KRR Languages: Quick Reference Ver. 5.3 (Feb 2024)

S. Baskaran

Introduction

This document outlines the specifications for six languages used in assignments for Knowledge Representation and Reasoning course offered by Prof. Deepak Khemani at IIT Madras. Abstract grammars for the languages are specified in BNF (Backus-Naur Form) notation; keywords, identifiers, constants, and operators (their precedence and associativity) are specified separately.

In the grammars, *non-terminal symbols are in italic font* and `terminal symbols are in blue monospace font` and BNF meta-characters include:

$\rightarrow$	defines a rule	?	zero or one
|	alternation	$*$	zero or more
ϵ	empty string	$+$	one or more
$\langle \cdots \rangle$	grouping	$\triangleright$	line comment

Keywords and operators are drawn from ASCII set, additionally, equivalent Unicode math operators are supported to express code in math notation.

The abstract grammars only depict ASCII keywords and operators, the equivalent Unicode math operators are listed separately. Take the Unicode math operators from the sample input files provided the bundle and *do not copy* Unicode operators from this PDF.

1 Propositional Logic

1.1 Grammar

1: *program* $\rightarrow \langle$ *sentence* | *block* $\rangle*$

2: *block* $\rightarrow$ `{` $\langle$ *sentence* | *block* $\rangle*$ `}` $\triangleright \{\alpha; \beta; \{\gamma; \}\}$

3: *sentence* $\rightarrow$ *formula* `;` $\triangleright \alpha;$

4: *formula* $\rightarrow$ `true` $\triangleright \top$
5: | `false` $\triangleright \bot$
6: | `VARIABLE`
7: | `[` *formula* `]`
8: | `(` *formula* `)`
9: | `not` *formula* $\triangleright \neg\alpha$
10: | *formula* `and` *formula* $\triangleright \alpha \wedge \beta$
11: | *formula* `or` *formula* $\triangleright \alpha \vee \beta$
12: | *formula* `=>` *formula* $\triangleright \alpha \supset \beta$
13: | *formula* `<=` *formula* $\triangleright \alpha \subset \beta$
14: | *formula* `<=>` *formula* $\triangleright \alpha \equiv \beta$

1.2 Symbols

Keywords: `true` ($\top$), `false` ($\bot$), `and` ($\wedge$), `or` ($\vee$), `not` ($\neg$).

User defined symbols:

Symbol	Syntax	Example
VARIABLE	`[a-zA-Z][A-Za-z0-9_]`	`H2O, k2, sun`

1.3 Operators

Listed from highest to lowest precedence and equal precedence listed together.

Arity	Operators	Assoc.
unary	`not`, $\sim$, $\neg$	right
binary	`and`, $\wedge$	left
binary	`or`, $\vee$	left
binary	`=>`, $\supset$	right
binary	`<=`, $\subset$	left
binary	`<=>`, $\equiv$	right

1.4 Example

```
# This is a line comment.
# Statements end with a semicolon;

# operator associativity

[not not P] <=> [not (not P)] ;
[P and Q and R] <=> [(P and Q) and R] ;
[P or Q or R] <=> [(P or Q) or R] ;
[P => Q => R] <=> [P => (Q => R)] ;
[P <= Q <= R] <=> [(P <= Q) <= R] ;
[P <=> Q <=> R] <=> [P <=> (Q <=> R)] ;

# operator precedence

[not P and Q] <=> [(not P) and Q] ;
[P or Q and R] <=> [P or (Q and R)] ;
[P => Q or R] <=> [P => (Q or R)] ;
[P => Q <= R] <=> [(P => Q) <= R] ;
[P <=> Q <= R] <=> [P <=> (Q <= R)] ;

# a block of statements

{ ~A or B;  ~B or ~C or D;  C; }
```

2 First Order Logic

2.1 Grammar

1: *program* → ⟨ *sentence* | *block*⟩∗

2: *block* → `{` ⟨ *sentence* | *block*⟩∗ `}` ▷ {α; β; {γ; }}

3: *sentence* → *formula* `;` ▷ α;

