Beyond Façade:

Pattern Matching for Natural Language Applications

By Bruce Wilcox, Telltale Games, Feb 2011

Understanding open-ended simple natural language (NL) requires huge amounts of knowledge and a variety of reasoning skills. And then there's dealing with bad spelling, bad grammar, and terse context-dependent input. No wonder games severely limit their use of NL.

But speech is a coming interface. It solves input issues on small devices as well as being more appropriate than a mouse or controller for some games. Those old text adventure games were a lot of fun even with extremely limited vocabulary and parsers. Google now provides free servers translating speech to text for webpages, so the focus will shift to using that text.

Scribblenauts accepts text nouns and adjectives in its latest puzzle game and Telltale Games has a planned product mixing nouns and verbs-- full natural language games are only a matter of time. Bot Colony is currently attempting this and FAÇADE already did so (sort of) 5 years ago.

In this paper I will look at successive refinements to NLP (natural language processing) as they apply to games, reviewing AIML, Façade, and ChatScript. I will cover how the state of the art of string matching has evolved from matching patterns of words to matching patterns of meaning.

This paper will make the following general points:

1. Script syntax and semantics have a big impact on ease of content authoring.
2. Matching sets of words instead of single words approximates matching meaning.
3. The absence of words is as important as the presence.
4. Wildcard matching (in place of parsing) should be reined in to reduce false positives.

AIML (1995)

Unrestricted NL input has existed for decades – chatbots like Eliza and A.L.I.C.E. (left). But the output is barely useful. It takes a lot of data to make any plausible approximation to natural language understanding. This places a premium on making it easy to author content.

A.L.I.C.E is written in AIML. AIML is a miserable system for authoring. As a programmer, I think of AIML as recursive self-modifying code at the assembly level of language processing. In other words, a content creation and maintenance nightmare. And its pattern matching is feeble.

Pattern Matching

AIML is simple. You describe a category (what others call a rule) as a pattern of words and wildcards and what to do when the rule matches the current input. AIML's syntax is XML based, with a **pattern** (input) clause and a **template** (output) clause, like the rule below:

```
<category>
<pattern>  I NEED HELP *  </pattern>
<template>Can you ask for help in the form of a question?</template>
</category>
```

The pattern must cover the entire input of case-insensitive words; punctuation discarded. The wildcard * binds to any non-zero number of words, creating patterns matching many inputs.

Problems with authoring arise instantly. Matching the sequence *I love you* in all sentences requires four rules, and still can't match *I really love you*.

```
<category>
<pattern>  I LOVE YOU  </pattern>
<template>Whatever</template>
</category>

<category>
<pattern>  * I LOVE YOU  </pattern>
<template>Whatever </template>
</category>

<category>
<pattern>  I LOVE YOU *  </pattern>
```

```
<template>Whatever</template>
</category>

<category>
<pattern>  * I LOVE YOU *  </pattern>
<template> Whatever</template>
</category>
```

If only AIML wildcards matched 0 or more words, then a single pattern would do. You can make the above patterns match *I really love you* by adding in a wildcard between *I* and *you,* but then it would also accept sentences like *I will never love you*. Wildcards in AIML are inadequate.

Recursive Substitution

The power of AIML lies in recursive substitution. It can submit input to itself using the <*srai*> tag and put the contents of * into the output using <*star/*>. The rules for mapping C*an you please tell me what LINUX is right now* to the question *What is LINUX?* might look like this:

```
<category>
<pattern> * RIGHT NOW <\pattern>
<template> <srai><star/></srai></template>
</category>
```

=> *CAN YOU PLEASE TELL ME WHAT LINUX IS* and then

```
<category>
<pattern> CAN YOU PLEASE * <\pattern>
<template> <srai> Please <star/></srai></template>
</category>
```

=> *PLEASE TELL ME WHAT LINUX IS* and then

```
<category>
<pattern> PLEASE TELL ME WHAT * <\pattern>
<template> <srai> TELL ME WHAT <star/></srai></template>
</category>
```

=> *TELL ME WHAT LINUX IS* and then

```
<category>
<pattern> TELL ME WHAT * IS <\pattern>
<template> <srai> WHAT IS <star/></srai></template>
</category>
```

=> *WHAT IS LINUX* and finally

```
<category>
<pattern> WHAT IS LINUX  <\pattern>
<template> LINUX is an operating system. </template>
</category>
```

=> *LINUX is an operating system.*

AIML does everything this way. A.L.I.C.E. has 1300 rules merely to remove what it considers useless adverbs like *really* and *accordingly* in sentences.

Extended Patterns

AIML allows you to augment patterns in two ways. One is with a *topic* you can set on the output side and test on the pattern side. This prioritizes rules inside a matching topic over rules outside of it. So if the topic became set to “tasty breakfast food” then this rule would get priority.

```
<topic name="* breakfast *">
<category>
<pattern>I like fish</pattern>
<template> Do you like sushi? </template>
</category>
</topic>
```

The problem with this is that you have to dedicate rules outside of this topic to set the topic name for you to match. How many rules you will need and how to craft them will be significant issues.

