Squl Documentation

Release 0.1

Michael van der Gulik

November 21, 2013

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This document describes the programming language Squl. Squl is a logic-based declarative programming language. It is quite similar to Prolog, but with a more verbose syntax.

1.1 Origins

Originally Squl was intended as a basis for creating an Artificial General Intelligence, but as a language it is also general enough for other uses. An Artificial General Intelligence is a construct which exhibits human-like intelligence.

- The Squl language is, hopefully, expressively complete. This means that the language can store and manipulate any concept a human can express.
- A Squl interpreter should be able to accept any syntactically valid Squl statement without having issues with being stuck in infinite loops. This allows experimentation with randomly generated statements.
- A Squl implementation is ideally implemented using persistence. Rather than reading modules from files every time an implementation is started, modules are stored in a database (a "long-term memory"), cached in memory (a "short-term memory" or "working memory") and imported or exported to files as required for archival.

The name "Squl" (pronounced "school") is related to the name "Faish". "Faish" is the initial implementation of the "Squl" programming language, and its goal is to be an environment which is about as intelligent as a fish - thus, an A.I. fish, or "Faish". "Squl", in turn, is related to the concept of a "school" of fish, or a place of learning. Faish is written using Smalltalk and is a slow, interpreted, single cored, in-memory implementation of Squl.

The implementation after Faish will be called "Daig", and its goal is to be an environment approximately as intelligent as a dog. Daig will probably be implemented using C and MPI in such a way that it can utilise a cluster to execute applications written in Squl, as well as being able to act as a database by paging statements to and from disk.

"Squl" is not related to SQL, other than both being able to be used as query languages, and both having similar names.

1.2 Differences from Prolog

Squl derives many of its charactoristics from the Prolog programming language. The syntax is quite different; Squl is more descriptive and lacks the conciseness of Prolog.

Whereas Prolog's execution semantics are based on a depth-first search, Squl's execution semantics are not defined explicitely, other than an informal approach of "trying not to be useless". Prolog requires that the programmer understand the execution semantics and uses them to his/her advantage by introducing cuts and recursion into specific places of a statement, both to aid efficient execution and to avoid infinite recursion. In comparison, Squl does not have a cut operator, and a Squl implementation is free to investigate clauses in whatever order it finds useful. This comes

with a performance penalty as dead ends are investigated, but allows much more flexibility with the implementation and a more pure approach to logic programming.

Prolog implements negation by failure. This is also available in Squl, but is an optional extra should the programmer want to define negation in this way.

Prolog allows custom pre-fix, post-fix or infix operators to be defined. These are not available in Squl. Squl does, however, have custom literal definitions which can achieve some of the same functionality.

Prolog does input and output in an impure manner, by relying on execution semantics. Squl, instead, does input and output by inverting control: rather than a module requesting input and output, the environment asks the module what it wants to input and output, and when it wants to do this. The module can include certain statements which inform the execution environment to enable various input and output devices.

1.3 Squl language specification

The Squl language is officially defined in this manual which allows other implementations of the language. The following chapters (TODO) describe the required components of an implementation of the Squl language:

- Language
- Modules
- Graphics
- Networking
- Inter-vat communication protocol

The module system in Squl is designed such that other basic language features, such as collections, can be built and transfered between implementations without requiring each language implementation to implement these libraries.

CHAPTER

TWO

FAISH TUTORIAL

This tutorial first describes some basic concepts of Squl, and then guides you through some of the interesting language features using the Faish implementation.

2.1 Language basics

Faish is the first implementation of the Squl programming language. The Squl programming language is designed as a basis for developing an "Artificial General Intelligence": an intelligent construct that has capabilities similar to a human. Squl is, at least hypothetically, capable of encoding any human thought, although no rigorous research has gone into attempting to prove the validity of this assertion.

Squl is a logic-based programming language. In contrast to conventional programming languages, a program written in a logic-based programming language describes the problem at hand to the computer, which then attempts to solve the problem. A program consists of a number of statements about the problem. This is an example statement:

```
father:alfred of:bob.
```

I.e. Alfred is the father of Bob. The language syntax is entirely trivial and can be learned in a few minutes. Using it, however, is going to take some time to master.

