FrameScript: A Multi-modal Scripting Language

M. McGill
mmcgill@cse.unsw.edu.au

C. Sammut
claude@cse.unsw.edu.au

J. Westendorp
jhw@cse.unsw.edu.au

W. Kadous
waleed@cse.unsw.edu.au

Abstract

This document is a user manual for the FrameScript scripting language. FrameScript combines rule based scripts with frames [6] and an untyped expression language. It has been developed for the purpose of rapid prototyping of conversational, speech and multi-modal interfaces.

FrameScript uses a number of techniques to allow scripts to be modularized and so simplify the task of writing them. It has also been designed in such a way that it is easily extensible and so can be used in a variety of applications and with a variety of technologies.

This user manual also describes MicaBot a FrameScript extension that allows FrameScript to be used with the Mica [4, 5] agent architecture. MicaBot allows Mica agents to be written using FrameScript.

Contents

1 Overview **8**
- 1.1 MICA . . . 8
- 1.2 FrameScript . . . 11

2 FrameScript **12**
- 2.1 Frames . . . 12
  - 2.1.1 Slots . . . 13
  - 2.1.2 Generic Frames . . . 13
  - 2.1.3 Daemons . . . 13
  - 2.1.4 Instance Frames . . . 14
  - 2.1.5 Multiple Inheritance . . . 14
- 2.2 Scripts . . . 16
  - 2.2.1 Rules . . . 16
  - 2.2.2 Patterns . . . 16
  - 2.2.3 Responses . . . 17
  - 2.2.4 Pattern Component Numbering . . . 18
  - 2.2.5 Domains, Topics and Triggers . . . 19
  - 2.2.6 Inheritance . . . 21
  - 2.2.7 Slots . . . 22
  - 2.2.8 Failsafes . . . 22
  - 2.2.9 Rule Ordering . . . 22
  - 2.2.10 Daemons . . . 23
  - 2.2.11 Abnormal Scripts . . . 24
  - 2.2.12 Pattern Matching Algorithm . . . 24
- 2.3 Expression Language . . . 26
  - 2.3.1 Statements . . . 26
  - 2.3.2 Numbers . . . 26
  - 2.3.3 Strings . . . 26
  - 2.3.4 Atoms . . . 27
  - 2.3.5 Lists . . . 27
  - 2.3.6 Expression Lists . . . 27
  - 2.3.7 Patterns . . . 27
  - 2.3.8 Sequences . . . 28
  - 2.3.9 Alternatives . . . 29
  - 2.3.10 Ripple Down Rules . . . 29
  - 2.3.11 Functions . . . 29
  - 2.3.12 Forall Loops . . . 31
  - 2.3.13 Variables . . . 31
- 2.4 Modules . . . 32
  - 2.4.1 Subroutines . . . 32
  - 2.4.2 MaxModule Example . . . 34

3 Multi-modal Interaction **34**
- 3.1 Multi-modal Example . . . 34

4 GUI **37**
4.1 Message Dialog . . . 37
4.2 Question Dialog . . . 38
4.3 RDR Maintenance GUI . . . 38
4.4 Frame Browser . . . 39
4.4.1 Menu . . . 39
4.4.2 Transcript . . . 40
4.4.3 Generic Frames . . . 40
4.4.4 Instance Frames . . . 40
4.4.5 Functions . . . 41
4.5 Script Browser . . . 42
4.5.1 Menu . . . 42
4.5.2 Domains . . . 43
4.5.3 Scripts . . . 44
4.5.4 Functions . . . 46
4.5.5 Conversation GUI . . . 46

5 MicaBot **46**
5.1 Speech Alternatives . . . 47
5.2 SimpleTextAgent . . . 47
5.3 MicaRunner . . . 47

6 Discussion **49**

A BNF **52**

B Built-in Subroutines **56**
B.1 Operators . . . 56
B.1.1 + . . . 57
B.1.2 - . . . 57
B.1.3 * . . . 57
B.1.4 / . . . 57
B.1.5 mod . . . 58
B.1.6 < . . . 58
B.1.7 <= . . . 58
B.1.8 > . . . 58
B.1.9 >= . . . 58
B.1.10 == . . . 59
B.1.11 != . . . 59
B.1.12 = . . . 59
B.1.13 and . . . 59
B.1.14 or . . . 59
B.1.15 not . . . 60
B.1.16 of . . . 60
B.1.17 in . . . 60
B.1.18 new . . . 60
B.1.19 ^ . . . 60
B.1.20 # . . . 61
B.1.21 to . . . 61
B.1.22 var . . . 61

B.1.23 forall . . . 61
B.1.24 -> . . . 62
B.1.25 Precedence Table . . . 62
B.2 General Functions . . . 62
B.2.1 trace . . . 62
B.2.2 verbose . . . 62
B.2.3 atom . . . 63
B.2.4 defined . . . 63
B.2.5 undefined . . . 63
B.2.6 set . . . 63
B.2.7 number . . . 63
B.2.8 integer . . . 63
B.2.9 list . . . 63
B.2.10 cons . . . 64
B.2.11 member . . . 64
B.2.12 head . . . 64
B.2.13 tail . . . 64
B.2.14 nth . . . 64
B.2.15 length . . . 64
B.2.16 append . . . 65
B.2.17 delete . . . 65
B.2.18 fixrdr . . . 65
B.2.19 rdr . . . 65
B.2.20 print . . . 66
B.2.21 error . . . 66
B.2.22 ask . . . 66
B.2.23 eval . . . 66
B.2.24 quote . . . 66
B.2.25 load . . . 66
B.2.26 load_module . . . 67
B.2.27 output_to_file . . . 67
B.2.28 close_output . . . 67
B.2.29 print_as_text . . . 67
B.3 Frame Subroutines . . . 67
B.3.1 frame . . . 68
B.3.2 generic . . . 68
B.3.3 instance . . . 68
B.3.4 instances_of . . . 68
B.3.5 put . . . 68
B.3.6 replace . . . 69
B.3.7 remove . . . 69
B.3.8 destroy . . . 70
B.4 Script Routines . . . 70
B.4.1 script . . . 70
B.4.2 domain . . . 70
B.4.3 pattern . . . 70
B.4.4 register . . . 70
B.4.5 goto . . . 70
B.4.6 current_context . . . 71
B.4.7 previous_topic . . . 71

B.4.8 new_event 71
B.4.9 bot 71
B.4.10 match 72
B.4.11 failsafe 72
B.4.12 question 72
B.4.13 return 73
B.5 GUIs 73
B.5.1 dialog_message 73
B.5.2 dialog_question 73
B.5.3 frame_browser 73
B.5.4 script_browser 73
B.5.5 fix_rdr_gui 74
B.6 MicaBot 74
B.6.1 micabot 74
B.6.2 mica_connect 75
B.6.3 mica_register 75
B.6.4 mica_unregister 76
B.6.5 mica_read_mob 76
B.6.6 mica_write_mob 76
B.6.7 mica_delete_mob 77
B.6.8 mica_query 77
B.6.9 mica_write_wait_for_reply 78
B.6.10 get_mob_name 78
B.6.11 current_micabot 78

C Utility Functions **79**
C.1 Subroutine Argument Type Checking 79
C.1.1 check_alternatives 79
C.1.2 check_atom 79
C.1.3 check_boolean 80
C.1.4 check_compound 80
C.1.5 check_domain 80
C.1.6 check_exprlist 81
C.1.7 check_frame 81
C.1.8 check_generic 81
C.1.9 check_instance 82
C.1.10 check_integer 82
C.1.11 check_list 82
C.1.12 check_number 83
C.1.13 check_pattern 83
C.1.14 check_rdr 83
C.1.15 check_script 84
C.1.16 check_sequence 84
C.1.17 check_string 84
C.2 Files/Modules 85
C.2.1 loadFile 85
C.2.2 loadModule 85
C.2.3 setOutput 85
C.2.4 closeOutput 85
C.2.5 FileNotFound 85

C.2.6 evloop 85
C.3 Miscellaneous 85
C.3.1 compress 85
C.3.2 getMessage 85
C.3.3 getName 85
C.3.4 getPattern 85
C.3.5 sortAtomList 85
C.3.6 IOError 86
C.3.7 isPattern 86
C.3.8 isUnaryPattern 86
C.3.9 isPatternElement 86
C.3.10 formatComment 86
C.3.11 unformatComment 86
C.3.12 unformatComment 86
C.3.13 checkAllReferences 86

D Serialisation 87

List of Figures

1 Agents communicate via MICA's blackboard 10
2 Example Generic Frame 13
3 Example Instance Frame 14
4 Multiple Inheritance Example 15
5 Script Example 15
6 Rule Condition Example 17
7 Rule Evaluation Condition Example 17
8 Sequence Example 18
9 Alternatives Example 18
10 Conditional Response Example 18
11 Match Component Example 19
12 Component Numbering Example Non-terminals 19
13 Topic Example 21
14 Script Inheritance Example 21
15 Script Daemons Example 23
16 Abnormal Script Example 24
17 Evaluation Statements 26
18 Function Definition Example 30
19 Function Type Checking Example 30
20 MaxModule Example 35
21 Example Multi-modal Script 36
22 Message Dialog 38
23 Question Dialog 38
24 RDR Maintenance GUI 39
25 Frame Browser 40
26 Instance Frame Browser 41
27 Function Browser 41
28 Script Browser 42
29 Domain Browser 43

30 Script Daemon Browser 44
31 Abnormal Script Browser 45
32 Function Browser 45
33 Conversation GUI 46
34 Speech Alternative Example 47
35 SimpleTextAgent 48
36 MicaRunner Startup Script Example 48

List of Tables

1 Component Numbering Example 20
2 Pattern Match Examples 25
3 Pattern Examples 28
4 Operator precedence 62

1 Overview

This document describes *FrameScript*, a language for creating multi-modal user interfaces. FrameScript is a multi-paradigm language that enables rule-based programming as well as frame representations and simple functional evaluation. It is derived from Probot [7], a language intended solely for text-based conversational agents. FrameScript extends Probot in providing mechanisms for interaction through a variety of devices; FrameScript also allows inheritance of scripts.

FrameScript has a companion program, called MICA, that coordinates communication between agents. FrameScript's design is better understood if the reader has some knowledge of the context in which it is intended to be uased. Therefore, the following sections give overviews of both programs before focussing on the details of FrameScript.

Our goal is to provide easy-to-use and intuitive mechanisms for interacting with intelligent environments. Two major software systems have been developed for this purpose: *FrameScript* is a language for writing scripts that implement multimodal user interactions. It incorporates rules for describing natural language conversations as well as interpreting events in other modalities including touch screens and gestures.

MICA is a system that provides a simple but powerful method for software agents to communicate with each other. These agents may control devices in an intelligent environment or they may provide services such as speech recognition, natural language processing or access to resources on the internet.

The main advantage of these programs is that they make the development of applications involving multi-modal interfaces relatively quick and easy.

1.1 MICA

MICA (Multimodal Inter-agent Communication Architecture) is a middleware layer for pervasive computing that is especially well suited to sharing of information between users, learning user's preferences and interacting with the user through many devices and modalities. It is based on the idea of a blackboard: a global shared memory which acts as both a communication and storage mechanism. It uses an extremely simple API that is easy to program, but still small enough to fit on PDAs and mobile phones.

In designing MICA, the design goals were to:

- Allow applications to be built in such a way that the interface to the application and the application themselves are clearly separated, to support multimodal interaction.
- Operate in heterogeneous environments, in particular, it should be able to run on low-power and mobile devices. This also means that it should support low-level programming languages such as C.
- Allow users to use multiple input and output modalities and devices, such as speech, gesture, audio and video; at the same time allowing both multimodal input fusion and multimodal output generation. For example, the user should be able to begin answering an e-mail on her PDA using

speech, then when she gets to her office continues working on the same e-mail on her desktop from where she left off.

- Support learning of user's preferences and patterns of usage.
- Use a simple abstraction that developers can learn and use quickly.
- Support both security and privacy measures.
- Act as an interpersonal communication mechanism, as well as allowing all of one person's communication tools to communicate with each other.

MICA's is based on the blackboard architecture [3]. Groups of agents use the blackboard to share knowledge and communicate. When something is written to the blackboard, each agent examines it and sees if it can make a contribution to solving a problem. In MICA, we update the idea of a blackboard by supporting distributed execution and network communication, using an object-oriented data representation and adding security and privacy mechanisms.

There are three entities used in the MICA design:

1. The blackboard is the core of MICA. It is similar to a telephone switch with memory. All interactions between agents flow through the blackboard in much the same way that all phone calls in a town go through a switch. Much of the power of the blackboard comes from its ability to allow many agents to share information. It is expected, that in use, there would be one blackboard for each person or environment, such as a car or house.
2. Agents are entities that access and contribute to the information on the blackboard. The two main types of agents are: interaction agents that convey information to and from the user such as GUIs, proxies for telephone connections and devices in the environment; and computational agents that provide services such as speech recognition, web access and e-mail. Once connected to a blackboard, agents can read, write and query objects on the blackboard. In addition, they can register for new objects written on the blackboard. When a new object of interest to the agent is written to the blackboard, the agent is informed of its arrival.
3. MICA Objects (or mobs for short) are the basic unit of information in MICA. Mobs are the things that agents actually read and write from the blackboard. Mobs are very similar to objects in an object-oriented framework, but also share some characteristics with frames. Each mob has a type defined by the user - similar to a class in an object-oriented language. In addition, mobs have slots, which are similar to fields in object-oriented programming - each slot has a name and one or more values. Currently, only string values are supported.

Figure 1 illustrates how agents interact through the blackboard. In this example, a user is requesting a listing of his emails. The audio input device, in this case a phone, posts an audio object to the blackboard. The speech recognition agent has registered its interest in audio objects, so it is notified of the arrival of the new object. The speech recognition system reads and processes the audio object, posting its output, a text object, back to the blackboard.

