npl Documentation

Release 1.0a1.dev

Enrique Pérez Arnaud

October 16, 2013

CONTENTS

npl is a programming language for building expert systems. It is very simple, yet extremelly powerful. It is distributed under the GNU General Public License V3.

Like other expert system languages, **npl** deals with rules and facts. Various facts can make a rule applicable. An applicable rule is then asserted. As an example, a program that applies Newton's third law of motion to interpersonal relations:

```
person are thing.
mike isa person.
sue isa person.
loves isa verb.
mike loves sue.
if: Person1 Verb1 Person2;
then: Person2 Verb1 Person1.
```

The elements `are`, `thing`, `isa`, `verb`, and `if: ... then: ...` are primitives of the language. And, with this, **npl** would conclude that `sue loves mike` [1].

[1] For clarity, in these examples I have removed a little bit of necessary syntax, that is only needed to allow for more complex developments.

CHAPTER

ONE

THE NPL LANGUAGE

In the use of **npl** there are basically 4 stages:

- Define *terms*;
- Build facts and rules with those terms, and add them to a *knowledge base* (a.k.a. kb);
- Extend the kb to all logical consequences;
- Query the kb.

In the example in the *home page*, the first four lines are defining terms (`person`, `mike`, `sue`, `loves`) using predefined terms (`thing`, `verb`), and the fifth to seventh lines are a *fact* and a *rule* built from those terms. If we extend the kb holding the program, and query it, we will find the, in this case, only conclussion: `sue loves mike`.

1.1 Basic programming elements

npl provides three basic elements for writing programs: atomic elements, complex elements for adding to the kb, and orders (commands) to manipulate the kb in different ways.

We also have macros, but macros are another language built on top of **npl**, and we will consider them *elsewhere*.

1.1.1 atomic elements

There are several types of atomic elements with which to build complex elements. The first distinction we make is among logical and non-logical elements, (following the terminology of first order logic). Logical elements are all predefined, identified by reserved words of the language. Non-logical elements are defined by the programmer.

Logical elements

Among the logical elements, we can distinguish several classes:

- Punctuation: dots separate sentences, commas are variously used, and spacing separate elements;
- Logical connectives: conjunction (`;`) and implication (`if: ... then: ...`), used to build rules, that are a class of complex element;
- Variables, again used in the construction of rules;
- Relations, mainly set and arithmetic relations; the set relations used to define non-logical elements, and the arithmetic relations as conditions in rules;

- Operators, mainly the predicate operator `[ ... ]`, which we will shortly touch again, and arithmetic operators;
- Individuals: the numbers, and a few others like `noun`, `verb`, `thing`, `exists`, `number`.

This is not an exhaustive listing of predefined atomic elements, for example we have not mentioned any time (or string?) related element. It is, however, an exhaustive list of classes of atomic elements.

Non-logical elements

The non-logical elements are all of the "individual" class, which we will in general call "terms". The main atribute of terms is that they can match variables in rules.

We can distinguish a few types of terms. We have verbs: `exists` and any number of non-logical verbs derived from it; nouns: `thing`, and any number of non-logical nouns derived from it; proper names, all non-logical; and we have numbers. We also have the "metanouns": `noun`, `verb`, and `number`.

1.1.2 Complex elements

There are a few classes of complex elements available to give shape to the knowledge in the kb.

Definitions

First, we have constructs that allow us to define new non-logical terms. There are three types of term that we can define: verbs, nouns, and names. For each type we have a different construct. An example of a name definition might be `john isa person`, where we define a term `john`, and assert (through `isa`) that it "belongs to" the term (or "is a") `person`, of type noun.

Predicates

Then we have predicate elements. Predicates are composed with the predicate operator `[]`, enclosing a verb and 0 or more objects. These objects can be terms of any type. Predicates are themselves terms, and can thus match variables in rules, or appear as objects in other predicates. In the example in the home page, a predicate would be `loves sue`. This would be missing the `[]` operator, that is needed for when predicates are used as objects in other predicates; so a more correct form would be `[loves sue]`. But this is not yet totally correct syntax. We can have more than one object in a predicate, and, in different facts, the same verb can form predicates with a different number of objects. Thus, we need to label the objects. To see this, consider that we have a "goes" verb that takes 2 objects to form a predicate, a "from" object and a "towards" object. And consider that we want to use this verb to form a fact where we only talk about where someone goes, and not where she comes from. Using "[goes madrid]" would not tell us whether "madrid" is a "towards" or a "from" object. So, the syntactically correct form would be `[loves who sue]`. Obviously, we would have to incorporate the `who` label in the definition of the `loves` verb, and in the rule.

Facts

Then we have fact elements. Facts are composed of a subject, a predicate, and an optional time. The subject can be any atomic term, and the predicate has to be a predicate term. We will leave time out of this introduction. In the example in the home page, a fact would be `mike loves sue`. Incorporating the corrections explained for predicates, we would have `mike [loves who sue]`.

Rules

Then we have rule elements. Rules are composed of a set of complex elements that we call the conditions, and another set of complex elements, the consecuences. Conditions and consecuences can contain variables, and all variables in the consecuences must appear in the conditions. They can be definitions, facts, arithmetic conditions, or several other special constructs. When we extend the kb, asserted facts match conditions in rules, and when all conditions in a rule are satisfied, its consecuences are asserted.

Questions

Then we have questions, that are basically correspond to standalone rule conditions, and that return the sentences in the kb that match them.

Orders

Finally, we have orders, that do not have any common form. An example of an order is `extend`, that extends the kb.

1.2 Language reference

Next I'm going to describe the **npl** language in a bit more detail going through its BNF grammar. If in the previous introduction I have gone from bottom to top element, here I will go top down. To illustrate the different constructs of the language, I will be refering to pieces from these tests.

