LAB PARSER: A PARSER FOR MEDICAL LAB DATA

Jeffrey L. Sponsler, MD MS, Charles Parker, BS
Alaska Brain Center LLC
4551 E Bogard Rd, Wasilla, AK USA 99654
jsponsler@akbraincenter.com
cparker@akbraincenter.com

ABSTRACT

A large volume of medical laboratory data enters the typical clinic. These files may be text or PDF format. We have designed and developed a prototype system, Lab Parser, to analyze files using natural language technology. The Parser uses recursive transition networks, sentence keyword scan to select appropriate rules, and rule macros for ease of coding. Benchmark files were employed for development. Test files yielded parsing rate average of 50%. After rule corrections, parsing score average increased to 75%. A module to transfer lab data to our electronic medical record has been developed and deployed. Storage of lab data into the record is fast and precise and data can then be incorporated into reports.

KEY WORDS:

Artificial Intelligence, Natural Language Parsing, Electronic Medical Record, Database Management System.

1. Introduction

The US Government requires interoperability via HL7 specification for lab data [1]. Some laboratories are not yet supporting HL7 format. In fact, most lab data for our clinic is PDF format. To obtain these data in a digital form we have designed and developed a system called Lab Parser to analyze such files and stored the information into our clinical electronic medical record (EMR). Lab Parser uses natural language parsing (NLP) technology [2] and specifically recursive transition network (RTN) knowledge structures. Storing lab data in digital format supports presentation on screen, presentation in reports, analysis of labs that are out of range, and lab flag alerts. Parsing natural language can be performed using a context free grammar [2] and is considered NP-Complete [3]. Some authors state that natural language cannot be described in a regular grammar [3]. Some authors also state that average case NL parsing is O(n3) [3].

2. Methods and Materials

See Figure 1 for the architecture of Lab Parser. The Lab Parser system is coded in Common Lisp [4], the Common Lisp Object System (CLOS) [5], Allegro Common Lisp MYSQL package [6], and PHP [7]. The principle modules of LP are listed here: OCR, Sentence Reader, Lab Parser, Data Extraction, Command Builder and Command Execution. Figure 1 shows the data flow path. We describe each module below. To illustrate the system we run a single lab "glucose" through the system.

2.1 Module: OCR

The OCR (optical character recognition) module was a commercial product OmniPage 18 OCR SYSTEM [8]. This system interprets a “lab.pdf” files and writes out a .txt (text) file. For our example, we present data that were derived from a test PDF file. This is not actual patient data and we have named the patient "Teal Testfive." To support this work, our EMR has a patient entry named the same. For the test patient, we will focus on "Glucose" test result.

```
Name: Teal Testfive
DOB: 2/23/1908
MAGNESIUM 2.3 1.8 2.4 mg/dL
SODIUM 142 130 145 mmol/L
POTASSIUM 4.8 3.5 5.3 mmol/L
CHLORIDE 105 98 110 mmol/L
GLUCOSE,QUANT 130 H 74 106 mg/dL
```

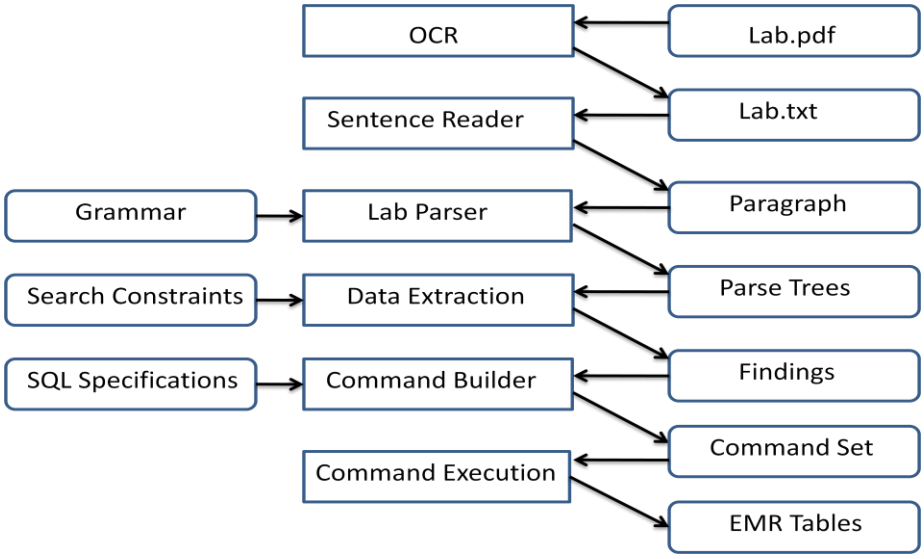

Figure 1. Modules of the Lab Parser. Software modules (e.g., Lab Parser) are in the center of the diagram. Data objects (e.g., Lab.pdf) are on the right and knowledge bases (e.g., Grammar) are on the left side.