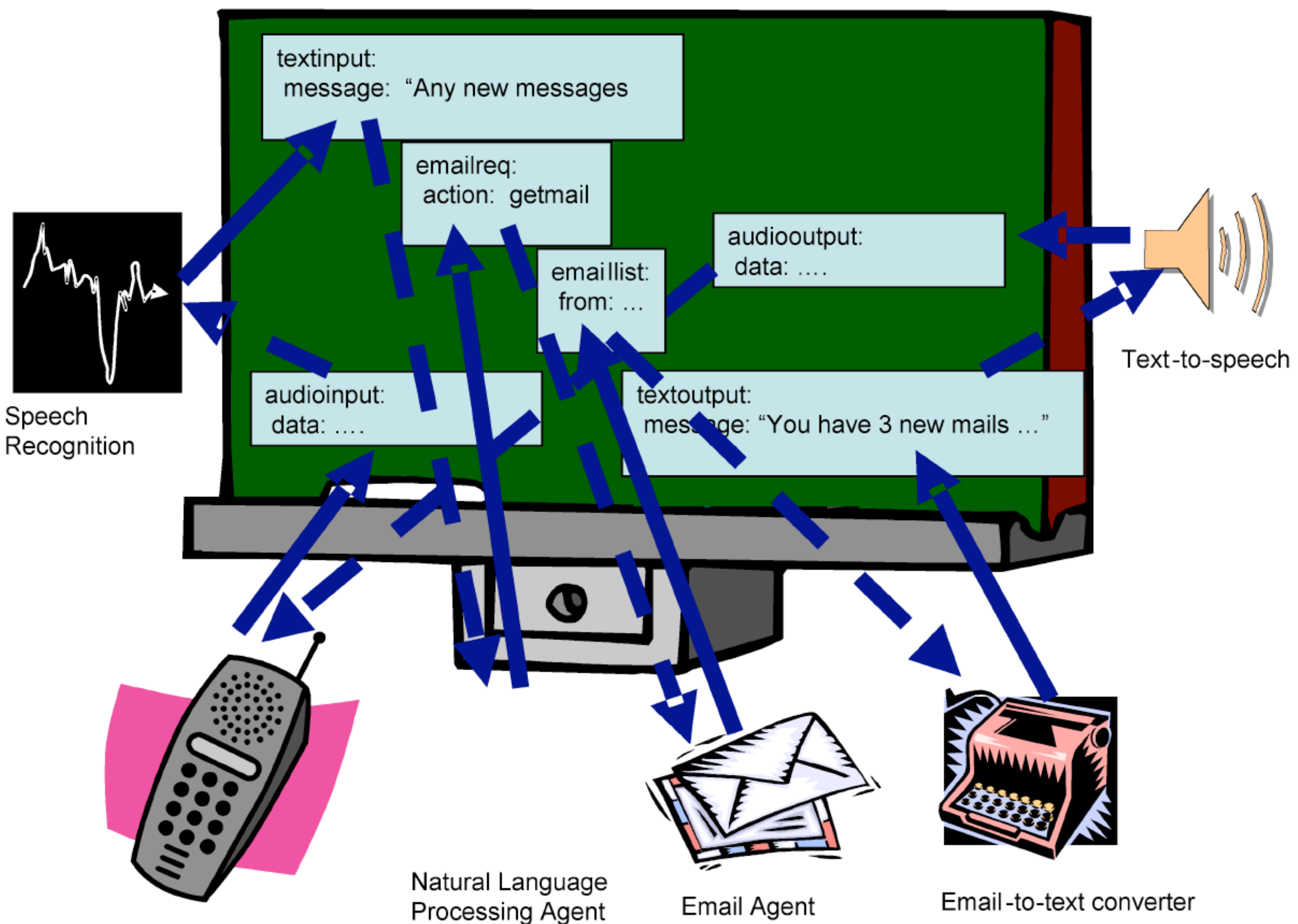

Figure 1: Agents communicate via MICA’s blackboard

The text object activates the natural language agent. This is a FrameScript program that understands email requests and translates the text into an object that represents a command to the email agent. The result of the email request goes onto the blackboard and is translated into text by the email-to-text converter. Finally, the text-to-speech agent creates the audio output that is sent to the output device.

1.2 FrameScript

Some of the agents that connect to MICA may be generic. For example, a speech recognition system such as Dragon Naturally Speaking (DNS) is an off-the-shelf program that only requires the addition of a stub that allows it to be invoked by the communication of objects through MICA. Other agents will be application specific. An example is a conversational agent that handles dialogues specific to the application, such as an in-car controller for non-critical devices such as the radio or air-conditioner.

FrameScript is a language that allows developers to rapidly script domain specific interactions for particular application. MICA and the FrameScript interpreter are separate programs that may be run completely independently. However, their usefulness is greatly enhanced when used together. Application dependent agents can be written using FrameScript and these connect through MICA to invoke other agents, such as DNS or to control devices. The FrameScript language provides a very rich set of tools for representing knowledge and interacting with users and external devices. Its strength is that new agents can be created quickly and easily. In this section, we briefly survey only the most important features of FrameScript. Later sections describe the language in detail.

Scripts provide FrameScript with rule-based processing for controlling interactions with users and devices. The rule system in FrameScript is derived from Sammut's Probot [7]. The basic unit of a FrameScript program is the pattern-response rule:

pattern ==> response

In stand-alone mode, rules are used to match input from the user. When coupled with MICA, the left-hand side of a rule contains a pattern than may match objects on the blackboard. Thus, if another agent posts a text object, a FrameScript agent that has registered interest in text objects can read the object and perform natural language processing to produce a response. Patterns can also match arbitrary objects. For example, a pointing gesture may be recorded by a location and direction in space, as well as a time stamp. A FrameScript agent that has registered interest in these kinds of objects will read the gesture information and match that against its rules to produce a response. Patterns can also be mixed, allowing multi-modal interactions.

We usually want the application of rules to be confined to particular contexts. For example, if the user says to an in-car control system, "turn it up", this can mean different things depending on whether the user had previously said, "turn on the radio" or "turn on the air-conditioner". Contexts are handled by grouping rules into different scripts. For example:

```
radio_script ::
```

```
trigger {* radio *}

* turn * up * ==>
    [I'll turn the radio up a little. #radio_volume(+10)]
        ......
```

The trigger is a pattern that is used to determine if the user had changed the context of the conversation. If the previous topic ha been the air-conditioner and the user now mentions the radio, this trigger will cause a context switch. The "*" appearing in the patterns is a wild card match. When the user says, "turn it up", in the radio context, the rule above is triggered and the system invokes an action from the radio (the '#' symbol precedes an action) and informs the user that the volume has been turned up. The radio_volume function posts to the blackboard an object that is intended for the radio controller agent. In this way, a FrameScript agent can act as a device controller or intermediary to external information sources.

When writing complex scripts that have similar behaviours, it is possible to use inheritance to enable rules to be shared between scripts. A script can also inherit variables that store information about the state of the interaction.

While the script is the main construct that is used to define interactions, there is a complete programming language underlying the script mechanism. Knowledge structures, called "frames", provide an object-oriented programming paradigm. Frames were originally developed as a knowledge representation system in Artificial Intelligence. Simple scripts will simply be a collection of rules like the ones above. Interactions that require more complex behaviour may use the full power of the frame representation system. For simplicity, we have also only described very simple rules. However, the rule language can describe complex grammars.

The remainder of this document describes the details of the language.

2 FrameScript

FrameScript is a scripting language that combines a rule-based system with a frame implementation for data storage and retrieval, both of which are built upon an expression language.

2.1 Frames

Frames[6] provide a way of representing and manipulating complex objects. There are two main features of frame systems:

- **Procedural Attachment**: Inference in a frame system is performed on an ad hoc basis by attaching procedures to the attributes of an object. These procedures are triggered depending on how the attribute is accessed.
- **Inheritance**: Like all all object-oriented systems, instance frames can inherit properties from generic frames in a hierarchy. In FrameScript frames can inherit from multiple generic frames.

2.1.1 Slots

Slots are the attributes of a frame. They hold the attributes of instances frames and provide points for attaching procedures to the attributes in generic frames through daemons.

Slots can be defined as being **multivalued** and **cache**. Multivalued slots can hold multiple values. Cached slots store the value for an instance's slot when the slot's value is first computed in either an `if_new` or `if_needed` daemon, otherwise the slot's value is recalculated every time is is accessed.

2.1.2 Generic Frames

Generic frames define values and behaviours that are applicable to all instances of the type, as such they act like classes in the standard OOP paradigm.

```
person ako object with
    name :
        if_needed "Anonymous"

    age :
        if_new 0
        cache true
;;
```

Figure 2: Example Generic Frame

2.1.3 Daemons

Daemons are procedures that are attached to slots in generic frames. They are executed when a frame's slot is accessed in a given way. FrameScript currently allows 9 daemons to be attached to a slot. These daemons are:

- **if_added** : The if_added daemon is run whenever a value is placed in a slot.
- **if_destroyed** : The if_destroyed daemon is executed when an instance frame is destroyed.
- **if_needed** : The if_needed daemon computes the value of a slot if the value isn't currently known when it is accessed.
- **if_new** : The if_new daemon is run when the instance frame is first created to give the slot an initial value.
- **if_removed** : The if_removed daemon is run whenever a value is removed from a frame.
- **if_replaced** : The if_replaced daemon is executed when the value of a slot is replaced with a new value.
- **range** : The range daemon is used to restrict the values allowed in a slot. It is used every time a slot's value is added or replaced.

- **help** : The help daemon is run whenever an attempt to add a value to a slot that is disallowed by the range daemon.
- **default** : The default isn't really a daemon but a non-evaluated default value used in the GUI for viewing/manipulating frames.

2.1.4 Instance Frames

Instance frames describe instances of generic frames, as such they are like objects in OOP. When a slot cannot be found within an instance frame FrameScript will search through the inheritance hierarchy for a procedure to compute the appropriate value for the slot.

```
john isa person with
    name : "John Doe"

    age : 35
;;
```

Figure 3: Example Instance Frame

Defining Slots
There are four ways of giving an instance frame a value for a slot. They are:

- Define the slot as cached and provide an `if_new` daemon.
- Define an `if_needed` daemon for a generic frame from which the instance inherits. This defines the slot for all instances of the type.
- Define a value for the slot in the instance's declaration. (eg definition of `name` in Figure 3)
- Put a value in the slot after it is instantiated.

Adding Values to Slots
There are two ways to put a value in a slot for an instance. They are:

- `put(instance, slot, value)` (eg. `put(john, age, 7)`)
- `slot of instance = value` (eg. `age of john = 7`)

However be aware that both of these methods with fail if there is already a value in the slot and it isn't multivalued.

2.1.5 Multiple Inheritance

FrameScript allows multiple inheritance, i.e. a frame may inherit attributes from more than one generic frame. When FrameScript looks up a value for an item the inheritance hierarchy is searched in a depth-first, left-to-right manner.

```
simon isa person, rabbit with
    name: Simon

    age: 4

    fur: brown
;;
```

Figure 4: Multiple Inheritance Example

```
greeting ::
    domain example

    init ==>
        [ hello ]

    { hello | hi } i'm * ==>
        [ hi ^1 ]

    { hello | hi } ==>
        [ hi, how are you? ]

    { goodbye | bye } ==>
        { bye | see you soon }
;;
```

Figure 5: Script Example

2.2 Scripts

Scripts are used to provide FrameScript with rule-based processing for control of conversational interactions. The rule system used in FrameScript is heavily influenced by the system used in ProBot[7]. Each script contains a list of rules that are matched against a user's input and used to determine the appropriate response. In addition to their rules scripts can also make use of inheritance and slots to allow reuse of scripts and retaining contextual information.

Because rules are used in a first come first served basis more specific rules should be placed before more generic rules in a script.

2.2.1 Rules

The rules in FrameScript consist of patterns and responses, where each pattern matches a user's input and the response determines the system's output. When a pattern is found that matches the user's input the associated response is returned. Rules are of the form:

```
    pattern ==> response
```

2.2.2 Patterns

Patterns can be written to match a variety of sentences. The patterns used in FrameScript are similar to context free grammars.[1]

Wild-cards

FrameScript allows the use of 2 wild-cards characters in patterns. They are `*` and `~`. `*` will match to 0 or more words/terms while `~` will match precisely 1 word/term. In addition `~` can be embedded into a word to match 0 or more characters.

Alternatives

Alternatives may be used if more than one word can be accepted at a point in a pattern (eg. synonyms). Alternatives are constructed with the form:

```
    { alternative 1 | alternative 2 }
```

Non-terminals

Non-terminals provide a way for commonly used patterns to be reused. To use a non-terminal write the name of the non-terminal surrounded by '<' and '>'. (eg. `I am < number > years old`)

While it is technically possible for any script to be a non-terminal they are often declared as a list of alternatives followed by ;;.
Example:

```
number ::
    {1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9};;
```

[1] FrameScript uses a greedy matching algorithm so some patterns may not work as expected.

 # and ˆ
Some patterns may be context sensitive so FrameScript allows conditions to be included in patterns. There are two types of conditions that FrameScript allows '#' and 'ˆ'. '#' conditions are evaluated and if they don't evaluate to true the pattern doesn't match. 'ˆ' conditions are evaluated and the result of the evaluation is matched against the current point of the input.

```
greeting ::
    #(have_met_user_before) hello ==>
        [hello, its good to see you again]
    #(not have_met_user_before) hello ==>
        [hi, its good to meet new people]
;;
```

Figure 6: Rule Condition Example

One interesting application of the 'ˆ' condition test is that if the condition evaluates to a generic frame then any instance frame that inherits from the given generic frame will match. This allows scripts to be written that perform actions in response to events such as use of a touch-pad or arrival of an email as long as the event causes an appropriate instance frame to be created and used as an input.

```
email ako object with
    from :
        if_needed "Anonymous"
;;

new_email ::
    ^email ==>
        [ you have a new email from
          ^(from of ^1) ]
;;
```

Figure 7: Rule Evaluation Condition Example

2.2.3 Responses

Responses tell the system how it is to respond to a user's input.

Sequences
To avoid writing scripts that users may find repetitive responses can include a sequence of responses where each response is given in turn every time the pattern is matched. After the last response is given it returns to the first response in the sequence.

Alternatives
Alternatives are similar to sequences except instead of responding with the

```
* ==>
    [ first response
    | second response
    | third response ]
```

Figure 8: Sequence Example

next response in the sequence the response is selected randomly each time from the list of all responses.

```
* ==>
    { a response
    | another response
    | yet another response }
```

Figure 9: Alternatives Example

#

Sometimes a response requires that the system do something not just say something. If a '#' is found in the response the following expression is evaluated and the result discarded. This allows a response to alter the system's state while not burdening the user with unnecessary dialog. A common use of '#' is `#goto(a_script)` which moves a conversation/interaction from one context to another.

^

In a response '^' is similar to '#' except that when the following expression is evaluated it is inserted into the response not thrown away. The other difference is that if '^' is followed by an integer then the numbered pattern component(see 2.2.4) associated with that integer is placed in the response.

Conditional Responses

Some responses may be dependent on some condition holding true. Conditional responses can be constructed in the form:

```
* ==>
    [ ^(condition) ->
        response if true
    | ^(! condition) ->
        response if false ]
```

Figure 10: Conditional Response Example

2.2.4 Pattern Component Numbering

Some pattern elements create a numbered match component when a pattern matches. These components are segments of the input (possibly transformed)

```
hello ::
    * i'm <number> years old ==>
        [ really? ^2 is really old ]

    i'm * ==>
        [ hi, ^1 ]

    * ==>
        [ what do you mean by ^0 ? ]
;;
```

Figure 11: Match Component Example

that can be referred to in a response using '^'. Pattern elements that produce match components are wild-cards (*, ~), alternatives, non-terminals, and possibly '^' conditions depending on the evaluation result (eg. generic frames do).

Component Numbering Example

Table 1 shows the match components generated for a number of inputs when some inputs are given. These examples use some non-terminals which are given in figure 12.

```
known_languages ::
    { english | french | german | italian | spanish };;

number ::
    { 1 | one } ==> 1
    { 2 | two } ==> 2
    { 3 | three } ==> 3
    { 4 | four } ==> 4
    { 5 | five } ==> 5
    { 6 | six } ==> 6
    { 7 | seven } ==> 7
    { 8 | eight } ==> 8
    { 9 | nine } ==> 9
    { 0 | zero } ==> 0;;
```

Figure 12: Component Numbering Example Non-terminals

2.2.5 Domains, Topics and Triggers

When an input is received from a user it is given to a *domain*. The domain is then responsible for ensuring the input is matched against the correct scripts. It is the domain that keeps track of which scripts are currently active.

Scripts can be registered as topics in a domain. By registering a script as a topic that script is telling the domain that no matter what the current script is when certain inputs are received the topic is to become the current script and

<table>
<tr><th>Pattern</th><th>Input</th><th>Match Components</th></tr>
<tr><td rowspan="2">translate * in~
<known_languages></td><td>translate i love you
into french</td><td>^0: translate i love you into french
^1: i love you
^2: french</td></tr>
<tr><td>translate i am in pain
in german in french</td><td>^0: translate i am in pain in german
^1: i am in pain
^2: german</td></tr>
<tr><td>* sky *
{red | blue | green}</td><td>the sky is blue</td><td>^0: the sky is blue
^1: the
^2: is
^3: blue</td></tr>
<tr><td rowspan="2">* {is | are}
{a {car | truck} |
<number> {cars | trucks}}</td><td>there is a car</td><td>^0: there is a car
^1: there
^2: is
^3: car
^4: a car</td></tr>
<tr><td>there are four trucks</td><td>^0: there are four trucks
^1: there
^2: are
^3: 4
^4: trucks
^5: four trucks</td></tr>
</table>

Table 1: Component Numbering Example

```
foul_language_filter ::
    domain example
    trigger { * <swear_word> * }

    * <swear_word> * ==>
        [ swearing is not tolerated ]
;;
```

Figure 13: Topic Example

process the input. When a script is registered as a topic the domain uses the script's trigger to determine whether or not an input activates that topic. If a topic doesn't have a trigger any input will activate it. Triggers can be inherited.

