Agent-oriented Parsing
with Empirical Modelling

Author: Antony Harfield

Supervisor: Dr Meurig Beynon

Year: 2002/2003

Copy 1

Abstract


The project is an investigation into agent-oriented parsing. It outlines the theory behind the parsing process and looks in detail at the behaviour of the parser. A new parser is developed that extends and improves on the existing implementation. A set of tools are developed to analyse notations and visual the parsing process.

Keywords: Empirical Modelling, Eden, agent, parser, Sand, Eddi
## Contents

Abstract .......................................................................................................................... 1
Contents .......................................................................................................................... 2
List of figures .................................................................................................................. 4
Author’s Assessment ...................................................................................................... 5

1. Introduction ................................................................................................................ 7
   1.1. Background ............................................................................................................ 7
   1.2. Aims ....................................................................................................................... 7
   1.3. Objectives ............................................................................................................ 8

2. Methodology .............................................................................................................. 9
   2.1. Modelling Philosophy .......................................................................................... 9
   2.2. Project Management ............................................................................................ 9
   2.3. Documentation ................................................................................................... 10

3. Research: Empirical Modelling .............................................................................. 11
   3.1. Principles ............................................................................................................ 11
   3.2. Tools ................................................................................................................... 12

4. Research: Agent-oriented Parsing ........................................................................... 13
   4.1. Behaviour ............................................................................................................ 13
   4.2. Writing notations ................................................................................................. 16
   4.3. Documentation ................................................................................................... 17
   4.4. A Possible Theory of Agent-oriented Parsing ...................................................... 18
   4.5. Dependency in Parsing ....................................................................................... 21

5. Comparative Study .................................................................................................. 24
   5.1. Theoretical Basis ................................................................................................. 24
   5.2. Performance ....................................................................................................... 24
   5.3. Conceptual Ideas ................................................................................................. 25
   5.4. Conclusion ......................................................................................................... 26

6. Modelling: Parser Improvements ........................................................................... 27
   6.1. Analysis .............................................................................................................. 27
   6.2. Pattern matching ................................................................................................. 27
   6.3. Agent Scripts ...................................................................................................... 30
   6.4. Containing Agents .............................................................................................. 36
   6.5. Other Extensions ................................................................................................. 39
   6.6. Testing ............................................................................................................... 41
   6.7. Summary ............................................................................................................ 42

7. Modelling: Parser Visualisation .............................................................................. 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1. Analysis &amp; Design</td>
<td>43</td>
</tr>
<tr>
<td>7.2. Initial Model</td>
<td>43</td>
</tr>
<tr>
<td>7.3. Second Iteration</td>
<td>45</td>
</tr>
<tr>
<td>7.4. Third Iteration</td>
<td>47</td>
</tr>
<tr>
<td>7.5. Summary</td>
<td>50</td>
</tr>
<tr>
<td>8. Modelling: Notation Builder</td>
<td>52</td>
</tr>
<tr>
<td>8.1. Motivation</td>
<td>52</td>
</tr>
<tr>
<td>8.2. Reading Notations</td>
<td>52</td>
</tr>
<tr>
<td>8.3. Writing Notations</td>
<td>55</td>
</tr>
<tr>
<td>8.4. Summary</td>
<td>62</td>
</tr>
<tr>
<td>9. Conclusion</td>
<td>63</td>
</tr>
<tr>
<td>9.1. Achievements</td>
<td>63</td>
</tr>
<tr>
<td>9.2. Critique</td>
<td>63</td>
</tr>
<tr>
<td>9.3. Final Thoughts</td>
<td>65</td>
</tr>
<tr>
<td>9.4. Further Work</td>
<td>65</td>
</tr>
<tr>
<td>9.5. Acknowledgements</td>
<td>66</td>
</tr>
<tr>
<td>10. References</td>
<td>68</td>
</tr>
<tr>
<td>Appendix A: Parser Documentation</td>
<td>69</td>
</tr>
<tr>
<td>Appendix B: Notations</td>
<td>77</td>
</tr>
<tr>
<td>Calc (old-style scripts)</td>
<td>77</td>
</tr>
<tr>
<td>Calc (new)</td>
<td>78</td>
</tr>
<tr>
<td>Palindrome</td>
<td>79</td>
</tr>
<tr>
<td>Appendix C: The Sand Model</td>
<td>80</td>
</tr>
<tr>
<td>The Sand BNF</td>
<td>80</td>
</tr>
<tr>
<td>Notation</td>
<td>80</td>
</tr>
<tr>
<td>Screenshots</td>
<td>82</td>
</tr>
<tr>
<td>Appendix D: Supporting material</td>
<td>85</td>
</tr>
<tr>
<td>Project Log</td>
<td>85</td>
</tr>
<tr>
<td>Project Files</td>
<td>87</td>
</tr>
</tbody>
</table>
List of figures

Figure 5-1: An example parse tree ........................................................................... 14
Figure 5-2: Basic agent operations ......................................................................... 15
Figure 5-3: The Chomsky hierarchy of formal grammars ....................................... 21
Figure 5-4: Coordination between agents ................................................................ 22
Figure 7-1: Communication between parent and child ............................................ 33
Figure 7-2: String replacements in action scripts ................................................... 34
Figure 7-3: Definition of agency ............................................................................ 37
Figure 7-4: AOP scripts ......................................................................................... 41
Figure 8-1: Text output of parse process ............................................................... 45
Figure 8-2: Visualisation model for an expression parse tree ............................... 46
Figure 8-3: Design of visualisation interface ......................................................... 48
Figure 8-4: The parser visualisation implementation .............................................. 49
Figure 8-5: Parser visualisation scripts ................................................................. 50
Figure 9-1: The Notation Builder interface ............................................................ 57
Figure 9-2: Notation builder scripts ...................................................................... 57
Figure 9-3: Grid component usage ........................................................................ 59
Figure 9-4: The grid component ........................................................................... 59
Figure C-1: Simple Sand example ......................................................................... 82
Figure C-2: Matrix multiplication using a systolic array ....................................... 83
Figure C-3: How Sand generates a parse tree ....................................................... 84
Author’s Assessment

What is the technical contribution of this project?

This project studies a style of parsing unique to empirical modelling. It builds on existing work to improve the parser. Supporting models for the parser are developed that enable notations to be created and the behaviour examined.

Why should this contribution be considered important to computer science?

Parsing is considered a well-understood area of computer science and has a firm theoretical basis. This project offers an alternative view of parsing. The project demonstrates that empirical modelling techniques can be applied to parsing. The result of this being a tool that enables new notations to be included Eden.

How can others make use of the work in this project?

The deliverables of this project allow others to build new notations in Eden. Documentation is available in order for modellers to quickly get up and running with the tools. Several notations are available as examples of using the parser (e.g. Sand). It is recommended that the reader familiarises themselves with the documentation in Appendix A as this will aid understanding of the report.

This work could be the basis for further work in this area. A list of suggested topics for research is listed in the concluding section.

Why should this project be considered an achievement?

The achievements of this project are in two areas. Firstly the empirical modelling tools have been enhanced with development of the parser, allowing others to add notations to the system. Secondly, the project is a unique piece of research and has achieved its objectives of gaining understanding into this method of parsing.
What are the weaknesses of this project?

This project has opened up many new directions for research in this area. A weakness of this is that some areas need more work. It is suggested that the work relates to other areas of computer science (e.g. natural language parsing), but there is not enough evidence to prove any such claims. The positive side of this is that it leaves scope for more research in this area.
1. Introduction

1.1. Background

Empirical modelling (EM) is a foundation of computing that emphasises the relationship between the real-world and the formal model. The EM group at University of Warwick has developed a range of tools that enable the modelling of environments. Different notations exist for different tasks, for example, a specialised notation exists for line drawing.

A previous final year project (Brown 2000) developed a parser for the tkeden environment. This allows new notations to be added to the tkeden environment. The parsing technique is different to traditional parsing in computer science. It has been termed ‘agent-oriented parsing’ because the technique uses a hierarchy of agents to parse an input.

The existing implementation is not well understood and only a few new notations have been developed. This is partly due to the lack of documentation for the parser. The implementation could be described as having some peculiar behaviours, and whether these are desirable (or not) is debateable.

1.2. Aims

The project is a piece of research into agent-oriented parsing, a novel method for parsing using the empirical modelling framework. The motivation for the project is the promising results of the existing parser (Brown 2000). The overall aim of the project is to use these ideas to develop the agent-oriented parser tool for use in empirical modelling.

Further aims of this project are to investigate the agent-oriented parser in relation to other parsing techniques and other applications in computer science.

The possible directions this project could have taken are many. These were outlined in the progress report (Harfield 2002).
1.3. Objectives

The initial objectives are to outline the project and research empirical modelling (carried out during summer vacation):

- Plan the project – write a weekly schedule and create a website to post project information
- Learn to use the empirical modelling tools – including the Eden, Donald and scout notations

The next objectives are research and analysis into agent-oriented parsing (term 1):

- Experiment with the current agent-oriented parser – write new notations using the parser and analyse existing notations (e.g. Eddi, SQL and logo)
- Analyse the agent-oriented parser – identify bugs and limitations, compare to other parsing methods
- Document the behaviour of the parser – giving the precise nature of the parse process
- Design improvements and extensions – to increase usability and understanding

The development objectives are a result of completing the analysis (term 1+2):

- Improve the parser – increased features, robustness, and error handling
- Develop debugging aids – methods of debugging notations and gaining understanding of the parser
- Develop notation tools – environments for developing notations
- Integrate with existing tools – for use in the empirical modelling group

The final objectives (term 2):

- Post analysis – analyse the power and versatility of the agent-oriented parser
- Further work – study possible project extensions
- Communicating results – plan for the presentation and the final report

The reader should refer to the progress report for a complete project schedule.
2. Methodology

2.1. Modelling Philosophy

The method used to write computer programs using Empirical Modelling is different from traditional techniques. A common structure for computer science projects is the waterfall method, where each stage is completed before beginning the next stage (Packwood 2001). This is inappropriate in Empirical Modelling. Instead, models are built-up iteratively by refining the model (Turner 2000). The modeller uses experimentation to evolve the model. A model is never finished, it can always evolve further. This process has similarities to prototyping or user-centred design, where a spiral approach enables continual assessment of a design or an implementation.

In this report, certain terms have been used more than others. It is often the case in EM that design, implementation and testing is all rolled into one, and therefore they may not be referred to individually. The terms modelling and experimentation have been used to encompass all these three activities.

2.2. Project Management

It is particularly important for this project to be well managed (Dawson 2000). The possible directions that could have been researched in this area are vast (Harfield 2002) therefore it was necessary to have a strict plan to adhere to.

Throughout the duration of the project, the project website has been synchronised with the progress of the project. A log is kept on the website to show the progress of the current work and to record any interesting conclusions along the way (see Appendix D). The models and extensions to the parser have been made available on the website for members of the empirical modelling group to experiment with. Documentation for the parser is also available on the website.

The project website has been useful for two reasons. First it has been a measure of the progress of the project and secondly it has been a useful resource for others who are working in related areas.
2.3. Documentation

Each of model on the website has documentation. The reader is encouraged not to take the words in this report for granted. Instead, experiment with the scripts and draw your own conclusions.
3. Research: Empirical Modelling

This section introduces the principles of Empirical Modelling. The material here does not delve deeply into the theory of EM but should introduce the reader to the EM concepts of project.

3.1. Principles

Empirical Modelling is creating a representation of an environment based on the experience gained from interactions with the environment. The model is based on three concepts: observation, dependency and agency (EM 1999).

3.1.1. Observation

The first concept that must be understood is that of an observable. Literally this is something that can be observed (e.g. an item of information about an object or the object itself). An observable represents an element in the environment (possibly the real-world) and a collection of observables represent the state of the environment.

3.1.2. Dependency

In an environment there can be dependencies between the observables. For example, the position of a lamp is dependent on the position of the table. It is also dependent on gravity, and it is interesting to note that empirical modelling does not assume that an environment must model the real-world. The modeller is responsible for adding these features (such as gravity) to a model.

3.1.3. Agency

An environment is static without any interaction. Agency permits interactions with the environment. An agent is something that interacts with the environment, thereby changing the observables and dependencies. For example, in the above environment, an agent could be a human that moves the table. The result of this would be the table observables have changed, and due to dependency the position of the lamp will have changed because it is on the table.
3.2. Tools

The EM group have developed a number of tools for modelling using these principles. The primary tool is the Eden (Evaluator of DEfinite Notations) environment, containing an interpreter for the Eden notation. The notation allows the definition of observables and dependencies. It also includes elements of a procedural language allowing functions and procedures. The syntax is similar to C.

There are several variants of the Eden interpreter, the most common of which is tkeden, a graphical environment. Others include ttyeden, a command line interpreter, and dtkeden, a distributed environment.