The other pattern augmentation mechanism is the *that* clause, which adds a pattern match on the last output sentence given by the system. Rules that match a *that* clause have priority over rules that don’t. You can use *that* to script continuations based on expected reactions to your output.

```
<category>
<pattern>yes</pattern>
<that> Do * sushi </that>
<template>I hate sushi</template>
</category>
```

This rule responds to a simple *yes* only if the last output was a “Do” question that ends in sushi.

With pattern augmentation, you can write some obscurely clever code. But obscure code is generally self-punishing and few people can write or read such code.

Complex computation

On output, aside from writing text and sending itself new input, you can call the OS system shell or use JavaScript. AIML lets you set variables, but other than the topic variable you can't consult them on the pattern side. AIML uses variables to handle pronoun resolution. When you author an output, you also author what pronouns you are affecting by specifying a new value for their corresponding variable and writing patterns to remap inputs with pronouns. Adds to the authoring burden but it works.

Implementation

Implementing AIML is easy. All the data is precompiled into a tree structure and input is matched in a specific order against nodes of the tree, effectively walking a decision tree. The first matching node then executes its template, which may start up a new match or do something else.

Inherent Weakness

AIML's basic mechanic is also its weakness. Recursively self-modifying input creates problems.

First, you can't glance at code, you have to read it and think about it. XML is not reader friendly. It's wordy, which is fine for a machine but hard on a human. Just consider the simple *that* rule about sushi above. You can't just look at it and know what it does. You have to think about it.

Second, you can't see a rule and know if it will match the input. You have to know all the prior transformations.

Because of this, writing and debugging become hard, and maintenance impossible. Though it's probably way better than writing rules using regular expressions, since few can easily read them.

Summary

AIML is clever and simple, and a good start for beginners writing simple bots. But after 15 years, A.L.I.C.E has a meager 120K rules. AIML depends on self-modifying the input, so if you don't know all the transformations available, you can't competently write new rules. And with AIML's simple wildcard, it's easy to get false positives (matches you don't want) and hard to write more discriminating patterns.

Still, A.L.I.C.E. is a top chatbot, coming in 2nd in the 2010 Loebner Competition.

FAÇADE (2005)

Façade came along 10 years later and concluded they could do better than AIML, at least for a story using natural language to interact with characters Trip and Grace undergoing marital strife.

Façade's NLP started with Jess, a Java extension of the CLIPS expert-system language. Jess allows you to declare and retract facts and have rules trigger when all of their preconditions match. Façade built on top of this a template script compiler that let them write NLP rules and compile them into Jess. They wrote 800 template rules, which translated into 6800 Jess rules, a small number for dealing with NLP. But then, Façade was not trying to do too much.

Discourse Acts

Façade limited input to a single short sentence at a time. One claim Façade made over AIML was: *We don't map text to a reaction. We map to discourse acts.* Façade didn't try to fully understand the input. Instead, it pattern-matched it into 50 *discourse acts*. These included *agree*, *disagree*, *criticize*, and *flirt.* So an input of *Grace isn't telling the truth* mapped into a discourse act of *criticizing Grace.* A single sentence could simultaneously map to multiple discourse acts.

After mapping sentences into discourse acts, they used different code to decide what context meant to a discourse act and which act to pay attention to. *Agreeing with Grace* might lead to Grace being happy or to Trip becoming angry that you sided with Grace.

Discourse acts didn't have to be correct interpretations of input. Facade preferred to misinterpret a sentence rather than fail to recognize something and say "huh?". They figured they got the right act 70% of the time. The story moves on, and the user may be none-the-wiser.

Façade complained: *AIML uses implementation-dependent, nonauthor- accessible heuristics to decide which single response to give and has a matching order fixed by the AIML interpreter.* I think that claim is false. I don't think the heuristics of AIML are implementation-specific nor do I think AIML is limited to single responses. Façade rules could be ordered by *salience* (inherited from Jess). Salience is a priority number on a rule and is not a good way to author control. Even the Jess manual discourages its use.

Façade used salience to create four tiers of rules. First, a bunch of low-level rules. Then, definitions of intermediate acts. Then they had rules to rewrite adjacent negative words, so "not bad" became "good". Finally they had the rules that mapped the discourse acts. All this could also have been done in AIML, admittedly by obscure rules.

Pronoun resolution was handled AIML-style by setting pronoun meanings with each output.

Facts

To work with Jess, each input word becomes a fact asserted into working memory.

