Unnatural Language Processing

Keynote Presentation SSST-3
NAACL HLT 2009 - Boulder, CO
June 5, 2009
The Concern-Location Problem in Software

What program elements are relevant to a requirement?

More than 50% of the cost of developing a program is spent in maintenance.

More than 50% of the maintenance time is spent understanding the program.

NLP + PLP can help!
Natural Languages

A *natural language* is a form of communication peculiar to humankind. [Wikipedia]

Popular spoken natural languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Speakers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese</td>
<td>1,205m</td>
</tr>
<tr>
<td>Spanish</td>
<td>322m</td>
</tr>
<tr>
<td>English</td>
<td>309m</td>
</tr>
<tr>
<td>Arabic</td>
<td>206m</td>
</tr>
<tr>
<td>Hindi</td>
<td>108m</td>
</tr>
<tr>
<td>Portuguese</td>
<td>178m</td>
</tr>
<tr>
<td>Bengali</td>
<td>171m</td>
</tr>
<tr>
<td>Russian</td>
<td>145m</td>
</tr>
<tr>
<td>Japanese</td>
<td>122m</td>
</tr>
<tr>
<td>German</td>
<td>95m</td>
</tr>
</tbody>
</table>

Ethnologue catalogs 6,912 known living languages.
Conlangs: Made-Up Languages

Okrent lists 500 invented languages including:

• Lingua Ignota [Hildegard of Bingen, c. 1150]
• Esperanto [L. Zamenhof, 1887]
• Klingon [M. Okrand, 1984]
  Huq Us'pty G'm (I love you)
• Proto-Central Mountain [J. Burke, 2007]
• Dritok [D. Boozer, 2007]
  Language of the Drushek, long-tailed beings with large ears and no vocal cords

[Arika Okrent, In the Land of Invented Languages, 2009]
[http://www.inthelandofinventedlanguages.com]
Programming Languages

**Programming languages** are notations for describing computations to people and to machines.

Underlying every programming language is a **model of computation**:

- **Procedural**: C, C++, C#, Java
- **Declarative**: SQL
- **Logic**: Prolog
- **Functional**: Haskell
- **Scripting**: AWK, Perl, Python, Ruby
Programming Languages

There are many thousands of programming languages.

Tiobe’s ten most popular languages for May 2009:

1. Java 6. Python
2. C 7. C#
3. C++ 8. JavaScript
4. PHP 9. Perl
5. Visual Basic 10. Ruby

http://www.99-bottles-of-beer.net has programs in 1,271 different programming languages to print out the lyrics to “99 Bottles of Beer.”
“99 Bottles of Beer”

99 bottles of beer on the wall, 99 bottles of beer.  
Take one down and pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.  
Take one down and pass it around, 97 bottles of beer on the wall.  

2 bottles of beer on the wall, 2 bottles of beer.  
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.  
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.  
Go to the store and buy some more, 99 bottles of beer on the wall.
“99 Bottles of Beer” in AWK

BEGIN {
    for(i = 99; i >= 0; i--) {
        print ubottle(i), "on the wall,", lbottle(i), "."
        print action(i), lbottle(inext(i)), "on the wall."
        print
    }
}

function ubottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "No more", n - 1 ? "s" : "")
}

function lbottle(n) {
    return sprintf("%s bottle%s of beer", n ? n : "no more", n - 1 ? "s" : "")
}

function action(n) {
    return sprintf("%s", n ? "Take one down and pass it around,", "Go to the store and buy some more," : \
                    "Go to the store and buy some more,")
}

function inext(n) {
    return n ? n - 1 : 99
}

"99 Bottles of Beer" in Perl

[Andrew Savage, http://search.cpan.org/dist/Acme-EyeDrops/lib/Acme/EyeDrops.pm]
“99 Bottles of Beer” in the Whitespace Language

[Edwin Brady and Chris Morris, U. Durham]
A Little Bit of Formal Language Theory

An *alphabet* is a finite set of symbols.

- \{0, 1\}, ASCII, UNICODE

A *string* is a finite sequence of symbols.

- \(\varepsilon\) (the empty string), 0101, dog, cat

A *language* is a countably infinite set of strings called *sentences*.