There exist many definitive notations in tkeden, designed for specialised tasks, including:

- Donald – Definitive Notation for Line Drawing
- Scout – SCreen layOUT
- Sasami – OpenGL-based 3D display engine
- Eddi – the Eden Definition Database Interpreter
4. Research: Agent-oriented Parsing

The work by Brown (2000) developed an agent-oriented parser. The implementation has been used to successfully develop notations for Eddi, SQL and Logo. However these notations contain some unpredictable behaviours and undesirable side effects. The aim of this section is to evaluate the existing agent-oriented parser and understand its behaviour. The research will uncover the possibility of a theory for this style of parsing in the empirical modelling context.

Note: It is not the aim of this section to study every detail in the AOP. The documentation (Appendix A) is the resource for understanding how to use the parser and it is a prerequisite for the sections to follow.

4.1. Behaviour

In order to understand the agent-oriented parser, it is useful to put aside any prior knowledge of parsing. As will be studied in a later section, this parser is quite different to traditional parsing methods. The behaviour is not concerned with a particular AOP implementation but details the concepts of any AOP.

4.1.1. Agents

An agent is an independent entity capable of acting on and interacting with an environment. In the agent-oriented parser, each agent will take an input string and a rule, and it will determine whether the rule can be applied. If the rule is applied, then the result may be passed to other agents. The agent will fail if it cannot apply the rule, which corresponds to a rejection of the input. If all agents in the hierarchy are able to apply a rule, then the input has been accepted.

The aim of an agent is to identify an important feature of the input. This observation allows the agent to understand something about the input and breaks the problem into smaller chunks. These sub-problems are examined by a new agent and the observation continues. When all the features of the input have been observed the parser has finished.
Figure 4-1 shows an example parse tree for an expression parser. Each agent observes an important part of the input and creates child agents to parse the sub-problems. In this case it is the operators that are the most important features of the string.

4.1.2. Rules

A rule defines the action an agent must take to parse an input. Every rule has a pattern that must be matched on the input and an operation that specifies how this pattern can be matched. A pattern can be a string or a regular expression. The basic operations are:

- Literal – match the pattern to the entire input.
- Prefix – match the pattern at the beginning of the input.
- Suffix – match the pattern at the end of the input.
- Pivot – match the pattern somewhere in the input (the first occurrence of the pattern).
- Split – match the pattern at all occurrences in the input.
A rule also specifies each rule that must be applied by the child agents. The number of
children an agent will create is dependent on the operation (see Figure 4-2).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Children</th>
<th>Matches beginning</th>
<th>Matches end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>0</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Prefix</td>
<td>1</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Suffix</td>
<td>1</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Pivot</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Split</td>
<td>unlimited</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-2: Basic agent operations

4.1.3. Blocks

There are situations when it is convenient to consider blocks of input. A block can be
defined as having a start string and an end string. The motivation for defining a block
is to instruct an agent to not match a pattern inside a block.

For example, consider the input (with java-style syntax):

```java
mystring = "this is my string + " + anotherstring;
```

And assume that one of the agents has the following input and must match an addition
character:

```java
"this is my string + " + anotherstring
```

Using the pivot operator, the first occurrence of the addition character is within the
string block. This is not the desired outcome; it should be the second addition character
that is pivoted on. The reason for this is that the agent’s aim is to simplify the problem.
If the agent pivoted on the wrong character then there would be two difficult strings to
deal with:

```java
"this is my string
```

and

```java
" + anotherstring
```
Therefore, a block is defined. The beginning of the block is marked by a double quotation mark and the end of the block by a double quotation mark. In the rule, a list of blocks can be defined that must be ignored by the agent. When an agent applies a rule that contains ignore blocks, it simply skips over any characters within these blocks. For example, the above agent would now view the input as:

```
"this is my string + " # anotherstring
```

Other inputs that might require blocks are languages containing brackets, curly braces or begin…end statements. For a detailed example, refer to the documentation.

4.1.4. Scripts

A script in the agent-oriented parser defines what actions an agent performs in the environment. Without scripts an agent does not perform any observable actions. A rule may contain a script to be executed by the agent. In the existing implementation the script can be Eden or any other notation (provided that notation is ‘switched to’ first). The scripting language is specific to an implementation of the AOP and therefore its behaviour cannot be generalised. Scripts will be studied in more detail during the next sections.

4.2. Writing notations

To experiment with the agent-oriented parser, it was necessary to study and develop some notations. Brown (2000) gives some example parsers, including Eddi, palindrome and pl0. The Eddi notation is further developed by Beynon (2002) and has been used extensively for the CS233 database course.

In order to evaluate the process of creating notations, a task was set to build an expression parser. The notation would evaluate expressions and output an answer, similar to the operation of a calculator. Although this notation could not be described as a definitive notation, it does however use definitive principles to evaluate expressions.
4.2.1. The `%calc` notation

This notation is relatively simple compared to Eddi. It consists of:

- Four expression rules that observe addition, negation, multiplication and division.
- Two rules that observe brackets.
- A rule that observes a natural number.
- A rule that prints an error message.

The behaviour of the parser benefits from the use of dependency. Each of the four expression rules setup a dependency definition such that the value of the agent variable is the value of the two child variables applied to the arithmetic function (e.g. addition). In Eden the dependency is of the format:

\[
\text{answer is child\_1\_answer + child\_2\_answer;}
\]

At the time of creation both the child answers are undefined, but later when these have a value the answer will be updated. A command is executed when all the child agents are complete that will print out the answer.

The above is a description of the original calculator. This was used as a basis for extended calculator notations that added new features (e.g. negative numbers) and used different script formats. The original calculator can be found in Appendix B.

4.3. Documentation

Given the need among the EM group for a better understanding of the agent-oriented parser, the results from the research were used to write some documentation. This is considered one of the most important deliverables of the project. For anyone new to the parser, the development of new notations has proved to be difficult. Even the relatively simple `%calc` notation took a considerable time to debug. Therefore documentation should increase the accessibility of the parser and ease the development of notations.
The calculator notation uses all the features of the AOP. It has a basic set of rules that can be easily understood, as it resembles an expression evaluator. With this in mind, the documentation is of the form of a tutorial that develops the calc notation. The benefits of this are that readers will be able to experiment with parser. Some exercises are given in the documentation for extending the calc notation.

The documentation captures the iterative nature of empirical modelling. One of the main benefits of the AOP over existing parsing methods is that notations can be developed in this way (refer to Comparative Study). The first model of the calc notation accepts only positive integers. The next iteration includes expressions containing the addition (+) sign. A further iteration allows bracketed expressions.

The complete documentation is found in Appendix A.

4.4. A Possible Theory of Agent-oriented Parsing

The observable behaviour of the parser has been examined in order to understand the existing parser. So far, there is has been no mathematical basis for this behaviour. Although a complete theory of the agent-oriented parser is beyond our understanding at this stage, it is worthwhile to reflect on a possible theory.

An agent-oriented grammar (AOG) consists of:

- a set of agent rules
- an initial rule
- a set of block definitions

An agent rule consists of:

- an operation
- a pattern to be matched on the input
- a list of names of agent rules (for each child agent)
- a fail rule (the name of an agent rule to apply if the pattern cannot be matched)
- a set of block names to ignore
A rule could be described as a function from a string to a set of pairs of strings and rules. Let \( f \) be an agent rule function, \( i \) be an input string and \( R \) be a set, such that \( f : i \rightarrow R \). The set \( R \) contains the agent result such that \( (s,g) \in R \) if \( s \) is the a resulting substring from performing the function \( f \), and \( g \) is the corresponding agent rule function to be applied to \( s \).

An agent-oriented grammar is deterministic. There is only one possible parse tree of each input.

An agent-oriented grammar could be defined as well-formed if the following criteria are met:

- The initial rule is well-formed.
- A rule is well-formed if:
  - The operation is a member of the set (literal, prefix, suffix, pivot, split).
  - The child rules are well-formed rules.
  - The fail clause is a well-formed rule.
  - The ignore clause is a set of well-formed blocks.
- A block definition is well-formed if:
  - It specifies a start and end string.
  - The sub-blocks are a set of well-formed blocks.

4.4.1. Completeness

Some useful results can be obtained about the completeness of the agent-oriented parser. Assuming that the above definition of an agent-oriented parser is complete, it can be shown that some restricted agent-oriented parsers are complete.

Consider an agent-oriented parser \((\text{AOP}_{\text{ piv}})\), the same as above, except that it only has only two operations: pivot and literal. The pivot and literal operations are exactly the same as their AOP counterparts described above. In order for \(\text{AOP}_{\text{ piv}}\) to be complete, it must be shown that each of the AOP operations can be defined in terms of the pivot operator. Consider each case:
• Prefix – This operation is equivalent to performing a pivot operation and then performing a literal operation on the second child to check it is the empty string.

• Suffix – Using the same method as prefix, but perform the literal operation on the first child to check it is the empty string.

• Split – This operation is equivalent to performing a pivot on the input and then performing the same operation on the second substring, and keep repeating this until no more pivots are found.

This proves that any language that can be described using AOP, can also be described using \( \text{AOP}_{\text{pin}} \). This parser also has the minimum number of operations.

Consider an agent-oriented parser (\( \text{AOP}_{\text{null}} \)), the same as above, except that it does not contain the literal operation. In order to show that \( \text{AOP}_{\text{null}} \) is complete, it must be shown that the equivalent of a literal operation can be achieved using the other operations. The literal operator matches a pattern to the entire input string. This can be achieved by matching the prefix and then checking that the child substring is the empty string. This seems feasible. The problem that arises is which operation to use to check the empty string. The only operation that does not create children is the literal operation, therefore it would be impossible to terminate the tree. For this reason, \( \text{AOP}_{\text{null}} \) is not complete.

4.4.2. Equivalence

It would be useful to know what languages can be accepted using the agent-oriented parser. Are we limited to regular languages, or context-free languages? This is not the case, Brown (2000) has already shown an example of a simple palindrome parser and has given a parser for a language which cannot be expressed in a context-free language. It appears that the AOP is as powerful as a Turing machine, but this has not been proved.

Despite the agent-oriented parser's obvious power, it is not correct to try to classify the types of languages that can be parsed. So far, no attempt has been made to classify AOP languages into Figure 4-3. The reason for this is that the agent-oriented parser is not primarily concerned with accepting and rejecting inputs. An integral part of the
AOP is the action (or side-effects) that are performed. The parsing and the actions should not be separated because actions are the basis for agent communication. This is difficult to relate to traditional parsing.

![Chomsky hierarchy of formal grammars](image)

**Figure 4-3: The Chomsky hierarchy of formal grammars**

The existing implementation of the parser could not parse an identifier (a letter followed by a possibly empty string of letters and numbers). Another simple language that cannot be parsed is the language that accepts only the empty string. A rule can be written that parses the empty string, but due to an agent removing the white-space around an input, any string containing just white-space will also be accepted. This issue is addressed in the section on parser improvements.

### 4.5. Dependency in Parsing

The agents within a parse tree are dependent on both their ancestors and descendents (Figure 4-4). Given the emphasis on dependency in empirical modelling, why not think of the parse process as building up a tree of dependencies? The obvious reason for this is that when an input is parsed it only needs to be processed once. It is inefficient to setup a tree of dependencies if the tree is static and is only required for a short period.
Figure 4.4: Coordination between agents

Although it appears unwise to not use dependencies between agents, it might be useful to research the applications of such a model. After some experimentation, a small model was developed that parses inputs by setting up a tree of dependencies. Several variations of this model exist on the attached CD in the “true_em_parser” directory (note that this model is called “The True EM Parser”). Some features of such a model are:

- An agent is dependent on an input string and the grammar.
- An agent’s input string is dependent on its parent agent. Therefore a child agent is dependent on its parent.
- The result of an agent is dependent on the result of its child agents. If a child agent fails to parse, then the result of the parent will also be to fail. Therefore a parent agent is dependent on its children.

A model with these features may be useful in an environment for writing code. When coding, only small changes are made at a time and so the input strings are similar. For example, if a large script is being edited then the effect of small changes can be seen without reparsing the entire script. The reason this could be useful is that small changes affect only a few child agents. The dependencies will cause these changes to propagate through the parse tree and the overall result can be observed. In the AOP, the result would be another script (to be executed).

Such ideas are behind the generation of language-based environments (Reps 1983). These environments are editors that have knowledge of the structure and semantics of
a programming language. With this knowledge, input strings can be examined to determine if they contain semantic errors (for example, undeclared identifiers). An editor would keep a parse tree up-to-date and could highlight errors as they become apparent during coding. Reps presents several algorithms for keeping parse trees consistent and up-to-date.

It appears that empirical modelling concepts could be applied to this area of research. Reps uses Attribute Grammars to describe the structure and semantics of languages, but a “True EM Parser” (as described above) could be used instead. This would automatically maintain a parse tree using dependencies. The difficulties that arose in Reps’ work, i.e. efficiently maintaining large amounts of data, are also applicable here.

Unfortunately this area is not developed further in this report, but it could be a project in its own right (see Further Work).
5. Comparative Study

This section aims to compare traditional parsing to the agent-oriented approach. By analysing these techniques to find similarities and differences, the key benefits will be identified.