Domains keep track of the current script/context and a history of the triggered topics. When a topic's trigger matches the input it becomes the current context and the current topic.

2.2.6 Inheritance

```
person ::
    age: 0

    who are you ==>
        [ i don't know who i am]
    how old are you ==>
        [ i am ^age years old ]
;;

john ::
    inherits person
    name: << john smith >>
    age: 4

    who are you ==>
        [ i am john ]
;;

peter ::
    inherits person
    name: << peter piper >>

    how old are you ==>
        [ i was born ^age years ago ]
;;
```

Figure 14: Script Inheritance Example

When writing complex scripts where scripts have similar behaviours it is

possible to use inheritance to enable rules to be shared between scripts. Not just rules are inherited, a script also inherits slots and daemons from it's parents as well.

The inheriting script is able to define new responses to patterns located in the inherited script if needed and new values for slots.

2.2.7 Slots

Scripts can have slots defined for them. This allows scripts to store and manipulate information about the state of the script and the state of the conversation. In most respects slots in scripts behave in much the same way as slots in instance frames. In fact many of the subroutines for accessing and manipulating slots such as `put`, `remove` and `replace` will work with scripts as well.

The main difference between slots in scripts and slots in instance frames is that slots in scripts don't have daemons attached to them. This is because scripts inherit from scripts not from generic frames. However scripts do inherit slots from their parents. Effectively the means that a slot in a script's parents acts like an `if_needed` daemon except it is just returned not evaluated.

In Figure 14 although `peter` does not have a defined age he inherits age from `person` so in response to the input "how old are you" `peter` would return "i was born 0 years ago".

2.2.8 Failsafes

It is possible to define failsafes for scripts. A failsafe is another script whose rules should be used if an input fails to match any of the rules for a script. Failsafes may be inherited.

Domains can also be given a failsafe by using the `failsafe` subroutine. Domain failsafes will only be used if an input fails to match any of the rules for the current context, the current topic and their associated failsafes.

Failsafes can be chained together. This means that when failsafes are being checked against an input and a failsafe(and its parents) doesn't match against the input the failsafe's failsafe will be tried.

2.2.9 Rule Ordering

When an input is received by a domain that domain is responsible for deciding which rules the input should be matched against. The order in which domains attempt to find a match is:

1. triggers of the topics (excluding the current topic)
2. the current context
3. the failsafe of the current context
4. the current topic
5. the failsafe of the current topic
6. the failsafe for the domain

When an input is compared to the rules of a script the input is first compared to the rules specifically defined by the script. If none of these rules match the input is matched against the rules of the script's parents. The rules of the scripts are tried in top to bottom order.

2.2.10 Daemons

FrameScript allows daemons to be defined that are executed when moving from one context to another. These are the `on_entry` and `on_exit` daemons. These daemons are attached to scripts and are evaluated when the current context is changed. The `on_entry` daemon for a script is run whenever the script becomes the current context. The `on_exit` daemon is run when a script stops being the current context. The `on_entry` and `on_exit` daemons run whenever the current context changes, this could be because of a trigger firing, a `goto` command or a `previous_topic` command.

Daemons are inherited.

```
normal_context ::
    dictate ==>  #goto(dictation_script)
    * ==> #do_command()
;;

dictation_context ::
    on_entry set_speech_rec_dictate(true)
    on_exit  set_speech_rec_dictate(false)

    finished ==> [ #use_saved_text() #goto(normal_context) ]
    * ==> #save_text(^0)
;;
```

Figure 15: Script Daemons Example

Figure 15 gives a small example that uses the `on_entry` and `on_exit` daemons. With this example the environment is that we have a speech recogniser that can either use rule grammars or use dictation mode. Usually the recogniser would use the rule grammars but on occasion it may need to use dictation. In the example the current context would be `normal_context` which would respond to the recognised speech from the rule grammars. But when the user says "dictate" the current context switches to `dictation_context` and its `on_entry` daemon is run which tells the recogniser to switch to dictation mode. Then all of the user's speech would be saved until the user says "finished" at which point the saved text is used for whatever purpose and the current context switches back to `normal_context`. In the course of switching back to `normal_context` `dictation_context`'s `on_exit` daemon is run which switches the speech recogniser back to rule grammar mode.

NOTE: If the `on_entry` or `on_exit` daemon is followed by a pattern that begins with `*` or any other operator then the pattern should be preceded by `_` so that the parser doesn't confuse it with a multiplication symbol. Using `_` just says that at the start of the pattern there is an empty space so in effect it does nothing.

2.2.11 Abnormal Scripts

It is possible to define the rules for a script using the expression language used in defining daemons, but for most applications it is simpler to use the standard rule-response syntax. Using the expression language however allows more complex rules to be written. For instance, using the expression language it is possible to create a script that responds with the instance frame whose slot matches a user's input.

```
contact_lookup ::
--
    var rval;
    forall C in instances_of(contacts) :
        if undefined(rval) and match(<< ^(name of C) >>) then
            rval = C;
    rval
;;

send_mail::
    Send the email to <contact_lookup>  ==>
        [#send_email(email of ^1, the_email)]
;;
```

Figure 16: Abnormal Script Example

Abnormal Script Example

Figure 16 gives an example of an abnormal script `contact_lookup` that loops through the list of all contacts to find a contact whose name matches the input. Any matching contact found is then returned.

Then if `send_mail` is the current context and the user says "Send the email to John" the rule will match as long as there exists a contact with the name John. It will then send the email to John.

2.2.12 Pattern Matching Algorithm

By default FrameScript uses a greedy matching algorithm without backtracking which means some patterns may not work as expected. For many patterns that do not work as expressed it is possible to rewrite the pattern so that it will work. Table 2 shows some examples of patterns that do and don't work.

A second fully back-tracked matching algorithm has been implemented for those applications that require it but it needs to be activated.

NOTE: Due to the greedy nature of the algorithm there is an implicit * at the end of every pattern (except when used as a non-terminal). Currently there is no known way to get around this situation.

NOTE: * inside alternatives should be avoided where possible, especially at the end of an alternative.

[2] `^1` is `c` not `c b c`

<table>
<tr><th>Pattern</th><th>Input</th><th>Pattern Match</th><th>Algorithm Match</th></tr>
<tr><td>a * b</td><td>a c b</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>a b c</td><td>No</td><td>Yes</td></tr>
<tr><td></td><td>a c b c b</td><td>Yes</td><td>Yes[2]</td></tr>
<tr><td>{ a * b * } #undefined(^2)</td><td>a c b</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>a b c</td><td>No</td><td>No</td></tr>
<tr><td></td><td>a c b c b</td><td>Yes</td><td>No</td></tr>
<tr><td>{ a | b * } c</td><td>a c</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>b c</td><td>No</td><td>Yes</td></tr>
<tr><td>{ a c | b * c}</td><td>a c</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>b c</td><td>Yes</td><td>Yes</td></tr>
<tr><td>{ _ | a } b</td><td>a b</td><td>Yes</td><td>No</td></tr>
<tr><td></td><td>b</td><td>Yes</td><td>Yes</td></tr>
<tr><td>{ a | _ } b</td><td>a b</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>b</td><td>Yes</td><td>Yes</td></tr>
<tr><td>a~b</td><td>ab</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>acb</td><td>Yes</td><td>Yes</td></tr>
<tr><td></td><td>abb</td><td>Yes</td><td>No</td></tr>
</table>

Table 2: Pattern Match Examples

2.3 Expression Language

In-order to attach procedures to objects a language must be defined in which those procedures can be written. FrameScript uses a fairly simple untyped procedural language that includes some programmatical elements inspired by machine learning techniques. FrameScript's untyped nature means that when we declare a variable we don't declare what type it is. The variable can can take any type at anytime, it is only when we come to use the variable's value that the type is taken into consideration.

2.3.1 Statements

FrameScript code is written as a sequence of statements that are evaluated in turn. There are 2 types of statements currently available in FrameScript; definition statements and expression statements. Definition statements are used to declare and define generic frames, instance frames and scripts. Evaluation statements are used to define an expression that needs to be evaluated. All statements whether definition or evaluation end with `;;`.

Figures 2, 3 and 5 are examples of definition statements, while some expression statements can be found in figure 17.

```
1 + 2;;

"Hello world.";;

print("Hello world,");;

head([ 1 2 3 ]);;

foo(bar) = bar + 1;;

foo(2);;

a = 1;
b = 2;
a * b;;
```

Figure 17: Evaluation Statements

2.3.2 Numbers

Numbers in FrameScript can be either integers or real numbers. When performing mathematical operations the numerical type returned depends on the product of the operation not on the types of the operands. (eg. `3 / 2 = 1.5`, `1.5 + 2.5 = 4`)

2.3.3 Strings

A string is just a sequence of characters enclosed between `"`s. (eg. `"the grass is green"`, `"roses are red"`)

2.3.4 Atoms

Atoms are a fairly simple type that provide FrameScript's expression language with a lot of flexibility and possible complexity. Atoms are essentially sequences of non-whitespace characters. There are some restriction on the characters that comprise an atom imposed by the parser, such as atoms cannot begin with any of the characters '0'-'9' and some characters such as '+' refuse to associate with others.

Atoms are used as the names of objects, such as frames or scripts,; the names of slots; the names of variables and a values in their own right. When an expression evaluates an atom to get its value. The atom's value is evaluated in a rather specific order. First if the atom is the name of a declared variable the value of the variable is used, second if the expression being evaluated is part of a daemon attached to a frame and the frame has a slot whose name is the atom the value of the slot is used, third if the atom is then name of an object the object is used, otherwise the atom itself is used as the value.

Some examples of atoms are: `+`, `red`, `hello`, `X`, `X23`, `my_frame`.

2.3.5 Lists

Lists are a compound data type that allow collections of terms to be grouped. Lists can contain any type of term including numbers, atoms, lists, frames. When a list is evaluated the result is a list whose elements are the evaluation results of the corresponding element in the original list. (eg `[ 1 + 1 ]` evaluates to `[ 2 ]`)

Some examples of lists are: `[]`, `[ 1 ]`, `[ a ]`, `[ 1 2 3 ]`, `[ a 1 ]`, `[ 1 + 1 ]`.

2.3.6 Expression Lists

Expression lists are a sequence of terms which are evaluated sequentially. The expressions in the list are separated by `;`.

The resultant value when evaluating an expression list is the result of the last expression in the list being evaluated. (eg `1;2;3` evaluates to `3`)

The following expression list shows the sequential nature of evaluating an expression list. When it is evaluated the first expression, `a = 1`, is performed setting the valued of `a` to 1. Then the second expression, `a = a + 1`, is evaluated which increments the value of `a` by 1 which in this case makes `a` 2. Then the third/final expression, `a`, is evaluated which retrieves the value of `a`, this final value then is the result of evaluating the list.

```
a = 1;
a = a + 1;
a
```

When writing expression lists common practice should entail enclosing the entire list in brackets. This is because by enclosing the list in bracket the expression list is being designated as a single expression. (eg `(a = 1; b = a + 1; b + 2)`)

2.3.7 Patterns

Patterns are a data type used primarily in defining and processing scripts. But they can be used virtually anywhere within FrameScript. Patterns are a sort

of hybrid type that results when a string crossbreeds with a list. A pattern is a sequence of zero or more pattern elements enclosed by `<<` and `>>`. (eg `<< >>`, `<< hello world. >>`) When writing scripts the `<<` and `>>` are implied for readability.

When a pattern is evaluated each pattern element is evaluated sequentially an a resultant pattern is returned. As each element in the pattern is evaluated it is first checked whether or not it is an atom. If it is an atom it is appended to the resultant pattern as is without evaluation, otherwise it is evaluated. If the evaluated value is nothing the value nothing is done with it, if it is a pattern the evaluated pattern is appended to the end of the resulting pattern if it is any other type of term it is simply appended to the end of the pattern.

Pattern Elements

A pattern element can be any type of term but when a pattern is parsed only specific term types are parsed. The types read in by the parser are alternatives, sequences, atoms, strings, numbers, ^commands and # commands. All other types can be found as pattern elements however as a result of evaluation.

^and # Commands

When parsing pattern elements ^and # are used as escape characters to allow expressions to be embedded within patterns. The # character indicates that when the pattern is evaluated the following expression should be evaluated but its result should not be returned in the result of evaluating the pattern. The ^character however says the following expression should be evaluated and inserted into the result of evaluating the pattern. The ^command however has one exception, when the expression following ^is an integer then it gets the associated match component for the pattern to a rule. This is fine when the pattern being evaluated is part of the response to that rule, but at other times it will result in an error.

Pattern Examples

The best way to demonstrate patterns is with some examples. See Table 3.

Example	Evaluated Result
`<< a b c >>`	`<< a b c >>`
`<< 1 + 2 >>`	`<< 1 + 2>>`
`<< ^(1 + 2) >>`	`<< 3 >>`
`<< #(1 + 2) >>`	`<< >>`
`a = 1; << a >>`	`<< a >>`
`a = 1; << ^a >>`	`<< 1 >>`

Table 3: Pattern Examples

2.3.8 Sequences

Sequences are a list of patterns. When a sequence is evaluated the result is that one of the patterns in the list is evaluated and returned. Each subsequent evaluation of the sequence will result in the evaluation of the next pattern

in the list. When the last pattern is evaluated the sequence returns to the beginning. Sequences are found within patterns and are enclosed by `[` and `]`. The patterns within the sequence are separated by `|`. (eg `[ a sequence ]`, `[ a pattern | another pattern ]`)

The patterns within a sequence can have conditions. When the sequence tries to evaluate a pattern and the pattern has a condition the condition is tested. If it is true then the pattern is evaluated, if not the sequence tries the next pattern in the list. The sequence will continue trying then next pattern in the list until it finds one with no conditions or one whose condition holds true.

Example:

```
[ ^(condition) -> the condition is true |
^(not condition) -> the condition is false ]
```

2.3.9 Alternatives

Alternatives are a list of patterns. When an alternative is evaluated the result is that a randomly selected pattern in the list is evaluated and returned. Alternatives are found within patterns and are enclosed by `{` and `}`. The patterns within the sequence are separated by `|`. (eg `{ an alternative }`, `{ a pattern | another pattern }`)

2.3.10 Ripple Down Rules

Ripple Down Rules(RDRs) [2, 1] are used in FrameScript to provide conditional branching evaluation paths. They also provide a basic mechanism for knowledge acquisition.

A very simple RDR will look like `if condition then conclusion`. This RDR will evaluate and return the conclusion if the condition evaluates to true otherwise it returns nothing.

An RDR can optionally have a cornerstone case, an exception or an alternative. An RDR with all 3 will look like:

```
if condition then
    conclusion
because cornerstone_case
except exception
else
    alternative
```

The cornerstone case is the name of the frame that represents the case that caused the rule to be constructed. The exception is a rule constructed to handle cases where the condition is true but due to other conditions require a different conclusion. The alternative is an expression to be evaluated and returned if the condition is false.

