Fresh Perspectives

A Google Talk on 11/1/2011 by Bruce Wilcox

If you are the kind of person who takes notes during a talk, you won't need to. A pdf copy will be available at the end.

I am an AI research engineer. I want to create things that people use, but that stretch the boundaries of the possible. Whatever I work on, I research it and then try to come at it with a fresh perspective. A fresh perspective is something Google and I have in common, though we approach things from opposite directions. Typically my fresh perspectives involve understanding a domain and then writing a new application-specific scripting language to encapsulate insights I have gleaned. Google's perspectives come from access to massive amounts of data and hardware.

I research everything. I even researched how to give this talk. I thought about making a lot of clever PowerPoint slides. Then I read up on using and abusing PowerPoint. I abandoned that plan.

I'm going to walk you through earlier days of my professional life for a moment, before focusing in on my natural language stuff. That way maybe you'll better understand what I do and how I approach things.

I started out in the early days of AI, when LISP was de rigeur. I became a knowledge engineer. A knowledge engineer gets into the mind of an expert in some domain, learns how he thinks, and tries to program it. I've done that a few times now.

Go

My first such task was to write a program to play the oriental game of Go. It was actually my job. The major LISP systems of the day were too slow, particularly for a game application requiring lookahead, so in preparation I wrote my own LISP interpreter, MTS-LISP. It was the start of my career writing application-specific scripting languages. MTS-LISP was fast, particularly for searching game trees. Coincidentally, it was also the LISP system Peter Norvig used at Brown.

Go has a 4000 year history. It's a game which has been studied endlessly in the Orient and for which professional schools have existed since the middle ages. Yet I was unique in all that history. Everyone else who learned Go did so to play the game. They learned and then forgot what they learned as it became internalized. I learned Go to program it, a fresh perspective. I learned consciously, watching what I learned so I could code it, reanalyzing what I learned so I could reinterpret it.

Fast forward 5 years. I had become a strong amateur player and a world-renowned Go teacher and theorist. Not only had my fresh perspective lead to a workable theory for programming Go, but I fundamentally shifted the conceptual landscape of Western Go.

I was the East Coast amateur champion, in a playoff against the West Coast one to see who would represent the US at the World Amateur's. I played an opening sequence unknown to everyone, something I called the Great Wall. A game of Go lasts about 250 moves. By the end of my first five, every strong player there knew I had a lost position. All my moves had completely violated the knowledge Go players throughout the world are taught as mere beginners. Not only was my opening game wrong, but I attacked my opponent's positions and defended my own with moves no one had seen before. The game ended with my opponent losing by a large margin. The strong players spent hours on replays trying to determine where things had gone wrong. They never found it. I was playing with a completely different map of the game. Had they asked me, I would have said that he lost on his tenth move, playing what they all thought was excellent and obvious.

I wrote a bunch of popular articles on my theories called *Instant Go,* which the American Go Association still describes as *quite simply the best and fastest teaching method ever devised, at least in English.* Nowadays not a game commentary or replay goes by without analysis using terminology I invented.

It's a strange feeling, changing the world. I imagine Google staff has that same feeling. Google has shifted the world's conceptual landscape, too. Google search is completely integrated into my way of life and didn't exist before. I don't buy piles of technical books. Heck, I don't even bother to remember details of programming languages I use. I just *Google* something as I need it. I am crippled without the Internet. And I give thanks to the gods that I don't have to say *give me a moment to "bing" that*.

My Go program was the first successful one. It could give a 9-stone handicap to a beginner and expect to win. That's like queen's odds in chess. At the time, all other Go programs could lose playing even to someone who had just been taught the rules. My boss and I published a bunch of seminal articles on computer Go. Eventually I released a Go program on the IBM-PC and sold it in Japan. The Japanese Fifth Generation Computer project spent millions trying to defeat my program and failed.

But AI has traditionally been won by brute force. Computer chess is the prototypical example, wherein search-based approaches trumped knowledge-based ones. For a while, knowledge-based approaches in Go such as mine held the upper hand, but I left the field and today monte-carlo search-based Go programs dominate.

Brute force today includes high speed searches on massive databases. You guys. Machine translation, for example, used to be the domain of knowledge-based approaches. Now Google has led us to data-based translation. But massive data still needs a clever insight to use it or a clever algorithm to process it. Predicting flu season by keyword search trends is a clever use of data. But despite your advances in machine translation, I still have trouble making sense of Japanese web-sites you translate.

Route Planning

When I worked on real-time vehicle AI for DARPA in the 80's, I did route-planning for tank platoon leaders. I talked to them, read the military manuals, even went inside an Abrams tank at Fort Knox. I learned to think like a tank platoon leader. The standard computer algorithm of the day for route planning was a flood-fill, which generated perfect answers. Even today the standard algorithm in video games is A* which is just a cleverly guided flood fill. Back then, the flood fill took hours on big machines and thus was completely useless in the field. I used a different perspective to make a program that took seconds instead of hours and did better than humans (within 85% of perfect). I tried to improve upon how the platoon leaders actually planned routes. I used expert-system-controlled ray-tracing and treated parts of terrain as optical surfaces. The system looked at a ray from start to goal, detected the forest obstruction in the way, determined the rays that would be needed to clear the forest on each side, and then recursed on each of those rays. And when the terrain was not obstructive, merely faster or slower, my system approximated the terrain as a series of lenses and computed the optimal path light would take through it using Snell's law. It would latch onto roads and angle thru swamps perfectly. Nowadays, of course, I would just go to Google maps and ask it to plot me a route.