1.2.1 language essentials

Sentences

```
sentence  ::=   statement
              | question
              | order
```

Sentences are the top level grammatical elements. These are the elements that can be entered (told or asked) into npl's knowledge base (kb). There are 3 kinds of sentences: statements, questions, and orders. Statements add to the information held in the kb, and questions query the kb. Orders do different things to the kb.

Order: extend

```
order   ::=   EXTEND DOT
EXTEND  ::=   "extend"
DOT     ::=   "."
```

If you issue the order `extend.`, the kb will be extended to all its logical consecuences.

Order: import

```
order   ::=   IMPORT URI DOT
IMPORT  ::=   "import"
URI     ::=   '"' <An URI> '"'
```

If you issue the order `import "http://example.com/ont.npl".`, the kb will download the file at the URI and load it into the kb.

Statements

```
statement  ::=   definition DOT
               | fact DOT
               | rule DOT
```

Statements are asserted: they have truth value. There are 3 main forms of statements, all ended with a dot, which we will call definitions, facts, and rules. Definitions are used to define terms. Facts use those terms to establish relations among them. And rules establish relations (logical relations, basically implication) among the relations expressed in the facts.

Definition of terms

```
definition  ::=   noun_def
                | name_def
                | verb_def
```

There are 3 types of terms we can define. We will call them (proper) names, nouns, and verbs.

Noun definitions

```
noun_def  ::=   TERM ARE TERM
ARE       ::=   "are"
TERM      ::=   "[a-z][a-z_]*\d*"
```

We define a new noun by relating it with another (already defined) noun through the reserved word `are`. To get started, we use a primitive predefined noun, `thing`. An example of a noun definition is `person are thing.`.

The symbols for atomic terms in general must start with a lower case letter followed by any number of lower case letters or underscores, and 2 underscores in a row are forbidden.

For more examples, you can look at lines 20-24 in this test program.

Name definitions

```
name_def  ::=   TERM ISA TERM
ISA       ::=   "isa"
```

Proper names are defined relating them with a noun through the reserved word `isa`. An example is `john isa person.`.

Name terms are special among the rest of terms in that they may end in any number of digits. The rest of terms must be terminated with an lower case letter.

For more examples, you can look at lines 26-35 in the test program.

Types of terms

Names and nouns establish a class structure. The relation established by `are` among 2 nouns has the same form as the subclass relation among 2 classes, and the relation established by `isa` among a name and a noun has the same form as the relation between an individual and a class it belongs to. So, for example, the mentioned definitions in the the test program entail that `document are thing`, or that `mary isa thing`. Also, this means that if we ask the system for a `thing`, `mary` will be retrieved, and if in a rule we require a `thing`, `mary` will match.

This class structure is explicit in the case of nouns and names, and is (implicitly) pervasive among the rest of terms in **npl**. So, for example, all predicates (predicates are complex terms composed of a verb and any number of objects, as we shall see below) are implicitly related by `isa` with their verbs. In addition, all verbs are to be thought of as related through `isa` with the predefined term `verb`, and all nouns with `noun`, all numbers with `number`, and all times with `time`.

This allows us to talk about types of terms. A type of terms is a term, and the terms that are of that type are the terms related with the type term through `isa`. Therefore, we have six mayor types of term: `noun` (what we called "nouns" when introducing the basic programming elements), `verb` (the verbs), `thing``(the proper names), ``exists` (which is the primitive predifined verb, and as type corresponds to the predicates), `number` (the numbers), and `time`, and any number of subtypes of `thing` (i.e., the programmer defined nouns) and `exists` (the programmer defined verbs). Metanouns would correspond to the hidden term `word`.

For example, `doc1` is a term of type `thing` (and also of type `document`), and `document` is a term of type `noun`.

NOTE: since the definitions of verbs set bounds on the predicates and facts where they can appear, we shall defer their introduction until we have introduced predicates and facts.

Facts

```
fact     ::=   subject predicate
subject  ::=   TERM
```

Facts are composed of a subject and a predicate. The subject is a name, a noun or a verb.

```
    predicate  ::=   LBRACK verb modification RBRACK
                     | LBRACK verb RBRACK
    verb       ::=   TERM
    LBRACK     ::=   "["
    RBRACK     ::=   "]"
```

The predicate is a complex term enclosed in square brackets, composed of a verb and an (optional) modification.

```
    modification  ::=   object COMMA modification
                        | object
    COMMA         ::=   ","
```

A modification is one or more objects, separated by commas.

```
object  ::=   LABEL object
object  ::=   TERM
              | predicate
LABEL   ::=   <same pattern as TERM>
```

A object is composed of a label and an object, that can be any kind of (atomic or complex) term except a time: a noun, a verb, a name, a number, or a predicate.

A simple example of a fact could be `john [view what img1]`, where `john` is the subject and `[view what img1]` the predicate, where `view` is the verb, and `img1` is a object with label `what`.

Definition of verbs

```
verb_def  ::=   A TERM CAN TERM LPAREN verbs RPAREN modification_def
                | A TERM CAN TERM modification_def
                | A TERM CAN TERM LPAREN verbs RPAREN
verbs     ::=   verb COMMA verbs
                | verb
CAN       ::=   "can"
A         ::=   "a"
```

In the definition of a verb (with name given as the second TERM in the verb_def) we can specify 3 different things. First, the type of term that can act as subject in a fact where the new verb forms the predicate (given by the first TERM in the definition); second, the (already defined) verb(s) from which we derive the new verb (given in the verbs part of the definition); and third, the objects that the verb can take to form the predicate (the modification_def). Both the verbs part or the modification_def part can be omitted. Omitting the verbs, we assume its parent to be `exists`; omittin the modification_def, the verb will inherit those of its parents.

```
modification_def  ::=   mod_def COMMA modification_def
                        | mod_def
mod_def           ::=   LABEL A TERM
```

The objects that a verb can take are specified through mod_defs, where we give the label that the object will take, connected through the reserved word `a` with the type of terms that can be used as that object.