NOTE: If the condition/conclusion/alternative of an RDR is an expression list it needs to be bracketed.

2.3.11 Functions

Functions are used in many programming languages to encapsulate computations so that programs can be written regardless of the implementation used

to perform the computation and the allow for the code to be reused relatively simply.

Figure 18 shows the definition of a simple function that finds the maximum of 2 numbers.

```
max(X, Y) = (
    if X > Y then
        X
    else
        Y
);;
```

Figure 18: Function Definition Example

As FrameScript is an untyped language the arguments to the function can be of any type. This means that if a function requires specific types for its parameters it must either explicitly check the types of its arguments or assume the types that the arguments posses. Figure 19 shows the same maximising function as Figure 18 except it explicitly checks that the parameters for `X` and `Y` are numbers and if either isn't a number it throws an error.

```
max(X, Y) = (
    if number(X) then
        if number(Y) then
            if X > Y then
                X
            else
                Y
        else
            error("Y is not a number")
    else
        error("X is not a number")
);;
```

Figure 19: Function Type Checking Example

FrameScript is implemented in Java and for some functions it may more efficient/necessary for them to be written in Java rather than FrameScript. Subroutines are functions that can be called like any other function from within FrameScript but are instead written in Java. Detailed information about writing subroutines can be found in section 2.4.1.

NOTE: If the body of a function is an expression list it needs to be bracketed.

Functions in FrameScript are defined by giving the function specification and assigning it an expression that is to be evaluated whenever the function is called. The most common way of defining a function is as a single expression in a statement. The code that defines the function is treated by frame script as any other expression so it can be placed in the body of a daemon or another function.

2.3.12 Forall Loops

When using lists it is not uncommon to need to perform some evaluation for each element in the list. For this reason FrameScript provides the `forall` loop that will loop through the elements of a list evaluating a given expression for each element. The `forall` loop looks like:

```
forall VARIABLE in LIST : EXPRESSION
```

The result of evaluating the `forall` loop is a list where each element is the result of evaluating the given expression for the corresponding element in the list argument of the `forall` loop. For example the result of evaluating `forall X in [ 1 2 3 ] : X + 1` is `[ 2 3 4 ]`.

It is possible to nest `forall` loop. The result of evaluating nested `forall` loops will be a list of lists of ...

NOTE: If the body of a `forall` loop is an expression list it needs to be bracketed.

2.3.13 Variables

When entering the body of a function or the body of a `forall` loop FrameScript will define and assign the value to a local variable using the parameter name in the function specification or `forall` loop. On occasion it may be necessary to define other local variables. This can be done using the `var`.

A `var` statement can be used in the body of either a function, a `forall` loop or a daemon to declare local variables. The use of a `var` statement necessitates that the body be an expression list where the `var` statement is the first expression in the list. If a `var` statement is not part of an expression list or not the first expression of the list its evaluation will result in an error.

A `var` statement is `var` followed by a non-zero comma separated list of variable names. The variable names must be atoms.

Some valid `var` statements are: `var X`, `var X, Y`.

When the body of the function/`forall` loop/daemon is finished being evaluated the memory used for storing the values for the local variables is freed and the values lost.

A simple example using local variables is:

```
example(X, Y) = (
    var Z;
    Z = X mod Y;
    Y mod Z
);;
```

When FrameScript is evaluating a daemon is declares some common variables that may be used in the body of the daemon. These variables are:

`current_object` This is the instance frame/script which caused the daemon to be run.

`current_slot` This is the name of the slot which the daemon is assigned to.

`new_value` When a value is being added to a slot, this is the value being added. In the `on_entry` and `on_exit` daemons this is the context being switched to.

`old_value` When a value is being added to a slot this is the previous value of the slot. When a value is being removed from a slot this is the value being removed. In the `on_entry` and `on_exit` daemons this is the context being switched from.

2.4 Modules

Modules a a way in which FrameScript can be extended to make it more flexible. Modules can be loaded into FrameScript using the `load_module` function. A standard module provided within the FrameScript distribution is `sitcrc.framescript.GUI` which provides some functions for displaying GUIs for communicating with a user. Details on the *sitcrc.framescript.GUI* module can be found in section 4.

Implementationally speaking a module is simply a Java class with a **public static void** `init()` method. When the module is loaded the class is loaded into memory and the init()— method is executed.

Commonly found within the body of the `init()` methods of modules are subroutine declarations. (see 2.4.1)

An example of a module is given in section 2.4.2.

2.4.1 Subroutines

Subroutines are FrameScript functions whose implementations have been written in Java rather than FrameScript. The most common way of adding subroutines to FrameScript is by creating a module that declares and defines the subroutines desired.

To create a new subroutine in FrameScript an instance of the `sitcrc.framescript.Subr` class needs to be created for each subroutine being added. The class `Subr` is abstract so it cannot be instantiated directly but must first be extended. A concrete class that extends `Subr` needs an implementation of the `public Term apply(Instance, Term[], StackFrame) throws FSError` method. Probably the simplest way of implementing a subroutine is in the `init()` method of a module instantiating an anonymous class for each subroutine being created. Each anonymous class would then implement the subroutine in its definition of the `apply()` method. The MaxModule example(see 2.4.2) is written in this manner.

Constructors

The `sitcrc.framescript.Subr` class defines 2 constructors that can be used in instantiating a subroutine. They are:

- `public Subr(String str)`
- `public Subr(String str, int nArgs) throws FSError`

The code that links the subroutine into FrameScript is located inside these constructors so one of them needs to be used when creating the subroutine. Both constructors take a `String` argument, this is the name of the function being created. If the `my_function()` subroutine were being implemented the argument should be `"my_function"`. Optionally the number of arguments the function expects can be given to the constructor, FrameScript will then ensure that the subroutine is only run when the function is called with the expected

number of arguments. If no expected number of arguments is given(or it is less than 0) the subroutine will be run whenever the function is called regardless of the number of arguments provided by the function call. If, when a subroutine is instantiated with a given number of arguments, there already exists a subroutine with the same name that handles an undefined number of arguments an exception will be thrown.

If a subroutine can expect a variable number of arguments it is possible to instantiate a subroutine with the same name for each possible number of arguments or to instantiate a single subroutine that takes an undefined number of arguments and in the body of the subroutine check the number of arguments given.

apply() Method

When during the evaluation of a term FrameScript encounters a function call it looks up the appropriate function/subroutine to handle the call. If the appropriate handler is a subroutine FrameScript will call the `Term apply(Instance, Term[], StackFrame)` method defined for that subroutine.

The `apply()` method has 3 parameters. These are:

`Instance currentObject` If the function call being processed is in the body of a daemon this is the instance frame/script which caused the daemon to run, otherwise this is `null`.

`Term[] args` This is an array that holds the arguments being passed to the subroutine. The 1st value in the array is the name of the subroutine being called. The actual arguments begin at index 1. (eg if the subroutine is called with 2 arguments `args.length` will be 3)

`StackFrame frame` This is a reference to the memory in which the local variables are being stored.

The arguments in `args` are stored in their unevaluated form. To get the actual values for the arguments they need to be evaluated. To evaluate an argument you need to call the `eval()` method on the argument and to pass it `currentObject` and `frame`. (eg `args[1].eval(currentObject, frame)`)

For the vast majority of subroutines the `currentObject` and `frame` arguments will only be used in the evaluation of arguments and serve no other purpose.

If there are any problems encountered during the execution of a subroutine's `apply()` method the subroutine can elect to throw a `sitcrc.framescript.FSError` exception.

Argument Type Checking

If your subroutines requires its arguments of a specific type there are some argument type checking methods in the `sitcrc.framescript.Utils` class that can help. These methods were developed to check argument types and to provide standard error messages if the arguments are not of the correct type. If the argument is not the correct type an `FSError` exception is thrown. To make thing simpler and the error messages more detailed these argument checking methods will evaluate the arguments for you. If the subroutine `my_subroutine`

expected an integer as its 1st argument the `check_integer()` method could be used, the code would then look something like:

```
FSInteger i = Utils.check_integer("my_subroutine", currentObject, args, 1, frame);
```

There is a method in `sitcrc.framescript.Utils` to check the argument type for virtually all of possible types of term in FrameScript. More details on the argument type checking methods can be found in appendix C.

Returning Values

When returning values in the `apply()` method any subclass of `sitcrc.framescript.Term` can be returned. If you have a numerical value being returned then `FSNumber.getNumber()` can be used to create an appropriate FrameScript numerical type. If a subroutine isn't really designed to return a value it is preferred that it returns `sitcrc.framescript.Atom._null` rather than just `null` as it makes FrameScript less prone to unrecoverable crashes.

2.4.2 MaxModule Example

Figure 20 shows a fairly simple FrameScript module. All this module does is define 2 subroutines that can then be used in FrameScript code. These subroutines are `max()` and `max2()`.

The `max2()` subroutine expects 2 arguments. The first thing that it does is to check that both of the arguments are numbers. If either of the arguments isn't a number the `Utils.check_number` function will throw an exception. Then `max2` finds the largest number and returns it. The constructor used to define the `max2()` subroutine will throw an exception if, when the module is loaded, there already exists a `max2()` subroutine that take an undefined number of arguments. This exception can safely be thrown by the surrounding `init()` method.

The `max()` subroutine doesn't define how many arguments it expects in its declaration so it can be called with any number of arguments. The first thing it does is to find out how many arguments it has. If there are none it throws an exception. Otherwise it checks that the first argument is a number. Then it loops through all of its remaining arguments checking that they are numbers and keeping track of the largest seen so far. Once all arguments have been checked the largest is returned.

3 Multi-modal Interaction

One of FrameScript's strengths is its ability to use frames to represent events. The events that can be represented could range from system events such as receiving an email or a program finishing or a user's input in the form of a using a touch-pad or a making a recognised gesture. This allows FrameScript to be used not just for conversations but also for scripting multi-modal interactions between the system and users.

3.1 Multi-modal Example

A simple multi-modal script is given in Figure 21. The problem domain in which the example is set is a simple street mapping program that shows houses and roads. This example is built using a few assumptions. These assumptions are:

```
package sitcrc.framescript.examples;

import sitcrc.framescript.*;

public class MaxModule {
  public static void init() throws FSError {
    new Subr("max") {
      public Term apply(Instance currentObject, Term[] args,
          StackFrame frame) throws FSError {

        if (args.length - 1 < 1)
          throw new FSError("max - No arguments given.");
        FSNumber max = Utils.check_number("max", currentObject,
                                          args, 1, frame);

        for (int i = 2; i < args.length; i++) {
          FSNumber x = Utils.check_number("max", currentObject,
                                          args, i, frame);
          if (max.lt(x))
            max = x;
        }
        return max;
      }
    };

    new Subr("max2", 2) {
      public Term apply(Instance currentObject, Term[] args,
          StackFrame frame) throws FSError {

        FSNumber x = Utils.check_number("max2", currentObject,
                                        args, 1, frame);
        FSNumber y = Utils.check_number("max2", currentObject,
                                        args, 2, frame);

        if (x.le(y))
          return y;
        return x;
      }
    };
  }
}
```

Figure 20: MaxModule Example

```
demo ::
    domain example
    request: ""
    a_click: ""

    what is this ==>
    [
        #put(request, "describe")
        [ ^(a_click != "") ->
            #goto(demo, a_click)
        ]
    ]

    ^click ==>
    [
        #put(a_click, "")
        [
            ^(request == "") ->
                #put(a_click, ^1)
        |
            ^(request == "describe") ->
                #put(request, "")
                #goto(describe, get_object(click))
        ]
    ]
;;

describe ::
    ^house ==>
    [
        it is a house.
        #goto(demo)
    ]

    ^road ==>
    [
        it is a road.
        #goto(demo)
    ]

    * ==>
    [
        i don't know what it is.
        #goto(demo)
    ]
;;
```

Figure 21: Example Multi-modal Script

- That when the user clicks on an object a click event is generated.
- There exists a function `get_object()` that processes the click event to determine what it was the user selected.

This example expects two types of events. Either the user will say "what is this" or the user will click on something. All other speech and events are ignored.

If the user says "what is this" the `demo` script stores the user's request for a description. It then checks if the user has clicked on something. If the user has clicked on something then we reenter the script using the previous click as the input.

When a click event occurs preceding clicks are deleted and the script checks if the user has requested a description if they haven't the click is stored in the slot `a_click`. If the user has requested a description then the system changes context to the `describe` context and uses the object clicked on as the input. How the `get_object()` function determines what was clicked is irrelevant.

When the `describe` context encounters an input it checks if it is a type of house or road. If it is neither it claims ignorance. If the input is a type of house the script says it is a house, if it's a road it says it's a road. All of the rules in the `describe` context switch the context back to the `demo` context for the next user input.

4 GUI

The standard FrameScript distribution includes the module `sitcrc.framescript.GUI` which contains several subroutines that open GUIs that can be used to facilitate interactions with the user.

The subroutines currently included in the module are:

- `dialog_message()` - Opens a simple message dialog. See section 4.1.
- `dialog_question()` - Opens a simple input dialog. See section 4.2.
- `fix_rdr_gui()` - Opens a dialog for providing a description of the difference between two cases when maintaining an RDR. See section 4.3.
- `frame_browser()` - Opens a GUI for creating and manipulating frames. See section 4.4.
- `script_browser()` - Opens a GUI for creating and manipulating domains and scripts. See section 4.5.

4.1 Message Dialog

The subroutine `dialog_message()` expects a single argument. It will open a simple dialog window to display the given argument. If the argument is a string or a pattern the surrounding quotation characters will not be displayed. The result of calling `dialog_message("Hello world.")` is given in figure 22.

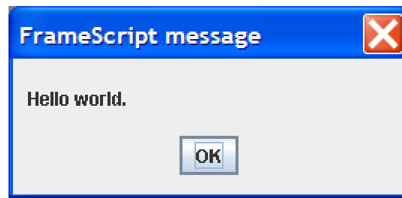

Figure 22: Message Dialog

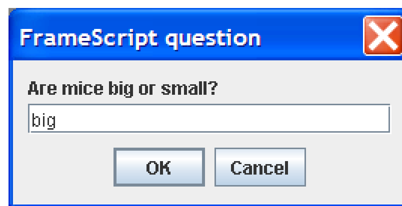

Figure 23: Question Dialog

4.2 Question Dialog

The subroutine `dialog_question()` expects a single argument. It will open a simple dialog window that will request an input from the user. If the argument is a string or a pattern the surrounding quotation characters will not be displayed. The result of calling `dialog_question("Are mice big or small?")` is given in figure 23.

The input dialog will accept simple expression from the user such as numbers, atoms, strings, list, patterns, function specs/calls, and will return the first such expression it finds in the input.

4.3 RDR Maintenance GUI

As RDR maintenance is a difficult task the `fix_rdr_gui()` subroutine was implemented to somewhat simplify the process. The `fix_rdr_gui()` subroutine is basically the same as the `fix_rdr()` subroutine except it uses a GUI to interact with the user rather than the console. Both subroutines are designed to be placed in the `if_replaced` daemon of a generic frame and to be called when an incorrect computed value of an instance frame is replaced by a correct value and both accept no arguments.

An example of the GUI opened by `fix_rdr_gui()` is shown in figure 24. This GUI is roughly divided into 4 components:

1. **Cases :** The case that was used in defining the rule that gave the incorrect value is given on the left. On the right is the new case whose value has been corrected. All slots and their values for each of the cases is given.

