produce a piece of software that can be used to build plans. This will involve the creation of a “planning language” which is compiled into a format the robot can understand. This system will be constructed using the principles of Set Theory, Lambda Calculus and Situation Calculus. An additional aim of the project is to be able to express the planning system in terms of mathematics and theoretical computer science.

Practical

The principle practical objective is the continual testing of the theory by downloading fragments to the robot and ensuring that they complete successfully. These tests will grow in complexity as the planning system is extended. The most elaborate test will be a simple maze that the robot is instructed to navigate. As the completion of the project will involve the creation of a user manual, theses tests will be developed into examples of functionality for the manual.

1.3 Relevance of Project to Computer Science Degree

It is not difficult to see how the development of a compiler is relevant to a Computer Science degree. Firstly, the theoretical computer science concepts of Finite State Automata were central to understanding how the robot moves from one situation to another. Studying knowledge management was also very helpful in understanding how to model the “knowledge” of the robot – particularly where temporal aspects come into play. Most of all, though, my studying of coding semantics was most useful – as being able to state the semantics of code in a machine-readable manner was central to making a workable system. The practical side of the project was not as relevant to my particular degree, and that is why I chose to focus on the actual creation of plans – rather than the plans themselves.

1.4 Structure of the Report

I have structured this report so as to focus centrally on how the planning system was conceived and produced. I have endeavoured, where possible, to delegate the more intricate details of design to appendices in order to avoid repetition, convolution and circumlocution.

The chapter contents are thus:

Chapter 1: Introduction to the report, setting out the objectives.

Chapter 2: Documentation on the background to the problem, explaining the robot for which a planning system is being produced.

Chapter 3: Documentation on the design of the planning system itself.

Chapter 4: Report on how the design was implemented, and what issues arose in stepping from abstract design to specific implementation.

Chapter 5: Report on how I tested the planning system, and how it responded to these tests.

Chapter 6: An evaluation of how successful the project was, both in terms of the validity of the planning theory, and how effectively that theory was put into practice.

Chapter 7: My personal conclusion on the project.

Chapter 8: My suggestions for how the concepts set out in this project could be developed in the future.

2 Background to the Problem

2.1 Preliminary Review of Sources

Before the project could be begun, it was necessary to conduct a review of the Lego Mindstorms system, in order to fully appreciate the materials which were to be worked with. Invaluable to this was the software and manuals provided with a standard Lego Mindstorms package. It was from this source that I learnt how to build an L.M. robot and what hardware was available to use with it – such as sensor and motor technology.

In understanding what approach to take for the creation of plans, the World Wide Web was a very useful resource. This gave me access to recent study on topics such as Situation Calculus, from groups such as the Stanford Formal Reasoning Group. In relation to the specifics of the project however, I could not find any directly helpful material. Planning for Lego Mindstorms appears to be an under-researched area. For me, though, this was not a considerable problem as I find myself most comfortable following my own direction.

2.2 Mindstorms Robots

Lego Mindstorms Robots are a combination of Lego pieces and what is called a brick. This is a microcomputer, powered by an RCX chip, that allows the Lego model to be controlled by a computer programme. The RCX brick can be connected to a number of sensor and motors, that allow the Lego robot to move in different ways in response to varying inputs. One of the great advantages of the Lego Mindstorms system is the speed and ease with which robots can be built.

2.2.1 The RCX Chip

The actual processor inside an RCX brick is an 8-bit Hitachi H8. The brick also provides three sensor ports, three motor ports and an infra-red serial port. The brick has 32Kb of external RAM, with a further 16Kb of space for the firmware. Although the processor does not have a lot of power, this is sufficient for most AI needs. The RCX brick also provides an LCD screen which can be used to communicate simple messages with the user – although the screen is very limited in what it may display.

2.2.2 Development Tools

There are a number of development tools available for working with the Lego Mindstorms systems. The first of these is the graphical coding tool that is provided with the Robots Invention System. This allows you to very quickly build small programs by dragging and dropping stock operations onto a visual workspace, and setting how these operations relate to each other. Unfortunately, the power of this system is very limited. It is simply not possible to develop a practically usable planning system with the RIS approach.

The second available tool for developing for Lego Mindstorms is Not-Quite-C. This is a C-like language which allows much more control over the robot. A major advantage of this system is that it is well-developed and mature. However, it does impose some significant restrictions which prevent it from being an option for this project. Firstly, it does not support recursion – which greatly complicates the processing of compiled plans.

This leaves two other major languages for developing in – LeJos and legOS. LeJos is a Java-based system; legOS is C-based (but supports much more functionality than NQC).
Because LeJos is Java-based, and therefore cross-platformable, it is a very attractive proposition for this project – as it is necessary to develop code on both the robot, and the host computer. With LeJos, therefore, some degree of code sharing on these two systems can be done. However, the LeJos API does not follow the JRE completely,

therefore some sections of code still need to be re-written. LeJos also does not yet currently support garbage collection – this makes memory management a critical issue.
legOS is a cross-compiler for GNU GCC to allow coding with GCC for the RCX's Hitachi H8 microprocessor. As with LeJos, legOS is still very immature – currently it is in version 0.2. Its documentation in particular is very sparse. Also its lack of a high-level, mostly device-independent API like LeJos make it more difficult to code with.

After consideration of the possible development languages, I decided that LeJos would be the system most capable of producing the system in.

For creation of the host-side tools, I decided to code mostly in C. However the downloading program, for copying action code from the host PC to the robot, was coded in Java – making extensive use of the LeJos extensions.

2.3 Previous Projects

As has been stated earlier, this project does not coincide with the previous work done on Lego Mindstorms by any great degree. Therefore, the previous projects were not especially useful. However, they didn't provide some guidance in the beginning in acquainting myself with the Mindstorms system.