2.2 Module: Sentence Reader

The module Sentence Reader (SR) creates a *Paragraph* class object. The SR module then opens the text file, reads each line, and generates a list of strings. The space character delimits each string and return character ends a sentence. The assumption is that one lab result will reside on one line. Special characters such as slash or hyphen are noted and marked as separate strings. For a single line, a *Lab-Sentence* class object is created. For a list of strings, a set of *Token* class objects are created and stored in slot tokens on the new Lab-Sentence object. A token is a single atomic component of a sentence and is the object of processing by the parser module. Below is a Paragraph Object for our on-going example file.

```
<PARAGRAPH labschemistry4 (53
Sentences)>:
SENTENCES(<Lab-Sentence 0 (<name>
<|colon|> <teal> <testfive> <status>)>
<Lab-Sentence 1 (<dob> <|colon|>
<|numeral| 2> <|slash|> <|numeral|
23>)>
<Lab-Sentence 2 (<magnesium> <|numeral|
2.3> <|numeral| 1.8> <|hyphen|>
<|numeral| 2.4>)>
<Lab-Sentence 3 (<glucose> <|numeral|
130> … >
<Lab-Sentence 4 (<sodium> <|numeral|
142> <|numeral| 130> <|numeral| 145>
<mmol>)> ...
```

2.3 Lab File Specific Grammar

By analyzing lab files received in our clinic, we wrote grammar rules for the sentences. Sentence analysis also produced keywords employed for context specific rules. A restricted grammar rule set narrows our rule search space and increases parse efficiency.

2.4 Module: Lab Parser

The lab parse has two major modules, the Context-Based Rule Selector and the RTN Parser.

Context-Based Rule Selector. We define the term "context-based rule" as a rule set selected based on keywords. The term context refers to a keyword identification system described below. The first step of the parsing context process requires the top-level parsing grammar rules to include keyword knowledge. For the “lab-glucose rule”, the keyword "glucose" is stored. For a given sentence S_i the context analysis routine loops over all tokens. A count of tokens that match the keywords for the rule is made and the resulting score is a fraction (keywords matched/keywords). All top-levels rules are analyzed in this manner. A *Context-And-Score* class object is created for each rule tree. The Context-and-Score objects are sorted and stored in a slot on the sentence object. The Context-And-Score object with the highest score is chosen as the best parsing

context and the rule tree for that selected context is used to parse the sentence. An example Context-And-Score object:

```
<Context-And-Score>:
Rule: <Rule Lab-Glucose>
Keywords: (glucose)
Score: 1.0
Context-And-Score objects:
CONTEXT-LIST-SORTED (<Context-And-Score LAB-GLUCOSE Score 1 ATN LABFILE-GRAMMAR> <Context-And-Score CSF-GLUCOSE Score 1/2 ATN LABFILE-GRAMMAR>)
```

The complexity of the context-based rule selector is O(i*j*k) where i is the number of sentences, j is maximum number of tokens in any given sentence, and k is the number of rule trees with keywords. Thus, the complexity is $O(n^3)$. By selection of a specific rule tree the probability of parsing success is higher and the operation is virtually instantaneous. Also, rule trees that have a number of literal strings will tend to generate parse trees that are so tagged by the constraint analysis system, with the result being data extraction is facilitated. Sentences with no keyword matches are not parsed as this is non-productive.

