

Finite Structure Software.

This paper describes a Java software that allows to define a finite structure, to define a monadic second order logic language and to evaluate the expressions in the defined language over the defined structure.

Logic overview.

The following passages are taken from "A Mathematical Introduction to Logic" - Herbert B. Enderton, Second Edition, Chapter 2.

"..." is used to show that text between passages as they appear in the book is omitted.

First Order Languages.

We assume henceforth that we have been given infinitely many distinct objects (which we call symbols), arranged as follows:

A. Logical symbols

0. Parentheses: (,).
1. Sentential connective symbols: \rightarrow , \neg .
2. Variables (one for each positive integer n):
 v_1, v_2, \dots
3. Equality symbol (optional): $=$.

B. Parameters

0. Quantifier symbol: \forall .
1. Predicate symbols: for each positive integer n , some set (possibly empty) of symbols, called n -place predicate symbols.
2. Constant symbols: Some set (possibly empty) of symbols.
3. Function symbols: For each positive integer n , some set (possibly empty) of symbols, called n -place function symbols.

...

An *expression* is any finite sequence of symbols. Of course most expressions are nonsensical, but there are certain interesting expressions: the terms and the wffs.

The terms are the nouns and pronouns of our language: they are the expressions that can be interpreted as naming an object. The atomic formulas will be those wffs having neither connective nor quantifier symbols.

...

The terms are defined to be those expressions that can be built up from the constant symbols and the variables by prefixing the function symbols. To restate this in the terminology of Chapter 1, we define for each n -place function symbol f , an n -place term-building operation F_f on expressions:

$$F_f(\varepsilon_1, \dots, \varepsilon_n) = f\varepsilon_1 \dots \varepsilon_n.$$

Definition. The set of *terms* is the set of expressions that can be built up from the constant symbols and variables by applying (zero or more times) the F_f operations.

...

An *atomic formula* is an expression of the form

$$Pt_1 \dots t_n,$$

where P is n -place predicate symbol and t_1, \dots, t_n are terms.

...

The *well-formed formulas* are those expressions that can be built up from the atomic formulas by use (zero or more times) of the connective symbols and the quantifier symbols. We can restate this in the terminology of Chapter 1 by first defining some formula-building operations on expressions:

$$\begin{aligned} \varepsilon_{\neg}(\gamma) &= (\neg\gamma), \\ \varepsilon_{\rightarrow}(\gamma, \delta) &= (\gamma \rightarrow \delta), \\ Q_i(\gamma) &= \forall v_i \gamma. \end{aligned}$$

Definition. The set of *well-formed formulas* (*wffs* or just *formulas*) is the set of expressions that can be built up from the atomic formulas by applying (zero or more times) the operations ε_{\neg} , $\varepsilon_{\rightarrow}$ and Q_i ($i = 1, 2, \dots$).

...

We define, for each wff α , what it means for x to *occur free* in α . This we do by recursion:

1. For atomic α , x occurs free in α iff x occurs in (i.e. , is a symbol of) α .
2. x occurs free in $(\neg\alpha)$ iff x occurs free in α .
3. x occurs free in $(\alpha \rightarrow \beta)$ iff x occurs free in α or in β .
4. x occurs free in $\forall v_i \alpha$ iff x occurs free in α and $x \neq v_i$.

...

Truth and Models.

A structure for a first-order language will tell us

1. What collection of things the universal quantifier symbol (\forall) refers to, and
2. What the other parameters (the predicate and function symbols) denote.

Formally, a *structure* U for our given first-order language is a function whose domain is the set of parameters and such that

1. U assigns to the quantifier symbol \forall a nonempty set $|U|$ called the *universe* (or *domain*) of U .
2. U assigns to each n -place predicate symbol P an n -ary relation $P^U \subseteq |U|^n$; i.e., P^U is a set of n -tuples of members of the universe.
3. U assigns to each constant symbol c a member c^U of the universe $|U|$.
4. U assigns to each n -place function symbol f an n -ary operation f^U on $|U|$; i.e. $f^U : |U|^n \rightarrow |U|$.

...

In order to define " σ is true U "

$$\models_U \sigma,$$

for sentences σ and structures U , we will find it desirable first to define a more general concept involving wffs. Let

σ be a wff of our language

U a structure for the language,

$s : V \rightarrow |U|$ a function from the set V of all variables into the universe $|U|$ of U .

Then we will define what it means for U to *satisfy* ϕ with s ,

$$\models_U \phi [s].$$

The informal version is

$\models_U \phi [s]$ if and only if the translation of ϕ determined by U , where the variables x is translated as $s(x)$ wherever it occurs free, is true.

