

# Psychinko: A Native Python Rule Engine

Yarden Katz, Bijan Parsia, Kendall Clark

March 24, 2005

MINDSWAP, University of Maryland  
<http://www.mindswap.org>

# Overview of PyChinko

- ▶ A forward chaining (“bottom up”) rule engine
- ▶ Implements C. Forgy’s classic Rete algorithm (1982)
- ▶ Simple, small (around 1400 LOC) and hackable
- ▶ Written in pure Python!

## Our specific motivation

- ▶ To develop a rule engine that is compatible with Semantic Web languages (more on the Semantic Web shortly)
  - ▶ Pychinko not limited to Semantic Web applications
- ▶ To work on limited resource environments
  - ▶ Example: the recent Nokia mobile phone Python development toolkit; not capable of running clunky Java code, but ideal for small-scale projects like Pychinko
- ▶ To beat in speed a Python rule engine for the SW called CWM (Closed-World Machine) that does not use Rete
  - ▶ CWM written by Tim Berners-Lee, inventor of the web, proponent of Semantic Web

## In general, why rules?

- ▶ Rules play an important role in computer science (e.g. logic programming and its theory, includes languages like Prolog and Mercury)
- ▶ Useful for wide-range of applications
  - ▶ processing web content (layout generation/format conversion, e.g. XSLT)
  - ▶ automating software and compilation in various ways (e.g. makefile rules)
  - ▶ foundations for **expert systems** such as CLIPS, used to build systems tailored for specific domains. An early application: MYCIN expert system in medical applications (helps diagnose and recommend treatment for various blood infections)

## Two kinds of rules

- ▶ A rule consists of a *left hand side* (LHS) and a *right hand side* (RHS). Rules operate on *facts* (ground data values, also called *assertions*)
  - ▶ LHS: the set of conditions that must hold for the rule to *fire*
  - ▶ RHS: the set of actions that are taken when the rule is fired (i.e. its LHS conditions are true), or in our case, the kind of inferences that are to be drawn
- ▶ Expressing knowledge (declarative rule): “All humans are mortal”
- ▶ Doing something (operational rule): “Print all two-column tables in green”

## Examples of rules

- ▶ “All humans are mortal”
  - ▶ for all  $x$ , if  $x$  is a *human*, then  $x$  is *mortal* (formally,  $\forall x(\text{Human}(x) \rightarrow \text{Mortal}(x))$ )
  - ▶ in traditional Lispy (CLIPS) syntax:

```
(defrule socrates
  (human ?x)
  =>
  (mortal ?x))
```

(drawing an inference)
- ▶ An operational rule: “Print all two-column tables in green”
  - ▶ 

```
(defrule handle-tables (table ?x)
  (has-two-columns ?x)
  =>
  (print-table ?x ‘‘green’’))
```

## Firing a rule: example

- ▶ Given the set of facts:  $\{(human\ bertrand), (human\ alfred)\}$
- ▶ Our rule `socrates` from before will fire, generating the following inferences:
  - ▶ `(mortal bertrand)`
  - ▶ `(mortal alfred)`

# Processing rules with Rete

- ▶ How to handle rules efficiently? Would like to avoid checking all facts against the LHS of every rule to determine which rule is to be fired
- ▶ Rete (latin for 'net') is the most efficient general method<sup>1</sup>, invented by Charles Forgy (circa. 1982, CMU)
- ▶ Builds a discrimination network (decision-tree) for channelling facts only to relevant rules (i.e. rules that might be fired in light of the new facts)
- ▶ Trades memory for speed (the rete (the decision-tree), which stores the matched and partly matched facts, is kept in-memory), complexity of building Rete is linear
- ▶ Especially efficient when there are *many* different rules

---

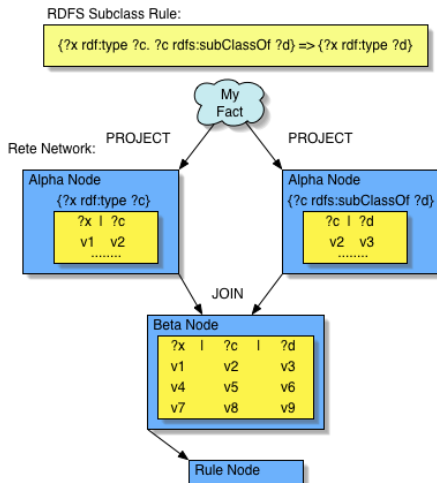
<sup>1</sup>Optimizations of Rete exist

# Rules in Pychinko

- ▶ Rules expressed in one of two ways:
  - ▶ Via language called Notation3 (N3)<sup>2</sup>, intended for the Semantic Web (more later)
  - ▶ Programmatically, as a Python object
- ▶ socrates rule in N3:
  - ▶ `{?x a :Human} => {?x a :Mortal}`
- ▶ socrates as a rule object:
  - ▶ `socrates = Rule([Pattern(Variable('x'), a, 'Human')], [Pattern(Variable('x'), a, 'Mortal')])`