5.1. Theoretical Basis

Areas of Computer Science regarding parsing and grammars have been studied in great detail. It is safe to say that conventional parsing techniques are well understood. The tools available for compiler construction are very successful and have a wide applicability (Grune 2000). As studied in the automata and formal languages course, these methods are strongly grounded in mathematics (Goldberg 2001).

The agent-oriented parser has little theoretical basis. It can be likened to how humans parse language, which is also low on mathematical foundation (as not enough is understood about the brain). In the previous section a possible theory for the agent-oriented parser was outlined, but it lacks the purity of traditional parsing techniques.

5.2. Performance

The performance of the agent-oriented parser is dependent on how the rules are arranged. The operations that each agent can perform take varying amounts of time. A literal operation is very quick because it is a basic string comparison. A pivot operation takes the longest because it must search through the entire string to find a pattern match. In between these in terms of performance are the prefix and suffix operations, which only have to search the beginning and end respectively. In many grammars, the order of some of the rules does not matter. The calc notation for example, the behaviour is not effected if an addition sign (+) is observed before or after a minus sign (-). These two rules can be swapped around without changing the language. Therefore, performance can be optimised by either:

- Ordering the rules in terms of performance.
• Ordering the rules in terms of popularity (based on the probability that they are observed).

The use of regular expressions, as studied later, also has a negative effect on performance.

Compared to traditional parsing techniques, the agent-oriented parser has very poor performance. The reason for this is that the AOP requires an infinite look-ahead. It requires that the entire string be accessible in order to parse. Some operations require the entire string to be searched. This is in contrast to traditional parsers that process each character at a time and use just a few characters of look-ahead. The AOP may search an input string many times as each agent observes a feature, whereas automata work from left-to-right (or right-to-left in some cases) discarding the characters as they are processed. This method is far superior in terms of performance.

5.3. Conceptual Ideas

It has been shown that agent-oriented parsing is weak in both the theoretical basis and the performance. At this point some might right off this work as pointless, but there is one further aspect that must be considered.

From a modeller’s point of view, an agent-oriented grammar has some desirable properties. The parser has a limited number of operators (literal, prefix, pivot, etc), each of which have a physical operation that can be observed. Traditional parsing techniques Automata and Turing Machines perform operations (like moving items onto a stack) which are difficult to relate to the overall task. Unlike Automata and Turing Machines that operate like black boxes, the tasks in the agent-oriented parser are observable. Each agent performs a specific task that breaks the input down into smaller tasks (i.e., partition the problem). As the parse process relates to physical operations that a human could perform, it makes the parse process easier to follow and understand.

The second point to be made about the agent-oriented parser is that it supports an iterative development model. A notation evolves as shown in the documentation, the calc notation is built-up and at each stage a significant feature is added to the grammar.
It could be argued that this is possible with traditional parsing techniques. The advantage of the AOP is its relation to empirical modelling. Since the grammar and rules are observables, these can be modified (through agency), and the result is a modified parser. This makes the tkeden interpreter an ideal environment for developing parsers iteratively. Compare this to developing parsers using yacc/lex/javacc which have to be compiled and run each time a change is made.

It has already been explained how the rules of a grammar are observables. There are further advantages to this approach. These observables are available to the whole environment, and can be shared between notations. For example, a rule can be defined that observes an ‘identifier’ (an equivalent regular expression might be \[a-zA-Z][a-zA-Z0-9_.]*\). This rule can be used by several notations, it might be used in Eddi to observe a table name or it might be used in Sand to observe an element name. The agent-oriented parser has instant reusability.

There is nothing to restrict a rule being an observable that contains dependency. A grammar does not have to be a static set of rules. A grammar may be dynamic. An example of a grammar that contains dependency is the palindrome parser (Brown 2000). An extended palindrome parser can be found in Appendix B.

5.4. Conclusion

_The emphasis of Empirical Modelling is to establish a stronger and more permanent link between what we observe in the ‘real-world’ and our model of it._

_(EM 2002)_

It has been shown that the agent-oriented parser is in no way a replacement for traditional parsing. Instead it demonstrates that there are some uses for this unique method for parsing. In particular, the agent-oriented parser aims to bridge the gap between traditional parsing techniques and a human model of parsing. It does not perform well against traditional parsing, but the agent-oriented parser does afford the modeller a simple parsing method closer to the real-world.
6. Modelling: Parser Improvements

This section critically analyses the existing parser and develops a new version of the parser.

6.1. Analysis

The research in the prior section realised that the existing parser has some undesirable qualities. A critical analysis uncovered a list of problems or inconsistencies in the parser (Harfield 2002).

Opinion of the existing parser is that it is not easy to understand. The modellers of the logo and Eddi notations found that it is difficult to comprehend what the parser is doing, making debugging a big problem. Although documentation has been written for modellers, there is a need to make changes to the parser to ease the modelling of notations further.

The main issues with the existing parser:

- Pattern matching is limited. For example, unable to parse identifiers.
- Defining scripts is complicated.
- The scripts created by a notation cannot ‘switch-to’ another AOP notation.
- Error messages are not useful.

6.2. Pattern matching

6.2.1. Analysis

The existing parser had three operations for pattern matching:

- Read all – This operation matches all the characters in the input string. Each character in the input must be in the range of characters specified by the pattern. E.g. a word could be defined as having characters only in the range “a”~“z” or “A”~“Z”.
• Read prefix – This operation matches characters at the beginning of the input string. Each character in the prefix must be in the range of characters specified by the pattern. The operation can match as many characters as possible on the input, or it can match an exact number of characters (if specified).

• Read suffix – This operation matches characters at the end of the input string, in the same manner as “read prefix”.

The shortcomings of this approach are that only a restricted set of patterns can be matched. Patterns specify ranges of characters, so it is not easy to match combinations of characters.

As mentioned previously, the existing parser cannot parse identifiers. Here an identifier is defined as a letter of the alphabet followed by a (possibly empty) list of alphanumeric characters. This reason for this will be explained in more detail.

Recall the definition of “read all” operation, noting that if the operation encounters the empty string it will accept. For this reason, the “read all” operation alone is not powerful enough to parse an identifier. The approach taken by most notations that use the existing parser is to use the “read prefix” operation on the input first and then pass the remaining substring to “read all”. The first agent will remove one character from the beginning of the input string if it is a letter of the alphabet. The second agent (a child of the first) will match the remaining string if it contains only alphanumeric characters. This appears to achieve the correct behaviour.

Although this arrangement of agents does parse all possible identifiers, it is not the correct behaviour. It parses identifiers, but it also parses some inputs that are not identifiers. It is incorrect and causes errors because the set of inputs it parses correctly is greater than the set of identifiers. Some examples of inputs it will parse:

```
myvar
m yvar
m yvar
```

The first example is an identifier. The second and third examples contain white-space after the first character and are therefore clearly not identifiers because an identifier cannot contain any spaces. All of these inputs will be accepted by the above rules.
The reason for these inputs being accepted is that each agent strips the white-space from the beginning and end of the input. Two agents have to be used parse the identifier and so white-space can be allowed in between. This problem would not exist if the pattern matching operations were more powerful.

6.2.2. Design

The solution to this is to design three new operations to replace the existing pattern matching operations. The operations are given new names so that the existing “read all/prefix/suffix” operations can still be used. This means that old notations that use these operations will function as before.

The new pattern matching operations must be more powerful. Pattern matching is used in many areas of computing, including lexical analysis, shell scripting and database searching. A common format for pattern matching is found in perl. Patterns to be matched are described using regular expressions. They are called regular after the regular languages that can be described using them, although perl regular expressions are more powerful than this (Wall 2000).

The Eden environment has functions to evaluate perl-compatible regular expressions. This style of regular expression is well-known and is used throughout many programming languages and text processing applications. It is also similar to the format used by lexical analysers. It is an ideal format for pattern matching because of its familiarity. Given the availability of these functions in Eden, the design of pattern matching operations should be consistent with these functions.

An example of the use of regular expression function (in Eden) that matches an identifier:

```plaintext
regmatch("^[a-zA-Z][a-zA-Z0-9]*", input);
```

The three new operations to be introduced are:

- `literal_re`
- `prefix_re`
- `suffix_re`
Their behaviour is similar to the basic operations: literal, prefix and suffix. The
different is that instead of the rule defining a string to be matched, the rule will define
a regular expression to be matched.

An example rule would be:

\[
\text{ident} = \left[ \text{"literal\_re"}, \text{"[a-zA-Z][a-zA-Z0-9]*"} \right];
\]

### 6.2.3. Implementation

The implementation required functions to be written for each of the new operations.
These functions are called by the agent to do the pattern matching. The functions take
an input string and a pattern as inputs. If the function is able to pattern match then it
returns the token matched and any substrings (to be eventually passed to the next
agent). If no pattern match can be made then the function returns a failure.

The 'regmatch' function included in Eden is used as the engine to process the regular
expression evaluation. This function returns a list of possible matches. The operation
functions use this information to determine if a match can be made. The 'prefix\_re'
operation checks for a match at the beginning. The 'suffix\_re' and 'literal\_re' operations
do similar checks. If a match is made then it is the job of the operation function to
determine what substrings are left (for the prefix and suffix operations only).

### 6.2.4. Conclusion

The problems with the existing parser have been overcome. The rule defined above
correctly observes identifiers and nothing else. This also solves the problems with
defining integers, where previously the “number” rule accepted the empty string.

### 6.3. Agent Scripts

#### 6.3.1. Analysis

The actions that an agent performs in the environment are determined by an agent
script language. The existing parser has what is described as “a very simple language”
with five instructions:
• declare – create new script variables
• setparas – indicates which script variables should be passed to the child agents
• allparas – indicates a script variable to be passed to all child agents
• execute – executes a string of Eden code
• later – executes a string of Eden code after the child agents have completed

The two execution instructions allow script variables to be included in the string using two format specifiers: $$ and %%%. The first of these substitutes the script variable into the string. The second substitutes the variable specified in the script variable. To understand this it is useful to recall the assembly language “Load Accumulator” operation, which allows you to load a value from a memory location (like $$) or load a value from a memory location specified by the given memory location (like %%).

There are script variables that are available to all scripts. These are variables containing information about the agent:

• v_string – the input string
• v_substrs – a list of substrings to be passed to child agents
• v_paras – a list of parameters passed to the agent

Despite the claim that this is a simple script language, it has proved to be difficult to develop notations with it. The reasons include:

• Using string substitutions, it is difficult to determine whether %% or $$ is needed. Often involves guess work.
• Passing script variables between agents requires first that the variable is declared and second that it is passed using “setparas” or “allparas”. Only after this can it be used in an execute instruction. Considering that the majority of agents will be passing variables around in order to communicate, this is a burden on the modeller to maintain a model of all these variables in their head.
• The automatic script variables (v_string, etc) are inconsistent between agent operations. For example, the pattern matching operations need a variable containing the token matched, and in some cases this appears to be stored in v_string. There also are some undocumented variables that have been used in scripts.
- The instructions are generally not intuitive. Even experienced modellers have trouble remembering when and where to declare variables, and whether to use “setparas” or “allparas” for passing variables.

- Debugging is difficult because of the lack of information the modeller receives about an error.

### 6.3.2. Design & Implementation

An agent generally only uses one script variable to communicate with its parent and one script variable to communicate with a child agent. This observation questions the need to declare multiple script variables. A simpler strategy would be to assign each agent a unique script variable. This would make the “declare”, “setparas” and “allparas” instructions no longer required in the scripting language.

This design still allows effective agent communication. An agent has a unique variable, which enables it to communicate with its parent. This variable allows the parent to send one piece of information to the child when the child begins execution and the child can return one piece of information.

See Figure 6-1 for the stages in the agent communication model. Before the parent agent executes any scripts it creates a child variable (for each of its children). When the agent executes its script, the script allows data to be placed in the child variable (stage 1). After this, the child agent is created and performs any actions. The child agent has access to its own script variable (stage 2). When the child agent has completed, it is no longer required but its associated variable persists. This allows the parent agent to examine the data returned by the child when it executes its “later” script (stage 3).
Figure 6-1: Communication between parent and child

The reason that multiple script variables are not needed is because variables can contain Eden lists. This means that agents can pass large list structures if necessary.

With this design the only instructions that are needed to describe scripts are the execute instructions. There are two such instructions, which in the design will be referred to as “now” and “later”. The “now” instruction executes a given script. It performs this execution as early as possible, which is when the agent has completed its observation of the input. This is before any child agents are executed. The “later” instruction uses the same format to execute a given script, but its execution is delayed. It waits until after all the child agents have completed before executing the script.

Another major concern of the existing parser is the substitution of variables using “%%%” and “$$”. Since the design does not allow multiple variables, only a finite number of variables exist for each agent. This means the substitution of variables can be simplified. There is only one script variable for each agent so this can be given a unique symbol. The variables belonging to the children could also be substituted and these can each be given a unique symbol. The other strings that could be substituted are the input string, the token string and any substrings. Each of these can be given unique symbols instead of using the general “%%%” and “$$”.