So, for example, in lines 37-40 in the test program, we define verbs that express actions that a person can perform on content. For this we use the primitive predefined verb we mentioned earlier: `exists`.

Derived verbs inherit the mod_defs that they do not override. Therefore, we do not need to specify a mod_def for a child verb if it coincides with one of its parents.

With these verbs, we can state facts such as `pete [owns what doc1].` or `sue [edit what img2].`

Rules

```
rule          ::=   IF COLON conditions SEMICOLON THEN COLON consecuences
conditions    ::=   conditions SEMICOLON condition
                    | condition
condition     ::=   fact
                    | name_def
consecuences  ::=   consecuences SEMICOLON consecuence
                    | consecuence
consecuence   ::=   fact
IF            ::=   "if"
COLON         ::=   ":"
SEMICOLON     ::=   ";"
THEN          ::=   "then"
```

A rule consists of 2 sets of statements, the conditions and the consecuences. Conditions and consecuences are, mainly, facts (though they can be other types of statements, as we shall be seeing below). Atomic facts (facts that are asserted on their own, outside of rules) can match the conditions of rules, and, when all conditions in a rule are matched, its consecuences are (atomically) added to the kb when we issue an `extend.` order.

An atomic fact matches a condition in a rule if (but not only if) they are identical (ignoring the order of objects in the predicate). It also matches when they are identical except that the atomic fact specifies more objects than the condition. Also, the order of the objects is immaterial for matching.

We can use logical variables in place of terms in the conditions and consecuences of a rule. A logical variable is a symbol that starts with a capital letter, followed by any number of lower case letters, digits, and underscores, and ends with any number of digits. For example, `Person1`. A logical variable has a range, that is a type of terms. The range of a variable can be obtained by lower casing its first letter and removing its final digits. A fact will match the condition of a rule if they are identical except that, where the condition has a variable, the fact has a term that is in the range of the variable. The scope of variables is the rule: if a term matches a variable, it does so for all its occurrences within the rule.

For a first example, we need to add a couple more of BNF rules:

```
subject  ::=   VAR
object   ::=   VAR
VAR      ::=   "([A-Z][a-z_]*)(Verb|Noun|Word)?\d+"
```

So, for example, in line 42 in the test program we define a verb `located`, which we use in a rule in line 44.

With this rule, and the facts in lines 51 and 52, the system will conclude that `doc1 [located where ctx2].`

Although we are seeing different types of variables corresponding to the different types of terms, under the hood there is really just one type of variable. The different forms of variables is just syntactic sugar for constraining the variable. For example, `Person1` would correspond to something like "X1 where X1 isa person".

Predicate variables

```
predicate  ::=   LBRACK VAR RBRACK
```

We have mentioned that we can use predicates as objects in the objects of other predicates. This means that, in rules, we must be able to use variables that range over predicates. We do this by building a variable from a verb, and enclosing it in square brackets. For example, from `locate`, we might have `[Locate1]` (the brackets are not part of the variable, but mark it as a predicate).

To provide a working example, we define a couple of verbs that take a predicate as object, in lines 60 and 61 in the test program, and build a rule with them in line 63.

With this rule, and the facts in lines 70-71, the system will conclude that `sue [view what doc1].`

Verb variables

```
predicate  ::=   LBRACK VAR VAR RBRACK
                 | LBRACK VAR modification RBRACK
```

Since we can have verbs as subject or object in facts, we need to be able to use variables in rules that range over verbs. We do this by capitalizing the name of a verb, and appending to it "Verb" and an integer. for example, a verb variable made from `locate` would be `LocateVerb1`. To show a more complete example of this, we define a verb `may` in line 79 in the test program, that will take a verb as object, and a rule that uses `may` in line 84. Now, if we add the facts in lines 92, 93, the system will conclude that `mary [view what doc1].`

So, as seen in line 85, we can use a verb variable in a predicate with objects. Also without objects, just by itself in the predicate, like `[Content_actionVerb1]`. This stands for a predicate where the content_action verb is alone without objects, as opposed to `[Content_action1]` where nothing is said of the number of objects.

If, in the rule in line 84, we had not wanted to relate the context in which the content is located with the context in which the person is allowed to do the content action, we might have said:

```
if:
    Person1 [wants that Person1, do [Content_actionVerb1 Content_action1]];
    Person1 [may what Content_actionVerb1];
then:
    Person1 [Content_action1].
```

Let's take a look at the construct `[Content_actionVerb1 Content_action1]`. It stands for a predicate, and any predicate matching it would also match `[Content_action1]`. However, we want to specify that the matching predicate's verb must be the one that matches the variable `Content_actionVerb1` in the second condition. Thus the oddly redundant form.

Noun variables

```
subject  ::=   varvar
object   ::=   varvar
varvar   ::=   VAR LPAREN VAR RPAREN
LPAREN   ::=   "("
RPAREN   ::=   ")"
```

The same we have said about verb variables can be said of noun variables. The only difference is when, in a condition, we want a variable form to range over names that have a type given by another (noun) variable. In that case, we give the name variable inmediately followed by the noun variable enclosed in parentheses. For example, `Person1(PersonNoun1)`.

In the rule in line 151 there is an example of the use of noun variables.

1.2.2 Time

```
fact : subject predicate time
```

We can specify a time as a distinguished part of a fact. This time has the form of either an integer or a pair of integers. An integer marks a fact whose interpretation is an instantaneous happening, and a pair represents an interval of time, a duration.

For examples of all this, you can look at this npl test.