To aid with writing larger programs, a module system has been included as part of the Squl specification (i.e. this document) and implementated as part of Faish. This allows you to modularise your applications and create re-usable modules for, potentially, sharing with other users. A module has a name, author, date and so forth, and contains a number of statements. Modules can also link to each other to re-use each other's statements.

Modules are described in their own chapter TODO.

2.2 Using Faish

You're probably keen to start writing code. Open up Faish.

This will show the module list:

TODO

Now create a new module and call it, for example, "Genealogy". Open your new module by double-clicking it. You should see something similar to this:

TODO

This is where most of the action happens. Many of the features in the menus are not yet implemented but will be in future versions of Faish.

You will notice that there is already a statement in this module that starts with "metadata:(name:...". You can ignore these. These statements are module metadata that describe the name of the module, which other modules they import, which queries they retain and so forth. These are standard statements that you can query, write and delete if you wish, although you will experience the usual side-effects of deleting, for example, the module's name.

Enter the following in the text pane at the bottom of the window (replacing whatever was already there, if anything) and press CTRL+Enter:

```
father:alfred of:bob.
```

You should now see this statement in the left pane. The left pane lists all statements in this module.

In this statement, "father" and "of" are labels. "alfred" and "bob" are atoms. "father:alfred" is a clause; statements are made of many unordered clauses and finish with a full-stop (or "period"). Labels and atoms always start with a lower-case character.

Now enter this in the text pane at the bottom of the window, again replacing whatever was already there, and press CTRL+Enter:

```
father:X of:bob?
```

This is a query that says "Who is Bob the father of?". Note that this statement ends with a question mark, which signifies to Faish that this is a query. Faish will add this to the query pane on the right hand side and then show any results of this query.

Here, "X" is a variable. Variables always start with an upper-case character. When a query is run, the Squl interpreter tries to find values for variables. Variables have their context within a single statement; the same variable must have the same value whereever it occurs only in the same statement or if-then clause. If two separate statements share the same variable name, their variable values are completely independent from each other.

You will notice that Faish responds with new statements rather than with values for X. If you enter, for example, this query:

```
father:alfred of:bob?
```

then Faish will just return the same statement. If you enter a statement that is not true, such as:

```
father:alfred of:edward?
```

then Faish will simply respond with "No results", which is the same as saying "false" if you consider failure to be negation.

If you close this window and then double-click on the module in the module list to re-open it, you will notice that your queries are no longer there. In order to keep a query for later re-use, you can right-click it and select "Retain query". This adds a special metadata statement to the module which is read by Faish when a module is opened to restore any retained queries.

2.3 Literals

Examples of literals are integers and strings. These are the data items in a programming language that the user types in directly rather than writes code to create. In Squl, all literals are encased in square brackets. The first character of a literal defines its type:

You can define and add your own custom literals and parsers to extend Squl.

2.4 Making a list

Lists, trees, queues and other data structures can be made using sub-statements. These are statements inside statements.

This is the list containing the atom “first”, the number “2” and the string “three”:

```
h:first emnut:(h:[+2] emnut:(h:["Three] emnut:end)).
```

A convention in Squl is to label the first element of a list “h” and the rest of the list “emnut”. The last element in a list is “end”.

Here, we see statements inside other statements. Embedded statements have parenthesis around them, and they share variables with their outer statements. The first statement is “h:first emnut:(...)” with the ellipses being the embedded statement “h:[+2] emnut:(...)”, again with this next ellipses being the embedded statement “h:[”Three] emnut:end”.

This sounds like a difficult way of making a list. This is in fact the way that lists are created in Squl and many other functional and deductive languages. There is a short-cut to defining the list: it can also be written as a literal using a comma as the first character to define a list. The result is the same as the syntax below is just syntactic sugar:

```
[, first, [+2]], ["three]]].
```

The double closing square brackets are a special syntax for including a single closing square bracket inside a literal. The characters inside the square brackets are actually “, first, [+2], [”three]” when the double closing square brackets are replaced with single ones.