4: *formula* → `true` ▷ ⊤
5: | `false` ▷ ⊥
6: | *term* `<` *term* ▷ predicate
7: | *term* `=<` *term* ▷ predicate
8: | *term* `=` *term* ▷ predicate
9: | *term* `>=` *term* ▷ predicate
10: | *term* `>` *term* ▷ predicate
11: | *term* `<>` *term* ▷ predicate
12: | `NAME (` *terms* `)` ▷ predicate
13: | `forall` *vars formula* ▷ ∀x α
14: | `exists` *vars formula* ▷ ∃x α
15: | `exactlyone` *vars formula* ▷ ∃!x α
16: | `(` *formula* `)`
17: | `not` *formula* ▷ ¬α
18: | *formula* `and` *formula* ▷ α ∧ β
19: | *formula* `or` *formula* ▷ α ∨ β
20: | *formula* `=>` *formula* ▷ α ⊃ β
21: | *formula* `<=` *formula* ▷ α ⊂ β
22: | *formula* `<=>` *formula* ▷ α ≡ β

23: *vars* → `VARIABLE` ⟨ `,` `VARIABLE`⟩∗

24: *terms* → *term* ⟨ `,` *term*⟩∗
25: *term* → *expr* | *list* ▷ term

26: *expr* → `INTEGER` ▷ constant
27: | `FLOAT` ▷ constant
28: | `STRING` ▷ constant
29: | `CONSTANT` ▷ constant
30: | `NAME (` *terms*? `)` ▷ function
31: | `VARIABLE` ▷ variable
32: | `(` *expr* `)`
33: | `−` *expr*
34: | *expr* `*` *expr*
35: | *expr* `/` *expr*
36: | *expr* `%` *expr*
37: | *expr* `+` *expr*
38: | *expr* `−` *expr*

39: *list* → `[ ]` ▷ nil list
40: | `[` *terms* `]` ▷ list
41: | `[` *terms* `|` *list* `]` ▷ list
42: | `[` *terms* `|` `VARIABLE` `]` ▷ list

2.2 Symbols

Keywords: `true` (⊤), `false` (⊥), `forall` (∀), `exists` (∃), `exactlyone` (∃!), `and` (∧), `or` (∨), `not` (¬).

User defined symbols:

Symbol	Syntax	Example
NAME	`[a-z][A-Za-z0-9_]*`	likes, age
CONSTANT	`[a-z][A-Za-z0-9_]*`	anna, elsa
STRING	`'...'`	'WALL-E'
VARIABLE	`[A-Z][A-Za-z0-9_]*`	X, Y, Z

2.3 Operators

Listed from highest to lowest precedence and equal precedence listed together.

Arity	Operators	Assoc.
unary	−	right
binary	*, /, %	left
binary	+, −	left
binary	<, =<, =, >=, >, <>	N/A
unary	not, ∼, ¬, forall, ∀, exists, ∃, exactlyone, ∃!	right
binary	and, ∧	left
binary	or, ∨	left
binary	=>, ⊃	right
binary	<=, ⊂	left
binary	<=>, ≡	right

2.4 Example

```
# This is a line comment.
# Statements end with a semicolon;

(forall X,Y p(X,Y))
    <=> (forall X forall Y p(X,Y))
    <=> (forall X (forall Y p(X,Y))) ;

[1,2,3] = [1 | [2 | [3 | [] ] ] ] ;
[1,2,3] = [1,2 | [3 | [] ] ] ;
[1,2,3] = [1,2,3 | [] ] ;

(exists X p(X) and q(X))
    <=> ((exists X p(X)) and q(X)) ;

2+3-4*5/10 = (2+3)-((4*5)/10) ;

{
  forall X ( man(X) => mortal(X) ) ;
  forall X ( mortal(X) <= man(X) ) ;
  true and not exists X ( p(X) and not q(x) ) ;
  forall X,Y,Z ( p(X,Y) and p(Y,Z) => p(X,Z) ) ;
}
```

3 Horn Clauses

Horn clause syntax is borrowed from SWI Prolog, with tiny differences. And it is easy to convert horn clause programs into SWI Prolog programs.

3.1 Grammar

1: *program* → *hornClause*∗

2: *hornClause* → *predicate* `:-` *body* `.` ▷ rule
3: | *predicate* `.` ▷ fact
4: | *body* `?` ▷ query

5: *predicate* → `NAME` `(` *terms* `)`

6: *body* → *subgoal* ⟨ `,` *subgoal*⟩∗
7: *subgoal* → `!` ▷ cut operator
8: | *literal*
9: | ⟨∼ | `not` | `\+`⟩ *literal*