Planners

In 2004 Radical Entertainment, a games company, contacted me and asked me to tell them where to invest time on next-generation AI algorithms. I researched that and told them that academic planning technology had gotten really interesting, but that they were written completely wrong for video-games and should be rebuilt. I had no experience with planning technology, but they asked me to design and build such a system, which I did. It was called HIPE. At a planning competition in 2000, the best and fastest commercial planner solved a 500 block stacking problem in 974 moves. For my first delivery milestone on the same hardware, HIPE solved the problem 300 times faster, found a solution requiring 5 fewer moves, and was built in keeping with video-games' demanding memory and CPU requirements. Shortly after HIPE was complete, the company got bought by a larger game company, who shut down the research section and HIPE disappeared.

Cellphones

Then I worked for LimeLife, a cell-phone application company in a fragmented market of different phone screen sizes and ideosyncratic OS implementions. It was a coding mess building anything. It was sort of like the Android market today.

When I was hired, of course, I knew nothing of cell phones or the languages for programming them. I worked through my first project using J2ME, a cut down JAVA for mobile. Size was always a critical problem back in the days of a 128k RAM limit for code and data. For my second project, the company was trying to win a really big customer- the publisher of InStyle and People magazines. I was warned of a meeting a month ahead wherein Limelife would learn what the product should be like. They only knew they would need a rapid prototype and that probably the product would have to be customizable somehow via download. LimeLife didn't have those capabilities.

I concluded we needed a language designed specifically for mobile, not ones carved from other languages. I designed FLIRT, a combination scripting and automatic screen layout language. You could call it a browser language for mobile. Its programs were half the size of J2ME programs and it could be downloaded and/or updated on the fly. FLIRT programs ran instantly on all phones. An initial cut of FLIRT was running in time to build the InStyle prototype, and I was just about able, in real time, to keep up with modifications the artist and designer kept providing as revised prototype sketches. Limelife won the contract and the InStyle product I programmed won the "best mobile strategy" award from the Magazine Publishers Association for 2008.

But LimeLife collapsed. Going into a financial tailspin has happened with a lot of companies I've worked for. I don't think I am responsible. But if they hadn't gone under, I'd still be working for them, so I think it must be fate's way of telling me to move on.

Chatbots

I got into chatbots as a complete newbie back in 2008. And I give two things credit for winning the Loebner in 2010 and 2011. First, my wife found England's winters too cold. And second, Google blocked me from using it.

We had moved to England for family reasons, but with the onset of the first winter, my wife found it too cold and wanted somewhere warmer. So I contacted a friend in Hawaii, to try to sell him a game I had lying around. He had just sold his cell-phone application company, but had founded a new company named Avatar Reality to build a virtual world called Blue Mars, a gorgeous high-end place designed to wipe-out Second Life. He wanted people to have persistent avatars that moved around like the user even when the user was offline. So he wanted me to do that. But then it turned out they didn't have functioning avatars yet, so I couldn't work on that task. They weren't ready to hire me. We moved to Hawaii anyway. I then suggested that if the avatars were going to pretend to be users, they should chat like them too. After a bunch of research on the web to learn about chatbots, I made a proposal and they agreed. Thus began my new-found passion for natural language processing.

There are two kinds of chatbots at present. The hand-scripted ones, like A.L.I.C.E., written in AIML (aka AI Markup Language), and the data-mining kind that memorizes everything ever said to it, epitomized by Cleverbot. Cleverbot is the brute-force approach. It has about 50 million lines of chat and automatically acquires more.

I proposed something like Cleverbot, to memorize everything a specific user said in chat. But since users wouldn't talk about everything, I thought that when necessary it should send out Google queries to find responses that would be appropriate. It's not so much looking up facts in Google, but finding out how people have completed the answer to a question. If asked "How often do you swim"… it would go Google for sentences of the form "I often swim …".

After a few months, the system seemed to work. But then Google started blocking my queries. I created a round-robin system to go across a dozen different search engines, but eventually decided to scrap the whole approach. This forced me into the hand-scripted method, creating my own chatbot language because AIML was too weak for what I wanted to do. And ultimately that allowed me to enter the Loebner's, which doesn't allow Internet connections. So I thank you for that.

I still think that the Cleverbot approach will eventually follow in the AI tradition of brute force winning out, but it came in 3rd last year and didn't qualify this year. Data is not enough. You still need a clever algorithm to exploit that data. That's something Cleverbot seems to lack, despite its name.

The Loebner's has long been dominated by regular entrants. Last year my competitors were multiple-year winners. Cleverbot, with 50 million responses, lost to ALICE, with a mere 120,000 rules. But Suzette, with only15,000 rules, fooled a human judge, defeating ALICE.