The scripts that are executed by an agent are the action it performs on the environment. The definition of an agent’s action must be included in the rule in the same way as the existing parser. In order to avoid conflict with existing notations (which uses the
“script” tag), a new tag is defined called “action”. The template for an agent with this new script is:

```plaintext
myagent = [ ...
    [ "action",
      [ "now", "somecommand" ],
      [ "later", "anothercommand" ],
    ...
  ];
```

The action definition is a list with a head element being the string "action" and the proceeding elements being the instructions. Each instruction is a tuple, with its first element being either "now" or "later" indicating when the script should be executed. The second element in the tuple is the script. A script may contain any strings substitutions from Figure 6-2.

<table>
<thead>
<tr>
<th>String</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v</td>
<td>the script variable of the agent</td>
</tr>
<tr>
<td>$i</td>
<td>the input string</td>
</tr>
<tr>
<td>$j</td>
<td>the input string (as a variable)</td>
</tr>
<tr>
<td>$t</td>
<td>the matched token/string</td>
</tr>
<tr>
<td>$sn</td>
<td>the nth substring of the input</td>
</tr>
<tr>
<td>$pn</td>
<td>the script variable of the nth child agent</td>
</tr>
<tr>
<td>$$</td>
<td>$</td>
</tr>
</tbody>
</table>

**Figure 6-2: String replacements in action scripts**

The design allows for an unlimited number of scripts to be executed. There can be more than one of each “now” and “later” instructions.

The implementation of this design required that old-style scripts still function correctly. It is possible that a rule may contain definitions for old and new style scripts, in which case a warning is displayed. By old-style scripts what is meant is script
definitions beginning with the tag “script”, and new-style scripts beginning with the tag “action”.

To implement the design the parser has been split into components to allow for functionality to be shared between old and new style scripts. As the design uses a simplified form of communication using just one variable, the agent can store this variable in the same manner that it stores variables from the existing parser. Some changes were needed to recognise the “action” tag and call the correct script functions. A new function processes the “action” definition and another specifically deals with string substitutions.

6.3.3. Conclusion

This model for script actions is much simpler than the old style scripting. There is no need to be concerned with multiple parameters, variable names or difficult string substitutions. The “action” definition is as powerful as the existing parser. There is nothing that can be achieved in the existing parser that cannot be in this new model.

The simplicity and conciseness of notations developed with this model can be compared to existing notations. To see the advantages, examine the differences between two rules with the same behaviour. The old rule:

```plaintext
eexpr =
    [ "pivot", "+", [ "expr", "expr" ],
    [ "script",
        [ "declare", "$a", "$b" ],
        [ "setpars", ["a"], ["b"] ],
        [ "execute", "$% is % + %;", "$v_paras[1] $a", "$b" ] ],
    [ "fail", "expr2" ] ];
```

The equivalent rule using the “action” tag:

```plaintext
eexpr =
    [ "pivot", "+", [ "expr", "expr" ],
    [ "action",
```
[ "now", "$v is $p1 + $p2;" ]
[ "fail", "expr2" ];

The number of lines for the "script" definition in the old rule is five, whereas the new rule achieves the same behaviour in only two lines.

The ability to debug notations has benefited from breaking the implementation into smaller functions (compared to the existing parser that consisted of a few large functions). An example of this is the "aop_action_execute" function, which takes a string as an input to be executed. Modification of the function allows the execution to be switched off and the output printed:

```haskell
func aop_action_execute {
  para script_str;
  writeln(script_str);
}
```

A variation of this could print the output and execute, allowing the modeller to determine the exact location of an error.

6.4. Containing Agents

6.4.1. More agent principles

One definition of an agent is "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" (Russell 2003). Figure 6-3 shows that an agent is a self-contained unit. This means that the only way an agent can interfere with another agent is through the environment.
In the agent-oriented parser, an agent perceives an input string and the grammar rules. The process of applying a rule to a string is internal to the agent and should not be affected by the environment. The actions an agent performs are the creation of sub-problems (to be tackled by an agent) and scripts that modify the environment.

6.4.2. Distributed agents

Due to the procedural nature of the parse process, the existing implementation could be viewed as being one agent that solves each observation sequentially. Another way to look at this is to use a fixed number of agents. These agents could be distributed across machines and networks.

A task could be defined as taking an input string and outputting actions and tasks. There would be a centralised pool of tasks, and any agent would be able to access the pool and retrieve a task. On completion of the task, the agent would execute any resulting scripts in the environment (actions) and add any child tasks to the pool. Given how a hierarchy of agents is built up, initially there would be one task, and this would be likely to be split into multiple tasks by the first agent. The amount of tasks would be likely to increase exponentially as the parse tree gets deeper.

This would be useful if the input was very large. Agents could be spread across many machines in a distributed environment. This processing power could be what is needed for parsing natural language.
Although this idea would be ideally suited to the empirical modelling framework, it is not going to be considered further. It is unlikely that there will be inputs large enough to require such large processing power.

6.4.3. Analysis

The problem with the existing parser is that the script definitions (in the rules) cannot switch to other AOP notations. Scripts are limited to using Eden, Donald, Scout and Sasami. This means that you cannot build AOP notations on top of other AOP notations. There is a work around, using the ‘todo’ function which delays the execution of a script. It is used by SQL0 to execute Eddi statements, but it introduces other problems (the CREATE ... SELECT bug).

Analysis of the problem revealed that the existing parser uses many global variables. These are used by the agent to reduce the amount of data that is passed through functions. This means that you cannot execute these agent functions whilst another agent is in progress. If a notation calls another notation then two parsers must be able to run simultaneously without affecting each other.

The solution to this problem is to make sure that agents obey the principles outlined above (see Figure 6-3). In this case, an agent's variables should be local to the agent, and other agents should not interfere with these variables. The only shared variables should be those that are used for communication (e.g., passing results around).

6.4.4. Design & Implementation

The implementation of these changes required the majority of the parser to be re-written. The existing parser used many global variables that should be local to each agent. Therefore the structure of the parser had to be changed significantly.

Whilst re-writing the parser, it became clear that the operation of the parser should be separated out into sensible modules. There should be some consistency throughout the modules. The reason for this is if someone wants to make a modification to the parser then it should be easy to understand and extend its capability. An example of this is that if a new agent operation was to be added, then it should not be necessary to have to change every area of the parser that is linked to agent operations. There should be a
module that deals with operations and this should be the only part that has to be modified. This flexibility has been built into the parser.

In order to avoid name clashes, a naming convention was used. All functions, procedures and variables used by the parser are prefixed with “aop_“. Any functions related to operations are prefixed “aop_op_“. Any functions related to agents are prefixed “aop_agent_“. Other extensions that were made to the parser also follow a similar convention. There are two reasons for this, firstly there are less likely to be name clashes with other models (as there are likely to be other modellers who use the identifier “agent”). Secondly it makes the parser more understandable. During debugging it is useful to view the definitions together in the “Eden definitions” window.

6.4.5. Conclusion

Although this improvement to the parser required many modifications to the structure of the existing parser, it was a worthwhile exercise. The new parser will allow extensions to be added without interfering with the parser code. This will be seen when further debugging aids are added in the next section.

The new parser allows an agent-oriented parser to execute scripts that call another agent-oriented parser. An example of where this is useful is in the SQLedd environment. Eddi is a data definition language and a data manipulation language, rooted firmly on the relational database model. A parser has been developed that translates SQL into Eddi, giving the user a choice of two notations for database management. Using the new AOP, it should be possible for the SQL parser to execute Eddi code in parallel with its own parser.

6.5. Other Extensions

A further area of the existing parser that caused the user difficulties is error handling. Development of the above improvements realised the need for a structure for raising errors. Generally in parsers, there are two types of error:

- Recoverable errors – These errors do not cause the parse process to halt, but usually display a warning to the user.
• Irrecoverable errors – These errors cause the parse process to halt.

The existing agent-oriented parser is generally only concerned with irrecoverable errors. There are three possible reasons why the parse process may halt:

• The parser rejects the input.
• The grammar is badly defined. Examples include missing rules and incomplete rules.
• A script causes an error.

With the existing parser, if it was unable to accept an input then it would prompt the user to enter a replacement for the part of the string that could not be accepted. This is inappropriate for large strings. Furthermore, it rarely proves to be useful because the agent only tries to match the new string with the last attempted rule. The difficulty with error detection in parsers is that the position at which the error occurred may be unrelated to the position of the actual error made by the user (Grune 2000). The replacement feature does not benefit the modeller, and therefore it has been removed in the new parser.

A common model for raising errors in the agent-oriented parser was developed. The design includes a function that raises an error or a warning, which can be used as follows:

\[
\text{aop_error(type, description)};
\]

Where \textit{type} is a constant (either AOP_ERROR or AOP_WARNING) and \textit{description} is a string containing the error message.

A Boolean variable (\texttt{aop\_show\_warnings}) can be set to determine if warning messages are shown or suppressed.

The use of these functions allows more appropriate errors to be used to inform the user of the result of the parse process. As the parser develops further, and as corrections are made, it is envisaged that these functions will allow errors to be handled in a structured way.
This concludes the improvements to the parser. The latest version of the parser can be found on the CD. Figure 6-4 shows a description of the scripts created.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action.e</td>
<td>Agent script actions</td>
</tr>
<tr>
<td>agent.e</td>
<td>The core agent process</td>
</tr>
<tr>
<td>error.e</td>
<td>Procedures for raising errors</td>
</tr>
<tr>
<td>operations.e</td>
<td>The agent operations module</td>
</tr>
<tr>
<td>parse.e</td>
<td>The main parser functions</td>
</tr>
<tr>
<td>Run.e</td>
<td>Load the agent-oriented parser</td>
</tr>
</tbody>
</table>

**Figure 6-4: AOP scripts**

### 6.6. Testing

The EM development cycle includes testing at each stage of the modelling process. This testing could be classified as unit testing. It is also necessary to perform testing at the end of implementation on the complete parser. The final testing plan details the range of tests to be carried out:

- **Platform testing** – Tkeden is available on several platforms including Sun Solaris, Linux and Microsoft Windows. Some differences were found between these implementations, the main one being that the Windows version is more likely to crash. The parser performed as expected on all three operating systems.

- **Eden tool varieties** – The parser was tested on both tkeden and ttyeden. Although no visual notations (that use Donald, scout, etc) will work on ttyeden, command line notations like Eddi and calc can be used. No testing has been carried out on dtkeden, but it is expected to perform the same as tkeden.

- **Version varieties** – There exists many versions of tkeden. The latest version at the time of writing is 1.46, which was used for the majority of the project. The parser has been tested on versions 1.43 onwards. It will not function correctly on versions of tkeden prior to this because of important changes made in this version.
• Conflicts between models – It is possible that variable/function names may conflict with other models. The chance of this happening has been decreased by the restructuring of the parser.

• Existing notations – All known existing notations were tested on the new parser. These include Eddi, sq10, sq2eddi, logo and krusty.

• Multiple notations – Several notations were installed in one tkeden environment. Each notation was tested to check that there were no differences between running the notation on its own and running it alongside other notations.

• Large inputs – An updated Eddi notation on the new parser was tested with some large inputs (i.e. the thesis database). Further testing is needed on other notations.

6.7. Summary

The parser has evolved taking on the changes described above. A summary of the benefits gained from these improvements:

• More powerful pattern matching using perl compatible regular expressions. This allows more languages to be described and can be used to remove bugs in existing notations.

• A simplified model for script definitions. The “action” tag gives a method for writing agent actions that is easier to understand and places less demands on the modeller.

• Parser restructuring. The improvements bring the parser closer to an ideal model where an agent is a self-contained entity. Relieves the modeller from unpredictable name conflicts and allows notations to call other notations.

• A structure for error handling. Throughout the AOP errors are raised in a consistent manner giving the modeller more debugging information.
7. Modelling: Parser Visualisation

This section describes the evolution of a parser visualisation model. It is a good example of the iterative nature of modelling in the project.

7.1. Analysis & Design

The motivation for parser visualisation is based on two needs. Firstly, the research into the existing parser realised that it is difficult to debug a notation because of the lack of information about what the parser is doing. Secondly, as the parsing technique is different from traditional parsing, a visual learning aid would be useful to show users how the parser works.

The design must support these needs. A parser visualisation model must demonstrate the behaviour of a parser. In tune with EM principles, the visualisation should respond to changes to the input and changes to the grammar.

Parser visualisation should be useful for debugging notations and examining particular inputs.

7.2. Initial Model

The first stage of the model was to enable the parser to output information about the parse process. The initial design specifies a method for generating and extracting this data. When an agent finishes it outputs the details of the operation and actions that have taken place. This information is stored in two tables. The first table holds information about the agent:

- Input string
- Operation performed
- Token matched
- Script action performed
- Script variable name
- Etc.
The primary key for this table is the name of the script variable, as this is the only unique piece of information about an agent (note this implies that agents without scripts could be anonymous). The second table holds the dependency information between agents, it has just two columns:

- Parent agent
- Child agent

For example, if agent y is a child of agent x then the tuple (x,y) is a member of the dependency table. To identify the parent and child agents the unique script variable is used.