10: *literal* → `true` | `false`
11: | *term* `<` *term* ▷ eval., compare
12: | *term* `=<` *term* ▷ eval., compare
13: | *term* `>=` *term* ▷ eval., compare
14: | *term* `>` *term* ▷ eval., compare
15: | *term* `is` *term* ▷ eval., compare
16: | *term* `=:=` *term* ▷ eval., compare
17: | *term* `=\=` *term* ▷ eval., compare
18: | *term* `=` *term* ▷ unifies with
19: | *term* `\=` *term* ▷ not unifies with
20: | *term* `==` *term* ▷ equivalent to
21: | *term* `\==` *term* ▷ not equivalent to
22: | *predicate*
23: | `(` *literal* `)`

24: *terms* → *term* ⟨ `,` *term*⟩∗
25: *term* → *expr* | *list*

26: *expr* → `INTEGER` ▷ constant
27: | `FLOAT` ▷ constant
28: | `STRING` ▷ constant
29: | `CONSTANT` ▷ constant
30: | `NAME` `(` *terms*? `)` ▷ function
31: | `VARIABLE` ▷ variable
32: | `(` *expr* `)`
33: | `−` *expr*
34: | *expr* `*` *expr*
35: | *expr* `/` *expr*
36: | *expr* `%` *expr*
37: | *expr* `+` *expr*
38: | *expr* `−` *expr*

39: *list* → `[` `]` ▷ nil list
40: | `[` *terms* `]` ▷ list
41: | `[` *terms* `|` *list* `]` ▷ list
42: | `[` *terms* `|` `VARIABLE` `]` ▷ list

3.2 Symbols

Keywords: `true`, `false`, `not`, `is`.

User defined symbols:

Symbol	Syntax	Example
NAME	[a-z][A-Za-z0-9_]∗	likes, age
CONSTANT	[a-z][A-Za-z0-9_]∗	anna, elsa
STRING	'...'	'WALL-E'
VARIABLE	[A-Z][A-Za-z0-9_]∗	A, B, X, Y

3.3 Operators

Listed from highest to lowest precedence and equal precedence listed together.

Arity	Operators	Assoc.
unary	−	right
binary	*, /, %	left
binary	+, −	left
binary	<, =<, >=, >, is, =:=, =\=, =, \=, ==, \==	N/A
unary	∼, not, \+	right

3.4 Example

```
# This is a line comment.
# Statements end with a dot.

# append([1], [2], [1,2]).

append([], B, B).
append([X|A], B, [X|C]) :- append(A, B, C).

append([1,2],[3,4],[1,2,3,4]) ?   # query
!, \+ append([1,2],[3,4],C)   ?   # query

parent(P,X)      :- mother(P,X).
parent(P,X)      :- father(P,X).
grandparent(G,X) :- parent(G,P), parent(P,X).

cousin(X,Y)  :- X \= Y,
                \+ sibling(X,Y),
                grandparent(Z,X),
                grandparent(Z,Y).

americanCousin(X,Y) :- cousin(X,Y), !,
                       american(X).

composite(N) :- N > 1, ~ prime(N).
composite(N) :- N > 1, \+ prime(N).
composite(N) :- N > 1, not (prime(N)).
```

4 Production Systems

The working-memory-elements (WMEs) and rules follow the syntax given in "Knowledge Representation and Reasoning" by Brachman and Levesque. We have made some enhancements to make it expressive. And it allows `insert` as an alias for `add` action.

4.1 Grammar

1: *program* → ⟨ *wme* | *rule* ⟩∗

2: *wme* → `(` `TYPE` *attrSpec*∗ `)` `;`
3: *attrSpec* → `ATTRIBUTE` `:` *evalExpr*

4: *rule* → `if` *condition*+ `then` *action*+ `;`
5: *condition* → *wmeTest* | `–` *wmeTest*
6: *action* → ⟨ `add` | `insert` ⟩ *wme*
7: | `remove` `INTEGER`
8: | `modify` `INTEGER` `(` *attrSpec*∗ `)`

9: *wmeTest* → `(` `TYPE` ⟨ `ATTRIBUTE` `:` *testSpec* ⟩∗ `)`

10: *testSpec* → `null` ▷ undefined attribute
11: | `_` ▷ defined attribute
12: | *evalExpr testExpr*?
13: | *testExpr*

14: *testExpr* → `{` *test* `}` ▷ enclose in curly braces
15: *test* → *cOP evalExpr* ▷ relational
16: | `(` *test* `)` ▷ boolean
17: | `not` *test* ▷ boolean
18: | *test bOP test* ▷ boolean