Because that engine and chatbot were owned by Avatar Reality, I couldn't do anything commercial with it. So about 5 months before the 2010 contest, I decided to start over from scratch. Suzette was going to become a dead end. I would have to rewrite my 50,000 lines of engine code and my 15,000 rules. Due to really bad timing, I started coding my new engine while starting a new job at Telltale Games, and signing a contract with a Japanese company to build them an ESL bot in what would become my new engine. I was significantly overloaded until April of this year, so only had a few months to actually build my new chatbot, Rosette. Rosette is only 2/3 the size of Suzette. And this year the tech savvy judges took great pains not to suffer the embarrassment of misjudgment that happened last year. Still, Rosette came in first in the qualifiers and was the top pick of all four judges in the actual contest for most human computer. What is my secret?

Chat Technology

OK. Now we leave history and delve into the underlying technology. As a human I was already a domain expert in conversation, so I did what I always do. I researched all the chatbot stuff I could, including AIML, a site called Personality Forge, a videogame called FAÇADE, Jabberwock, a winning chatbot from early 2000's, etc. They all had different interesting bits but, as usual, I found I had serious complaints. It's those complaints that motivated my design for yet another application-specific scripting language. I also researched knowledge representation and inferencing, eventually picking CMU's Scone Project as my rough model.

All hand-scripted chatbots consist of a collection of rules- the classic expert system of IF-THEN rules. A rule consists of a pattern which is matched against the input. The pattern of a rule usually consists of words and/or wildcards that can match an arbitrary sequence of words.

AIML

I describe AIML for a moment for a couple of reasons. First, it is the dominant open source platform. Second, ALICE is written in AIML and she is perennially one of the top chatbots.

AIML is based on XML. In AIML, the pattern must precisely cover all input. If the pattern matches, the output code is executed. The output side can perform various simple computations, and output specific words or wildcards that were matched to one or more words. When the output side of the initially matching rule is complete, the response is done.

AIML is elegant in its simplicity. The primary clever thing AIML supports is generating a new input sentence and submitting it to itself for further computation, sometimes submitting multiple sentences and then combining their output. All of AIML's power comes from this recursive ability.

Here is a simple AIML rule that responds precisely to "Do you love me?".

```
<category>
<pattern> DO YOU LOVE ME </pattern>
<template>I love you </template>
</category>
```

Category is AIML's word for a rule. *Pattern* is what to match against. *Template* is what do execute or say in response. AIML and other chatbot languages discard all punctuation and AIML is case insensitive (input normalizes to upper case).

The AIML I showed you doesn't match "Do you love me always" or "Will you love me". To match something more broadly requires a wildcard. AIML wildcards require 1 or more words to match them.

```
<category><pattern> * YOU LOVE ME * </pattern><template>I love you </template> </category>
```

The above matches *Do you love me always* but not *Will you love me,* because a wildcard must match at least one word and the trailing wildcard doesn't.

To handle all forms of "you love me" in sequence, but not consecutively (e.g. *Will you really love only me tonight),* would require 16 AIML categories using various combinations of the * wildcard.

Now consider rejoinders. After you script a chatbot's output you can sometimes guess the human's response, and if you have a clever rejoinder ready, your bot looks really good. If the human asks *do you love me* and your bot replies *I love you*, maybe the human will then reply *You are just saying that*. Glib counters like *No, I really* *mean it* go a long way to creating the illusion of understanding. AIML manages rejoinders by adding an extra pattern to a category, called *that*:

```
<category>
<pattern> YOU ARE JUST SAYING THAT</pattern>
<that> I LOVE YOU </that>
<template>No, I mean it</template>
</category>
```

If the last thing the bot said was “I love you” and the human says “you are just saying that”, this matches. Because *that* is a pattern, it can include wildcards and be made to match the output of multiple rules. But that capability is overkill. One generally only ties a rejoinder to a specific rule, and so one just restates the output it matches.

Basic point—AIML is incredibly wordy. Because of that you can’t glance at AIML code and see what it does. You have to read it. AIML damages itself by being based on XML. It is catering to the computer-science machine view, instead of the human authoring view. One can build tools to try to hide the XML. But if you are going to hide it, why have it in the first place? And that’s just for input patterns.

On the output side, you can also do many esoterically wild things using recursive input, but one thing is certain, what is happening will become confusing and obfuscated because it will cross through many categories before being fully processed. It becomes harder and harder to keep in your head how a large chatbot works when its guts are spilled out across many different categories in probably many different files.

Enough of AIML.

ChatScript

ChatScript is similar in that it is a collection of IF-THEN rules, which match patterns against the input and executes output code when the pattern matches. So what makes ChatScript better?

First, it is much cleaner to read and write. A chatbot needs a lot of material. Hand-authored text takes time and the more characters you have to type, the longer it takes to author. In addition to merely writing lots of material, serious chatbots with a large number of rules become harder to author because it becomes hard to track what you do and do not have in it and it becomes harder to add new material without impacting access to the older material.