Some functions were written to enable this information to be stored. Due to the restructuring of the parser, only a minor change was required to the parser, which instructs each agent to call a procedure at the end of its execution. This procedure (aop_agent_end) can be redefined for different applications. For example, another model might be developed for the parse process and this procedure could be redefined to trigger that model.

The information from the parse process could be used in two ways. The debugger may wish to query the information directly using Eddi. There are some limitations to this as the relational model (as implemented by Eddi) does not support recursive operations (Date 2000). This means there is no simple way to write queries that return an entire hierarchical structure. A useful model is the EddiGUI which takes an Eddi statement and gives a graphical output of the table returned. This model is available on the CD, but it will not be discussed further as it does not directly relate to the project.

The second way of using this information would be to output it in a readable format. The first attempt at this traversed the dependency table using a depth-first algorithm and for each agent output a few lines of data. This data contains information about the input, the rule and the string that was matched. An output of a simple calculator expression is shown in Figure 7-1.
Each of the boxes in Figure 7-1 is an agent. The first agent uses the pivot operation to observe an addition symbol (+) in the input string. The indentation shows the level of the agent in the hierarchy, for example, the second agent is a child of the first. This model is useful in debugging as modellers can see exactly where parsing has halted.

7.3. Second Iteration

Although the initial model is useful in debugging, it does not demonstrate the behaviour of the parser effectively. The text output is not easy to understand if someone is learning the concepts of the parser. A more appropriate solution would be a graphical model.
Donald is a notation for definitive line drawing. The design of a graphical model includes the ability to draw a hierarchy of agents on the screen. The initial model can be used to store the parse information in tables. Instead of a procedure to output text to the command line, a new procedure can be created that displays a graphical output in a Donald window.

Throughout the project, a tree structure of observations has been used to demonstrate the behaviour of the parser (see Figure 4-1). This diagram is the basis for the design of a graphical model. The model should be able to construct a diagram of this nature for any input and any grammar.

The basic elements of the line drawing are agents. These are displayed as rectangles with a string inside the rectangle representing the input string. These rectangles are connected by a line which indicates a parent-child relationship. The first agent is the root agent and is positioned at the top. The level below contains its child agents, and so on. The leaf nodes are those agents that have no children.

Figure 7-2 shows the model in action on a simple calculator expression.

![Diagram of Donald](image)

**Figure 7-2: Visualisation model for an expression parse tree**

In implementing the graphical model, the most important procedure is to draw the rectangles in the correct position. Each agent can have zero or more siblings, and so it is important that the tree branches in the correct manner. With large trees it is possible
that some rectangles lower in the tree may overlap. This can be avoided by increasing the size of the Donald window.

A further refinement of the model was made that gave the rectangles a specific colour based on the type of agent operation that had been performed. The colour yellow indicates a pivot operation, blue indicates a prefix, pink indicates a suffix and grey indicates a literal.

7.4. Third Iteration

Analysis of the two models realised some limitations. The first model (text based) gave the user more information, but the second model (graphical based) displayed the behaviour in a clearer format. Other limitations were that each time the input was changed, first the parser had to be run and then secondly the procedure called to display the model.

Motivation to develop the model further was the need to examine particular agents more closely. The third iteration attempts to build a graphical model which can display more information with fewer demands on the user.

The design for the third model is an interface that allows the user to examine the parse tree. The interface should allow the user to enter an input string and an initial rule to apply. Using this information the interface should display a parse tree, as in the graphical model. To enable the user to obtain information about the parse process, it is necessary to see the data associated with each agent. The user should be able to select a specific agent and view all the details of that agent on the screen. The design for the described interface is shown in Figure 7-3.
Scout is the notation for screen layout. It allows only a few basic components, including textboxes, buttons and Donald viewports. For the interface, the majority of the screen will be taken up by a Donald viewport to display the parse tree. Textboxes will be used to input strings for the rule and input boxes. A button will be used to create a parse tree.
Figure 7-4: The parser visualisation implementation

The implementation of the design is shown in Figure 7-4. The scripts can be found on the CD, a table of descriptions can be found in Figure 7-5. The gui.s script describes the scout interface. During the modelling process, some observations were made about using scout:

- New scout screens (windows) can be created with specific sizes. This is useful because the original ‘screen’ can be left untouched for use by other models, meaning that the visualisation can run in parallel with other models.

- The position of components on the screen can be specified exactly using co-ordinates. The trouble with this is that if you change the size of one window, you are likely to have to move many others. However, there is a smart solution. Due to scout being a definitive notation, the position of components can be defined in terms of the size and position of other components. In the interface created, the position of each component is dependent on another component. Therefore if the modeller changes the position of one component, the
remaining components adjust without interference from the modeller. This is a particularly useful feature of dependency in screen layout.

- Some characters are unable to be displayed in scout and Donald because they are special characters in the underlying tcl interpreter. These characters are: \"$\{\}\]. With some experimentation, a fix was made to several Eden procedures that ‘escapes’ these characters (by prefixing a backslash) before passing them to tcl. This fix should be included in a future release of tkeden.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.e</td>
<td>Stores agent information in a table</td>
<td>Basis for all models</td>
</tr>
<tr>
<td>graph.e</td>
<td>Draws an agent tree using Donald</td>
<td>Used in graphical models</td>
</tr>
<tr>
<td>gui.s</td>
<td>Builds a scout interface</td>
<td>Interface only</td>
</tr>
<tr>
<td>action.e</td>
<td>Defines the interactions with the interface</td>
<td>Interface only</td>
</tr>
<tr>
<td>colours.e</td>
<td>Defines the colours to be used for the tree</td>
<td>Used in graphical models</td>
</tr>
<tr>
<td>Run.e</td>
<td>Load the model</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7-5: Parser visualisation scripts**

To use the model, a user would enter a rule and an input string, and then to display the parse tree the user would click the ‘Parse’ button. If the parse process is not successful then the tree may not be displayed completely. The user can click on an agent in the tree to select it, and detailed information on the agent will be displayed in the lower part of the interface.

### 7.5. Summary

The final parser visualisation model provides a tool for modellers to understand the parse process. The model demonstrates the concepts behind the parser and will be useful in teaching modellers about the behaviour of a notation. The interactive nature of the model will benefit the debugging of notations, as modellers can find out the specific results of each agent.
The development of this model follows a more typical EM development model compared to the parser improvements. The final model is an evolution of several iterations of development.
8. Modelling: Notation Builder

This section suggests some ideas for developing notations. A model is created to enable users to experiment with and develop new notations.

8.1. Motivation

The modeller can write notations more efficiently now due to the documentation and parser improvements. Writing notations still requires knowledge of the syntax and structure of the rule definitions. This will always lead to user errors when developing.

8.2. Reading Notations

8.2.1. Analysis

Generally, rule definitions are not easy to read. Although Eden lists are ideal data structures for storing rules, this does not make them easy to read. A common format for rule definitions is adopted throughout this project. This format can be outlined as follows:

- Each rule begins on a new line.
- Each square open bracket begins on a new line (unless it is immediately followed by a square close bracket). The line is indented proportionally to the depth of the current sub-list.
- Between brackets and quotation marks there is a single space.

An example of a rule in the correct format:

```plaintext
calc_term =
  [ "literal_re", "[0-9]+",  
    "action",  
    [ "now", "$v = $t;" ] ],  
  [ "fail", "calc_err" ];
```
When developing notations in tkeden it is not convenient to always write rules in this fashion. It can also be difficult to keep track of which rules belong to a notation. However when a notation is complete, it is likely that the modeller will want to store all the rules in a script. These should be ordered and formatted in such way that they can be understood. As with all models, someone may wish to examine it and modify it, in which case it should be easily understandable for others.

There is a need for a method of displaying all the rules belonging to a notation in a standard, clear format.

8.2.2. Design & Implementation

The model has two parts, a method to obtain a list of all rules belonging to a notation, and a method to output a list of rule definitions in a clear format. Therefore the design consists of an algorithm for traversing a notation and a function for printing rules.

The traversal algorithm must identify every possible rule that can be used in a notation. The user will specify a beginning rule, with which the algorithm will proceed as follows:

1. Create a set of traversed agents (initially empty).
2. Create a set of untraversed agents containing just the beginning rule.
3. Remove a rule (referred to as the current rule) from the untraversed set.
4. Add the current rule to the set of traversed agents.
5. If the current rule specifies child rules, then add these rules to the untraversed set.
6. If the current rule specifies a fail rule, then add that rule to the untraversed set.
7. If the untraversed set is not empty then repeat from step 3.
8. Return the set of traversed agents.

This algorithm will return a list of rules that can be used by any given notation. The implementation of this algorithm can be found on the CD in the ‘listnot’ model.
The next stage is to print out the rule names and definitions. The design adopts the format from the analysis above. This requires that sub-lists are identified and indented to match their depth. A recursive algorithm is used to print the lists. The design as a piece of pseudo code:

```
Func printList (data, indent):
    If isList (data) Then
        Output newline
        For i = 1 to indent
            Output space character
        Next i
        Output open square bracket
        For i = 1 to length(data)
            printList( data[i], indent+1 )
        Next i
        Output comma
        Next i
        Output close square bracket
    Else
        Output data
    EndIf
EndFunc
```

This algorithm is implemented in Eden. The only remaining algorithm is a simple loop that, for each rule, outputs the name and then calls this function. The data argument is the rule definition, and the indent argument is initially set to one.

The implementation provides one function for the user called `print_notation`. It takes one argument which is the name of the first rule in the notation. It outputs the notation as described above.

A further script is designed that outputs the notation to an Eddi table. The same traversal algorithm is used. A new output function is created, based on the existing one, to create Eddi statements that insert rows into a table. A tuple in the table contains a rule name, an operation, a pattern and a fail rule. More information could be included in this table, or other relations.
8.3. Writing Notations

To write a notation is to define a set of Eden lists that satisfy a particular syntax. This syntax for rule definitions is specified in previous sections. There is no reason why this should be the only way of writing notations.

One suggestion for further work is to develop a new syntax for defining notations. This could even be implemented using the agent-oriented parser. It would probably have the confusing name ‘a definitive notation for notations’. Refer to ‘Further Work’ for more information.

Another option is a graphical interface for writing notations. This would be a model that enabled the user to examine a notation and make changes. The model would be dependent on the rule definitions and the two would have to maintain integrity. Any changes to the model would directly effect the notation, and therefore the result would be instantly observable.

8.3.1. Design & Implementation

The model from the previous section that traverses a notation can be used in this model too. Instead of generating text output, the rule definitions will be output to components in a graphical user interface.

The aim of the model is to build notations, hence the name ‘Notation Builder’. A mini design specification can be defined as follows:

- Given the beginning rule for a notation, the interface should display a list of all rules in the notation.
- The exact definition of each rule must be observable.
- A rule definition must be modifiable. The interface must enable the user to change the entire rule or just a small portion of it.
- The interface must allow new rules to be added.

Other desirable properties the model may have include the ability to analyse a notation for errors. A useful feature would be an indication of missing rules.
After some initial experimentation it was noticed that to display a notation on the screen requires a lot of space. It is difficult to display more than a few complete rules on the screen at any one time. One of the requirements of the model is to be able to view all the rules in a notation. A clear and concise method is needed to display the rule list in a graphical interface.

The chosen method for displaying this information is a grid component. The development of this is explained later in this section.

The grid is used to display only a partial amount of information, because a complete rule definition cannot be displayed along a single row. The columns of the grid are rule name, operation and pattern (see Figure 8-4). The grid is sensitive, and allows a row to be selected. The complete rule definition of the currently selected row is displayed next to the grid. The design so far allows the user to examine the definition of any rule in the notation.

The next stage of the design allows the user to make modifications to the notation. When a rule is selected in the grid, the rule definition is displayed in a series of textboxes. The values of these textboxes can be modified. A save button enables the user to commit these changes to the notation. A clear button enables the user to cancel the changes. To create new rules and copy existing rules the rule name must be modified and then saved. It is important that the data in the grid is consistent with the notation. Therefore when modifications are made the notation must be rescanned, and the grid updated. If a new rule has been added then the grid must resize accordingly.
A screenshot of the implementation is in Figure 8-1, and the scripts can be found on the CD. A description of the scripts is in Figure 8-2.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>traverse.e</td>
<td>Traversal algorithm to find all rules in a notation (also used for text output)</td>
</tr>
<tr>
<td>extract.e</td>
<td>Extracts the rule definitions and stores in a table</td>
</tr>
<tr>
<td>grid.e</td>
<td>A general grid component</td>
</tr>
<tr>
<td>gui.e</td>
<td>The scout definitions to display gui</td>
</tr>
<tr>
<td>deps.e</td>
<td>Maintains the dependencies between the notation and the gui</td>
</tr>
<tr>
<td>Run.e</td>
<td>Load the model</td>
</tr>
</tbody>
</table>

Figure 8-2: Notation builder scripts
8.3.2. Grid Component

As discussed before, Scout provides only a few basic GUI components. Most visual programming languages have many complex components like grids, trees and scroll bars.