The formal definition of satisfaction proceeds as follows:

I. Terms. We define the extension

$$\bar{s} : T \rightarrow |U|,$$

a function from the set T of all terms into the universe of U . The idea is that $\bar{s}(t)$ should be the member of the universe $|U|$ that is named by the term t . \bar{s} is defined by recursion as follows:

1. For each variable x , $\bar{s}(x) = s(x)$.
2. For each constant symbol c , $\bar{s}(c) = c^U$.
3. If t_1, \dots, t_n are terms and f is an n -place function symbol, then

$$\bar{s}(ft_1\dots t_n) = f^U(\bar{s}(t_1), \dots, \bar{s}(t_n)).$$

...

II. Atomic formulas. The atomic formulas were defined explicitly, not inductively. The definition of satisfaction of atomic formulas is therefore also explicit, and not recursive.

1. $\models_U = t_1 t_2 [s]$ iff $\bar{s}(t_1) = \bar{s}(t_2)$.

(Thus $=$ means $=$. Note that $=$ is a logical symbol, not a parameter open to interpretation.)

2. For an n -place predicate parameter P ,

$$\models_U P t_1 \dots t_n [s] \text{ iff } \langle \bar{s}(t_1), \dots, \bar{s}(t_n) \rangle \in P^U.$$

III. Other wffs. The wffs we defined inductively, and consequently here satisfaction is defined recursively

1. For atomic formulas, the definition is above.
2. $\models_U \neg \phi [s]$ iff $\not\models_U \phi [s]$.
3. $\models_U (\phi \rightarrow \psi) [s]$ iff either $\not\models_U \phi [s]$ or $\models_U \psi [s]$ or both.
4. $\models_U \forall x \phi [s]$ iff for every $d \in |U|$, we have $\models_U \phi [s(x|d)]$.

Here $s(x|d)$ is the function which is exactly like s except for one thing: At the variable x it assumes the value d . This can be expressed by the equation

$$s(x|d)(y) = \begin{cases} s(y) & \text{if } y \neq x \\ d & \text{if } y = x \end{cases} .$$

....

Software.

The software is written for a variation of many sorted monadic second order logic, where in addition to the symbols as described above, we have sort symbols. Each object in a structure is then assigned a sort. Thus it is possible to talk about integers, chairs, tables, universities, etc.

The set variables allow quantifiers to range over sets. A primitive *contains* (S, o) can be used to test whether o is in a set S .

The set of sentential connectives is extended to include: $\&\&$ and $\|\|$ which are conjunction (and) and disjunction (or).

An existential quantifier \exists "there exists" is also included in the language via *TE* identifier. A query symbol Q is included in the language. Then for a wff ϕ we may form an expression

$$Q\phi,$$

which is interpreted as a set of objects for which ϕ is true. That is, if x is a variable

$$Q(x, \phi) = \{d \mid \models_U \phi[s(x|d)]\} .$$

The software allows to define sorts, sort inheritance relation, objects, constant symbols, function symbols, predicate symbols and their interpretations in a structure. The definitions may be done in a Java programming environment: using the library provided, or using command line interface.

An identifier *DO* is added to a language. The syntax for using the symbol is

$$DO(((x_1, s_1), \dots, (x_k, s_k)), \phi_1, \phi_2) .$$

The symbol's interpretation is as follows: for every variable assignment as defined, if ϕ_1 is true, then the result of evaluating ϕ_2 on the variable assignment is added to a list of results.

Syntax.

Instead of \forall we use *FA*. Instead of \exists we use *TE*.

Define *variable definition*: $\text{varDef} = (\text{varName}, \text{varSort} \mid \text{setName} \mid \text{expr})$, where varDef is variable name, varSort is variable sort, setName is a name of a set, expr is expression evaluating to a set.

The syntax for wffs involving quantifier symbols or queries is:

$$X((\text{varDef}_1, \dots, \text{varDef}_k), \text{"}\phi\text{"}) ,$$

where X is FA , TE or Q and ϕ is a wff. The variables defined in $varDef_j$ are the variables which will be assigned values by the quantifier or query commands. ϕ is a wff that is to be evaluated.

We will use $\&\&$ instead of \wedge , $\|\|$ instead of \vee , $!$ instead of \neg , and $==$ instead of $=$.

Strings: there are two ways to enter strings: 1. strings may be delimited by quote symbol: "this is a string". 2. a beginning of a string may be delimited by a single quote symbol. The end of a string still have to be a double quote symbol. This allows to have nested strings: 'this is 'a string" inside a string". Quote symbols can be put inside a string by using an "escape" character: \ \backslash " or \ \backslash ', in which case they are not interpreted as indicating beginning or end of string.

Identifiers: instead of symbols, the software allows to enter identifiers. An identifier is an upper or lower case letter followed by any number of characters from the following set {a-z, A-Z, 0-9, $_$ }.

Objects and their identifier: objects may have identifiers associated with them. These will be called object ids, or just ids. Note that for integers, strings and lists, identifiers are a function of the object's content. That is the id of integer n is n. (Id of 34 is "34"). The id of a string is the string itself. The id of a list is the string representation of the list.