The reason we distinguish time (it would in principle suffice to represent times as just another modifier in the predicate) is because we want to allow for the present continuous (this is, for facts that have a starting instant but not an ending instant). To do this, we employ some non-monotonic technique. Now, the logic we have drawn up to this moment is strictly monotonic. And non-monotonicity scares the hell out of me. So, we isolate time in a reserved place and treat it very carefully, and make it optional.

Time can thus be given as an instant or as a duration. To assert facts, we can only use the present tense. We assume a closed world where everything is in known the instant it happens, i.e., we know everything about the past but nothing about the future, and are changing (learning) in the present.

Instants

```
time : NOW

order : NOW DOT

NOW : "now"
```

The time can be specified with the term `now`. We can say `sue [views what doc1] now.`.

Internally, every instance of **npl** keeps a record of time. When **npl** is started, this record is set to the UNIX time of the moment. It is kept like that till further notice. And further notice is given with the order `now.`. This order causes **npl** to update its internal record with the UNIX time of the moment. this internal record represents the 'present' time in the system.

When we say something like fact `sue [views what doc1] now.`, the time that is being stored for that fact is the content of the said 'present' record at the time of saying. So, if we say several facts with time `now` without changing the internal time with `now.`, they will all have the same time.

The `now` term is optional, and we might have just said `sue [views what doc1].`. If we do not specify a time, it is assumed to be `now`.

Durations

```
time : ONWARDS

ONWARDS : "onwards"
```

To build a duration, we can use the reserved word `onwards` as the time component. This will set the starting instant of the duration to the present, and will set a special value as the end of the duration. This value will stand for the 'present' time of the system, irrespectively of its changes. So, if the present time is 10, and we assert a fact `onwards`, both the starting and the final instant of its duration will evaluate to 10; but if we change the present (through `now.`) to 12, the starting instant will still evaluate to 10, whereas the end will evaluate to 12.

Time in conditions

```
time : VAR
     | AT VAR

AT : "at"
```

In conditions in rules, we can use, either an instant variable (like `at I1`), or a duration variable (like `D1`).

The `during` condition

```
condition : VAR DURING durations

DURING : "during"
```

We can build a special condition with `during`, where we give an instant variable and any number of duration variables, like `I1 during D1, D2, D3`. This condition will evaluate to true when the instant that matches `I1` is contained in the durations that match `D1`, `D2`, and `D3`.

Time in consecuences

```
time : SINCE instant TIL instant
     | SINCE instant UNTIL durations
     | SINCE instant ONWARDS
     | AT instant

instant : arith
        | VAR

durations : VAR COMMA durations
          | VAR

arith : NUMBER

SINCE : "since"

TIL : "till"

UNTIL : "until"
```

In consecuences in rules, we can use the same constructs as in conditions, and we can specify the starting and ending instants in any way we want. There is also a special construct for durations, in which we express the starting instant with an instant and the ending instant with the reserved word `until` followed by any number of duration variables (bound in the conditions of the rule): `since I1 until D1, D2, D3`. This will create an `onwards` duration that will be bound to the durations that have matched the duration variables specified, so that whenever any of them is terminated, the new one will also be terminated. If two rules produce the same consecuence, the system will do the right thing (require a condition of each to be terminated before terminating the consecuence).

Terminating the continuous present

```
consecuence : FINISH VAR

FINISH : "finish"
```

There is a special type of consecuence, built with the reserved word `finish`, that can be given as a consecuence in rules, like `finish D1;`. This sentence will change the special value of the final instant of `D1`, to replace it with the present. Terminating a duration will terminate all durations that are derived from it through the `until` operator.

1.2.3 Arithmetics

Here we will see the syntax for arithmetics in **npl**. An example program that uses arithmetics can be seen here.

Numbers

```
object : arith-obj
```

We can use numbers as modifiers in predicates. To do so, we must define the modifiers of verbs with the term `number`. At the moment, numbers are simply floats, but this may change.

```
a thing can has_position x a number, y a number.
```

With this, we can say something like `thing1 [has_position x 1, y 2].`.

Arithmetic operations

```
arith-obj : NUMBER
          | LCURL arith-operation RCURL

arith-operation : arith-operand arith-operator arith-operation
                | arith-operand arith-operator arith-operand

arith-operand : NUMBER
              | VAR
              | LPAREN arith-operation RPAREN

arith-operator : PLUS
               | MINUS
               | MULTIPLICATION
               | DIVISION

instant : arith-obj
```

We can use arithmetic operations in the consecuences of rules, both in place of number modifiers and in place of instants. We enclose the operations in outermost curly brackets, and any internal grouping id done with parentheses. The available operators are sum +, subtraction −, multiplication $\star$ and division /. For example, `{4 + (N1 - N2)}`.

Arithmetic conditions

```
condition : arith-condition

arith-condition : LCURL arith-predication RCURL

arith-predication : arith-operand arith-predicate arith-operand

arith-predicate : LT
                | GT
                | EQ
                | NEQ
```

We can specify arithmetic conditions. We do so enclosing the condition in curly brackets. The available predicates are less than <, greater than >, equals =, and not equals <>. For example `{ I1 < 33 };`. We can use instant variables in the condition, as well as number variables. We can also use arithmetic operations within arithmetic conditions.

1.2.4 Negation and counting sentences

We can use 2 forms of negation in **npl**. The first is classical negation, and the second negation by failure. In both cases we can only talk about negation of facts; neither predicates nor definitions can be negated.

Classical negation

```
assertion : NOT fact DOT

NOT : "not"
```

With classical negation, a negated fact is equivalent to a non-negated fact. You add a negated fact "not P" to the kb in the same way you add a non-negated fact "P". If you ask (in a query or in the condition of a rule) for a negated fact, you will only succeed if the negated fact is in the kb. The fundamental meaning of classical negation lies in contradiction: you cannot have both a non-negated fact and its negation in the kb. If you try to do so, the system will not accept it and warn you of the contradiction. And if 2 rules produce contradictory facts, the system will break.