Here’s simple ChatScript, comparable to the AIML you saw:

```
?: (you * love * me) I love you.

        a: (you are just saying that) No, I mean it.

                b: (no)  You think I love someone else?
```

The ?: is the rule type. ChatScript strips off only the ending punctuation, but it uses that punctuation to guide which rules should be tried. *s:* rules only react to statements. *?:* rules only react to questions. *u:* rules react to the union of both. ChatScript can detect questions either because a question mark was used, or because the sentence is structured as a question.

The pattern is encased in ( ) and the output occurs after the closing) .

ChatScript doesn't have to match all words of the input. And ChatScript's * wildcard matches 0 or more words instead of 1 or more. So one ChatScript rule can match everything that requires 16 AIML categories.

ChatScript rejoinders occur visually immediately after the corresponding rule and get tested only on user input immediately after that rule fired. You can have multiple levels of rejoinders, marked by increasing letters of the alphabet. The *a:* is a level 1 rejoinder. The *b:* is a level two rejoinder for responses to the a:.

OK. Formatting differences do not a winning chatbot make, but it's a start.

ChatScript Topics

ChatScript requires that you organize rules into collections of topics. AIML has a topic concept, but the mechanism for it is so hard to use, no one does, not even ALICE. Being able to organize, manage, and automatically test large quantities of rules is an important trait for any expert system (or any large program for that matter). AIML offers no support for this. ChatScript does.

A Chatscript topic is a name, a collection of keywords, and a collection of rules. Topics allow one to encapsulate rules related to it and independently author sections of the chatbot. The system doesn't have to consult rules in topics unless it is currently in that topic, has been told to try that topic, or the sentence has one of the topic keywords in it, suggesting that maybe this topic is relevant.

The mini-topic below illustrates most of ChatScript's syntax and will be explained in detail. I am going to cover a lot of capabilities (and gloss over a lot of subtleties) in explaining it.

```
concept: ~baseball_team [ Giants "Red Sox" Yankees ]

topic: ~baseball [ baseball~1 ~baseball_team "home run" runs base~n ]

t: ( !$histeam ) What is your favorite baseball team?
        a: ( not ) You don't like baseball? How un-American.
        a: ( Giants ) ^reuse(GIANTS)
        a: ( ~baseball_team ) They're OK, I suppose.
t: GIANTS ( !$histeam )   I like our home team, the Giants.

#! I really love baseball
#! I enjoy softball
s: ( !not I *~2 ~love *~2 baseball~1 ) So do many Americans.

#! I like the Yankees.
s: ( !Giant I * ~like * _ baseball_team ) I'm a Giants fan.    $histeam = _0

#! What team do you most like?
#! Your favorite team?
#! Do you have a most special baseball team?
?: ( << you ~favorite team >> ) ^reuse(baseball.giants)

#! I hit 46 home runs last year.
#! I had three thousand and twenty home runs this season
```

```
s: (  I [hit have]  ~number>40 * home run )   That's impressive.
```

In addition to the responders (*s: u: ?:*) and rejoinders (*a: b: etc*), topics have a type of rule called a gambit (*t:*) . It's something the chatbot can say when it has control. Even if the system cannot find a direct response to your input, if your input suggests we are talking about baseball, the chatbot can offer you a relevant gambit. Or the bot can initiate a topic and say a gambit.

Gambits allow the chatbot to tell a story in the topic. If the user metaphorically nods his head in acknowledgement after a gambit, the system is free to issue the next one in sequence. If the user asks a question or makes his own statement, the system can try rejoinders or other responders to reply.

One of the implied rules of a conversation is maintaining a balance of intimacy. If I ask you a question, I am expected to share my answer to it as well. So I often author a topic in a style of gambits asking a question and then volunteering the chatbot's answer, as is seen in the baseball topic.

Gambits do not require patterns, but they can have them. Typically this is done to test conditions unrelated to the user's input. In the gambits above, we want to volunteer the question and our own response ONLY if the user has not already told us what his favorite team is, that is, only if the variable $histeam is undefined. It will be set by a responder previously if it detected the user telling us his favorite team.

Rules can have labels, and be the target of other rules. The label is after the rule type and before the pattern component. reuse(GIANTS) tells the system to execute the output of the labeled rule. This saves having to rewrite that output multiple times, and supports avoiding repetition as described later.

# is the comment character. #! is a special comment that says: here is a sample input for the immediately following rule. Not only does this help the author read what his rules cover, but the system can be run to verify that all such commented rules would actually match if given the corresponding input (would you believe that sometimes I actually write a faulty pattern for what I intend?). And verification can tell you if the rule can be found if you aren't currently in that topic or if a rule is masked by an earlier rule in the same topic.

ChatScript Generalization

ChatScript allows you to generalize words. This is gives you the power to match related words and thus write compact scripts.

You can declare a set of words as a concept and then use that concept name in places where you would use an ordinary word. You see above, the concept of ~baseball_team defined. You can then use a concept, e.g., in a rejoinder like: *a: (~baseball_team)* and it matches any baseball team's name in the list. You can even use phrases as keywords, like "home run", and they will behave in a pattern like a single word.

Also when you use a word, you can annotate it to restrict its part of speech. Hence, *base~n* means *base* is a keyword of ~baseball if used as a noun in input but not if used as a verb.

