1 Introduction

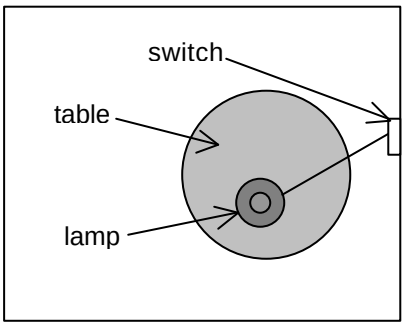
1.1 Introduction to Empirical Modelling

In order to introduce this project it is necessary to have some knowledge about the area of research known as Empirical Modelling (EM). As the purpose and structure of the project are closely related to some fundamental ideas within EM, this first section is an introduction to EM leaving the following section to introduce the project.

Observables, dependency and agency

EM suggests a way of looking at a world and creating a representation of it, usually in the form of a computer-based model. Experimenting is the key to the modelling process, as the name "empirical" suggests. When the modeller encounters the world, he interacts with it, performing experiments and recording three types of information about it: *observables*, *dependencies* and *agency*. These are best illustrated using an example.

Consider a room containing a table upon which there is a lamp controlled by a switch. The modeller enters the room and interacts with it, moving the table, operating the switch. These experiments - discovering the things that can change - have given the modeller knowledge about *observables* of the room - the position of the table, the state of the light switch.



Next, maybe the modeller moves the table and notices that the position of the lamp moves too, or changes the state of the switch and notices that the light turns on and off. These are instances of *dependency* - relationships between observables.

The last type of knowledge the modeller can acquire is about *agency*. An agent in a world is something autonomous that can cause changes to occur. In our room, the most obvious example of an agent is the modeller himself, but another example is gravity - if the modeller pushes the lamp sideways off the table, gravity pulls it downwards towards the floor. Agents are independent of each other and can act on a system concurrently.

Definitive notations

Observables and dependencies are recorded using a *definitive notation*. Variables are declared to represent observables and are assigned persistent values but observables are defined using definitions. A definition is like a spreadsheet formula in that it is not assigned a constant value but it is defined in terms of other variables (or definitions). Whenever one of its dependencies changes, the current value of the observable is recalculated. A definition takes the form:

```
x is f (v1, v2, v3)
```

Where x is the defined variable and f is the function used to calculate it given its dependencies: v1, v2 and v3. Here are some variables and definitions from the room example given above:

```
table_x_position = 100;
table_y_position = 50;
lamp_x_position is table_x_position + 10;
lamp_y_position is table_y_position + 10;
```

A set of variables, definitions and functions describing a model is referred to as a script. In order to use the information recorded in a script, it must be evaluated using a program called an interpreter. The interpreter permits the user to adjust and monitor variables and it maintains the dependencies between them.

EDEN

The interpreter most widely used at Warwick is called EDEN (Evaluator of DEfinitive Notations). It has a built in notation, also referred to as EDEN, and this includes a procedural language for writing support code and implementing the functions used in definitions. EDEN allows the modeller to "include" scripts stored in files and also to enter new pieces of script, line by line, while the model is being interpreted. The dynamic nature of the interpreter means that the model has no well-defined boundaries but grows as more pieces of script are added.

There are two versions of EDEN, a textual version called ttyeden and a graphical version called tkeden. Tkeden incorporates built-in translators for the notations DoNaLD and Scout, which allow users to interact visually with a model through a graphical front end. A modeller can write a script in DoNaLD or Scout and the respective translator will convert each line into EDEN as it is entered.

The use of an interpreter saves the modeller from having to write code that updates each of the definitions and allows dependencies to be recorded quickly using only a few lines of script. Even when a dependency is not fully understood it can be declared in terms of a function to be implemented later.

Model development

Because a script can be built up quickly and easily, the distinct software development stages of analysis, design and implementation do not exist. Instead, the cycle of *experiment*, *observe*, *record / implement* is iterated. There is not usually a point when the model is finished - it is repeatedly refined as the designer experiences more of the world.

An important fact to note is that the model is constructed *while* the modeller is experiencing the world. The properties of a system that are most significant in the mind of the modeller, i.e. those which the modeller sees as most significant to the nature of the system, are noticed first, and so they are recorded first. In this way the recorded model is built up with a similar structure to that of the designer's mental model.

1.2 Introduction to the project

The problem

Like DoNaLD and Scout, different notations are suited to writing different kinds of model. For example, the language ARCA was designed to describe Cayley diagrams and the language eddi is used to manipulate relations in a database. To write a model in a notation other that DoNaLD, Scout or EDEN an external translator must be used. This usually involves writing a script to a file, translating the script into EDEN and then including it in tkeden or ttyeden. If you want to grow a model by adding script line at a time then using a translator adds a significant delay. The process becomes much more cumbersome because instead of just typing a line of script into EDEN you must save that line to a file, translate the file and then include it.

Ideally, tkeden would have a built-in translator that a user could configure to translate any custom notation into EDEN. The purpose of this project, in part, is to develop such a translator.

The format of the project

The approach taken by EM modellers to software engineering is quite different to that associated with the standard software development process and it can give new insight into

problems other than model building that are considered to be well-understood in terms of classical software engineering. It seemed appropriate to develop the translator using such an EM approach and at the same time to investigate the problem of parsing from an EM perspective.

The project can be broken down into three parts. During the first part, I experimented by writing a parser for the definitive notation eddi. The parser was built up slowly using an iterative process reflecting that of EM model development. At each stage I observed a syntactic feature that seemed significant to the language, wrote a piece of code that parsed that feature and incorporated it into the parser. In the second part of the project I analysed this experimental process by which the parser took shape and formed some general ideas about parsing from an EM perspective. For the third part, I implemented these ideas in creating a configurable translator.