2.5 Recursive Transition Network Parser

We utilize a recursive transition network (RTN) parser. See [2] for a description of RTNs. We show the rule set for our test sentence. This rule set is derived from the above Rule Context Computation. The syntax for rules follows: A rule is defined as (*rule-name rule-clauses+*). Rule-clauses are processed in order, and the logical "or" is implicit. A rule-clause that begins with "=" references another rule. A rule-clause that begins with "=" and contains one or more symbols is translated into a directed graph (i.e., a transition network). So a rule-clause (= glucose comma quant) is stored as a graph with nodes (A, B, C, and D) and arcs (A=glucose=>B, B=comma=>C, C=quant=>D). The term glucose is a recursive call to another RTN where the rule-name is "glucose." Any non-literal term must be defined in the rule set or an error is flagged. For this example, D would be a terminal node (and represents a parsing solution). Recursion in the RTN occurs when an arc label is another rule name (Rule lab-glucose contains arc glucose-tag which is another rule name). An arc label may be a literal string, may name a part of speech (such as "noun"), or may be *special words* (such as hyphen, slash, period). The lexicon used is obtained from the National Library of Medicine Lexicon [7] and is used for looking up an input token to see if there is a match by part of speech. Special words are included in a Special Dictionary and tend to be punctuation. We also define a restricted *Domain Lexicon* that includes literal strings found in our grammar rules. Examples would be “glucose” and “sodium.” A rule clause with syntax (network-words *word-list*) defines the Domain Lexicon.

A rule-clause that has this form (= = literal+) is a list of literal strings for matching. The "= =" designation means that all subsequent terms are literal strings. A rule-clause that begins with "keyword" notes the list of keywords used by the Context Based Rule Selector. A rule-clause with syntax (*rule-name*) is expanded to a rule (<rule> (= "<rule>")) and so functions like a macro for quickly defining rules that are satisfied by a single literal string. For example, (glucose) expands to (glucose (= "glucose")).

2.6 Grammar Rules for Glucose

Exam set of rules for parsing lab “glucose.”

```
(lab-glucose (= glucose-tag glucose-data)(Keywords glucose))
(glucose-tag (= glucose comma quant))
(glucose)
(quant (== quant))
(glucose-data (= glucose-value glucose-flag-and-range))
(glucose-value (= numeral))
(glucose-flag-and-range (= lab-flag ref-low-and-hi glucose-units))
(lab-flag(== hi))
(ref-low-and-hi (= numeral numeral))
(glucose-units (= milligrams-per-deciliter))
(milligrams-per-deciliter (== milligrams per deciliter))
```

The rule `(milligrams-per-deciliter (== milligrams per deciliter))` defines a rule "milligrams-per-deciliter" that is satisfied by three literal strings ("milligrams" "per" and "deciliter").

The RTN Parser uses a depth-first search strategy. A successful parse generates a parse tree that contains the tokens and the matching rules organized to recapitulate the search. A rule tree is satisfied if all branches are traversed to a terminal node and if all input tokens have been consumed.

The parse contains a *Flexible Match algorithm* that uses wild cards to match mispelled words. Consider the string "sodium." The algorithm generates a list of wild card strings ("*odium", "s*dium", "so*ium", "sod*um", etc.). A string such as "sotium" is matched against each wild card and returns "true" if a flexible match is made. A lab file where all labs were mispelled was tested and results are discussed below.

The parser supports Rule Macros that expand into rule trees and facilitate rule creation. This was developed because lab sentences are stereotyped. The chemistry macro expands into rules for lab name, lab units, lab reference ranges, and lab flags.

2.7 Parse Tree Derived from Rules

We present a parse tree for our example sentence and the matching rule network Lab-Glucose. A class, *Token-Link*, points to an input token and to a transitional arc that matches the input token. During parsing when an arc and token match, a Token-Link object is created to record the match. The new Token-Link object becomes part of the parse tree. The tree is stored as a binary tree which is fundamental to Lisp. The new parse tree is stored on the sentence object.

```
PARSE-TREE (((#<Net LAB-GLUCOSE>
(#<Net GLUCOSE-TAG> (#<Net GLUCOSE>
#<TOKEN-LINK STRING glucose>) …
(#<Net GLUCOSE-DATA> (#<Net GLUCOSE-
VALUE> #<TOKEN-LINK SYMBOL |numeral|>)
(#<Net GLUCOSE-FLAG-AND-RANGE> (#<Net
LAB-FLAG> (#<Net LAB-FLAG-HI> #<TOKEN-
LINK STRING h NIL NIL>))
(#<Net REF-LOW-AND-HI> (#<Net REF-LOW>
#<TOKEN-LINK SYMBOL |numeral|>)
(#<Net REF-HI> #<TOKEN-LINK SYMBOL
|numeral|>))
(#<Net GLUCOSE-UNITS> (#<Net
MILLIGRAMS-PER-DECILITER> (#<Net
MILLIGRAMS> #<TOKEN-LINK STRING mg>)
(#<Net DIVIDED-BY> (#<Net SLASH>
#<TOKEN-LINK SLASH |slash|>))(#<Net
DECILITER> #<TOKEN-LINK STRING
dl>)))))))))
```