ChatScript uses WordNet, a computerized dictionary, with the WordNet ontology (a collie is a dog is a canine is a mammal is a being) and you can refer to a WordNet meaning in your patterns, which automatically stands for any refining word below it in the WordNet hierarchy, providing a bunch of instant sets of words. In the topic definition, baseball~1 is a reference to the Wordnet definition one of baseball, which also covers lower level words hardball and softball. That is a small use, but a reference like plant~2 refer to thousands of plants (plant~1 refers to industrial plants).

ChatScript also generalizes words automatically, simultaneously matching both the original and its canonical form of the input against your keywords. This means you can write things that are insensitive to case, plurality, determiner, degree, and tense.

A ~baseball rule uses ~number (a predefined concept of all numbers) and matches all of its test inputs. For this pattern, however you write a number, in digits or in words, the digit form is the canonical form. You can even put tests in the pattern.

The baseball topic uses *runs*, which is not its canonical form, which means it will not be triggered by *ran* or *run* or *running*. You can also suppress canonical matching of a word merely by putting a single-quote in front of it.

Topics have keywords so the ChatScript engine can automatically find the most relevant topic for an input sentence by looking at the number of matching keywords it has and how big the keywords are. The system will try rules in the most likely topic first, and if they all fail, it will try lesser matching topics. Topics that don't match don't get tried unless you explicitly tell the system to try them (you can completely control how processing is done).

A concept name always begins with a ~ and so does a topic name. Topics are also concepts and the topic name stands for all its keywords.

Sometimes there are idiomatic sentences that have no useful keywords in them. *What do you do?* for example. Rosette handles these by always first calling a topic that tests for idioms, which marks things so that the correct topic will get attention anyway.

Having topics means the system can prioritize by examining the current topic first. Or return to recent topics.

Via generalization, a single ChatScript rule may do the work of hundreds or thousands of AIML rules. When I say that ALICE has 120,000 rules and Suzette had only 15,000, that comparison is meaningless, except that it indicates it took me a lot less time to write Suzette's rules than it took them to write ALICE's.

ChatScript pattern matching specificity

AIML patterns are simplistic and imprecise. AIML does a pattern match on words. I characterize ChatScript as doing a pattern match on meaning. Generalization is an important part of this. So are wildcards, but AIML's are unrestricted.

Errors in pattern matching come in two flavors. False negatives (failing to match what you want) and false positives (matching what you don't want). It is easy to match things. This AIML wildcard "*" can match anything (just not nothing). The trick is to avoid matching the wrong things, which AIML doesn't do well.

Since you don't want to write a rule for every sentence you could encounter, AIML generalizes by adding wildcards, "I * love * you *". This instantly goes way too far. It can match *"I love filet mignon, what you would call steak"* and *I would rather die than love you ever*

ChatScript's wildcards can be limited to specific word counts, or limited to within small ranges. *~n means 0 or more words up to n inclusive. This means you can write patterns that don't go off on wild tangents

```
#! I really love baseball
s: (!not  I *~2 love *~2 baseball)
```

The *s:* rule matches a meaning wherein the user says they love baseball. For me, *~2 is a common idiom. It allows a determiner + adjective, or a pair of adverbs to creep in, but not much more.

AIML automatically captures the words its wildcard matches. In ChatScript, you can request such a capture by placing an _ before it. Thus s: (_ *) would memorize the entire user sentence. During output, you can refer to the capture by saying _0 or _1 or whichever wildcard you want when you had multiple. Typically one rarely has more than 3 captures in a pattern, though you may well have more wildcards. In the baseball topic, if the user says *I like the Yankees*, the responder for that captures *Yankees* and stores it on the global variable $histeam. We might use that in some gambit later to inquire more about his team. It would help prove the chatbot understood what he said.

An AIML pattern is heavily dependent upon word order. ChatScript can write rules that don't care about the order. << *you ~favorite team* >> means all of the given terms, but in any order. The convention, by the way to match a word aligned at the start or end of a sentence is to use the < or > markers, as in *(< do you love me >)*, which is functionally exactly the same as the AIML pattern I first showed you.

In Art, people talk about the importance of negative space. Negative space, the absence of words, is important in meaning. "I do not love baseball" is hugely different from "I do love baseball". Simple wildcards hide the *not*. ChatScript allows you to confirm the non-existence of it. *!not* tests that the word *not* does not appear anywhere after where we are now.

Matching sets of words, combined with restricted wildcards and using negative space, enables one to write precise patterns about a specific meaning.

In case you are thinking – why not just parse the input – I do that too (another thing that chatbots don't tend to do- in part for performance reasons), and you can write patterns that depend on the parse. But chatbots cannot count on Wall Street Journal style sentences being passed in. Lots of input is not very parseable – such as bad mangled English, texting, and short phrases. Just like some people type in all lower case and others in upper case.

And the 97% accuracy rate of most statistical Part-of-Speech-taggers trained on the Wall Street Journal means a lot of parse errors in casual chat (the word *like* is statistically a conjunction for WSJ whereas in chat it's usually a verb). As you might expect, I've been working on my own tagger and. It's rule-based with statistics for tie-breaking when rules aren't enough. It currently runs about 30 times faster than the Stanford parser and I'm aiming for 99.5% accuracy (but I'm not there yet).