Negation by failure

Negation by failure relies on a closed world assumption, i.e., on the assumption that all possible knowledge about the universe of discourse is present in the kb. All true facts are present in the kb, and all facts that are not present in the kb are false. So, if we ask a negated fact "not P", the system need not look for it; it can look for "P", and if it is not found, our query will succeed.

Counting facts

```
arith : COUNT fact

COUNT : "count"
```

So, negation by failure is basically a matter of counting facts. **npl** allows us to count facts in rules. If you count a fact like `count johnny [goes to hollywood] at 3`, you will obtain either 1 or 0. So, to set a condition for a negated (by failure) fact, you would add an arithmetic condition like `{count johnny [goes to hollywood] at 3 = 0};`.

We can also use variables in fact counts. For example, to count how many trips there have been to egipt, we would `count Person1 [trips to egipt] at I1`.

In **npl**, counting facts only makes sense if we use its time facilities, and use a further restriction: that we can only make one assertion per instant. Under the assumption of a timeless interpretation, the same set of facts must produce the same set of consecuences under any circumstances, whatever the order they are asserted. If we allow counting sentences in rules, the same set of facts will produce different sets of consecuences depending on the order in which they are added to the kb. However, if we (as we do in **npl** when we use the present continuous) assume that the past is closed, and the present, in which things change, only accepts one assertion at a time, the order in which we add the facts is part of the general truth, and the same set of facts added in different order would represent different realities.

NOTE: Counting sentences is implemented but not yet exposed in **npl**, because I'm not sure about the syntax.

1.2.5 Questions

```
sentence : question

question : fact QMARK
         | definition QMARK
```

You query **npl** with sentences and definitions ended with a question mark. You can use variables in questions.

In **npl**'s tests, you must follow every question with a (python) regular expression that matches the expected answer.

1.2.6 Macros

Macros are a way of reusing code. If you have several rules or facts that share a common form, you can put that common form in a macro definition, and provide the special parts as parameters when you call the macro.

for an example, see this npl test

the actual grammar is in this module

1.2.7 Glossary

fact Complex element of the language with which we assert *facts* about our universe of discourse. Composed of a subject, a predicate and (optionally) a time.

knowledge base A set of

rule Complex element of the language with which we assert *facts* about our universe of discourse.

term Atomic element of the language,

CHAPTER

TWO

TUTORIAL: A CONTENT MANAGEMENT SYSTEM

In this section we will try to develop a not so simple ontology that might be used as the metadata backend for a content management system (CMS). Let us imagine that such a CMS would basically consist of the following parts:

1. A web application: mainly an HTTP request dispatcher and a system of views that the dispatcher calls to get responses for the requests;
2. A database where actual content (text, images, etc.) is stored;
3. A user authentication backend;
4. The metadata backend that the views consult to produce responses.

The metadata backend would therefore know about content types, users, permissions, workflows, etc., and is the only part we will develop here.

2.1 Brief introduction to the anatomy of sentences in npl

There are 4 basic kinds of sentences in npl. The first kind (noun definitions) have the form "<noun1> are <noun2>", and are used to define nouns in terms of other nouns. Nouns are akin to classes, where proper names would be the elements of those classes. Defining a noun in that way would mean that all individuals (proper names) of noun1 are also of noun2. If we enter `dog are animal`, it will mean that all dogs are animals.

The second kind of sentence (name definitions) have the form "<name> isa <noun>", and are used to define proper names in terms of nouns. If we enter `bruto isa dog`, it will mean that Bruto is a dog (and, from the previous paragraph, that Bruto is an animal).

The third kind have the form "<subject> [<verb> <label> <object>(, <label2> <object2> ...)] <time>", and are used to assert facts. They have a subject, that can be any kind of term, and a predicate, enclosed in square brackets. The predicate is itself composed of a verb and any number of objects (or objects). The objects are composed of a label and another term, of any kind. Finally, we have a term of type time.

The fourth kind have the form "a <term> can <verb> <label> a <term>(, <label2> a <term2> ...)", and are used to define verbs in terms of the kind of subject and objects that they can take on facts.

For more details, see the language reference.

2.2 CMS

So let's start our CMS logic by defining a `person` noun:

```
person are thing.
```

Now we can define `person` individuals, identified by proper names of `person`. For example, we can name a `person` `john`:

```
john isa person.
```

We also need a "content" class of things. `content` will be the noun of all content objects, and the various content types (e.g., document) will be nouns derived from `content`:

```
content are thing.
document are content.
image are content.
```

Thus we can have content individuals:

```
img1 isa image.
```

And now, we will define verbs that refer to the different actions that people can perform with content objects (or individuals). First, we define an abstract `action` verb that will be the ancestor of any other action:

```
a person can action what a content.
```

The operational semantics of this would be that `action` is defined as a verb, that when used in a fact will require a `person` individual as subject of the fact, and that in such a fact, it may have an object or object (labelled `what`) of type `content` (i.e., a content object).

We can now define more verbs derived from `action`, that inherit its (just one) objects:

```
a person can view (action).
a person can edit (action).
```

Another verb that affects people that we are going to define is `wants`. Whenever a user attempts to perform an action on the system (calls a URL mapped to a document object, clicks an edit button, etc.), we will tell the metadata backend (npl) that the user wants to perform the action. A basic pattern for the metadata rules will be "if someone wants to do such thing, and this and that conditions are met, (s)he does such thing". Therefore, a basic pattern for the views will be to tell nl that a user wants to do something, extend the knowledge base, and then ask whether the user does it.:

```
a person can wants what a exists.
```

By using `exists` as the type of object, we are saying that the objects of `wants` must be predicates.