2.8 Module: Data Extraction

The next step in the process involves data extraction. A knowledge base (KB) of constraint specifications has been created. Each specification states how information can be extracted from a parse tree. Conceptually, if a parse tree exists this implies that a list of tokens has satisfied a rule set and extraction can proceed. A class *Lab-Flexible-Search-Constraint* stores the specification. The knowledge required to extract parse tree data is defined in the *Lab-Flexible-Search-Constraint*. Arguments of this function include Id (name of constraint), Net (the parse tree root id), Subnets (list of networks that might be found in the parse tree), Table (storage object), Attribute (storage slot on the Table), and Storage-Object-Class (data storage object).

An example is below. The definition instantiates an object that is assigned to a specific grammar rule and in our example this is the rule "lab-glucose."

```
(define-lab-flexible-search-constraint
:id 'LAB-GLUCOSE :net 'lab-glucose
:subnets '((:net glucose-value
:datatype numeral :storage-slot value)
(net ref-low :datatype numeral
:storage-slot ref-low)
(:net ref-hi :datatype numeral
:storage-slot ref-hi)
(:net lab-flag-hi :default-value high
:storage-slot lab-flag)
(:net milligrams-per-deciliter
:default-value "milligrams/deciliter"
:storage-slot lab-units))
:table 'lab-value :attribute 'glucose
:storage-object-class 'glucose-finding)
```

The definition includes a list of subnet specifications. Each subnet specification is stored on the constraint object and is available for processing when needed. All subnet specifications are processed. For a given parse tree, the subnet specification is used to search through the parse tree and to extract data and finally to store that data. The constraint includes the storage-object-class and in our example below that is stated to be "glucose-finding".

Consider the first subnetwork `(:net glucose-value :datatype numeral :storage-slot value)`. The specification states that if the network "glucose-value" is found, then search for a datatype "numeral" within that sub-network and if found, store the datum into the slot "value" of the glucose-finding. The sub-network *(:net lab-flag-hi :default-value high :storage-slot lab-flag)* specifies that if the parse tree contains net lab-flag-hi, then store a default value "high" in slot "lab-flag"of the glucose-finding object.

The new finding object is finally stored on attribute "glucose" on the "lab-values" object.

2.9 SENTENCE WITH PARSE TREE

Printed below is a sentence object, its tokens, and the successful parse tree.

```
SENTENCE ID 9
TOKENS (<glucose> <|comma|> <quant>
<|numeral| 130> <h> <|numeral| 74>
<|numeral| 106> <mg> <|slash|> <dl>)
PARSE-TREE (((#<Net LAB-GLUCOSE>
(#<Net GLUCOSE-TAG> (#<Net GLUCOSE>
#<TOKEN-LINK STRING glucose>)
(#<Net COMMA> #<TOKEN-LINK COMMA
|comma|>) (#<Net QUANT> #<TOKEN-LINK
STRING quant>))
(#<Net GLUCOSE-DATA> (#<Net GLUCOSE-
VALUE> #<TOKEN-LINK SYMBOL |numeral|>)
(#<Net GLUCOSE-FLAG-AND-RANGE> (#<Net
LAB-FLAG> (#<Net LAB-FLAG-HI> #<TOKEN-
LINK STRING h>))
(#<Net REF-LOW-AND-HI> (#<Net REF-LOW>
#<TOKEN-LINK SYMBOL |numeral|>)
(#<Net REF-HI> #<TOKEN-LINK SYMBOL
|numeral|>))
(#<Net GLUCOSE-UNITS>
(#<Net MILLIGRAMS-PER-DECILITER> (#<Net
MILLIGRAMS> #<TOKEN-LINK STRING mg NIL
NIL>)
(#<Net DIVIDED-BY> (#<Net SLASH>
#<TOKEN-LINK SLASH |slash|>))
(#<Net DECILITER> #<TOKEN-LINK STRING
dl NIL NIL>))))))))
CONTEXT-LIST-SORTED (<Context-And-Score
LAB-GLUCOSE Score 1 ATN LABFILE-
GRAMMAR> <Context-And-Score CSF-GLUCOSE
Score 1/2 ATN LABFILE-GRAMMAR>)
FINDINGS (<Lab-Finding Glucose 130
milligrams/deciliter HIGH RR 74 106>)
```