Since language translation and parsing are considered well-understood subjects, a more conventional approach to writing the translator would have probably yielded quite an efficient, complete product. In this case, however, a lot of time was spent on developing the ideas behind the translator rather than on implementing them. Additionally, the incremental approach favoured by EM trades an efficient solution for one that has the advantages of being developed and implemented simultaneously. For these two reasons I consider the concepts outlined in the report to be a result of the project that is as important as the software. This could be re-implemented more efficiently using conventional software engineering methods now that the ideas behind it have been developed.

2. A parser for eddi

This section describes the experimental process by which I wrote a parser for the definitive notation eddi. The parallel between this process and the EM model building process is the key to approaching the problem of parsing from an EM perspective.

The experimentation stage of model building here corresponds to examining an example eddi string and mentally "parsing" it in order to understand it. The most important syntactic feature of the string should be easily recognised and this corresponds to the observation stage of model building. Finally, a parsing step is written to simulate this recognition and this corresponds to implementing / recording the observation. This cycle was repeated until the parser correctly recognised the eddi language.

The code for each iteration can be found on the attached disk.

2.1 The initial parser

To serve as a starting point, my project supervisor gave me the EDEN code for a simple parser that he had written. The feature that it recognises is the way that brackets give structure to an expression, which is important when recognising a relational algebraic expression as would be found in an eddi statement that creates a definition:

```
A is (B + C) . ((B - C) % D)
```

The parser takes a string resembling a bracketed regular expression and produces a set of strings containing no brackets. It does this by first finding the most deeply nested pair of brackets and creating a new string storing their contents. It then inserts the name of the new string in place of the bracketed expression in the original string. In this way, the parser builds up a set of pseudo-definitions. They are not true definitions because a variable is not linked to the variables on its right hand side using the EDEN mechanism. The right hand sides are instead strings that could contain the names of other variables so, with knowledge of the references within the strings, a function could be written to evaluate them as EDEN would evaluate a definition.

As an example, consider the string:

```
S = “A – ((B + C) – D)”
```

First, the parser recognises the bracketed expression (B + C) and creates a new string:

```
S1 = “B + C”
```

It then inserts the name of the new string into S, removing one set of brackets from the expression:

```
S = “A – (S1 – D)”
```

Eventually, it builds up a set of simple “definitions”, none of which contains brackets:

```
S  = “A – S2”
S1 = “B + C”
S2 = “S1 – D”
```

2.2 The first iteration

The next important part of the string to be parsed seemed to me to be the operators. If I used EDEN functions to implement each relational operator then instead of strings, the parser could generate actual EDEN definitions that used these functions. This would later allow the expression to be evaluated in EDEN. I copied the functions from an older eddi parser that is partly written in EDEN. I created a list of tuples, each of which associated an operator with the function implementing it. A fragment of the list is given below:

```
[[+, union], [–, difference], [%, select]]
```

Consider again the example string:

```
S = “A – ((B + C) – D)”
```

Now, the parser recognises the bracketed expression (B + C) and creates a definition:

```
S1 is union (B, C)
```

As before, it inserts the name of the new definition into S:

```
S = “A – (S1 – D)”
```

Eventually, S contains no more brackets and is itself converted to a definition:

```
S  is difference (A, S2)
S1 is union (B, C)
S2 is difference (S1, D)
```

2.3 The second iteration

At this stage, the parser only accepted expressions where each bracket pair contains just one operator and overcoming this limitation seemed to me to be the next important parsing step to implement. Parsing more than one operator at the same bracket level can also be expressed as implementing operator precedence. When I read a bracket-free expression containing

multiple operators, I visualise brackets around the operator with highest precedence and its operands. True to EM principles, this is how I implemented the next parsing step.

Consider this example string:
S = “((A + B) . C + D) – E”

If the deepest bracket pair contains just one operator:
S = “((A + B) . C + D) – E”

Then the parser removes the string, as before, and creates a definition:
S = “(S1 . C + D) – E”
S1 is union (A, B)

But if the deepest bracket pair contains more than one operator:
S = “(S1 . C + D) – E”

Then the parser chooses the operator of highest precedence and inserts brackets around it and its operands. The list of relational operators and their functions is now ordered by precedence so that this operator of highest precedence can easily be found.
S = “((S1 . C) + D) – E”

Now the most deeply nested bracket pair is guaranteed to contain just one operator and it will be extracted on the next iteration of the parser.

2.4 The third iteration

The parser can now almost evaluate relational algebraic expressions but several relational operators require further parsing. The "select" operator has on its right hand side a string representing a predicate and this string must be processed to extract the parameters required by the function implementing the operator. The "project" operator has on its right hand side a string representing a set of columns, each of which must be supplied as a separate parameter to the implementing function. I wrote both of the intermediate wrapper functions that perform these parameter conversions using simple string operations, searching for sub-strings, concatenating, etc.

The resulting definition can now be evaluated giving the value of S, given that A, B, C, D are variables of the type required by the old eddi parser functions.

2.5 The fourth iteration

The parser now parses relational algebraic expressions but these only appear in eddi as the second half of a relation definition statement. The next step seemed to be to parse an entire definition statement and to implement the remaining eddi statements. The table below describes the form of each eddi statement with keywords shown in bold.

Define table	`<table name> **is** <relational algebraic expression>;`
Display table	`**?**<table name>**;**`
Create table	`<table name>**(**<column name> <data type> [<key>], …**);**`
Insert rows	`<table name> **<< [**<data>, <data>, …**], [**<data>, <data>, …**], …;**`
Delete Rows	`<table name> **!! [**<data>, <data>, …**], [**<data>, <data>, …**], …;**`

When I read one of these strings the first feature that I notice is the eddi keyword that indicates which type of statement it is. The keyword separates the string into the sub-string to the left of the keyword and the sub-string to the right of the keyword. For example, the define table statement is split into a table name and a relational algebraic expression by the "**is**" keyword and the create table is split into a table name and a list of column descriptions by the "**(**" keyword. The only exception is the display table statement, which has no sub-string to the left of the "**?**" keyword.