2.4 Project Methodology

Due to the investigative nature of this project, I was disinclined to create too many rigid boundaries for design and implementation. In practice, the LeJos system proved to be very tedious and difficult to develop with, and this problematic nature did greatly impact on the course the implementation took.
In the most part, I endeavoured to take an iterative approach to the tasks. Designing specific elements of the system, then implementing them, then testing and re-

designing them where necessary. Much of the designed theory was not reproduced in a practical setting – which re-iterates the theoretical focus I placed on the project. Finally I would highlight that the title of the project is *Introducing* Planning to Lego Mindstorms Robots: the project involved a great deal of trial-and-error and at some times ad-hoc design was necessary, as stumbling blocks to implementation were not identified in advance.

2.5 Project Schedule

As stated above, the LeJos system proved to be more difficult to work with than originally expected. I underestimated the tedious nature of developing for an embedded system. Therefore several elements of the initial project schedule proved to not transpire as planned. In particular, the development of a workable robot intelligence took much longer than was initially foreseen.

3 Design of the System

3.1 The Instruction Assembler

3.1.1 Overview

The Instruction Assembler allows the creation of instructions, which bridge the gap between the functional paradigm of the actions and the “real world”. Instructions are built as subroutines in an assembly language. The structure of the assembly code enforces a functional setup. Instructions are executed inside actions in two places: in Pre-Conditions, Post-Conditions and Ramifications to calculate the effects of an action; and in the action code itself to move the robot.

3.1.2 Assembly Overview

The assembly language used for the interpreter is pretty straightforward to anybody with a prior understanding of assembly. The only major point to begin with is the independent nature of the various modules of the system’s design. Therefore, for example, the compiled assembly code provides for up to 256 registers, but the robot executing the code may well not. Due to this nature, there are inevitable matters of compatibility – functions written in the assembly language must be coded with a foreknowledge of what they will be executed on. The division between code and robot can be seen as analogous to vertex shaders for 3D GPUs – a GPU which only supports v1.0 shaders cannot execute v2.1 shader code.

However, there are some general rules: the robot’s interpreter supports a number of floating-point registers. Theses registers are used for manipulation of data which can come from one of three sources – 1.) data read from a sensor; 2.) data loaded immediately in the assembly code; 3.) data passed to the function as a parameter. In addition to these temporary registers, the interpreter also supports a number of

parameter registers. These are loaded with values before the function is executed, the data held in the parameter registers can then be moved into the temporary registers for manipulation in the function. Note that the assembly language provides no means to poll what or how much data has been passed into a function – the process of type-checking function calls must be done, if it is done at all, by the compiler software.

In addition to the registers, the interpreter also supports a standard set of arithmetic, branching and stack operations to allow the manipulation of those registers. A complete index of opcodes, with commentary, can be found in the Appendix B.

3.1.3 Interpretation

When code written in the planning compiler calls functions, it is then that the interpreter is invoked. Function calls in compiled code link into the robot's instruction firmware. Firstly, the values passed as parameters to the function are loaded into the interpreter's parameter registers. Then the Instruction Pointer is set to point to the beginning of the executed function. Interpretation then begins. Note that the stack is not reset when a new call is made, and therefore can be used (theoretically) to hold data across function calls – though obviously this is bad design and could cause instability.

3.2 The Planning Compiler

3.2.1 Overview

The Planning Compiler brings together code that makes up actions. This currently consists of four groups of data – Pre-Conditions, Post-Conditions, Ramifications and Code. Theoretically, the scope of the compiler could be greatly enhanced, but this section will focus on the current implementation.

The Planning Compiler is used to build a “model” of the world the robot is “inhabiting”, with instructions to the robot as to how to interact with that world. The world is modeled via a collection of Boolean and integer variables. Note that these variables are not declared in advance, when the robot comes across a variable it has not seen before it declares it new and sets it’s value to 0 / true. In the most part, integer variables are used to store information such as Cartesian co-ordinates; and Boolean variables are used to store decision control states.

3.2.2 Compiled Elements

The Planning Compiler compiles action lists. Each action in the action list specifies the code and semantics for the action. An action consists of a set of pre-conditions, a set of post-conditions, a set of ramifications, and a code set.
Theses sets are sets of lambda strings. A lambda string is a series of tunneled function calls – that is that each function call is the parameter to another function call – with a root function. The root function tells the robot what information the lambda string evaluates, and is based on Situation Calculus. Usually this is H(…,…) meaning *holds* – which declares a variable to hold a specific value.

Occasionally a function will contain no lambda string; this is when it is H(…) or N (…). H() when given only a single parameter declares the given Boolean variable to be true; N() declares the given Boolean variable to be false.

3.2.2.1 Pre-Conditions

Pre-Conditions are those elements of the model that must be set a particular way in order for the action to be executed. They are the first thing used to ascertain what actions can be executed by the robot. They are often used in conjunction with Boolean variables to provide decision control – changing what actions can and cannot be executed by the robot, dependent on the current circumstance. For example, if the robot has just bumped into a wall the a *bumped* Boolean state

variable may be set to true – any actions then that require the robot to move forward will declare that *bumped* must be false in their pre-conditions.

3.2.2.2 Post-Conditions

Post-Conditions define how the model will have changed after the action has been executed. Post-Conditions are deterministic, and therefore can are very useful in an environment that can be accurately predicted. In the "real world" though, their use is heavily limited as the robot will not always (sometimes may never) do exactly as it is expected to.

It is possible, though not necessarily desirable, to create non-deterministic post-conditions – that is post-conditions that are not actually conditional. Again, if such an approach is taken in a plan then the robot executing it must be compatible with such an approach. In practice, ramifications can also be used to negate the conditionality of post-conditions – e.g. if a post-condition declares X to be incremented by 1, and a ramification increments it by 1 also; then when the action is finished X will have increased by 2.

3.2.2.3 Ramifications

Ramifications resolve the problem of deterministic Post-Conditions in the non-deterministic "real world". Ramifications are like Post-Conditions, except that they are only affected on the planning model under certain circumstances. They are specified with the action, like Post-Conditions, but their execution is done in response to *queries* in the action code.

Like in many other areas, there is a degree of flexibility with execution here. Ramifications may be evaluated immediately, or their evaluation may be delayed until the action has finished. Furthermore, if ramifications are evaluated at the end then they may be evaluated before, or after, post-conditions are. These decisions are,

in the current implementation, not made at the compilation stage, but rather delegated to the robot. Therefore, again, any plans where such matters might cause problems must be built with knowledge of how they will be executed in mind.