2. **Possible Conditions :** A number of possible conditions that could explain the differences between the 2 case are listed with check boxes. As these conditions a selected/deselected the condition to be used to for the

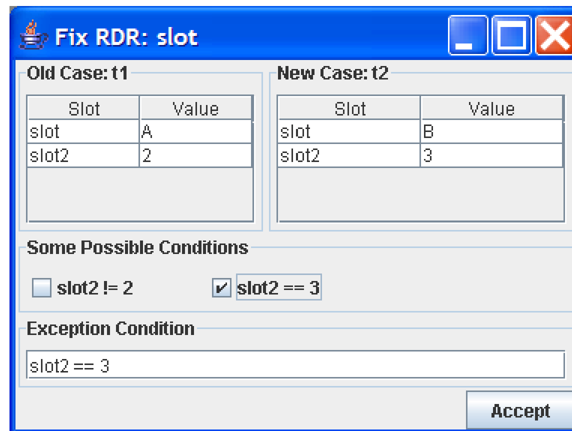

Figure 24: RDR Maintenance GUI

new rule will update. These conditions do not have to be used, they are only given as a way indicating differences between the 2 cases.

3. **Exception Condition :** The condition for the exception rule. This condition should be an expression that evaluates to `true` or `false` that can be used to identify exceptional cases that should take the new value.

4. **Accept Button :** When pressed this button will perform some basic syntax checks on the exception condition and if it's fine close the GUI and create the new exception rule.

If the GUI is closed without a new rule being created `fix_rdr_gui()` will throw an error.

4.4 Frame Browser

The frame browser(see figure 25), which can be opened by `frame_browser()`, is a GUI specifically designed to enable a user/developer to create new and manipulate existing frames.

4.4.1 Menu

The frame browser's menu allows a number of operation to be performed. A summary of the operations available through the menu is:

- **File**
  - **Load** - Loads a FrameScript file into memory.
  - **Load Module** - Load a FrameScript module into memory.
  - **Serialise** - Serialises FrameScript's symbol table to a file.
  - **Unserialise** - Loads the symbol table stored in a file.
- **Help**

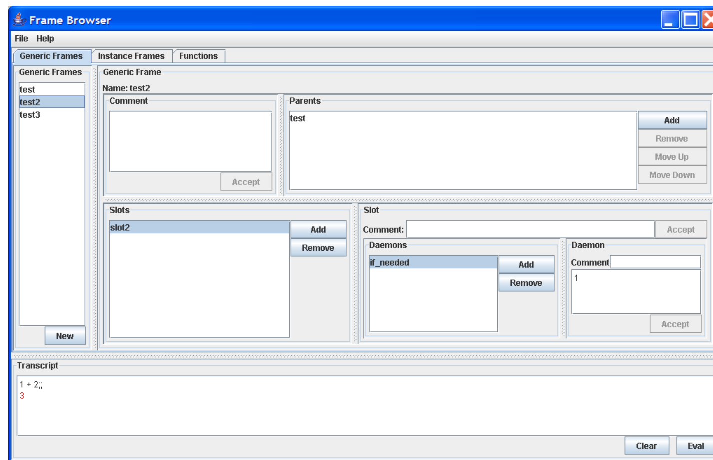

Figure 25: Frame Browser

- **Set Look & Feel -** Changes the UI look and feel.
- **About -** Describes the frame browser.

4.4.2 Transcript

In the lower section of the frame browser is the transcript. The transcript is essentially a block of FrameScript code to be evaluated. When the user clicks on the 'Eval' button the frame browser will parse statements written in the transcript and evaluate them. The results of the evaluation will be appended to the end of the transcript.

The purpose of the transcript is to allow sections of FrameScript code to be written and evaluated without having to resort to console input.

4.4.3 Generic Frames

The upper section of the frame browser has a series of tabs. The 'Generic Frames' tab shows the generic frames that are currently named in the symbol table. When a generic frame is selected from the list on the left its parents and slots will be displayed.

The frame browser can be used to add and remove parents from a generic frame or to rearrange the order in which parents are inherited. (FrameScript uses depth-first searches for inheritance so this can change a frame's behaviour)

Slots can be added to or removed from a generic frame and the daemons attached to a slot in a generic frame can be added or removed.

The 'New' button below the list of generic frames can be used to create a new generic frame.

4.4.4 Instance Frames

The 'Instance Frames' tab in the frame browser gives access to all the instance frames for the known generic frames.

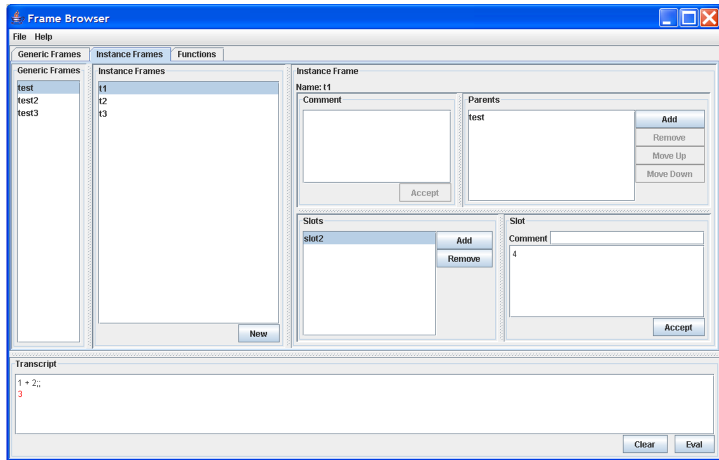

Figure 26: Instance Frame Browser

When a generic frame is select all of the instance frames that inherit from it will be listed. When one of these instances is selected its parents and slots will be displayed.

The frame browser can be used to add or remove the parents and slots of instance frames and the reorder the parents of an instance frame.

The 'New' button below the list of instance frames can be used to create a new instance frame that inherits from the currently selected generic frame.

4.4.5 Functions

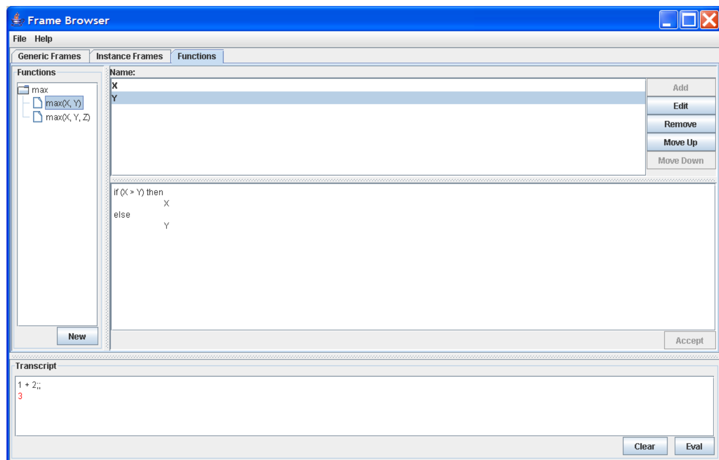

Figure 27: Function Browser

The 'Functions' tab of the frame browser allows users to view/create functions in FrameScript. (**NOTE:** this doesn't include subroutines)

On the left of the 'Functions' tab is a tree showing all of the currently active functions when a function is selected the parameters used in the function are listed and the body of the function is displayed.

It is possible to add parameters/remove/swap the parameters of a function. (**NOTE:** Calls to the function will need to be updated separately or they will break) It is also possible to edit the function's body.

The 'New' button below the known functions can be used to define new functions.

4.5 Script Browser

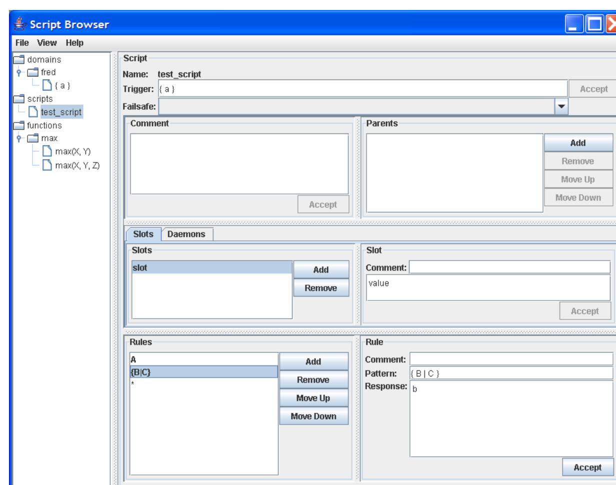

Figure 28: Script Browser

The script browser(see figure 28), which can be opened by `script_browser()`, is for the viewing and manipulation of scripts. It can be used to examine Frame-Script's state as a conversation progresses and to create and edit scripts.

On the left side of the script browser is a tree that displays all of the known domains and scripts named in the symbol table and the active functions. Also shown in the tree are the registered topics of a domain listed with the topics' triggers.

4.5.1 Menu

The script browser's menu allows a number of operation to be performed. A summary of the operations available through the menu is:

- **File**
    - **New**
        - **New Domain** - Creates a new domain.
        - **New Script** - Creates a new script.
        - **New Function** - Creates a new function.

- **Load File** - Loads a FrameScript file into memory.
- **Load Module** - Load a FrameScript module into memory.
- **Save Scripts** - Writes out the scripts in the symbol table to a file.
- **Serialise** - Serialises FrameScript's symbol table to a file.
- **Unserialise** - Loads the symbol table stored in a file.

- **View**

  - **View Recognition Daemon** - Switches the view of a script to the abnormal script view so that normal scripts can be made abnormal.
  - **Start Conversation** - Opens the conversation GUI(see section 4.5.5) using a given domain.

- **Help**

  - **Set Look & Feel** - Changes the UI look and feel.
  - **About** - Describes the script browser.

4.5.2 Domains

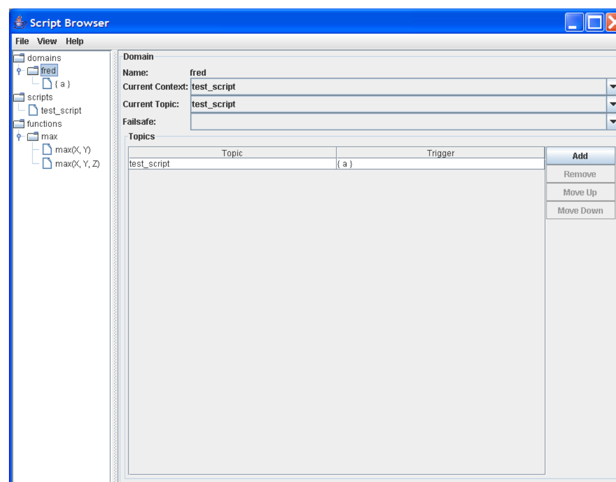

Figure 29: Domain Browser

When a domain is selected in the tree the details of that domain are shown in the right side of the script browser. (See figure 29)

Shown are the domain's current context, current topic, failsafe and the list of topics registered for the domain.

The domain's current context, current topic and failsafe can all be changed using comboboxes and its topics can be added, removed or reordered.

4.5.3 Scripts

If a script is selected in the script browser's tree. (Either by its name or as a topic in a domain) That script's details will be displayed on the right. The script browser will display any trigger specifically defined for the script, the script's failsafe, the script's parents, the script's slots, the script's daemons and the rules defined in the script.

The script browser can be used to change a script's trigger, failsafe, parents and slots. The rules for the script can be reordered and removed and new rules can be added. When a rule is selected the pattern and response for the rule are shown. Both the pattern and response can be edited.

Script Daemons

The 'Daemons' tab for a script shows the `on_entry` and `on_exit` daemons

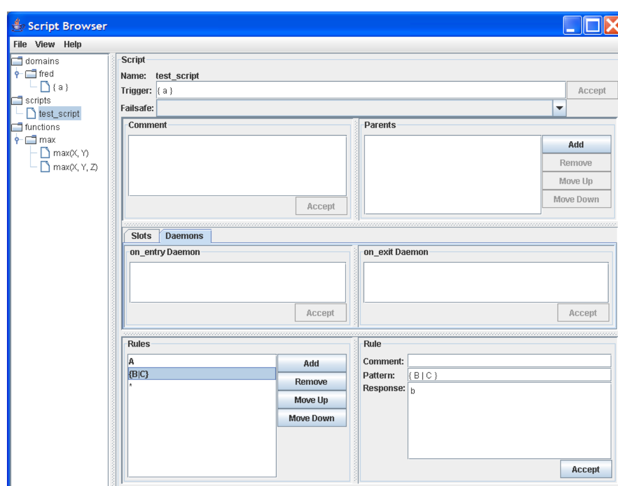

Figure 30: Script Daemon Browser

defined for a script and allows them to be edited.

Abnormal Scripts

Some scripts do not use the standard rule/response mechanism for processing inputs. If such a script is selected the recognition daemon for that script will be displayed. If a users selects 'View Recognition Daemon' in the menu the actual recognition daemon that implements the rules for the script will be shown. (Example shown in figure 31)

It is recommended that if a script is using the rule/response mechanism for its rules that the rules are added/edited using the standard rule interface and not using the recognition daemon interface as whenever a script is selected the script browser checks the recognition daemon to see whether or not the normal pattern/response rule interface can be used and writing recognition daemons that can be displayed though this interface is not a simple task.

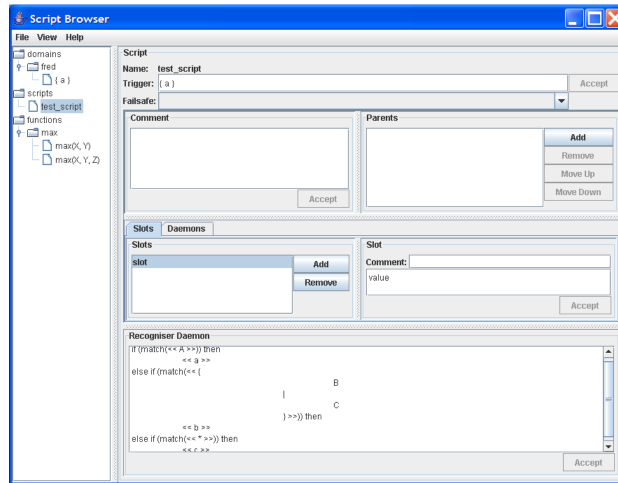

Figure 31: Abnormal Script Browser

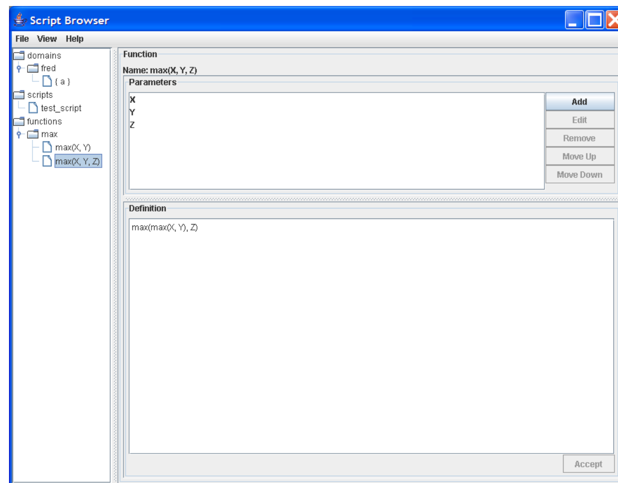

Figure 32: Function Browser

4.5.4 Functions

If a user selects a function from the tree then that function's details are shown. Both the function's parameters and body are shown and can be manipulated.

Beware when using this interface to add/remove/reorder parameters for a function as all calls to the function will need to be checked to ensure that they still correspond to the new function specification.

4.5.5 Conversation GUI

Figure 33: Conversation GUI

If a user selects 'Start Conversation' from the script browser's menu they will be asked to select a domain to manage the conversation. When a domain is selected the conversation GUI is opened using that domain to manage a conversation with the user. The conversation shows the conversation's history and at the bottom has a text field the user can use to enter their side of the conversation. The user can use the arrow buttons of their keyboard to flip through previously entered statements.