2.5 If-then rules

So far we have described a language which can store lots of interesting pieces of information, but cannot process it. In order to get interesting behaviour, we define “if-then” rules. These are statements which have any number of “if” clauses and a single “then” clause. For example:

```
if:(man:X) then:(mortal:X).
```

This means “If X is a man, then X is mortal”.

We usually write these clauses over several lines in this format, putting the “then” clause first:

```
then:(
    mortal:X )
if:(
    man:X ).
```

When investigating this statement, the Squl interpreter will try to find values for X.

Now if we run the query:

```
mortal:socrates?
```

We get no results. In the world we have defined, there are no men. We need to define a statement which can satisfy the “if” clause:

```
man:socrates.
```

Now if we re-run the query, we find that socrates is, unfortunately for him, mortal.

When we run a query, the Squl interpreter tries to find a value for a query by examining "then" clauses. If one matches, it tries to find solutions for all of the "if" clauses in that statement, again by examining "then" clauses in other statements.

For example, if we had the following statements:

```
then:(
    mortal:X )
if:(
    man:X ).

then:(
    man:X )
if:(
    human:X )
if:(
    alive:X ).

human:socrates.
alive:socrates.
```

Here, we say "if X is a man, X is mortal", and we say "if X is human, and if X is alive, then X is a man.".

Note that there are two separate variables named "X" here: one for each statement. A variable exists only within a statement. If another statement re-uses the same variable name, it is considered a completely different variable. There is no such thing as a global or shared variable in Squl.

We run this query:

```
mortal:X?
```

Faish will try to find any statement matching "mortal:X". It finds the first statement: "then:(mortal:X) if:(man:X).".

Then it tries to satisfy each if-clause by searching for any statement that has a then-clause matching "man:X". It finds the second statement.

Then it tries again to satisfy all the if-clauses, asking whether "human:X?" (finding "human:socrates." with X=socrates) and whether "alive:X?", or actually "alive:socrates?" as it has already decided that maybe X=socrates. It indeed finds "alive:socrates." as a statement.

Then Faish heads back to the top of the proof. We find that "man:socrates.". Then we go back up a level again and find "mortal:socrates." which satisfies our original query.

2.5.1 Recursion

Recursion is used in declarative programming languages where iteration is used in conventional programming languages. It is the only mechanism available for repeating anything in Squl.

If-then rules can contain their own then-clauses as if-clauses. When Faish tries to find an answer, it will then use the same rule many times over. For example, to find the last element of a list, we could use these statements:

```
list:(h:LastElement emnut:end) lastElement:LastElement.

then:(
    list:( h:H emnut:Emnut )
    lastElement:Last )
if:(
```

```
    list:Emnut
    lastElement:Last ).
```

You might need to stare at these statements for a while until your brain stops hurting. The author certainly did, but thankfully it becomes much easier with practise.

Briefly explained, the first statement is a "base case" for recursion. It is where the recursion will stop and a result is found. This statement means "The element of a list just before 'end' is the last element of the list".

The second statement states "the last element of a list is somewhere in the tail of the list". The tail of a list is all elements of the list other than the first. Faish will keep applying this statement, skipping over all elements in the list, until the first statement can be used to find the actual result.

Don't worry if you don't understand the example above yet. Recursion is a tricky concept, but thankfully most problems have the same pattern and, over time, using recursion becomes easier to understand.

What happens if we include a nasty statement which does infinite recursion on itself, such as:

```
then:(
    a:X )
if:(
    a:X ).
```

In this case, nothing spectacular happens. The Faish interpreter as of version 0.2 will just run the query for a while and find nothing interesting. If any results could be found from other statements, they might be found a bit slower. Hopefully in a future version of Faish, pointless recursive loops such as this one would be automatically detected and ignored rather than waste CPU cycles. In other words, you don't need to worry about left-recursion as you do in Prolog.