3.2.2.4 Code

The code is the actual instructions that the robot executes in running the action. Code is specified as a series of lambda strings which are executed in succession. Code strings may also specify queries: these are when the result of a lambda string is tested and a specific Ramification is chosen to modify the planning model based on this test.

3.2.3 Source Code

3.2.3.1 Syntax

The syntax of the planning compiler may seem unusual at first, an example follows:

```
# Moves forward one unit.
ACTION_START: move_forward_one

#  List of pre-conditions.
H(started):

#  List of post-conditions.
H(X', add(X, mul(10, cos(R))));
H(Y', add(Y, mul(10, sin(R))));

#  List of ramifications.
H(bumped');
N(bumped'):
```

```
# List of code
forward(10, 1)?0,1;
turn(90):

ACTION_END.
```

The ACTION_START: token tells the compiler that a new action is about to be begun, the string following it is the name to assign to this action. After this, the elements of the action are specified in their order. The ACTION_END. token then specifies that the data for this action is finished.

Each item in a list of strings is separated by a semicolon; each list of strings is separated by a colon. The strings themselves are far closer to traditional programming language design, written as the name of the function followed by a tuple of parameters.

In the code section we see a slight variance – a ? symbol following one of the strings: this denotes a query. When the *forward* function is executed, it's return value is queried: if it returns TRUE then ramification 0 is evaluated, if it returns FALSE then ramification 1 is executed.

The final point of syntax to note is the distinction between an unprimed and a primed variable. X and X', for example, are treated differently by the robot. Therefore it is very important to ensure that the correct form is written. Generally this should be unprimed in pre-conditions and code, and primed in post-conditions and ramifications.

3.2.3.2 Semantics

The definition of semantics in the planning compiler is very important (note the distinction between X and X' above). The reason for this is that the purpose of pre-conditions, post-conditions and ramifications is to attribute understanding of the

semantics of an action to the robot. If the lambda strings are not specified properly the robot will not choose actions correctly.

Each lambda string given in the compiler code is recursed by the compiler. It then converts it into a tree. Each node in the tree has an associated string, which is the name given in the code. Function calls will also have a number of parameters which are specified as the children of this node. In this way functions are tunneled into each other. It should be noted at this point that the compiler system provides on provision for the creation of functions which take no parameters – as a function with no parameters is interpreted as a variable. However this limitation should not pose a problem if functional programming rules are being properly followed.

Each function call specified in a lambda string refers to a function specified in the firmware assembled by the instruction assembler. It is here that an important delegation of duties is made by the system – as functional programming can often be messy in performing such things as arithmetic, the actual data manipulation is performed by an imperative interpreter; the lambda strings are used to structure the functions into a compact and easily evaluable form.
In addition to the operations that can be performed in the instruction firmware, the system also provides for the use of built-in functions for such things as trigonometry. As with many other areas, the availability of built-in functions is dependent on the intelligence on the robot, and is not addressed directly by the compiler.

3.3 The Downloader

3.3.1 Overview

The downloader is a simple piece of code which is used to send instruction and action information to the robot. It is executed on the command line with a list of object files to send. The robot must be placed in view of the IR tower and told to accept new data, this data is then copied to the robot. etc.

3.4 The Robot's Intelligence

3.4.1 Overview

The robot's intelligence is the code that is downloaded onto the robot to make it read and execute plans. It contains some basic structure code for moving the robot; an interpreter for the assembly code; and a process for deciding which action to take.

3.4.2 Code Structure

Due to LeJos being a Java system, the code on the robot is divided into classes. Each class handling a different encapsulated field. Therefore there is a class to handle lambda strings; a class to handle sets of these strings; a class to execute bytecode; et cetera.

3.4.3 Decision Model

To reiterate a constant point, the planning system is designed to be modular. Therefore, each separate entity in the system is independent, and the plan built in a compiler poses no restrictions on how it is utilized. This means that a single plan can be used with many different approaches to decision making, and the best can be found. It also means that sometimes a plan will simply not be compatible with some decision processes.

The most simple kind of decision model which can be used is the deterministic state approach. This is the most simplistic interpretation of the design of the system. When the robot is to choose an action, it is in a given state. It calculates all the possible actions it could perform to create a list of possible state transitions. It then makes a choice as to which transition to make, and performs the relevant action. This choice is made by deciding which action has post-conditions closest to the goal state.

This approach is very easy to implement, and to predict how it will behave. Unfortunately, it is also very easy for such an approach to quickly fall into problems. Although it may be convenient to treat the world space as a finite number of deterministic states, this is realistically never true. Therefore some non-deterministic approach must be taken.

The probabilistic approach may be helpful. Although at current the planning compiler has no system to assign probabilities to post-conditions and ramifications, this could be added in without too much difficulty.
The most simple form of probabilistic approach is to factor in ramifications into the decision making process, and providing a bias to post-conditions so that these still form the main decision point.

3.4.4 Limitations

Unfortunately, LeJos and Lego Mindstorms have a number of limitations which make the creation of such a system a great difficulty. Firstly, robots can be extremely tedious and difficult to code – as the amount of feedback which can be acquired when an error occurs is minimal. It also takes a great amount of time to download programs onto the robot. Furthermore, the memory available on the robot is very small, and LeJos at current provides no garbage collection mechanism.

3.5 The Simulator

3.5.1 Overview

The Simulator is a piece of code run on the host computer. It emulates a PlanBot and can be used to test the validity of plans without the work of downloading and setting

up an environment. It is limited in two principle ways: firstly, there is a limit to what you can simulate; secondly, the “real world” rarely turns out the same as the theory. The purpose behind the Simulator is that a “map” of the real world is created in an art package, such as Microsoft Paint. The map, along with some extra data, is loaded into the simulator and the simulation is run. The extra data consists of – an instruction firmware file, a list of goals, a plan, and the co-ordinates of where in the map the robot is to begin.