> *I like beef*

becomes facts:

> *position-1-word is I*
> *position-2-word is like*
> *position-3-word is beef.*

Jess responds to these facts by triggering rules having those facts as preconditions. Rules map input words into additional intermediate facts and more rules combine intermediate facts to get one or more final discourse act facts. E.g., here are three pseudo rules:

> *hello* => iGreet
> *Grace* => iCharacter(Grace)
> *iGreet* AND *iCharacter(?x)* => DAGreet(?x)

Seeing *Hello* creates the intermediate *iGreet* fact. Seeing *Grace* creates an intermediate fact of *iCharacter(Grace)*. And if you see *iGreet* immediately followed by *iCharacter*, you get a *greeting* discourse act of the character.

Pattern Matching

Façade patterns were more powerful than AIML. They expanded the unrestricted AIML * so that it matched 0 or more words. This avoids writing 4 forms of many rules. And Façade's new patterns abilities were critical to authoring content that matches closer to actual meaning.

Their template language supported the following nestable relationships among words:

> (X Y) - an ordered relationship of words in sequence (**AND**)
>
> (X | Y) - the presence of either word (**OR**)
>
> ([X]) - X can be null (**OPTIONAL**)

AIML is based on the **AND** relationship. Façade's **OR** operator was a critical improvement. Facade could map of sets of words to an intermediate positional fact. AIML matched patterns of words. Façade matched patterns of sets of words. This is closer to doing a match on meaning.

To support doing this easily, they created syntactic sugars. One such was:
(tor X Y) which means *((* X *) | (* Y *))*
which means look for input in which X or Y occurs anywhere in the sentence.

Another sugar was:
(tand X Y) which meant ((* X * Y*) | (* Y * X *))
which meant find X and Y in any order.

And a third was *(tnot X)* which means don't find X. This, too, is an important addition because the absence of a word (particularly negative words like *not*) is often as important as the presence. Thus Façade defined the *praise* intermediate fact using the following three rules:

```
(defrule positional_Is
(template (tor am are is seem seems sound sounds look looks))
=> (assert (iIs ?startpos ?endpos)))
```

The above rule says if you see any words that approximate the verb to be, assert into working memory a new fact of *iIs* at the position found.

```
(defrule positional_PersonPositiveDesc
(template (tor buddy comrade confidant friend genius go-getter pal sweetheart))
=> (assert (iPersonPositiveDesc ?startpos ?endpos)))

(defrule Praise_you_are_PersonPositive
(template ( {iPersonPositiveDesc} | (you [{iIs}] [a | my] {iPersonPositiveDesc} *)))
=> (assert (iPraise)))
```

Facade went wrong in two ways here. First, they match the entire input, hence they still have leading and/or trailing * in patterns. Second, the templates are LISP-like and ugly to read and write. *{ iPersonPositiveDesc }* in rule 3 means check for the word as an intermediate fact instead of as a regular word. It's a clumsy notation that obscures reading the rule. In ChatScript (below) you will see a much cleaner and more comprehensible syntax.

WordNet

Facade allowed WordNet-stemmed words and the WordNet expansion set of words. They didn't say how you scripted this nor did the examples they published use them when logically they were appropriate. The idea was clever and has great utility, but it wasn't right for two reasons.

First, stemming is good for treating *go, going, goes* the same. But stemming does not handle irregular conjugation (*be am was been)* nor does it necessarily result in real words, so you have to know the stem. Consider the stem of *community*. Would you know it is *communiti*?

Second, their WordNet expansions are just sets of synonyms. But WordNet has a full ontology, so you can know that a collie is a dog is a mammal is a being. Facade makes no use of this.

Complex computation

Façade uses Jess, so it can access Java as its scripting language on the pattern side and the output side. But Jess uses LISP-style prefix notation. Many programmers hate that. Jess supports declaring facts and querying for facts, sort of like using SQL. This is good for simple relationships but not efficient for complex graph traversal. But then they never had to do any.

Summary

Facade's major NLP clevernesses were matching to intermediate facts and then to discourse acts, the OR operator to match sets of words, and the NOT operator to exclude words. My big complaint with Façade's NLP, however, was that it was still wordy and too hard to author.

ChatScript (2010)

I created a new chatbot language for Avatar Reality called CHAT-L, and later revised it into ChatScript, an open-source chatbot engine. It covers Façade's NLP capabilities more generally and more simply.

Suzette is the chatbot I wrote. She won Best New Bot in her debut at the 2009 Chatterbox Challenge and won the 2010 Loebner Competition (Turing Test), fooling 1 of the 4 judges.

AIML was a simple word pattern-matcher. Façade pattern-matched into discourse acts, a tightly restricted form of meaning. ChatScript aims to pattern-match on general meaning. It therefore focuses on detecting equivalence, paying heavy attention to sets of words and canonical representation. It also makes data available in a machine searchable form (fact triples).