2.6 Managing module imports

Say that you want to use a statement in another module.

TODO

Click on "Edit", then "Add Module Import". Select a module you want to import and click "Okay". You can now use any statements in that other module which have been exported.

See chapter on Modules TODO for more information.

To make life as simple for the programmer, modules will be automatically downloaded from a module repository. TODO

2.7 Language conventions

To help code to be as readable as possible by different programmers, several conventions are used.

In real code, statements become quite complex so it is necessary to format them over several lines. Nested statements are indented. Closing parenthesis are included at the end of a line (for vertical compactness) and parenthesis have spaces on the inside rather than the outside (e.g. "(head:X tail:end)") unless they are adjacent to another parenthesis.

The "then" clause is included first by convention. "if" and "then" clauses occur on a line by themselves.

For example:

```
then:(
    sorted:(h:H emnut:(h:E emnut:Mnut))
    fn:SortFn
if:(
    fn:SortFn
    a:H
    b:E )
if:(
    sorted:(h:E emnut:Mnut) ).
```

If your statement takes in a particular data type and does something with it, then one label should show what data type is expected, and the other label shows the result:

```
list:In sorted:Out.
tree:In balanced:Out.
queue:In removeOverdue:Out.
n:Number doubled:NumberDoubled.
```

If an operation takes in a third argument, then the result can simply be called "result":

```
list:In append:Element result:Out.
tree:In removeAll:Element result:Out.
mapping:In removeKeys:KeyName result:Out.
```

Some common clause labels are:

fn:	A function name. This is an atom.
result:	The "output" from a function or operation.
a:	The first argument of a function
b:	The second argument of a function
c:	The third argument of a function
n:	A number, usually an integer.
i:	An index corresponding to the location of an element in a list.
s:	A statement.
q:	A query.

Higher-order operations take a function name (as an atom) as a value. "fn" is used as a short label name for the function when it is defined. For example, to double all elements in a list:

```
then:( list:In doubled:Out )
if:( collect:double list:In result:Out ).

then:(
    fn:double list:In result:Out )
if:(
    n:In multiply:[+2] result:Out ).
```

Here, "collect" applies the named function "double" to every element of a list.

Variables can have little suffixes to add information. "Out" is added (e.g. "TailOut") to annotate that a variable is a result. Conversely, "In" is used to annotate a variable is incoming, although usually just the variable name suffices. "Inc" can be suffixed to annotate that an integer is incremented by any amount and "Dec" for when a variable is decremented by any amount, e.g. "N", "NInc" and "NDec".

For lists and binary trees, there are three very useful variable names: Hemnut, Bokluz and Dagpos. These are akin to the canonical foo, bar and baz for variable names. Single letters denote an individual element; multiple letters denote

part of a list or a tree. Hemnut, Bokluz and Dagpos are specifically formatted as they are with consonents and vowels so they can be split up as follows:

H|Emnut	Single head element and the remaining tail of a list.
H|E|Mnut	First two elements of a list plus a tail.
Hemnu|T	Most of a list followed by a single tail element.
Hem|N|Ut	Some of a list or tree, a middle element, and the rest of the list.
He|M|N|Ut	Some of a list or tree, followed by two elements (M and N), followed by the rest.
Hem|Nut	Two branches of a binary tree.

"Hemnut" is used for input, "Bokluz" is used if a list is output, and "Dagpos" is used in emergencies. Hemnut is originally derived from "H" for head, "M" and "N" from the middle two letters of the alphabet, and "T" for tail.

Lists are made using "hemnut" as well, using "end" as the end of list or empty list marker:

```
h:firstElement emnut:(h:secondElement emnut:end).
```

Binary trees follow the same pattern as follows:

```
hem:leftBranch nut:rightBranch.
```

Ideally, however, a custom literal would be used, e.g. "[,firstElement, secondElement].".

2.8 Writing Tests

To open a module's test module, open a code module and use the menu item Modules → Open tests.

