CS405 Introduction to Empirical Modelling
Lab 6: Notations and Parsing

In previous labs you have been using the standard notations in EDEN (e.g. DoNaLD or SCOUT). You may also have experienced some other notations (e.g. Sasami, ARCA and EDDI) and there are others that you probably haven't seen. Each of these notations is generally used for a particular type of model (e.g. Sasami for 3D modelling). When developing models, the user is free to mix and match several notations – the best notation is used for the task. However, you might find that no notation exists for the type of model you wish to produce ...

All is not lost; you can add new notations to tkeden yourself using the Agent-Oriented Parser (AOP). This parser can be used to translate your custom notation into eden code (or any other notation). It does this all within tkeden, so that you can build up your parser, as you would any other model, by adding definitions.

How does it work?

The AOP will take any input string and pass it to one or more agents which will attempt to parse the input using a set of rules (that you define). An agent may create sub-agents to work on different portions of the string, until a hierarchy of agents can parse the whole string. Each of these agents may perform actions in the environment.

The types of operations an agent can perform on an input are:

- 'literal' matches the entire input string.
- 'prefix' matches a string at the beginning of the input and passes the remainder of the string to a child agent.
- 'suffix' matches a string at the end of the input and passes the remainder of the string to a child agent.
- 'pivot' matches the leftmost occurrence of a string anywhere in the input and creates two child agents (one for the left substring and one for the right substring).

As well as matching strings, the AOP can also use regular expressions for certain operations. A new parser is defined by a set of definitions known as parser rules. Each rule is written using a custom made notation (which is itself written in the AOP) and specifies the type of operation to be performed and a pattern to be matched. The rule may produce child agents, in which case the rule must specify which rules to use for the child agents. The rule may include a 'fail' clause which is the name of another rule to be applied if the current one does not match the pattern. The notation for defining a parser (`%aop`) is only supported in version 1.65 and above of the EDEN tools.

1 My first notation

In keeping with Computer Science tradition, the first notation you write will emulate a simple 'hello world' program. The specification for this is:

- If the parser observes the input 'hello_tkeden', then it will output 'hello world' to the console.
- For any other input the parser will fail.

To implement this notation we need a rule which looks for 'hello_tkeden' as the input string. Each rule is an Eden variable which describes the action the parser should take. This 'helloworld' rule can be described as:

```
%aop
<helloworld> = "hello_tkeden" : do {writeln("hello world");} now;
```

In this example the rule is called 'helloworld', rule names are denoted within chevrons (i.e. `<helloworld>`). On the right hand side of the definition, various properties of this parser agent can be specified. In this example, the agent `<helloworld>` is defined to wait for the text `"hello world"`, and to perform an action on a successful match. The action here is to execute the Eden code `writeln("hello world");`. The only required part of a `%aop` parser-agent (rule) definition is the pattern that the agent is matching against. Throughout the lab, we will introduce various ways of defining this pattern and also introduce other optional parts of the definition syntax. However, the basic syntax of a rule definition is as follows:

```
<RULE_NAME> = PATTERN_DESCRIPTION            (The pattern description, Required.)
            : do {EDEN_ACTIONS} NOW_OR_LATER (Eden actions, Optional.)
            : ignore [IGNORE_LIST]           (Any ignore blocks, Optional.)
            | <NEXT_RULE_NAME>;              (The next rule to try, Optional.)
```

To test our parser we must install it as a notation.

```
%aop
notation %helloworld = <helloworld> split on "\n";
```

Task 1

Make the two `%aop` definitions given above and a new notation should appear in the tkeden environment. You can switch to this notation and begin parsing input. Check that parsing "hello_tkeden" prints the correct response, any other input should result in an error message being printed in the terminal.

2 Counting hello worlds

Now we will make the parser's actions dependent on previous definition. The `%helloworld` notation will now count the number of statements successfully parsed.

- If the parser observes the input 'hello_tkeden', then it will output 'hello world 1' to the console.
- If the parser observes the input 'hello_tkeden' a second time, then it will output 'hello world 2' to the console. And so on for 3, 4, 5, ... etc.
- For any other input the parser will fail.

Task 2

Redefine `<helloworld>` parser rule so that the parser has this behaviour. Remember to switch back to `%aop`.