ChatScript Patterns

ChatScript uses a simple visual syntax, borrowing neither from XML nor LISP. And ChatScript discards the need to match all words of the input. This avoids both the 4-rule syndrome and a surfeit of wildcards on the ends of patterns. Here is a simple ChatScript rule:

```
s: ( I love meat )  Do you really?
```

The rule tests statements (*s:*). The pattern is in parens (parens mean *find in sequence*). It matches if *I*, *love*, *meat* are in direct sequence anywhere in the input. The output is after the parens.

Rule Types

s: and *?:* are rule types that say the rule gets applied to statements and questions respectively. Façade and AIML discard punctuation and with it the distinction between statements and questions. ChatScript tracks if the input has a question mark or has the structural form of a question. Rules can be restricted to statements, questions, or *u:* for the union of both.

You can script witty repartee using continuations (*a:*, *b:*, etc.) which test input immediately following a successful ChatScript rule output. It's much clearer than AIML's *that*.

```
s: ( I like spinach ) Are you a fan of the Popeye cartoons?
    a: ( yes ) I used to watch him as a child. Did you lust after Olive Oyl?
        b: ( no ) Me neither. She was too skinny.
        b: ( yes ) You probably like skinny models.
    a: ( no )  What cartoons do you watch?
```

b: (none) You lead a deprived life.
b: (Mickey Mouse) The Disney icon.

Concepts

ChatScript supports sets of words called concepts, which can represent word synonyms or affiliated words or a natural ordering of words:

concept: ~meat (bacon ham beef meat flesh veal lamb chicken pork steak cow pig)

Here ~meat means *a close approximation of meat* and is the equivalent of a Façade {iMeat} but is easier to read and write. Then you can create a rule that responds to all sorts of meat:

s: (I love ~meat) Do you really? I am a vegan.

The ordered concept below shows the start of hand ordering in poker.

concept: ~pokerhand ("royal flush" "straight flush" "4 of a kind" "full house")

The pattern:

?: (which * better * ~pokerhand * or * ~pokerhand) …

detects questions like *which is better, a full house or a royal flush* and the system has functions that can exploit the ordered concept to provide a correct answer.

You can nest concepts within concepts, so this is fine:

concept: ~food (~meat ~dessert lasagna ~vegetables ~fruit)

Hierarchical inheritance is important both as a means of pattern generalization and as a mechanism for efficiently selecting rules to test. Concepts can be used to create full ontologies of verbs, nouns, adjectives, and adverbs, allowing one to match general or idiomatic meanings.

Canonical Form

Instead of Façade's stemming, ChatScript simultaneously matches both original and canonical forms of input words if you use the canonical form in a pattern.

For nouns, plurals canonize to singular, and possessive suffixes *'* and *'s* transform to the word *'s*. Verbs switch to infinitive. Adjectives and adverbs revert to their base form. Determiners *a an the some these those that* become *a.* Text numbers like *two thousand and twenty one* transcribe into digit format and floating point numbers migrate to integers if they match value exactly. Personal pronouns like *me*, *my*, *myself*, *mine* move to the subject form *I*, while *whom, whomever, whoever* shift to *who* and *anyone somebody anybody* become *someone*.

Façade wrote the following hard-to-read rule which only works in present tense:

(defrule positional_Is
(template (tor am are is seem seems sound sounds look looks))
=> (assert (iIs ?startpos ?endpos)))

ChatScript's simple concept below accepts all tenses and conjugations of the listed verbs:

concept: ~be (be seem sound look)

If you quote words or use words not in canonical form, the system will restrict itself to what you used in the pattern:

u: (I 'like you) This matches *I like you* but not *I liked you*.
s: (I was) This matches *I was* and *Me was* but not *I am*

WordNet Ontology

Façade failed to use a major value of WordNet, its ontology. In ChatScript, WordNet ontologies are invoked by naming the word and meaning you want.

concept: ~buildings (shelter~1 living_accomodations~1 building~3)

The concept ~*buildings* represents 760 general and specific building words found in the WordNet dictionary – any word which is a child of: definition 1 of shelter, definition 1 of accommodations, or definition 3 of building in WordNet's ontology.

Pattern Operators

AND word relations are done using ( ) or using a quoted string.

You can do **OR** choices using a concept or [] :

s: (I ~love [bacon ham steak pork (fried egg) "green egg"]) Me, too.

Similarly ChatScript supports **OPTIONAL** using { } :

u: (I ~go to { a } store) What store?

The absence of words, **NOT**, is represented using ! and means it must not be found anywhere after the current match location :

u: (![not never rarely] I * ~ingest * ~meat) You eat meat.
u: (!~negativeWords I * ~like * ~meat) You like meat.

And ChatScript finds words **UNORDERED** using << >>, a simpler syntax than Façade's. Finding words in any order makes it easy to write a single pattern like:

u: (<< you ~like ~meat >>) I do like meat.

to cover *Do you like meat?* and *Is bacon something you desire?* and *Ham is something you like.*

If you need to know where the start or end of the sentence is, you can use operators < or >. But commonly one doesn't care.

s: (< I know) You say you know.
?: (what is love >) Love is a wonderful thing.