Ontology

You already know that ChatScript supports concepts. It comes with some 1400 predefined concepts, you can define your own, and it has WordNet's ontology. Their noun ontology is often good, but other times it is not what I would want, and their non-noun ontologies are poor. I looked at SUMO's ontology (Suggested Upper

Merged Ontology), but it, too, I find often useless. It shouldn't come as a surprise that I have my own perspective on ontology (everyone does), and have created my own ontologies for nouns, verbs, adjectives, and adverbs. My ontologies are oriented for use with a chatbot to generalize meaning. My viewpoint is based on classifying the primary function of a word.

Take the verbs "paint" and "dirty" and "sculpt". In WordNet, the hierarchies are:

```
paint    => cover
dirty    => change
sculpt   => mould        => create
```

In SUMO, the hierarchies are:

```
paint    =>  coloring             = > surface change      =>  internal change
dirty    =>  combining
sculpt   =>  shape change                                 =>  internal change
```

Those hierarchies are useless to me- the words have nothing in common. To me, the words are all related. I view them as primarily about aesthetics. The ChatScript hierarchy is:

```
paint   =>  colorize  => alter aesthetics better   =>  alter aesthetics          =>  affect objects
dirty   =>               alter aesthetics worse    =>  alter aesthetics          =>  affect objects
sculpt =>  cutArt    =>  alter aesthetics better   =>  alter aesthetics          =>  affect objects
```

ChatScript directly supports chat itself.

ChatScript supports "the word". It has a dictionary with parts of speech and word attribute knowledge (it knows common male first names etc). It does spell-correction, part-of-speech-tagging, parsing.

ChatScript has topics with gambits, directly enabling telling a story and maintaining a balance of shared intimacy. And topics enable the system to order collections of rules by relevance to the input.

Chat is also self-extinguishing. You don't want to repeat yourself. If you ask me what my job is and I tell you I work for the phone company, I shouldn't later volunteer that same information. By default ChatScript both marks rules when they get used, to avoid using them again, and looks up its current output to see if it has already said it recently. In either case, the current rule would fail and the system would move on to find another matching rule. This means that I often write rules that share data using the ^reuse() function. If the system gets asked what is your favorite baseball team, the matching rule says reuse the gambit labeled *Giants*. That rule will output the answer and mark itself as having been used. This means the system will no longer volunteer it on its own, which would be redundant.

In addition, because you can test variables in the pattern side, you can avoid asking the user questions which he already volunteered the answer for. You see that with the first baseball gambit. AIML only matches input with its patterns. It cannot test variables.

ChatScript can represent and manipulate arbitrary data

ChatScript supports facts, represented as triples. You can build tables of data and combine facts into arbitrary graphs. Concepts are represented using facts, as is WordNet's ontology. ChatScript has a programmable query function to go search and inference your facts. If you ask *what is the capital of France*, Rosette does not have a rule that spews forth a canned response for exactly that. Instead the script has a rule to manage the capital city of anywhere. It tells her to search for cities in the named place, looking for ones that are also capitals.

For what was probably the most difficult Loebner qualifier question:

Dave is older than Steve but Steve is older than Jane. Who is youngest, Steve or Jane?

Rosette was the only entrant to try, and got the question right, replying*: Jane is youngest*. Here are the other Loebner finalists' responses for comparison:

1. I can't tell you because I don't know what these are: Steve, Jane.
2. You just told me something interesting about Dave and Steve but Steve is older than Jane. why are you asking me?
3. You're right. You don't know who youngest Steve or Jane is? You should know.

And then for good measure, here are ALICE and Cleverbot's answers.

1. And older than Steve but Steve is older than Jane is Dave. No one that I have talked to. (ALICE)
2. Ok so your name is Ed. (CleverBot)

Was Rosette's reply a triumph of understanding meaning? She could have just picked at random from the given two choices and gotten lucky. Even that would have required some understanding, to not react to the first sentence and to pick a choice from the second. But she was far cleverer than that.

Rosette used a rule to detect in the first sentence that you were offering comparisons of people by adjective. She ran the same rule over that sentence several times to swallow all of the information it contained. She built facts ordering them by size, keeping a canonical orientation.

```
(Dave old Steve)
(Steve old Jane)
```

Had you said *Dave is older than Steve. Jane is younger than Steve*, Rosette would have generated the same internal set of canonical facts across two sentences, compensating for the flip of adjective viewpoint from older to younger. Word opposites are also stored as facts she can query.

Then when the question came, Rosette determined what was being asked and which way it was being asked compared to how the facts were stored. Younger is the opposite of how she had stored the facts so she retrieved the data compensating for that and replied: *Jane is youngest*.

It took just 6 rules dedicated to detecting various sentence formats providing the adjective comparison data and 3 rules for different kinds of questions that might be asked about that data. Those 3 rules covered being asked the following kinds of questions. Bear in mind that these example questions from my script are about height comparisons, but in the Loebner's the questions and data were about age. It's all the same script using plug & play adjectives.

```
#! Who is tallest
#! Of Tom, Mary, and Sarah, who is the shortest
#! Who is the least tall
#! Who is taller than Harry?
#! Name someone shorter than Harry
#! Who is taller, Tom or Harry?
```

#! Who is the taller of Mary and Harry
#! Of Tom and Harry, who is least tall?