Now we define a fairly general verb, `has`, that can have anything as subject and can have two objects, `what`, that can be any thing, and `where`, that has to be a context:

```
context are thing.

a thing can has what a thing, where a context.
```

We will use `has` for various things. For example, to tell the system that a certain user has a certain role in some context, or that a certain role has a certain permission, or that a certain content object has some workflow state.

Next, let's define 2 nouns: `role` and `permission`. As we have hinted above, roles are related with permissions through the verb `has`:

```
permission are thing.
role are thing.
```

We could now define a few more proper names for our ontology:

```
member isa role.
editor isa role.
manager isa role.

basic_context isa context.

view_perm isa permission.
edit_perm isa permission.
manage_perm isa permission.

member [has what view_perm] onwards.
editor [has what view_perm] onwards.
editor [has what edit_perm] onwards.
manager [has what view_perm] onwards.
manager [has what edit_perm] onwards.
manager [has what manage_perm] onwards.
```

We can now assert that, for whichever context, the admin person has the manager role:

```
admin isa person.

if:
  Context1 isa context;
then:
  admin [has what manager, where Context1] onwards.
```

We now define a verb, `located`, that allows us to locate content objects in contexts:

```
a content can located where a context.
```

Next, we define a `status` noun, that refers to the different workflow states that a content object can be in. As a starting point, we shall define 2 different states, public and private:

```
status are thing.
public isa status.
private isa status.
```

We now define an abstract workflow action, that will be primitive to any workflow action:

```
a person can wfaction (action).
a person can publish (wfaction).
a person can hide (wfaction).
```

Now we define a `required` verb, that is used to state that a certain permission is required to perform a given action over any content that is in a certain workflow state. Note that in this case, we are using an actual verb, and not a predicate, as the object for the `required` verb: We define it with a `verb` object. For the moment, we can not set bounds to the possible verbs that can be used as objects for these verbs: we use `verb`, that is the only class we have for verbs:

```
a permision can required to a verb, over a status.
```

At this point, we can define a rule that, when someone wants to perform an action over some content, decides whether (s)he is allowed to perform it or not, according to her roles and to the workflow state of that content. We want to assert that, if someone wants to perform some action on some content, and that content has some state and is located in some context, and the person has some role in that context that has the required permission to perform that action over that workflow state, then (s)he performs it:

```
if:
  Person1 [wants to [ActionVerb1 what Content1]] at I1;
```

```
  Permission1 [required to ActionVerb1, over Status1] D1;
  Content1 [has what Status1] D2;
  Content1 [located where Context1] D3;
  Person1 [has what Role1, where Context1] D4;
  Role1 [has what Permission1] D5;
  I1 during D1, D2, D3, D4, D5;
then:
  Person1 [ActionVerb1 what Content1] at I1.
```

Note the use of the `ActionVerb1` verb variable to range over actual `action` verbs.

We can now protect some actions with permissions:

```
view_perm [required to view, over public] onwards.
edit_perm [required to edit, over public] onwards.
manage_perm [required to hide, over public] onwards.
manage_perm [required to view, over private] onwards.
manage_perm [required to edit, over private] onwards.
manage_perm [required to publish, over private] onwards.
```

Next, we are going to give meaning to workflow actions. For that, we are going to define a `workflow` noun, an `assigned` verb that will relate workflows to content types (depending on the context the content object is in), and another verb `has_transition` that relates a workflow with an initial and a final workflow state and the workflow action that performs the transition:

```
workflow are thing.

a workflow can assigned to a noun, where a context.
a workflow can has_transition start a status, end a status, by a verb.
```

With these terms in place, we can add a rule that states that, if some person performs some workflow action on some content, and that content is in the initial state of the transition corresponding to that action, and that action embodies the transition of some workflow that is assigned to the content type of the content object in the context in which the object is located, then the object ceases to be in the initial state and starts being in the final state of the transition:

```
if:
  Person1 [Wfaction1 what Content1] at I1;
  Workflow1 [has_transition start Status1, end Status2, by Wfaction1] D1;
  Workflow1 [assigned to ContentNoun1, where Context1] D2;
  Content1(ContentNoun1) [located where Context1] D3;
  Content1 [has what Status1] D4;
  I1 during D1, D2, D3, D4;
then:
  Content1 [has what Status2] until D1, D2, D3;
  finish D4.
```

So, let's provide a workflow for `document` and assign it to `document` in the basic context, and a couple of transitions for that workflow:

```
doc_workflow isa workflow.

doc_workflow [has_transition start private, end public, by publish] onwards.
doc_workflow [has_transition start public, end private, by hide] onwards.

doc_workflow [assigned to document, where basic_context] onwards.
```

With all this, we can start adding people and content objects, and test our ontology so far.

So, let's star using this ontology. We are going to define 2 contexts, 2 documents, one located in each context, both with an initial state private, and two people, each with the manager and editor role in opposite contexts:

```
john isa person.
mary isa person.

context_of_john isa context.
context_of_mary isa context.

doc_of_john isa document.
doc_of_mary isa document.
```

Let's start time:

```
now.

john [has what manager, where context_of_john] onwards.
john [has what editor, where context_of_mary] onwards.
mary [has what editor, where context_of_john] onwards.
mary [has what manager, where context_of_mary] onwards.
doc_of_john [located where context_of_john] onwards.
doc_of_john [has what private] onwards.
doc_of_mary [located where context_of_mary] onwards.
doc_of_mary [has what private] onwards.
```

We extend the knowledge base:

```
extend.
```

And now we can see that Mary cannot view or edit John's document, but john can:

```
mary [wants what [view what doc_of_john]] now.
mary [wants what [edit what doc_of_john]] now.
john [wants what [view what doc_of_john]] now.
john [wants what [edit what doc_of_john]] now.

extend.

mary [view what doc_of_john] now?
False

mary [edit what doc_of_john] now?
False

john [view what doc_of_john] now?
True

john [edit what doc_of_john] now?
True
```

Time passes:

```
now.
```

Mary cannot publish John's doc, but John can:

```
mary [wants what [publish what doc_of_john]] now.
john [wants what [publish what doc_of_john]] now.

extend.

mary [publish what doc_of_john] now?
False
```

```
john [publish what doc_of_john] now?
True
```

And, now, john's document is in the public state, and so, Mary can view it, but Mary's is private and John cannot view it:

```
doc_of_john [has what public] now?
True

mary [wants what [view what doc_of_john]] now.
john [wants what [view what doc_of_mary]] now.

extend.

mary [view what doc_of_john] now?
True

john [view what doc_of_mary] now?
False
```

Etc. etc.