Wildcards

ChatScript has a collection of wildcards. The unrestricted wildcard * means 0 or more words. The ranged wildcard *~2 means 0-2 words, while *~3 would be 0-3 words. Ranged wildcards are useful because they don't lose control by letting unrelated stuff make a match.. Specific length wildcards *1, *2, *3 … require that many words exactly. And there are even backward wildcards like *-2 which will find the word two words before the current position.

Consider the following ranged wildcard pattern:

```
s: ( I  *~2  ~love *~2  ~meat )
```

This allows *I love ham* and *I almost really love completely uncooked steak* but won't accept *I love you and hate bacon*.

The ability to match concepts combined with ranged wildcards yields both economy and precision. You can detect thousands of insults with this simple pattern:

```
( !~negativewords  you *~2 ~negativeAffect )  Why are you insulting me?
```

Using the ~negativeAffect set of 4000 words, the above responds to *you dork* and *you have an ugly face* but does not react to *you aren't stupid* (~negative words includes *not, never, rarely, hardly)*, nor will it react to *you can always tell the poor from the merely stupid.*

AIML automatically binds matching wildcard text so you can retrieve it during output. Façade generates no text output so it doesn't capture anything. ChatScript allows you to capture matched words using an _ prefix. If you say *I really adore raw chicken in the morning with soggy beans*, the following pattern captures the specific meat and vegetable words found in the input and echoes them in the output:

```
s: ( I  *  ~like  *  _~meat  * and  *  _~vegetable )   I hate _0 and _1.
```

Variables

User variables in ChatScript start with a $ and contain text (which auto-converts to and from numbers as needed). They can be used within patterns as well as for output.

```
s: (  I  be *  ~genderWords  ) refine()
        a: ( ~maleGenderWords ) $gender = male
        a: ( ~femaleGenderWords ) $gender = female

?: ( $gender << what be I gender  >> ) You are $gender.
s: ( $gender=male  I ~like ~male )  I prefer women.
```

The first rule detects the user saying they are some form of gender (girl, king, policeman) and immediately refines it by testing each continuation until it finds a match and then stores away the user's gender. The second rule (for later) says if the user's gender has been defined and they ask what it is, tell them. The third rule matches only if the user's gender is male and the rest matches.

ChatScript also has system variables including %rand (a random value), %length (sentence length), %month (current month) and %tense. You can use an infinitive verb concept and yet still request verbs be in past tense as follows:

```
u: ( %tense=PAST  I ~like you ) What happened?
```

Fact Triples

ChatScript supports *fact triples* and represents concepts and other things internally using them. You can write tables of data yourself, and process them however you want, typically forming sets

or user-defined graphs, storing information about each member. A table definition names the columns of the table, then the code for processing each table line, and then you just fill in the table. For example:

```
table: ~malebooks( ^author  ^title  ^copyright  )
    createfact( ^author member ~author )
    createfact( ^title member ~book )
    createfact( ^title exemplar ^author )
    if ( ^copyright != "*" ) { createfact( ^copyright birth  ^title )  }

    DATA:
    "Orson Scott Card"    "Ender's Game"        1985
    "Daniel Defoe"        "Robinson Crusoe"     *
```

With appropriate rules, you can then recognize when the user types in a title or author and know which is connected to what, the genre, and when the book was published. For example:

```
    u: (  who *  [ write author pen ]  *  _~book  objectquery( _0 exemplar ) )  @0object
```

The rule reacts to a book's name only if it knows the author. The object query (a predefined function) looks for all facts where the contents of _0 are the subject and the verb is *exemplar*. If it fails, the pattern fails. Queries always store found facts in specified places (the default for this query is the fact set @0). The output prints out the object field of one of @0's found facts (e.g., "Robinson Crusoe" exemplar "Daniel Defoe"), i.e. Daniel Defoe.

Functions

You can define pattern functions and output functions. One could create a WHAT_IS routine to detect all forms of *what-is* questions and use it like this:

```
    u: ( WHAT_IS ( LINUX ) )
```

This doesn't involve recursive modification of the input. You know what the function is supposed to do by its name and you can separately inspect the function to see if it covers all forms of the question with direct patterns.

Summary

ChatScript patterns are compact and powerful yet understandable, easy to read and write. Unless you go out of your way to be obscure, you can know immediately if the incoming sentence should be matched by your pattern. So patterns can be debugged and maintained readily (you can ask the system what concepts a word belongs to and visa versa). You can even add a special comment in front of a rule that gives a prototypical input sentence. This both documents what the rule should match and allows the system to verify that it does as a part of a regression test.