3 The calc notation

In the directory `/dcs/emp/empublic/teaching/cs405/lab6/` you will find a basic notation for parsing numerical expressions. Open the file `calc.aop` and try to understand how the parser works. The 'pivot' operation is used extensively in this notation. This operation will attempt to match a symbol somewhere in the string and pass the two substrings either side of this symbol to two new parser agents.
For an example of a 'pivot' parser rule, look at the pattern description of `calc_expr`. With this rule the parser pivots on the addition operator (+).
Also, note the pattern descriptions of `<calc_expr5>` and `<calc_expr6>`, in these rules the string is positioned either as a prefix or a suffix to the name of the subagent.

Task 3

Execute the definitions in `calc.aop` and the `%calc` notation should appear.
Test your parser by trying various inputs (e.g. 1+2*3/4). Does your parser observe usual precedence rules? Identify any peculiarities in the way the parser evaluates particular expressions (e.g. 1+2-3-5). How do you explain these?

4 More calc extensions

The `%calc` notation is able to handle any numerical expression containing integers and operators (add, subtract, multiply and divide). The next stage in our calculator model is to allow real numbers to be used.
To match a real number we will need to be able to use a regular expression rather than a regular string in our pattern description. To do this we use parentheses rather than quotes in the pattern description. Regular expressions are available with literal, suffix and prefix rules but not with pivot rules.

```
<number> = ([0-9]+);                    Matches one or more numeric digits


<name> = ([A-Z][a-z]*) <surname>;       Matches a character prefix (with leading capital
<surname> = ([A-Z]+);                   letter) and then a surname in capitals.
```

Task 4

Modify the parser so that it can calculate expressions containing real numbers (e.g. 1+2.3-4.5).

Now that the notation can take real numbers, add 'pi' to the notation. The notation should now:

- Understand the operators +, -, * and /.
- Take each term to be either an integer, a real or the constant pi.

Examples of acceptable inputs are:

```
1 + 2 * 3 - 4
8.8 / 2.2
5.67 * 2 * 8.9
1 + pi - 4.1415926
```

5 A palindrome parser

In the previous exercises, we have introduced literal, prefix, suffix and pivot pattern descriptions using patterns which are either strings or regular expressions.
It is also possible to use the contents of an eden observable (assumed to be a string) as the pattern. References to observables are made within backticks (`). In the following example, the two rules have the same functionality:

```
%aop
<course> = "hello" : do {writeln("bye!");} now;
%eden
query = "hello";
response = "bye!";
%aop
<course2> = `query`: do {writeln(response);} now;
```

Although these two rules have the same functionality, we can extend the power of the `<course2>` rule by adding a dependency in Eden. How about an international notation?

```
language = "french";
query is (language == "french") ? "salut" : "hello";
response is (language == "french") ? "au revoir!" : "bye";
```

Try to use dependency on external observables in the next task:

Task 5

Build a palindrome parser with as few rules as possible. A palindrome is a string of characters such that if you reverse the order of all the characters then you still have the same string. E.g. 'divid', 'abccba', 'abcdefedcba', etc.

6 Building a textual interface for another model

So far we have been simply parsing textual inputs and responding to them, this is input/output command parsing. The real motivation for parsing EDEN is to parse definitive notations. These are notations which define a state.
In this section, you will be introduced to a small graph model which represents nodes and edges in a DoNaLD display. The positions of nodes can be modified by clicking and dragging a node with the left mouse button pressed. Edges can be introduced using the right mouse button.
This model has two underlying observables, `nodes` and `edges` and the graphical representation is dependent on those. A notation for adding and positioning nodes could be designed like this. The eden procedure `addNode()` and the function `updatePosition()` are already defined.

```
%nodes
node n              -> action: addNode("n");
position n 3 3      -> action: nodes = updatePosition("n",[30,30],nodes);
```

Task 6

Include the file `/dcs/emp/empublic/teaching/cs405/lab6/graph.eden`. Try moving nodes with the mouse. This action is redefining the `nodes` observable.

Task 7

Define the notation %nodes as specified above.

Task 8

If you fancy a challenge, try adding query support for the notation. Executing `?n` should print out information about a node's current position. Is it possible to provide an eden procedure which saves-out the current `%nodes` definitions to a file?

Conclusion

If you are able to complete all the exercises then you have completed the basic introduction to the agent-oriented parser. There are several features that have not been covered in this lab, which you can find in the documentation. If you have time then read about blocks and the different types of script actions that you can create. Think about how you might use these features to parse a definitive language like donald or scout.