5 MicaBot

FrameScript is a language for scripting of verbal and multi-modal interactions. In order for it be used in an application it needs to be able to receive speech and events from somewhere and to be able to return its responses. The simplest way to manage this is to extend FrameScript and in the extension to provide an interface to whichever technology/architecture is being used to handle the device/agent/system communications.

MicaBot is a FrameScript extension that acts as a bridge between FrameScript and the Mica[3] agent architecture.

[3] Mica[4, 5] is a middle-ware layer for pervasive computing that aims to simplify communications between devices and agents and facilitate the separation of applications from their interfaces.

Among the extensions provided by MicaBot are a number of subroutines that allow the standard Mica function calls: writeMob, readMob, register, etc ... to be accessible in FrameScript, so that they can be used within responses. It also has the responsibility of turning Mica mobs into utterances and events to be parsed/processed by FrameScript.

When a MicaBot agent is first initialised it uses a TypeManager to explore the mob inheritance hierarchy. It then recreates this hierarchy using generic frames in FrameScript.

On initialisation MicaBot registers for 'textFromUser' mobs from which it uses the utterance slot to provide the speech input to FrameScript.

5.1 Speech Alternatives

Speech recognition systems don't just give one result when recognising speech but instead give a list of possible statements in the order of their likely probabilities. MicaBot allows these possible alternatives to be checked when processing the response to a user's statement. In the 'textForUser' mob the 'utterance' slot holds the most likely alternative. If there are other possible alternatives then they are placed in a list in the 'alternatives' slot of the 'textFromUser' mob.

```
failsafe ::
    NOMOREALTS ==> [ Sorry I didn't understand that. ]
    * ==> RECOGALT
;;
```

Figure 34: Speech Alternative Example

The most common way to access the alternatives from within FrameScript would then be to use a failsafe similar to Figure 34. In this example 'RECOGALT' is a simple response that tells MicaBot to use the next alternative. If there are no more alternatives to try MicaBot will give the input 'NOMOREALTS'.

5.2 SimpleTextAgent

SimpleTextAgent is a fairly simple Mica interface agent that writes 'textForUser' mobs to the blackboard and displays 'textForUser' and 'textFromUser' mobs as they arrive.

5.3 MicaRunner

If you wish to use MicaRunner to start/stop MicaBot then there are some parameters that can be passed to the MicaBot agent. These arguments are:

`domain` - the name of the FrameScript domain that the MicaBot uses to define its responses

`file` - the name of a script file to be loaded

`init` - a string used to initialise the conversation/interaction

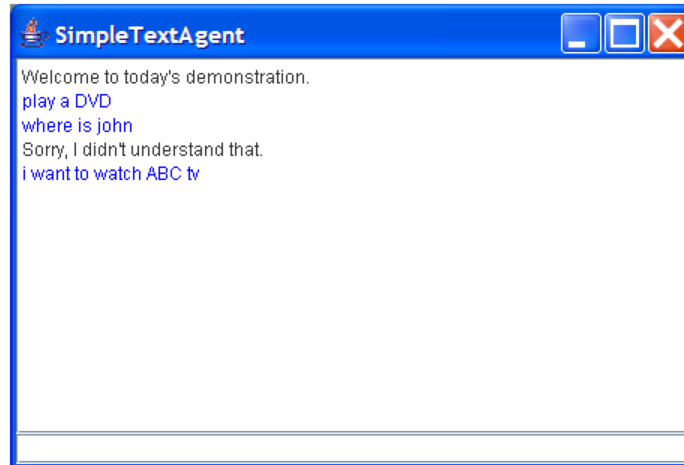

Figure 35: SimpleTextAgent

```
<runner host="localhost" port="8500">
    <blackboard>
        <restore value="false" />
        <debug level="information" />
    </blackboard>

    <agent class="sitcrc.framescript.SimpleTextAgent" />

    <agent class="sitcrc.framescript.MicaBot">
        <arg param="file" value="example.frs"/>
        <arg param="domain" value="example"/>
        <arg param="init" value="init"/>
    </agent>

</runner>
```

Figure 36: MicaRunner Startup Script Example

`transient` - either `true` or `false`, it tells MicaBot whether or not to make its textForUser responses transient(defaults to true)

The parameters should be used in this order: `transient`, `file`, `domain`, `init`.

A simple example startup script for MicaRunner that starts MicaBot and SimpleTextAgent is given in Figure 36. This script will start MicaBot with the file `example.frs` loaded using the `example` domain. It will then initialise MicaBot by giving it "init" as its first input.

6 Discussion

While it will never be a simple matter to write scripts for use in conversational agents it is possible to simplify the task. FrameScript's use of script inheritance is such a technique. Using script it is possible to define a set of basic behaviours in a script. It is then is a simple task to inherit these behaviours in subsequent scripts so that these behaviours can be used or altered. This allows common behaviours to defined only once rather than for each context in which they occur. This greatly simplifies editing scripts as the scripter only has to apply modification at one point and not have to search for each implementation of the behaviour.

Abnormal scripts are also a useful tool for scripting because they allow for the look-up of objects/frames whose characteristics match a given portion of an input text. Not only do abnormal scripts allow the look-up of objects they can do so in a way that means the look-up is performed as part of the function that compares the rule to the input. Because an abnormal script can be used to match against the characteristics of all instances of a generic frame they can provide a basic dynamic element to scripts as long as instances of the generic frame can be dynamically created, modified and destroyed.

Similarly because FrameScript allows not just text to be given as input but also events (where the event is represented by an instance frame) it can respond not only to user actions but also to system changes. For example FrameScript could tell a user if they receive an email or if their car was exceeding the speed limit. This could give a system a proactive feel as the system can initiate a conversation with a user rather than always waiting for the user to initiate a conversation. It also allows FrameScript to respond to non-verbal inputs such as the use of a touch pad or mouse and so allow the construction of multi-modal scripts.

Allowing daemons to be attached to scripts makes it possible for the system to alter environmental variables as it moves between contexts. For example the `on_entry` daemon for a script could be used to tell a speech recogniser which grammar to use for a specific context. Alternative if the system enters a 'watching movie' context it could turn off the lights and turn on the TV, then when it leaves it could turn the TV off and the lights back on.

For all the techniques employed in FrameScript to simplify the task of writing of scripts, writing scripts is still not simple. The script author has to take into account the ways a conversation can diverge and the myriad of ways that people can say the same thing. Also, they need to understand how the same sentence can take on different meanings depending on the current context of the conversation. The script writer must also take into account the very real possibility that the system may have to interact with hostile users.

FrameScript provides script writers with tools for creating and editing scripts but the interfaces they provide are very simple. There are probably better ways of visualising and editing scripts that make it easier for script writers to understand and manipulate the progression of a conversations through the contexts the scripts embody.

It may also be possible to create a library of scripts that provide the behaviours necessary for implementing the computer's side of common interactions that users have with their systems. Such interaction could include getting a person's address or phone number. Such a library could then be used across a number of possible systems and so allow script writers to focus on those interactions that are specific to their system.

One way to ease the burden on script writers would be to enable the scripts adapt themselves in response to interactions with users. Doing so could result in conversational agents that resemble Turing's Child Machine [8] in that they can learn by examples. A simple method of adding such a feature is the use of rules that construct/modify other rules. Such rules are feasible for limited domains where the patterns and responses for the new rule can be easily defined using some form of template. More generalised learning is a much more daunting problem, especially when it has to take into consideration the possible development of new contexts and sources of information.

Much of the recent work with FrameScript has been looking at using it to build speech interfaces to devices and programs. Very little however has been done to examine how it can be used in multi-modal interactions. While it can accept virtually any type of event a system can produce including events related to any input modality, work is still needed to identify the best methods for processing multi-modal input.

Presently work is exploring the use of FrameScript in multi-modal environments and for reporting events to a user. It is also looking at a variety of ways of representing and manipulating the current state of a conversation.

References

[1] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut. Ripple down rules: possibilities and limitations. In *6th Banff AAAI Knowlede Acquisition for Knowledge Based Systems Workshop*, 1991.

[2] Paul Compton and R. Jansen. A philosophical basis for knowledge acquisition. In *3rd European Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 75–89, 1989.

[3] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The hearsar-oo speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253, 1980.

[4] Mohammed Waleed Kadous and Claude Sammut. *The MICA Manual*, 0.1 edition, 2003.

[5] Mohammed Waleed Kadous and Claude Sammut. Mica: Pervasive middleware for learning, sharing and talking. In *PerCom Workshops*, pages 176–180. IEEE Computer Society, 2004.

[6] Marvin Minsky. A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[7] Claude Sammut. Managing context in a conversational agent. *Electronic Transactions on Artificial Intelligence*, 5(B):189–202, 2001.

[8] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, October 1950.

A BNF

Below is a BNF grammar defining the syntax for FrameScript. The terminal symbols *atom*, *number* and *string* have been left undefined.

statement :=
  frame-declaration "**;;**"
  |expression-list "**;;**"

frame-declaration :=
  generic-frame
  |instance-frame
  |script

generic-frame :=
  atom "**ako**" parent-list ["**with**" slot-definitions]

instance-frame :=
  atom "**isa**" parent-list ["**with**" slot-values]

parent-list :=
  atom ["**,**" parent-list]

slot-definitions :=
  atom "**:**" [daemon-definitions]

daemon-definitions :=
  daemon-definition [daemon-definitions]

daemon-definition :=
  daemon-name expression-list
  |slot-type "**true**"
  |slot-type "**false**"

daemon-name :=
  "**if_added**" |"**if_destroyed**"|"**if_new**" |"**if_needed**" |"**if_removed**"
  |"**if_replaced**" |"**default**" |"**range**" |"**help**"

slot-type :=
  "**multivalued**" |"**cache**"

slot-values :=
  slot-value [slot-values]

slot-value
  atom "**:**" expression

script :=
  atom "**::**" [script-header] [script-rules |"–" expression-list]

script-header :=
  script-modifier [script-header]

script-modifier :=
“**inherits**” atom
|“**instanceof**”[4] atom
|“**domain**” atom
|“**topic**”[5] atom
|“**trigger**” pattern-element
|“**on_entry**” expression-list
|“**on_exit**” expression-list
|“**failsafe**” atom
|atom “**:**” factor

script-rules :=
pattern
|script-rule [script-rules]

script-rule :=
pattern “**==>**” pattern-element

pattern :=
pattern-element [pattern]

pattern-element :=
sequence
|alternative
|non-terminal
|“ˆ” factor
|“**#**” factor
|atom [6]
|number
|string

sequence :=
“[” pattern-list “]”

alternative :=
“**{**” pattern-list “**}**”

non-terminal :=
“**<**” atom “**>**”

pattern-list :=
conditional-pattern [“|” pattern-list]

conditional-pattern :=
pattern [“**->**” pattern]

expression-list :=
expression [“**;**” expression-list]

expression :=
assignment-expression

[4] used for regression purposes
[5] used for regression purposes
[6] Except ˆ, #, },], >>, |, ==>, ->, ;;

assignment-expression :=
disjunction-expression
|assignment-expression “**=**” disjunction-expression

disjunction-expression :=
conjunction-expression
|disjunction-expression “**or**” conjunction-expression

conjunction-expression :=
relational-expression
|conjunction-expression “**and**” relational-expression

relational-expression :=
additive-expression
|comparison-expression comparison-operator additive-expression

relational-operator :=
“<” |“<=” |“>” |“>=” |“==” |“**!=**” |“**to**” |“**in**”

additive-expression :=
multiplicative-expression
|additive-expression “+” multiplicative-expression
|additive-expression “**-**” multiplicative-expression

multiplicative-expression :=
unary-expression
|multiplicative-expression “*****” unary-expression
|multiplicative-expression “/” unary-expression
|multiplicative-expression “**mod**” unary-expression

unary-expression :=
slot-retrieval
|“+” unary-expression
|“**-**” unary-expression
|“**new**” unary-expression
|“**not**” relational-expression
|“#” unary-expression
|“^” unary-expression

slot-retrieval :=
factor [“**of**” unary-expression]

factor :=
atom
|number
|string
|list
|“(” expression-list “)”
|“<<” pattern “>>”
|rdr-expression
|forall-expression
|variable-declaration
|compound

list :=
  "[" list-values "]"

list-values :=
  expression-list [list-values]

forall-expression :=
  "**forall**" atom "**in**" expression "**:**" expression

variable-declaration :=
  "**var**" var-list

var-list :=
  atom ["**,**" var-list

compound :=
  atom "**(**" [arg-list] "**)**"

arg-list :
  expression-list ["**,**" arg-list]

rdr-expression :=
  "**if**" expression "**then**" expression ["**because**" case] ["**except**" rdr-expression] ["**else**" expression]

case :=
  atom

B Built-in Subroutines

In order for a scripting language to be of much use it needs to provide a number of standard functions for performing basic operations on its basic types. As such FrameScript provides a set of mathematical and logical operations and functions for interacting with lists and frames.

While FrameScript is an untyped language its operations and functions often operate on specific types of terms. The following function descriptions give the types that each of FrameScript's standard functions expect. If they are given parameters of types they are not expecting they will throw an error. Below is a list of the parameter types that the standard functions may expect.

`atom` - an atom

`boolean` - either `true` or `false`

`domain` - a domain

`filename` - the name of a file, it can be in the form of a string or an atom

`generic` - a generic frame

`instance` - an instance frame (unless otherwise specified this includes scripts)

`integer` - an integer

`list` - a list

`modulename` - the name of a module, it can be in the form of a string or an atom

`number` - a number

`list` - a list

`pattern` - a pattern

`script` - a script

`slot` - the name of a slot, it must be an atom that will not get evaluated

`term` - a term, it can be any type of term including atom, string, number, pattern, RDR, expression list, ...

`variable` - the name of a variable, it must be an atom that will not get evaluated

B.1 Operators

FrameScript provides a number of mathematical and logic operations that can be used within scripts and frames.

B.1.1 +

Usage: `+number`
Precedence: 10
Description: Standard unary mathematical addition operator.
Returns: number
Parameters: `number` numerical operand

Usage: `number + number2`
Precedence: 50
Description: Standard mathematical addition operator.
Returns: number
Parameters:
`number` first numerical operand
`number2` second numerical operand

B.1.2 -

Usage: `-number`
Precedence: 10
Description: Standard unary mathematical subtraction operator.
Returns: number
Parameters: `number` numerical operand

Usage: `number - number2`
Precedence: 50
Description: Standard mathematical subtraction operator.
Returns: number
Parameters:
`number` first numerical operand
`number2` second numerical operand

B.1.3 *

Usage: `number * number2`
Precedence: 40
Description: Standard mathematical multiplication operator.
Returns: number
Parameters:
`number` first numerical operand
`number2` second numerical operand

B.1.4 /

Usage: `number / number2`
Precedence: 40
Description: Standard mathematical division operator.
Returns: number
Parameters:
`number` first numerical operand
`number2` second numerical operand
Error: Divide by 0 error if `number2` equals 0

B.1.5 mod

Usage:	`integer mod integer2`
Precedence:	40
Description:	Modulo operator.
Returns:	integer
Parameters:	`integer` first integral operand
	`integer2` second integral operand
Error:	Divide by 0 error if `integer2` equals 0

B.1.6 <

Usage:	`number < number2`
Precedence:	70
Description:	Less than comparison operator.
Returns:	boolean
Parameters:	`number` first numerical operand
	`number2` second numerical operand

B.1.7 <=

Usage:	`number <= number2`
Precedence:	70
Description:	Less than or equal to comparison operator.
Returns:	boolean
Parameters:	`number` first numerical operand
	`number2` second numerical operand

B.1.8 >

Usage:	`number > number2`
Precedence:	70
Description:	Greater than comparison operator.
Returns:	boolean
Parameters:	`number` first numerical operand
	`number2` second numerical operand

B.1.9 >=

Usage:	`number >= number2`
Precedence:	70
Description:	Greater than or equal to comparison operator.
Returns:	boolean
Parameters:	`number` first numerical operand
	`number2` second numerical operand

B.1.10 ==

Usage:	`term == term2`
Precedence:	70
Description:	Term equality check operator.
Returns:	boolean
Parameters:	`term` first operand
	`term2` second operand

B.1.11 !=

Usage:	`term != term2`
Precedence:	70
Description:	Term inequality check operator.
Returns:	boolean
Parameters:	`term` first operand
	`term2` second operand

B.1.12 =

Usage:	`term = term2`
Precedence:	100
Description:	Term assignment operator. Is right branching.
Returns:	term
Parameters:	`term` assignee
	`term2` value being assigned
Error:	If `term` cannot be assigned a value.