Name resolution: when software encounters an identifier or a string it attempts first to resolve it by matching it to a constant, function or a predicate. If this is not successful then the software attempts to match the identifier or string to a variable. If not successful, the identifier or string is matched to an object by comparing it with object's id. If not successful in case of the identifier, the software throws an exception. If not successful in case of a string the software creates a string object with the content the same as that of the encountered string.

Built-in object types:

integers - the software starts up with 64 integer objects: 0,1,...63

lists - may be entered as: (a1,a2,...,ak). Example: (1,2,3), (1,2,(1,2,3))

sets - different from lists in that no duplicates are allowed. May be entered as {a1,a2,...,ak}.

Example: {1,2,3}, $\{\{1,2,\{1,2,3\}\}$

strings - example: 'string', 'another 'string'"

graphs: the software supports directed graphs. A graph is created by using command: addObject(name,graph).

Then vertices may be created by specifying a graph and some other object which is used to mark the vertex:

addVertex(g,1)

addVertex(g,2)

edges are entered by specifying the graph object, object marking source vertex, and finally an object marking the destination vertex.

addEdge(g,1,2)

Boolean objects: the software starts up with three built-in boolean objects: TRUE, FALSE, UNCERTAIN.

Arithmetic operators and operator precedence:

The following may be used:

+ - add two numbers

* - multiply two numbers

- subtract second number from the first

== equality

= assignment (the correct usage is: c=expr, where 'c' is a constant's id, expr is expression.

The value of the expression is then assigned to c)

< - less

The correct precedence of the operators is not currently supported. When it is necessary to specify the order in which operations are to be performed the parenthesis are to be used: $2*(3+4)$.

Built-in functions and predicate:

listFunctions() - lists functions known to the software

listConstants() - lists constants

listPredicates() - lists predicates

listSorts() - lists object sorts

listObjects() - lists objects over which the quantification is defined

add(x,y) - x,y are integers. The result is x+y

sub(x,y) - x,y, integers. The result is x-y.

mult(x,y) - x,y, integers. The result is x*y.

addSort(sortId) - creates a new sort with a specified id. If a sort with the specified id already exists, throws an exception.

addObject(objectId, sortId) - creates an object with objectId of the specified sort. If an object with the specified id already exists, throws an exception.

relateSorts(parent, child) - creates an inheritance relation between two sorts, where 'child' is a subsort of 'parent'.

loadFile(fileName, flag) - loads a file with the name fileName (it is suggested that fileName is a string). The flag is a boolean value TRUE or FALSE. If TRUE is specified, the progress of loading a file is output onto the display.

saveFile(fileName) - save the structure into the specified file. The contents are saved as a text file with instructions as they might be entered via a command line interface.

addFunction(functionId, arity) - functionId is identifier or a string which is to be function's id, arity is integer specifying the number of arguments for a function.

addPredicate(predicateId, arity) - predicateId is identifier or a string specifying predicate's id, arity is integer specifying the number of arguments for a predicate

`addConstant (constantId)` - adds a constant with the specified id.
`setF(functionId, (arg1,arg2,...,argk),v)` - sets a function extension entry so that `f(arg1,arg2,...,argk)` evaluates to `v`.
`setP(predicateId, (arg1,arg2,...,argk),v)` - sets a predicate extension entry so that `p(arg1,...,argk)` evaluates to `v`, where `v` is `TRUE` or `FALSE`.
`setC(constantId, value)` - equivalent to: `constantId=value`.
`showExtension(id)` - lists extension entries for a specified function or predicate. The list will contain only the modifications made to an extension by a user using `setF()`, `setP()` commands. This will not list entries for built in function, or functions defined using `fdef` command.
`make(id)` - returns an object with the specified id if it already exists, otherwise creates a string with the specified id.
`appendList (listId, element)` - append the specified list with the specified element. Example:

```

>addConstant(c)
'c"
>c=(1,2,3)
(0:'1", 1:'2", 2:'3") [object]
>appendList(c, 4+5)
(0:'1", 1:'2", 2:'3", 3:'9") [object]

```

`appendSet (setId, element)` - append a set with the specified element. Example:

```

>set(s,{1,2,3})
{'1", '2", '3"} [set]
>appendSet(s,4+5+6)
{'1", '2", '3", '15"} [set]

```

`removeFromList(listId, index)` - removes the element with the specified index from the list. The index of the element is the position of the element in the list starting with 0. Example:

```

>c=(1,2,3,4)
(0:'1", 1:'2", 2:'3", 3:'4") [object]
>removeFromList(c,2)
'3" [integer]

```

`getFromList(listId, index)` - returns the elements of the list with the specified index. Example:

```

\bigskip >c=(1,2, (1,2,3))
(0:'1", 1:'2", 2:(0:'1", 1:'2", 2:'3")) [object]
>getFromList(c,2)
(0:'1", 1:'2", 2:'3") [object]