To run all tests, click on "Run all tests" in the test module's query pane's context menu. Note that this will clear all existing queries.

Modules containing user-written code can have its own test module. This provides the programmer with a convenient facility to write unit tests for his code. Test modules have special import rules; the usual import mechanism is bypassed in the interpreter and all statements in the code module are made available to the test module. In this way, tests can test all code defined in the code module without tests needing to be included in the code module.

When you click on "Open tests", a new test module is created if there isn't one already. If the code module already has an associated test module then it is downloaded and opened. Test modules are associated with a code module by including a statement of the form "module:_ metadata:(testModule:_ uri:_ name:_)." in the code module.

To make tests, enter statements of the form "test:X" into the test module where X is a statement. A test is assumed to pass if it returns at least one result, and assumed to fail if it returns no results.

For example, here is an example test to determine if the "n:_ plus:_ result:_." built-in statement is correctly setting the variable X:

```
test:(
    and1:(
        n:[+5] plus:[+1] result:X )
    and2:(
        equal:X w:[+6] ) ).
```

Here is an example of a test that passes if there are no results, where the user has defined "noResults:" elsewhere:

```
test:(
    noResults:(
        a:a b:b ) ).
```

In the code module, you can convert a query into a test by using “Add as test” in the query’s context menu. This will take the query, wrap it in a “test:X” clause and add it to the associated test module.

CHAPTER

THREE

THE SQUL LANGUAGE

3.1 Squl language syntax

The syntax of the Squl language is trivial.

A Squl application is made of many Squl statements. Each statement describes how a part of your application works.

A simple Squl statement looks like the following:

```
list:( head:H tail:Tail ) head:H.
```

This statement defines the head of a list to be the first element of a list.

Each statement consists of a number of clauses. Each clause has a label, followed by a colon, followed by a value. So in this case, the statement consists of two clauses. The first clause has the label "list", and a value which is a sub-statement. The second clause has the label "head" and the value "H".

Finally, a statement is concluded with a period. Queries are statements that conclude, instead, with a question mark.

Labels can consist of any printable Unicode characters other than a period, question mark, parenthesis, colon or uppercase Latin characters. Each label in a statement must be unique within that statement, with the exception of the special label "if".

The clauses in a statement can appear in any order. For example, these two statements are identical:

```
father:alfred of:bob.
of:bob father:alfred.
```

Values can be **atoms**, **varibles**, **sub-statements** or **literals**.

Atoms, such as "alfred", consist of the same characters as labels, and exist as placeholders. For example, a list with a single element would be "head:singleElement tail:end.", where "singleElement" and "end" are atoms.

Variables, such as "X", are values that begin with an uppercase Latin character followed by any number of characters that would be valid in a label. Their context is in a single statement - all instances of the same variable in a statement must have the same value, but if the same variable occurs in another statement, there is no relationship between the two statement's variables. One complication is when two statements are unified: variables with the same name in the two statements are considered by the implementation to be different variables, but this is seldem visible to the user.

Variables differ in usage from imperative programming. Variables are not "assigned" values in a way that they can be assigned a different value later. Instead, they are "matched" or "bound" to values to create a new statement. This is decribed in more detail later.

Sub-statements are just statements, enclosed in parenthesis to prevent them from falling apart. Any variables in sub-statements are considered to be part of the whole statement, so that instances of those variables, should they be deeply nested in substatements, are still bound to other instances elsewhere in the same statement.

Literals are integers, floats, characters, strings, and so forth. These are enclosed in square brackets. The very first character of a literal determines what type that literal is. Some examples of literals are:

[+52] The positive integer 52. '+' and '-' define integers.

[-10] The negative integer -10.

['K] The uppercase character 'K'. A single quote defines a character.

["Hello, world!] The string "Hello, world!".