B.1.13 and

Usage:	`boolean and boolean2`
Precedence:	75
Description:	Logical conjunction operator. If `boolean` evaluates to `false` then `boolean2` isn't evaluated.
Returns:	boolean
Parameters:	`boolean` first operand
	`boolean2` second operand

B.1.14 or

Usage:	`boolean or boolean2`
Precedence:	80
Description:	Logical disjunction operator. If `boolean` evaluates to `true` then `boolean2` isn't evaluated.
Returns:	boolean
Parameters:	`boolean` first operand
	`boolean2` second operand

B.1.15 not

Usage: `not boolean`
Precedence: 72
Description: Logical negation operator.
Returns: boolean
Parameters:

`boolean`	operand

B.1.16 of

Usage: `slot of instance`
Precedence: 5
Description: Gets the value of a slot from an instance frame or a script. Is right branching.
Returns: term
Parameters:

`slot`	the name of the slot, **must be hard coded as it isn't evaluated**
`instance`	the instance frame/script whose slot value you a looking for

B.1.17 in

Usage: `term in list`
Precedence: 70
Description: List membership check operator.
Returns: boolean
Parameters:

`term`	the term that is a possible member
`list`	the list that `term` is a possible member of

B.1.18 new

Usage: `new generic`
Precedence: 50
Description: Generic frame instantiation operator. Creates an instance frame that inherits from the given type.
Returns: instance
Parameters:

`generic`	the generic frame being instantiated

B.1.19 ^

Usage: `^term`
Precedence: 20
Description: If the unevaluated `term` is an integer it gets the associated match component, otherwise it just returns the evaluated value of `term`. It's main usages are for retrieving sections of input for analysis and for dynamically generating responses.
Returns: term
Parameters:

`term`	either the index for a match component, or term to be evaluated

B.1.20

Usage: `#term`
Precedence: 20
Description: Evaluates `term` and returns nothing. Is is mainly used to insert FrameScript code for execution in responses that produce no visible response. It also is used in the pattern for a rule to provide a means of including the state of the system in the conditions of the rule.
Returns: nothing
Parameters:

`term`	term to be evaluated

B.1.21 to

Usage: `number to number2`
Precedence: 70
Description: An operator to ensure that a value inserted into a slot falls within a given range. It only works in daemons that define `new_value`. (eg. `range`, `if_added`, `if_replaced`)
Returns: boolean
Parameters:

`number`	inclusive lower bound of the range
`number2`	inclusive upper bound of the range

Error: If `new_value` is not a number.

B.1.22 var

Usage: `var [list_of_vars]`
Description: Not really an operator just a reserved atom used to define local variables in daemons/functions/forall statements.
Parameters:

`list_of_vars`	comma separated list of variable names

Error: If it is not at the top of the daemon/function/forall statement.

B.1.23 forall

Usage: `forall variable in list: term`
Description: Not really an operator just a reserved atom used to loop through a list. It returns the values of evaluating `term` for each value in `list`.
Returns: list
Parameters:

`variable`	name of a variable that takes its value from the elements in the list
`list`	a list of values for the variable
`term`	a term to be evaluated for each value in `list`

Error: If it is not at the top of the daemon/function/forall statement.

B.1.24 ->

Usage: `pattern -> pattern2`
Precedence: 90
Description: An operator that allows conditional responses to rules.
Returns: pattern
Parameters:

`pattern`	condition for production of the response (should evaluate to either `<< true >>` or `<< false >>`.)
`pattern2`	response to use if `pattern` evaluates to `<< true >>`

B.1.25 Precedence Table

Precedence	Operator
100	=
90	->
80	or
75	and
72	not
70	!=, ==, <, <=, >, >=, to, in
50	+(infix), -(infix), new
40	*, /, mod
20	#, ^
10	+(prefix), -(prefix)
5	of

Table 4: Operator precedence

B.2 General Functions

FrameScript has a number of standard functions for testing the types of terms and to manipulate list. It also provides mechanisms for interacting with the user.

B.2.1 trace

Usage: `trace(boolean)`
Description: Enables/disables trace reporting.
Returns: boolean
Parameters:

`boolean`	true to enable trace reporting, false to disable

B.2.2 verbose

Usage: `verbose(boolean)`
Description: Enables/disables verbose output.
Returns: boolean
Parameters:

`boolean`	true to enable verbose output, false to disable

B.2.3 atom

Usage: `atom(term)`
Description: Tests if `term` is an atom.
Returns: boolean
Parameters: `term` term to be tested

B.2.4 defined

Usage: `defined(term)`
Description: Tests if `term` has a defined value.
Returns: boolean
Parameters: `term` term to be tested

B.2.5 undefined

Usage: `undefined(term)`
Description: Tests if `term` does not have a defined value.
Returns: boolean
Parameters: `term` term to be tested

B.2.6 set

Usage: `set(atom, term)`
Description: Tests if `term` does not have a defined value.
Returns: term
Parameters:
- `atom` name of a variable to be given a global value
- `term` global value to be given to `atom`

B.2.7 number

Usage: `number(term)`
Description: Tests if `term` is a number.
Returns: boolean
Parameters: `term` term to be tested

B.2.8 integer

Usage: `integer(term)`
Description: Tests if `term` is an integer.
Returns: boolean
Parameters: `term` term to be tested

B.2.9 list

Usage: `list(term)`
Description: Tests if `term` is a list.
Returns: boolean
Parameters: `term` term to be tested

B.2.10 cons

Usage:	`cons(term, list)`
Description:	Constructs a list with `term` at its head followed by the values of `list`.
Returns:	list
Parameters:	`term` term to be the head of the list
	`list` the tail of the list

B.2.11 member

Usage:	`member(list, term)`
Description:	Tests if `term` is a member of `list`.
Returns:	boolean
Parameters:	`list` list whose membership is being tested
	`term` term whose membership of `list` is being tested

B.2.12 head

Usage:	`head(list)`
Description:	Gets the first value of a list.
Returns:	term, nothing if `list` is empty
Parameters:	`list` list of values

B.2.13 tail

Usage:	`tail(list)`
Description:	Gets the tail of a list. (ie. all values except the first)
Returns:	list
Parameters:	`list` list whose tail we want

B.2.14 nth

Usage:	`nth(integer, list)`
Description:	Gets the nth element of a list.
Returns:	term
Parameters:	`integer` index of the element (indices go from `0` to `length(list) - 1`)
	`list` list whose value is being retrieved
Error:	If `integer` is not a valid index of `list`.

B.2.15 length

Usage:	`length(list)`
Description:	Gets the number of elements in a list.
Returns:	integer
Parameters:	`list` list whose elements are to be counted

B.2.16 append

Usage: `append(list, list2)`

Description: Creates a new list where `list2` is appended to the end of `list`.

Returns: list

Parameters:

`list`	list whose elements are to be the start of the new list
`list2`	list whose elements are to be the tail of the new list

B.2.17 delete

Usage: `delete(term, list)`

Description: Creates a list with the same elements as `list` except the first occurrence of `term`.

Returns: list

Parameters:

`term`	the term to be removed from `list`
`list`	the list `term` is being removed from

B.2.18 fixrdr

Usage: `fixrdr()`

Description: Used to generate the construction of an exception to a Ripple Down Rule. (Ideally should be placed inside the `if_replaced` daemon of the slot whose value was replaced with the correct value)

Returns: RDR

Error: If no RDR has been evaluated, or the slot has no value, or no conditions are given to explain the exception.

B.2.19 rdr

Usage: `rdr(generic, slot)`

Description: Loops through the instances of `generic` and gets the value of their slot `slot`. Then it asks the user to verify the value evaluated. If it is wrong the user is asked for the correct value and the value of the `slot` slot is replaced for that instance.

Returns: nothing

Parameters:

`generic`	a generic frame whose instances' slots are to be tested
`slot`	the name of the slot where the ripple down rule is being tested

Error: Possible IOExceptions/SyntaxErrors when communicating with the user.

B.2.20 print

Usage: `print(term, ...)`
Description: Prints its arguments to the output stream.
Returns: term
Parameters: `term` term to be written to the output stream

B.2.21 error

Usage: `error(term, ...)`
Description: Throws an error. Uses its arguments to construct a message for the error.
Parameters: `term` message to be used for the error
Error: An error with `term` as its message.

B.2.22 ask

Usage: `ask(term)`
Description: Writes `term` to the output stream and waits for the user to respond with a term.
Returns: term
Parameters: `term` request to be written to the output stream
Error: Possible IOExceptions/SyntaxErrors when communicating with the user.

B.2.23 eval

Usage: `eval(term)`
Description: Gets the value of `term` and evaluates it.
Returns: term
Parameters: `term` the value to be reevaluated

B.2.24 quote

Usage: `quote(term)`
Description: Returns `term` without evaluating it.
Returns: term
Parameters: `term` the value to be given

B.2.25 load

Usage: `load(filename)`
Description: Reads in FrameScript code from a file.
Returns: nothing
Parameters: `filename` name of the file to be read in
Error: If file not found or, problems reading from the file.

B.2.26 load_module

Usage:	`load_module(modulename)`
Description:	Loads a module into memory.
Returns:	nothing
Parameters:	`filename` name of the module to be loaded
Error:	Number of possible error reading the Java class file the module is implemented in.

B.2.27 output_to_file

Usage:	`load(filename)`
Description:	Sets the output stream to go to a given file. Overwrites the file.
Returns:	nothing
Parameters:	`filename` name of the file to be output to
Error:	If problems writing to the file.

Usage:	`load(filename, boolean)`
Description:	Sets the output stream to go to a given file.
Returns:	nothing
Parameters:	`filename` name of the file to be output to `boolean` whether or not to append to the file
Error:	If problems writing to the file.

B.2.28 close_output

Usage:	`close_output()`
Description:	If the output stream is going to a file it is closed and the previous output stream is set to be the output stream.
Returns:	nothing

Usage:	`close_output(filename)`
Description:	If the output stream is going to the file `filename` it is closed and the previous output stream is set to be the output stream.
Returns:	nothing
Parameters:	`filename` name of the file to stop outputting to

B.2.29 print_as_text

Usage:	`print_as_text(term)`
Description:	Prints a term to the output stream. If `term` is either a frame or script it is written in a textual format that as much as possible can be read back in.
Returns:	term
Parameters:	`term` term to be written to the output stream

B.3 Frame Subroutines

As FrameScript uses frames to provide structure to data it needs functions that enable frames to be manipulated. Several of the functions below require

`current_object` which means they are intended for use in daemons and apply to the daemon whose access/manipulation resulted in the running of the daemon.

B.3.1 frame

Usage:	`frame(term)`
Description:	Tests whether or not a term is a frame.
Returns:	boolean
Parameters:	`term` term being tested

B.3.2 generic

Usage:	`generic(term)`
Description:	Tests whether or not a term is a generic frame.
Returns:	boolean
Parameters:	`term` term being tested

B.3.3 instance

Usage:	`instance(term)`
Description:	Tests whether or not a term is an instance frame. **NOTE:** As scripts are a subtype of instance frames this will return `true` if `term` is a script.
Returns:	boolean
Parameters:	`term` term being tested

B.3.4 instances_of

Usage:	`instances_of(generic)`
Description:	Gets all the instances of a generic frame.
Returns:	list
Parameters:	`generic` generic frame whose instances are desired

B.3.5 put

Usage:	`put(slot, term)`
Description:	Puts the value `term` into the `slot` slot of `current_object`.
Returns:	term
Parameters:	`slot` the slot the value is being put into `term` the value to be added to the slot
Error:	If there is no `current_object`, `slot` already has a value and is not multivalued or an error is thrown by a daemon.

Usage:	`put(instance, slot, term)`
Description:	Puts the value `term` into the `slot` slot of `instance`.
Returns:	term
Parameters:	`instance` the instance frame the value of the slot is being put into `slot` the slot the value is being put into `term` the value to be added to the slot
Error:	If `slot` of `instance` already has a value and is not multivalued or an error is thrown by a daemon.

B.3.6 replace

Usage:	`replace(slot, term)`
Description:	Replaces the value of the `slot` slot of `current_object` with `term`.
Returns:	term
Parameters:	`slot` the slot whose value is being replace `term` the value to be put into the slot
Error:	If there is no `current_object` or an error is thrown by a daemon.

Usage:	`replace(instance, slot, term)`
Description:	Replaces the value of the `slot` slot of `instance` with `term`.
Returns:	term
Parameters:	`instance` the instance frame whose slot value is being replaced `slot` the slot whose value is being replace `term` the value to be put into the slot
Error:	If an error is thrown by a daemon.

B.3.7 remove

Usage:	`remove(slot)`
Description:	Removes the given slot from `current_object`.
Returns:	term
Parameters:	`slot` the slot to remove from `current_object`
Error:	If there is no `current_object` or an error is thrown by a daemon.

Usage:	`remove(instance, slot)`
Description:	Removes the given slot from `instance`.
Returns:	term
Parameters:	`instance` the instance frame the slot is being removed from `slot` the slot to remove
Error:	If an error is thrown by a daemon.

Usage:	`remove(instance, slot, term)`
Description:	Removes a specific value from a multivalued slot in the given instance frame.
Returns:	term
Parameters:	`instance` the instance frame the slot value is being removed from `slot` the slot whose value is being remove `term` the value to be removed from the slot
Error:	If `slot` of `instance` is not multivalued or an error is thrown by a daemon.

B.3.8 destroy

Usage:	`destroy(instance)`
Description:	Destroys an instance frame. Runs `if_destroyed` daemons and removes the instance from the instance lists of all it's parent generic frames. Does not work on scripts.
Returns:	nothing
Parameters:	`instance` the instance frame to be destroyed
Error:	If `instance` is a script or an error is thrown by a daemon.

B.4 Script Routines

FrameScript provides a number of functions for moving from one script to another. It also has some functions to initiate conversations using domains.

B.4.1 script

Usage:	`script(term)`
Description:	Tests whether or not a term is a script.
Returns:	boolean
Parameters:	`term` term being tested

B.4.2 domain

Usage:	`domain(term)`
Description:	Tests whether or not a term is a domain.
Returns:	boolean
Parameters:	`term` term being tested

B.4.3 pattern

Usage:	`pattern(term)`
Description:	Tests whether or not a term is a pattern.
Returns:	boolean
Parameters:	`term` term being tested

B.4.4 register

Usage:	`register(domain, script)`
Description:	Registers `script` as the dominant topic of `domain`.
Returns:	nothing
Parameters:	`domain` domain the script is a topic of `script` script that is being registered as a topic

B.4.5 goto

Usage:	`goto(script)`
Description:	Sets the given script as the current context of the current domain.
Returns:	script
Parameters:	`script` the script to be the current context
Error:	If there is no current domain or an error is thrown by a daemon.

