Implementing the Object Spaces model of Algol

by

Andrew J Wallace

A dissertation submitted in partial fulfillment for the degree of Bsc (Hons) Computer Science with Maths

in the

Department of Computer Science

May 2010
Declaration of Authorship

Implementing the Object Spaces model of Algol

Submitted by: Andrew Wallace

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognize that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: __________________________________________________________________________

Date: __________________________________________________________________________

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to the libraries for the purposes of consultation.
Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors

Charles Antony Richard Hoare
People describe semantics of imperative programming languages in terms of functions on some global state. Reddy proposes a new approach for representing these languages in terms of observable behavior. In this approach states are regarded as part of the internal structure of objects and are not viewed when talking about observable behavior. This model is called the object spaces model and this dissertation discusses a tool to generate object based semantics for an imperative language based on syntactic control of interference.
Acknowledgements

I would like to thank the following people for helping me produce this dissertation:

Guy McCusker
Lilla Maniscalco
William Lee
# Contents

Declaration of Authorship i  
Abstract iii  
Acknowledgements iv  
List of Figures ix  
List of Tables x  

1 Introduction 1  
1.1 Problem Description ............................................. 1  
1.2 Aims ........................................................................... 1  
1.3 Document Structure .................................................. 2  

2 Literature Survey 3  
2.1 Introduction ................................................................. 3  
2.2 Algol ........................................................................... 3  
2.3 Background .................................................................... 4  
2.4 Idealized Algol ............................................................... 5  
2.5 Syntactic Control of Interference ...................................... 5  
2.6 Abstraction .................................................................... 7  
2.7 Game Semantics Model .................................................. 8  
  2.7.1 Introduction ............................................................. 8  
2.8 The Object Spaces Model ................................................ 10  
  2.8.1 Introduction ............................................................. 10  
  2.8.2 Events ..................................................................... 10  
  2.8.3 Categorical Structure ................................................. 11  
  2.8.4 Integer Expressions .................................................. 11  
  2.8.5 Commands ............................................................... 11  
  2.8.6 Semantics ............................................................... 12  
  2.8.7 Examples ............................................................... 13  
2.9 Conclusion ..................................................................... 14
3 Requirements and Objectives 15
  3.1 Introduction .................................................. 15
  3.2 Functional Requirements ...................................... 15
  3.3 Non-Functional Requirements ................................. 16
  3.4 Objectives ..................................................... 17

4 Design 19
  4.1 Introduction .................................................. 19
  4.2 Designing the Language ....................................... 20
  4.3 Understanding the Language ................................... 22
  4.4 The Internal Representation of the Language ............... 22
  4.5 The Internal Representation of the Semantics .............. 23
  4.6 Generate Semantics From Language ........................... 23

5 Implementation 26
  5.1 Introduction .................................................. 26
  5.2 Implementing the Language .................................... 26
    5.2.1 Basic Language Features ................................ 26
    5.2.2 'New' Statement ........................................ 31
    5.2.3 While Loop ............................................. 31
    5.2.4 Procedures ............................................. 32
  5.3 Internal Representation of the Language .................... 34
    5.3.1 Main Body of Code ...................................... 34
    5.3.2 Declarations ........................................... 35
  5.4 Representation of the Semantics .............................. 37
  5.5 From Language to Semantics .................................. 38
    5.5.1 Introduction ............................................ 38
    5.5.2 Basic Transformation .................................... 38
    5.5.3 New Statement ........................................... 40
    5.5.4 While Loop ............................................. 41
    5.5.5 Procedures ............................................. 44
      5.5.5.1 Procedure Declaration ............................. 45
    5.5.6 Procedure Application .................................. 46
  5.6 Interface .................................................... 48

6 Testing and Experimentation 49
  6.1 Introduction .................................................. 49
  6.2 Functional Tests ............................................. 49
    6.2.1 Test1 and Test2 ........................................ 49
    6.2.2 Test3 ................................................... 50
    6.2.3 Test4 ................................................... 51
    6.2.4 Test5 ................................................... 53
    6.2.5 Test6 ................................................... 53
    6.2.6 Test7 ................................................... 56
  6.3 Non-Functional Tests ......................................... 59
    6.3.1 Test1 .................................................. 59
    6.3.2 Test2 .................................................. 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.6.5</td>
<td>Test : Finding Factorial Numbers</td>
<td>81</td>
</tr>
<tr>
<td>A.6.6</td>
<td>Results</td>
<td>81</td>
</tr>
<tr>
<td>A.6.7</td>
<td>Test : Finding Fibonacci Numbers</td>
<td>82</td>
</tr>
<tr>
<td>A.6.8</td>
<td>Results</td>
<td>82</td>
</tr>
<tr>
<td>A.6.9</td>
<td>Test : Bubble Sort</td>
<td>82</td>
</tr>
<tr>
<td>A.6.10</td>
<td>Results</td>
<td>84</td>
</tr>
<tr>
<td>A.6.11</td>
<td>Test : Finding Larger Fibonacci Numbers</td>
<td>84</td>
</tr>
<tr>
<td>A.6.12</td>
<td>Results</td>
<td>85</td>
</tr>
<tr>
<td>A.7</td>
<td>Test7 : Working with procedures'</td>
<td>85</td>
</tr>
<tr>
<td>A.7.1</td>
<td>Test : Simple Procedure</td>
<td>85</td>
</tr>
<tr>
<td>A.7.2</td>
<td>Results</td>
<td>86</td>
</tr>
<tr>
<td>A.7.3</td>
<td>Test : Swap Procedure</td>
<td>86</td>
</tr>
<tr>
<td>A.7.4</td>
<td>Results</td>
<td>86</td>
</tr>
<tr>
<td>A.7.5</td>
<td>Test : Bubble Sort Procedure</td>
<td>87</td>
</tr>
<tr>
<td>A.7.6</td>
<td>Results</td>
<td>88</td>
</tr>
<tr>
<td>A.7.7</td>
<td>Test : Bubble Sort with nested swap procedure</td>
<td>89</td>
</tr>
<tr>
<td>A.7.8</td>
<td>Results</td>
<td>90</td>
</tr>
<tr>
<td>B</td>
<td>Non-Functional Tests</td>
<td>91</td>
</tr>
<tr>
<td>B.1</td>
<td>Test1 : Performance Test</td>
<td>91</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Test : While Loop</td>
<td>91</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Results</td>
<td>91</td>
</tr>
<tr>
<td>B.1.3</td>
<td>Test : Factorial Program</td>
<td>92</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Results</td>
<td>92</td>
</tr>
<tr>
<td>B.1.5</td>
<td>Test : Swap Program</td>
<td>92</td>
</tr>
<tr>
<td>B.1.6</td>
<td>Results</td>
<td>93</td>
</tr>
<tr>
<td>B.2</td>
<td>Test2 : User Test</td>
<td>93</td>
</tr>
<tr>
<td>B.2.1</td>
<td>User Test</td>
<td>94</td>
</tr>
<tr>
<td>B.2.2</td>
<td>User Study</td>
<td>94</td>
</tr>
<tr>
<td>B.3</td>
<td>Results</td>
<td>94</td>
</tr>
<tr>
<td>C</td>
<td>Experimental Tests</td>
<td>95</td>
</tr>
<tr>
<td>C.1</td>
<td>Test1 : Variable Behavior</td>
<td>95</td>
</tr>
<tr>
<td>C.1.1</td>
<td>Test : Factorial Function</td>
<td>95</td>
</tr>
<tr>
<td>C.1.2</td>
<td>Results</td>
<td>96</td>
</tr>
<tr>
<td>C.1.3</td>
<td>Test : Fibonacci Function</td>
<td>96</td>
</tr>
<tr>
<td>C.1.4</td>
<td>Results</td>
<td>97</td>
</tr>
<tr>
<td>C.2</td>
<td>Test2 : Looping</td>
<td>98</td>
</tr>
<tr>
<td>C.2.1</td>
<td>Test : No Incrementation</td>
<td>98</td>
</tr>
<tr>
<td>C.2.2</td>
<td>Results</td>
<td>98</td>
</tr>
<tr>
<td>C.3</td>
<td>Test3 : Sorting Comparison</td>
<td>99</td>
</tr>
<tr>
<td>C.4</td>
<td>Test : Two Sorting Algorithms</td>
<td>99</td>
</tr>
<tr>
<td>C.4.1</td>
<td>Results</td>
<td>102</td>
</tr>
</tbody>
</table>

Bibliography 104
List of Figures

4.1 High-Level View Of The System ........................................... 19
4.2 Diagram To Show How Input Code is Understood .......................... 22
4.3 Interpreter Design Pattern .................................................. 23
4.4 Semantics of a Program Phrase ............................................. 24
4.5 Semantics of a Program ....................................................... 24
4.6 Overall System ................................................................. 25

5.1 Expression Layout ............................................................. 28
5.2 Command Layout ............................................................... 28
5.3 Assignment Layout ............................................................. 29
5.4 IF-Statement Layout ............................................................ 29
5.5 Grammatical layout of 5.3 ...................................................... 30
5.6 New Layout .......................................................... 31
5.7 While Layout ................................................................. 32
5.8 Using the Interpreter Pattern ................................................ 35
5.9 Segment Class Structure ..................................................... 37
5.10 Generic Language to Semantics Transformation .......................... 38
5.11 Evaluator Class Structures .................................................. 39
5.12 While-loop Algorithm ......................................................... 44
5.13 Procedure Declaration ....................................................... 45
5.14 Procedure Application ....................................................... 46
5.15 Graphical User Interface ..................................................... 48

6.1 A graph to show the relation between number system size and computa-
tion time ................................................................. 60

B.1 A graph to show the relation between number system size and computa-
tion time ................................................................. 92
B.2 A graph to show the relation between number system size and computa-
tion time ................................................................. 93
List of Tables

2.1  Player Interaction of (2.8) .......................... 8
To my Lilla...
Chapter 1

Introduction

1.1 Problem Description

This project is based on the study of logic and semantics of programming languages. In particular it studies the theoretical language of syntactic control of interference (SCI) [1] together with a model based upon SCI called the object spaces model [2]. This model represents programming languages in terms of observable behavior rather than functions on a global state. This means programs can be described with many variable sequences which represent the behaviors in those variables. The syntactic control of interference language is designed to keep variables distinct in particular contexts so that interference or aliasing cannot occur. The object spaces model records events in each variable but does not record the relative order of events in distinct variables. This is okay thanks to the non interference aspect. It is possible to implement the model in order to generate semantics for programs which adhere to the SCI specification. This tool can also be used to teach people about the model.

1.2 Aims

The aims for this project are the following:

- Implement a language based on SCI
- Create a tool to generate the object based semantics of this language
- Build on the tool to make it instructional so that it can be used to teach people
Chapter 1. *Introduction*

The aims make it clear that the project has two main deliverables. The first is a specification and implementation of a SCI language. The second is the implementation of a tool capable of transforming this language into object based semantics.

### 1.3 Document Structure

The document will begin by reviewing the literature for this topic. This will involve a study of SCI and the object spaces model together with some additional related work. The chapter after that will describe the requirements and objectives for this project discussing the required functionality of the tool together with some objectives we aim to accomplish with the tool. Then the document will focus on the design, implementation, and testing of the language and tool. The final chapters will evaluate the tool and draw conclusions on the project as a whole. The document will contain an appendix at the back which will contain information discussed in the main chapters.
Chapter 2

Literature Survey

2.1 Introduction

In this dissertation I intend to build a system which reflects the Object Spaces Model proposed by Reddy in his paper [2]. In reviewing the associated literature it is important to highlight some key topics which have relevance to this task. The clearest direction to begin discussing the subject area is to discuss ALGOL, in particular ALGOL 60 and why this is used as a prototypical language for studying semantics. I will then give an overview of higher order functions describing what a 2nd order and 3rd order function looks like. I will give a brief introduction to Reynolds’s idealized ALGOL describing the main mechanics of the theoretical language. I will then move on to talk about the topics of interference [1, 3] and abstraction. Then I will discuss the Game Semantics Model [4] and highlight some particular implementations such as HOMER [5] before I finally discuss the Object Spaces Model in some depth.

2.2 Algol

In 1959, John Backus outlined a language in his paper [6] which eventually became known as ALGOL 60. ALGOL 60 is a language described by Hoar as a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors. This statement should help the reader understand why ALGOL 60 is still a subject for many theoretical computer scientists. It is a safe, clean language to which elegant mathematical models can be formed to discuss various topics within semantics of programming languages. ALGOL 60 was the first formally written language and was also the first language to use a context free grammar named a BNF grammar after Backus and Naur.
\[ \text{exp} := n \mid \text{exp} + \text{exp} \] (2.1)

In (2.1) I show how a BNF grammar is formed, the symbol on the left can be replaced by one of the production rules on the right. A particular example of this can be seen with \((3 + ((4 + 5) + 6))\) which demonstrates how an expression can simply return a number \(n\) or can be a recursive call to another two expressions. It is safe to conclude that a language like ALGOL 60 is relatively clean to work with when one would like to construct mathematical models to convey particular programming constructs in a semantic context.

### 2.3 Background

In functional programming languages such as LISP there is frequent use of higher order functions. A higher order function is a function which will either take one or more functions as input or output functions or both. An easy example to illustrate this concept would be the derivative in mathematical analysis because this maps a function to another function shown in (2.2).

\[ f'(x) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h} \] (2.2)

Another example from computer science is the \textit{map} function included in many functional languages. This function will take a function argument \(f\) together with a list and then will return a list with \(f\) applied to all of the elements. To further illustrate the higher order function we can use lambda calculus to create a constant function as shown in (2.3)

\[ \lambda x. y \] (2.3)

(2.3) is a constant function because we can provide any argument and it will alway return \(y\). Higher order functions are not only in functional programming languages but can also appear in imperative programming languages and object orientated languages. A simple example for this is if you have some data structure written in Java and you have a comparison function set as a parameter to the constructor. These are all examples of 2nd order functions. It is also possible to create 3rd order functions. 3rd order functions are higher order functions which take 2nd order functions as input. In the discussion which follows I will eventually talk about HOMER which is an implementation
of the Game Semantics Model which is fully capable of dealing with 3rd order functions. This dissertation will be covering an implementation capable of dealing with 2nd order functions only.

2.4 Idealized Algol

In [7], Reynolds proposed a theoretical language which is a higher order procedural language and consists of the following:

- The language of while-programs:
  
  \[
  y := 0; \\
  \text{while } y < 10 \text{ do} \\
  x := y + 1; \\
  y := y + 1;
  \]

- The simply-typed $\lambda$-calculus

- local variables
  
  \[\text{new } x \text{ in } C\]

It is possible to construct a BNF grammar which can accurately implement the above for example:

\[
\text{newstatement} = \text{'new'} \text{ IDENT in (statement)+} \\
\text{where} \\
\text{statement} = \text{newstatement} — \text{Ifstatement} — \text{whilestatement}
\]

The model to be implemented for this dissertation will be based on the idealized Algol subset of Algol and in particular will focus on the 2nd order fragment.

2.5 Syntactic Control of Interference

In [1] Reynolds describes interference as a phenomenon where distinct identifiers can represent data structures which share storage or procedures with interfering side effects. The simplest form of this phenomenon is aliasing which is when two variables share some data or point to the same piece of information. I will quickly demonstrate this with an example of two variables which are alias’s of each other.
Chapter 2. Literature Survey

Class Var {

    int v;
    void set(int v) {
        this.v = v;
    }
    int get() {
        return v;
    }
}

Var x = new Var();
Var y = x;

y.set(3);
x.set(4);

y.get();

Listing 2.1: Example Code to demonstrate the effects of Aliasing

In the above code x and y are aliases of each other and so when one calls y.get(); the answer will give 4 because y points to the Var object x. Another example can be seen when the context of a λ-application includes variables which are in the context of a λ-abstraction as shown in (2.4).

\[ y : \text{var} \vdash (\lambda x.y = 3; x = 4 \text{ if}(y == 4 \text{ do} M))y \]  

(2.4)

This function has the potential for aliasing because by β reduction we swap all x’s with y’s and therefore this results in y == 4 evaluating to true when this was not intended. This shows how aliasing can seriously change the meaning of a program. In this case M will be executed in circumstances which are not supposed to be.