```

getFromSet(setId, index) - returns the element of the set with the specified index. The index of the element is the position starting with 0 of the element in the string representation of the set.

addToList(listId, index, element) - inserts the specified element in the position with the specified index in the list.

fdef (functionId, (varDef1, ...varDefk), 'expr') - defines a function with the functionId. The function is to accept k argument which are defined in the vardef list. The value of the function is computed by evaluating the expression. Example: we will define a tertiary function $f(x, y, z) \equiv (x + y)z$.

```
>fdef(f, ((x, integer), (y, integer), (z, integer)), '(x+y)*z')
'TRUE' [object]
>f(1,2,3)
'9' [integer]
>f(2,3,1)
'5' [integer]
>f(2,3,2)
'10' [integer]
```

Q, DO, FA, TE - these functions are already described above.

cat (x, y) - concatenates two strings, or lists. Examples:

```
>cat('this is a ', 'string')
'this is a string' [string]
>cat((1,2), (3,4))
(0:'1', 1:'2', 2:'3', 3:'4') [object]
```

and(a,b) - logical function. Evaluates to $a \wedge b$. Same as $a\&\&b$.

or(a,b) - logical function. Evaluates to $a \vee b$. Same as $a||b$.

not(a) - logical function. Evaluates to $\neg a$. Same as $!a$.

then (a,b) - logical function. Evaluates to $a \rightarrow b$ or equivalently $b \vee (\neg a)$. Same as $a- > b$.

=(c,v) - constant assignment function. Assigns value of v to a constant c .

count(x) - x is a set or a list. Computes number of elements in the set or a list.

addVertex(g,x) - adds a vertex to a graph g marked by object x.

vertex(g,x) - returns a vertex from graph g which is marked by an object x.

addEdge(g,x1,x2) - adds an edge to a graph g, from vertex marked by x1 to a vertex marked by x2.

indexOf(list, element) - returns an index of a specified element in the list.

inDegree(g,x) - returns in-degree of a vertex marked by x in a graph g.

outDegree(g,x) - returns out-degree of a vertex marked by x in a graph g.

edgeSet(g) - returns a set of edges of the specified graph. The set is a set of lists of two elements, where each element is of the form (s,d) where s is a source vertex, d is destination vertex.

vertexSet(g) - returns a set of vertices of the specified graph.

powerSet(s) - generates a set of all subsets of the specified set. The set can then be referenced by id P_s where s is the id of the argument set.

set(setId, expr) - identifies a set specified by expr with the id: setId.

graphSet(vertexNum) - generates all the directed graphs with the specified number of vertices. Please note that due to the memory limitations the command works for vertexNum one of the following {0,1,2,3}.

integerSet(l,u) - generates a set of integers $\{x \mid l \leq x \leq u\}$. Example:

```
>integerSet(2,50)
{'2", '3", '4", '5", '6", '7", '8", '9", '10", '11", '12", '13", '14", '15", '16",
'17", '18", '19", '20", '21", '22", '23", '24", '25", '26", '27", '28", '29",
'30", '31", '32", '33", '34", '35", '36", '37", '38", '39", '40", '41", '42", '
43", '44", '45", '46", '47", '48", '49", '50"}
```

integer(stringValue) - creates an integer object with the specified value. Example:

```
>integer('1024")+1
'1025"
```

true() - built-in predicate that always evaluates to true. This is useful in listing objects of a certain sort. Example:

```
>Q((x,integer),"true()")
(0:'0", 1:'1", 2:'2", 3:'3", 4:'4", 5:'5", 6:'6", 7:'7", 8:'8", 9:'9", 10:'10",
11:'11", 12:'12", 13:'13", 14:'14", 15:'15", 16:'16", 17:'17", 18:'18", 19:'19",
20:'20", 21:'21", 22:'22", 23:'23", 24:'24", 25:'25", 26:'26", 27:'27", 28:'28",
29:'29", 30:'30", 31:'31", 32:'32", 33:'33", 34:'34", 35:'35", 36:'36", 37:'37",
38:'38", 39:'39", 40:'40", 41:'41", 42:'42", 43:'43", 44:'44", 45:'45", 46:'46",
47:'47", 48:'48", 49:'49", 50:'50", 51:'51", 52:'52", 53:'53", 54:'54", 55:'55",
56:'56", 57:'57", 58:'58", 59:'59", 60:'60", 61:'61", 62:'62", 63:'63")
```

equals(x,y) - two place predicate. Return TRUE if x equals to y. The command is equivalent to $x == y$.

contains(x,y) - x is a list, set, string or a graph. Returns true if whenever x is a list or a set, x contains y as one of its elements. If x is a string, returns TRUE if y is a substring of x. If x is a graph then contains(x,y) returns true if a vertex marked with y is in the vertex set of x.