CHAPTER

THREE

MOTIVATION

The **npl** language may be viewed as a proof of concept for the solution of a problem that has been present in logics since Gottlob Frege uncovered it. Here I will try to introduce this problem and my proposed solution.

Note that this "motivation" page is mainly for logicians. If you just want to use the language because it seems useful, you can (must!) leave this page and head on to the *reference* or the *tutorial*. And if you are a logician, please bear in mind that I am a self taught "logician", so you should not expect the orthodox academic terminology of any established line of thought.

3.1 Historical introduction

3.1.1 Frege

Predicate logic was first developed by Gottlob Frege. He developed it in the belief that he was unraveling the foundations of the natural logic behind scientific theories (or behind any unambiguous text expressed in a natural language) (see this). He pursued a mathematical formalism capable of expressing any informal scientific theory. To that end, he developed predicate logic and, at the same time and on top of it, a particular formal theory which we may call "theory of concepts". In this theory he had individuals (or values) and he had classes (or value-ranges), connected through class predicates (belongs, subclass). He also had an unlimited amount of other predicates (what he called concepts) (see here). With this, he very nearly had all the elements he needed to have a formal system with the same expressive power as the natural languages. With his class predicates he was able to represent the class relations in the natural languages, generally expressed in them through copular verbs; and with his concept/predicates he was able to represent the rest of relations expressed in the natural languages through non-copular verbs.

However, there was a problem with this scheme. His theory was second order, so he had 2 types of variables: those that range over individuals, and those that range over concepts. This, in itself, was not satisfactory. In the natural languages you do not have 2 types of variables, but just one, that can range over anything. I think that this can be easily seen with an example. Let us examine the sentence: "if Mary wants something, she gets it". I will take that this sentence is equivalent to "for all X, if mary wants X, mary gets X". So, we have constructs in the natural languages that behave very much like variables. And "to go to the cinema" can fall under X, so predicates can be in the range of variables. And also "an apple" can fall under the same X, which would correspond to an individual; so variables are first order, there is only one order of variables.

To bridge this gap, Frege had his Basic Law V, that basically established a correspondence between classes and concepts. (again, see here). This allowed him to unify the use of his 2 orders of variables. But along came Bertrand Russell, showing that this axiom leads to contradictions and must be dropped. This was a heavy blow for Frege, who felt that his project had failed, and never quite recovered from it.

His failure was however a very productive one, and predicate logic became a hugely successful technology. After Russell's antinomies, predicate logic developed in 2 separate but interconnected ways. On one hand, logicists, following the lead of Zermelo, developed a first order set theory with which they provided a foundation for most mathematics.

On the other hand, Russell, and the logical positivists, pursued the development of higher order logics with the original aim of Frege, of developing a formal system with the expressive power of the natural languages. They did not succeed in this effort.

3.1.2 First order logic

The problem encountered by Frege can also be seen in first order logic. To show this, we will imagine a first order set theory, with 3 predicates: "equals", "belongs to", and "is a subset of". In principle, this theory will have 3 axioms, of equality, extensionality, and definition of subsets. This axioms provide the predicates with the basic form of class/set relations, and can thus be put in correspondence with the natural usage of the copular verbs. We then have a formal system where we can express any scientific taxonomy, and reason about it. Obviously this is not enough to express the logic of any scientific theory; we need other predicates apart from copular ones, to represent other relations apart from that of belonging to classes.

First order logic allows us to use more predicates, of course. We can define new predicates and give them whatever form we like through additional axioms. Nevertheless, there is still one problem to overcome if we aspire to a mechanization of the scientific use of the natural language. And that is the ability to have variables that range over predicates, the ability to express predicates through constraints, classes of predicates, etc. In first order predicate logic, you cannot have variables ranging over predicates. In the natural languages, I hope I have shown that you can do the equivalent.

The natural way out of this problem is an axiom schema of unrestricted comprehension, that (like Frege's Basic Law V) establishes a correspondence between predicates and sets (thus classes). But if we add UC to the system, Russell's paradox apply, that same paradox that toppled Frege's edifice. As Paul Bernays stated in the introduction to his "Axiomatic set theory":

> What really is excluded by the antinomies is only that interpretation (easily suggested at first) of set theory (...) whose domain of individuals contains for every predicate `B` an assigned individual `p` such that:
>
> ```
> (x)(x € p <-> B(x)).
> ```
>
> —Paul Bernays

So with first order logic we come to exactly the same point where Frege foundered.

3.2 Modern manifestations of the problem

This problem I have described can be seen in many modern logic programming systems, where any attempt to use our natural logic as design model for software development is futile.

A paradigmatic example is the OWL language of the semantic web. This language had two flavours: DL, and full (well, and lite). It is an ontology language that can be processed by reasoners to extract consecuences. It has basic class predicates, and it has UC: you can have anonymous classes defined by predicates. But, in the DL flavour, you cannot treat classes as individuals: you cannot have variables ranging over them. In the full flavour, you can, but, as they say, "It is unlikely that any reasoning software will be able to support every feature of OWL Full" (see here). And it is the full flavour that would provide full (natural) expresivity.

