CS405 Introduction to Empirical Modelling
Lab 7: Notations and Parsing

Introduction

In previous labs you have been using the standard notations in tkeden (e.g. eden, donald, scout). You may also have experienced some other notations (e.g. sasami, arca and eddi) and there are probably others that you 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 parser will take any input 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 occurence of a string anywhere in the input and creates two child agents (one for the left substring and one for the right substring).

Other actions are similar to the above but include regular expressions.

Each rule specifies the tupe 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.

Note: For these exercises you might find it helpful to refer to the parser documentation which can be found at '/dcs/emp/ant/public/lab7/aopdoc.pdf'.

Exercise 1: Loading the AOP

Using the tkeden environment, load in the agent-oriented parser from '/dcs/emp/ant/public/lab7/aop/' by executing the file 'Run.e'. This will not add anything visually to your tkeden environment, but the environment is now capable of adding new notations.

Exercise 2: 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:

```
helloworld = [ "literal",                      ## list element 1
               "hello tkeden",                  ## list element 2
               [ "action",                      ## list element 3
                 [ "now", "writeln(\\"hello world\\");" ] ] ];
```

The rule is described as an eden list. Each element in the list describes the behaviour of an agent applying this rule. In this example the rule is called 'helloworld'. The first element in the list is a string which describes the operation to be performed (in this case 'literal' will attempt to match the whole input with a pattern). The second element is the pattern to be matched (in this case it is the string 'hello tkeden'). The third element is a list beginning with the string 'action' which indicates that this is a list of actions to be performed. An action is a portion of code to be executed in tkeden (in this case the code is a 'writeln' statement).

To test our parser we must install it as a notation in tkeden. First we create an initialisation rule:

```
helloworld_init = [ "\n", "helloworld", [] ];
```

And then we install the notation in tkeden:

```
installAOP("\%helloworld", "helloworld_init");
```

A new notation will now appear in the tkeden environment. You can switch to this notation and begin parsing input. You should find that if you enter "hello tkeden" then tkeden will print out the desired response. If you enter anything else then you will receive a parse error.

Exercise 3: Counting hello worlds

Switch back to eden and redefine the 'helloworld' rule so that the notation has a different behaviour:

- 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.

Hint: You should only need introduce a counting variable and modify the 'action' section of the rule.

Exercise 4: The logo notation

A 3rd year project by Gavin Edwards produced a model of logo (the mathematical drawing tool). In this model, the turtle was controlled by executing eden functions, so that for instance the procedure call `fd(100);` moves the turtle forward 100 steps. The original logo tool (if you remember back to the old BBC computers) used commands of the form `forward 100`. With the new parser, Chris Roe was able to build a `%logo` notation that translated the natural logo commands into the eden equivalent.

Ensure that you have a tkeden environment with the AOP loaded (as in exercise 1). In the directory '/dcs/emp/ant/public/lab7/logo/' you will find some scripts containing the logo model and the parser. Execute 'Run.e' to load this model.

You should see the logo window with the turtle in the centre. Also, in the tkeden environment a new notation will have been installed (`%logo`). In `%eden`, type `fd(100);` to make the turtle move 100 steps forward (this is how the model originally worked). Now move into the `%logo` notation and try to move the turtle using the standard logo commands: `forward x, backward x, left x, right x, pen up, pen down` (where x is an integer).

Open up the 'notation.e' file in a text editor and examine the rules. Can you explain how the parser works?

Gavin Edwards implemented some other functions to give the turtle more features. There exists a function called pickRed() which changes the pen colour to red, and similar functions for blue, yellow, black. Can you modify the notation so that these colours can be used?

Hint: Modify the rule for 'pen down' so that it matches 'pen black'. Create rules for 'pen red', 'pen blue', 'pen yellow'. Link the rules such that if 'pen black' fails to match then the 'pen red' rule is chosen, and so on for all the colours.

Exercise 5: The calc notation

In the directory '/dcs/emp/ant/public/lab7/calc/' you will find a basic notation for parsing numerical expressions. Open the file 'basic.e' 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. In the expression rules it is used to search for a plus (+) or a minus (-) in the input.

Load 'basic.e' and the `%calc` notation should appear in tkeden. Try some basic calculator expressions and check the result on the console (e.g. 1+2-3).

Extend the set of calculator expressions that can be parsed by adding a rule for multiplication (*) and a rule for division (/). 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?

Exercise 6: More calc extensions

The `%calc` notation from exercise 5 is able to handle any numerical expression containing integers and operators (add, subtract, multipy and divide). The next stage in our calculator model is to allow real numbers to be used. 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 a constant (e.g. pi).

Examples of acceptable inputs are:

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

For further reading on the calc notation you should refer to the parser documentation.

Exercise 7: A palindrome parser (grand finale)

Try to 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.

Hint: You may be able to use dependency in your rules for this exercise, and hence discover the real power in the agent-oriented parser!

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.