The Syntactic Control of Interference (SCI) language is a restricted version of Idealized Algol. The types used for this language are:

\[ A ::= \text{nat}|\text{var}|\text{comm}|A \rightarrow A \]  

(2.5)

For this dissertation nat might well be replaced with int. This is not seen as any great change and is simply a choice made by me.
The approach put forward by Reynolds is to restrict applications $MN$ to where $M$ and $N$ have no variables in common. In [1] Reynolds formally describes a phrase to mean a variable, expression, statement, or procedure denotation and he describes $M \# N$ to indicate that it is syntactically detectable $M$ and $N$ do not interfere, where $M, N$ are phrases. This interference control can be more accurately shown in 2.6

\[
\Gamma \vdash M : A \rightarrow B \Delta \vdash N : A \\
\Gamma, \Delta \vdash MN : B
\]  
(2.6)

In 2.6 the two contexts $\Gamma$ and $\Delta$ must be disjoint. This simply means that the arguments presented to a procedure must not include variables which are included inside the procedure. It is worth noting here that this does not mean variables are used at most once, sequential composition is as you would expect 2.7

\[
\Gamma \vdash M : \text{comm}\Gamma \vdash N : \text{comm} \\
\Gamma \vdash M; N : \text{comm}
\]  
(2.7)

### 2.6 Abstraction

Denotational semantics is the formalization of the meaning of programming languages. This is done by constructing mathematical objects or denotations which can be used to describe the meanings of expressions from languages. This allows us to map programs into some mathematical structure which can then be analyzed. It is important to understand certain properties of denotational semantics when one is using semantic models to describe programming languages. These properties are often looked for when a model is being constructed. The first to note is the act of having syntax independence which means the denotations of programs should not involve the syntax of the source language. Another property is soundness which means all observable different programs should have different denotations. A fully abstract model is one where equivalent terms in a program have equal denotations in the model, they are observationally equivalent. If a model is decidable it means that there is an algorithm for saying if two programs have the same denotation in the model.
2.7 Game Semantics Model

2.7.1 Introduction

The Game Semantics Model [4] is a fully abstract model where the computation is modeled via an interaction between two players. If $A$ is a game between two players $P$ and $Q$, $Q$ represents the environment where the program runs and $P$ represents the program. Then we describe a strategy for $P$ to play in the game $A$. A strategy is a set which forms a play book describing how $P$ should respond when he should play. The play is a sequence of moves which includes a pointer to a move from earlier. A game is said to run to completion when every play has been completed. It is possible to observe equivalence in this model by comparing two complete plays in a particular strategy. I shall briefly describe this player interaction by using the game semantics to model (2.8)

\[ v: \text{var} - x := !x + 2 \quad (2.8) \]

<table>
<thead>
<tr>
<th>O: run</th>
<th>(get the value from x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P: read(\langle x\rangle)</td>
<td>(O supplies the value 3)</td>
</tr>
<tr>
<td>P: write(5)(\langle x\rangle)</td>
<td>(write 5 into x)</td>
</tr>
<tr>
<td>O: ok(\langle x\rangle)</td>
<td>(the assignment is complete)</td>
</tr>
<tr>
<td>P: done</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Player Interaction of (2.8)

The table above (2.1) shows a game view of the processes involved in computing (2.8) where the strategies for both $P$ and $Q$ are displayed along side the player declarations for example $\text{read}\langle x\rangle$. The play sequence can be viewed as $(\text{run}.\text{read}(n).3.\text{write}(5).\text{ok}.\text{done})$ where all plays are on the variable $x$ apart from $\text{run}$ and $\text{done}$.

This model relates to the Object Spaces Model as it is based on observable behavior rather than state changes. It is also fully abstract and is based on Reynold’s Idealized ALGOL. The play sequences are very similar to the variable event sequences in the Object Spaces Model.

During my research I encountered three implementations of the Game Semantics Model which include HOMER, MAGE, and GAME CHECKER. I will briefly outline the main ideas involved in each and the usefulness obtained from such implementations. This should help further understand the potential advantages of implementing the Object Spaces model.
HOMER [5] is a tool which checks observational equivalence of order 3 terms. It uses
the visibly pushdown automata rather than a finite automata which is more expressive
moreover VPA is decidable. The main testing for the model has been based on comparing
sorting algorithms. It is clear that although sorting algorithms can differ in design and
complexity they are essentially the same when viewed semantically. HOMER correctly
identifies the equivalence of the bubble sort algorithm along with any other sorting
algorithm.

MAGE [8] is based on the Game Semantics Model and is intended as a further improve-
ment to another implementation called GAME CHECKER. This implementation unlike
HOMER can only check for equivalence of order 2 terms.
2.8 The Object Spaces Model

2.8.1 Introduction

In [2] Reddy describes a model of syntactic control of interference which he calls Object Spaces. The computation for this model is understood through a sequence of observable events. A program is interpreted as a relation between these sequences. This model is a precursor of the game based models similar to the Game Semantics Model discussed earlier.

2.8.2 Events

In [9] McCusker illustrates a type described in Reddy’s model [2] as a set containing observable events at that type. The semantics of types can be described in the following:

\[
\begin{align*}
\hat{\text{int}} & = \mathbb{Z} \\
\hat{\text{comm}} & = \{\ast\} \\
\hat{\text{var}} & = \{\text{read}(n), \text{write}(n) | n \in \mathbb{N}\} \\
\hat{A \rightarrow B} & = [A]^* \times [B]
\end{align*}
\]

In the above it is easy to see that an event of type \textit{int} will be the action of evaluation returning an answer of some \(x \in \mathbb{Z}\).

The \textit{comm} event will always perform the same action which is termination \(\ast\). In [9] Guy describes \textit{comm} as a singleton set which is a set with cardinality 1 i.e in this case the element \(\ast\).

The variables will be of the form \(x, y, z, \ldots\) so the variables can be considered as some \(k \in E^*\) where \(E\) is an alphabet. It is possible to treat variables as objects and discuss potential events which relate to these variables.

Two events can be observed:

\[
\begin{align*}
\text{read}(n) & \quad (2.9) \\
\text{write}(n) & \quad (2.10)
\end{align*}
\]

The action (2.9) will simply read an integer \(n\) out of some variable and action (2.10) will assign an integer value \(n\) to a variable where \(n \in \mathbb{Z}\). As an example of how multiple actions can be expressed I will use (2.8) from earlier. I will conveniently describe the
sequence of actions corresponding to a language of only two variables $x$ and $y$ (In actual fact in this case we are only talking about $x$)

$$(\text{read}(n).\text{write}(n + 2), \epsilon, *)$$

(2.11)

In (2.11) it is clear to see that we model the events of $\text{read}(n).\text{write}(n + 2)$ for $x$ which correspond to $x :=!x + 2$ in (2.8) and we produce an empty event $\epsilon$ for the variable $y$ followed by a $*$ because the whole sequence is a $\text{comm}$ which terminates. It is important to note that this model will only show the observable behavior in each variable and not interactions between variables. This is why the Object Spaces model will only model Syntactic Control of Interference subset of the Idealized Algol language.

### 2.8.3 Categorical Structure

In ([10]) McCusker outlines a categorical structure for the object spaces model. The category is named $\text{MonRel}$ which has objects as monoids and the morphisms $\text{hom}_{\text{MonRel}}(A, B)$ are defined as relations between $A$ and $B$. In ([10]) McCusker shows that $\text{MonRel}$ is isomorphic to $(\text{Mon})^{op}$ and therefore uses this to translate categorical structure from $\text{Mon}$ to $\text{MonRel}$. This allows products in $\text{Mon}$ to give monoidal structure on $\text{MonRel}$, coproducts in $\text{Mon}$ give products in $\text{MonRel}$ and exponentials $A - B$ exist in $\text{MonRel}$ for all kleene monoids $B$. This categorical structure supports the object spaces model.

### 2.8.4 Integer Expressions

In these two sections I will describe the syntax involved in the language so that I can then discuss the semantics using the Object Spaces Model.

- $1, 2, 3, ... n$. Constants
- $M + N$ Arithmetic
- $!x$ Look Up Variables

### 2.8.5 Commands

- $x := M$, where $M$ is an Integer Expression and $:$ is assignment.
- $C_1; C_2$, where $C_i$ are Commands and $;$ defines sequence composition do left hand side followed by right.
• \texttt{if zero}M\texttt{ then }C_1\texttt{ else }C_2\texttt{, if the Integer Expression }M\texttt{ evaluates to 0 do }C_1\texttt{ if not do }C_2.

• \texttt{while if not }0\texttt{ }M\texttt{ do }C

### 2.8.6 Semantics

The statements written in the previous section can be described as program phrases. These phrases can be thought of as tuples of the form \((S_1, S_2, \ldots, S_n, a)\) where \(S_i\) with \(i \in \mathbb{N}\) is a sequence of variable events such as that shown in (2.9) and (2.10) and \(a\) is an action of the appropriate kind depending on the program phrase.

In this subsection I will describe the meaning of each program phrase using the system described above. The notation used for describing the semantics of will be \([P]_E\) where \([\ ]\) represent the semantics of, \(P\) stands for Program Phrase, and \(E\) is the language used for the variables. It is possible to not include \(E\) where it is obvious what is involved.

In the following we will assume \(E = \{x, y, z\}\)

\[
[n] = (\epsilon, \epsilon, \epsilon, n) \tag{2.12}
\]

\[
[M + N] = \{(s_1 t_1, s_2 t_2, s_3 t_3, m + n) | (s_1, s_2, s_3, m) \in [M] \land (t_1, t_2, t_3, n) \in [N]\} \tag{2.13}
\]

\[
[x] = (\text{read}(n), \epsilon, \epsilon, n) | n \in \mathbb{Z} \tag{2.14}
\]

\[
[y] = (\epsilon, \text{read}(n), \epsilon, n) | n \in \mathbb{Z} \tag{2.15}
\]

\[
[z] = (\epsilon, \epsilon, \text{read}(n), n) | n \in \mathbb{Z} \tag{2.16}
\]

\[
[x := M] = \{(s_1, \text{write}(m), s_2, s_3, *) | (s_1, s_2, s_3, m) \in [M]\} \tag{2.17}
\]

\[
[C_1; C_2] = \{(s_1 t_1, s_2 t_2, s_3 t_3, *) | (s_1, s_2, s_3, *) \in [C_1] \land (t_1, t_2, t_3, *) \in [C_2]\} \tag{2.18}
\]

\[
[\text{IF} \text{zero}M\text{then}C_1\text{ else }C_2] = \tag{2.19}
\]
(2.25) In (2.25) I show the semantics of (2.24) in the model.  
\[
\{ (s_1, t_1, s_2, t_2, s_3, t_3, *) | (s_1, s_2, s_3, 0) \in [M] \land (t_1, t_2, t_3, *) \in [C_1] \} 
\cup \{ (s_1, t_1, s_2, t_2, s_3, t_3, *) | (s_1, s_2, s_3, m) \in [M], m \neq 0 \land (t_1, t_2, t_3, *) \in [C_2] \} 
\]

\[
\| \text{whileNonzeroMdoC} \| = (2.20) 
\]

\[
\{ (s_1^1, t_1^1, \ldots, s^n_1, t^n_1, s_1, s_2, t_2, s_2, t_2, s_3, t_3, s_3, *) | (s_1^1, s_2^n, s_3^*), m) \in [M], m \neq 0 \land (t_1^n, t_2, t_3^*, *) \in [C] \} \cup \{ (s_1, s_2, s_3, *) | (s_1, s_2, s_3, 0) \in [M] \} 
\]

\[
\| \text{newxinM} \| = \{ (s_2, s_3, *) | (s_1, s_2, s_3) \in [M] \} \text{ and } s_1 \text{ has good variable behavior } \} (2.21) 
\]

\[
\| \lambda x.M \| = \{ (s_1, s_2, (s_3, *)) | (s_1, s_2) \in [M] \} \text{ and } s_3 \text{ is } x \text{ behavior } \} (2.22) 
\]

\[
\| M.N \| = \{ (s_1, s_2, s_3, *) | (s_1, s_2) \in [M] \} \text{ and } s_3 \in [N] \} (2.23) 
\]

### 2.8.7 Examples

In this section I will describe some basic examples to encourage understanding of the model and to illustrate some key concepts. I will provide an example of a simple swap program together with a modified swap program.

(2.24) is a simple program phrase which swaps the values contained in \(x\) and \(y\) using a temporary variable \(z\). In (2.25) I show the semantics of (2.24) in the model.

\[
\| \text{swap} \| = (\text{read}(n), \text{write}(m), \text{read}(m), \text{write}(p), \text{write}(n), \text{read}(p), *, | n, m, p \in \mathbb{Z}) \) (2.25) 
\]

In (2.25) the semantic definition is not enforcing typical behavior for a variable, so \(n\) and \(p\) do not have to equal. However if we look at \textit{new z in swap} we notice that the...
semantics will select the events in \( z \) when \( z \) behaves like a good variable and hence \( n = p \). This is shown in \( (2.26) \)

\[
\llbracket \text{new } z \ \text{in } \text{swap} \rrbracket = (\text{read}(n).\text{write}(m), \text{read}(m).\text{write}(n), \ast) | n, m \in \mathbb{Z}) \quad (2.26)
\]

\section*{2.9 Conclusion}

The object spaces model provides me with a fully abstract model which was proved by McCusker \([10]\). This means that if \( M \) is equivalent to \( M' \) where \( M, M' \) are program phrases then the semantics should also be equivalent. In other words full abstraction means two programs have the same denotations when they are observationally equivalent. It is worth noting that this contribution provides the additional converse argument to Reddy’s proof of soundness. In \([3]\) Laird proves that observational equivalence is decidable in finitely observable sequential SCI. It is important to note here that the model is only decidable for a restricted version of the language in which there is a finite data set instead of all of the integers. An instructional program can be built to analyze the model and would involve a finite data set. The program could be used to analyze the sequence steps of particular variables and also to show program traces. There are further potential possibilities in that a user could use the object spaces implementation to check or force a program to have all variable \textit{write()} actions before \textit{read()} actions. It could be used further to verify many properties of programs and in principle we could compare programs similar to HOMER\([5]\) due to the fact that the object spaces model is decidable.
Chapter 3

Requirements and Objectives

3.1 Introduction

In this chapter I will discuss the functional requirements for the program detailing some specific high level requirements I aim to achieve with the program. This project is based upon experimentation with a mathematical model and as such I feel that it is not important to discuss requirements in terms of program speed and usability as much as the other requirements. The program created for this project is to be used to facilitate the task of experimenting with the object spaces model discussed in chapter 2.

I will also discuss some non-functional requirements detailing the various quality attributes desired for the program including the look and feel, together with some usability and efficiency requirements.

This chapter is important because it gives me targets to aim for when I implement the program and will also help with evaluating the success of the project.

3.2 Functional Requirements

In this section I will briefly discuss the functional requirements of the program. This project involves two deliverables. The first is a specification of a programming language and the second is a piece of software which computes Reddy-style semantics of this language. In an effort to avoid confusion and to allow clarity I have decided to call any set of sequences ,as discussed in chapter 2, a segment.
1. The system should accept code written in a language based on SCI.