19: *evalExpr* → *atom*
20: | `[` *expr* `]` ▷ enclose in sq. brackets

21: *expr* → *atom*
22: | `(` *expr* `)`
23: | `–` *expr* ▷ arithmetic
24: | `not` *expr* ▷ boolean
25: | *expr aOP expr* ▷ arithmetic
26: | *expr cOP expr* ▷ relational
27: | *expr bOP expr* ▷ boolean

28: *atom* → `true` ▷ constant
29: | `false` ▷ constant
30: | `INTEGER` ▷ constant
31: | `FLOAT` ▷ constant
32: | `STRING` ▷ constant: 'John'
33: | `CONSTANT` ▷ constant: john
34: | `VARIABLE` ▷ variable: X, Y

35: *aOP* → `*` | `/` | `%` | `+` | `–` ▷ arithmetic
36: *cOP* → `<` | `=<` | `=` | `>=` | `>` | `<>` ▷ relational
37: *bOP* → `and` | `or` ▷ boolean

4.2 Symbols

Keywords: `if`, `then`, `add`, `insert`, `modify`, `remove`, `true`, `false`, `and`, `or`, `not`, `null`, '`_`'; where a `null` denotes an undefined attribute, and an underscore '`_`' denotes a defined attribute.

User defined symbols:

Symbol	Syntax	Example
`TYPE`	`[a-z][A-Za-z0-9_-]*`	`car, bus`
`ATTRIBUTE`	`[a-z][A-Za-z0-9_-]*`	`hue, size`
`CONSTANT`	`[a-z][A-Za-z0-9_-]*`	`red, big`
`STRING`	`'...'`	`'WALL-E'`
`VARIABLE`	`[A-Z][A-Za-z0-9_-]*`	`A, X, Y`

4.3 Operators

Listed from highest to lowest precedence and equal precedence listed together.

Arity	Operators	Assoc.
unary	`–`	right
binary	`*`, `/`, `%`	left
binary	`+`, `–`	left
binary	`<`, `=<`, `=`, `>=`, `>`, `<>`	N/A
unary	`~`, `not`	right
binary	`and`	left
binary	`or`	left

4.4 Example

```
# This is a line comment.
# Statements end with a semicolon;

(phone item:a15 color:red mem:[32+32] camera:2) ;
(stock item:a15 qty:10) ;

IF  (phone item: X
        color:  {=red or =blue}
        memory: {>= 32 and =< 128}
        cost:   MRP {=< 10 or >= 20}
        tax:    TAX {=< [MRP*5/100]}
        warranty: _         # defined attr.
        touchScreen: null  # undefined attr.
    )
    (stock     item:X qty:Q {> 0} )
    (delivery  item:X days: {not > 5} )
    (loan      item:X emi:E {< [(MRP-TAX)/10]})
   -(insurance item:X cost: {>= [E/2]} )
THEN
    REMOVE 1
    MODIFY 2 (qty: [Q - 1])
    ADD (cart  item:X  price:MRP  qty:1)
;
```

5 Description Logic: $\mathcal{DL}$

This is an implementation of the description logic language $\mathcal{DL}$ from "Knowledge Representation and Reasoning" by Brachman and Levesque.

5.1 Grammar

1: *program* → ⟨ *sentence* | *block*⟩∗

2: *block* → `{` ⟨ *sentence* | *block*⟩∗ `}` ▷ {α; β; {γ; }}

3: *sentence* → *concept* `=>` *concept* `;` ▷ $A \sqsubseteq B$;
4: | *concept* `=` *concept* `;` ▷ $A \doteq B$;
5: | *symbols* `->` *concept* `;` ▷ $a, b \rightarrow A$;

6: *symbols* → *symbol* ⟨ `,` *symbol*⟩∗

7: *symbol* → `STRING` | `CONSTANT`

8: *concept* → `NAME` ▷ atomic concept
9: | `[ fills` *role constant* `]`
10: | `[ all` *role concept* `]`
11: | `[ exists INTEGER` *role* `]`
12: | `[ and` *concept concept*+ `]`

13: *role* → `:NAME`

14: *constant* → `INTEGER`
15: | `FLOAT`
16: | `STRING`
17: | `CONSTANT`
18: | `(` *constant* `)`
19: | `−` *constant*
20: | *constant aOP constant*

21: *aOP* → `*` | `/` | `%` | `+` | `−`

5.2 Symbols

Keywords: `fills`, `all`, `exists`, `and`

User defined symbols:

Symbol	Syntax	Example
NAME	[A-Z][A-Za-z0-9_-]∗	Man, Mortal
CONSTANT	[a-z][A-Za-z0-9_-]∗	anna, elsa
STRING	'...'	'WALL-E'

5.3 Operators

Listed from highest to lowest precedence and equal precedence listed together.