The map should consist of a white pixels on a black background. The white pixels are interpreted by the simulator as walls, any other colour is ignored as empty space. This means that as well as a background, other colours can be used to identify points of interest on the map.

4 Implementation of Design

4.1 Overview

The implementation of the system was a partial success. The Instruction Assembler and Planning Compiler were completed and are fully functional. Although the Planning Compiler could be made more flexible if it were developed using standard tools such as Flex, rather than hard-coded as one file. The execution of the plans on the robot, however, could not be done. Although a lot of code was written and tested, the implementation of such a system on a Lego Mindstorms robot proved to be much harder than envisaged. The simulator of the robot on a host machine, was begun but not completed.

4.2 Requisite Changes of Design

The design of the planning system stretches far beyond the capabilities of the Lego Mindstorms robot, and the time given to develop such a system in for a Final Year Project. Therefore, there have been a large number of theoretical principles I have considered, which could not be realized even in the host development tools. Where appropriate, I have endeavoured to considered such matters in the concluding section of the report.

4.3 The Robot

Implementation of the intelligence for the robot was particularly difficult. On the occurrence of an error, the only available information for debugging is a simple number – of which I was unable to locate any explanation of. It also takes a small degree of time to download a program to a Lego Mindstorms robot; this makes it take a reasonably long time to get real results.

A robot was made which could execution instruction firmware, however the memory footprint of a class to do this execution proved to be too great for the LeJos' restricted space.
Although a complete, working downloader tool was developed for the robot; the process of downloading to the robot failed to be completed. Whether or not this was due to memory limits on the robot is not clear, as I was unable to ascertain the exact cause of the error.

4.4 The Instruction Assembler

The Instruction Assembler was completed successfully. It was developed in C/C++ as my programming language of choice. In addition to the assembler, a simple testing tool was written which reads in an assembled file and ensures that it has been correctly assembled.
The Assembler is executed on the command-line with the name of an input and output file specified. The Assembler is a three-pass assembler, as I did not consider efficiency to be a major concern considering the power of modern computing, and the very small size assembly files were expected to be.
The first pass of the assembler reads the assembly source and identifies all of the labels which are declared in it. The second pass then does the actual assembly, storing a list of all the places in the file where a reference is labeled, and storing the actual address that label refers to. Finally, in the third place label references are replaced with their actual addresses in the compiled object file.
The format of the final file is then a list of labels and their addresses, then followed by the assembled object code. The list of labels is maintained for external linkage – when the robot executes a function, it needs to know where in the assembled file that function is. In the current implementation, all labels are considered to be exported.

4.5 The Planning Compiler

The Planning Compiler was completed successfully, again in C/C++. Again an extra tool was developed which tests whether or not the object file has been compiled successfully.
The Compiler is executed on the command-line with the input and output files specified, along with an instruction firmware file. This is the firmware file which will be used on the robot, and is specified so that the compiler can check that the functions called will actually exist at run-time.
Compilation is done in a single pass. The compiler reads in the file piece by piece, building up trees for the lambda strings as it goes. Due to the compressed nature of the compilers design – which was done for speed of completion – it is not greatly flexible. As has been specified earlier, use of tools such as Flex could produce a more user friendly solution.
In the current implementation, all elements of data for a single action are group together. The data is split up into four sections: pre-conditions, post-conditions, ramifications and then code. The data must be given in this order, as there is no way to move it around. Although it might be more user-friendly to add some flexibility here, it strikes me that there is little benefit to being able to change the order items are specified in, and therefore it was not a design priority.

In addition to the compiler for the plan; there is also a Goal Compiler. This is essentially a stripped down version of the Planning Compiler which builds an object file that specifies what the goals for the robot will be. This program works much the same way as the Planning Compiler does.

4.5 The Simulator

Unfortunately, I did not manage to have a simulator completed by the deadline. The simulator is, again, executed from the command-line; but instead of data being passed on the command-line it reads it from a configuration file. This file specifies where the robot should begin in the simulator, what the names of the firmware, plan and goal files are; and what the name of the map file is.

At current, the simulator simply loads up the map file and displays it. Although there is some code in the simulator to evaluate lambda strings and such, this code is untested and not fully implemented.

4.6 Plans

Although in the end no plans were tested on a working robot, some plans were constructed. Appendix C show some examples of planning files that were used for testing. These files can be compiled in the planning compiler, and then their trees can be seen by running the testing utility on them.

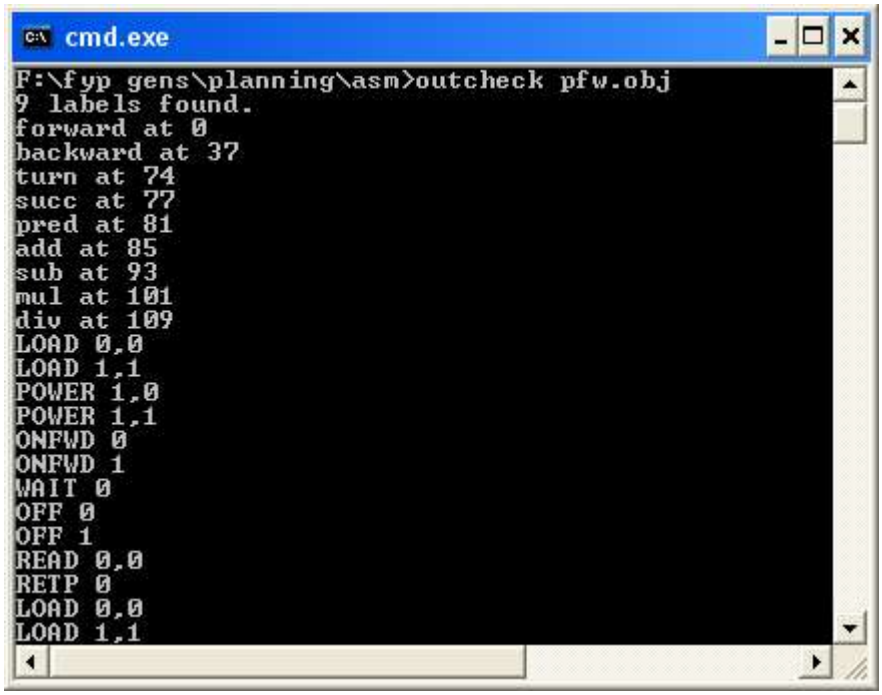
5 Final Testing

5.1 How the System was Tested

The testing of the system was greatly limited by the fact that most of it was not functional. However the object files created by the Instruction Assembler and Planning Compiler were tested, with the use of tools that printed to the console the data stored in them. Manual analysis of the resultant information was then used to ensure that the assembled/compiled file was correct.