To perform this separation I wrote an EDEN function called "splitaround" that, given a string and a keyword, returns a list containing the prefix of the string ending before the first occurrence of the keyword and the suffix starting after it:

```
splitaround ("A is B + C", "is") = ["A", "B + C"]
splitaround ("A + B + C", "+") = ["A", "B + C"]
```

The older eddi parser contains EDEN functions to display, create and edit tables so I needed only to convert the string data on the right of each keyword into the parameters required by their respective functions. I converted them using simple string operations in the same way as the parameters of the "select" and "project" operations were converted as described in the previous section.

2.6 The fifth iteration

During the final stage of development, I noticed the similarity between string operations the parser was performing: splitting a string around a keyword, splitting a string around an operator and splitting a list of parameters around the spaces between elements. I re-implemented all of these, the fundamental string operations performed by the parser, using the splitaround function described in the previous section.

3 An EM approach to parsing

In this section, I start by comparing the parser that I constructed with an accepted, conventional method of parsing such as that used in the original eddi parser. I then try to summarise the differences between them to form an alternative approach to parsing. To make the following section easier to read, note that I have used the word "conventional" to mean a yacc- / lex-style LR (1) or LL (1) parser and the word "token" to mean a character or character string.

3.1 Significant tokens and order of observation

I would first like to make three observations about the grammar accepted by a conventional parser. Firstly, it describes a language without saying exactly how to parse it. Secondly, it attaches no special significance to any particular token. Finally, it always parses in the same way, by reading tokens from left to right and matching them in that order.

Arguably, most meaningful strings have a salient feature that is the first to be recognised and which is the most significant when understanding the string. A good example is the “:=” token in a PASCAL assignment statement: it is conclusive evidence of the string's meaning and it is the first feature to be recognised by a reader. The eddi parser takes into account these salient features because at each step, it observes the token that I felt to be the most significant. For example: when the eddi parser reads a relational algebraic expression, brackets are recognised first and operators next.

The absence of significant tokens in a conventional parser leads to the arbitrary choice to observe tokens in a left-to-right order. Conversely, the order of significance that was implied when I read an eddi statement has led to a corresponding order of observation in the eddi parser. Whereas a conventional grammar does not specify how to parse a language, the grammar implemented in the eddi parser has clear steps defining when to read each token.

An LL parser interprets a string by removing the leftmost token of the string and attempting to match it to a pattern. This leaves the remainder of the string to be parsed using the same procedure. The leftmost token is not always the most significant, however. This could be located at the end or in the middle of the string, or even distributed throughout it like the commas in a comma-separated list of parameters. Because it observes the most significant feature first, the eddi parser often reads a token at the end or in the middle of a string. The latter procedure results in two sub-strings, one on either side of the token, not necessarily to be parsed using the same procedure. In a divide-and-conquer manner, each parse step breaks down the problem of parsing a string into one or more sub-problems of parsing smaller strings.

Because of the approach taken when writing the eddi parser, the way that an EM modeller discovers a significant observable in the world and builds it into his model is reflected in this idea of parsing tokens in order of significance. We have seen the EM principle of observation applied to parsing; next we look at how these observations are recorded.

3.2 Recording observations

The function of a language is to convey meaning with symbols, so when a string is considered as written in a particular language, it means something. When the eddi parser breaks down a string into sub-strings, each of these also has a meaning. If we are to interpret a string then, in keeping with the divide-and-conquer method described above, we must understand how the meanings of the sub-strings are combined to give the meaning of the whole.

For example: the meaning of a relational algebraic expression is the table that it equates to. When the eddi parser reads a string of the form "A + B", it observes the "+" operator and produces the two sub-strings "A" and "B". The operator that was read signifies that the tables that "A" and "B" equate to should be combined using the union function to give the value of "A + B". An EDEN variable representing the whole string is defined in terms of variables representing the sub-strings and the function used to combine them. As the modeller records an observation using definitions, so the eddi parser takes an observation of how a string is composed of concatenated sub-strings and records it as an EDEN definition.

In conclusion, our EM parser should be like a modeller, observing and recording the features of a string. The parser must be a directed observer, however, instructed by a suitable grammar to make particular observations in a particular order. In other words, in contrast to conventional grammars, we want a grammar that describes how to parse a language rather than just describing its syntax, and that treats tokens with unequal importance.

4. An agent-based parsing system

This section describes the main characteristics of a parsing system that would reflect the EM principles discussed in the previous section.

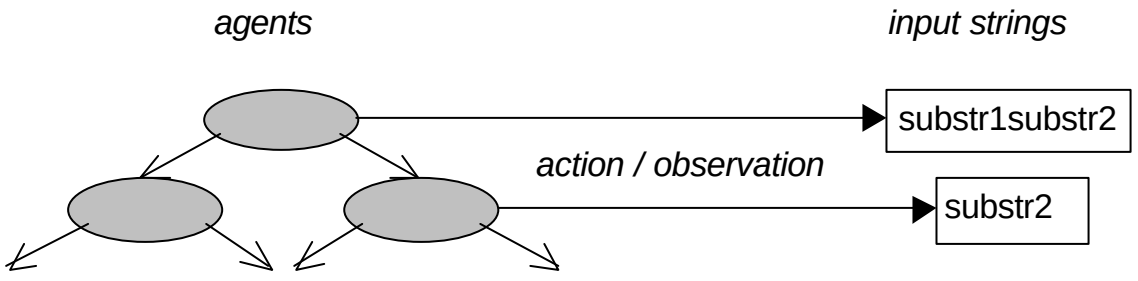
4.1 Agents as observers

We have seen that the operation of an EM parser would be divided into individual observations. Each observation identifies one or more parts of a string and in doing this it recognises that the string is composed of sub-strings. The sub-strings that were observed are no longer a part of the parsing problem and those not observed are parsing sub-problems to be observed further.

Each of these observations can be thought of as being performed by an EM *agent*. In this context, an agent has associated with it a string, called its *input string*, and an *observation*, which consists of a string to observe, called its *token*, and a method by which to observe it, called its *operation*. The basic operation of an agent is to look at its input string and try to observe its token. If the token is observed then the input string is broken up into one or more sub-strings that, together with the token, compose the input string. These sub strings are called the *output strings* of the agent. The agent creates a new agent for each of its output strings. These agents are called its *child agents* and their input strings are the respective output strings of the parent agent. The process that an agent performs is called its *action* and the term *parsing* here refers to the overall process consisting of the actions of all agents.