Complex computation

ChatScript intermixes direct output text with a C-style scripting language, which can be used in patterns as well as during output. ChatScript supports declaring facts and querying for them and has an extensible generic graph traversal mechanism, so it directly supports walking around the ontology hierarchy or a user-defined map of nodes and links (e.g., for describing travel on a city grid of one-way streets). ChatScript can even do the AIML thing of synthesizing new input and getting it processed through the input stream, which Suzette uses to rewrite sentences with automatically resolvable pronouns.

Implementation

Data Representation

The starting fundamental of the implementation is a dictionary array of words, initially from WordNet. Aside from direct indexing, words are accessed by hash. Each word holds static information like parts of speech and form of the word (tense, comparative status, plurality, etc) and connects it to related words (e.g., all conjugations of an irregular verb are linked into a ring.) This information is essential for canonization. Other information about the word like gender, whether it refers to a human, a location, or a time is also kept here. Words also hold WordNet synsets – lists of words that could be synonyms in some context.

Word entries can be marked with various "been here" bits for doing inference and hold a list of where in the current sentence the word occurs. This means when you encounter a word in a pattern, you can do a fast look-up and instantly find all locations in the input where the word can be found. This applies to normal words, as well as to concept names, strings, and other special words. Variables are also stored in the dictionary.

The other system fundamental is the *fact*. A fact is a triple of fields *subject verb object*. The value of a fact field is either an ontology pointer or a fact pointer. An ontology pointer consists of an index into the array of dictionary words and a meaning index (which WordNet meaning of the word or the part of speech class of the word). A meaning index of 0 refers to all meanings of the word and can be written *help* or *help~0*. If a field is a fact pointer, the value is an index into the fact array.

Each dictionary entry and each fact keeps lists of facts that use it in each position. So the word "bill" has a list of all facts that have "bill" as the subject, another list of with "bill" as the verb, and a third list that has "bill" as the object.

The WordNet ontology is built of facts using the verb *is.* So reading in WordNet will create facts similar to these (*Persian~3 is cat~1)* and *(cat~1 is animal~1)* and *(animal~1 is entity~1)*.

Concepts use the verb *member*, so *concept: ~birdrelated (bird~1 "bird house")* creates facts (*bird~1 member ~birdrelated)* and *(bird_house member ~birdrelated)*.

Data Processing

Tokenization

The tokenizer first breaks user input into a succession of sentences, automatically performing spell-correction, idiom substitution, proper name and number joining, etc. It strips off trailing punctuation, but remembers whether or not the sentence had an exclamation or a question mark. The control script will later examine the construction of the sentence, so questions lacking a question mark will still get appropriately marked.

Marking Words and Concepts

The result is then run through canonization, which generates an alternate for each word. These two forms of input are then used to mark what Façade thinks of as initial and intermediate facts, though for ChatScript they are not facts, they are just annotations on dictionary entries saying where they occur in the sentence. For each word (regular and canonical), the system chases up the fact hierarchy of them as subject, looking for verbs *member* and *is* to allow it to switch to the object to continue processing. As it comes to a word or a replacement word, it marks on the dictionary the position in the sentence where the original word was found. It does this same thing for sequences of contiguous words that are found in the dictionary.

Pattern Matching

Pattern matching can now begin. Matching cost is at most linear in the number of symbols in the pattern (excluding patterns using the query function). Suppose the input is *I really love pigeons* and this is a rule to be matched:

s: (I *~2 ~like * _~birdrelated) I like ‘_0 , too.

This rule, only applicable to statements, asks if *I* occurs anywhere in the sentence. *I* can be literal or canonical (*me myself mine my*). We merely look up *I* in the dictionary and see if it has been marked with a current position. It has, so we track the position as the most recent match. If it wasn’t there, this pattern would fail now.

Next is a range-limited wildcard, which swallows up to two subsequent words, so we note that and move on. Reaching *~like*, we look that up in the dictionary and find it is marked (from *love*) later than *I* and legally ranged. We track our new match position.

We note the next * wildcard and move on to find an _. This is a signal to memorize the upcoming match so we set a flag. We then look up *~birdrelated* in the dictionary. It is marked from *pigeons* and appropriately later in the sentence. Because of the _ flag, we copy the actual sentence match found into special match variables, copying the position found: *start 4 end 4*, the actual form *pigeons*, and the canonical form *pigeon*.

Since we completed the pattern and matched, we execute the output and write I *like pigeons, too.* ‘_0 means retrieve the 0^{th} memorization in its original form.

Since output is generated, the system passes control back to the control script to decide what to do next. Had it not generated output, it would have moved on to the next rule.

Pattern matching can do limited backtracking. If the system finds an initial match but fails later, the system will replace the match of the first pattern word if it can and try again. Eg., if the rule being tested is the (*I *~2 ~like * _~birdrelated)* one and the input was *I hate dogs but I really love pigeons*, then when *I* matches in *I hate* and *~like* does not come soon enough, the system will deanchor *I* and start the match again with the *I* of *I really*. This is simpler and more efficient than full backtracking, which is rarely profitable.