Glucose-Finding Object. The final result of processing the input string "GLUCOSE, QUANT 130 H 74 106 mg/dL" is this object:

```
Glucose-Finding
 Value: 130 Ref-Low: 74  Ref-Hi: 106
 Units "milligrams-per-deciliter"
 Flag HIGH
```

When all sentences have been parsed, and each sentence has a newly constructed parse tree, and data extraction has been completed, there is now a set of finding objects stored on a Lab-Values object.

Parser Testing: A benchmark file was used to develop and test the Parser. Four test files were parsed and lab extraction was performed. The average number of lab sentences that parsed was 50%. Analysis revealed missing or incomplete grammar rules. Once rules were added or corrected, the average number of sentences that parsed increased to 75%. Some lab sentences had extra text at the end and so a "match-anything" rule was written and added to most of the rules. This was very beneficial and increased parsing success. Another modification was to filter out "comma" punctuation from lab sentences. That filtering was beneficial; rules were simplified and there is little semantic information in the "comma" mark.

2.10 Module: Command Builder

The Command Builder module loops over all sentences with a parse tree and findings and for each creates a SQL-Command object and an insertion command string.

Find Patient in the EMR. The next step involves finding patient name and date of birth (DOB) for the current lab file. The algorithm follows: First we query the EMR to obtain all records from the table with name and DOB. Each record from that selection is stored in a *mysql-patient* class object. This class contains slots first-name, last-name, DOB, and medical-record-number. The objects are stored in a list.

We loop over each sentence, loop over each token in that sentence S_i, and search in the list of all mysql-patients for last name match. The complexity of this search is O(x*y*z) where x is the number of patients, y is the number of sentences, and z is the maximum number of tokens in any sentence. Thus the complexity is $O(n^3)$. This search is nearly instantaneous. The matching sentence is searched for first-name of the matched MYSQL-patient. The sentence with the matching name is stored. The date-of-birth (DOB) for the MYSQL-patient is obtained and all tokens (for all sentences) are searched for a match. The complexity for this step is O(y*z) and thus $O(n^2)$. If the match is found the DOB containing sentence is stored. The matching mysql-patient is stored for later database insertions.

2.11 Sentence With Patient Name

```
;; Example sentence object with name
data and matching mysql-patient object.
<Lab-Sentence 0>:
  TOKENS (<name> <|colon|> <teal>
<testfive>)
  PARAGRAPH <PARAGRAPH labschemistry4
(53 Sentences)>
  PATIENT-NAME-FINDING <Patient-Name-
Finding Teal Testfive>
```

```
  PATIENTS (<MYSQL-PATIENT 5446 Teal
Testfive 1908-02-23>)
```

After the patient/DOB search, the paragraph object will have the two matching sentences as shown below:

```
  SENTENCE-WITH-NAME <Lab-Sentence 0
(<name> <|colon|> <teal> <testfive>
<status>)>
  SENTENCE-WITH-DOB <Lab-Sentence 1
(<dob> <|colon|> <|numeral| 2>
<|slash|> <|numeral| 23>)>
```

The test MYSQL-Patient is shown here:

```
<MYSQL-PATIENT 5446 Teal Testfive 1908-
02-23>:
  FIRST-NAME "Teal"
  LAST-NAME "Testfive"
  DATE-OF-BIRTH "1908-02-23"
  MEDICAL-RECORD-NUMBER 5446
```

2.12 Finding To SQL Mapping Definition

A knowledge base that maps from finding objects to MYSQL tables has been constructed. This base includes one *sql-lab-spec-for-class* object for each finding class. This specification is a mapping from the finding to a sql-table and a sql-lab-name. A set of standard attributes common to the finding object and to the sql table are defined and include lab_value, ref_low, ref_hi, lab_units, and lab_flag. The MYSQL connection is opened, patient name is inserted into table "ocr_patient" by medical-record-number and then a unique id for that patient is obtained by query "Get Last Id". This id number is used in the following step.