contains2(x,y) - if x is a string, contains2(x,y) is the same as contains(x,y). However if x is a list of a set, returns true if y is an element of x, or x is an element of one of the elements z of x whenever z is a list or a set. Example:

```
>contains2((1,2,{3,4}),4)
'TRUE"[object]
>contains((1,2,{3,4}),4)
'FALSE"[object]
```

less(x,y) - two place predicate for x,y integer. Return TRUE if $x < y$.

hasPath(g,x,y) - returns TRUE if there is a path from a vertex marked by x to a vertex marked by y in an undirected graph derived from g.

hasEdge(g,x,y) - returns TRUE if there is an edge from a vertex marked by x to a vertex marked by y in a graph g.

Example 1: arithmetic on a subset of integers.

Suppose that we have the set of integers in our universe: $0, 1, 2, \dots, 63$, function symbols: $+, *, -$ which is interpreted as a standard addition, multiplication and subtraction, binary relation $<$ which are interpreted as standard "less than" relation. We assume that quantification and queries are defined only over finite subset of integers: $0, 1, 2, \dots, 63$.

Suppose that we are interested in defining a relation $prime(x)$ which returns true if x is a prime and false otherwise. The formal definition of prime is (using p.91, "Mathematical Introduction to Logic", Enderton, Second Edition)

$$prime(x) \quad \forall y \forall z \ y * z = x \rightarrow (y = 1 \vee z = 1).$$

We will use this definition to define $prime$ relation:

```
>fdef(prime,((x,integer)), 'FA((y,integer),(z,integer)), '(y*z)==x)->((y==1) || (z==1))")
"TRUE"[object]
```

And now we can use the definition to determine whether a number is a prime or not:

```
>prime(19)
"TRUE"[object]
>prime(22)
"FALSE"[object]
```

Suppose that we are interested in determining all the divisors of 24: this can be expressed as

$$x : \exists y \ x * y = 24.$$

We express this as a query:

```
>Q(((x,integer)), 'TE(((y,integer)), '(x*y)==24")")
(0:'1", 1:'2", 2:'3", 3:'4", 4:'6", 5:'8", 6:'12", 7:'24")
```

The answer is a set of objects: $\{1, 2, 3, 4, 6, 8, 12, 24\}$ which indeed are the divisors of 24.

Now consider the following linear programming problem (given in CME305 class on 02-01-2007):

$$\min_{y_1, y_2, y_3, y_4} 8y_1 + 3y_2 + 2y_3 + 3y_4$$

over the set of nonnegative integers subject to the constraints:

$$\begin{aligned} 3y_1 + y_4 &\geq 6 \\ 2y_1 + 3y_2 + 2y_3 + y_4 &\geq 9 \\ 2y_1 + y_3 + y_4 &\geq 5. \end{aligned}$$

Suppose further that we already obtained a solution where $8y_1 + 3y_2 + 2y_3 + 3y_4 = 21$, and are interested in finding similar or better solutions. This can be formulated as query:

y_1, y_2, y_3, y_4 such that

$$\begin{aligned} 8y_1 + 3y_2 + 2y_3 + 3y_4 < 22 \text{ and } 3y_1 + y_4 \geq 6 \text{ and } 2y_1 + 3y_2 + 2y_3 + y_4 \geq 9 \text{ and} \\ 2y_1 + y_3 + y_4 \geq 5. \end{aligned}$$

We use this definition, noting that $q \geq x$ can be rephrased as $x - 1 < q$:

```
>Q(((y1,integer),(y2,integer),(y3,integer),(y4,integer)), '(((8*y1)+(3*y2)+(2*y3)
+(3*y4))<22) && (5<((3*y1)+y4)) && (8<((2*y1)+(3*y2)+(2*y3)+y4)) && (4<((2*y1)+y
3+y4)))"
(0:(0:'0", 1:'1", 2:'0", 3:'6"), 1:(0:'1", 1:'0", 2:'2", 3:'3"), 2:(0:'2", 1:'1"
, 2:'1", 3:'0"))
```

and obtain three solutions:

$(0, 1, 0, 6)$, $(1, 0, 2, 3)$, $(2, 1, 1, 0)$.

Suppose we would like to solve the following problem:

$$\min_{x \in J} x^2 - 6x + 9.$$

We can formulate the problem as follows:

$$\text{find } x \text{ in } J \text{ such that for all } y \text{ in } J \quad (x^2 - 6x + 9) \leq (y^2 - 6y + 9).$$

We express this definition and express it as a query:

```
>Q(((x,integer)), 'FA(((y,integer)), '(((x*x)-(6*x)+9)<((y*y)-(6*y)+9)) || (((x*x)
-(6*x)+9)==((y*y)-(6*y)+9)))"
(0:'3")
```

The answer is 3 which of course is a minimizer of the quadratic function.