4.2 Enforcing order of observation

When a string is parsed, an agent is created with the string to be parsed as its input string. The agent acts, creating child agents, which create more child agents, building up a tree-like structure of actions with the outputs of one connected to the inputs of the next. The diagram below illustrates this tree, which represents a model of synchronisation that shows how the parsing process is controlled.



The order in which observations are made is determined by the rule that an action can only occur after its parent action has occurred. This rule is enforced by the fact that an agent creates its children after it has acted so they can never act before it. Note that an order is only defined between an agent and its descendants and ancestors. Agents that are not vertically linked can act in any order with respect to each other, even concurrently.

4.3 Methods of observation

Having described the observers in the parsing system, we now look at the observations themselves and how they collectively form the parsing process. The function of a parser is to test whether a string fits a description. The writer of a conventional grammar extracts from this description the features of the language that fits it. The agent-based parsing system, however, is not merely concerned with an overall description of the string, and associates a description with each agent.

When a string is described, there is usually one feature of the string that is more important to the description than any other is. For example, in a PASCAL assignment statement the ":=" token signifies assignment and describing a string as a PASCAL assignment statement implies that it contains this string more than it implies any other feature. The description also dictates how to describe the sub-strings that result if this token is observed. For example, the description of a PASCAL assignment statement consists of a PASCAL identifier, followed by the ":=" token, followed by a PASCAL expression.

In terms of agents, in addition to the properties listed earlier, an agent in the system has an implicit description of how its input string should look, and this description is present in the form of the agent's token and its child agents with their own descriptions. When the agent with description "PASCAL assignment statement" acts on the string "A := 3 + 1" it breaks the string into the sub-string "A", the input string to a child agent with description "identifier"; and the sub-string "3 + 1", the input to a child agent with description "expression".

Thinking about the eddi parser, I identified five main methods of observing strings. The *prefix* operation looks for the token at the beginning of the input string, checking if it is a prefix. The *suffix* operation looks for the token at the end of the input string, checking if it is a suffix. Both the prefix and suffix operations create one child agent to observe the remaining characters. The *pivot* operation searches for the token in the middle of the input string, breaking it into two output strings, one on either side of the pivoting token. A child agent is created for each of the two output strings. The *split* operation searches the input string for all occurrences of the token, using it as a separator to break the string up into one or more output strings. A child agent is created for each one. Lastly, the *literal* operation compares the whole string to a

particular string. When it is performed, no sub-strings are left unobserved so no output strings or child agents are created.

4.4 Recording observations

Although agents fit the description of “directed observers” found in the previous section, they do not yet record their observations. The EM approach taken in this project suggests using a definitive notation to record these observations. This has the benefit of building up information about the parser input slowly, over the actions of many agents. In addition to its input string, each agent is now given a set of input variables, called its parameters. Each agent also creates a number of output variables that are passed, as parameters, to its child agents with its output strings. Depending on its action, an agent can now also create definitions that relate its parameters and its output variables. As the previous section described, these definitions should reflect the way that the meaning of its input string is understood by combining the meaning of its output strings using the meaning of its token.

5 An agent template notation

The next two sections describe the configurable translator of definitive notations, the writing of which was the aim of this project. The translator implements in EDEN the actions of the agents that make up the parsing system that was described in the previous section. In this section, I explain a notation for representing the behaviour of an agent, and in the next, I summarise the code that creates and runs such an agent.

In order to configure the translator so that it can read lines of script written in a new notation and create EDEN definitions from them, a way to describe notations is required. We have already mentioned that an EM grammar would specify exactly how to translate a notation and, because the translation will be performed by agents, such a grammar would have to specify the actions of each agent. From this perspective, what we need is a way of describing agent templates so that when we have a string written in a particular notation, we can create an instance of the correct agent that will act on the string, creating child agents to act on it and eventually translating it into EDEN.

An agent template is a list, the only complex data type supported by EDEN. The list has a name similar to the agent description, such as “assignment” or “expression”. The first element of the list is a string holding the operation that the agent performs. The second element is a string containing the token of the agent. The third element is a sub-list, each element being a string containing the name of an agent template from which to create a child agent. Below is an example agent template for the PASCAL assignment statement:

```
assignment = ["pivot", ":=", ["identifier", "expression"]];
```

The list may have other optional elements containing tagged properties of the agent. Each is a tuple (two-element EDEN list), the first element containing a description of the second element, i.e. [tag, value].

5.1 The agent script language

The first of the optional agent properties that we discuss is the agent’s script. This script is written in a very simple language and its purpose is to accompany the parsing process with the execution of EDEN code, including but not limited to creating definitions. Its tag is “script” and its value is a list of instructions, each being a list where the first element is the instruction name and the remaining elements are arguments. There are five instructions: `declare`, `execute`, `later`, `setparas` and `allparas`; and they are explained in turn below.