- \(\{ a^n b^n | n \geq 0 \}\), \(\{ s | s \text{ is a Java program} \}\), \(\{ s | s \text{ is an English sentence} \}\)

A language has properties such as a *syntax* and *semantics*. 
Language Translation

Given a source language $S$, a target language $T$, and a sentence $s$ in $S$, map $s$ into a sentence $t$ in $T$ that has the same meaning as $s$. 
Specifying Syntax: Regular Sets

Regular expressions generate the regular sets

\[ a(a|b)^* \] generates all strings of \( a \)'s and \( b \)'s beginning with an \( a \)

Finite automata recognize the regular sets

![Finite automata diagram](image-url)
Some Regular Sets

All words with the vowels in order

  facetiously

All words with the letters in increasing lexicographic order

  aegilops

All words with no letter occurring more than once

  dermatoglyphics

Comments in the programming language C

  /* any string without a star followed by a slash */
Some Regular Expression Pattern-Matching Tools

egrep

    egrep 'a.*e.*i.*o.*u.*y' /usr/dict/words

AWK
C
Java
JavaScript
Lex
Perl
Python
Ruby
Context-Free Languages

Context-free grammars generate the CFLs

Let $G$ be the grammar with productions $S \rightarrow aSbS \mid bSaS \mid \varepsilon$.

The language denoted by $G$ is all strings of $a$’s and $b$’s with the same number of $a$’s as $b$’s.

Parsing algorithms for recognizing the CFLs

Earley’s algorithm

Cocke-Younger-Kasami algorithm

Top-down LL(k) parsers

Bottom-up LR(k) parsers
Ambiguity in Grammars

Grammar \( S \rightarrow aSbS \mid bSaS \mid \varepsilon \) generates all strings of \( a \)'s and \( b \)'s with the same number of \( a \)'s as \( b \)'s.

This grammar is ambiguous: \( abab \) has two parse trees.

\[
(ab)^n \text{ has } \frac{1}{n+1} \binom{2n}{n} \text{ parse trees}
\]
Programming Languages are not Inherently Ambiguous

The grammar $G$ generates the same language

$$S \rightarrow aAbS \mid bBaS \mid \varepsilon$$
$$A \rightarrow aAbA \mid \varepsilon$$
$$B \rightarrow bBaB \mid \varepsilon$$

$G$ is unambiguous and has only one parse tree for every sentence in $L(G)$. 
Natural Languages are Inherently Ambiguous

I made her duck.

[5 meanings: D. Jurafsky and J. Martin, 2000]

One morning I shot an elephant in my pajamas. How he got into my pajamas I don’t know.

[Groucho Marx, Animal Crackers, 1930]

List the sales of the products produced in 1973 with the products produced in 1972.

[455 parses: W. Martin, K. Church, R. Patil, 1987]
Methods for Specifying the Semantics of Programming Languages

Operational semantics
translation of program constructs to an understood language

Axiomatic semantics
assertions called preconditions and postconditions specify the properties of statements

Denotational semantics
semantic functions map syntactic objects to semantic values
Translation of Programming Languages

source program → Compiler → target program

input → Compiler → output
Target Languages

Another programming language

CISCs

RISCs

Vector machines

Multicores

GPUs

Quantum computers
An Interpreter Directly Executes a Source Program on its Input
Java Compiler

source program

Translator

intermediate representation

input

Java Virtual Machine

output
Phases of a Classical Compiler

source program


token stream → syntax tree → annotated syntax tree → interm. rep. → interm. rep.

Symbol Table
Compiler Component Generators

Lexical Analyzer Generator (lex)

Syntax Analyzer Generator (yacc)

source program

Lexical Analyzer

token stream

syntax tree
Lex Specification for a Desk Calculator

number [0-9]+\.|[0-9]*\.[0-9]+
%
[ ] { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
  return NUMBER; }
\n| { return yytext[0]; }
Yacc Specification for a Desk Calculator

%token NUMBER
%left '+'
%left '*'
%
lines : lines expr 'n' { printf("%g\n", $2); } |
/* empty */
|
expr : expr '+' expr { $$ = $1 + $3; } |
expr '*' expr { $$ = $1 * $3; } |
'(' expr ')')' { $$ = $2; } |
NUMBER
|
%
#include "lex.yy.c"
Creating the Desk Calculator

Invoke the commands

```
lex desk.l
yacc desk.y
cc y.tab.c -ly -ll
```

Result

```
1.2 * (3.4 + 5.6) ➔ Desk Calculator ➔ 10.8
```
The Compilers Course at Columbia University

Week  Task
2     Form a **team of five** and think of an innovative new language
4     Write a **whitepaper** on your proposed language modeled after the Java whitepaper
8     Write a **tutorial** patterned after Chapter 1 and a **language reference manual** patterned after Appendix A of Kernighan and Ritchie’s book, *The C Programming Language*
14    Give a ten-minute **presentation** of the language to the class
15    Give a 30-minute **working demo** of the compiler to the teaching staff
15    Hand in the **final project report**
Some of the Languages Created in the Compilers Course in the Spring Semester 2009

AMFM: a fractal music composition language
GWAPL: a language for designing games with a purpose
PIGASUS: a language for distributed computing
ROBOT: a language for learning programming
sn*w: a language for specifying genetic algorithms
viski: a language for 2d animations
viski simulation of the inner planets

[V. Narla, I. Deliz, S. Dey, K. Ramasamy, I. Greenbaum: http://www.viski2d.com/]

32 Al Aho
The Concern-Location Problem in Software

A **concern** is any consideration that can impact the implementation of a program.