Arity	Operators	Assoc.
unary	−	right
binary	∗, /, %	left
binary	+, −	left
binary	=>, =, ->, ⊑, ≐, →	N/A

5.4 Example

```
# This is a line comment.
# Statements end with a semicolon;

Surgeon => Doctor ;

BlendedRedWine =
     [AND Wine
          [FILLS :Color red]
          [EXISTS 2 :GrapeType]
     ] ;

# a block of statements

{
    ProgressiveCompany =
         [AND Company
              [EXISTS 7 :Director]
              [ALL :Manager
                   [AND Woman
                        [FILLS :Degree phD] ] ]
              [FILLS :MinSalary 24.00/hour]
         ] ;

    joe -> Person ;

    canCorp ->
         [AND Company
              [ALL :Manager Canadian]
              [FILLS :Manager joe]
         ] ;
}
```

6 Subset of OWL 2

This is a subset of OWL 2, it follows OWL Manchester Syntax[1] from "OWL 2 Web Ontology Language Manchester Syntax (Second Edition) 11 Dec. 2012"[2].

In the grammar[3], the meta rule $\alpha LIST$ denotes a comma separated list of α values, similarly, $\alpha 2LIST$ denotes a list of two or more α values.

6.1 Grammar

kb → *frame*∗

frame → *classFrame*
| *roleFrame*
| *individualFrame*
| `EquivalentClasses:` *description2LIST*
| `DisjointClasses:` *description2LIST*
| `EquivalentProperties:` *role2LIST*
| `DisjointProperties:` *role2LIST*
| `SameIndividual:` *individual2LIST*
| `DifferentIndividuals:` *individual2LIST*

classFrame → `Class: NAME`
⟨ `SubClassOf:` *descriptionLIST*
| `EquivalentTo:` *descriptionLIST*
| `DisjointWith:` *descriptionLIST*
| `DisjointUnionOf:` *description2LIST*
⟩∗

roleFrame → `ObjectProperty: NAME`
⟨ `Domain:` *descriptionLIST*
| `Range:` *descriptionLIST*
| `Characteristics:` *characteristicLIST*
| `SubPropertyOf:` *roleExprLIST*
| `EquivalentTo:` *roleExprLIST*
| `DisjointWith:` *roleExprLIST*
| `InverseOf:` *roleExprLIST*
| `SubPropertyChain:` *roleChain*
⟩∗

roleExpr → `ROLE` | `inverse ROLE`

roleChain → *roleExpr* ⟨`o` *roleExpr*⟩+

characteristic → `Functional`
| `InverseFunctional`
| `Reflexive`
| `Irreflexive`
| `Symmetric`
| `Asymmetric`
| `Transitive`

individualFrame → `Individual: NAME`
⟨ `Types:` *descriptionLIST*
| `Facts:` *factLIST*
| `SameAs:` *individualLIST*
| `DifferentFrom:` *individualLIST*
⟩∗

fact → `ROLE INDIVIDUAL`
| `not ROLE INDIVIDUAL`

description → `CONCEPT`
| *roleExpr* `only` *description*
| *roleExpr* `some` *description*
| *roleExpr* `value` *description*
| *roleExpr* `min INTEGER` *description*
| *roleExpr* `max INTEGER` *description*
| *roleExpr* `exactly INTEGER` *description*
| `not` *description*
| `(` *description* `)`
| *description* `and` *description*
| *description* `or` *description*

6.2 Symbols

Keywords (in OWL Manchester Syntax) are **case sensitive**: `not`, `and`, `or`, `inverse`, `only`, `some`, `min`, `max`, `exactly`, `value`, `o`, `Functional`, `InverseFunctional`, `Reflexive`, `Irreflexive`, `Symmetric`, `Asymmetric`, `Transitive`, `Prefix:`, `Ontology:`, `Class:`, `SubClassOf:`, `EquivalentTo:`, `DisjointWith:`, `DisjointUnionOf:`, `ObjectProperty:`, `Characteristics:`, `Domain:`, `Range:`, `SubPropertyOf:`, `InverseOf:`, `SubPropertyChain:`, `Individual:`, `Types:`, `Facts:`, `SameAs:`, `DifferentFrom:`, `EquivalentClasses:`, `DisjointClasses:`, `EquivalentProperties:`, `DisjointProperties:`, `SameIndividual:`, `DifferentIndividuals:`.