The `declare` instruction is used to create new script variables. Its tag is “declare” and its value is a list of strings, each holding the name of a new variable to declare. EDEN variables

with a unique name are created for each variable. This is to ensure that different variable instances are used in multiple instances of the same agent and in multiple agents containing a variable with the same name. Variables are permanent their EDEN names can be passed between agents and used to build up definitions. Here is an example use that declares three variables named “a”, “b” and “my_var”:

```
[“declare”, “a”, “b”, “my_var”]
```

The `setparas` instruction is used to indicate the script variables that should be passed as parameters to the child agents. There is an argument for each child, each being a list of the names of the script variables to pass to that child. There is no restriction on the number of variables to pass and a variable can be passed to more than one child. The parameters to this agent are stored in the list script variable named “`v_paras`”. A parameter from the parent agent can be passed through to a child agent without having to create an intermediate script variable by specifying an element of the `v_paras` list in the `setparas` instruction. Here is an example for an agent with two children that passes no variables to the first child and two to the second child, one of which was the third parameter to this agent:

```
[“setparas”, [], [“a”, “v_paras[3]”]]
```

The `allparas` instruction is similar; it takes as its only argument the name of a variable to declare, which it makes into a list. For each child agent, the instruction automatically declares a script variable and makes it a parameter of that agent. It then collects the names of these variables into the list. This instruction is useful when the agent performs a split operation and the number of children is unknown. Given that an agent has two children, this example creates a list named “children” containing, for each child, a variable that is in that child’s parameter list:

```
[“allparas”, “children”]
```

The `execute` instruction takes as its first argument is a string of EDEN code. The string may contain one of two format specifiers where script variables are to be substituted. The remaining arguments are the names of script variables, one for each format specifier. For any occurrences of the format specifier “%%”, the script parser substitutes the EDEN variable name of the corresponding script variable. For any occurrences of “$$”, the script parser substitutes the value of the corresponding script variable. After substitutions are made, the string is executed in EDEN. The example below defines the agents parameter as a function of two other variables:

```
[“execute”, “%% is f (%%, %%);”, “v_paras[1]”, “a”, “b”]
```

The `later` instruction saves the code to be executed after the agent and its children have acted. This is necessary when, for example, a variable whose name is extracted by a child agent is to be defined:

```
[“later”, “$$ is f (%%, %%);”, “var_name”, “p1”, “p2”]
```

This last example, taken from the eddi parser described later, shows a whole script that defines one of its parameters in terms of those of its children using a relevant function:

```
expression = ["pivot", "+", ["expression", "expression"], [“script”,
      [“declare”, “p1”, “p2”], [“setparas”, [“p1”], [“p2”]],
      [“execute”, “%% is union (%%, %%);”, “v_paras[1]”, “p1”,
      “p2”]]];
```

Assume that the script is passed, as its sole parameter, the name of a variable to define, as in the eddi parser. After the child agents have acted they will have modified their variables and information will have been passed "up the tree" back to this agent. This is possible because a parameter is always a reference to a variable, its name, not its value. Information can also travel "down the tree" by setting the value of a variable in a script before it is passed, allowing the children to read from it. Three useful script variables are automatically defined for an agent: "v_substrs" is a list of its output strings, "v_string" is its input string and "v_paras" is a list of its parameters.

5.2 Optional observation

A problem with the agent described so far is that its observation cannot be optional. Along the same lines, you cannot specify that any one of a list of observations should be made. The tagged property described below, with name “fail”, solves these problems by specifying the template name of an agent to create if the observation fails.

We start by looking at how an observation can fail. A conventional grammar may allow patterns of the form "`expression = term '+' term`" where, in order to see if a string matches the pattern "expression", the parser must first check to see if it starts with the pattern "term". Calls to the parsing function will be nested, in this case, and the parser will have to backtrack if a mismatch occurs in the nested call. An agent, however, always starts by making its observation and since child agents only act if their parent acted successfully, recursive calls of this sort cannot occur. If an observation fails then an alternative agent can be created straight away, so its template name can be specified in a tag.

The example template below shows how optional matches can be implemented. It matches a statement with an optional semicolon at the end:

```
["suffix", ";", ["statement"], ["fail", "statement"]]
```

This example shows how to specify that one of a list of patterns should be matched. The last instruction omits the optional failure clause; if this instruction fails then the whole parsing process has failed:

```
atom   = ["pivot", "+", ["identifier", "identifier"],
      ["fail", "atom_1"]];
atom_1 = ["pivot", "-", ["identifier", "identifier"],
      ["fail", "atom_2"]];
atom_2 = ["pivot", "*", ["identifier", "identifier"],
      ["fail", "atom_3"]];
atom_3 = ["pivot", "/", ["identifier", "identifier"]]
```

5.3 Blocks and nested syntax

Another limitation of the agent-based system is that it cannot easily express the syntax of a bracketed regular expression or any other notation that has a nested structure. The solution to this problem came from another observation on how I read and understand programming languages. While I was developing the agent template notation, I wrote a description of the procedural language PL/0, included on the attached disk. I noticed that when reading example PL/0 programs I first ignored the contents of any functions and just noticed the general outline of the program. I then looked closer and just noticed the outline of each function, how it was divided into variable declarations and code, and only after having done this did I examine each section in detail.

I started by looking at the least deeply nested code and worked inwards. Usually the way that the code was indented made this easy, but if not, I only read the symbols that marked the beginning and the end of a nested block. For example, at the outermost level, I ignored everything between the word "procedure" and the next semicolon. The only problem here is telling which semicolon belongs to the "procedure" keyword and which belong to the statements within the procedure. When it was unclear, I had to read the symbols that marked the beginning and the end of the blocks within a procedure block, and the blocks within those blocks.

This led me to define the concept of a "block" that the string operations could understand. A block has a starting symbol and an ending symbol and it specifies the other types of block that it may contain. Blocks are not things to be observed explicitly but when looking for a token the string operation (such as pivot) can be instructed to ignore everything within a type of block. To implement the idea, I wrote a string-searching function that has the option of ignoring blocks, which I could use to build up functions that are more complex.