(a) The language should have the basic language of while programs and local
variable allocation
   i. Terms e.g Integers, variable lookup
   ii. Expressions
   iii. Commands
      A. Assignment statements
      B. New allocation of storage variables
      C. Logical statements
   iv. The language of while loops
(b) Version 2 should have basic procedures
(c) Version 3 should full higher-order procedures
(d) To ensure that the language is sufficiently powerful, we will require that code
can be written for:
   i. A program to find factorial numbers
   ii. A program to find a number in the Fibonacci sequence
   iii. Multiple sorting algorithms

2. The system should have a graphical user interface

(a) Import/Export code with easy to use file browser
(b) Text area for instant code writing
(c) Save feature to store results and code written
(d) Indication of execution
(e) Ability to analyze two programs simultaneously

3. The system should provide a list of printed segments for the semantic results of a
program of the form \((S_1, S_2, ..., S_n, a)\)

(a) The results should correctly represent all input code in the object spaces
model

3.3 Non-Functional Requirements

Further to the functional requirements I will describe some non functional requirements
for the system.
1. The system should be easy to use
   (a) The user should be able to learn the syntax for the language within 1 day
   (b) The user should be able to identify the semantic results and clearly recognize to which program they correspond to
   (c) The system should provide a help feature to support the user

2. The system should be portable and work on many operating systems

3. The system should be reliable and provide the required output on given input

4. Experiments and Tests carried out in the project should be replicable

5. The system should compute the semantics within 2 seconds
   (a) The system will be constrained on a finite number system chosen by the user. This is because the object spaces model is only decidable on finite data sets as discussed in chapter 2.

6. The system should display semantic results in an attractive way

\section*{3.4 Objectives}

In this project I will use an evolutionary development process and create several prototype implementations which will each be an improvement on the last. The process can be viewed as exploratory as well as throwaway prototyping as I will start by building the system with the parts which are understood well and then I will experiment with new features along the way as they become better understood.

The prototypes for this project will each involve having the language for the specific prototype as well as the internal implementation and correct semantic results. The prototypes can be broken down into stages as follows:

- **Basic Functionality**, This includes basic statements, variables, logicals in the language and their correct semantic representations
- **New Storage Allocation**, This is an improvement on the above adding new storage statements to the language and semantics
- **While Loop Functionality**, This is an improvement on the above adding while loops to the language and their semantic results
- **Basic Procedure Functionality**, Again an improvement on above with procedures capable of returning commands
• Full Lambda Calculus, An improvement to above by allowing function return types and function arguments

The Implementation will follow a pattern which adheres to the prototype stages. In the design chapter I will discuss the main functional parts of the system including the language, the internal representations of language and semantics, and the engine to change language into semantics. In the implementation chapter I will stick to structure of the design chapter but will break the sections down into subsections for each prototype implementation. An example of this format can be seen in the following:

• Implementation of the Language
  – Basic Syntax
  – ‘New’ Construct
  – While Loop
  – Procedures

This simple structure should help information flow through out the document specifically in the implementation section.
Chapter 4

Design

4.1 Introduction

The main objective of my project is to take input code which conforms to the theoretical SCI language and then provide a description of this code in the object spaces model. This chapter will focus on the high level design features for the whole system and will be split up according to language and semantics design.

In figure 4.1 I show the most basic view of the system. I intend to slowly provide more detail as we progress through the chapter.

In chapter 3 I outlined the objectives required for this project specifically the need to understand code written in the SCI language and representation of this language in the object spaces model. It seems logical to first address designing the language which hopes to represent the SCI language and then discuss how this language will be understood or parsed and lastly discuss the design ideas for representing this parsed language together with its semantics. In the following sections I will describe a high level design for the processes involved in reaching these objectives.
4.2 Designing the Language

The input code is based upon the Idealized Algol and Syntactic Control of Interference (SCI) proposed by Reynolds. This means that is should allow a user to write programs using while loops, the simply typed lambda calculus, and local allocation of storage variables as discussed in chapter 2. I decided to start each piece of code with a keyword `Program` followed by the name of the program and lastly a semicolon. Listing 4.1 shows how this looks.

```
1  Program  Test  {  
2       //  Code  
3  }  
```

Listing 4.1: Code to show "Program" Layout

I then created another keyword `begin` to indicate the beginning of the code which separates the code from variable declarations. This can be seen in listing 4.2.

```
1  Program  Test  {  
2       //  Declarations  
3        begin  
4       //  Main  Code  
5        end  
6  }  
```

Listing 4.2: Code To Show "begin" Layout

In the above you will notice the `end` keyword which is used to signify the end of the main code. The above setup has no relevance to the main objectives but is a design choice which should help meet the requirements of having a easy to use language and one which is comparable to common language designs.

In listing 4.3 I show how a user might write a common logical statement using the designed syntax.

```
1  Program  TestIF  {  
2        begin  
3            if  !x > 0  then  
4                y:=1;  
5            else  
6                y:=2;  
7            endif;  
```
I have tried to keep the syntax closely resembling that of Algol 60.
I have addressed the local allocation of storage variables with the following command structure.

```plaintext
Program TestNew {
    begin
        new z in
        z := !x;
        x := !y;
        y := !z;
        endnew
    end
}
```

**Listing 4.4: Code To Show How "new" is Used**

The following code snippet illustrates how a user might write a typical while loop in the language.

```plaintext
Program TestWhile {
    begin
        while !x < 2 do
            y := 1;
            x := !x + 1;
        endwhile;
    end
}
```

**Listing 4.5: Code To Show How "while" is Used**

The following code snippet illustrates how a user might write a typical procedure in the language.

```plaintext
Program TestProcedure {
    var : x ;
    num : 3 ;
    procedure foo () {
        x := !x + 1;
    }
    begin
```
In this example I have included some extra syntax for declaring variables. The variable \( x \) is declared with `var : x;` and a finite number system is chosen by the user with `num : n;`. This constraint was discussed in the non functional requirements in chapter 3.

### 4.3 Understanding the Language

In all of the prototypes the language will be understood the same way using lexer and a parser. I will do this using ANTLR. I will create a grammar in ANTLR and then generate a lexer which transforms the input text into tokens and a parser which attaches meaning to the tokens as described in the grammar. The process can be seen in figure 4.2

**Figure 4.2: Diagram To Show How Input Code is Understood**

![Diagram To Show How Input Code is Understood](image)

### 4.4 The Internal Representation of the Language

The language will be represented internally by a tree. The main idea is to keep the internal representation structured so that it accurately describes all of the input code written by the user. An example of this is an if statement should have a branch which connects to the representation of an expression together with two commands for true or false. It is possible at this stage to identify a design pattern which could provide the desired structure. This pattern is the the interpreter design pattern which will represent the order of the language with the lowest terms being represented by the leafs in the tree or terminal expressions. In this pattern everything in the language will be represented as an interpreter with program phrases which have children being non terminal and terms being terminal. Figure 4.3 shows the generic structure of this pattern.
4.5 The Internal Representation of the Semantics

To represent the semantics in the model I need to define a data structure which can capture all of the characteristics defined in the model. I decided to call the container which holds the variable sequences and the end sequence a segment. This allowed me to discuss these items with ease when talking with my supervisor.

The semantics described in the object spaces model involves multiple segments which each have multiple variable sequences, depending on the context, and one end event. In chapter 2 I discussed the semantics for many program phrases including logicals and loops. In this discussion it was apparent that some program phrases will be described semantically by a large collection of segments and not just 1. It is for this reason and the fact that no segments should be repeated that a good choice for representing the semantics would be a set of segments. Further to this it seems reasonable to represent the internal variable sequences as lists of events according to that variable sequence’s type. The end event can simply be some event from a collection depending on the particular program phrase. In 4.4 I have illustrated this semantic representation of a program phrase.

The semantics for a program will be all of the sets of segments which describe each program phrase in the program combined via the semantic definition which governs the relationship between those particular program phrases. This is illustrated in 4.5.

4.6 Generate Semantics From Language

The above design ideas allow me to consider an overall view of the system including subsystems as shown in figure 4.6. In 4.6 I have split the overall system into sub systems.
The raw code is parsed and an internal representation is created for the SCI language. The language representation will then be transformed into the semantic representation described via the engine system. The semantic representation will generate output which describes the input code in the object spaces model.

Following on from the discussion of the interpreter design pattern it seems natural at this point to have an operation inside each interpreter which will provide the semantics of each interpretation represented in the form described in the previous section. In the case of a non terminal interpreter the semantics of its children can be used to perform some computation which would then provide that interpreters semantics. In the case of a leaf/terminal interpreter there will be no children and so the semantic representation will be created dependent on the semantics of that term.
Figure 4.6: Overall System

Diagram showing the overall system with input, parser, language representation, semantics representation, engine, output, and interface.
Chapter 5

Implementation

5.1 Introduction

In this chapter I will follow the structure described in chapter 4. I will begin by discussing the implementation of the SCI language. I will then discuss the internal interpretation of the language followed by the interpretation of the semantics. Then I will discuss the transformation process from the internal structure of the language to the internal structure of the semantics. In these sections I will follow the evolution of the system by describing the specific sections with reference to the prototype development. Lastly I will discuss the interface implemented which allows a user to visually generate an object spaces interpretation from raw code.

I decided to use Java to implement the system. This is because I am familiar with Java and wanted to gain more experience using Java. It would have been possible to use other languages such as Lisp or Haskell, however I felt experience in a language of my choosing is much better for my future career. I used the Eclipse IDE to write the code for my project and I downloaded a ANTLR plug in to help create a lexical analyzer and a parser.

5.2 Implementing the Language

5.2.1 Basic Language Features

The basic functionality consists of all of the common language requirements. These include allowing expression of the form \((2 + 3) \ast (9 - 1) + 1\) and command structures such as assignment \(x := 4\). This functionality is mandatory for users to be able to create meaningful programs in the language which is a key objective for this project,
it was not seen as a prototype but more a goal to be completed so that interesting
developments could be explored. The first part of implementing the basic functionality
involved creating a grammar with ANTLR so that a lexer and parser could be built to
parse the code.

I created the grammar in Java using ANTLR. The grammar involved defining trivial
items such as comments, identifiers, integers, and whitespace but I feel the most impor-
tant aspects of the grammar are writing expressions, statements, and commands using
the BNF grammatical style.

I will begin by discussing the formation of an expression. In chapter 2 I discussed the
BNF grammar and illustrated the formation of this grammar with

\[
exp := n|exp + exp
\]  

(5.1)

I shall illustrate how the expression works in the program diagrammatically as shown
in figure 5.1

In figure 5.1 it is possible to see how a typical expression is formed. I will illustrate
this with an example, let an expression be \((2 + 3) \times 9\) in this example \((2 + 3)\) and 9 are
split into two different Mult types. The first Mult points to a term which is another
expression split into two Add types. The terms which the numbers eventually point to are Integers.

Once expressions have been correctly added to the language the next logical move is
to introduce command statements of some kind. In 5.2 I show the command structure
used in the grammar.

A statement describes a main program phrase and can be described with a snippet from
the ANTLR code as shown in 5.1.

<table>
<thead>
<tr>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>: assignment</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

Listing 5.1: Grammar Code For Statements
Figure 5.1: Expression Layout

![Expression Layout Diagram](image)

Figure 5.2: Command Layout

![Command Layout Diagram](image)
In figure 5.3 I illustrate how an assignment is formed by some identifier \( x, y, z \) together with a \( := \) operator and finally some expression which could take the form of some integer or a variable lookup e.g \( !y \).

**Figure 5.3: Assignment Layout**

In 5.2 I have shown what the If-statement looks like in terms of ANTLR code. The ? means this code can be written but is not compulsory.

```
ifstatement
  : 'if' expression 'then' op1 = command
     ('else' op2 = command)?
     'endif' ';'
  ;
```

**Listing 5.2: Example Code for If-Statement**

The logical statement *ifthenelse* has a simple structure which evaluates the truth of some expression and then deploys some command dependent on the boolean evaluation.

**Figure 5.4: IF-Statement Layout**

I will now discuss a simple example of a program and describe its grammatical layout. In 5.3 I have written an example of a simple swap program.
In this example I use \( z \) as a temporary variable while I assign \( x \) to \( y \) and then I assign \( y \) to \( z \). This is a very simple program but I would like to demonstrate the grammatical layout attributed to this program. This can be shown in 5.5.

**Figure 5.5: Grammatical layout of 5.3**

The code that is written between the `begin` and `end` key words is a command. This command has three statements which are each assignments.
5.2.2 ‘New’ Statement

In this subsection I will outline the prototype which added the new allocation of storage variable to the language. In 5.4 I have written a simple program for swapping values contained in variables. This is a familiar example shown in 5.3 but this time we make \( z \) a local storage variable. This command consists of one \textit{new} – \textit{statement} which contains another command. This last command contains three statements which are all \textit{assignments}.

```plaintext
begin
  new z in
  z := !x;
  x := !y;
  y := !z;
end
```

Listing 5.4: Example Code for a Swap Program Using ‘new’

In figure 5.6 I show the structure of the new-statement. In our example it is clear to see how this corresponds to the code. The important aspect is that we include a reference to another command. In figure 5.2 we showed how a command can point to multiple statements. In our example 5.4 we have multiple assignment statements.

5.2.3 While Loop

The while loop was implemented as an additional prototype and allowed more interesting programs to be written. The ANTLR code for the while loop can be seen in 5.5 and the grammatical structure is illustrated in 5.7.
A good example to discuss the while language implementation is with a simple factorial program. The while implementation also allowed me to write code for finding a Fibonacci number. These two examples are discussed in the requirements and so the introduction of the while allowed these requirements to be met.

5.2.4 Procedures

The last prototype implementation involved the simply typed lambda calculus. This meant an implementation of procedures in the language. A procedure is implemented such that a user can declare a procedure along with other variable declarations after the identification of the number system. This declaration corresponds to lambda abstraction and the bound variables are written as parameters to the procedure. A procedure call is viewed as a statement and is therefore bundled with other statements in the grammar such as assignment and if-statement. This prototype only allowed a default command return type which can be best described as a void return type in Java. The arguments for this procedure do not allow procedure types them selves and so the full lambda calculus is not supported with this prototype. In 5.6 I show the grammar code for the procedure declaration and in 5.7 I show the grammar code for calling a procedure.
procedure
    : 'procedure' IDENT '(' parameters ')' '{' command '}'
    ;

Listing 5.6: Grammar Code For Procedure Declaration

The parameters refer to multiple parameters and allow a user to write them in the form $x : Type$ for the types $var$ and $int$.

procedure call
    : IDENT '(' actualparameters ')' ';'
    ;

Listing 5.7: Grammar Code For Procedure Call Statements

The actual parameters are the arguments the user wishes to provide to the declared procedure and can written as any term including $!y$ and $4$. This corresponds to the application part of the lambda calculus.
5.3 Internal Representation of the Language

In the following I will describe how the implemented language is represented internally. The representation is handled differently for the two cases of declaration and main code. The main code is any code present between the begin and end keywords.

5.3.1 Main Body of Code

I used the interpreter design pattern described in chapter 4 to construct the internal structure of the main code with the idea that each program phrase has an interpreter class. In the implementation the interpreter classes are named evaluator classes. The evaluator classes are instantiated during the parsing of the language. An example of this can be seen in 5.8.

command returns [Evaluator e]
   :(op1 = statement) {e = $op1.e;}
   (op2 = statement {e = new StateEvaluator(e,$op2.e;}})*
;

Listing 5.8: Example Code for Command

In 5.8 I show how a command returns an evaluator class. The evaluator class can be one statement evaluator class in the case where there is one statement or a StateEvaluator class which will contain two statement evaluator classes. The evaluator class can have different internal processes depending on the program phrase however the structure will remain the same containing one set of segments which describe the semantics and one method to retrieve this set. The classes have pointers to other evaluator classes depending on the specific program phrase. The idea is rather simple to follow and so I will discuss the internal structure with one example 5.9

Program TestSimple {
   begin
     x:=2 + 3;
   end
}

Listing 5.9: Example Code for a Simple Assignment Statement

In listing 5.9 I have written a simple program which assigns the expression 2 + 3 to a variable x. It is clear that 2 and 3 are terms, + is an expression, and x := is an
assignment. During parsing of this code an evaluator class will be instantiate for each of these items. The assignment and expression will have a non terminal evaluator class while the integer terms will each have a leaf evaluator class. This is illustrated in figure 5.8.