Example 2:

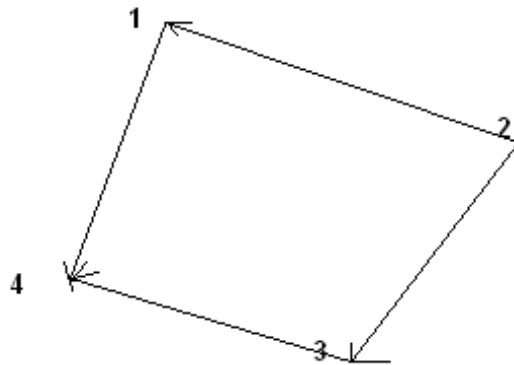
The following will define the property of graph g being connected:

$$\text{connected}(g) : \forall x \forall y (x \in g) \wedge (y \in g) \rightarrow \text{hasPath}(g, x, y).$$

We use this definition:

```
>fdef (connected, ((g, graph)), 'FA(((x, vertex), (y, vertex)), '(contains(g, x) && contains(g, y)) -> hasPath(g, x, y)')')
'TRUE' [object]
```

Now consider the following graph:
(figure 1)



The graph is connected. We will define this graph and check that it is connected:

```
>addObject(g, graph)
g({}, {}) [graph]
>addVertex(g, 1)
g({v[g.1]}, {}) [graph]
>addVertex(g, 2)
g({v[g.1], v[g.2]}, {}) [graph]
>addVertex(g, 3)
g({v[g.1], v[g.2], v[g.3]}, {}) [graph]
>addVertex(g, 4)
g({v[g.1], v[g.2], v[g.3], v[g.4]}, {}) [graph]
>addEdge(g, 2, 1)
g({v[g.1], v[g.2], v[g.3], v[g.4]}, {(v[g.2], v[g.1]), }) [graph]
>addEdge(g, 2, 3)
```

```

g({v[g.1], v[g.2], v[g.3], v[g.4]},{(v[g.2], v[g.1]), (v[g.2], v[g.3]), })[graph
]
>addEdge(g,3,4)
g({v[g.1], v[g.2], v[g.3], v[g.4]},{(v[g.2], v[g.1]), (v[g.2], v[g.3]), (v[g.3],
v[g.4]), })[graph]
>addEdge(g,1,4)
g({v[g.1], v[g.2], v[g.3], v[g.4]},{(v[g.1], v[g.4]), (v[g.2], v[g.1]), (v[g.2],
v[g.3]), (v[g.3], v[g.4]), })[graph]

>connected(g)
'TRUE"[object]

```

Now we will add another vertex "5" thus making a graph not connected:

```

>addVertex(g,5)
g({v[g.1], v[g.2], v[g.3], v[g.4], v[g.5]},{(v[g.1], v[g.4]), (v[g.2], v[g.1]),
(v[g.2], v[g.3]), (v[g.3], v[g.4]), })[graph]
>connected(g)
'FALSE"[object]

```

Example 3.

This example will demonstrate how a language and a model can be defined using a command line interface:

suppose that we are interested in discussing a situation where two people Tom and Bob are sitting on a chair at the table having some food and drink. In particular, there is tea, smoothie and two figs. Tea and smoothie is a drink whereas figs are food. The food and drink is on the table which is near a chair where Tom and Bob are sitting. Tom is having a smoothie and Bob is having a tea.

We then would like to define the following:

sorts: person, table, chair, drink, food: which are all subsorts of a sort "object".

relations: on, near, utilize.

We would like to say that:

Tom is a person, Bob is a person, table1 is a table, chair1 is a chair, tea is a drink, smoothie is a drink, fig1 is a food and fig2 is a food.

Further we would like to say that: fig2 is on the table, Bob is on the chair, Tom is on the chair, tea is on the table, smoothie is on the table, fig1 is on the table, table1 is near chair1, chair1 is near table1, Tom is utilizing smoothie and Bob is utilizing tea.

We do this as follows:

```

addSort("person")
relateSorts("object", "person")
addSort("table")
relateSorts("object", "table")
addSort("chair")

```

```

relateSorts("object", "chair")
addSort("drink")
relateSorts("object", "drink")
addSort("food")
relateSorts("object", "food")

addPredicate("on", 2)
addPredicate("near", 2)
addPredicate("talk", 2)
addPredicate("utilize", 2)

addObject("Tom", "person")
addObject("Bob", "person")
addObject("table1", "table")
addObject("chair1", "chair")
addObject("smoothie", "drink")
addObject("tea", "drink")
addObject("fig1", "food")
addObject("fig2", "food")

setP("on", ("fig2", "table1"), TRUE)
setP("on", ("Bob", "chair1"), TRUE)
setP("on", ("Tom", "chair1"), TRUE)
setP("on", ("tea", "table1"), TRUE)
setP("on", ("smoothie", "table1"), TRUE)
setP("on", ("fig1", "table1"), TRUE)
setP("near", ("table1", "chair1"), TRUE)
setP("near", ("chair1", "table1"), TRUE)
setP("utilize", ("Tom", "smoothie"), TRUE)
setP("utilize", ("Bob", "tea"), TRUE)