Many models in the past have developed custom tables for displaying data. The most prominent of these is Spreadsheet model (Roe 2002). This model has some important qualities that could make it useful in the notation builder. It allows different sized spreadsheets by specifying the numbers of rows and columns. The spreadsheet cells resize automatically dependent on the contents of the cells. The values in the cells are dependent on the value of Eden variables. These are useful qualities that are useful for a grid component for displaying rule definitions.

The design of a grid component for the notation builder requires further features. A mini design specification includes these points:

- The grid can be any number of rows and columns.
- The number of rows and columns can be changed.
- The grid may contain a header row.
- The grid cells must resize to fit the contents.
- The value of the cells can be static or dependent on other variables.
- A single row may be selected and highlighted in the grid.
- Multiple grids may be displayed in the same scout window.

This specification could be the basis for any grid component. A general component with these features could be used in other models in the EM group. It could be seen as a replacement for the many table-like structures that modellers have manually created in scout. With this idea in mind, it was important at the design stage to ensure the grid component could be generally applied to other models. As the model developed, specific grid properties like colours and borders were defined as variables that a modeller could tailor to their needs. The usage can be seen in Figure 8-3.
To create a new grid:
  grid_define( "mygrid", cols, rows, xposition, yposition );
where:
  mygrid = the name of the new grid
  cols = number of columns
  rows = number of rows
  xposition = distance along x-axis from top-left of screen
  yposition = distance along y-axis from top-left of screen

To add the grid to the screen:
  %scout
  screen = mygrid_display;

To change the contents of cell in col 2, row 4:
  mygrid_data[4][2] = "new cell contents";

To change the heading of column 3:
  mygrid_header[3] = "new heading";

Other editable properties (mygrid_property):
  _font, _bgcolor, _fgcolor, _bdcolor, _relief,
  _headbgcolor, _head_fgcolor, _selectedbgcolor, _selected_fgcolor,
  _sensitive, _cellheight, _border

Figure 8-3: Grid component usage

To prove the versatility of this component, it was used in the EddiGUI model to show
the resulting table of any Eddi statement. It has also been used by another student and
it is hoped that the grid component will be useful to others in their models. This could
also be the beginning of more complex components in Scout.

<table>
<thead>
<tr>
<th>rule name</th>
<th>operation</th>
<th>pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>calc expr</td>
<td>pivot</td>
<td>+</td>
</tr>
<tr>
<td>calc expr2</td>
<td>pivot</td>
<td>-</td>
</tr>
<tr>
<td>calc expr3</td>
<td>pivot</td>
<td>*</td>
</tr>
<tr>
<td>calc expr4</td>
<td>pivot</td>
<td>/</td>
</tr>
<tr>
<td>calc expr5</td>
<td>prefix</td>
<td>(</td>
</tr>
<tr>
<td>calc expr6</td>
<td>suffix</td>
<td>)</td>
</tr>
<tr>
<td>calc err</td>
<td>read all</td>
<td>[]</td>
</tr>
<tr>
<td>calc term</td>
<td>literal re</td>
<td>[0-9]+</td>
</tr>
</tbody>
</table>

Figure 8-4: The grid component

8.3.3. The SAND notation

In order to evaluate the notation builder it was necessary to use it to write a notation
from scratch. A previous final year project developed a model for systolic arrays
(Sockett 1992). The project provides a definitive notation for the description and
specification of a systolic array. This notation is called Sand.
The project provides an environment for simulating systolic arrays that uses a Scout/Donald interface. This has not been used for some time, as the Eden environment has changed considerably in the 11 years that have passed since the projects completion.

The original Sand translator was implemented in C++ and no working version exists. It is an ideal candidate for an agent-oriented parser notation. It is hoped that the model can be resurrected by building a notation that behaves the same as the original translator. The source code is available in the project report, along with the BNF description of the notation (the BNF is shown in Appendix C).

The first stage of building the model was to write Eden equivalents of the C++ procedures that generate the translated code. This proved to be the more difficult than writing the notation. With these procedures implemented, the Eden code to model a systolic array can be generated from the environment.

The second stage was to write a notation using the AOP. The Sand notation as described by the project was implemented using the notation builder model. The time taken to develop this notation was less than had been expected. The builder supports an iterative-style modelling cycle. A notation can start with only a few rules, which parse only a very basic input. Each of the iterations adds a significant feature, until the notation is complete. This is very useful because a notation can be tested as it is developed. The ability to model iteratively also means that the task can be broken down into smaller (less daunting) chunks.

The rules define the agent actions that interact with the environment. In the Sand notation, the rule actions call the new Eden procedures. These procedures execute some code in the environment, most of which is Donald code to draw the systolic array model.

The basic rules in the Sand notation can be found in Appendix C (in the format advocated earlier in this section).

The final stage was to make some minor adjustments to the graphical user interface and test the model. There already existed some Sand scripts for defining systolic
arrays, so these were used as a benchmark. A couple of other scripts were created to make sure that every feature of the notation had been tested.

The existing C++ translator was able to detect syntactic and semantic errors. The notation that has been written does detect syntactic errors, but it does not check the semantics of an input. An example of a semantic error is declaring an input variable and then using it as an output variable. It is important to note that this is not a limitation of the agent-oriented parser. With more time the modeller could introduce some simple error detection for the semantics of an input.

There are further extensions that could be made to the Sand model. A useful feature would be the ability to reset the model, so that a new systolic array could be created. The Sand model could also be the basis of some similar work on Artificial Neural Networks.

The end result is the resurrection of the Sand model. Considering its age, the model demonstrates several desirable qualities that other models could use. The interface enables the user to scale and size the Donald viewport with ease. The model is at its most impressive when the matrix multiplication systolic array is loaded, which can be found in sand/examples on the CD. Refer to Appendix C for screenshots of the Sand model.

8.3.4. Evaluation

In developing the sand notation, a critical evaluation was carried out on the notation builder. Some points are worth considering:

- The sand notation is possibly the largest notation that has been developed for the agent-oriented parser. It is difficult to manage 55 rules at once. The main problem this causes for the builder is that the grid goes off the screen because there are too many rows to fit in. This could be fixed with some scroll bars to move up and down the list.
- There is some error detection when the user modifies a rule, but it does not give the user any detailed information about the error. It would also be useful to know which rules are not well-formed (see ‘A possible theory’ in an earlier
section). These features would eradicate the majority of bugs before a notation is executed.

- The order of rules is not always convenient. The list of rules is ordered by the sequence in which they were traversed. An improvement to this would be to make the column headings clickable, and the action to be reordering the data based on that column.

8.4. Summary

A brief summary of this section on building notations:

- Methods for writing notations were analysed to allow notations to be understood by a wider audience.
- The design of a model to build notations realised the need for more powerful scout components. A generic grid component was developed.
- The grid component is used in the notation builder. This model provides a graphical user interface for developing notations, allowing the modeller to manipulate rule definitions freely.
- To evaluate the notation builder, the sand notation was developed. This has successfully resurrected this model for systolic arrays.
9. Conclusion

9.1. Achievements

The achievements of this project can be summarised as follows:

- The concepts behind the agent-oriented parser have been examined. The nature of the parse process is understood and the precise behaviour is explained.
- A short comparative study has identified the differences between traditional parser theory and the agent-oriented parser.
- Documentation has been written for the agent-oriented parser. The tutorial enables modellers to understand the parser and build notations.
- An example notation (calc) is developed to show the features of the agent-oriented parser.
- A new agent-oriented parser is designed and implemented which includes:
  - Powerful pattern matching operations.
  - Simplified script definitions.
  - Flexible parser structure and standardisation of functions.
  - Better error handling.
  - Removal of various bugs and undesirable features.
- A parser visualisation model is developed to demonstrate the parser behaviour and assist the debugging process.
- A notation builder is developed to enable the modeller to view and modify notations using a clear and simple graphical interface.
- The systolic array model is resurrected from 1992. The notation builder is used to create a Sand notation that uses the agent-oriented parser.

9.2. Critique

In concluding the project, a critique is offered to indicate weaknesses and to highlight areas that need more work.
Firstly, it is important to be critical of the underlying principles of the agent-oriented parser. From the results of this research, it is clear that this parsing style is not going to compete against traditional parsing. This parsing style does not have a strong logical foundation like Automata and Turing machines, and it is debateable whether it ever will. From a computational point of view, this parser has little or no advantages. It has been shown that the advantages arise when a human being is brought into the picture. This parser benefits the user in that it affords the user some real-world concepts. The bridging of the gap between the user and the computer is why this project is applicable to empirical modelling.

It has been shown that the implementation is a considerable improvement on the existing parser. However, there are still some areas that could be improved. The first of these is performance. Although the performance of all the notations used in this project is satisfactory, it could still be improved. The parse process requires that the agents pass considerable amounts of data around which is not very efficient. The design of the parser focuses more on conceptuality than performance. Another area that would improve efficiency of the parser is the creation and disposal of variables. There are often a large number of variables generated for script variables, and when the parsing of an input is complete then these are no longer needed. The implementation would benefit from better management of these variables.

The parser visualisation model is helpful in understanding the behaviour of the parser as well as particular notations. The main criticism to be made here is the lack of dependency. The model only takes a snapshot of the parse tree. If the input or grammar changes then the user must perform a parse to observe the changes. A better (more EM style) model would maintain the parse tree as the input and grammar changes. The reason this is not implemented is it would be inefficient to reparse for every change.

The notation builder is a useful tool for writing notations. Although this model could be improved (as discussed in the evaluation), it serves the modeller adequately as proved with the development of Sand. The only criticisms are lack of features.
9.3. Final Thoughts

The initial aim of this project was to investigate a strange parser that was not well understood. This small idea has given rise to a hefty project and has motivated even further research in this area.

The specification outlined three basic sections to the project; understanding the parser, improving the parser, and building tools for the parser. The project has successfully addressed these three areas. The project has not only built a tool for tkeden, but it has given some useful insights into a unique style of parsing. The research has strengthened the relationship between empirical modelling and the agent-oriented parser.

9.4. Further Work

This project has opened up more areas of research than it has covered. Below are some projects ideas relating to the agent-oriented parser.

9.4.1. Attribute grammars and the AOP

One of the main issues in parsing is the semantics of a language. Common causes for errors are variables being referenced without being declared. Attribute grammars are concerned with such issues. It appears that the concepts behind attribute grammars could be modelled in an EM environment (i.e., the dependency of attributes in a semantic tree). Previous research into AGs has yielded complex algorithms for building semantic trees (Reps 1983), but given EM techniques these could be redundant. A possible project would be to see how EM could be used to model AGs. This would be in comparison to the AOP, which works like a simplified AG model.

9.4.2. Modelling Artificial Neural Networks

Given the work by Patrick Sockett (1992) on modelling systolic arrays (see ‘Sand: the resurrection’), it is clear that EM environments are well-suited to these types of model. Neural Networks is an exciting area of research, based on arranging simple computational components in a distributed fashion. A project would be to examine the sand notation for defining systolic arrays, and compare it to neural networks (which
are more complicated). Implement some models of neural networks in Eden and/or implement a notation (using the AOP) for defining neural networks.

**9.4.3. Developing the scout notation**

Most GUIs have a vast range of components available to them: textboxes, grids, tree components, etc. The scout notation only has basic components. Following the work of Ward (2002), it is now possible to extend notations. Using the AOP, a project would be to write some extensions to scout, to be able to create grids (see the grid component), tree structures and other useful GUI objects. Also see Roe's Attribute Explorer for ideas of further components (scroll bars, etc).

**9.4.4. Distributed agents**

It was hinted at in researching the agent-oriented parser that the parse process could be distributed. The reasons for doing this would be if the input is very large and the grammar contains a large number of rules (an example might be natural language processing). Using the dtkeden tool, a distributed parser could be implemented that allowed many distributed agents to process the input. Such a model is likely to require a pool of parsing tasks that each agent can add to and perform.

**9.4.5. A Definitive Notation for Notations**

A notation could be developed for the agent oriented parser, called a Definitive Notation for Notations (Deno?). This notation would enable notations to be described. The motivation for this is that writing rules using a specific notation would be easier than defining Eden lists. This could be developed along with improving the Notation Builder.

**9.5. Acknowledgements**

Special thanks to Dr Meurig Beynon for supervising this project. The encouragement, support and ideas provided have been much appreciated.
Also, praise must be given to the Empirical Modelling group at Warwick whose work is inspiring a unique area of research. Particular thanks go to Christopher Roe and Ashley Ward for developing my understanding of this subject.
10. References

Beynon WM, 2002, SQLedd environment documentation, DCS.


Beynon WM, Yung E, 1987, Implementing a Definitive Notation for Interactive Graphics, DCS.

Brown C, 2000, An agent-based parsing system in Eden, 3rd year project, DCS.


