11 Appendix 4 - A tool for exploring syntactic structures, complexity and simplification

11.1 Overview

In order to aid the exploration of the syntactic structures, I have written a program. This allows the specification of a flattened sequence of interdependent syntactic structures (as specified in section 5.5.4 on page 112 and formalised in section 10 on page 182). Each structure has its own rules, allowing for substitutions from its immediate sub-syntax. Each such structure can then generate its members on demand. Thus this tool implements the automatic generation from an arbitrary dependent chain of syntactic structures as discussed in section 5.5 on page 108 and formalised in Appendix 3 - Formalisation of Syntactic Structure.

11.2 Examples

Here are a couple of simple examples to illustrate its use. Apart from the launching menu each syntactic structure has its own window, with its name in the window title bar, the name of any sub-syntax it might draw substitutions from just below, the list of generating rules on the right and the generated members on the left. Since, in this tool dependency on sub syntaxes is limited to a linear chain, it can be left to the context the symbols in the rules for which substitution is required.

Thus figure 29 shows a simple syntactic structure with two simple rules, to generate an indefinite number of unique items.

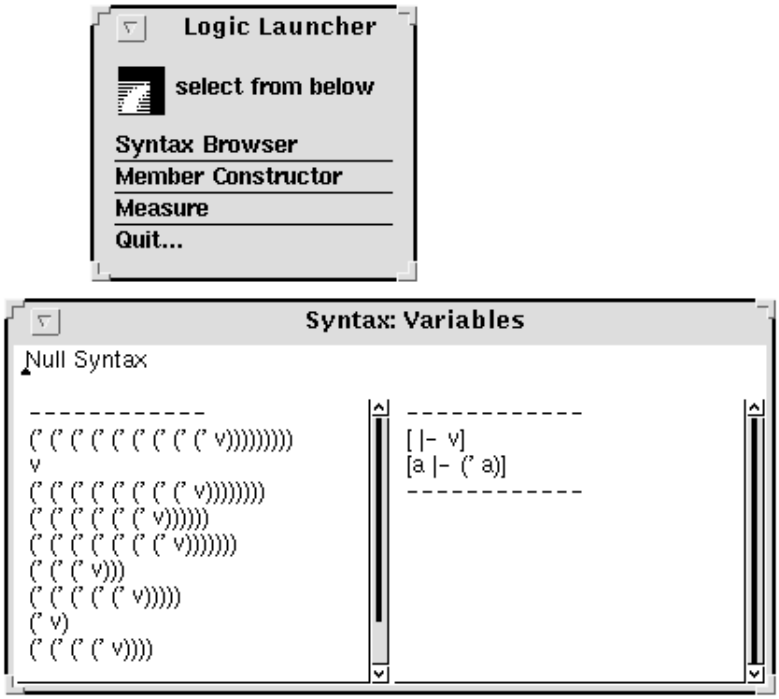

Figure 29. A window on a single syntactic structure

While figure 30 shows three syntactic levels: the first of which specifies a fixed list of three variable names; the second which generates well formed implicational formulas using the variables syntax and the final level which generates the theories of the implicational fragment of R.