```

We can now ask the following questions:
 What drink is Bob having?

```

>Q(((x,drink)), "utilize(Bob,x)")
(O:'tea")

```

What is on the table1 that no one is having?

```

>Q(((x,object)), 'on(x,table1) && (!TE(((y,object)), 'utilize(y,x)'))')
(O:'fig1", 1:'fig2")

```

What is near the place on which there is Bob?

```
>Q(((x,object)), 'on(x,table1) && (!TE(((y,object)), 'utilize(y,x)'))")
(0:'fig1", 1:'fig2")
```

Example 4 (subsets containing primes).

Suppose that we would like to determine which subsets of $Z_6 = \{0, 1, 2, 3, 4, 5, 6\}$ consist only of primes.

This can be expressed as:

$$X \subseteq Z_6 \text{ such that for all } x \in X \text{ } x \text{ is a prime.}$$

The power set of Z_6 can now be referenced by "PZ6".

```
>Q(((X,powerSet(integerSet(0,6)))),'FA(((y,X)), 'prime(y)"))
(0:{}, 1:{'1"}, 2:{'2"}, 3:{'1", '2"}, 4:{'3"}, 5:{'1", '3"}, 6:{'2", '3"}, 7:{'1", '2", '3"}, 8:{'5"}, 9:{'1", '5"}, 10:{'2", '5"}, 11:{'1", '2", '5"}, 12:{'3", '5"}, 13:{'1", '3", '5"}, 14:{'2", '3", '5"}, 15:{'1", '2", '3", '5'})
```

The above query produces all the subsets of Z_6 consisting of primes.

Now we can choose a subset of greatest cardinality:

```
>set(result,Q(((X,powerSet(integerSet(0,6)))),'FA(((y,X)), 'prime(y)"))
){ {}, {'1"}, {'2"}, {'1", '2"}, {'3"}, {'1", '3"}, {'2", '3"}, {'1", '2", '3"}, {'5"}, {'1", '5"}, {'2", '5"}, {'1", '2", '5"}, {'3", '5"}, {'1", '3", '5"}, {'2", '3", '5"}, {'1", '2", '3", '5'}}[set]
>Q(((Y,result)), 'FA(((X,result)), 'count(X)<count(Y) || (count(Y)==count(X))")
)
(0:{'1", '2", '3", '5'})
```

Example 5 (independent set of a graph)

The following definition for undirected graph was given in CME305 class on 02-06-2007: an independent set is a set $S \subseteq V$ of vertices such that $\forall u, v \in S \quad \{v, u\} \notin E$.

Since we will be dealing with a directed graph, we'll change the definition slightly:

an independent set is a set $S \subseteq V$ of vertices such that $\forall u, v \in S \quad \{v, u\} \notin E$ and $\{u, v\} \notin E$.

Now consider the graph in figure 1.

```
>g
g({v[g.1], v[g.2], v[g.3], v[g.4]},{(v[g.1], v[g.4]), (v[g.2], v[g.1]), (v[g.2], v[g.3]), (v[g.3], v[g.4]), })[graph]
```

We will now determine all maximal independent subsets of g :
 set $S \subseteq V$ of vertices which is independent and such that if $w \notin S$ then there is an edge to/from w to some $u \in S$.

That is

want $S \subseteq V$ such that $\forall u, v \in S [\{v, u\} \notin E \text{ and } \{u, v\} \notin E]$ and $\forall w \in V [w \notin S \rightarrow \exists u \in S [\{w, u\} \in E \text{ or } \{u, w\} \in E]]$:

```
>Q(((S,powerSet(vertexSet(g)))),'FA(((u,S),(v,S)),'(!hasEdge(g,u,v)) && (!hasEdge(g,v,u))) && FA(((w,vertexSet(g)),'(!contains(S,w))->TE((u,S)),'hasEdge(g,w,u) || hasEdge(g,u,w)'))))
(0:{v[g.1], v[g.3]}, 1:{v[g.2], v[g.4]})
```

As you can see the answer is that there are two such independent sets of g : $\{1, 3\}$ and $\{2, 4\}$.

Example 6: (vertex cover of a graph)

A vertex cover as defined in CME305 lecture 02-08-2007 is

a subset S of vertices such that every edge of a graph is incident to some element of S .

This can be expressed as follows, for a graph $G(V, E)$:

want $S \subseteq V$ such that for all $e \in E$ there is $v \in S$ such that $v \in e$.