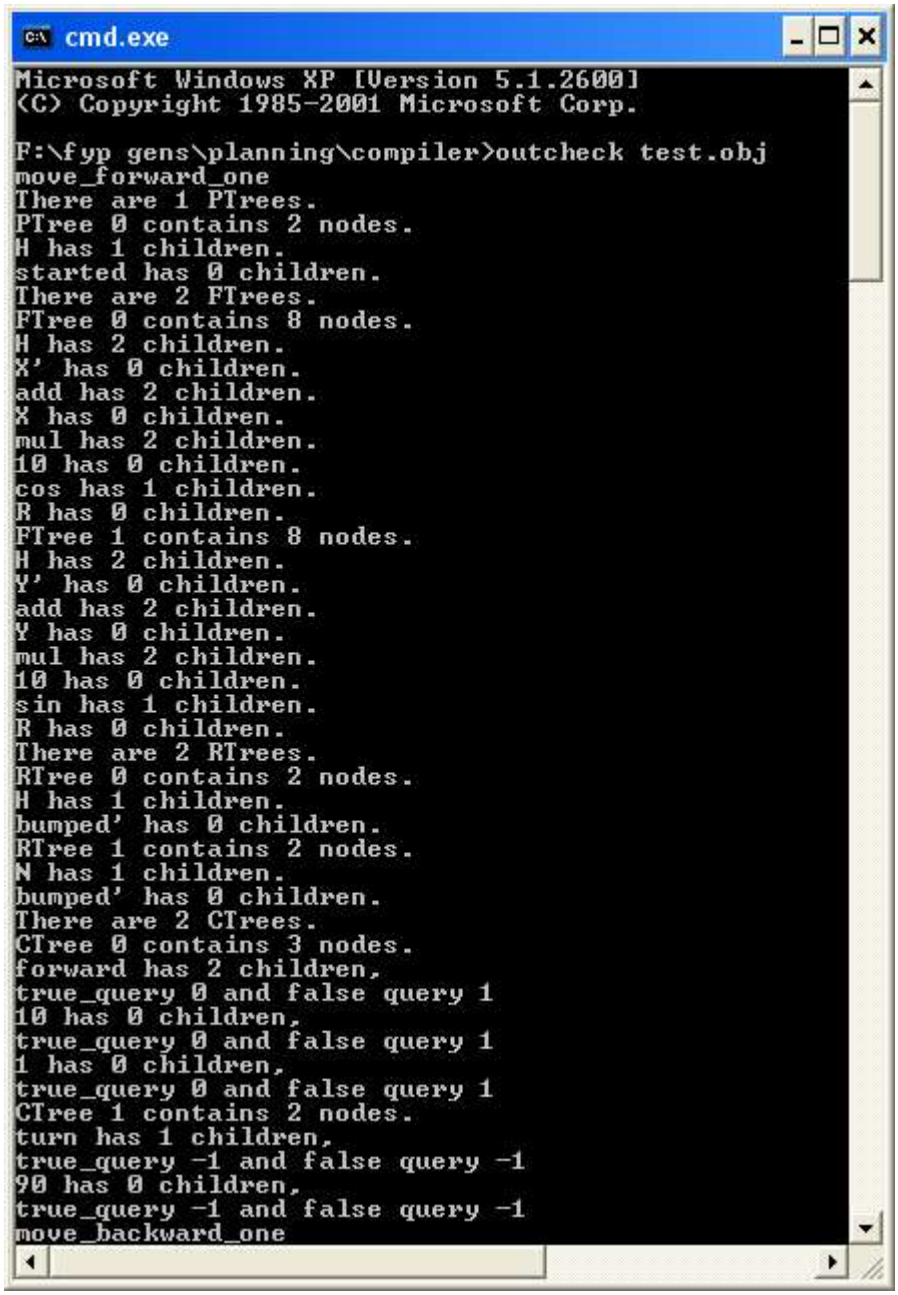
5.2 The Instruction Assembler

The testing utility for the Instruction Assembler reads in the label block, printing out the names of all the labels and the addresses they refer to; it then prints out each opcode in order. An example of the output of the testing file follows:



5.3 The Planning Compiler

The testing utility for the Planning Compiler reads in a compiled file and prints out the information stored. This consists, for each action, of: the name of the action, the number of each compiled element there is followed by a textual representation of each tree. At first glance, this does not seem to be an easy file to read; but when there is a problem in the compiled file it is not to difficult to spot it.


cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
F:\fyp gens\planning\compiler>outcheck test.obj
move_forward_one
There are 1 PTrees.
PTree 0 contains 2 nodes.
H has 1 children.
started has 0 children.
There are 2 FTrees.
FTree 0 contains 8 nodes.
H has 2 children.
X' has 0 children.
add has 2 children.
X has 0 children.
mul has 2 children.
10 has 0 children.
cos has 1 children.
R has 0 children.
FTree 1 contains 8 nodes.
H has 2 children.
Y' has 0 children.
add has 2 children.
Y has 0 children.
mul has 2 children.
10 has 0 children.
sin has 1 children.
R has 0 children.
There are 2 RTrees.
RTree 0 contains 2 nodes.
H has 1 children.
bumped' has 0 children.
RTree 1 contains 2 nodes.
N has 1 children.
bumped' has 0 children.
There are 2 CTrees.
CTree 0 contains 3 nodes.
forward has 2 children,
true_query 0 and false query 1
10 has 0 children,
true_query 0 and false query 1
1 has 0 children,
true_query 0 and false query 1
CTree 1 contains 2 nodes.
turn has 1 children,
true_query -1 and false query -1
90 has 0 children,
true_query -1 and false query -1
move_backward_one

6 Evaluation

6.1 Capability

The capability of the system is divided into two parts. The tools used on the host machine are, mostly, fully functional. The Assembler and Compiler were perfectly. However, they could be improved.