EM Group, 1999, Idiot’s guide to Empirical Modelling, DCS.


Packwood RA, 2001, CS223 Introduction to Software Engineering, DCS.


Roe C, 2002, A Generalised Spreadsheet, EMod Data Repository, DCS.


Sokcett P, 1992, Simulation of Systolic Arrays using Definitive Principles, 3rd year project, DCS.


Ward A, 2002, Tkeden documentation and change log, DCS.

Appendix A: Parser Documentation

This is an abbreviated form of the documentation.

Parser concepts

In order to learn to use the Agent-oriented Parser in Eden we must put aside our prior knowledge of parsing. Conventional parsers read each input character, one at a time, and not until the entire string is read can any meaning be derived. Instead, it is useful to think about how we, as humans, read languages (natural or otherwise). When we read sentences, we do not remember each character, or even each word, but our brains register the important words. From this we are able to derive meaning.

An agent is an independent entity capable of acting on and interacting with an environment. In the Agent-oriented Parser, the agents have a set of rules (a grammar) with which to parse any input. Each agent will take an input string and a rule, with which it will determine whether the rule can be applied. If the rule is applied, then more agents could be generated to work on substrings. The agent will fail if it cannot apply the rule.

A rule specifies a string that must be observed in the input string. For example, an agent might have the input ‘1+2’ and the most important string it must find could be ‘+’. When the agent observes a match, it will pass the remaining substrings (if any) to new agents. These are called child agents.

Writing notations

A notation is defined as a set of rules, which specify the behaviour of our agents. A rule is simply an Eden list. The basic template for a rule:

```plaintext
myrule = [ operation, pattern, [ rule, ... ], [ "fail", rule ] ];
```

An operation is a string containing the name of the operation an agent should perform. These will be explained in detail later. A pattern defines what string the agent is trying to find (or observe) in the input. A rule is a string containing the name of a rule (a variable name of an Eden list).
The first item in the list is always an operation. The second item is always the string to be matched. The third item is optional, it is a list of rules to apply to the resulting substrings – the number of substrings depends on the operation to be performed. This is the minimum that a rule can contain. It is likely that most rule definitions will have a fail clause, such that if the agent fails then it will try another rule. The fail clause is a two item list, the first item is always the string "fail" and the second item is a string containing the name of a rule.

**Developing a calculator notation**

A simple calculator is able to accept digits and arithmetic operators. It will calculate the result of the input expression. Calculators also allow nested expressions using brackets.

**Start with a simple model**

Recall that our Agent-oriented Parser uses an agent to observe a string in the input. The simplest agents therefore match the input entirely. On a calculator, the user can input just a digit, and the result will be that same digit.

The first operation we will be introduced to is "literal". This operation attempts to match the entire input string (see diagram right). It does not create any child agents. Lets define a rule that matches the digit ‘1’:

```plaintext
number1 = [ "literal", "1" ];
```

To invoke the parser, you must supply an input string and a starting rule. The only input our rule should match is ‘1’, so to test this run the parser:

```plaintext
dfparse("1", "number1", []);
```

It should accept. The first parameter is the input and the second is the starting rule. Try other inputs to get the parser to reject.

Our language is not very powerful, being only able to accept the digit ‘1’. Now we will introduce the fail clause. We can get our parser to try another rule should the operation fail:

```plaintext
number1 = [ "literal", "1", [ "fail", "number2" ] ];
```
number2 = [ "literal", "2" ];

The language will now accept either of the digits ‘1’ or ‘2’. We could extend this method to accept all the digits.

**Iteratively developing the model**

Now that our calculator can parse digits, we will introduce operators to the input. This is an important concept in Empirical Modelling, the ability to build up a model in an iterative fashion.

Our current model allows us to parse positive numbers, so an obvious extension is to allow negative numbers too. We can define an integer as a number with an optional minus symbol (−) in front of it.

The next agent operation we shall introduce is "prefix". This operation matches a string at the beginning of the input. If a match is made then a child agent is created with its input as the unmatched part of the agents input (see diagram right). This operation can be used to detect a minus symbol at the beginning of our number:

    term = [ "prefix", "-", "number1", [ "fail", "number1" ] ];

Notice that we use the same rule for the child (third item) and the fail clause. The parser will try to match a minus sign at the start of the input, if it matches then it will match the remainder of the input as a number, else it will match the entire input as a number.

A similar operation is "suffix" which does exactly the same as "prefix", but at the end of a string. An example of using suffix is to remove the semi-colon from the end of a string (i.e. to parse a statement in C/Java/Eden)

The parser should now accept any positive and negative numbers. We shall now look at how we can parse arithmetic operations (e.g. +, −, *, /). As always we will begin with something simple and build the model up. We will first try to parse expressions containing only additions of terms, where our terms are the integers we learnt to parse above.
This is where the most powerful agent operation comes in. The "pivot" operation searches the input (left to right) for a specified string. If it finds a match, then it creates two child agents, one for the left substring and one for the right substring (as shown in the diagram on the right). Our parser would pivot on the addition sign:

```plaintext
expr = [ "pivot", "+", "expr", "expr", [ "fail", "term" ] ];
```

An "expr" agent is looking for an addition sign, and if it finds one it creates two children that also search for expressions. If the agent finds no addition sign on the input, then the fail clause specifies that the input must be a term.

In order for our parser to recognise expression containing other operators it is necessary for us to have a rule for each string to be matched:

```plaintext
expr = [ "pivot", "+", "expr", "expr", [ "fail", "expr2" ] ];
expr2 = [ "pivot", "+", "expr", "expr", [ "fail", "expr3" ] ];
expr3 = [ "pivot", "+", "expr", "expr", [ "fail", "expr4" ] ];
expr4 = [ "pivot", "+", "expr", "expr", [ "fail", "term" ] ];
```

Notice that we search for operators in their reverse precedence order. This can be explain by taking an example, say the input is ‘1+2*3’. First we would pivot on the addition sign giving us two substrings ‘1’ and ‘2*3’. We have broken the calculation down into two sub-calculations which we will add together later. The deepest level will get calculated first, which in this example is ‘2*3’. Therefore we are observing the rules of precedence correctly.

**Regular expressions**

Using just the pivot and literal operations gives you all the power you need to develop languages. However, the rules would be quite cumbersome without regular expressions. The Agent-oriented Parser has 3 other operations that deal with basic regular expressions.

The first is "read_all", which is an r.e. version of the "literal" operation. This will attempt to match every character in the input string with a set of characters specified in the rule. For example, the following will match any number using a single rule (compare with our inefficient earlier method):
number = [ "read_all", [["0","9"]]] ;

The second item in the list is a list of tuples. The tuples define the range of characters included in the set to be matched (inclusive). Note that the "read_all" operation accepts the empty string.

The other two regular expression operations are "read_prefix" and "read_suffix", which operate in much the same way but you also specify the number of characters to attempt to match. For example, the following will match one letter of the alphabet at the beginning of the input string:

letter = [ "read_prefix", [["a","z"],["A","Z"]], "nextrule" ];

Perl-style regular expressions

The above regular expressions lack power, for example, you cannot specify rules for real numbers or identifiers. Our definition of a "number" will accept the empty string – not desirable in most cases!

Three more regular expression operations exist in the latest version of the Agent-oriented Parser. These are "literal_re", "prefix_re" and "suffix_re". The first matches the whole input, whereas the other 2 match the beginning and the end of the input respectively. The pattern to be matched is a perl-style regular expression. For example, the following correctly parses a number:

number = [ "literal_re", "[0-9]+" ];

More information on perl regular expressions can be found in any good perl book or on the web.

Blocks

Now we shall look at adding brackets to our notation. This will give our calculator the ability to work out expressions like (1+2)*3. We have seen how to use the "prefix" and "suffix" operations, so we can use these to parse brackets:

expr5 = [ "prefix", "(" , "expr6", [ "fail", "term" ] ];
expr6 = [ "suffix", ")" , "expr" ];
This gives us a bit of a problem. Think about the input ‘(1+2)*3’. The first agent will use the pivot operation on the ‘+’, leaving two substrings ‘(1’ and ‘2)*3’. Instead, we really want the parser to pivot on the ‘*’ and break the input into the two smaller expressions ‘(1+2)’ and ‘3’. The parser needs to be sensitive with our brackets.

This is where blocks are a very useful feature of the Agent-oriented Parser. We can define a block and instruct agents to ignore that block. To define a block:

```plaintext
bras = [ "\( \), "bras" ];
addblocks("bras");
```

The first statement is the block definition. The first item of the list is a pair of strings, the first being the starting string of the block and the second the end string. The second item is a list containing names of blocks that may be contained within the block. The second statement adds the block definition to the environment.

Now for a particular rule we can specify it to ignore blocks. For our calculator notation, we want to ignore any strings between brackets when the agent is looking for an arithmetic operator (+,-,*,/). We add an ignore clause to our rule:

```plaintext
expr = [ "pivot", "+", "expr", "expr" ,
        [ "ignore", ["bras"] ],
        [ "fail", "expr2" ] ];
```

This behaviour is demonstrated in the diagram (right). The string within the brackets is ‘greyed-out’ because it is ignored. Hence the most important string to be observed is the multiplication sign (∗) and a pivot is made.

It is important to remember that when an agent ‘ignores’ a block it is not removing that block from the input. It is perhaps better described as preserving the block. The agent simply preserves the contents of the block and leaves it for another agent to parse.

**Scripting**

A parser that either accepts or rejects an input is of little use unless it produces some output. Our simple calculator needs some way of outputting the result of an expression. This is achieved with agent actions. If an agent does not fail to match the
input then it can optionally perform some actions. Each action can be performed before
or after the actions of the agent’s children.

The format for including agent actions in a rule definition is similar to the other
optional components of a rule. Here we modify our "term" rule by adding an action
that prints out some random comment:

```
  term =
    [ "literal_re", "[0-9]+",
      [ "action",
        [ "now", "writeln("somerandomcomment")" ] ] ];
```

The third item in the list above is the action declaration. This sublist begins with the
"action" tag to recognise it from the other optional tags. The items following the head
tag are the commands to execute. Each command is a list containing only 2 items, the
first being either "now" or "later" depending on whether the command will be executed
before or after the child agents. The second part of the command is the command
string which is typically some eden code to be executed.

An agent has some data associated with it that can used in its actions. Each agent also
has a unique variable associated with it. This agent data can be substituted into the
command string using the following:

- $i = the input string to the agent
- $j = the name of a variable containing the input string
- $t = the token/string that was matched by the agent
- $v = the variable name that belongs to the agent
- $s1 = the first substring of the input
- $s2 = the second substring of the input (and so on for 3rd, etc)
- $p1 = the variable name of the first child agent (parameter 1)
- $p2 = the variable name of the second child agent

Now we can make our "term" rule more useful by adding an action that stores the term
value in the agent variable:

```
  term =
    [ "literal_re", "[0-9]+",
      [ "action",
        [ "now", "$v = $t;" ] ] ];
```

We would then add actions to our other rules. The "expr" rule can do the addition of
the two sub-expressions:

```
  expr =
    [ "pivot", "+", [ "expr", " expr" ],
      [ "action",       
```

Final Report :: Agent-oriented Parsing :: Antony Harfield :: 0001036 :: Page 75
Take a look at the final calculator notation at the end of the document for more examples of agent actions.

Note: The dollar sign is a special character in the command string. If you want to print a single dollar sign ($) in your command string then you must follow it by another dollar sign ("$$" will produce a single dollar in the command string).

The original version of the Agent-oriented Parser had a different method for writing scripts, using the "script" tag. Although the parser will still accept these scripts, it is recommended you use the "action" notation. For more information on the "script" tag, refer to Chris Brown’s third year project.

**Installing notations**

Now that we are happy with our calculator notation, it is probably a good idea to make it more accessible. We can install new notations into the Eden environment, which can then be used in scripts as you would existing notations (e.g. %eden, %donald, %scout). In tkeden this will add a radio button for our new notation to the environment.

First we must create an initialisation rule:

```
calc_init = [ "\n", "calc", [] ];
```

The first item in the list is the string to split the input on. For our calculator notation we would like to separate each command by the end-of-line character (\n). For other notations you may wish to split the input on other characters (e.g. semi-colon for C/Java/Eden). The second item is the starting rule. The third item is a list of blocks to ignore in the splitting procedure.