**Figure 5.8: Using the Interpreter Pattern**

In figure 5.8 each integer value creates a leaf evaluator class. These classes are children of the add evaluator class. The add evaluator class performs computation with the segments retrieved from its children. The add evaluator class is a child of the assignment evaluator class. The assignment evaluator class performs computation on the segments retrieved from the add evaluator. In 5.10 I show the full code for the assignment statement which includes the creation of the evaluator class.

```java
assignment returns [Evaluator e]
    : IDENT ':=' expression ';' {e = new AssigEvaluator($IDENT.text , $expression.e);} ;
```

**Listing 5.10: Example ANTLR Code Assignment Statement**

### 5.3.2 Declarations

The declaration consists of variable declaration, number system declaration, and procedure declaration. The syntax for these items are discussed in the previous section. All of
the declaratives are represented internally via an Environment class. This class is never instantiated and is used statically by the Evaluator class when needed. The environment is setup during the parsing of the declarations. The number system is added as a field in the environment along with the amount of declared variables. A map is also created which relates variable names to their types. The procedure declaration is more complex. In 5.11 I show the ANTLR code for a procedure declaration.

```java
procedure :
    { par_list = new LinkedList<String>();
      'procedure' IDENT '(' 'parameters' ')' '{'
      command { new Procedure($IDENT.text, par_list,$command.e); }
      '}
    ;
```

Listing 5.11: Example ANTLR Procedure

The procedure class receives three arguments. The first two are the procedure name and the parameter list. The last one is an evaluator class which contains a set of segments describing all of the code inside the procedure declaration. The procedure class uses the list of parameters together with the segments obtained from command to produce new segments which represent the semantics of lambda abstraction shown later in section 5.5. The set of semantics is then stored in the environment inside a map with the name of the procedure as the key.
5.4 Representation of the Semantics

In this section I will describe the segment data structure which is used to represent the semantics of program phrases. This data structure is created and modified inside the evaluator design pattern via the class constructors. In figure 5.9 I show a UML class diagram which illustrates the segment data structure.

![Segment Class Structure Diagram](image)

The class structure shown in 5.9 shows how Segment has 1 to many VarEvent objects these correspond to a variable sequence and therefore contain multiple Event objects. In this structure it is possible to have IntegerEvent objects in the variable sequences as well as ReadEvent and WriteEvent objects. This is to allow for both var and int types in the language. It is worth noting that Segment is also an Event object and can therefore be included as an 'end' event. This also means that I can handle returning procedures as answers, but I can’t handle taking procedures as arguments.

The Segment class has internal functions which help manipulate the structure of the variable sequences. In particular there is a `concatenate()` method which accepts a Segment as an argument and concatenates all of the variable sequence together.
5.5 From Language to Semantics

5.5.1 Introduction

The transformation process takes place in the constructor of each evaluator class. The segments are generated internally in leaf classes and a set is produced which describes that particular evaluator class instance. This set of segments is then passed up by a method `pass()` so that a parent evaluator class above can receive the semantics of its children. The evaluator classes which are not leaf classes and hold other evaluator classes perform some kind of algorithmic computation on the semantics of their child evaluator classes. This process can be seen from a generic perspective in figure 5.10.

Figure 5.10: Generic Language to Semantics Transformation

In 5.10 The circle and right hand side items are discussing the processes involved inside the constructor of the evaluator class. The . notation is a generic algorithmic operation of some kind which combines the segments from (a) and (b) to create new segments which represent the semantics of (a.b).

5.5.2 Basic Transformation

I will continue discussing the example in 5.9 with some more detail into the transformation of language into semantics. I will now describe the specific evaluator classes which relate to the example shown in figure 5.9. The evaluator classes shown in this example are IntegerEvaluator, AddEvaluator, and AssignmentEvaluator. Figure 5.11 shows the class structure for these evaluators and the processes involved in evaluating `x := 2 + 3;`.

In figure 5.11 changes made to the segments are displayed in curly brackets. The IntegerEvaluator class is a leaf class in the interpreter design pattern and therefore does not have any children. In this case segments are generated depending on the integer value written in the code and the size of the variable system. The size of the variable system will dictate the amount of variable sequences to be created. In this example we have assumed that the environment consists of three variables hence why we have
In the constructor of the AddEvaluator class the method `pass()` is called on both of the IntegerEvaluator classes which provides two set’s with one containing 
\(((),(),(),2)\) and the other containing 
\(((),(),(),3)\). The variable sequences in these two segments are concatenated together using the `concatenate()` method contained in the segment classes and a new end event class is created with the two end events from each IntegerEvaluator added together. The AssignmentEvaluator creates an instance of a WriteEvent and adds the integer value it obtains from the segment received from AddEvaluator. It then adds this WriteEvent after all of the other events contained in the \(x\) variable sequence. I should note here that in order to correctly identify the position of a particular variable sequence a call is made to the Environment class which contains a map of all of the current variables and their index in a sequence.

This example has been simple and is used to familiarize the reader with the evaluator pattern. In the next sections I will discuss the various prototypes achieved and in these sections I will describe the algorithms involved in the evaluator classes with more detail as they become more complex.
5.5.3 New Statement

The constructor in the NewEvaluator class contains two arguments. One is a String which represents the local variable allocation for this new block. Another is a pointer to another Evaluator class. The algorithmic processes involved in this evaluators constructor are straightforward. The first process involves identifying the segments passed by the argument evaluator which have good variable behavior. The second process is to hide variable sequences corresponding the the local variable. The last process is to update the environment so that it removes all references and information about the local variable which was used.

The variable behavior is checked by making a call to a Behavior class. This class can correctly identify good variable behavior in variable sequences. The main processes can be seen in the Java code in 5.12.

```java
Event e_1 = lists.get(k); // read or a write
Event e_2 = lists.get(k+1); // read or a write

if (e_1.isRead() && e_2.isRead()) {
    if (e_1.getVarValue() == e_2.getVarValue()) {}
    else {
        toRet = false;
        break;
    }
} else if (e_1.isWrite() && e_2.isRead()) {
    if (e_1.getVarValue() == e_2.getVarValue()) {}
    else {
        toRet = false;
        break;
    }
}
```

Listing 5.12: Java code to force good variable behavior

A sequence is considered to have good variable behavior if all of the read events that follow each other match and a read event which follows a write event also matches. In 5.12 these checks are made by calling specific event methods which provide answers to whether or not an event is a read or a write. The values of each event are then compared and if they are not equal we break out of a loop which is checking all of a particular
variable sequence.
The second process is done quite easily by removing all event objects which belong to
the local variable sequence.
The last part is also done quite easily and involves changing some fields in the Environment
class which modify the current global representation of variables.

5.5.4 While Loop

The while-loop was the first clear checkpoint which would allow interesting programs
to be written and represented in the model. The implementation for while-loop was complex mainly because of the large amount of segments which needed to be generated.
If code is written with some \(!x\) the model does not force this variable to have good variable behavior and therefore regardless of the variable events preceding or succeeding it there is a large finite amount of possibilities when generating read events for this. At this stage in the implementation I used a constrained number system which consisted of 0,1, and 2 together with a finite variable system of x,y, and z.

The while-loop had a similar grammar layout to if-statement. The most important aspect of while-loop is the interpreter class which builds the segments describing its behavior. The interpreter class similar to other interpreter classes inherits other interpreters and then performs some computation on the set of segments which they contain.
This interpreter in particular inherits a set of segments from a boolean expression inter-preter together with a set of segments from a command interpreter. This means that the set coming from the boolean expression interpreter will contain segments of the form \((S_1, S_2, S_3, T/F)\) and the set from the command interpreter will contain segments of the form \((S_1, S_2, S_3, \ast)\). In chapter 2 the semantics of a specific while loop were shown. This can be made more general by using the boolean operators true and false. The following is the semantics for a generic while loop.

\[
\llbracket \text{while } M \text{ do } C \rrbracket = \left\{ \left( s_1^1, t_1^1, \ldots, s_1^n, t_1^n, s_2^1, t_2^1, \ldots, s_2^n, t_2^n, s_3^1, t_3^1, \ldots, s_3^n, t_3^n, \ast \right) \mid \left( s_1^1, s_1^n, s_3^1, s_3^n, T \right) \in \llbracket M \rrbracket \land \left( t_1^1, t_2^1, t_3^1, \ast \right) \in \llbracket C \rrbracket \right\} \cup \left\{ \left( s_1, s_2, s_3, \ast \right) \mid \left( s_1, s_2, s_3, F \right) \in \llbracket M \rrbracket \right\}
\]

The main algorithm for while is discussed in the following. The first thing to notice from the semantics is that there are some terminal segments which belong to \(M\) and not necessarily only 1. If \(M := !x == 2\); we would have the following segments which would not enter the loop:
• \((\text{read}(0), (), (), 0)\)

• \((\text{read}(1), (), (), 1)\)

In this case the boolean evaluator will have modified the end event to F and so the boolean evaluator in this example would have returned a set of segments including the following :

• \((\text{read}(0), (), (), F)\)

• \((\text{read}(1), (), (), F)\)

• \((\text{read}(2), (), (), T)\)

The first part of the algorithm separates the terminal segments from the loop segments (the False segments from the True). The second part of the algorithm iterates over the set of command segments and concatenates the loop segments together with the command segments and then concatenates with all of the terminal segments. This provides a large set of segments which have executed the loop once. This is continued to compute all of the segments when the loop is executed twice. An example I used in testing is shown in listing 5.13

```
1 Program TestWhile {
3 begin
4 while !x != 0 do
5 y:=1;
6 endwhile;
7 end
8 }
```

Listing 5.13: Example Code for a While-Loop

In this example I will keep to the number system described earlier. The output segments can be describes as follows :

0-loop:

• \((\text{read}(0), (), (), *)\)

1-loop

• \((\text{read}(1)\text{read}(0), (), (), *)\)
Chapter 5. Implementation

- \((\text{read}(2)\text{read}(0),(),(),\ast)\)

2-loop

- \((\text{read}(1)\text{read}(1)\text{read}(0),(),(),\ast)\)
- \((\text{read}(2)\text{read}(1)\text{read}(0),(),(),\ast)\)
- \((\text{read}(1)\text{read}(2)\text{read}(0),(),(),\ast)\)
- \((\text{read}(2)\text{read}(2)\text{read}(0),(),(),\ast)\)

It is clear from this example that the amount of segments grows exponentially and eventually we will have an infinite set of possibilities and this is for the constrained number system involving only 0,1, and 2. At this stage it would seem that something needs to be done in order to judge which segments should be included in the semantics of a while and generating all of the possibilities is not sufficient.

The answer to the problem of generating the segments for the while-loop came by modifying the algorithm so that it forced the segments to have good variable behavior. This task was handled by the Behavior class discussed earlier. The new while algorithm is shown in figure 5.12

In this algorithm it was possible to identify when an infinite loop would occur for a particular segment because the algorithm will repeat for good behaving non terminal segments. If a segment is identified as having a read event which matches one of the terminal segments this segment breaks out of the algorithm and is added to the set of segments. I created a fixed loop limit which is based on the number system provided by the user in the declaration. This limit is seen as satisfactory coding because it is based on the finite number system and therefore does not constrain the results. The limit is set as 2 times the number system because it was thought that the maximum amount of loops is largely dependent on the incrementation which at its lowest form is incrementing by 1. It is multiplied by two for insurance that no segments will be lost however tests showed that setting it to the number system provided satisfactory results. The only segments which can be attributed to a constant loop like this are segments of the kind

- \((\text{read}(n)\text{read}(n)\text{read}(n)\text{read}(n),(),(),\ast)\)

which do not change. These are dealt with as the algorithm stops when it reaches the finite limit. The end result shows only the segments which were excepted so in the case
of a loop with no incrementation the terminal segments will be shown where as the looped good behaving segments will be lost but were printed out during some test trials.

5.5.5 Procedures

The implementation for the simply typed lambda calculus involved creating a new syntax which allowed the user to create procedures which could use parameters as discussed earlier. This subsection can be split into two more sections one which will deal with declaring the procedure (lambda abstraction) and another which will deal with calling
the procedure (application). The implementation of procedures was split further into two parts with one basic form which only returns a command type and another higher order procedure version. The latter version was not implemented because of time constraints.

5.5.5.1 Procedure Declaration

The main idea for lambda abstraction was to generate the correct segments and store them in the Environment class for later use in the body of the code. The storage procedure was straightforward and involved using a map with the function name as the key and a set of segments to describe the semantics as its lookup. The process of generating segments which accurately describe the procedure was complex and deserves further attention.

The main idea for the semantics of lambda abstraction is to create segments which contain an additional segment as the end event which is possible due to the semantic interpretation in the program. The end segment contains all of the variable events which are present in the parameter declaration. The high level process of creating these segments can be seen in figure 5.13.

Figure 5.13: Procedure Declaration

Figure 5.13 illustrates an example with one global variable of type var and one parameter of type int. It is clear that the parameter moves into a new segment which is placed at the end. In the following I will describe how this process works.

The segments for the code inside the procedure declaration are retrieved via the normal way (with pass()) then these segments are looped over and their variable sequences are
looped over. If a variable sequence corresponds to a parameter for the procedure it is removed from the segment and added to a new segment. This process continues and eventually a new segment is created of the form \((s_1, s_2, (s_3, *))\). This is shown by Java code in 5.14.

```java
SegmentEvent segs = new SegmentEvent(seg.var_list, new SegmentEvent(var_list, new CommEvent()));
```

**LISTING 5.14:** Java Code to Create Lambda Abstraction Segments

### 5.5.6 Procedure Application

In the application part of a procedure the argument segments need to be matched up with the corresponding parameter segments which are inside the additional end segment as described above. The high level process of matching up segments can be seen in figure 5.14.

**Figure 5.14:** Procedure Application

Figure 5.14 continues the same example shown in 5.13 and discussed earlier. It is clear to see what is happening as the argument semantics \(((), n)\) is matched up with the declared procedure semantics \((\text{write}(n), (n, *))\) resulting in the procedure call semantics \((\text{write}(n))\). In the following I will describe how this process works.

If the procedure was declared with no parameters we simply return the semantics obtained via a lookup of the procedure name in the environment class. If the procedure declaration did contain parameters we retrieve the semantics of the actual arguments
and loop over their segments. If these segments match their respective parameter position in the end segment, which is the end event of the declared procedure semantics, they are kept and concatenated with the rest of the procedure segment not including the end segment. This can best be described with an example 5.15 shows a swap program written as a procedure.