Usage: `goto(script, term)`
Description: Sets the given script as the current context of the current domain. Then gives `term` as an input to the domain.
Returns: script
Parameters:

`script`	the script to be the current context
`term`	a term to be used as an input to the domain

Error: If there is no current domain or an error is thrown by a daemon.

B.4.6 current_context

Usage: `current_context()`
Description: Gets the current context of the current domain.
Returns: script
Error: If there is no current domain.

B.4.7 previous_topic

Usage: `previous_topic()`
Description: Returns to the previous topic. Sets the topic as the current context.
Returns: nothing
Error: If there is no current domain or an error is thrown by a daemon.

Usage: `previous_topic(term)`
Description: Returns to the previous topic. Sets the topic as the current context. Then gives `term` as an input to the domain.
Returns: nothing
Parameters:

`term`	a term to be used as an input to the domain

Error: If there is no current domain or an error is thrown by a daemon.

B.4.8 new_event

Usage: `new_event(domain, term)`
Description: Gives a new input to a domain to be responded to.
Returns: pattern
Parameters:

`domain`	the domain that is being given a new input
`term`	a term to be used as an input to the domain

Error: If an error is thrown during processing of the response.

B.4.9 bot

Usage: `bot(domain)`
Description: Starts a simple console conversational interface using `domain` as the domain for the conversation.
Returns: atom
Parameters:

`domain`	the domain that is being used to determine the system's responses

Error: If an error is thrown during the processing of an input.

Usage: `bot(domain, term)`
Description: Starts a simple console conversational interface using `domain` as the domain for the conversation. Starts the conversation using `term` as the first input.
Returns: atom
Parameters:

`domain`	the domain that is being used to determine the system's responses
`term`	a term used to initialise the conversation

Error: If an error is thrown during the processing of an input.

B.4.10 match

Usage: `match(term)`
Description: Comparison to see if `term` is a valid pattern to match the current position in the current input.
Returns: boolean
Parameters:

`term`	a term to be used as a pattern to be matched against the current input

Error: If there is no current input.

Usage: `match(term, term2)`
Description: Comparison to see if `term` is a valid pattern that matches `term2`.
Returns: boolean
Parameters:

`term`	a term to be used as a pattern to be matched against the `term2`
`term2`	a term to matched against the pattern `term`

B.4.11 failsafe

Usage: `failsafe(domain, script)`
Description: Sets a script to be the global failsafe for a domain.
Returns: script
Parameters:

`domain`	the domain that for which the global failsafe is being defined
`script`	the global failsafe for the domain

B.4.12 question

Usage: `question(domain, term)`
Description: Initiates a conversation with the user that continues until a value is returned.
Returns: term
Parameters:

`domain`	domain that provides the system's side of the conversation
`term`	an input to be used to initiate the conversation

B.4.13 return

Usage:	`return(term)`
Description:	Is used to provide a return value for `question(domain, term)`. If used outside of a `question(domain, term)` call it does nothing.
Returns:	nothing
Parameters:	`term` the value to be returned by `question(domain, term)`
Error:	If there is no current domain.

B.5 GUIs

In the standard FrameScript distribution is the module 'sitcrc.framescript.GUI'. This is a small module that implements GUIs for communicating with the user using message and input dialogs and more complex GUIs for maintaining RDRs, frames and scripts. These GUIs are not active by default but can be enabled by calling `load_module("sitcrc.framescript.GUI")`.

B.5.1 dialog_message

Usage:	`dialog_message(term)`
Description:	Opens a dialog box with a message for the user.
Returns:	nothing
Parameters:	`term` a term used to provide the message for the user

B.5.2 dialog_question

Usage:	`dialog_question(term)`
Description:	Opens a dialog box requesting an input from the user.
Returns:	term
Parameters:	`term` a term used to provide the input request message
Error:	If there is an error parsing the user's input.

B.5.3 frame_browser

Usage:	`frame_browser()`
Description:	Opens the frame browser user interface.
Returns:	nothing

B.5.4 script_browser

Usage:	`script_browser()`
Description:	Opens the script browser user interface.
Returns:	nothing

B.5.5 fix_rdr_gui

Usage:	`fix_rdr_gui()`
Description:	Opens a graphical RDR maintenance interface.
Returns:	nothing
Error:	If there is no `current_object`, no RDR has been evaluated, no conditions are given to explain the difference, no value in the slot of `current_object`.

B.6 MicaBot

When using MicaBot several subroutines are loaded into FrameScript to allow access to the mobs on Mica's blackboard. Along with some new functions MicaBot defines some new term types. Many of the new functions use these new types. The new types are listed below.

`host` - the name of a machine hosting a blackboard, will usually be a string

`micabot` - a MicaBot

`mob` - an instance frame that inherits from the mob generic frame

B.6.1 micabot

Usage:	`micabot(domain)`
Description:	Creates a new MicaBot agent that uses the specified domain. The agent transport is configured using the current micabot.
Returns:	micabot
Parameters:	`domain` the domain the new MicaBot will use to talk to FrameScript
Error:	If there are any problems creating the new MicaBot or there is no current MicaBot.

Usage:	`micabot(domain, term)`
Description:	Creates a new MicaBot agent that uses the specified domain. The agent transport is configured using the current micabot. Then it initialises the conversation with a given input.
Returns:	micabot
Parameters:	`domain` the domain the new MicaBot will use to talk to FrameScript `term` an input used to initialise the conversation
Error:	If there are any problems creating the new MicaBot or there is no current MicaBot.

Usage: `micabot(domain, host, integer)`

Description: Creates a new MicaBot agent that uses the specified domain. The new MicaBot connects to the blackboard identified by the `host` and `port` parameters.

Returns: micabot

Parameters:

`domain`	the domain the new MicaBot will use to talk to FrameScript
`host`	the host the blackboard resides on
`integer`	the port on the host the blackboard has open

Error: If there are any problems creating the new MicaBot.

Usage: `micabot(domain, term, host, integer)`

Description: Creates a new MicaBot agent that uses the specified domain. The new MicaBot connects to the blackboard identified by the `host` and `port` parameters. Then it initialises the conversation with a given input.

Returns: micabot

Parameters:

`domain`	the domain the new MicaBot will use to talk to FrameScript
`term`	an input used to initialise the conversation
`host`	the host the blackboard resides on
`integer`	the port on the host the blackboard has open

Error: If there are any problems creating the new MicaBot.

B.6.2 mica_connect

Usage: `mica_connect(term)`

Description: Disconnects the current MicaBot and reconnects with the given name. It then registers for 'textFromUser' mobs.

Returns: atom

Parameters:

`term`	the agent name to use when reconnecting

Error: If there are any problems reconnecting or there is no current MicaBot.

Usage: `mica_connect(micabot, term)`

Description: Disconnects `micabot` and reconnects with the given name. It then registers for 'textFromUser' mobs.

Returns: atom

Parameters:

`micabot`	the micabot to disconnect and reconnect
`term`	the agent name to use when reconnecting

Error: If there are any problems reconnecting.

B.6.3 mica_register

Usage: `mica_register(term)`

Description: Registers the current MicaBot for the given mob type.

Returns: atom

Parameters:

`term`	the type of mob the current MicaBot is interested in

Error: If there are any problems registering or there is no current MicaBot.

Usage:	`mica_register(micabot, term)`
Description:	Registers `micabot` for the given mob type.
Returns:	atom
Parameters:	`micabot` the micabot that is interested in the mob type `term` the type of mob the MicaBot is interested in
Error:	If there are any problems registering.

B.6.4 mica_unregister

Usage:	`mica_unregister(term)`
Description:	Unregisters the current MicaBot for the given mob type.
Returns:	atom
Parameters:	`term` the type of mob the current MicaBot is no longer interested in
Error:	If there are any problems unregistering or there is no current MicaBot.

Usage:	`mica_unregister(micabot, term)`
Description:	Unregisters `micabot` for the given mob type.
Returns:	atom
Parameters:	`micabot` the micabot that is no longer interested in the mob type `term` the type of mob the MicaBot is no longer interested in
Error:	If there are any problems unregistering.

B.6.5 mica_read_mob

Usage:	`mica_read_mob(term)`
Description:	Uses the current MicaBot to get the named mob.
Returns:	mob or nothing
Parameters:	`term` the name of the mob to be read
Error:	If there are any problems reading the mob or there is no current MicaBot.

Usage:	`mica_read_mob(micabot, term)`
Description:	Uses the MicaBot to get the named mob.
Returns:	mob or nothing
Parameters:	`micabot` the micabot to read the mob `term` the name of the mob to be read
Error:	If there are any problems reading the mob.

B.6.6 mica_write_mob

Usage:	`mica_write_mob(mob)`
Description:	Writes a mob using the current MicaBot.
Returns:	atom
Parameters:	`mob` the mob to be written
Error:	If there are any problems writing the mob or there is no current MicaBot.

Usage:	`mica_write_mob(micabot, mob)`
Description:	Writes a mob using `micabot`.
Returns:	atom
Parameters:	`micabot` the micabot to write the mob `mob` the mob to be written
Error:	If there are any problems writing the mob.

B.6.7 mica_delete_mob

Usage:	`mica_delete_mob(term)`
Description:	Uses the current MicaBot to delete the named mob.
Returns:	nothing
Parameters:	`term` the name of the mob to be deleted
Error:	If there are any problems deleting the mob or there is no current MicaBot.

Usage:	`mica_delete_mob(micabot, term)`
Description:	Uses the MicaBot to delete the named mob.
Returns:	nothing
Parameters:	`micabot` the micabot to delete the mob `term` the name of the mob to be deleted
Error:	If there are any problems deleting the mob.

B.6.8 mica_query

Usage:	`mica_query(term)`
Description:	Uses the current MicaBot to get a list of mobs matching a query.
Returns:	list
Parameters:	`term` the query that defines the mobs being requested
Error:	If there are any problems querying the blackboard or there is no current MicaBot.

Usage:	`mica_query(micabot, term)`
Description:	Uses the MicaBot to get a list of mobs matching a query.
Returns:	list
Parameters:	`micabot` the micabot to query the blackboard `term` the query that defines the mobs being requested
Error:	If there are any problems querying the blackboard.

B.6.9 mica_write_wait_for_reply

Usage:	`mica_write_wait_for_reply(mob, integer)`
Description:	Uses the current MicaBot to write the given mob and waits for a reply. If no reply is received within the given time period(in milliseconds) then it returns nothing.
Returns:	mob, null if a timeout occurs
Parameters:	`mob` the mob to be written `integer` the number of miliseconds to wait for a reply
Error:	If there are any problems writing or receiving the mobs.

Usage:	`mica_write_wait_for_reply(micabot, mob, integer)`
Description:	Uses the MicaBot to write the given mob and waits for a reply. If no reply is received within the given time period(in milliseconds) then it returns nothing.
Returns:	mob, null if a timeout occurs
Parameters:	`micabot` the micabot to write the mob and wait for a reply `mob` the mob to be written `integer` the number of miliseconds to wait for a reply
Error:	If there are any problems writing or receiving the mobs.

B.6.10 get_mob_name

Usage:	`get_mob_name(mob)`
Description:	Gets the name of a mob.
Returns:	atom
Parameters:	`mob` the mob whose name is being requested

B.6.11 current_micabot

Usage:	`current_micabot()`
Description:	Gets the current MicaBot.
Returns:	micabot
Error:	If there is no current MicaBot.

C Utility Functions

TODO ...

C.1 Subroutine Argument Type Checking

TODO ...

C.1.1 check_alternatives

Usage:
```
check_alternatives(String functionName,
    Instance currentObject, Term arg[], int n,
    StackFrame frame)
```

Description: Function for checking that a argument to a function is an Alternatives.

Returns: Alternatives

Throws: FSError if the designated argument is not an Alternatives

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.2 check_atom

Usage:
```
check_atom(String functionName,
    Instance currentObject, Term arg[], int n,
    StackFrame frame)
```

Description: Function for checking that a argument to a function is an Atom.

Returns: Atom

Throws: FSError if the designated argument is not an Atom

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.3 check_boolean

Usage: `check_boolean(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is either `true` or `false`.

Returns: Atom._true or Atom._false

Throws: FSError if the designated argument is not `true` or `false`

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.4 check_compound

Usage: `check_compound(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Compound.

Returns: Compound

Throws: FSError if the designated argument is not a Compound

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.5 check_domain

Usage: `check_domain(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Domain.

Returns: Domain

Throws: FSError if the designated argument is not a Domain

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.6 check_exprlist

Usage: `check_exprlist(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is an ExprList.

Returns: ExprList

Throws: FSError if the designated argument is not a ExprList

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.7 check_frame

Usage: `check_frame(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Frame.

Returns: Frame

Throws: FSError if the designated argument is not a Frame

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.8 check_generic

Usage: `check_generic(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Generic frame.

Returns: Generic

Throws: FSError if the designated argument is not a Generic frame

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.9 check_instance

Usage:
```
check_instance(String functionName,
    Instance currentObject, Term arg[], int n,
    StackFrame frame)
```
Description: Function for checking that a argument to a function is a Instance frame. This check will also succeed on Scripts.

Returns: Instance

Throws: FSError if the designated argument is not a Instance frame

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.10 check_integer

Usage:
```
check_integer(String functionName,
    Instance currentObject, Term arg[], int n,
    StackFrame frame)
```
Description: Function for checking that a argument to a function is an integer.

Returns: FSInteger

Throws: FSError if the designated argument is not an integer

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.11 check_list

Usage:
```
check_list(String functionName,
    Instance currentObject, Term arg[], int n,
    StackFrame frame)
```
Description: Function for checking that a argument to a function is a list.

Returns: FSList

Throws: FSError if the designated argument is not a list

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.12 check_number

Usage: `check_number(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a number.

Returns: FSNumber

Throws: FSError if the designated argument is not a number

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.13 check_pattern

Usage: `check_pattern(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a pattern.

Returns: Pattern

Throws: FSError if the designated argument is not a pattern

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.14 check_rdr

Usage: `check_rdr(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is an RDR.

Returns: RDR

Throws: FSError if the designated argument is not an RDR

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.15 check_script

Usage: `check_script(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Script.

Returns: Script

Throws: FSError if the designated argument is not a Script

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.16 check_sequence

Usage: `check_sequence(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a Sequence.

Returns: Sequence

Throws: FSError if the designated argument is not a Sequence

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.1.17 check_string

Usage: `check_string(String functionName, Instance currentObject, Term arg[], int n, StackFrame frame)`

Description: Function for checking that a argument to a function is a string.

Returns: FSString

Throws: FSError if the designated argument is not a string

Parameters:

`functionName`	name of the function that is checking the argument
`currentObject`	enclosing object for the code which called the subroutine
`arg`	array of arguments to the function
`n`	index of the argument to check
`frame`	StackFrame that houses variable values

C.2 Files/Modules

TODO ...

C.2.1 loadFile

TODO ...

C.2.2 loadModule

TODO ...

C.2.3 setOutput

TODO ...

C.2.4 closeOutput

TODO ...

C.2.5 FileNotFound

TODO ...

C.2.6 evloop

TODO ...

C.3 Miscellaneous

TODO ...

C.3.1 compress

TODO ...

C.3.2 getMessage

TODO ...

C.3.3 getName

TODO ...

C.3.4 getPattern

TODO ...

C.3.5 sortAtomList

TODO ...

C.3.6 IOError

TODO ...

C.3.7 isPattern

TODO ...

C.3.8 isUnaryPattern

TODO ...

C.3.9 isPatternElement

TODO ...

C.3.10 formatComment

TODO ...

C.3.11 unformatComment

TODO ...

C.3.12 unformatComment

TODO ...

C.3.13 checkAllReferences

TODO ...

D Serialisation

TODO ...