Putting it all together: Topics

The system does not execute all rules. The author organizes collections of rules into topics. A topic also has a set of related keywords (an implied concept set). These are used to find topics most closely related to an input.

> topic: ~RELIGION (~fake_religious_sects ~godnames ~religious_buildings ~religious_leaders ~religious_sects ~worship_verbs sacrament sacred sacrilege saint salvation sanctity sanctuary scripture sect sectarian secular secularism secularist seeker seraph seraphic seraphim soul spiritual spirituality "supreme being" tenet theocracy theology tithe pray venerate worship)

A topic introduces an additional rule type. In addition to responders for user input (*s: ?: u:*) it has topic gambits it can offer (*t:*). It can nest continuations (*a: b: ...*) under any of those. Gambits create a coherent story on the topic if the bot is in control of the conversation (the user passively responds by saying thinks like "OK" or "right"). Yet if the user asks questions, the system can use a responder to either respond directly or use up a gambit it was going to volunteer anyway.

A topic is executed in either gambit mode (meaning *t:* lines fire) or in responder mode (meaning *s: ?:* and *u:* fire). Rules are placed in the order you want them to execute. For gambits, the order tells a story. For responders, rules are usually ordered most specific to least specific, possibly bunched by subtopic.

```
t: An atheist is a man who has no invisible means of support.
        a: ( << what mean >> ) It means God (invisible and supporter of life ) doesn't exist
        for an atheist.
t: Do you have any religious beliefs?
        a: ( ~no ) How about ethical systems that dictate your behavior instead?
        a: ( ~yes ) What religion are you?
                b: ( _~religious_sects ) Were you born _0 or did you convert?
t: RELIGION () Religion is for people who are afraid to stand on their own.
?: (<< [ where what why how who ]  [ do be ]  God >>)
        [ There is no God. ]
        [ A God who is everywhere simultaneously and not visible is nowhere also. ]
```

```
?: (  be you *~2 follower *~2  [ ~fake_religious_sect ~religious_sects ~religious_leaders
] )          reuse( RELIGION )
u: ( << [I you] ~religion >> ) You want to talk about religion?
```

The *reuse* output function is important in avoiding double-authoring information. It is a "goto" the output section of another labeled rule.

By default, the system avoids repeating itself, marking as used-up rules that it matches. All rules can have a label attached and even gambits can have pattern requirements, so the system can dish out gambits conditionally if you want or make rules share output data.

The topic system helps ChatScript efficiently search for relevant rules. User sentence keywords trigger the closest matching topic (based on number and length of keywords) for evaluation. If it can find a matching responder, then it shifts to that topic and replies. Otherwise it tries other matching topics. Eventually, if it can't find an appropriate responder, it can go back to the most relevant topic and just dish out a gambit. So if you say *Were Romans mean to Christians?* it can come back with *An atheist is a man with no invisible means of support.* This at least means it will begin conversing in the appropriate topic.

Topics make it easy to bundle rules logically together. Topic keywords mean the author can script an independent area of conversation and the system will automatically find it and invoke it as appropriate. Control is not based on the initial sequence of words as AIML is. Instead it is based on consanguinity with the sentence. It doesn't matter if the new topic has rules that overlap some other topic's rules. You can have a topic on *~burial_customs* and another on *~death*. An input sentence *I don't believe in death* might route you to either topic, but that's OK.

Instead of Façade's salience, topics make ChatScript rule-ordering visual. You can define tiers of rules merely by defining separate topics and calling one from another.

Control Script

Something that ChatScript can do and Façade had no need for is reflection—the system can use its abilities on itself. It can track its decisions and can pattern match on what it says.

Suzette uses reflection to set up information to decide if the user responds directly to a question she poses. If she says "How many people …" then if the user input has a number or a word representing a number like *few*, she can recognize that the user answered the question and just continue in the topic. But if the user responds with something off the wall, she may use keywords in his sentence to try to switch to a new topic. One could implement this by putting appropriate continuations after every output involving questions. That would be tedious. Instead it's all managed by the control script. It only requires a small amount of script.

The control script for ChatScript is itself just a topic, and topics can invoke other topics. So you can define whatever control structure you want and even switch control structures on the fly. A chatbot can simulate becoming drunk, being drunk, and recovering, by adjusting its control data.

The control script can also do multiple passes of processing with different collections of rules, generating multiple outputs and/or storing away facts about the user and itself.

The fixed control strategy of ChatScript is:

a. Execute an author-specified topic before processing any of the user's input.

b. Execute an author-specified topic for each user sentence in input.

c. Execute an author-specified topic after processing all of the user's input.

The pre-script allows one to reset variables and prepare for new input.

During the main script analysis of each sentence, Suzette runs topics that: rewrite idiomatic sentences, resolve pronouns, map sentences into discourse acts as data for later pattern matching, learn facts about the user, search for the closest matching response, generate quips and verbal stalls, etc.