Program proctest {
    var : x ;
    var : y ;
    num : 3 ;

    procedure swap(m : var , n : var ) {
        new z in
        z := !m;
        m := ! n ;
        n := ! z ;
        endnew

    begin
    swap(x , y ) ;
    end
    }
}

Listing 5.15: Procedure Swap Program

The declaration will generate semantics of the form (((), (read(n), write(m)
, read(m), write(n), *)) via the process described earlier. These segments will be stored in the Environment class. The procedure call swap(x, y) will start by iterating over the declaration segments. Inside this loop will be another loop over all the possible segments for the arguments x and y. These segments will be of the form (read(n), read(n)) and (write(n), write(n)). The application variables x and y are uniquely identified with their parameter location m and n respectively. This means that firstly we will be trying to match up the segments of x with the m events in the main segments i.e (((), (), (read(n), write(m), read(m), write(n), *))). There is an additional loop which keeps track of the event in the variable sequence that we are trying to match. if a read(n) and write(n) can be obtained from x they are placed together in the x position of the main segment and the respective parameter events are removed.
5.6 Interface

The interface was created using NetBeans GUI editor. It is simple and allows a user to write code or import it using a file browser. I attempted to create some additional features such as syntax highlighting and compile messages feature. This was implemented to help achieve the objective of creating a user friendly tool capable of teaching the object based semantics to people. The output segments are printed in a different color to the code and displayed in a window on the right. There is a create button which will start the process of parsing the code and generate the semantics. There is also a clear button and exit button where the clear button removes all of the code and segments and the exit button terminates the program. The messages box displays any errors in the system and will also catch the syntax errors thrown by ANTLR. This was done by overriding the method in the parser which deals with these messages. The syntax coloring is in a beta stage and was added near the end. I built this by creating a thread which will loop over the code checking for key words. This looping stops if there is no difference in the size of the code since its last loop. In figure 5.15 I show a screen shot of this GUI.

![Figure 5.15: Graphical User Interface](image)

The interface also includes a check box which allows a user to test programs with good variable behavior forced on every statement or without this feature (more like object spaces model). This feature was largely used for testing however I feel that these features help teach people about the model and so they have been left on the interface. Another feature displayed on the interface is a particular message in the message box which shows the running time in millisecond for the program used. This is also a good feature to leave on the system as it allows users to analyze their programs.
Chapter 6

Testing and Experimentation

6.1 Introduction

In this chapter I will split the sections up according to different forms of testing which includes testing that the program correctly transforms language into the required semantics, Usability and performance testing, and experimentation which will describe some parts of the program which were experimented with in order to achieve interesting results. The test items and their results are held inside the appendix however I will show some of the more interesting parts here.

6.2 Functional Tests

The functional tests were carried out during the creation of particular prototype implementations. The tests will therefore follow the familiar format of basic functionality, new allocation of storage variable, while loops, and lastly procedures. In these tests the good variable behavior switch in the interface has been turned on to create more understandable semantics. It is worth pointing out that the model does not force such variable behavior and this is purely to show the segments which represent what you would expect from the output. In the experimentation section I show some of the differences when the switch is turned off.

6.2.1 Test1 and Test2

In Test1 the aim was to successfully compute an empty program. This involved using the `skip();` command which should return an empty sequence of a size dependent on the
amount of variables. The results clearly show the required output and A.1.3 shows the ability to change the variable system and the effect it has on the empty sequence. The following are the results you would expect for different variable contexts:

Context — var : x, var : y, var : z — ((),(),(),*)
Context — var : x, var : y, var : z, var : w, var : v — ((),(),(),(),(),*)

The results above show the introduction of an empty sequence for each variable in the context.

Test2 is also very simple and clearly shows the correct semantics being created for an assignment. In A.1.3 the results show an increase in segments which are dependent on the defined number system. This means that a simple assignment will generate more segments for a larger number system. The following are the results for when the number system is 10:

((write(9)),(read(9)),(),*)
((write(1)),(read(1)),(),*)
((write(2)),(read(2)),(),*)
((write(7)),(read(7)),(),*)
((write(8)),(read(8)),(),*)
((write(0)),(read(0)),(),*)
((write(4)),(read(4)),(),*)
((write(6)),(read(6)),(),*)
((write(5)),(read(5)),(),*)
((write(3)),(read(3)),(),*)

In the above it is clear that 0-9 integer values are being read and written to x.

6.2.2 Test3

In Test3 the idea was to test a nested logical statement. The following results are for A.3.1 which has three nested if statements which call the skip() statement if the variable x is equal to 0, 1, 2 in that order. The if statements are nested in the else part of the previous if statements with the last if statement having its else part as assigning 1 to y.
((read(1)read(1)),(),(),*)

((read(2)read(2)read(2)),(),(),*)

((read(3)read(3)read(3)),(write(1)),(),*)

((read(0)),(),(),*)

This program had a data set of 0, 1, 2, 3 and so if you consider the variable $x$ to be 3 it is clear from the semantics that $x$ is read three times in each logical statement before eventually writing 1 to the variable $y$. This is highlighted in the above example. The test in A.3.3 is a similar example.

### 6.2.3 Test4

This test is a familiar test and was used in the poster presentation. In A.4.1 there is code for a simple swap program which will temporarily place values in a variable $z$ while assigning $x$ to $y$ values and then assign $y$ to values in $z$. This program can be seen here in 6.6.

```plaintext
Program swap {
  var : x ;
  var : y ;
  var : z ;
  num : 3 ;
  begin
    z := !x ;
    x := !y ;
    y := !z ;
  end
}
```

**Listing 6.1: Testing Material**

The results in A.4.2 are large because they contain all read possibilities and variables are not constrained to good variable behavior. These results can be seen here:

((read(2)write(0)),(read(0)write(1)),(write(2)read(1)),*)

((read(2)write(2)),(read(2)write(0)),(write(2)read(0)),*)

((read(2)write(0)),(read(0)write(2)),(write(2)read(2)),*)

((read(0)write(0)),(read(0)write(1)),(write(0)read(1)),*)
The results in A.4.4 are for the modified swap program which involves a new allocation of the variable $z$ and good variable behavior is forced upon $z$ before it is removed from the segments. This gives much smaller results of what you would expect and $x$ and $y$ are swapping values as you would expect. I have highlighted the semantics of the old swap to illustrate this good variable behavior. The final results for the new version of swap can be seen in the following:
6.2.4 Test5

These tests simply demonstrate the possibilities of having nested new statements. In A.5.3 I show the possibility of defining a local variable of the same name as a global variable. This works fine because the local variable is renamed if there is possible interference with global variable names. This happens in the Environment class when the new statement is parsed.

6.2.5 Test6

The first major prototype was the introduction of while loop. These tests demonstrate some of its uses in the program. The tests in A.6.1 and A.6.3 show how standard incrementing or decrementing of variables produces the required output semantics. An example for this is to consider the results in A.6.2 and imagine the incremented variable is initially the value 0. The semantics correctly show the variable being read twice on 0 and then there is a write of 1 followed by two more reads for 1 which continues until the terminal value of 2 is read. This can be seen clearly in the following:

\[((\text{read}(0)\text{read}(0)\text{write}(1)\text{read}(1)\text{read}(1)\text{write}(2)\text{read}(2)),*)\]
\[((\text{read}(1)\text{read}(1)\text{write}(2)\text{read}(2)),*)\]
\[((\text{read}(0)\text{write}(1)),\text{read}(1)\text{write}(0)),*)\]
\[((\text{read}(0)\text{write}(2)),\text{read}(2)\text{write}(0)),*)\]
It is worth noting that the results for while are smaller and more correct because good variable behavior is forced upon segments being processed as part of the while algorithm. This was discussed in the implementation 5. In tests A.6.5 and A.6.7 I have demonstrated the ability to create factorial and Fibonacci programs in the language and produce the correct semantics. In both tests I use a variable to write out the results expected for the sequences. I will only show the factorial example here because further down is a more interesting Fibonacci example.

```
Program factorial {
    var : x ;
    var : y ;
    num : 3 ;
    begin
        y := 1 ;
        while ! x > 0 do
            y := ! y * ! x ;
            x := ! x − 1 ;
        endwhile
    end
}
```

The above code is used to detect factorial numbers. In this example I turned the "all" good variable behavior feature off. The output gives correct segments describing semantics for finding !0, !1, and !2. I show the results here and I have highlighted the correct segments which exhibit good variable behavior.

```
((read(1)read(1)read(1)write(0)read(0)),(write(1)read(1)write(1)),*)
((read(1)read(1)read(1)write(0)read(0)),(write(1)read(2)write(2)),*)
((read(2)read(2)read(2)write(1)read(1)read(1)read(1)write(0)read(0)),
    (write(1)read(0)write(0)read(0)write(0)),*)
((read(1)read(1)read(1)write(0)read(0)),(write(1)read(0)write(0)),*)
((read(0)),(write(1)),*)
((read(2)read(2)read(2)write(1)read(1)read(1)write(0)read(0)),
    (write(1)read(1)write(2)read(2)write(2)),*)
```
It is clear from above that these segments are correctly describing the factorial sequence but only for a small data set. In test A.6.9 I have shown the first implementation of a bubble sorting algorithm which does not use procedures and uses multiple variable declarations to create an array. The results shown in A.6.10 clearly show the three variable values being read on the right and a sorting assignment of these values on the left. I will not show these results here because later I repeat the same program using procedures.

In test A.6.11 I have revisited the Fibonacci program and applied good variable behavior on the statements in the program along with initiating the \textit{counter} variable to 1. I have also increased the number system to find larger Fibonacci numbers. This looks like the following:

\begin{verbatim}
Program fib {
  var : fib1 ;
  var : fib2 ;
  var : fib3 ;
  var : counter ;
  num : 9;
  begin
    fib1 := 0;
    fib2 := 1;
    counter := 0;
    while !counter < 5 do
      fib3 := !fib1 + !fib2;
      fib1 := !fib2;
      fib2 := !fib3;
      counter := !counter + 1;
  endwhile
  end
}
\end{verbatim}

Listing 6.3: Testing Material

This provides one large segment for the results which clearly shows the program identifying the Fibonacci sequence reaching the value 8 as shown below.

\[(\text{write}(0)\text{read}(0)\text{write}(1)\text{read}(1)\text{write}(1)\text{read}(1)\text{write}(1)\text{read}(1)\text{write}(2)\text{read}(2)\text{write}(3)\text{read}(3)\text{write}(5)),\]

\[(\text{write}(1)\text{read}(1)\text{write}(1)\text{read}(1)\text{write}(1)\text{read}(1)\text{write}(2)\text{read}(2)\text{write}(3)\text{read}(3)\text{write}(5)\text{read}(5)\text{write}(8)),\]
(write(1)read(1)write(2)read(2)write(3)read(3)write(5)read(5)write(8)read(8)),

(write(0)read(0)read(0)write(1)read(1)read(1)write(2)read(2)
read(2)write(3)read(3)write(4)read(4)read(4)write(5)read(5)),

These results are interesting because during this test I noticed that the number system had to contain the answer I was looking for. This is why the number system is equal to 9 so that 8 can be understood by the program. This is because the $!get$ term will generate semantics dependent on the finite number system and so in order to receive 8 from the addition of two variables $fib2$ and $fib3$ this number must be possible.

6.2.6 Test7

In this series of tests the main aim was to replace older programs with procedures and to test the possibility of multiple arguments. The test in A.7.1 demonstrates a simple procedure with no arguments which increments a global variable $x$. The results A.7.2 show what one would expect where each starting variable value is incremented. In test A.7.3 the familiar swap program is recreated inside a procedure. In this program the $z$ variable is a local variable and $x$ and $y$ are swapped via the procedure. It is important to note here that the procedure declaration has distinct variable names from the variable applied to the procedure which is as you would expect because of syntactic control of interference. The program will not allow variables bound in a lambda abstraction to be the same as variables contained in application. The results shown in A.7.4 are exactly the same as the results from the earlier version of swap shown in A.4.4 showing that these two programs are semantically equivalent to each other. The test in A.7.5 shows the same bubble sort algorithm in test A.6.9 but written as a procedure. The results in A.7.6 are the same as in A.6.10. The last test in A.7.7 shows the bubble sort procedure with a nested internal swap procedure. This last example was one of the main objectives discussed in chapter 3. The following code demonstrates this bubble sort program:

```
Program Sort {
  var : x ;
  num : 3 ;
  procedure swap(i : var, j : var) {
    if !i > !j then
      new temp in
      temp := !i ;
      i := !j ;
      j := !temp ; endnew
    else
```
The arguments for this procedure are integer values and the results shown in A.7.8 correctly illustrate the variable $x$ working as an output stream with the values written to the variable in a sorted order. I present these finding below to show this more clearly:

$$(((\text{write}(0)\text{write}(1)\text{write}(2)),*)$$
It is also possible to provide variables to the sort procedure to generate multiple alternative results which are dependent on the number system 0, 1, 2. This can be understood as imagining if a user would like to sort the numbers 2, 1, 1 instead of the above or any other arrangement possible in the finite data set. The following results show the segments generated when \texttt{sort(!_m, !_n, !_p)} is called.

\begin{verbatim}
((write(1)write(1)write(2)),(read(2)),(read(1)),(read(1)),*)
((write(0)write(1)write(1)),(read(1)),(read(1)),(read(0)),*)
((write(0)write(1)write(2)),(read(2)),(read(0)),(read(1)),*)
((write(0)write(1)write(2)),(read(2)),(read(1)),(read(0)),*)
((write(2)write(2)write(2)),(read(2)),(read(2)),(read(2)),*)
((write(0)write(0)write(2)),(read(0)),(read(2)),(read(0)),*)
((write(0)write(0)write(0)),(read(0)),(read(0)),(read(0)),*)
((write(1)write(2)write(2)),(read(1)),(read(2)),(read(2)),*)
((write(0)write(1)write(2)),(read(0)),(read(2)),(read(1)),*)
((write(1)write(1)write(1)),(read(1)),(read(1)),(read(1)),*)
((write(0)write(1)write(2)),(read(1)),(read(2)),(read(0)),*)
((write(0)write(0)write(1)),(read(1)),(read(0)),(read(0)),*)
((write(0)write(0)write(1)),(read(0)),(read(1)),(read(0)),*)
((write(0)write(1)write(2)),(read(0)),(read(1)),(read(2)),*)
((write(0)write(0)write(1)),(read(0)),(read(0)),(read(1)),*)
((write(0)write(1)write(2)),(read(1)),(read(0)),(read(2)),*)
((write(1)write(1)write(2)),(read(1)),(read(2)),(read(1)),*)
((write(0)write(0)write(2)),(read(0)),(read(2)),(read(2)),*)
((write(0)write(2)write(2)),(read(2)),(read(2)),(read(0)),*)
((write(1)write(2)write(2)),(read(2)),(read(2)),(read(1)),*)
((write(0)write(2)write(2)),(read(0)),(read(2)),(read(2)),*)
\end{verbatim}
In this example I made the variables $m, n, p$ global variables so that they would be shown in the segments. The variable sequences on the right illustrate the particular values taken by the variables and the variable sequence on the left is the ordered sequence of values written to $x$ at the end.

### 6.3 Non-Functional Tests

The non-functional tests were carried out at the end of the final implementation for the purpose of meeting the non-functional requirements specified in chapter 3.

#### 6.3.1 Test1

The purpose of these tests was to test the performance of the program on some carefully selected examples. The first test shown in B.1.1 is a simple while program which has an incremented variable to exit the loop. The test involved generating the semantics for the while loop on varied sizes of data sets. A simple stopwatch class was used to time the difference in milliseconds between starting the analysis and ending the analysis. This analysis involves parsing the code, creating internal structures, and generating the segments. The results are displayed below in the table.