A double-quote defines a string. A string can have any valid Unicode codepoints in it, including all whitespace and combining characters. A single closing square bracket ("]") terminates a string, even if it is part of a combining pair of Unicode codepoints. To include a closing square bracket inside the string, include two square brackets: ["A single square bracket:]]] is the same as "A single square bracket:] ". To include special characters, you just type them in using whatever mechanisms are available (i.e. in Faish 0.1, you probably can't). To include a newline character, you just type it in:

```
["Hello, world!
]
```

TODO: Floating point numbers are not yet implemented.

Another statement can be included as a statement literal rather than as a sub-statement. This will prevent any variables in that statement literal from being matched. The defining character for this is the backslash. For example, a statement literal appears like this: [head:a tail:end.].

One other type of literal is the module literal, but this is rarely used directly by a user but rather by utilities that the environment provides. These use the tab character as their defining character.

Literals can also be defined by the user.

TODO: user-defined literals not implemented yet.

3.2 Squl language semantics

3.2.1 Matching / Unification

An implementation of Squl would produce useful results from your program by combining statements to produce new ones, until a solution is found.

Take, for example, the list head example above:

```
list:( head:H tail:Tail ) head:H.
```

Say that we want an answer to this query ("what is the head of the list [,a, b]?"):

list:(head:a tail:(head:b tail:end)) head:X?

The following matches are made, by finding possible substitutions for all the variables in both statements:

```
H = a
Tail = ( head:b tail:end )
H = X
```

Here, we can see that H = a, and H = X, therefore X = a. The implementation then produces the following statement by substituting a for X:

```
list:(head:a tail:(head:b tail:end)) head:a.
```

This would be the result returned by the implementation. Note that no mention is made of the original X; you as a user are expected to be intelligent enough to see what happened to it.

If multiple substitutions, other than variables, are found for a variable then matching will fail and no result will be returned. For example:

```
list:( head:H tail:Tail ) head:H.
list:( head:a tail:end ) head:b?
```

This produces the following substitutions:

```
H = a
Tail = end
H = b
```

Here we have both H = a and H = b. Obviously a is not b, so this match would fail. If we had both H = a and H = a again, then it would succeed.

Substitutions also work with sub-statements. Say that we have these contrived statements:

```
a:( a:X ) b:B.
a:B b:( a:( b:c )).
```

Unifying (i.e. substituting variable) these together would produce the following substitutions:

```
a:X = B
B = a:( b:c )
```

Thus we can see that a:X = B = a:(b:c), thus a:X = a:(b:c), thus X = b:c, producing:

```
a:( a:( b:c )) b:( a:( b:c )).
```

There are some instances where unification can cause infinite loops of sub-statements in a statement. This is not implemented in the current version of Faish, but might be an interesting and useless esoteric feature to later include. Currently such statements cause the matching to fail.

3.2.2 If-then rules

To produce useful behaviour, we need some mechanism for being Turing-Complete. If-then rules achieve this by allowing for recursion.

An example if-then rule is:

```
if:(bounces:X) then:(ball:X).
```

However, it is more typically written in this format:

```
then:(
    bounces:X )
if:(
    ball:X ).
```

This is just another statement, spread over several lines. This is the conventional syntax for complex statements and is described in the language conventions section below.

Given, for example, the statement and query:

```
ball:myBlueBall.
bounces:myBlueBall?
```

Faish will then try to solve your query. First it searches for anything that matches "bounces:myBlueBall", and finds the if-then rule above. It then unifies the if-then statement to produce "then:(bounces:myBlueBall) if:(ball:myBlueBall).". Finally, it verifies the truth of the if-clause, by finding ball:myBlueBall, returning as result:

```
bounces:myBlueBall.
```

An if-then statement can have as many if-clauses as it wants. The then-clause is considered usable if all variables in the then-clause have values, and all if-clauses have been deduced or found in the modules.

3.2.3 Searching

When trying to find a solution to a query, multiple search paths are usually possible. For example:

- If a statement has multiple matches in the modules, then each of these could be attempted in any order.
- If a then-clause has multiple if-clauses, then those if-clauses could be investigated in any order.

Implementations are free to traverse the statements of a Squl application in whichever order they choose. There is, in fact, no guarantee your application will even run, but an implementation of Squl that does not actually produce useful results from your code will not be particularly popular.