Rosette similarly correctly answered: *Which is larger, an ant or an anteater?* by inferencing among sets of sizes of things, deciding that ants were in insects, anteaters were in animals, and animals were bigger than insects. There are two normal rules dedicated to that kind of question and 8 rules for trick questions like *Which is larger, a huge ant or a tiny ant*. ALICE was the only other program to even attempt to answer the ant vs the anteater question. My bet is she got it right by guessing.

The previous year Suzette got 11 of 20 qualifier questions correct. This year, Rosette got 16 of the 20 different but similar questions correct, compared to 10-11 for her best competitors. 3 of Rosette's 4 errors were pattern generalization errors I made in my haste. She could have answered the questions correctly, but matched wrong rules. She could not have answered *what letter comes after T* because I had failed to give her instruction about the alphabet and how to manipulate it. Obviously since corrected.

ChatScript supports full scripting output

AIML has limited output capabilities, but some versions allow you to use JavaScript as well. ChatScript is a complete scripting language. You can write the output, as AIML allows, and you can do loops, arithmetic, assignment, if conditions, etc. You can declare and call functions. You can do graph queries and you can invoke topics. You can even submit input to the engine as AIML does, which is what happens for pronoun resolution – Rosette rewrites the sentence replacing the pronoun, then cancels the current one and submits the new one.

ChatScript supports introspection

The control code is just another topic, which you can script to do anything you want. The system will call a topic you name before the input, so you can perform initializations. Another topic is called for each user sentence in the input (if the input is multiple sentences). And a final topic will get called after all sentences, to handle any post-processing you want.

The engine can detect automatically repetition. It parses both input and output, so it supports automatic pronoun resolution , and script can compute what it "expects" if the user is to reply reasonably. If, for example, Rosette asks you *how much xxx* , then the post-processing topic determines that she asked that kind of question and notes she is expecting a quantity kind of answer. Which it tests for on the next user input to see if expectations are met. The control topic makes decisions about whether you "nodded" your head, asked for a change of topic, answered the question appropriately, etc.

ChatScript Summary

This completes the essentials of ChatScript. Using patterns which combine fine-granularity control over wildcards with broad generalization using ontology and the handling of negative space, you can easily write script to detect inputs with specific meanings. Using facts and customized inferencing you can manipulate pre-existing or built-on-the-fly collections of data. The topic structure and visually sparse layout allows you to quickly and easily author and organize independent areas of knowledge. And the ability to write your own control structures and introspect what the engine did allows you to craft any personality you want and address handling social conversational clues, transitional sentences, and other nice-ities of conversation.

Using ChatScript, a hastily crafted bot named Rosette qualified first and then won this year's Loebner's by unanimous judgment.

FairyTales:

Telltale Games hired me in part for my natural language skills. When I joined, I was told that designers had always dreamed of doing a story-telling product. You would feed it simple sentences and it would act out what you said. It was merely a wish, but it seemed like a perfect project for me. I would get to explore issues in meaning from a different perspective (you know me and perspectives) and I had a wealth of chatbot experience and ontology data I had built for Suzette. So, using ChatScript, in a week, I cobbled up a fixed demo where you could create sentences of "subject verb object" using fairy tale characters and a small number of verbs to do something with or to another character. The output was a text description of the result, because I had no way to make it visual. The verdict was "this is cool, now make it visual using our tool".

Thus began a background project to do that. Telltales's goal is to expand its technology while potentially making a product. For me, it's an opportunity to look at computerizing meaning using extremely simple sentences instead of the complexity of full user input as seen by a chatbot.

The research parallel is the game Scribblenauts, a recent side-scroller puzzle game where a character named Maxwell has to overcome obstacles to grab a "starite". You control his movement and you can summon into the world one or more of some 20,000 nouns. Each has inherent behaviors and you have to pick nouns that will help your quest. Find a bunch of bees in your way? Summon a beekeeper, or honey, or a fire-breathing dragon, or an atomic bomb (some choices have consequences of use that may not work well for you). Scribblenauts doesn't address verbs, except that you can "use", "mount", "attach" and "detach" things through the interface. I'd been jealous of Scribblenauts when it came out, wishing I could have built such a clever little world. This was my chance. I could be the first to handle complete control using verbs. No mouse, just pure sentence input.