<table>
<thead>
<tr>
<th>Number System</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6.8</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>20</td>
<td>13</td>
<td>12</td>
<td>13</td>
<td>16.4</td>
</tr>
<tr>
<td>15</td>
<td>26</td>
<td>35</td>
<td>33</td>
<td>29</td>
<td>39</td>
<td>32.4</td>
</tr>
<tr>
<td>20</td>
<td>166</td>
<td>76</td>
<td>79</td>
<td>152</td>
<td>178</td>
<td>130.2</td>
</tr>
<tr>
<td>25</td>
<td>198</td>
<td>174</td>
<td>225</td>
<td>231</td>
<td>178</td>
<td>201.2</td>
</tr>
<tr>
<td>30</td>
<td>386</td>
<td>402</td>
<td>496</td>
<td>367</td>
<td>337</td>
<td>397.6</td>
</tr>
<tr>
<td>35</td>
<td>984</td>
<td>910</td>
<td>969</td>
<td>766</td>
<td>772</td>
<td>880.2</td>
</tr>
<tr>
<td>40</td>
<td>1518</td>
<td>1513</td>
<td>1486</td>
<td>1397</td>
<td>1482</td>
<td>1479.2</td>
</tr>
<tr>
<td>45</td>
<td>2442</td>
<td>2505</td>
<td>2299</td>
<td>2407</td>
<td>2598</td>
<td>2432.2</td>
</tr>
<tr>
<td>50</td>
<td>3676</td>
<td>3928</td>
<td>3678</td>
<td>3654</td>
<td>3424</td>
<td>3672</td>
</tr>
</tbody>
</table>
It is clear to see that there is a correlation between the size of the number system and the time it takes to compute the results. The time seems to grow exponentially which is because of the large amount of computations that take place between the segments. The graph below shows this exponential growth:

**Figure 6.1:** A graph to show the relation between number system size and computation time

I tested the factorial program discussed earlier. I was able to correctly generate the results for 4! however the program produce a overflow error for 5!. The following table displays the results of the time in millisecond for the computation.

<table>
<thead>
<tr>
<th>Factorial</th>
<th>Number System</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>2!</td>
<td>3</td>
<td>18</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6.2</td>
</tr>
<tr>
<td>3!</td>
<td>7</td>
<td>28</td>
<td>25</td>
<td>36</td>
<td>21</td>
<td>28</td>
<td>27.6</td>
</tr>
<tr>
<td>4!</td>
<td>25</td>
<td>9794</td>
<td>10400</td>
<td>9790</td>
<td>10616</td>
<td>9720</td>
<td>10064</td>
</tr>
<tr>
<td>5!</td>
<td>121</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

The resulting segment for 4! can be seen below I have highlighted a write event which clearly shows 4! = 24.

```
((read(4)read(4)read(4)write(3)read(3)read(3)read(3)write(2)read(2)
read(2)read(2)write(1)read(1)read(1)read(1)write(0)read(0)),
```
Chapter 6. Testing and Experimentation

\[(\text{write}(1)\text{read}(1)\text{write}(4)\text{read}(4)\text{write}(12)\text{read}(12)\text{write}(24)\text{read}(24)\text{write}(24)),*)\]

\[((\text{read}(1)\text{read}(1)\text{write}(0)\text{read}(0)),(\text{write}(1)\text{read}(1)\text{write}(1)),*)\]

\[((\text{read}(0)),(\text{write}(1)),*)\]

\[((\text{read}(2)\text{read}(2)\text{write}(1)\text{read}(1)\text{read}(1)\text{read}(1)\text{write}(0)\text{read}(0)),\]

\(\text{write}(1)\text{read}(1)\text{write}(2)\text{read}(2)\text{write}(2)),*)\]

\[((\text{read}(3)\text{read}(3)\text{write}(2)\text{read}(2)\text{read}(2)\text{read}(2)\text{write}(1)\text{read}(1)\text{write}(0)\text{read}(0)),\]

\(\text{write}(1)\text{read}(1)\text{write}(3)\text{read}(3)\text{write}(6)\text{read}(6)\text{write}(6)),*)\]

6.3.2 Test 2

This test was a test to see how much a student could understand and use the tool. The test was completed by two students and both students were able to answer the first of the two questions shown in B.2. The students understood the results generated for these questions after given a short introduction on the model. The last question involved asking the participants to write swap programs which is considered a difficult task for anyone who is new to the model and the language.

6.4 Experimentation

The experimentation tests were created during each prototype and resulted from the evolutionary development cycle. The three main experimental observations were adding alternative variable behavior to programs, testing for infinite loops in programs involving while loops, and identifying semantic equivalences of two different sorting algorithms. The experimentation of adding alternative variable behavior has largely been covered in the functional tests and was naturally experimental in order to produce readable semantics. The model only forces new allocation of storage variables to have good variable behavior but the while loop produced so many segments that it became impossible to work with even on small data sets. This meant that I implemented good variable behavior for the while loops. I also added a check box to the interface so that users can force good variable behavior on all statements.

The other two experiments are more important on their own and this section is dedicated to them.
6.4.1 Test2

In this series of tests I show an aspect of the algorithm described in 5.12 in chapter 5. The aspect tested here is the ability to recognize infinite loops and correctly display the semantics for the foreseeable infinite loop structure. I demonstrate this in C.2.1 with a program which enters a loop but does not increment a variable to exit the loop. This program can be seen here:

Program empty {
    var : x ;
    var : y ;
    var : z ;
    num : 4 ;
    begin
    while !x < 2 do
        y:=1;
    endwhile ;
    end
}

Listing 6.5: Testing Material

The results shown in C.2.2 show the segments which were looped on which have good variable behavior and are semantically correct but can never continue in the algorithm because they never reach a terminal event such as one of the events included in the standard results. These results can be seen here:

Inf Loop :

looped on
((read(0)read(0)read(0)read(0)read(0)read(0)read(0)read(0)
read(0)read(0)read(0)read(0)read(0)read(0)read(0)read(0)),
(write(1)write(1)write(1)write(1)write(1)write(1)write(1)write(1)
write(1)write(1)write(1)write(1)write(1)write(1)write(1)write(1)),(),*)

looped on
((read(1)read(1)read(1)read(1)read(1)read(1)read(1)read(1)
read(1)read(1)read(1)read(1)read(1)read(1)read(1)read(1)),
(write(1)write(1)write(1)write(1)write(1)write(1)write(1)write(1)
write(1)write(1)write(1)write(1)write(1)write(1)write(1)write(1))
write(1)write(1)write(1)write(1)write(1)),(),*)

Standard Result :

((read(2)),(),(),*)
((read(3)),(),(),*)

6.4.2 Test3

This experiment can be seen as the ultimate objective. In this test I generated the semantics for two different sorting algorithms to demonstrate that they are semantically equivalent. The results suggest that they are indeed semantically equivalent which is what you would expect because given a list of numbers both algorithms sort these numbers in ascending order. I should point out here it was difficult to construct general sorting algorithms given the data set is so small, however I was able to create an algorithm which is different from the bubble sort algorithm using maximum, minimum, and medium variables which works for the data set 0, 1, and 2. These program can be seen in C.4 and I show the new sorting algorithm here:

Program Sort {
  var : x;
  var : m;
  var : n;
  var : p;
  num : 3;

  procedure sort(par1 : int, par2 : int, par3 : int) {
    new a0 in
    new a1 in
    new a2 in

    a0 := par1;
    a1 := par2;
    a2 := par3;

    new min in
    new max in
    new med in
    min := !a0;
    max := !a0;
    med := !a0;
if !a1 < !min then
  min := !a1;
else
  if !a2 < !min then
    min := !a2;
  else
    skip();
  endif
endif

if !a1 > !max then
  max := !a1;
else
  if !a2 > !max then
    max := !a2;
  else
    skip();
  endif
endif

if !a0 < !max then
  if !a0 > !min then
    med := !a0;
  else
    skip();
  endif
else
  skip();
endif

if !a1 < !max then
  if !a1 > !min then
    med := !a1;
  else
    skip();
  endif
else
  skip();
endif

if !a2 < !max then
  if !a2 > !min then
    med := !a2;
  else
    skip();
  endif
else
  skip();
endif

a0 := !min;
a1 := !med;
a2 := !max;
endnew
dendnew
dendnew

x := !a0;
\begin{verbatim}
x := !a1;
x := !a2;

endnew
endnew
endnew

begin
sort (!m,!n,!p);
end
\end{verbatim}

Listing 6.6: Testing Material
Chapter 7

Results and Discussion

7.1 Introduction

This chapter will critically evaluate the program and the work done for this project. The following will give a detailed discussion of the results obtained and will explain some difficulties which were met during the development of the program. The results will be analyzed and evaluated to see how well the requirements and objectives were met described in chapter 3.

7.2 Overview

Many simple programs were created using the program made in this project. The most interesting programs were constructed after the program fully supported while loops. I will begin by discussing some interesting programs developed using while loops, then I will discuss some bigger programs which involved using procedures.

During the implementation of the while loop I began small tests on simple loops which would assign an integer to a variable. The purpose of this was quite simply to identify that the semantics of the program was accurately modeled in the object spaces model. At first the results for the while loop were infinity long because good variable behavior was not forced upon the segments involved. This meant that a variable could act like a bad variable and have unmatched read and write events. This meant that the sequences increased exponentially. This was fixed by adding a check for good variable behavior in the algorithm for the while loop. The first interesting program was a simple loop which assigned a value to a variable and incremented another. This experiment successfully produced the semantics one would expect. In the algorithm I was able to identify the
segments which would cause a while loop to loop continuously. This was done by simply recognizing the fact that if a segment entered a loop and was continuously being analyzed by the algorithm and satisfied all of the good behavior constraints it should have eventually terminated within a finite amount of loops. The finite number system constraint placed on the program by the user can be used to halt segments which are looping in this way. I tested this idea with a program which did not increment and was able to detect the correct segments which had good behavior but were not changing or incrementing in any direction. This is discussed in 6.

The next programs I tested were the famous Fibonacci and factorial functions. The results for these two functions would suggest the object spaces model can accurately describe the behavior one would expect from a variable which is having either of the sequences written to it. The main problem was the ability to produce large factorials or identify a large Fibonacci number due to segments growing exponentially. This is discussed in 6.

The main program written was the procedure sort. I started by using the same algorithm used in Homer which was bubble sort. I intended to copy the approach used in Homer which was to use a variable as a input output stream for reading the values which were to be sorted and then write the sorted results. This was not possible due to the good variable behavior checks made in the while loop and so I added additional variables which could be read for the individual values to be sorted. The results show that the internal behavior of sort are hidden and all that can be seen is the read events and most importantly the write events to the global variable. This is discussed in 6.

I was also able to write an alternative sort program so that I could test the model and check the equivalences of two sorting algorithms. The results were discussed in 6 and showed that the two programs were equivalent.

### 7.3 Requirements and Objectives

This section aims to evaluate the success of the project by comparing the results with the requirements and objective discussed in chapter 3. The language requirements were largely satisfied with the exception of higher order procedures. This was due to time constraints and it is felt that this could be accomplished with more time. The language is sufficiently powerful and allows one to write interesting program as discussed in the overview.

The interface has a text area to allow code to be written and has an import feature. The import feature has a file browser which is easy to use, fast, and attractive. The interface could benefit from some improvements to allow multiple programs to be written and a save feature to store the semantic results. The printed segments are correct and do
represent all input code in the object spaces model. The segments are displayed in a window next to the code in a different color. This could be improved by displaying them in a more attractive way highlighting interesting parts when necessary.

The results from the user test would suggest that the system is easy to use however the data in the results do not accurately reflect the larger population because the amount of students who participated was very low. The participants who were involved were able to use the syntax and were capable of understanding the semantics generated by the program.

The system does not provide a help feature to support the user as this was seen as unnecessary because the system is intuitive and buttons are clearly labeled.

The system was tested on a Windows machine and a Unix machine and worked on both operating systems.

The system is reliable and provides the required output as discussed in previous chapters. The experiments and tests can be replicated by a user and the code will be supplied with the main program.
Chapter 8

Conclusions

8.1 Introduction

In this chapter I will describe the conclusions drawn from the project. The conclusions will be based on the success of the project, lessons learned and possible improvements, and contributions to current knowledge is the subject area.

8.2 Project Success

In the introductory chapter 1 the aim was defined as building a tool which would generate object based semantics for code based upon the theoretical language of SCI. The results shown in 6 show that the tool correctly generates the semantics of the implemented language. The implemented language has most of the features described in SCI with the exception of the full simply typed lambda calculus in particular higher order procedures. The requirements and objectives shown in 3 specified the need to have a tool which is capable of computing relatively interesting programs such as Fibonacci, factorial, and sorting algorithms. All of these programs were accomplished and their results correctly identified with the object spaces model. It was a further objective to make the tool user friendly and allow people to learn the processes involved in the object spaces model. This was satisfied with in the project by making a user interface which is intuitive and the user test results implied students were able to learn how to use the implemented language together with understanding the semantics generated. Another key objective I was keen on completing was showing that two sorting programs are semantically equivalent in the model. This was satisfied by using the bubble sort algorithm and an alternative algorithm based on minimum and maximum values. The bubble sort algorithm produced the same results as the game semantics model checker Homer [8].
8.3 Further Improvements

If the project was to continue the next step would be to add higher order procedures to the implementation. This could be done by adding return types to the grammar and modifying the segment data structure to allow for procedures as arguments. Higher order procedures would allow development of recursive versions of the Fibonacci and factorial programs which would be interesting programs to test.

The Interpreter design pattern is implemented incorrectly and the main functionality should not be in the constructor but rather inside a method. The design pattern also works on the fly which causes a lot of memory usage. This could be improved by building an internal structure of the language first then generate the semantics by traversing this structure.

The current program does not identify equivalences between programs although one can observe the equivalences in the segments generated. This could be improved by allowing multiple programs to be written and having a automatic detection feature which would correctly display program equivalences.

The interface could be improved with a better visual layout which could interactively teach users about the object spaces model. The current interface syntax highlighting feature is not fully operational and could be improved to highlight syntax efficiently along with applying indentation to code.

8.4 Contributions

The main results for the various programs and in particular the sorting algorithms do demonstrate the capability of this model providing correct meaning for programs. The results of this project suggests good variable behavior is required in the semantics of the while loop. The while loop is difficult to check without good variable behavior checks because the segments grow exponentially and continue infinity.

The model does not seem to consider the size of the segments for large data sets and in practice it is difficult to produce segments for programs on a large data set.

The identification of possible infinite loops works dependent on knowledge of the current finite data set being used and shows how variable behaviors can be used to detect infinite loops.

The results for program comparison do suggest that the model is capable of detecting semantic equivalences resulting from the fact that the model is fully abstract as discussed in 2.

This tool involved using an environment to store the declared variables and procedures
which seems to suggest that some global state is required. This is discussed in a paper called A Global State Considered Necessary.
Appendix A

Functional Tests

In this appendix I shall provide the test code used together with the semantic results obtained from the program. In all of the tests good variable behavior is assumed and applied to all statements unless otherwise stated.

A.1 Test1 : Empty Program

The following tests provide results for a simple empty program which computes a skip command and returns an empty list of variable sequences. Further to this test I will demonstrate the ability to increase the finite variable system.

A.1.1 Test : x,y,z variable system on [0,1,2]

Program empty {
var : x ;
var : y ;
var : z ;
um : 3 ;
begin
skip () ;
end
}

Listing A.1: Testing Material

A.1.2 Results
Appendix A. Functional Tests

A.1.3 Test : increased variable system

Program empty {
    var : x ;
    var : y ;
    var : z ;
    var : w ;
    var : v ;
    num : 3 ;
    begin
    skip ( ) ;
    end
}

A.1.4 Results

Listing A.2: Testing Material

A.2 Test2 : Assignment Program

The following tests provide results for a program which contains one assignment to the variable x. The test proceeds to test increase in the number system.