As an example use of the function, a PL/0 program can have global variables at the top level of the program and local variables within procedures, both of which are defined using the "var" keyword. To find the global variables, the function could search for the first occurrence of the string "var" in a program, ignoring any occurrences within a "proc_block" where:

```
proc_block = ["procedure", ";", ["var_block", "const_block", etc]];
```

The first element of the list is the starting symbol, the second is the ending symbol and the third is a list of the other types of block it contains. This block idea also solves the problem of parsing bracketed arithmetic expressions by defining a "brackets" block that contains other "brackets" blocks. To tell an agent to ignore particular blocks, include in the agent template a tag called "ignore" whose value is a list of block names. To add a new block, an EDEN list variable is declared as above and the function addblocks is called. The example below illustrates both of these:

```
brackets = ["(", ")", ["brackets"]];
addblocks ("brackets", "block name 1", "block name 2", etc.);
expression_1 = ["pivot", "+", ["expression", "expression"],
      ["ignore", ["brackets"]], ["fail", "expression_2]];
expression_2 = ["pivot", "-", ["expression", "expression"],
      ["ignore", ["brackets"]], ["fail", "expression_3]];
```

5.4 Additional string operations

The only string operations discussed so far match literal strings. However, parsers are not always looking for literal strings; for example, a number could be one of many combinations of the characters 0 to 9. A parsing system for the language of numbers could probably be written in the template notation by looking for each digit in turn using the "fail" tag described above but this would clearly be cumbersome. Instead, I added three new string operations, extensions of the prefix, suffix and literal operations, to allow regular expression-like string matches. Each of these, called `read_prefix`, `read_suffix` and `read_all`, specify a set of characters that they can match.

The `read_all` operation takes as its token a list, each element of which is also a list. If the sub-list contains a single element, it holds a character that can be matched. If it contains two elements, it is an interval where all characters from the first element to the second inclusive can be matched. If it is empty then it will match any character. The operation fails unless each character in the input string matches one of the sub-lists.

In the read_prefix and read_suffix operations, the token is a tuple, the first element containing this list and the second element containing either a number specifying how many characters to match or a "*" specifying that as many characters as possible should be matched. The

characters observed by any of these three operations are made available in a script variable named "v_match".

This example illustrates how a number could be specified:

```
number = ["read_all", [["0", "9"]]];
```

This two-part example specifies a standard C-style identifier. The first agent matches the first character of its input to a letter or underscore, and the second matches the rest of the string to letters, underscores or numbers:

```
ident   = ["read_prefix", [["a", "z"], ["A", "Z"], ["_"]], 1],
            ["ident_1"];
ident_1 = ["read_all", [["a", "z"], ["A", "Z"], ["_"], ["0", "9"]]];
```

On the subject of string operations, the pivot operation searches a string to from left to right, finding the first occurrence of the pivoting sub-string. It is sometimes necessary to find the last occurrence and the rev_pivot operation does this by searching a string from right to left.

6. Implementing the translator

Having designed a notation for describing agent behaviour using templates, all that is required for a translator is a program that, given an agent template and input data, will simulate the actions of the agent on that data.

6.1 EDEN scripts

The translator is implemented in five main EDEN scripts and in a few small changes made to the tkeden source code. The first script, "`utils.e`", contains utility functions used by one or more of the other files. It mostly consists of simple string manipulation functions.

The second script, "`blocks.e`", contains functions that implement the translator's awareness of "blocks" as described in the previous section. It maintains a list of all known blocks, allows new blocks to be registered and provides a function named "`extract`" that searches a string for a sub-string, ignoring particular blocks.

The third script, "`ops.e`", contains the functions that implement the string matching operations that agents can perform. These include `prefix, suffix` and `pivot`, and they all use the `extract` function to process strings, splitting them into substrings.

The fourth script, "agents.e", contains a function named "run_agent" that performs the actions of a single agent. This script also contains the functions that implement the different parts of this process, including script execution.

The fifth script, "parsers.e" contains an assortment of high-level translation functions that accept a string and an initial agent to run on it. They maintain a list of all agents waiting to run and execute them one by one using the run_agent function. Because an agent has children that run after it, the agents are synchronised in a tree-like manner. There are functions that run them in a depth-first and breadth-first fashion, dfparse and bfparse, and a function that prints a detailed account of each agent's actions step, verbose_dfparse.

The "program.e" file is optional and contains useful agent templates for numbers and identifiers, etc.

6.2 Modifying the tkeden interpreter

Most of the modifications made to the tkeden code duplicated the functionality of the built in DoNaLD and Scout translators and I copied existing code where possible. I wrote a small amount of code to recognise an EDEN command to change notation to one supported by the

translator and to look up the information required to translate it. I also wrote code to implement a built in EDEN function called "notation" that installs a new language and allows script to be entered in it.

Whereas a conventional parser can parse a string from left to right while it is being typed in, the translator must operate on entire strings because it could read from any position, the beginning or the end. However, since it is connected in the same way as the internal Scout and DoNaLD parsers, which use yacc and lex, it receives its input character by character. The best way to break the input into blocks that could be parsed individually was unclear, but I decided to parse it line by line, buffering the characters and passing the buffer to the dfparse function on reading a carriage return. This is not ideal but anything more complicated would require additional language-specific parsing.

7 An example parser

In this section, I hope to clarify the agent template notation using an example. The following agents translate statements in a cut down version of eddi into EDEN.

The eddip.e file contains the functions to implement the relational operators and those create, modify and display relations.

```
include ("eddip.e");
```

The first agent optionally observes a trailing semi-colon. It is optional because the agent to run on success is the same as that to run on failure.

```
eddi_statement = ["suffix", ";", "eddi_statement_1",
    ["fail", "eddi_statement_1"]];
```

This agent checks if the statement is a "show relation" statement. If so, it creates a child to match the table name and stores script to show the relation later. If not, it goes to the next possible statement form, the "add rows" statement, "eddi_statement_2".

```
eddi_statement_1 = ["prefix", "?", "table_val", ["script",
    ["declare", "relname"], ["setparas", ["relname"]],
    ["later", "showrel (%%);", "relname"]],
    ["fail", "eddi_statement_2"]];
```

This agent checks if the statement is an "add rows" statement. If so, it creates a child to match the table name and one to match the rows to add and it stores script to add the rows later. If not, it goes to the next possible statement form, the "definition" statement, "eddi_statement_3". The string to the right of the "<<" operator is substituted literally into the addvals function, the tuple parsing agents do not translate anything, they only check syntax.

```
eddi_statement_2 = ["pivot", "<<", ["table_name", "tuples"],
    ["script",
    ["declare", "relname"],
    ["setparas", ["relname"], []],
    ["later", "$$ = addvals ($$, %%);", "relname", "relname",
        "v_substrs[2]"]],
    ["fail", "eddi_statement_3"]];
```