The major drawback of the Assembler is the inability to control what functions are exported. This could be used to prevent subroutines in the assembly source, which are not functions, from being mistaken for them. It may also be useful to be able to specify parameter information for functions in the assembler, in order to prevent erroneous calling. I consider the capability of the Instruction Assembler to be a qualified success.

The Planning Compiler performs enough operation to make plans which could be used on a PlanBot. The number of enhancements that could be made to it are great, though these would really involve extensions to the planning system in general, rather than specifically the compiler. Such enhancements are discussed in the concluding chapter. I consider the capability of the Planning Compiler to be a complete success.

The best enhancement to the capability of the Simulator would be to make it actually work. More time is needed to be put into it to make it a useable tool. As of yet it remains a work-in-progress.

The Downloader tool manages to successfully download a file to the robot, even if the robot can successfully download it. Therefore I would consider this small tool to be completely capable.

The Robot was unable to be gotten to work in any manner. Therefore the capability of the Robot is, unfortunately, nil.

6.2 Usability

When considering the usability of the command-line tools, the major point to make is that they are command-line tools. This makes them difficult to use for many people, a solution to this problem is discussed later.
In addition, the tools for testing created object files are aimed at developers of the tools and not end-users. Therefore the way they present their information could be greatly enhanced – again the command-line is a major limiting factor here.

The usability of the Planning Compiler could also be enhanced. Certain elements of the syntax of the plan source files is, perhaps, unusual to most programmers. Considering furthermore that the demographic for the Planning Compiler should be people whose field of expertise is more towards AI than Computer Science, this problem is exacerbated.

6.3 Portability

The portability of the various tools varies across them. The most portable tool, of course, is the Downloader because it is written in Java. The only real portability issue here is if somebody wished to port the system to work with something other than LeJos or Lego Mindstorms.

The Instruction Assembler and Planning Compiler are coded to C/C++ standards, and therefore should be re-compilable to any CLI without much difficulty. Though one may meet CR/LF problems.

The Simulator, in as much as it is built, is built specifically for Windows. Any attempt to port it to a different Operating System would requiring re-writing a great deal of it.

The Robot Intelligence is, again, written in Java with LeJos. Therefore, if somebody were to develop a strand of LeJos for another robot system, porting the Intelligence to it would not require a great deal of work. However, there was not a lot of successful code written for the robot anyway, so this is probably an irrelevant point.

7 Conclusion

7.1 Project Requirements

I think it is fair to say that the project did not meet it's minimum requirements. I think that in trying to do too much I fell in to the all-too-common mistake of failing to do what was expected in the first place. The project, for certain, was too ambitious. Neurosis is defined as the inability to resolve the ideal with the reality – and in that sense I think it was definitely a neurotic project.

Having said that, the theoretical aims were, I feel, successfully completed. An assembler and compiler were produced, they were used to produce some, albeit small, plans. I do believe that somebody in the future could extend this work to produce something practically useful – I simply wasn't able to do that myself.

I would, finally, remind the reader that the title of this project is *Introducing* Planning to Lego Mindstorms Robots.

7.2 LeJos

Personally, I blame LeJos entirely for my downfall. The problem with LeJos is that it is too immature. There simple isn't the software available for the Lego Mindstorms robots at the moment to produce a system of this complexity. Having said that, the Lego Mindstorms Robots may never be able to implement my system. It's a first principle of Computer Science that the greater the level of abstraction used in the development of code, the greater the need for memory and performance from the target machine. It is perhaps necessary to have a technically "better" robot to use my planning system effectively.

7.3 The Finished Toolset

I feel that the finished toolset is a useful and effective set of functionality. There is a lot of work that could be done to make them better – but what this goes to show is how great the scope of the design was to begin with. The usability of the tools, in particular, I think is their weakness. This is where I, personally, would focus further development of them.

8 Potential Future Work

8.1 The Planning Compiler

The Planning Compiler is the section where the most scope for enhancement is, although the relative benefits of such enhancements are not easy to predict.

8.1.1 Use of Generation Tools Such as Flex

At the moment the compiler is entirely hand-coded. For its current needs this is sufficient, but if you wanted to greatly improve the capabilities of the system then you may wish to separate the components of the compiler in accordance with current compiler design methodologies. This would include such things as using Flex to auto-generate a lexical analyzer for the planning language.

8.1.2 Syntax Extensions

The syntax of the planning source could be greatly enhanced by making it much closer to traditional coding. Implementation of things such as conditionals could allow for flow control statements, for example. The syntax could also be better set out so that the code is easier to read than it is at the moment.

8.1.3 Conditionals

Loosely speaking, conditionals are to Pre-Conditions what Ramifications are to Post-Conditions. A Conditional is a lambda string which is evaluated to produce an integer or Boolean result. This result is then tested for a specific value and a block of code is or is not executed based on that test. They could be used to produce more dynamic and flexible actions – though it is important to point out that this would further chip away at the determinism of such actions.

8.1.4 Auto-Generation of Lambda Strings

This concept is only really appropriate to deterministic modeling, and involves enhancements to the Instruction Assembler as well as the Planning Compiler.

The basic principle here is that the lambda strings which define semantics would be attached to functions, rather than actions. As an action is simply a string of functions executed consecutively, the lambda strings for those functions could be tunneled in the same way that they are in the code strings.
The principle follows from the mathematical law that a f(x) followed by g(x) is equivalent to f(g(x)). The same law can be applied to the semantic functions used in lambda strings:

i.e if f(x).g(x) ? f(g(x)) => s(x).t(x) ? s(t(x))

(where s(x) and t(x) are semantics to f(x) and g(x) respectively).

Of course, the inclusion of conditionals and flow control into action code makes this process much harder to do automatically. Ramifications, probably, cannot be generated automatically.