A.2.1 Test : x,y,z variable system on [0,1,2]

Listing A.3: Testing Material

Listing A.4: Testing Material
Appendix A. Functional Tests

```
x := !y;
end
}
```

Listing A.5: Testing Material

A.2.2 Results

```
(( write (0) ),( read (0) ),(),*)
(( write (1) ),( read (1) ),(),*)
(( write (2) ),( read (2) ),(),*)
```

Listing A.6: Testing Material

A.2.3 Test: increased number system

```
Program assign {
var : x ;
var : y ;
var : z ;
um : 10 ;
begin
 x := !y;
end
}
```

Listing A.7: Testing Material

A.2.4 Results

```
(( write (9) ),( read (9) ),(),*)
(( write (1) ),( read (1) ),(),*)
(( write (2) ),( read (2) ),(),*)
(( write (7) ),( read (7) ),(),*)
(( write (8) ),( read (8) ),(),*)
(( write (0) ),( read (0) ),(),*)
(( write (4) ),( read (4) ),(),*)
(( write (6) ),( read (6) ),(),*)
(( write (5) ),( read (5) ),(),*)
(( write (3) ),( read (3) ),(),*)
```

Listing A.8: Testing Material
A.3 Test3 : Logical Statements

These programs demonstrate the use of If-statements in the code and large nested If-statements.

A.3.1 Test : Nested If statements

Program logical {
var : x ;
var : y ;
var : z ;
num : 4 ;
begin
if ! x == 0 then
skip ();
else
if ! x == 1 then
skip ();
else
if ! x == 2 then
skip ();
else
y := 1;
endif
endif
endif
end
}

Listing A.9: Testing Material

A.3.2 Results

((read (1) read (1)) , () , () , *)
((read (2) read (2) read (2)) , () , () , *)
((read (3) read (3) read (3)) , (write (1)) , () , *)
((read (0)) , () , () , *)

Listing A.10: Testing Material
Appendix A. Functional Tests

A.3.3 Test : Nested If Statements

Program logical {
    var : x ;
    var : y ;
    var : z ;
    num : 3 ;
    begin
        if !x == 0 then
            y := 1;
            y := 1;
            y := 1;
        else
            if !x == 1 then
                y := 1;
                y := 1;
            else
                if !x == 2 then
                    y := 1;
                else
                    skip();
            endif
        endif
    end
}

Listing A.11: Testing Material

A.3.4 Results

((read(0)), (write(1) write(1) write(1)) , (), *)
((read(2) read(2) read(2)) , (write(1)) , (), *)
((read(1) read(1)) , (write(1) write(1)) , (), *)

Listing A.12: Testing Material

A.4 Test4 : Swap Program

This program demonstrates the use of multiple statements and provides the correct results for their semantics. It includes segments which have bad variable behavior because
the model does not force good variable behavior at this stage. However good variable behavior is forced with the new allocation of storage variable prototype.

**A.4.1 Test : x,y,z and [0,1,2]**

Program swap {
  var : x ;
  var : y ;
  var : z ;
  num : 3 ;
  begin
    z := !x ;
    x := !y ;
    y := !z ;
  end
}

Listing A.13: Testing Material

**A.4.2 Results**

\[(\text{read}(2) \text{write}(0)) , (\text{read}(0) \text{write}(1)) , (\text{write}(2) \text{read}(1)) , *\]
\[(\text{read}(2) \text{write}(2)) , (\text{read}(2) \text{write}(0)) , (\text{write}(2) \text{read}(0)) , *\]
\[(\text{read}(2) \text{write}(0)) , (\text{read}(0) \text{write}(2)) , (\text{write}(2) \text{read}(2)) , *\]
\[(\text{read}(0) \text{write}(0)) , (\text{read}(0) \text{write}(1)) , (\text{write}(0) \text{read}(1)) , *\]
\[(\text{read}(2) \text{write}(2)) , (\text{read}(2) \text{write}(2)) , (\text{write}(2) \text{read}(2)) , *\]
\[(\text{read}(2) \text{write}(1)) , (\text{read}(1) \text{write}(2)) , (\text{write}(2) \text{read}(2)) , *\]
\[(\text{read}(1) \text{write}(0)) , (\text{read}(0) \text{write}(0)) , (\text{write}(1) \text{read}(0)) , *\]
\[(\text{read}(2) \text{write}(1)) , (\text{read}(1) \text{write}(1)) , (\text{write}(2) \text{read}(1)) , *\]
\[(\text{read}(1) \text{write}(0)) , (\text{read}(1) \text{write}(0)) , (\text{write}(1) \text{read}(0)) , *\]
\[(\text{read}(0) \text{write}(0)) , (\text{read}(0) \text{write}(2)) , (\text{write}(0) \text{read}(2)) , *\]
\[(\text{read}(1) \text{write}(2)) , (\text{read}(2) \text{write}(1)) , (\text{write}(1) \text{read}(1)) , *\]
\[(\text{read}(1) \text{write}(2)) , (\text{read}(2) \text{write}(2)) , (\text{write}(1) \text{read}(2)) , *\]
\[(\text{read}(0) \text{write}(2)) , (\text{read}(2) \text{write}(0)) , (\text{write}(0) \text{read}(0)) , *\]
\[(\text{read}(2) \text{write}(0)) , (\text{read}(0) \text{write}(0)) , (\text{write}(2) \text{read}(0)) , *\]
\[(\text{read}(2) \text{write}(2)) , (\text{read}(2) \text{write}(1)) , (\text{write}(2) \text{read}(1)) , *\]
\[(\text{read}(1) \text{write}(0)) , (\text{read}(0) \text{write}(2)) , (\text{write}(1) \text{read}(2)) , *\]
\[(\text{read}(1) \text{write}(2)) , (\text{read}(2) \text{write}(0)) , (\text{write}(1) \text{read}(0)) , *\]
\[(\text{read}(1) \text{write}(1)) , (\text{read}(1) \text{write}(1)) , (\text{write}(1) \text{read}(1)) , *\]
\[(\text{read}(2) \text{write}(1)) , (\text{read}(1) \text{write}(0)) , (\text{write}(2) \text{read}(0)) , *\]
Appendix A. Functional Tests

((read(0) write(2)) , (read(2) write(1)) , (write(0) read(1)) , *)
((read(1) write(1)) , (read(1) write(2)) , (write(1) read(2)) , *)
((read(0) write(1)) , (read(1) write(2)) , (write(0) read(2)) , *)
((read(0) write(0)) , (read(0) write(0)) , (write(0) read(0)) , *)
((read(0) write(1)) , (read(1) write(1)) , (write(0) read(1)) , * )

Listing A.14: Testing Material

A.4.3 Test : x,y,z and [0,1,2]

Program swap {
    var : x ;
    var : y ;
    num : 3 ;
    begin
    new z in
    z := ! x ;
    x := ! y ;
    y := ! z ;
    endnew
    end
}

Listing A.15: Testing Material

A.4.4 Results

((read(1) write(2)) , (read(2) write(1)) , *)
((read(0) write(0)) , (read(0) write(0)) , *)
((read(2) write(0)) , (read(0) write(2)) , *)
((read(1) write(1)) , (read(1) write(1)) , *)
((read(2) write(2)) , (read(2) write(2)) , *)
((read(1) write(0)) , (read(0) write(1)) , *)
((read(2) write(1)) , (read(1) write(2)) , *)
((read(0) write(1)) , (read(1) write(0)) , *)
((read(0) write(2)) , (read(2) write(0)) , *)

Listing A.16: Testing Material

A.5 Test5 : Working with 'new'

Testing the limits of the new construct
A.5.1 Test : \(x,y,z\) and \([0,1,2]\) one nested new statement

Program newtest {
    var : x ;
    num : 3 ;
    begin
        new z in
        new y in
            x := ! y + ! z ;
    endnew
endnew
end

Listing A.17: Testing Material

A.5.2 Results

\((\text{write}(1)),*\)
\((\text{write}(2)),*\)
\((\text{write}(0)),*\)
\((\text{write}(4)),*\)
\((\text{write}(3)),*\)

Listing A.18: Testing Material

A.5.3 Test : \(x,y,z\) and \([0,1]\) repeat variable name

Program newtest {
    var : x ;
    var : y ;
    num : 2 ;
    begin
        new z in
        new y in
            x := ! y + ! z ;
    endnew
endnew
end

Listing A.19: Testing Material
A.5.4 Results

((write(1)),*)
((write(0)),*)
((write(2)),*)

Listing A.20: Testing Material

A.6 Test6: Working with ’while’

A.6.1 Test: Increment loop

Program whiletest {
var : x ;
num : 3 ;
begin
while !x < 2 do
x := !x + 1;
endwhile
end
}

Listing A.21: Testing Material

A.6.2 Results

((read(0)read(0)write(1)read(1)write(2)read(2)),*)
((read(1)read(1)write(2)read(2)),*)
((read(2)),*)

Listing A.22: Testing Material

A.6.3 Test: Decrement loop

Program whiletest {
var : x ;
num : 3 ;
begin
while !x > 0 do
x := !x - 1;
endwhile
A.6.4 Results

\[
\begin{align*}
&((\text{read}(2)\ \text{read}(2)\ \text{write}(1)\ \text{read}(1)\ \text{write}(0)\ \text{read}(0)),*) \\
&((\text{read}(0)),*) \\
&((\text{read}(1)\ \text{read}(1)\ \text{write}(0)\ \text{read}(0)),*)
\end{align*}
\]

Listing A.24: Testing Material

A.6.5 Test: Finding Factorial Numbers

Program factorial {
var : x;
var : y;
um : 3;
begin
y := 1;
while !x > 0 do
  y := !y * !x;
x := !x - 1;
endwhile
end
}

Listing A.25: Testing Material

A.6.6 Results

\[
\begin{align*}
&((\text{read}(1)\ \text{read}(1)\ \text{write}(0)\ \text{read}(0)),(\text{write}(1)\ \text{read}(1)\ \text{write}(1)),*) \\
&((\text{read}(2)\ \text{read}(2)\ \text{write}(1)\ \text{read}(1)\ \text{read}(1)\ \text{write}(0)\ \text{read}(0)),(\text{write}(1)\ \text{read}(1)\ \text{read}(1)\ \text{write}(0)\ \text{read}(0)),*) \\
&((\text{read}(0)),(\text{write}(1)),*)
\end{align*}
\]

Listing A.26: Testing Material
A.6.7 Test: Finding Fibonacci Numbers

Program fib {
    var : fib1 ;
    var : fib2 ;
    var : fib3 ;
    var : counter ;
    num : 3 ;
    begin
    fib1 := 0 ;
    fib2 := 1 ;
    while ! counter < 2 do
    fib3 := ! fib1 + ! fib2 ;
    fib1 := ! fib2 ;
    fib2 := ! fib3 ;
    counter := ! counter + 1 ;
    endwhile
    end
}

Listing A.27: Testing Material

A.6.8 Results

(( write (0) read (0) write (1) ), ( write (1) read (1) read (1) write (1) ) ,
( write (1) read (1) ), ( read (1) read (1) write (2) read (2) ),* )

(( write (0) ), ( write (1) ), (), ( read (2) ),* )

(( write (0) read (0) write (1) read (1) write (1) ) ,
( write (1) read (1) read (1) write (1) read (1) read (1) write (2) ) ,
( write (1) read (1) write (2) read (2) ) ,
( read (0) read (0) write (1) read (1) write (2) read (2) ),* )

Listing A.28: Testing Material

A.6.9 Test: Bubble Sort

Program Sort {
    var : x ;
    var : m ;
    var : n ;
    var : p ;
num : 3 ;
begin

new a0 in
new a1 in
new a2 in

a0 := !m;
a1 := !n;
a2 := !p;

new flag in
flag := 0;
while !flag < 2 do

if !a0 > !a1 then
new temp0 in
temp0 := !a0;
a0 := !a1;
a1 := !temp0; endnew
else
skip();
endif

if !a1 > !a2 then
new temp1 in
temp1 := !a1;
a1 := !a2;
a2 := !temp1; endnew
else
skip();
endif

flag := !flag +1;
endwhile endnew

x := !a0;
x := !a1;
x := !a2;

endnew
endnew
endnew
A.6.10 Results

\begin{verbatim}
end
}
\end{verbatim}

\textbf{Listing A.29: Testing Material}

\section*{A.6.11 Test: Finding Larger Fibonacci Numbers}

\begin{verbatim}
Program fib {
    var : fib1;
    var : fib2;
\}
\end{verbatim}
Appendix A. Functional Tests

![Listings A.31 and A.32: Testing Material]

A.6.12 Results

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>write (0) read (0) write (1) read (1) write (1) read (1) write (2) read (2) write (3) read (3) write (5)</td>
</tr>
<tr>
<td>2</td>
<td>write (1) read (1) write (1) read (1) write (1) read (1) write (2) read (2) read (2) write (3) read (3) read (3) write (5) read (5) read (5) write (8)</td>
</tr>
<tr>
<td>3</td>
<td>write (1) read (1) read (1) write (2) read (2) write (3) read (3) read (3) write (4) read (4) read (4) write (5) read (5)</td>
</tr>
</tbody>
</table>

A.7 Test7 : Working with procedures’

A.7.1 Test : Simple Procedure

Program simpleproc {
  var : x ;
  num : 3 ;
  procedure foo() {
    x := !x +1;
  }
  begin
    foo();
  end
}
A.7.2 Results

\[
\begin{align*}
&((\text{read}(2)\ \text{write}(3)),*) \\
&((\text{read}(1)\ \text{write}(2)),*) \\
&((\text{read}(0)\ \text{write}(1)),*)
\end{align*}
\]

A.7.3 Test : Swap Procedure

Program proctest {
var : x ;
var : y ;
um : 3 ;

procedure swap(m : var , n : var) {
new z in
z := !m;
m := !n;
n := !z;
endnew
}

begin
swap(x,y);
end
}

A.7.4 Results

\[
\begin{align*}
&((\text{read}(1)\ \text{write}(2)),(\text{read}(2)\ \text{write}(1)),*) \\
&((\text{read}(1)\ \text{write}(1)),(\text{read}(1)\ \text{write}(1)),*) \\
&((\text{read}(2)\ \text{write}(2)),(\text{read}(2)\ \text{write}(2)),*) \\
&((\text{read}(2)\ \text{write}(1)),(\text{read}(1)\ \text{write}(2)),*) \\
&((\text{read}(0)\ \text{write}(1)),(\text{read}(1)\ \text{write}(0)),*) \\
&((\text{read}(1)\ \text{write}(0)),(\text{read}(0)\ \text{write}(1)),*)
\end{align*}
\]
A.7.5 Test : Bubble Sort Procedure

Program Sort {
var : x ;
var : m;
var : n ;
var : p ;
um : 3 ;

procedure sort (par1 : int, par2 : int, par3 : int) {
new a0 in
new a1 in
new a2 in

a0 := par1;
a1 := par2;
a2 := par3;

new flag in
flag := 0;
while !flag < 2 do

if !a0 > !a1 then
new temp0 in
temp0 := !a0;
a0 := !a1;
a1 := !temp0; endnew
else
skip();
endif

if !a1 > !a2 then
new temp1 in
temp1 := !a1;
a1 := !a2;
a2 := !temp1; endnew

}

Listing A.36: Testing Material
Appendix A. Functional Tests

```plaintext
else
  skip();
endif

flag := !flag +1;
endwhile endnew

x := !a0;
x := !a1;
x := !a2;
endnew endnew endnew

}
begin
  sort (!m, !n, !p);
end

Listing A.37: Testing Material

A.7.6 Results

(( write(1) write(1) write(1) ), (read(1)), (read(1)), (read(1)), *)
(( write(0) write(1) write(1) ), (read(0)), (read(1)), (read(1)), *)
(( write(0) write(1) write(1) ), (read(1)), (read(0)), (read(1)), *)
(( write(0) write(2) write(2) ), (read(0)), (read(2)), (read(2)), *)
(( write(0) write(0) write(1) ), (read(0)), (read(0)), (read(1)), *)
(( write(0) write(1) write(2) ), (read(1)), (read(2)), (read(0)), *)
(( write(0) write(0) write(2) ), (read(0)), (read(0)), (read(2)), *)
(( write(0) write(0) write(1) ), (read(0)), (read(1)), (read(0)), *)
(( write(2) write(2) write(2) ), (read(2)), (read(2)), (read(2)), *)
(( write(0) write(0) write(2) ), (read(0)), (read(0)), (read(0)), *)
(( write(1) write(2) write(2) ), (read(1)), (read(2)), (read(2)), *)
(( write(0) write(0) write(0) ), (read(0)), (read(0)), (read(0)), *)
(( write(1) write(2) write(2) ), (read(1)), (read(2)), (read(1)), *)
(( write(0) write(0) write(2) ), (read(0)), (read(2)), (read(1)), *)
(( write(0) write(1) write(2) ), (read(0)), (read(2)), (read(0)), *)
(( write(1) write(1) write(2) ), (read(1)), (read(2)), (read(1)), *)
(( write(0) write(1) write(1) ), (read(1)), (read(1)), (read(0)), *)

```
Appendix A. Functional Tests

Listing A.38: Testing Material

A.7.7 Test : Bubble Sort with nested swap procedure

Program Sort {
  var : x ;
  num : 3 ;

  procedure swap(i : var, j : var) {
    if !i > !j then
      new temp in
      temp := !i;
      i := !j;
      j := !temp; endnew
    else
      skip();
    endif
  }

  procedure sort(par1 : int, par2 : int, par3 : int) {
    new a0 in
    new a1 in
    new a2 in

    a0 := par1;
    a1 := par2;
    a2 := par3;

    new flag in
    flag := 0;
    while !flag < 2 do
swap(a0, a1);
swap(a1, a2);

flag := !flag +1;
endwhile endnew

x := !a0;
x := !a1;
x := !a2;
endnew endnew endnew

}

begin
sort(1, 0, 2);
end
}

Listing A.39: Testing Material

A.7.8 Results

((write(0) write(1) write(2)), *)

Listing A.40: Testing Material
Appendix B

Non-Functional Tests

In this appendix I shall provide the test code used together with the semantic results obtained from the program.