This agent checks if the statement is a "definition" statement. If so, it creates a child to match the table name and one to create a definition tree representing the algebraic expression on the RHS. It stores script to define the table variable later. If the string is not a definition statement then it fails, as there are no other eddi statements implemented here.

```
eddi_statement_3 = ["pivot", "is", ["table_name", "rel_exp"],
    ["script",
    ["declare", "relname", "expr"],
    ["setparas", ["relname"], ["expr"]],
    ["later", "$$ is %%;", "relname", "expr"]]];
```

This agent matches a table name and returns it in a form that can be used in the last four statements when creating definitions. The function "ident_ex", defined in "utils.e", creates the template for an agent that matches a standard C-style identifier. It allows the user to specify additional tags; here some script is added to set the first parameter to the table name that was read.

```
table_name = ident_ex ([["script", ["execute", "%% = \"%%\";", "v_paras[1]", "v_string"]]]);
```

This agent matches a table name and returns it in a form that can be used when creating a definition tree.

```
table_val = ident_ex ([["script", ["execute", "%% is %%;",
"v_paras[1]", "v_string"]]]);
```

These agents make up the relational algebraic expression parser. They try to observe one operator at a time, the least precedent first, ignoring everything between brackets. On success, they create children that match relational algebraic expressions and they define the parameter in terms of the child variables.

```
rel_exp   = ["pivot", "+", ["rel_exp", "rel_exp"], ["script",
    ["declare", "p1", "p2"],
    ["setparas", ["p1"], ["p2"]],
    ["execute", "%% is union (%%, %%);", "v_paras[1]", "p1", "p2"]],
    ["ignore", ["bras"]],
    ["fail", "reg_exp_1]];

rel_exp_1 = ["pivot", "-", ["rel_exp", "rel_exp"], ["script",
    ["declare", "p1", "p2"],
    ["setparas", ["p1"], ["p2"]],
```

```
    ["execute", "%% is diff (%%, %%);", "v_paras[1]", "p1", "p2"]],
    ["ignore", ["bras"]],
    ["fail", "reg_exp_2]];
```

The project statement does not match a relational algebraic expression on the RHS. To ensure that the RHS is parsed properly, it is searched right to left using rev_pivot. This ensures that "TABLE % COL_A % COL_B" is not broken into "TABLE" and "COL_A % COL_B" but instead it is broken into "TABLE % COL_A" and "COL_B".

```
rel_exp_4 = ["rev_pivot", "%", ["rel_exp", "attr_list"], ["script",
    ["declare", "expr_part", "attr_part"],
    ["setparas", ["expr_part"], ["attr_part"]],
    ["execute", "%% is project (%%, %%);", "v_paras[1]",
        "expr_part", "attr_part"]],
    ["ignore", ["bras"]],
    ["fail", "rel_exp_5"]];
```

If no operators were found then the next two agents check if the string is a bracketed expression. If not, then the only other thing that it could be is a table name.

```
rel_exp_6 = ["prefix", "(", "rel_exp_7", ["fail", "table_val"], ["script", ["setparas", ["v_paras[1]"]]]];
rel_exp_7 = ["suffix", ")", "rel_exp", ["script", ["setparas", ["v_paras[1]"]]]];
```

An "insert rows" command has tuples on its RHS. These agents ensure they fit the proper tuple description. There is no script associated with these agents because if the strings fit the description then they already are in the form required by the addvals function, and can just be used further up the tree – see the eddi_statement_2 agent.

```
tuples  = ["split", ",", "tuple", ["ignore", ["sq_bras"]]];

tuple   = ["prefix", "[", "tuple_1"];
tuple_1 = ["suffix", "]", "tuple_2"];
tuple_2 = ["split", ",", "tuple_3"];
```

Each element is either a number or a quoted string. The "alphanum" and "float_num" agents are built in.

```
tuple_3 = ["prefix", "\"", "tuple_5", ["fail", "float_num"]];
tuple_5 = ["suffix", "\"", "alphanum"];
```

These agents recognise the column names over which to project. They use the "allparas" instruction to collect all column names into a list that can be passed to the project function.

```
attr_list = ["split", ",", "col_name", ["script", ["allparas", "attrs"], ["execute", "%% is %%;", "v_paras[1]", "attrs"]]];
col_name = ident_ex ([["script", ["execute", "%% = \"%%\";", "v_paras[1]", "v_string"]]]);
```

The final line installs the eddi notation into EDEN. The first parameter specifies that the EDEN code "%eddi" will change to the eddi notation. The second parameter specifies the first agent to run on a string to be translated.

```
notation ("eddi", "eddi_statement");
```

8 Features of the parsing system

The agent-based parsing system approaches several aspects of parsing from an interesting perspective.

8.1 Error detection and correction

Consider how a conventional parser detects errors. An LL parser reads characters from left to right, matching the characters against patterns. These patterns usually contain references to sub-patterns, and the parser traverses this pattern tree in a depth-first fashion. When a mismatch occurs, the parser must backtrack up one level of recursion and try a different branch. If the bad character does not fit any of the possible patterns then the parsing fails. Having tried so many possibilities, the parser does not know which pattern the character was supposed to match and so the whole string must be re-entered.

In the agent-based parsing system, only a small sub-string need be re-entered because error detection is independent of context. To understand this statement, consider that the only kind of syntax error that can occur is if the significant token of an agent is not matched. In contrast with the paragraph above, only one pattern match could have failed – that of the current agent. The string that must be re-entered is, intuitively, not more than the input string to that agent, regardless of the rest of the string seen by the agent's ancestors, i.e. the context. In this way, the translator makes error correction more interactive than in conventional parsers, asking the user for different corrections depending on the nature of the error.