I started by designing for the worst case: all nouns and all verbs. We'll have to limit it somehow later because we won't be able to create all the art assets. You get to pick one of the game characters as the subject, any verbs, and some object which can either be a game character or a concrete noun. Verbs are always done in present tense base form. There is no game design yet, but imagine the following prototype scenario:

It's a fairy tale, with the usual cast of characters available- prince, king, princess, wizard, dragon. There is a princess in a castle. She wants to get married. You must achieve that for her. Here is what you type:

1. *princess marry prince.* The system says- She will only marry a heroic prince.
2. *prince kill dragon* - Dragon arrives. They fight. The unarmed and unarmored prince dies.
3. *prince take shield* – a new prince arrives and grabs a shield.
4. *prince take sword* – he grabs a sword
5. *prince kill dragon* – They fight. This time the dragon dies
6. *prince marry princess* - The system says- She will not marry someone evil. Turns out the new prince is not as good as the old one.
7. *princess sanctify prince* - The system says- You lack a crucifix.
8. *princess take crucifix* - The princess takes a crucifix
9. *princess sanctify prince* – The prince loses all his evil nature.
10. *princess marry prince* – You win.

The ongoing work on this project is Telltale proprietary, so I can't describe solutions. I do have permission to share the issues in understanding meaning that have arisen. My solution, of course, is to create another application-specific scripting language called WorldScript, to allow designers and content programmers to define a computerized handling of words with as little typing as possible.

Easy Case

king hit princess

This is straight-forward and simple. A king arrives at the castle, walks over to the princess, hits her. She suffers damage, dislikes the king, and either retaliates or flees.

Multiple Interpretations

Issues with words of multiple interpretations crop up immediately.

king take xxx (pill, breath, break, purse, bus)

How does one handle them? The meanings of take are idiomatically different depending on the object. Just using take to mean "get physical control over" would be acceptable. But I actually do try to use the idiomatic interpretation.

king take bat – is it a baseball bat or a flying bat?

The simple answer for multiple meanings is: it doesn't matter. Pick one.

Incomplete input

The next issue is sentences that are incomplete, that need further elaboration.

king slice princess

Slice her with what? His hands won't work. He'll need a tool. The system will have to find something appropriate in the scene to use and effectively rework the sentence internally as:

king slice princess using sword

With *king throw baseball* the issue is where. In this case, I would usually interpret this to mean select a suitable indirect object, something he could throw the baseball to, as in *king throw princess baseball.*

Differences in degree

Then, different verbs can imply different degrees of effectiveness. Can that be represented?

king nick princess

Nicking is probably less damaging than slicing. Certainly *immolate* is be more severe than *singe*

Journalism 101

The more general issue of words that need filling in is answering journalism's classic questions: *who*/*what*/*where*/*when*/*why* and *how*. Some of them are easy. *Who/what* will be the subject, object, and indirect object. *Where* will be located in the current scene tied to subject, object, or indirect object (unless you say *king visit prison* , in which case we start in one scene and end in another). *When* is now. *How* I interpret to mean "with what tool", though there are other meanings for *how* related to adverbs of degree, for example. And for now *why* is easy: "because the user said to".

OK, so we take a user sentence of subject-verb-object and remap it to contain subject-verb-object-indirectobject and tool. Is that it? No.

Generic verbs

king shoot derringer vs *king use derringer*

There's the question of performing generic actions. *Shoot* is a specific action. *Use* is a generic action which in this context likely means the same as shoot. But the effects of *use* vary depending on the object of use, so objects need to know what their generic uses are and submit revised sentences.

Generic Nouns

Normally when the user mentions a noun, we create it or find it in the scene. But not all nouns should be treated that way.

king throw object

Clearly that is a generic reference to an object and is not an object to create. Neither, actually, is the word gun. There are many kinds of guns, and it is really a generic word covering all sorts of specific instances. So if the user says gun, we have to find or create a specific kind of gun, allowing the user to enter both *gun* and *colt-.45*, if that's what we created.

Part-Whole relations

Then there's *king pull trigger*. We don't want to create a free-standing trigger object and have the king pull it around in a wagon (though maybe that would be funny). It's a part of some other object. And, in fact, the verb-noun pair together have an idiomatic meaning… we are back to an equivalent of *king shoot gun*.

So we have to know parts of objects that refer us to those objects. *King pull trigger* will need another internal reference - the part. The fully decoded sentence is more like *King pull machine-gun (trigger)* which becomes *King shoot machine-gun* which becomes *King shoot princess using machine-gun*.

Similarly, part knowledge allows us to treat *king slice finger* and *king slice eye* as having somewhat different effects if we choose. In either case we have to impute the actual thing we are damaging as well as the actual object to use as a tool.

Noun Elaboration

Then there is the need to sometimes expand upon what the user refers to, to make it more useful.

king take anthrax. Is that airborne anthrax, placing the character already in danger or is it contained in a vial making it transportable but available for use as a weapon. The system should relabel the object to "vial of anthrax" when it reports back to the user, and accept *anthrax* or *vial* in user input. This makes it possible for the user to have useful expectations about how things work.

king take bat - The system could expand that to say *baseball bat* or *flying bat,* to remove the ambiguity.

And that's what I've considered and managed so far. This stuff will feed back into my chatbots eventually to improve them.

Summary

Some people search for the meaning of life. I haven't gotten to that yet. I'm still searching on the meaning of meaning. My quest to have a computer understand and manipulate meaning is a step or two closer. Only a few thousand more to go.

I hope you found this story both entertaining and instructive. A copy of this paper can be gotten from Alison. If you want to read anything else I've written, *you can just Google me*. Meanwhile, I'm happy to take any questions you now have.