SQL INSERT Patient and Lab File. A MYSQL insert into table "ocr_lab_files" is performed and a unique ID for the current file and lab data set is obtained. The insert command contains these data: patient medical record number, lab file name, and lab collection date. After the insert, the table would contain this:

(2 5446 "2017-03-21" "labschemistry4" "2017-06-15 15:26:04")

A command to obtain the unique id for this data set is performed and stored. In the example case, the unique insert ID is "2." The function mysql-get-last-insert-id obtains unique id for the file that was just inserted.

```
(mysql-get-last-insert-id)
;; Returns 2
```

2.13 SQL INSERT for the Lab Data

A lab specification and a finding are processed with the result being an *SQL-Command-Set* object. This *SQL-Command-Set* contains an SQL-Insert object and associated Command-String objects. These data structures are filled via the function *sql-lab-spec-for-class* and the result is a MYSQL insert string. The unique MYSQL ID is included in the insert command strings. One SQL-INSERT object is shown below.

```
(sql-lab-spec-for-class
 :class-name 'glucose-finding
 :sql-table 'lab_results
 :sql-lab-name 'glucose
 :use-standard-lab-specs t)
;;  Standard  lab  specs  include
lab_value, ref_low, ref_hi, lab_units,
lab_flag
```

Processing this specification for one glucose-finding object yields an SQL-INSERT object. In this example, the unique ID for this patient and this lab data entry is "2".

```
<SQL-INSERT 5 commands>:
COMMAND-STRINGS
(<COMMAND-STRING ('2', 'glucose',
'lab_value', '130'),> …
<COMMAND-STRING ('2', 'glucose',
'lab_units', 'milligrams/deciliter'),>
<COMMAND-STRING ('2', 'glucose',
'lab_flag', 'high'); Last Command>)
SQL-COMMAND-STRING
"INSERT INTO `ocr_results` (`ocr_id`,
`result_category`, `result_name`,
`result_value`) VALUES"
FINDING <Lab-Finding Glucose 130
milligrams/deciliter HIGH RR 74 106>
SQL-INSERT-STRING
"INSERT INTO `ocr_results` (`ocr_id`,
`result_category`, `result_name`,
`result_value`) VALUES
('2', 'glucose', 'lab_value', '130'),
('2', 'glucose', 'ref_low', '74'),
('2', 'glucose', 'ref_hi', '106'),
('2', 'glucose', 'lab_units',
 'milligrams/deciliter'),
('2', 'glucose', 'lab_flag', 'high');"
```

When all parsed sentences have been processed by this module, a set of SQL-INSERT command objects is ready for the Command Execution step.

2.14 Module: Command Execution

The Command Execution module first opens a database connection and then loops over all SQL-

Insert objects and the SQL-Insert-String is passed to function *mysql-insert* and the side effect is that the insert command is interpreted by mysql and the data is stored into the "ocr_results" table of the EMR. The database connection is then closed. Functions used for the interface include *connect*, *sql*, and *disconnect*. Details regarding ACL mysql functions are found in the online manual [6].

2.15 Lisp Select to See the Stored Data:

To test if the insertions of lab data were successful, execute function *db-select-by-table*. Only a subset of returned values is printed and specifically the glucose lab data.

```
(db-select-by-table "ocr_results")
(34 2 "glucose" "lab_value" "130" "2017-06-15 15:26:04")
(35 2 "glucose" "ref_low" "74" "2017-06-15 15:26:04")
(36 2 "glucose" "ref_hi" "106" "2017-06-15 15:26:04")
(37 2 "glucose" "lab_units" "milligrams/deciliter" "2017-06-15 15:26:04")
(38 2 "glucose" "lab_flag" "high" "2017-06-15 15:26:04")
```

2.16 EMR Presentation of Lab Data

Using PHP and its functions that support mysql connectivity, the lab data now stored in the EMR database can be presented for a patient [6]. See Figure 2 (for an abbreviated example).

Label	lab_value	lab_units	ref_low	ref_hi	lab_flag
Absolute Eosinophil Count	0.1	k/mm3	0.0	0.2	
Absolute Lymphocyte Count	3.4	k/mm3	1.3	2.9	high
Absolute Monocyte Count	0.3	k/mm3	0.3	0.8	
Absolute Neutrophil Count	2.0	k/mm3	2.2	4.8	low
Glucose	130	milligrams/deciliter	74	106	high
Hematocrit	39.3	%	33	43	

Figure 2. For our test example (glucose), this is the EMR display for the lab data.