To install the notation in the environment:

```installAOP("%mynotation", "calc_init");```

Notations must begin with a percent (%) character.

You can now switch to the new notation by typing %mynotation. Do not forget to switch back to %eden for Eden code!
Appendix B: Notations

Calc (old-style scripts)

/* A simple calculator. Accepts expressions of the form:
  *  1+1
  *  2*3–4
  *  (5+6)/7
*/

mancalcs = ["\n", "calc", [1];

calc = ["read_prefix", [[], 0], "expr",
  ["script",
    ["declare", "a"],
    ["setparas", ['a']],
    ["later", "writeln(\"=\", %\%);\" a\""]],
    ["fail", "calc_err"]];

func op_expr { para op, next;
  return
  ["pivot", op,
   ["expr", "expr"],
   ["ignore", ["brs"]],
   ["script",
    ["declare", "a", "b"],
    ["setparas", ["a"], ["b"]],
    ["execute", %\% is %\% // op // %\%;\" %\% v_paras[1]", "a", "b"]],
    ["fail", next]];
}

expr = op_expr("+", "expr2");
expr2 = op_expr("–", "expr3");
expr3 = op_expr("\*/", "expr4");
expr4 = op_expr("\*/", "expr5");

expr5 = ["prefix", ",", "expr6",
  ["script",
   ["setparas", ["v_paras[1]\"]],
   ["fail", "term"]];

expr6 = ["suffix", ")", "expr",
  ["script",
   ["setparas", ["v_paras[1]\"]],
   ["fail", "calc_err"]];

term = ["read_all", ["0", "9"],
  ["script",
   ["later", %\% = %\%;\" v_paras[1]", "v_string"]],
   ["fail", "calc_err"]];

calc_err = ["read_all", [],
  ["script",
   ["execute", "writeln(\"calc: syntax error\")"]];
installAOP("%calc", "manycalls");

Calc (new)

calc_init =
  [ "\n", "calc", [] ];

calc =
  [ "prefix", "\", "calc_expr",
    [ "action",
      [ "latex", "\texttt{\text{calc}}\text{\text{\_\_\_}}$v$ = $s_{1} + s_{2}$;" ] ],
    [ "fail", "calc\_err" ] ];

calc_expr =
  [ "pivot", ",\", [ "calc\_expr", "calc\_expr" ],
    [ "ignore", [ "bras" ] ],
    [ "action",
      [ "now", "$v$ = $s_{1} + s_{2}$;" ] ],
    [ "fail", "calc\_expr2" ] ];

calc_expr2 =
  [ "pivot", ",\", [ "calc\_expr", "calc\_expr" ],
    [ "ignore", [ "bras" ] ],
    [ "action",
      [ "now", "$v$ = $s_{1} - s_{2}$;" ] ],
    [ "fail", "calc\_expr3" ] ];

calc_expr3 =
  [ "pivot", ",\", [ "calc\_expr", "calc\_expr" ],
    [ "ignore", [ "bras" ] ],
    [ "action",
      [ "now", "$v$ = $s_{1} \times s_{2}$;" ] ],
    [ "fail", "calc\_expr4" ] ];

calc_expr4 =
  [ "pivot", ",\", [ "calc\_expr", "calc\_expr" ],
    [ "ignore", [ "bras" ] ],
    [ "action",
      [ "now", "$v$ = $s_{1} / s_{2}$;" ] ],
    [ "fail", "calc\_expr5" ] ];

calc_expr5 =
  [ "prefix", ",\", "calc\_expr6",
    [ "action",
      [ "now", "$v$ = $s_{1}$;" ] ],
    [ "fail", "calc\_term" ] ];

calc_expr6 =
  [ "suffix", ",\", "calc\_expr",
    [ "action",
      [ "now", "$v$ = $s_{1}$;" ] ],
    [ "fail", "calc\_err" ] ];

calc_term =
  [ "literal\_re", [0-9]+,
    [ "action",
      [ "now", "$v$ = $t$;" ] ],
    [ "fail", "calc\_err" ] ];

calc\_err =
  [ "read\_all!", [],
    [ "action",
      [ "now", "\texttt{\text{calc}}: syntax error\";" ] ] ],
  [ "fail", "calc\_init" ] ];

installAOP("%calc", "calc\_init");
Palindrome

palindrome =
    [ "prefix", "", "pal_1" ,
      [ "action", 
        [ "later", "if($p1)\n writeln("\":-\")\n;\n else\n writeln("\":-(\")\n);" ] ] ]

pal_1 =
    [ "literal_re", "[a-zA-Z]?\", 
      [ "fail", "pal_2" ],
      [ "action", 
        [ "now", "$v = 1;" ] ] ]

pal_2 =
    [ "prefix_re", "[a-zA-Z]\", "pal_3" ,
      [ "fail", "pal_err" ],
      [ "action", 
        [ "now", "pal_str = "$t"; $v is $p1;" ] ] ]

pal_3 is
    [ "suffix", pal_str, "pal_1" ,
      [ "fail", "pal_err" ],
      [ "action", 
        [ "now", "$v is $p1;" ] ] ]

pal_err =
    [ "read_all", [], 
      [ "action", 
        [ "now", "$v = 0;" ] ] ]

palindrome_init = [ "\n", "palindrome", [] ];
installACP("$palindrome", "palindrome_init");
Appendix C: The Sand Model

The Sand BNF

Taken from Sockeyt (1992).

```
statements  == statement | statements statement
statement    == timer | outputis | outputon | inputequals | functionstate
                | outputdec | inputdec | elementdec | inelement | pos | inputfrom
timer        == 'time' '(( ' + ' + ' )') ' ;'
                | 'time' '(( ' - ' - ' )') ' ;'
function  == ';' | 'numlabel' | 'varlabel' | 'unlabel'
              | 'trace' | 'untrace' | 'display'
functionstate == function var_or_complisy ' ;'

elementdec == 'element' variablelist ' ;'
inelement  == 'inelement' variable '{' elementdlist '}

elementdlist == elstatement elementdlist | elstatement

elstatement  == timer | outputis | inputfrom | outputon
                | inputdec | pos | functionstate

pos         == variable ' = ' '{' integer ' , ' integer ' }' ' ;'
outputis    == variable ' is ' expression ' ;'
outputon    == variable ' on ' var_or_comp ' ;'
inputequals == variable ' = ' numlist ' ;'
              | variable integer ' = ' numlist ' ;'
inputfrom   == variable ' from ' var_or_comp ' ;'
outputon    == 'output' variablelist ' ;'
inputdec    == 'input' variablelist ' ;'

var_or_complisy == var_or_comp symCOMMA var_or_complisy
                   | var_or_comp
var_or_comp    == variable
              | compvar
variablelist  == variable ' , ' variablelist | variable
numlist       == integer ' , ' numlist | integer

expression     == expression operator expression
                | '(' expression ')' | variable | integer
operator       == '+' | '-' | ' ' | '/'

compvar  == variable ' _ ' variable
            | ' _ ' variable ' _ ' variable
variable    == letter | variable letter
integer     == digit | integer digit
letter      == 'a' | 'b' | ... | 'z' | 'A' | .. | 'Z'
digit       == 0 | 1 | ... | 9
```

Notation

```ruby
## define types of blocks
curlyes = [ ['"{"', "}"], [ ] ];
addblocks("curlyes");
## initialisation rule
```
sand_init =
[ ";", "sand", [ "curlies" ] ];

## agent rules
sand =
[ "suffix", ";", 
  [ "sand_statement_1" ] ];
sand_statement_1 =
[ "prefix", "element ",
  [ "sand_var_list" ],
  [ "fail", "sand_statement_2" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_def_element($pl[sand_i],0,0);" ] ] ];
sand_statement_2 =
[ "prefix", "input ",
  [ "sand_var_list" ],
  [ "fail", "sand_statement_3" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_def_input($pl[sand_i]);" ] ] ];
sand_statement_3 =
[ "prefix", "output ",
  [ "sand_var_list" ],
  [ "fail", "sand_statement_4" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++) { 
    sand_def_output($pl[sand_i]); sand_def_line($pl[sand_i]); }" ] ] ];
sand_statement_4 =
[ "prefix", "inelement ",
  [ "sand_inelement_construct_1" ],
  [ "fail", "sand_statement_5" ] ];
sand_statement_5 =
[ "prefix", "time", 
  [ "sand_time_function_1" ],
  [ "fail", "sand_statement_6" ] ];
sand_statement_6 =
[ "prefix", "trace", 
  [ "sand_full_var_list" ],
  [ "fail", "sand_statement_7" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_trace($pl[sand_i]);" ] ] ];
sand_statement_7 =
[ "prefix", "untrace", 
  [ "sand_full_var_list" ],
  [ "fail", "sand_statement_8" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_untrace($pl[sand_i]);" ] ] ];
sand_statement_8 =
[ "prefix", "numlabel", 
  [ "sand_full_var_list" ],
  [ "fail", "sand_statement_9" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_label_num($pl[sand_i]);" ] ] ];
sand_statement_9 =
[ "prefix", "varlabel", 
  [ "sand_full_var_list" ],
  [ "fail", "sand_statement_10" ],
  [ "action", 
    [ "later", "for (sand_i=1;sand_i<$pl#;sand_i++)\n    sand_label_var($pl[sand_i]);" ] ] ];
sand_statement_10 =
[ "prefix", "unlabel", 
  [ "sand_full_var_list" ],
  [ "fail", "sand_statement_11" ],
installAP("%sand", "sand_init");

## further rules omitted (see notation.eden)

**Screenshots**

![Screenshot 1](image1.png)

![Screenshot 2](image2.png)

*Figure C-1: Simple Sand example*
Figure C-2: Matrix multiplication using a systolic array
Figure C-3: How Sand generates a parse tree
Appendix D: Supporting material

Project Log

28/10/02 Finished off the website. Added specification, log and download sections.
28/10/02 I developed a calculator notation over the summer. It is a good example to
begin with as I built the parser up iteratively by starting with addition, then
adding other operations and finally adding brackets. This is available in the
download section.
28/10/02 Over the last week I have been experimenting with agents and their
dependencies. An agent can be thought of as a dependency on its two inputs:
a string and a parse rule. If either of these change, then it is likely that the
agent will have to recalculate its output. On the other hand, an agent is
dependent on its children in that if one child fails then the agent itself fails. This
idea of dependency of agents fits in well with the concepts of EM.
29/10/02 First draft of the AOP documentation has just been loaded up. Not finished the
section on scripting yet!
04/11/02 Identified bugs in the built-in agent templates: 'number' accepts the empty
string, 'ident' accepts an input that contains whitespace after the first character.
04/11/02 Added 3 new operations to the AOP to handle perl-style regular expressions.
06/11/02 Updated the documentation and added HTML version to site.
06/11/02 Presentation given on AOP concepts to EM project students.
10/11/02 Currently re-writing the agent functions of the AOP. This is to give a better
understanding of how the agents behaviour, especially regarding scripting
actions. Next step will be to develop the 'simple scripting' notation!
10/11/02 Added the regular expression extensions to the download page.
13/11/02 I have a clear idea of an interface for creating notations, which borrows ideas
from Chris Roe's spreadsheet and JP's symbol table eddi database. Agents
could be specified in eddi tables, with columns for op, pattern, fail, ignore,
script, etc. This table could be displayed in a number of scout windows, like a
spreadsheet, which could be manipulated to alter the notation.
13/11/02 Completed re-working of existing parser. All operations are now consistent (i.e.
all return [token, childlist]). Separated operations from the agent so that the
agent function does not need to be redefined to add new operations.
Separated script execution from the agent such that agents now only use local
variables. An agent also has the ability to run a different type of script
("action").
18/11/02 New agent-oriented parser released - go on, download it and see if it works! Includes the new agent "action" model, plus rewritten most of the existing code. Should now be able to bolt on extra things to the AOP without having to radically change anything.

22/11/02 Currently experimenting with scout to see what I could achieve with the AOP GUI. Using Chris Roe's spreadsheet model I created a Query GUI for EDDI. This would be a good project area in its own right! I am thinking of developing a universal grid object for scout, which I can see as being quite useful.

27/11/02 Experimenting with using the new AOP as a translator. My first translator takes a list of eden list definitions, and outputs the definitions in a nicely indented easily read list. Would be interesting to create a translator for XML or something similar.

28/11/02 Updated the AOP documentation - includes scripting and regular expressions. Corrected some errors. Now available in PDF!

12/01/03 Over the holiday I re-wrote the eddi notation using the new parser. A couple of bugs appeared in the parser that have now been sorted in the AOP v2.02, plus added a few enhancements.

12/01/03 Finished the first version of the scout grid component. Tried to make it generic so can be used in many models. I have many ideas for extending this, but the current implementation should be enough for my needs.

23/01/03 The agent-oriented parser is based on the principles of a theorem prover. A parser is trying to prove that an input can be derived from a set of rules. Looking at Natural and Structural Operational Semantics, it appears that all these topics are related. A theorem prover takes a set of axioms together with a set of rules and tests if a given theorem is valid.

07/02/03 Made an extension to the AOP that stores parse information in a table. This can be used to view the parse process, using graphical tools.

23/02/03 Made some major improvements to the grid component I have been developing, including: resize a grid, cells can be sensitive, a row can be selected, and added more customisable properties. This is used in the notation builder.

02/03/03 Many updates this week, including AOP, grid component and list notation. Plus finished visualisation and builder.

08/03/03 Idea: A definitive notation for defining notations (DN2). Would go something like: agent name { operation: ; pattern: p; fail: f; etc.. }; block bras {];

09/03/03 Presentation preparation complete. Slides available on the website shortly.
Project Files

CD includes: models, notations, documentation, reports and website.