The dfparse function used by the translator implements this interactive error correction; if a string entered into tkeden contains an error, it prompts the user to re-enter the relevant sub-string. For an example, in PL/0, constants are declared with a statement of the form:

```
const a := 1, b := 2, c := 3;
```

This would probably be parsed by matching a prefix of “const” and a suffix of ";", splitting around the commas into individual constant declarations and, for each one, pivoting around the “:=” sign to get the variable name and initial value. If the string contained “b == 2” instead of “b := 2”, the error would detected at the pivot operation, when an agent looks at the string “b == 2” and tries to find “:=”. The only string that needs to be re-entered is the agent's input string, “b == 2”. The rest of the string is irrelevant to the error.

8.2 Building parsers

A hallmark of EM is that a modeller does not have to conceive an entire system in his mind when he starts to model it. In conventional grammars, the syntax of a language often has to be fully understood before starting to write the grammar. Using the template notation, you can start to describe a language while you are coming to understand it and without having to think about the whole language. This makes building parsing systems more intuitive and they develop naturally.

A good example is writing an arithmetic expression parser. To correctly handle operator precedence, an "expression" is conventionally defined as one or more "terms" combined with the "+" or "-" operators where a "term" can be an identifier, number or an "expression" in brackets. This is not a natural way to think about an expression. When writing the eddi parser included on the attached disk, I started by checking with an agent that pivots around the "+" operator, ignoring anything within brackets. If this succeeds then I match the agent's output strings to an expression. I next implemented agents, to create on failure, that try to pivot around each operator in turn. The operator with lowest precedence is checked before those with higher precedence. I then added an agent that acts if this fails, which looks for brackets at the beginning and end of the string and matches the middle to an expression. If this fails then the string must match a table name. The operator precedence problem is solved by the fixed order in which the agents act and I was able to build the parser slowly and naturally.

The approach to parser design that the agent template notation encourages is particularly relevant to the expected users of the translator. Since tkeden is used mostly by modellers who are familiar with EM perspectives, they would feel comfortable taking the same approach when defining their notations and when building their models.

8.3 Definitive agent templates

The template notation becomes more powerful when you consider that each agent template is an EDEN variable and, as such, could be a definition. Templates can be written that generate agents with different behaviours based on the value of a variable, and templates can be changed during the parsing process. The palindrome parser given below uses this idea to change the suffix it is looking for to be the same as the prefix it has just read. It maintains a variable called p_str, which the initially it sets to the first character of the input string. To ensure that the empty palindrome is matched, on failure it matches `nothing`, which is a build in agent that matches the empty string.

```
palindrome = ["read_prefix", [[], 1], "pal_1", ["script",
    ["execute", "p_str = \"%%\";", "v_match"]], ["fail", "nothing"]];
```

The next two agents ensure that the input string either ends with the character p_str or is the empty string. If neither of these conditions holds then the string is not a palindrome and the parsing fails. Note that "pal_2" is not a normal EDEN variable but a definition.

```
pal_1 = ["literal", "", ["fail", "pal_2"]];
pal_2 is ["suffix", p_str, "palindrome"];
```

The more powerful example below matches sequences of a square number of '1's. This is a difficult parsing problem but, using definitive templates, a count of the number of ones can be kept and used to calculate what the next agent should match. The first agent reads a '1' and sets the "onecount" variable to 1.

```
onecount = 0;

squareones = ["prefix", "1", "sq_2", ["script",
      ["execute", "onecount = 1;"]]];
```

An "sq_1" agent is created whenever a square number of '1's has not been read. It must match a '1' or the parsing process will fail.

```
sq_1 is ["prefix", "1", issqare (onecount), ["script",
      ["execute", "onecount++;"]]];
```

An "sq_2" agent is invoked if a square number of ones has been read. It can match nothing, indicating that the whole string has been read and it contains a square number of '1's, or it can match another '1', aiming for the next highest square number.

```
sq_2 is ["literal", "", ["fail", "sq_1"]];
```

The "issquare" function is used by the "sq_1" agent to determine which child should be created. If the value of its parameter plus one is a square, it returns the string "sq_2". If not, it returns the string "sq_1".

```
func issqare {
    para num;
    auto sroot;

    sroot = sqrt (float (onecount + 1));
```

```
    if (sroot == (float (int (sroot))))
        return "sq_2";
    else
        return "sq_1";
}
```

9. Conclusions

I think that the ideas about parsing that I have developed could have relevance in other areas of computing, especially within EM, and when approaching some less understood subjects, they could at least point in the right direction. In particular, the management and control of agents that the project demonstrates could provide some indication of how an agent-oriented programming paradigm would look. In addition, the structured observation performed by the agents could be generalised to provide the somewhat free, undirected observation of EM with some method and motive, or to suggest how the simple data types of EDEN could be developed into something with more structure.

Another interesting application is in the area of syntax directed parsing. The fact that synchronisation is only enforced between directly related agents means that a particular section of a string could be either omitted by the user or replaced by a dummy value until they were ready to fill it in. The parser could delay parsing that section, observing the other parts of the string that the user had supplied.

The project could have been developed further had I the time to write a parser for one of the more powerful and commonly used notations such as DoNaLD or Scout. This would be an interesting and useful extension. Another part of the project that I would have liked to develop is the clumsy notation used to describe agents. This could be improved in readability and ease of use.

In terms of the problem that I originally set out to solve, this project has been a success. I have modified the tkeden interpreter to translate into EDEN a script that is written in any definitive notation that it has been configured to understand. The design incorporates EM ideas and, as such, I think that it is suited to the environment in which I expect it to be used. I feel that I have come to understand many new ideas and perspectives on parsing and computing in general and that this project has been a significant learning experience for me.

Acknowledgements

Appendix I – the contents of the attached floppy disk

The disk included with this report contains the following directories.

EDEN – contains all EDEN source code for the translator from chapter 6

tkeden – contains all modified source files for the tkeden interpreter and one new source file, as discussed in chapter 6

notations – contains parser definitions for the eddi notation from chapter 5, the Pl/0 language from chapter 8 and the parsers that recognise palindromes and the sequences of a square number of '1's from chapter 8

iterations – contains the code from a number of iterations of the eddi parser discussed in chapter 2