Test & Notes	Sentence Count	# Parsed	Findings Count	Parse Rate	Corrected Parse Rate
Benchmark	112	100	92	89%	
Test 1	57	43	39	45%	75%
Test 2	66	48	45	60%	73%
Test 3	86	75	73	54%	87%
Test 4	41	28	25	42%	68%
5 - Misspelled	11	11	7	100%	

Table 1. Parsing and Lab Extraction data. The accuracy column records how many labs were correctly identified out of total labs in the file. Test#5 contained labs that were intentionally misspelled. Average corrected rate is 75%.

3. Test Results

Five test lab reports were processed and results were analyzed. The results are in Table 1. All test files were actual patient data. File modifications included removing some extraneous non-lab data. Test file 5 included chemistry labs where each lab name was mispelled. The flexible match algorithm does not attempt to match on short strings (less than 5 characters). Thererfore a mispelling such as "anion gup" was not matched. However, "sotium" was correctly matched with "sodium."

The accuracy of the Lab Parser ranged from 45% to 60%. Analysis revealed missing rules which were added and the files reparsed giving results that ranged from 73% to 87%. With each iteration of parsing and analysis, the rule library should improve. The goal of the project development is parse rate of 95% or greater.

4. Discussion

The Lab Parser prototype is complete and functioning. Assuming that lab files are consistent in their organization, the parser will be a useful system for our clinic. If lab file organization changes, the program can be adapted because knowledge is encoded in parsing rules and new rules can be written for novel syntax.

The complexity of parsing natural language has been reduced to polynomial time due to heuristics discussed above. Using keyword based rule selection, we employ highly specific rules and parsing is fast. We also use literal strings as terminal match targets.

The accuracy of the five test lab reports is less than the 95% goal. We plan to improve the system by new rules. The system cannot be utilized unless the accuracy is very high. Our goal is 95% parse rate.

The current system adds a new and useful component to our AI Neurology systems described in other publications [10, 11, 12, 13, 14].

5. Conclusion and Future Work

This prototype solves an important problem for our EMR and our clinic specifically digitizing lab reports. The system is fast and can easily be augmented by new rules. The current accuracy level is below our goal and will be improved by knowledge augmentation. Rule based systems are easily changed given the explicit nature of the knowlege programming.

References

1. Introduction to HL7 Standards. http://www.hl7.org/implement/standards/.
2. Gazdar, G, C. Mellish, 1989. *Natural Language Processing in LISP* (Addison-Wesley, 1989), pp. 118-120.
3. Pratt-Harmann. Computational Complexity in Natural Language, http://www.cs.man.ac.uk/~ipratt/papers
4. /nat_lang/handbook.pdf.
5. Steele, G. (1990) Common Lisp the Language, 2nd Edition, *Digital Press.*
6. Keene, S. (1989). Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS, *Addison Wesley*, ISBN: 0201175894.
7. ACL MYSQL Manual, https://franz.com/support/documentation/current/doc/mysql.htm.
8. PHP Manual, http://php.net/manual/en/index.php.
9. www.nuance.co.uk/for-business/by-product/omnipage/Guide_ENG-1.pdf.
10. McCray, Srinivas, Browne, 1994. Lexical Methods for Managing variation in Biomedical Terminologies, *the Proceedings of the 18th Annual Symposium on Computer Applications in Medical Care*, 1994, 235-239.
11. Sponsler, JL. 2012. Automated Analysis of Electromyography Data. *Proceedings of the IASTED Conference for Telehealth*, Innsbruck Austria, February 2012.
12. Sponsler J. 2013. NEUROBRIDGE: An AI development environment for neurology. *Proceedings of the 24th IASTED Conference on Modeling and Simulation*, Banff, Canada, July 17-19, 2013.
13. Sponsler J. 2013. PLEXBASE: A digital model of the brachial plexus. *Proceedings of the 24th IASTED Conference on Modeling and Simulation*, Banff, Canada, July 17-19, 2013.
14. Sponsler J, HPARSER: Extracting formal patient data from free text history and physical reports using natural language processing software. *Proceedings of the American Medical Informatics Association Annual Symposium*, Washington DC, Nov 3-11, 2001.
15. J. Sponsler, F. Pan, 2011. An electronic medical record for neurology. *Proceedings of IASTED Conference on Telehealth*, Feb 2012.