What program elements are relevant to a concern?

More than 50% of the cost of developing a program is spent in maintenance.

More than 50% of the maintenance time is spent understanding the program.

Natural language information retrieval and compiler program analysis techniques can help!
Concern Location Problem

What program elements are relevant to a concern?

Concern location is vital for debugging, software evolution and systems maintenance.

Concern–code relationships are often undocumented.

How can we construct these relationships reliably?
Marc Eaddy’s Prune Dependency Rule

A program element is relevant to a concern if the program element should be removed or otherwise altered when the concern is pruned.

Code dependent on removed code may need be altered to prevent compile errors.

Easy (but time consuming) for humans to apply.

[M. Eaddy, A. Aho, G. Murphy, Identifying, Assigning, and Quantifying Crosscutting Concerns, ICSE ACOM, 2007]
Concern-Location Problem Case Study

**ECMAScript Language Specification ECMA-262 v3**
- International standard for JavaScript
- 172-page document written in English
- 360 concerns ("leaf" paragraphs)

**RHINO JavaScript Interpreter Version 1.5R6**
- 32,134 source lines of Java code
- 1,870 methods
- 1,339 fields

---

15.4.4.5 Array.prototype.join
The elements of the array are separated by occurrences of the separator.
The join method takes one argument:
1. Call the [[Get]] method of
2. Call ToUint32(Result(1))
3. If separator is undefined
4. Call ToString(separator)
5. If Result(2) is zero, then
6. Call the [[Call]] method of

```javascript
case Id_toSource:
    return toStringHelper(cx);

case Id_join:
    return js_join(cx, thisOk);

case Id_reverse:
    return js_reverse(cx, thisOk);
```
Manual Concern Location

Concern–code relationship determined by a human

Existing tools were impractical for analyzing all concerns of a real system

– Many concerns (>100)
– Many concern–code links (>10K)
– Hierarchical concerns

Eaddy’s ConcernTagger System used to assign elements to concerns and determine coverage statistics.

[Eaddy, Zimmerman, Sherwood, Garg, Murphy, Nagappan, Aho
void addRegexp(String regexp, String flags)
{
    addToken(Token.REGEXP);
    appendString('/' + regexp + '/\' + flags);
}

void addNumber(double n)
{
    addToken(Token.NUMBER);
    /* encode the number in the source stream.
    * Save as NUMBER type (char | char char char char)
    * where type is
    * 'D' - double, 'S' - short, 'J' - long.
    */
}

---

7.8 - Literals
7.8.1 - Null Literals
7.8.2 - Boolean Literals
7.8.3 - Numeric Literals
7.8.4 - String Literals
7.8.5 - Regular Expression Literals

- CallFrame
- Decompiler
- addRegexp(String, String)
- decompile(String, int, UintMap)
- InterpretedFunction
- Interpreter

---

Bugs
- 6063 - rhino - function.prototype.toString uses 'anonymous'
- 6313 - Rhino: "new Function" doesn't work properly with sup
- 6350 - JavaScript regular expression crashes the system
- 6705 - Rhino: eval with var statement: inside function
- 7625 - Number.toString(base != 10) not working for negative
- 7703 - Rhino: variable in loop, interpreted mode
- 11077 - ScriptableObject.defineFunctionProperties() uses inv
- 13416 - Rhino: invalid bytecode
- 13658 - Rhino: null pointer exception on class with duplicate
- 14060 - exec methods not callable through LiveConnect
- 15711 - Can't call java.lang.Class methods from JS
- 19980 - catch() block can't span multiple lines in interactive r
- 23699 - NullPointerException occurring next event
Using ConcernMapper, for a prior study Marc Eaddy had manually determined 10,613 concern-code links between the 360 concerns in the ECMAScript Specification and the 32,134 lines of code in RHINO.

It took him 102 hours!