The post-script allows the system to analyze all of what the user said and what the chatbot responded with. Suzette makes heavy use of post-script. She decides if she changed topics and if so, inserts a transitional sentence. She looks to see if she ended up asking the user a question and prepares data to help her recognize if the user answers it appropriately or not. And she considers the user's input again briefly, to decide if she should inject an emotional reaction in addition to the actual response. So if you say *What is 2 + 2, dummy?* She can reply

Who are you calling a dummy? It's 4.

In ChatScript you choose how you want pronouns handled. One can duplicate the AIML-style of resolution, which means you author pronoun values at the time you author the output. Or, like Suzette, you can use automate pronoun resolution with post-script to analyze the output and compute pronoun values.

Results

The Loebner Competition is the annual Turing Test for chatbots. Human judges chat blindly with a human confederate (who is trying to be human) and a chatbot (who is trying to fool the judge into voting it as the human). In 2010, bot entrants had to take a human knowledge quiz to qualify for the main competition. Here were the 4 finalists:

#1 Suzette - Bruce Wilcox -	11 pts.	1st time entry
#2 A.L.I.C.E. - Richard Wallace	7.5 pts.	(2000, 2001, 2004 Loebner winner)
#3 Cleverbot - Rollo Carpenter	7 pts.	(2005, 2006 Loebner winner)
#4 Ultra Hal - Robert Medekjsza	6.5 pts.	(2007 Loebner winner)

Why did Suzette, a newcomer started two years ago, easily out-qualify multi-year winners?

The nature of the test (not the exact questions) was published in advance.

1. Questions relating to time, e.g., *What time is it? Is it morning, noon, or night?*

2. General questions of things, e.g., *What would I use a hammer for? Of what use is a taxi?*

3. Questions relating to relationships, e.g., *Which is larger, a grape or a grapefruit?*
   John is older than Mary, and Mary is older than Sarah. Which of them is the oldest?

4. Questions demonstrating "memory", e.g.,
   Given: *I have a friend named Harry who likes to play tennis.* Then:

   What is the name of the friend I just told you about?
   Do you know what game Harry likes to play?

Just general questions of things means tens of thousands of facts about nouns and maybe dozens of patterns all of which really mean *what does one use an xxx for?*

So what? It's just data. A.L.I.C.E. has hand-entered rules with no good ability to store or retrieve data. Cleverbot has a large database mined from human chat but which is unsuitable for this test. Therein lies a ChatScript advantage. It is easy to enter and retrieve data. I created topics for each of the areas of the test, including a table on objects and their functions:

```
table: :tool (^what  ^class  ^used_to_verb  ^used_to_object  ^used_to_adverb  ^use )
-- table processing code not shown ---
[table coffee_table folding_table end_table]   furniture     rest objects *     "eat meals on"
[ladder step_ladder]   amplifer     reach * higher     "reach high areas"
```

The table handled:

Actual Test: *What would I do with a knife?*
Answer: *A knife is used to cut food.*

Suzette's topic for memorizing facts the user said fielded:

Actual Test*: My friend Bob likes to play tennis. What game does Bob like to play?*
Answer: *The game is tennis.*

And Suzette has a broad math topic (55 rules) which allows her to calculate, count up and down, and handle some simple algebra and story math problems. She correctly fielded:

Actual Test: *What number comes after twelve?*
Answer: *13*

But a human knowledge test is not chat. Success in the qualifiers might not translate into success against human judges in the actual contest. In 2009 the judges had a mere 10 minutes to spend sorting out a pair of human and computer. In 2010 they had 25 minutes. Despite that, Suzette fooled one of the four judges into thinking she was the human and won the competition.

Conclusion

A large amount of data is critical to handling natural language. But data isn't everything. Cleverbot came in third in the Loebner Competition despite 45 million lines of automatically acquired chat, because it lacked a good mechanism for appropriate retrieval. Old-style AIML-based A.L.I.C.E. took second place with only 120K rules. ChatScript-based Suzette, the winner, has somewhat more data than A.L.I.C.E. but has a far more compact and accurate retrieval.

ChatScript supports a simple visual syntax and powerful pattern-matching features. Matching concepts instead of single words and being able to specify words that must not be in the input allow you to approximate patterns of meaning. Using ranged wildcards limits false positives. Subdividing rules into topics accessible via keywords makes large collections of rules efficient to search and makes it easy to author orthogonal content. And that is only a better start.

References:

alicebot.blogspot.com/

www.interactivestory.net/ (Façade)

www.chatbots.org/chatbot/suzette/ (version Feb/2010, not the Loebner Oct/2010 version)
www.sourceforge.net/projects/chatscript (Loebner chatbot engine)

www.loebner.net/Prizef/loebner-prize.html

www.cleverbot.com/

www.personalityforge.com (a good alternative to AIML but privately hosted)