User defined symbols:

Symbol	Syntax	Example
`NAME`	`[a-zA-Z][A-Za-z0-9_-]*`	`owns, Car`
`CONCEPT`	`NAME`	`Car, Bus`
`ROLE`	`NAME`	`owns, eats`
`INDIVIDUAL`	`NAME`	`Lucy, Jack`

6.3 Operators

Operators from highest to lowest precedence.

Arity	Operators	Assoc.
`unary`	`not`	`N/A`
`binary`	`and`	`left`
`binary`	`or`	`left`

[1] With suitable `Prefix:` and `Ontology:` entries, these files can be opened in Protégé ontology management tool.

[2] `https://www.w3.org/TR/owl2-manchester-syntax/`

[3] A frame is a block of statements, it differs from Frames discussed in Brachman and Levesque, but has similar syntax.

6.4 Example

```
# This is a line comment.

Class: Person
    SubClassOf:      eats some Fruit
    EquivalentTo:    Human
    DisjointWith:    Fruit, Meat
    DisjointUnionOf: Man, Woman

Class: TOAD
    EquivalentTo: Teen and owns some Apple
    SubClassOf:   Happy

ObjectProperty: hasChild
  Domain: Person
  Range:  Person
  InverseOf:  hasParent

ObjectProperty: hasSibling
  Characteristics: Symmetric
  Domain: Person
  Range:  Person

ObjectProperty: hasBrother
  Domain: Person
  Range:  Man
  SubPropertyOf: hasSibling

ObjectProperty: hasSister
  Domain: Person
  Range:  Woman
  SubPropertyOf: hasSibling

Individual: Lucy
  Types:  Woman, hasChild only Woman
  Facts:  hasHusband Manny, not owns Car157
  SameAs: SmartLucy
  DifferentFrom: Manny, Car157

EquivalentClasses: Dead, not Alive
DisjointClasses:   Fruit, Meat

EquivalentProperties: owns, hasOwnershipOf
DisjointProperties: hasBrother, hasSister

SameIndividual: Manny, LazyManny
DifferentIndividuals: Manny, Diego, Sid, Lucy
```

6.5 Note

The subset of OWL 2 that we implemented is a superset of $\mathcal{ALC}$, for example, the subset supports min/max cardinality restrictions, inverse roles, etc., that are not supported in $\mathcal{ALC}$. While testing $\mathcal{ALC}$ Tableau use only $\mathcal{ALC}$ constructs discussed in the lecture.

6.6 $\mathcal{ALC}$ Examples

Persons who do not own a car.

$\text{Person} \sqcap \neg\exists\text{owns.Car}$

```
Person and not (owns some Car)
```

Those who do not travel by bus or train.

$\neg\exists\text{travelsBy.}(\text{Bus} \sqcup \text{Car})$

```
not (travelsBy some (Bus or Train))
```

$\forall\text{travelsBy.}\neg(\text{Bus} \sqcup \text{Car})$

```
travelsBy only not (Bus or Train)
```

Owns a thing that has battery.

$\exists\text{owns.}(\exists\text{hasPart.Battery})$

```
owns some (hasPart some Battery)
```

Those with friends who own only electric cars.

$\exists\text{hasFriend.}(\forall\text{owns.}(\text{Electric} \sqcap \text{Car}))$

```
hasFriend some (owns only (Electric and Car))
```

Lucy is a mother and an engineer, she works for Acme Co. Her brother Jack is a doctor and he owns a car.

$\text{Mother}(\text{Lucy}),\ \ \text{Engineer}(\text{Lucy}),$

$\text{worksFor}(\text{Lucy}, \text{AcmeCo}),\ \ \text{hasBrother}(\text{Lucy}, \text{Jack}),$

$\text{Doctor}(\text{Jack}),\ \ (\exists\text{owns.Car})(\text{Jack})$

```
Individual: Lucy
  Types: Mother, Engineer
  Facts: worksFor AcmeCo, hasBrother Jack

Individual: Jack
  Types: Doctor, owns some Car
```

Domain of hasBrother is Person and range is Man.

$\exists\text{hasBrother.}\top \sqsubseteq \text{Person}$ (domain axiom)

$\top \sqsubseteq \forall\text{hasBrother.Man}$ (range axiom)

```
ObjectProperty: hasBrother
  Domain: Person
  Range:  Man
```