This extensive effort strongly motivated this work.
Cerberus: Automated Concern Location

**Concern–code relationship predicted by “experts”**

Experts look for clues in documentation and code
Existing techniques only consult 1 or 2 experts

Cerberus is a system for automated concern location that combines

1. Information retrieval
2. Execution tracing
3. Prune dependency analysis

IR-based Concern Location

Goal: find locations of program entities relevant to a given requirement (concern)

Program entities are documents

Requirements are queries

Requirement “Array.join”

Source Code

join

Id_join

js_join()
Vector Space Model

Parsed code and requirements to extract term vectors

- `NativeArray.js_join()` method → “native,” “array,” “join”
- “Array.join” requirement → “array,” “join”

Extensions

- Expanded abbreviations
  - `numconns` → `number, connections, numberconnections`
- Indexed field accesses

Term weights computed using standard $tf \times idf$ formula

- Term frequency ($tf$)
- Inverse document frequency ($idf$)

Calculated cosine distance to get similarity score

- Cosine distance between document and query vectors
Execution-tracing-based Concern Location

Observed elements activated when concerns executed
– Analyzed run-time behavior of unit tests when each concern is exercised
– Found elements uniquely activated by a concern

Unit Test for “Array.join”

```javascript
var a = new Array(1, 2);
if (a.join(',') == "1,2") {
    print "Test passed";
} else {
    print "Test failed";
}
```

Call Graph

`js_construct js_join`
Execution-tracing-based Concern Location

Compared traces for a set of concerns to distinguish **elements specific to a particular concern**

Output is a list of methods ranked by their *Element Frequency–Inverse Concern Frequency* score:

\[
EF-ICF = \frac{\# \text{element activations by the concern}}{\text{total } \# \text{element activations}} \times \log \left( \frac{\# \text{concerns that activate any element}}{\# \text{concerns that activate the current element}} \right)
\]
Prune Dependency Analysis

Infer code elements related to concerns based on structural relationships to relevant seed elements
- Need to identify initial relevant seed elements

Prune dependency analysis
- Automates prune dependency rule
- Finds references to a given seed
- Finds superclasses and subclasses of that seed using the program dependency graph
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }
    public void bar() { ... }
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
PDA Example

Source Code

```java
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }
    public void bar() { ... }
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
```

Program Dependency Graph

```
contains  |
|         |
| C       |
|         |
| inherits  |
|         |
| B       |
|         |
| refs    |
|         |
| A       |
|
contains  |
|         |
| main    |
|         |
| calls   |
|         |
| B       |
|         |
| contains |
|         |
| bar     |
|         |
| contains |
|         |
| C       |
|         |
| foo     |
|         |
| contains |
|         |
| A       |
```

Al Aho
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }
    public void bar() { ... }
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
PDA Example

Source Code

```java
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }
    public void bar() { ... }
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
```

Program Dependency Graph
interface A {
    public void foo();
}
public class B implements A {
    public void foo() { ... }
    public void bar() { ... }
}
public class C {
    public static void main() {
        B b = new B();
        b.bar();
    }
}
Cerberus System for Concern Location

[M. Eaddy, A. Aho, G. Antoniol, Y-G. Gueheneuc
Cerberus: Tracing Requirements to Source Code Using Static, Dynamic, and Semantic Analysis
IEEE ICPC 2008]
Effectiveness Measures

Precision

\[ P = \frac{\text{# relevant elements retrieved}}{\text{total # retrieved}} \]

Recall

\[ R = \frac{\text{# relevant elements retrieved}}{\text{total # relevant}} \]

\[ F-\text{Measure} = \frac{2PR}{P + R} \]
Applying Cerberus to ECMAScript and RHINO

ECMAScript Specification

- 360 ECMAScript requirements ("concerns")
- 939 tests in the ECMAScript test suite cover 67% of the concerns

RHINO Interpreter

- 4,530 unique RHINO source code terms
- 3,345 RHINO documents (one for every type, method, and field)
- 1,870 methods

Threshold $t$

Concern location technique produces a list of retrieved elements for each concern ranked by a relevance score. Elements whose relevance is below $t$ are discarded.
Comparison of Technique Effectiveness

CERBERUS MOST EFFECTIVE

PDA IMPROVES IR BY 155%
PDA IMPROVES TRACING BY 104%

$ t = 0.01\%$

$ t = 1\%$

Number of Concerns Located by Technique

F-Measure

IR + Tracing + PDA
("CERBERUS")
IR + PDA
IR + Tracing
IR
Tracing + PDA
Tracing
Summary

The combination of the three techniques is the most effective at locating concerns.

– combining expert judgments reduces the impact of “unqualified experts”

Each technique and technique combination is effective at locating concerns.

Prune dependency analysis is effective at boosting the performance of the other techniques.
Open Problems

How well do these techniques work in other software domains?

Are there better combinations of techniques?

Is there an effective software engineering process for keeping track of concern-location relationships in requirements and code?

Can we use NLP + PLP techniques to produce better documentation for software?
Ultimate Open Problem: Is there a good computational model for the human brain?
Thanks for listening!