Consider the graph in figure 1:

```
>g
g({v[g.1], v[g.2], v[g.3], v[g.4]},{(v[g.1], v[g.4]), (v[g.2], v[g.1]), (v[g.2], v[g.3]), (v[g.3], v[g.4]), })[graph]
```

Now run a query:

```
>Q(((S,powerSet(vertexSet(g)))),'FA(((e,edgeSet(g)),'TE((v,S)),'contains(e,v)'))))
(0:{v[g.1], v[g.3]}, 1:{v[g.1], v[g.2], v[g.3]}, 2:{v[g.2], v[g.4]}, 3:{v[g.1], v[g.2], v[g.4]}, 4:{v[g.1], v[g.3], v[g.4]}, 5:{v[g.2], v[g.3], v[g.4]}, 6:{v[g.1], v[g.2], v[g.3], v[g.4]})
```

Now let's obtain a vertex cover of minimal cardinality:

```
>set(result,Q(((S,powerSet(vertexSet(g)))),'FA(((e,edgeSet(g)),'TE((v,S)),'contains(e,v)'))))
{{v[g.1], v[g.3]}, {v[g.1], v[g.2], v[g.3]}, {v[g.2], v[g.4]}, {v[g.1], v[g.2], v[g.4]}, {v[g.1], v[g.3], v[g.4]}, {v[g.2], v[g.3], v[g.4]}, {v[g.1], v[g.2], v[g.3], v[g.4]}}[set]
>Q((X,result),'FA((Y,result),'(count(X)<count(Y))||(count(X)==count(Y))'))
(0:{v[g.1], v[g.3]}, 1:{v[g.2], v[g.4]})
```

The result is that there are two vertex covers of cardinality 2: $\{1, 3\}$ and $\{2, 4\}$.

Example 7: (Euler ϕ -function)

Consider problem #5 p.8 from "Abstract Algebra", Third Edition by David S. Dummit and Richard M. Foote:

Determine the value of $\phi(n)$ for each integer $n \leq 30$ where ϕ denotes the Euler ϕ -function.

Solution: from p.7 of the book: the Euler ϕ -function is defined as follows: for $n \in \mathbb{Z}^+$ let $\phi(n)$ be the number of positive integers $a \leq n$ with a relatively prime to n , i.e., $(a, n) = 1$.

We will now use the following definitions for positive integers:

divides (n, a) there exists $k \leq a$ such that $nk = a$.

divisors (a, b) all n such that *divides* (n, a) and *divides* (n, b)

$\text{gcd}(a, b) = \max(\text{divisors}(a, b))$

eulerPhi (n) the number of $a > 0$ such that $a \leq n$ and $\text{gcd}(a, n) = 1$.

We will enter the definitions:

```
>fdef(divides,((n,integer),(a,integer)), 'TE(((k,integer)), 'k<(a+1)&&((n*k)==a)')")
'TRUE"[object]
>fdef(divisors,((a,integer),(b,integer)), 'Q(((n,integer)), 'divides(n,a)&&divides(n,b)')")
'TRUE"[object]
>fdef(gcd,((a,integer),(b,integer)), 'getFromList(divisors(a,b),count(divisors(a,b))-1)')
'TRUE"[object]
>fdef(eulerPhi,((n,integer)), 'count(Q(((a,integer)), '0<a)&&(a<(n+1))&&(gcd(a,n)==1)')")
'TRUE"[object]
>D0(((n,integer)), '0<n)&&(n<31)', 'eulerPhi(n)')
(0:'1", 1:'1", 2:'2", 3:'2", 4:'4", 5:'2", 6:'6", 7:'4", 8:'6", 9:'4", 10:'10",
11:'4", 12:'12", 13:'6", 14:'8", 15:'8", 16:'16", 17:'6", 18:'18", 19:'8", 20:'1
2", 21:'10", 22:'22", 23:'8", 24:'20", 25:'12", 26:'18", 27:'12", 28:'28", 29:'8")
```

Which is a list where k th element is $\phi(k + 1)$. To make a list more readable consider the following command:

```
>D0(((n,integer)), 'n<31)', 'eulerPhi(n)')
(0:'0", 1:'1", 2:'1", 3:'2", 4:'2", 5:'4", 6:'2", 7:'6", 8:'4", 9:'6", 10:'4", 1
1:'10", 12:'4", 13:'12", 14:'6", 15:'8", 16:'8", 17:'16", 18:'6", 19:'18", 20:'8
", 21:'12", 22:'10", 23:'22", 24:'8", 25:'20", 26:'12", 27:'18", 28:'12", 29:'28", 30:'8")
```

Here k th element has value $\phi(k)$. However the reader should disregard 0th element since $\phi(0)$ is not defined.

Conclusion.

The Java software package described provides a capability to define programmatically or via command line interface arbitrary monadic second order language and an arbitrary finite structure which interprets the language. The package then evaluates monadic second order expressions and queries involving monadic second order expressions in the defined structure.