It is entirely possible to have multiple processes or even multiple computers investigating the different options.

3.2.4 Built-in operations

Some statements are intercepted by the implementation to provide primitive operations, such as adding two numbers together. These are described below:

TODO - list them all.

::

n:X plus:Y result:Z. X plus Y equals Z.

n:X mult:Y result:Z. X multiplied by Y equals Z.

n:X divide:Y result:Z. The inverse of multiplication, X divided by Y equals Z. Y may not be zero.

head:X tail:Y string:Z. X is the first character of the string Z, and Y is the rest of the string.

n:X bitAt:Y result:Z. The Yth bit of X in binary is Z.

n:X bitAnd:Y result:Z. X in binary logically ANDed with Y, results in Z.

n:X bitOr:Y result:Z. X in binary logically ANDed with Y, results in Z.

bitNot:Y result:Z. Y in binary with all bits reversed results in Z.

n:X bitShift:Y result:Z. X in binary with all bits shifted right Y times results in Z.

n:X bitXor:Y result:Z. X in binary with all bits XORed with Y results in Z.

equal:X is:Y. X and Y are interchangable.

lesser:X greater:Y. X as an integer is less than Y.

char:Character codePoint:Integer. The given Character has Unicode codepoint of the given Integer.

query:Query numResults:N searchDepth:Depth timestamp:Time. The query, Query, when executed will have N results, when searched to a depth of Depth. Timestamp should always be a variable, and will be populated when this statement is used.

statement:Statement asLiteral:StatementLiteral. Convert the given Statement to a statement literal, or vice versa.

When the mathematics allows, these rules can also be used in reverse:

```
n:X plus:[+4] result:[+6]?
```

The built-in "query:Query numResults:N searchDepth:Depth timestamp:Time." allows for negation-by-failure as follows:

```
then:(
    noResults:Query )
if:(
    query:Query numResults:[+0] searchDepth:10 timestamp:Time ).
```

Usually the programmer is responsible for including a similar rule in his module when desired. The programmer must use a searchDepth as appropriate; a large value may cause huge inefficiencies as infinitely recursive clauses are investigated, while a too-small value may cause valid solutions to not be found.

Using "not:X" for negation-by-failure is considered bad form. Negation-by-failure is not actual negation.

Note that version 0.2 of the Squl language will have added semantics for determining which built-in operations are available.

CHAPTER

FOUR

MODULES

The feature of Squl which allows for scalable programming is modules. All statements are in a module. A module has a name, an author, a date and links to other modules.

Modules can be published for other users to download. Modules can be mutable (editable) or read-only; they must be read-only in order to be published for others to use.

4.1 Module importing

Modules are the basis for creating reusable components in Squl. A one-way link can be made between modules to allow one module to use statements in another module. This link is referred to as a "module import". This is usually facilitated by the user interface; in Faish, the user can browse other available modules and add imports using a simple dialog box.

After a module has been imported (i.e. a link has been made), not all statements in the imported module are available. Statements need to be exported first, by using the special "export" label. For example, if a statement contained a lot of parent-child relationships, they can all be made available in this way:

```
parent:alice of:bob.
parent:bob of:charles.
...thousands more.
export:( parent:X of:Y ).
```

The variables in an export-clause are only placeholders; any statement which matches with the given sub-statement will be made available to other modules.

If-then clauses get special treatment. If the then-clause has been exported, then results will be returned if they can be determined within this module. For example:

```
export:( grandparent:U of:V ).

then:( grandparent:X of:Z )
if:( parent:X of:Y )
if:( parent:Y of:Z ).

...parent clauses...
...
```

However, if the *importing* module defines some "parent:X of:Y." relationships, these will be ignored by the if-then rule defined in the imported module.

In order for an importing module to use the if-then rules of an imported module, those if-then rules need to be exported:

```
(module A, imports module B)
...lots of parent:X of:Y rules...

(module B)
export:( then:(grandparent:X of:Y) if:A if:B ).
then:( grandparent:X of:Z )
if:( parent:X of:Y )
if:( parent:Y of:Z ).
```

In this case, a "grandparent:X of:Y?" query on module A will succeed in returning results based on the parent relationships in module A.