B.1 Test 1: Performance Test

In this test I will test the speed of computation on large programs.

B.1.1 Test: While Loop

This test will use the a simple program which includes a while loop with incrementation and will involve timing the computation in milliseconds. The tests will be on different sized number systems.

B.1.2 Results

<table>
<thead>
<tr>
<th>Number System</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>6.8</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>20</td>
<td>13</td>
<td>12</td>
<td>13</td>
<td>16.4</td>
</tr>
<tr>
<td>15</td>
<td>26</td>
<td>35</td>
<td>33</td>
<td>29</td>
<td>39</td>
<td>32.4</td>
</tr>
<tr>
<td>20</td>
<td>166</td>
<td>76</td>
<td>79</td>
<td>152</td>
<td>178</td>
<td>130.2</td>
</tr>
<tr>
<td>25</td>
<td>198</td>
<td>174</td>
<td>225</td>
<td>231</td>
<td>178</td>
<td>201.2</td>
</tr>
<tr>
<td>30</td>
<td>386</td>
<td>402</td>
<td>496</td>
<td>367</td>
<td>337</td>
<td>397.6</td>
</tr>
<tr>
<td>35</td>
<td>984</td>
<td>910</td>
<td>969</td>
<td>766</td>
<td>772</td>
<td>880.2</td>
</tr>
<tr>
<td>40</td>
<td>1518</td>
<td>1513</td>
<td>1486</td>
<td>1397</td>
<td>1482</td>
<td>1479.2</td>
</tr>
<tr>
<td>45</td>
<td>2442</td>
<td>2505</td>
<td>2209</td>
<td>2407</td>
<td>2598</td>
<td>2432.2</td>
</tr>
<tr>
<td>50</td>
<td>3676</td>
<td>3928</td>
<td>3678</td>
<td>3654</td>
<td>3424</td>
<td>3672</td>
</tr>
</tbody>
</table>
Appendix B. Non-Functional Tests

Figure B.1: A graph to show the relation between number system size and computation time

B.1.3 Test: Factorial Program

This test will use the program described in ?? and will involve timing the computation in milliseconds. The tests will be on different sized number systems. (number sys in line with results)

B.1.4 Results

<table>
<thead>
<tr>
<th>Factorial</th>
<th>Number System</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>2!</td>
<td>3</td>
<td>18</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>6.2</td>
</tr>
<tr>
<td>3!</td>
<td>7</td>
<td>28</td>
<td>25</td>
<td>36</td>
<td>21</td>
<td>28</td>
<td>27.6</td>
</tr>
<tr>
<td>4!</td>
<td>25</td>
<td>9794</td>
<td>10400</td>
<td>9790</td>
<td>10616</td>
<td>9720</td>
<td>10064</td>
</tr>
<tr>
<td>5!</td>
<td>121</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

B.1.5 Test: Swap Program

This test will analyze the swap program and will involve timing the computation in milliseconds. The tests will be on different sized number systems.
Appendix B. Non-Functional Tests

B.1.6 Results

<table>
<thead>
<tr>
<th>Number System</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>3.6</td>
</tr>
<tr>
<td>10</td>
<td>94</td>
<td>173</td>
<td>188</td>
<td>79</td>
<td>94</td>
<td>125.6</td>
</tr>
<tr>
<td>15</td>
<td>422</td>
<td>360</td>
<td>469</td>
<td>750</td>
<td>359</td>
<td>472</td>
</tr>
<tr>
<td>20</td>
<td>1592</td>
<td>1265</td>
<td>1437</td>
<td>1467</td>
<td>1546</td>
<td>1461.4</td>
</tr>
<tr>
<td>25</td>
<td>3433</td>
<td>3277</td>
<td>3371</td>
<td>3480</td>
<td>3137</td>
<td>3339.6</td>
</tr>
<tr>
<td>30</td>
<td>7208</td>
<td>7005</td>
<td>8534</td>
<td>6850</td>
<td>7442</td>
<td>7407.8</td>
</tr>
<tr>
<td>35</td>
<td>13729</td>
<td>14712</td>
<td>13417</td>
<td>13978</td>
<td>12888</td>
<td>13744.8</td>
</tr>
<tr>
<td>40</td>
<td>24821</td>
<td>21860</td>
<td>24330</td>
<td>24512</td>
<td>24669</td>
<td>24038.4</td>
</tr>
</tbody>
</table>

Figure B.2: A graph to show the relation between number system size and computation time

B.2 Test 2: User Test

In this test we tested the capabilities of a student using the program having given a short introduction to the material and concepts including a brief description of the syntax for the language.
B.2.1 User Test

The following is a copy of the test given to a group of students:

- Start the application
- Write the following programs:
  - A while loop which includes if-statement and an incremented variable [2]
  - A program to detect factorial numbers [3]
  - A program to sort the numbers 1,0, and 2 into the order 0,1, and 2 [5]
- Analyze the semantics for each:
- Answer the following questions:
  - Identify the variable sequences associated with incrementation from the output of question 1. [1]
  - Identify the variable sequences associated with the factorial number in the output of question 2. [1]
  - If the new statement did not hide its associated variables behavior what would the output of question 3 look like? [3]

B.2.2 User Study

The above test was conducted by giving students a copy of the literature review, language design specification, and a demonstration of how to use the program.

B.3 Results

Unfortunately there were only two volunteers for this test, however both students were capable of completing the first two tasks and understood the semantics generated. It is possible that the other questions are too difficult for people who are new to the subject area.
Appendix C

Experimental Tests

In this appendix I shall provide the test code used together with the semantic results obtained from the program.

C.1 Test1 : Variable Behavior

The following examples are programs which are used with good variable behavior switched off.

C.1.1 Test : Factorial Function

Program factorial {
var : x ;
var : y ;
um :  3;
begin
y:=1;
while !x > 0 do
y:= !y * !x ;
x:= !x - 1;
endwhile
end
}

Listing C.1: Testing Material
Appendix C. Experimental Tests

C.1.2 Results

((read(0)),(write(1)),*)
((read(1) read(1) write(0) read(0)),
  (write(1) read(0) write(0)),*)
((read(1) read(1) write(0) read(0)),
  (write(1) read(1) write(1)),*)
((read(2) read(2) read(2) write(1) read(1) read(1) write(0) read(0)),
  (write(1) read(0) write(0) read(0) write(0)),*)
((read(2) read(2) read(2) write(1) read(1) read(1) write(0) read(0) write(0)),
  (write(1) read(1) write(2) write(2) write(2)),*)
((read(1) read(1) write(0) read(0)),
  (write(1) read(2) write(2)),*)

Listing C.2: Testing Material

C.1.3 Test: Fibonacci Function

Program fib {
  var : fib1 ;
  var : fib2 ;
  var : fib3 ;
  var : counter ;
  num : 3 ;
begin
  fib1 := 0 ;
  fib2 := 1 ;
  while !counter < 2 do
    fib3 := !fib1 + !fib2 ;
    fib1 := !fib2 ;
    fib2 := !fib3 ;
    counter := !counter + 1 ;
  endwhile
end
}

Listing C.3: Testing Material
Appendix C. Experimental Tests

C.1.4 Results

Listing C.4: Testing Material
C.2 Test 2: Looping

In this test I will demonstrate the capability of the program to identify infinite loops.

C.2.1 Test: No Incrementation

Program empty {
var : x ;
var : y ;
var : z ;
um : 4 ;
begin
while ! x < 2 do
y := 1;
endwhile
end
}

Listing C.5: Testing Material

C.2.2 Results

Inf Loop >>

looped on

\((\text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0) \text{read}(0)), \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1)), (, *)\)

looped on

\((\text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1) \text{read}(1)), \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1) \text{write}(1)), (, *)\)

Standard Result >>

\((\text{read}(2)), (, ), (, *)\)
\((\text{read}(3)), (, ), (, *)\)
C.3 Test3 : Sorting Comparison

In this test I will demonstrate the capability of the program to correctly identify semantic equivalence in syntactically different programs.

C.4 Test : Two Sorting Algorithms

The following programs contains different procedures for sorting. They are both different algorithms with one based on bubble sort and another based on determining maximum and minimum values. In these programs both produce identical semantics.

Listing C.6: Testing Material

```
Program Sort {
  var : x ;
  var : m;
  var : n ;
  var : p ;
  num : 3 ;

  procedure sort(par1 : int ,par2 : int ,par3 : int) {
    new a0 in
    new a1 in
    new a2 in

    a0 := par1;
    a1 := par2;
    a2 := par3;

    new min in
    new max in
    new med in
    min := !a0;
    max := !a0;
    med := !a0;

    if !a1 < !min then
      min := !a1;
    else
```
if !a2 < !min then
    min := !a2;
else skip();
endif endif

if !a1 > !max then
    max := !a1;
else
    if !a2 > !max then
        max := !a2;
    else skip();
    endif
endif

if !a0 < !max then
    if !a0 > !min then
        med := !a0;
    else skip();
    endif
else skip();
endif

if !a1 < !max then
    if !a1 > !min then
        med := !a1;
    else skip();
    endif
else skip();
endif

if !a2 < !max then
    if !a2 > !min then
        med := !a2;
    else skip();
    endif
else skip();
endif

a0 := !min;
a1 := !med;
a2 := !max;
endnew
endnew
endnew

x := !a0;
x := !a1;
x := !a2;
endnew
endnew
endnew

} 

begin
sort (!m, !n, !p);
end
}

Program Sort {
var : x;
var : m;
var : n;
var : p;
um : 3;

procedure sort (par1 : int, par2 : int, par3 : int) {
new a0 in
new a1 in
new a2 in

a0 := par1;
a1 := par2;
a2 := par3;

new flag in
flag := 0;
while !flag < 2 do

if !a0 > !a1 then
new temp0 in
temp0 := !a0;
a0 := !a1;
a1 := !temp0; endnew
else
skip();
endif

if !a1 > !a2 then
new temp1 in
temp1 := !a1;
Appendix C. Experimental Tests

Listing C.7: Testing Material

```
\{(write(0) write(1) write(1) , (read(1) , (read(0) , (read(1) , *,
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(0) , *,
\{(write(1) write(2) write(2) , (read(2) , (read(1) , (read(1) , *,
\{(write(0) write(0) write(2) , (read(0) , (read(0) , (read(2) , *
\{(write(0) write(0) write(0) , (read(0) , (read(0) , (read(0) , *
\{(write(1) write(1) write(2) , (read(1) , (read(1) , (read(2) , *
\{(write(0) write(1) write(2) , (read(0) , (read(2) , (read(1) , *
\{(write(2) write(2) write(2) , (read(2) , (read(2) , (read(2) , *
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(2) , *
\{(write(0) write(1) write(2) , (read(2) , (read(0) , (read(1) , *
\{(write(1) write(1) write(2) , (read(1) , (read(2) , (read(2) , *
\{(write(0) write(1) write(1) , (read(1) , (read(0) , (read(0) , *
\{(write(0) write(2) write(2) , (read(2) , (read(0) , (read(0) , *
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(1) , *
\{(write(0) write(0) write(2) , (read(0) , (read(2) , (read(2) , *
\{(write(0) write(0) write(2) , (read(0) , (read(2) , (read(2) , *
```

C.4.1 Results

\{(write(0) write(1) write(1) , (read(1) , (read(0) , (read(1) , *
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(0) , *
\{(write(1) write(2) write(2) , (read(2) , (read(1) , (read(1) , *
\{(write(0) write(0) write(2) , (read(0) , (read(0) , (read(2) , *
\{(write(0) write(0) write(0) , (read(0) , (read(0) , (read(0) , *
\{(write(1) write(1) write(2) , (read(1) , (read(1) , (read(2) , *
\{(write(0) write(1) write(2) , (read(0) , (read(2) , (read(1) , *
\{(write(2) write(2) write(2) , (read(2) , (read(2) , (read(2) , *
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(2) , *
\{(write(0) write(1) write(2) , (read(2) , (read(0) , (read(1) , *
\{(write(1) write(1) write(2) , (read(1) , (read(2) , (read(2) , *
\{(write(0) write(1) write(1) , (read(1) , (read(0) , (read(0) , *
\{(write(0) write(2) write(2) , (read(2) , (read(0) , (read(0) , *
\{(write(0) write(0) write(1) , (read(0) , (read(1) , (read(1) , *
\{(write(0) write(0) write(2) , (read(0) , (read(2) , (read(2) , *
\```
Listing C.8: Testing Material

( (write (0) write (0) write (2)) , (read (0)) , (read (2)) , (read (0)) , * )
( (write (1) write (1) write (1)) , (read (1)) , (read (1)) , (read (1)) , * )
( (write (1) write (2) write (2)) , (read (2)) , (read (2)) , (read (1)) , * )
( (write (1) write (1) write (2)) , (read (1)) , (read (2)) , (read (1)) , * )
( (write (1) write (2) write (2)) , (read (2)) , (read (1)) , (read (2)) , * )
( (write (0) write (0) write (1)) , (read (0)) , (read (0)) , (read (1)) , * )
( (write (0) write (1) write (1)) , (read (1)) , (read (1)) , (read (0)) , * )
( (write (0) write (1) write (2)) , (read (1)) , (read (2)) , (read (0)) , * )
( (write (0) write (2) write (2)) , (read (2)) , (read (0)) , (read (2)) , * )
( (write (1) write (2) write (2)) , (read (2)) , (read (1)) , (read (0)) , * )
( (write (0) write (1) write (1)) , (read (1)) , (read (1)) , (read (0)) , * )
( (write (0) write (1) write (2)) , (read (1)) , (read (2)) , (read (0)) , * )
Bibliography


[6] John Backus. the proposed international algebraic language. *the proposed international algebraic language*.