8.1.5 Calibration

The principle of calibrator variables is that variables can be coded into an action which are simply placeholders for immediate values assigned by the robot. These allow the effects of actions to be calibrated. For example, instead of having an action which always moves the robot one unit forward, you have an action which moves it a given number N units forward. The variable N is given a constant value by the robot when it executes the action.

In order for this to be useful, it is necessary for the robot to understand how change the value of N will change how the action is executed. The robot would do this teleologically – that is by understanding what value of N will take it to the situation it wants to be in (or as close to it as possible). In order to do this, it is necessary to produce an inverted form of the usual H(…,…) holds root function.
The inverted form I(…,…) is used to "trace-back" from the destination to the source. So instead of, for example, the current X and Y co-ordinates of the robot being fed into a H(…,…) lambda string; the desired X and Y co-ordinates are fed into an I(…, …) lambda string. This provides a value to assign to the natural variable N.

This approach could improve the elegance of plans by making the robot able to achieve a higher level of reasoning – not just understanding how it's actions form a response, but also how to get a specific response by choosing a specific action.

This is, without doubt, the greatest fundamental change that could be made to the overall design of the planning system. It would probably be a lot of extra work, and I am not convinced that it would produce better results in the real world. In contrived circumstances, though, it could definitely produce more *impressive* results.

8.2 The Robot's Intelligence

8.2.1 More Powerful Hardware

I have already criticized LeJos and the Lego Mindstorms system. I reiterate here that probably the most useful thing you could do to improve the overall system is find a better target for it. At the current moment in time, the Lego Mindstorms system is just not, in my opinion, a practical host for the planning system.

8.3 Planning

8.3.1 Experimentation

Obviously, if a working system was produced then a lot of experimentation could be done with the planning system. At current there is no opportunity to do this. I believe that the system has a lot of scope if the situation arose for it to prove it.

8.4 Integrated Development Environment

I have already talked about the usability problems of only having a CLI for use the Assembler et al. One very obvious enhancement is provide some kind of IDE for the system. My suggestion would be essentially a text editor, which we bring together all the various source files and automatically re-compile them where necessary.
An IDE could also provide some RAD features for the creation of maps which could be tested in the Simulator and goal files could be built automatically from these maps.

9 Background Research

So far, and it is expected will continue in the future, the research has been web-oriented. The reason for this is that AI, and computing in general, is a topic in which there is a lot of information available on the World Wide Web. This is verbose, easily accessed, and can be done while working on the project, so it is a very useful way to research.

Again, due to the investigative nature of the project, research will generally focus on theoretical concepts, rather than practical (such as heuristics). The main topics of research are thus:

9.1 Previous Projects

As the nature of this project is very different from the one previously done on Lego Mindstorms, these projects are not especially useful. However, they were helpful in gaining an initial familiarity with the RCX system.

9.2 Lego Mindstorms

The research into this area has principally involved reading the leaflets, and using the software, provided with the Lego Mindstorms package. This has taught me how to use Lego Mindstorms robots at a basic level.
Further research had to be done in to the different ways in which Lego Mindstorms robots can be programmed. These include:

- the graphical tool provided with the Lego package
- the LeJos java compiler [1]
- the LegOS C/C++ compiler [2]
- Legolog prolog-style language [3]

9.3 Java

As I have decided to use a Java compiler to develop the robot's intelligence, there will be some need to research on Java topics. Principal of these is the LeJos API [4].

9.4 Set Theory

As this is a fairly simple area, the only necessary research was a refreshing of mathematics done in the first year in MA11.

9.5 Lambda Calculus

Lambda calculus will be used for specifying the nature of facts, as they are known by the robot. I have chosen to do this for two reasons – firstly because it provides a universal[1] way of expressing the effects of an action, and secondly because it's syntax is similar to situation calculus (and therefore the two can be combined).

As well as utilizing the theory learnt in SE23, I will read up on the more technical aspects (such as the context-free grammar).

Wikipedia has a vast collection of information on lambda calculus [5], as well as related topics such as SKI combinator calculus [6] and context-free grammars [7].

9.6 Situation Calculus

Situation calculus is central to the overall theme of the project – giving the robot a degree of consciousness. Although it is not the most important aspect of implementation, it directs how the robot will behave.

[1] Meaning that it can be used to express any computable algorithm.

The Stanford Formal Reasoning Group provides a good introduction to artificial intelligence [8]. This provides a wealth of knowledge about situation calculus, and a great deal of other topics similar in the high-level implementation of robot behaviour.
The Formal Reasoning Group is a collection of people who focus on logic-based AI, and building systems of reasoning. Over the fifty years since it's creation it has produced a vast amount of written work on many areas in this field.

Foundations for the Situation Calculus(Authors: Hector Levesque, Fiora Pirri, Ray Reiter; Publisher: Linköping Univesity Electronic Press) is also a great source of information. Particularly it's provision of mathematical description.

9.7 References

I do not have a lot of references, as I am not the type of person who likes to read a lot of background material. I am the type of person who runs of into a fantastic world of ideals I cannot live up to.

The references are:

[1] http://lejos.sourceforge.net

[2] http://www.noga.de/legOS

[3] http://www.cs.toronto.edu/cogrobo/Legolog

[4] http://lejos.sourceforge.net/apidocs/index.html.

[5] http://en.wikipedia.org/wiki/Lambda_calculus

[6] http://en.wikipedia.org/wiki/SKI_combinator_calculus

[7] http://en.wikipedia.org/wiki/Context-free_grammar

[8] http://www-formal.stanford.edu/

Appendix A: Personal Evaluation

Firstly I must say that despite the numerous problems I had throughout the project, I did enjoy it mostly. There is no doubt that developing in the LeJos system was extremely tedious and difficult, and I found this very hard to do. From the very beginning I definitely underestimated the difficulty of working with such an embedded system.

Another major stumbling block in the development of the system was my perpetual inability to start working at anything. My complete lack of an apparent work-ethic has always been a problem for me. This, probably, is largely because of my lack of underlying enthusiasm for my course, and University education in general. I am, alas, a victim of my own intelligence.