4.2 Module metadata

The syntax for module metadata is as follows:

TODO: module metadata

4.3 Module export file format

Modules can be exported to a file and imported again later or into another Squl implementation. The file format is a standard text file using UTF-8. Statements are stored in a sorted order within the file to ensure that the file will work well with version control tools.

Exported modules have a MD5 digest attached to them, as well as to every module they import. This digest is used primary as an identifier for that module. If a module is modified, it is considered to be a completely new module with a new identifier. Module imports are recorded using these MD5 sums. This entails that the onus is on the programmer to update the dependencies of a module to their respective latest versions, rather than on the user. This guarantees that a module will execute with exactly the same dependencies that it was developed and tested in. Conveniently, the MD5 digest also provides a way of checking file integrity, although this is optional.

Modules imported into a running implementation refer to each other using a direct reference rather than by MD5 digest. MD5 digests are not used after a module is imported. This allows modules to be mutable when they are "alive".

TODO: export file format.

CHAPTER

FIVE

USING THE FAISH USER INTERFACE

5.1 Organising modules

Make a new module and open it. If the module uses the canvas or CLI, it will open that instead.

5.2 Entering statements

Type a statement in and press CTRL-Enter.

If it ends with a period, it's a statement. If it ends with a question mark, it's a query.

5.3 Queries

Normal queries

Simple queries.

Limiting results. Persisting queries.

5.4 Using the deduction browser.

5.5 Adding imports

This should be self-explanatary.

CHAPTER

SIX

GRAPHICS AND USER INTERACTION

Faish provides two ways to interact with the user: a simple command-line interface, and a canvas-based graphics API.

6.1 Ticks and Tocks

A module contains statements which are considered to be true, for all time. In order to account for changes in the state of any object, timestamps must be added to every declared state of objects. In other words, you say an object has a property or state, but only for a certain time or period of time. This also applies to actions; actions happen at a particular time.

A tock is a point in time. Each tock has another tock preceeding it (except for the first tock) and a tock following it (except for the latest tock). These are defined using the following relationship, which defines Tock1 to preceed Tock2:

```
tick:Tock1 tock:Tock2.
```

When a command line or graphics interface is presented to the user, a "working module" is made which imports the module containing the desired behaviour. The user interface then receives events from the user such as mouse and keyboard clicks. For each event, the following happens:

A new tock is created and added to the working module with its relationship to the previous tock:

```
tick:t0001 tock:t0002.
```

Information about the event is then added to the working module:

```
tock:t0002 timestamp:[+283882848].
tock:t0002 device:canvas action:(event:mouseClick at:[@12,3]).
```

The environment then performs the query:

```
at:t0002 device:canvas perform:X?
```

The desired response to this is in the module imported from the working module. The result found for X will considered to be a command and will be executed by the user interface.

6.2 Command-line Interface

The command-line interface is intended for simple applications which just want basic interactivity.

To enable the command line interface, include the following statement in your module:

TODO device:cli name:[”My Command line].

You can replace the name with whatever you want the window title to be.

Whenever the user enters a sentence on the command line, a statement such as this will be injected into the working module, as well as the tock relationship:

TODO device:cli tock:t1234 input:[”User input.].

where the string literal is the text that the user typed in.

You then need to implement logic to respond to the following queries:

TODO at:TockN device:cli perform:(output: X)?

where X is a string to present to the user in responce to the user's input.

For example, to respond “No” to everything the user enters:

    at:Tock device:cli perform:(output:[”No]).

6.3 Canvas interface

The canvas is a two-dimensional event and drawing API. Events from the user are timestamped with a tock and injected into the working module. Behaviour is done in response to queries.

- **drawing operations**
  - coordinate system
  - text
- events
- creating subcanvases
- creating link areas

CHAPTER

SEVEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*