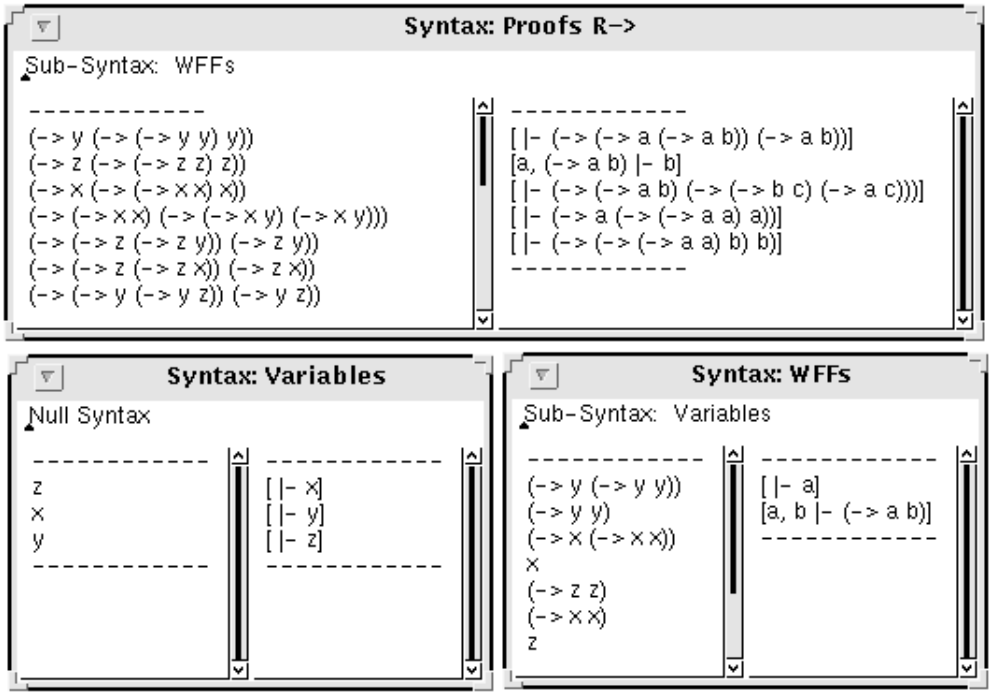

Figure 30. Three inter-dependent syntactic structures in separate windows

11.3 Programming

The algorithm was implemented in the object-orientated programming language Objectworks/Smalltalk 4.1. It is composed of a hierarchy of classes to implement the recursive generation of tree structures. Each class in the hierarchy is an incremental generator, in that it only needs to store minimal data in order to produce the next tree in the sequence (in order to avoid an exponential explosion of memory usage). For example, the "TreeGenerator" class uses two other generators: one to generate possible skeletons of valid polish notation strings and the other to generate the possible substitutions of symbols into this skeleton.

Although a few critical generators cache their results for time efficiency (such as the generator that produces the output corresponding to the cross product of two other generators), they act as 'lazy lists', in that elements are only generated as required.

Since Objectworks/Smalltalk 4.1 is an object-orientated language the work in generating from the syntactic structures is done by a series of object instances of classes. Broadly it is the classes that hold the program code and the instances of those classes that

execute the code and store the data. Classes are arranged in a hierarchy so that general code that is placed in a higher class can be ‘inherited’ by lower classes (unless this is overridden). Objects communicate by sending messages to each other.

The generating class hierarchy is show in figure 31 below. Dependency is indicated by the indentation, those other classes it uses are listed after the class name. There are also classes for syntaxes, trees, rules, trees of rules, lists and the empty syntax, but I will not show these are they are either standard or preform simple storage and display functions.

```
Generator
    TransparentGen
        StorageGen
            StoragePowerCrossSymbolGen (calls PowerCrossGen)
        RuleTreeLeafGen
        SingleListGen
        RepeatingGen
    PairGen
        PairListGen
            PowerCrossGen (calls NullGen or SingleListGen or
                                                PairListGen)
    NoGen
        NatNoGen
            ModNoGen
    SizeGenGen
        SizeGenSyntaxGen (calls StoragePowerCrossSymbolGen
                                and one of: NatNoGen
                                or ModNoGen)
    BunchGen
    TreeSubsGen (calls SizeGenSyntaxGen and TreeGen)
    TreeGen (calls PolNoLenGen and GenBunchGen)
    GenBunchGen (calls BunchGen)
    PolNoLenGen
    ListGen
```

Figure 31. The hierarchy of generator classes

Each syntactic structure has generator assigned to it dependent on the type of its rules and whether there is a sub-syntax which it will need to access for substitutions. This could be either a

- RuleTreeLeafGen generator on a ListGen of rules if the rules have no antecedents - this just generates the rules in turn as a rule tree;

- otherwise a TreeGen if there is no non-null sub-syntax - this generates all possible legal trees of rules using PolNoLenGen to generate polish notation skeletons into which the possible rule substitutions (generated by SizeGenSyntaxGen) are slotted;
- and a TreeSubsGen if there is - which does the same as TreeGen but substutes all combinations of substitutions from lower syntaxes as required.

The full code for all these classes is far to cumbersome to include here but can be read if desired at URL: http://www.cpm.mmu.ac.uk/~bruce/thesis/code or supplied on request by mailing me at (b.edmonds@mmu.ac.uk).

11.4 Interface

The interface is very rudimentary, consisting only of scrollable passive output in smalltalk windows and input via smalltalk expressions executed on the smalltalk workspace.