3.3 A possible solution

My proposition is to use the predicates of set theory to express the natural copular verbs, just as Frege (or OWL) did, but then, as we shall see below, instead of representing the rest of the natural verbs as formal predicates, we represent them through individuals of the theory. We limit our (first order) theory to only have the basic predicates of set theory in their barest form. With their barest form, I mean defined by just equality, extensionality and a definition of subset, and perhaps some boundary axioms to define a universal and an empty "set". In contrast, Paul Bernays, in the text

quoted above, after dismissing UC, goes on to provide a number of additional axioms, like the axiom of choice or the axiom of infinity; what he called constructive axioms, that gave further form to the set predicates (and quite estranged them from the natural copulas). But his aim was different from ours. He wanted a formalism to base on it the whole of mathematics, so he needed a few axioms to produce a finished theory that would contain all mathematical structures. We do not need a closed theory, nor an initial complex structure as model for our theory. All we need is a simple and empty starting theory, that allows us to extend it with ad hoc new individuals and axioms to model each particular informal theory.

3.3.1 The theory NPL

We call this theory NPL. To sketch it, we will only use implication `->` and conjunction `&` as logical connectives, and the only production rule will be modus ponens. Variables are denoted by `x1`, `x2`... and are always universally quantified in their outernmost scope (sentence); and individuals are denoted by any sequence of lower case letters. The predicates are `isa`, equivalent to "belongs to", and `are`, equivalent to "is a subset of" (for this quick sketch of the theory, we do not need equality). We use these predicates in an infix form, and we have that:

```
x1 isa x2 & x2 are x3 -> x1 isa x3

x1 are x2 & x2 are x3 -> x1 are x3
```

Now to the representation of natural verbs other than copulas. For simplicity, we will only consider natural verbs that represent binary relations, so a natural sentence with such a verb would have the form of a triplet subject-verb-object. To represent this relation, we use a ternary operator `f` (from fact). So, a non-copular sentence, in our system, would have the form `f(s, v, o)` (where `s`, `v`, and `o` are just individuals of the theory). Since `f` is an operator, this sentence stands for just another individual of the theory, and has no truth value. We will call this sort of individuals "facts". To attach truth value to facts, we use the set predicates, to put them in relation with another individual of the theory, `fact`. So a complete non-copular sentence, in this theory, would have the form (with prefix operators and infix predicates):

```
f(s, v, o) isa fact
```

Since we only have 2 (or 3, with equality) formal predicates, we do not need UC at all, and yet we can have variables that range over the equivalents of our natural verbs (and also over whole "facts"). The point is that we can model the forms of natural logic with very few predicate and operator symbols, and that any new term we may want to introduce, when modelling any kind of natural discourse, will be quantifiable by first order variables. Those symbols that can not be quantified, like `are` or `isa` or `f`, are so few that do not merit to be so.

We can be even more fine-grained. If we call "predication" to a pair verb-object, we may want to have variables that range over them. To do this, we can define a new operator `p`, that produces predication individuals, so that now the `f` operator takes 2 operands, the subject and a predication, to have something like:

```
p(v, o) isa predication

f(s, p(v, o)) isa fact
```

And, to show a little more of what can be obtained from such a system, note that facts and predications are individuals of the theory, so we can use them where we have used `s` or `o`, to build as complex a sentence as we may want (I think it wouldn't make much sense to use them in place of `v`).

3.3.2 An example derived theory

An example developed on top of this theory might be (using a primitive universal set `word`):

```
person isa word
```

```
man are person

john isa man

woman are person

yoko isa woman

verb isa word

loves isa verb

x1 isa person &
x2 isa verb &
x3 isa person &
f(x1, x2, x3) isa fact
->
f(x3, x2, x1) isa fact
```

Now, `john loves yoko` will imply that `yoko loves john`.

There is a semantics for this theory here.

CHAPTER

FOUR

INSTALLATION

4.1 Dependencies

- Python 2.7

To install the software, you will need:

- buildout
- virtualenv
- git
- UNIX ?

4.2 Install

to install:

```
$ git clone git://github.com/enriquepablo/nlp.buildouts.git
$ cd nlp.buildouts
$ cp buildout.cfg.in buildout.cfg
$ virtualenv --no-site-packages --python=python2.7 .
$ source bin/activate
$ python bootstrap.py
$ bin/buildout
...
```

If you obtain any errors trying this, please report at the issue tracker.

CHAPTER

FIVE

INTERFACING WITH NPL

You can use **npl** in several different ways. You can start an ircbot and talk to it in an irchat, in a REPL manner. Or you can start a daemon and talk to it over HTTP. Or you can use it from python (for this, refer to the nl documentation).

5.1 Ircbot

From the root of the nl buildout, do:

```
$ bin/ircbot some_name
Generating LALR tables
Signed on as some_name_bot.
Joined #nlpbot_some_name.
```

Now you can talk to the ircbot on freenode at channel #nlpbot_some_name. You provide definitions, facts and rules ending them with a dot, and you ask facts ending them with a question mark. Question facts can contain variables. Whenever you tell or ask something to the bot, you have to address it prefixing your message with its nickname: "some_name_bot: <message>" where <message> is any definition, fact, rule, or question.

5.2 HTTP

You can now also start a daemon, and, as I said, talk to it over HTTP:

```
$ bin/npldaemon
$
```

A telnet session now with the daemon:

```
$ telnet localhost 8280
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
person are thing.
Noun Person defined.Connection closed by foreign host.
```

CHAPTER

SIX

SUPPORT

There is a mailing list for nlproject at google groups. You can also open an issue in the tracker. Or mail me at <enriquepablo at google's mail domain>.