With regards the Assembler and Compiler, I am happy with the work I have produced. This is not the first time I have ever built an Assembler or Interpreter for a bytecode; but it is the first time I've ever built anything like a Compiler and I feel that it was a learning experience.

In conclusion, when one completes a journey it is not what has been traveled that matters, but how the travel has affected us. And I can say without equivocation that I am a different person after finishing this project, and I have learnt a great many things as a result of it.

Appendix B: Instruction Assembly Language

The following is the opcode list for the assembly language used to produce functions. Instructions are structured in the following order:

[Conditional Modifier] Opcode [Destination Register][Source Registers, …] [Address]

Addresses are stored with the least-significant byte first.

The opcodes are as follows:

Hex Code	Pneumonic	Description
00	nop	No-operation is performed.
01	loop	Decrement r0 and jump on non-zero.
02	jump	Jump to given address.
03	jumpc	Jump to given address if condition is true.
04	call	Push IP to stack, then jump to address.
05	callc	Call given address if condition is true.
06	ret	Pop stack into IP.
07	push	Push register value onto stack.
08	pop	Pop stack into register.
09	wait	Suspend interpretation for some milliseconds.
0A	interrupt	Switch on interrupts.
0B	reti	Return from interrupt hook.
10	load	Load parameter register into temporary register.
11	save	Save immediate value into temporary register.
12	inc	Increment register by 1.
13	dec	Decrement register by 1.
14	add	Add two registers together, store result in a third.

15	sub	Subtract two regs. from each other, store result in third.
16	mul	Multiply two regs. together, store result in a third.
17	div	Divide one reg. by another, store result in a third.
18	copy	Copy one register into another.
19	retp	Return with parameter[2].
1A	cpl	Ones-complement register.
1B	neg	Negate register.
20	power	Set power of a motor.
21	onfwd	Switch a motor on forwards.
22	onbwd	Switch a motor on backwards.
23	off	Switch a motor off.
28	read	Read input from a sensor.
29	setin	Set the type of input to expect from a sensor.

The condition modifier is a single byte:

Hex Code	**Pneumonic**	**Description**
30	z	On zero.
31	c	On carry,
32	p	On plus (positive).
33	m	On minus (negative).
38	nz	On non-zero.
39	nc	On non-carry.
3A	np	On non-plus.
3B	nm	On non-minus.

[2] This opcode returns from the interpreter, passing the value stored in the given register as the return result to the function. All functions in the planning system must complete with this operation.

Appendix C – Some Example Plans

Simple Plan Used to Test the Compiler's Function

```
#
# Test planning source file.
#

# Moves forward one unit.
ACTION_START: move_forward_one

#  List of pre-conditions.
H(started):

#  List of post-conditions.
H(X', add(X, mul(10, cos(R))));
H(Y', add(Y, mul(10, sin(R)))):

#  List of ramifications.
H(bumped');
N(bumped'):

forward(10, 1)?0,1;
turn(90):

ACTION_END.

# Moves backward one unit.
ACTION_START: move_backward_one

:

H(X', sub(X, mul(10, cos(R))));
H(Y', sub(Y, mul(10, sin(R)))):

:

backward(10, 1):
```

```
ACTION_END.

#
```

Plan Used to Show the Use of Boolean State Variables

to Create an Initializer Function for the Plan

```
# state1
#
# An example of the use of state variables
# to create an initializer function.
#
# (Code based on test.pl)
#

# Initializer function.
#
# Initializes X and Y values for cartesian
# tracking of robot position.
ACTION_START: init

N(inited):

H(inited):

H(X', 10);
H(Y', 20):

ACTION_END.

# Moves forward one unit.
ACTION_START: move_forward_one

#  List of pre-conditions.
H(inited):

#  List of post-conditions.
H(X', add(X, mul(10, cos(R))));
```

```
H(Y', add(Y, mul(10, sin(R)))):

#  List of ramifications.
H(bumped');
N(bumped'):

forward(10, 1)?0,1;
turn(90):

ACTION_END.

# Moves backward one unit.
ACTION_START: move_backward_one

H(inited):

H(X', sub(X, mul(10, cos(R))));
H(Y', sub(Y, mul(10, sin(R)))):

:

backward(10, 1):

ACTION_END.

#
```

Example Goal File for a Simple Navigation Task

```
# Robot must get to (-402, -104).
# The robot's start position is taken as (0, 0).
H(X, -402);
H(Y, -104):
```

Appendix D – Tools' User Manual

Overview

The tools for developing plans are divided into an Instruction Assembler, a Planning Compiler, and a Simulator. There are also some associated file checking utilities to ensure that the produce object files are accurate.

The Instruction Assembler

The Assembler can be found in the *asm* directory. To use the Assembler simply place the *asm.exe* file in your directory of choice, and execute it from the command line. The Assembler takes two parameters: firstly the name of the file you wish to assemble, secondly the name to give the produce object file.

```
asm.exe <input file> <output file>
```

Examples of assembly source can be found in the *asm* directory. The syntax for the language should not be unfamiliar to those who know assembly. If you are not familiar with assembly language, it is advised that you read up on such concepts before attempting to use the Instruction Assembler.
A complete list of instructions can be found in the document *bytecode.rtf* in the *asm* directory.

The Planning Compiler

The Compiler can be found in the *compiler* directory. To use the Planning Compiler simply place the *pc.exe* file in your directory of choice, and execute it from the command line. The Compiler takes three parameters: firstly the input file, then the instruction firmware file, then finally the output file.

```
pc.exe <input file> <firmware file> <output file>
```

To use the Goal Compiler simply place the *gc.exe* file in your directory of choice, and execute it from the command line. The Goal Compiler is invoked in the same way as the planning compiler.

The Simulator

The Simulator is a Windows program and therefore can simply be double-clicked on in Explorer. The Simulator can be found in the *simulator* directory. The Simulator retrieves it's information from a configuration file called *config.txt*.
The configuration file has is a simple text file which has the following format:

```
<name of bitmap file>
<start x position for robot>
<start y position for robot>
<name of instruction firmware file>
<name of plan file>
<name of goals file>
```

To change where the Simulator gets it's source from simply change the values written here.