<sup>2</sup>See <http://www.w3.org/DesignIssues/Notation3.html> 

# An inference rule



## Quick Digression: Semantic Web (SW) & Rules

- ▶ Recall original motivation: rule engine compatible with SW languages and more specifically, faster than a slow but popular Python rule engine for the SW called CWM
- ▶ SW Goal: give useful semantics to otherwise meaningless (to a computer) documents; allow for intelligent software agents to do *some* things that would (now) require a human eye and brain
- ▶ In short, bringing KR/AI techniques to the web (and leaving the hype at home!)

## Digression (cont'd): Semantic Web & Rules

- ▶ Core language is RDF (Resource Description Framework), a W3C standard; serializable into *triples* of the form (subject predicate object)
- ▶ Knowledge representation language called OWL (Web Ontology<sup>3</sup> Language) layered ontop of RDF
- ▶ Example: FOAF is an OWL ontology. Large instance data (RDF documents) for FOAF generated by LiveJournal, among others.

---

<sup>3</sup>Term borrowed from philosophy

## Taste of SW languages

- ▶ RDF describes *resources* (anything w/URI, essentially)
- ▶ Assertion: *Noam Chomsky is the author of Manufacturing Consent, published 1988*
- ▶ One way to think of it: there exists something in the world,



namely this object:

`http://www.chomsky.info/ManuCons`, such that something (call it “Noam Chomsky”) wrote it..

In RDF/XML..

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
<rdf:Description rdf:about="http://www.chomsky.info/ManuCor
  <dc:creator>Noam Chomsky</dc:creator>
  <dc:title>Manufacturing Consent</dc:title>
  <dc:description>The Political Economy of the Mass Media</
  <dc:date>1988</dc:date>
</rdf:Description>
</rdf:RDF>
```

Or in N3 (which would you prefer writing?):

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.
```

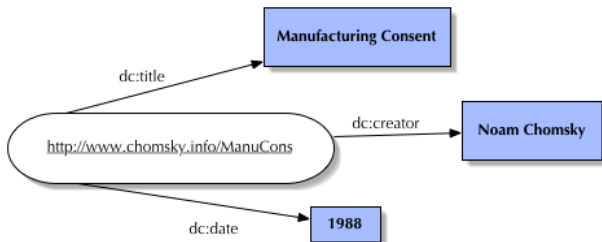
```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
<http://www.chomsky.info/ManuCons>
```

```
  dc:creator "Noam Chomsky";
```

```
  dc:title "Manufacturing Consent";
```

```
  dc:date "1988".
```



And as a graph:

# Beating CWM

- ▶ CWM (Closed-World Machine) is a Python rule engine for SW that uses naive forward chaining algorithm
- ▶ Compared with our Rete (on 1200 facts KB w/two match all rule):
  - ▶ 2400 inferred facts (double original)
  - ▶ **Pychinko time:** 0m1.609s
  - ▶ **CWM time:** 0m28.691s
  - ▶ Increase to 7000 initial facts, **our time:** 11.729s, **CWM time:** 14m58.707s (!)
- ▶ This is one of Rete's worst cases (only two rule)

## Applications of rules (on the web)

- ▶ *Social networking*: Processing FOAF documents. FOAF is a semantic web vocabulary for forming social networks. Consider a web application performing social network analysis on large, spidered FOAF documents. An intelligent filtering mechanism to avoid 'bland' documents (perhaps autogenerated by a different application) unfit for analysis is needed. A filtering mechanism of this kind might follow `foaf:knows` links, and, based on running a series of tests (in the form of rules) on the values of other FOAF properties present, decide whether the document is to be included in the analysis.

## Applications of rules on the web (cont'd)

- ▶ *Trust*: policies on the web; permissions/restrictions, "I grant access to X only to senior employees—others can only view databases Y and Z" or "Only known FOAF friends can view mypicture gallery" are naturally captured in the form of rules. The rules are easily expressed in Pychinko, while the terms to which the rules refer ("employee", "database Y", etc.) are likely to be references to a concept in an OWL ontology or RDFS document.

## Future work (in progress, actually)

- ▶ Generalizing Pychinko (native Python syntax in addition to N3, N-ary patterns)
- ▶ Moving Pychinko to the Nokia. On the phone, resources are limited, and the hardware limitations prevent one from using heavy-duty reasoners such as those written in Java. Can use Pychinko to process rules on the phone.

# Questions?