LaSEWeb: Automating Search Strategies over Semi-structured Web Data

Oleksandr Polozov*
University of Washington
polozov@cs.washington.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

ABSTRACT

We show how to programmatically model processes that humans use when extracting answers to queries (e.g., "Who invented typewriter?", "List of Washington national parks") from semi-structured Web pages returned by a search engine. This modeling enables various applications including automating repetitive search tasks, and helping search engine developers design micro-segments of factoid questions.

We describe the design and implementation of a domain-specific language that enables extracting data from a webpage based on its structure, visual layout, and linguistic patterns. We also describe an algorithm to rank multiple answers extracted from multiple webpages.

On 100,000+ queries (across 7 micro-segments) obtained from Bing logs, our system LaSEWeb answered queries with an average recall of 71%. Also, the desired answer(s) were present in top-3 suggestions for 95%+ cases.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*query languages*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering, search process, information filtering*

Keywords

Semi-structured data; domain-specific languages; structure extraction; Web programming; question answering.

1. INTRODUCTION

The creation of search engines changed the way people locate information on the Web. 91% of online adults currently use search engines to find information on the Web, 59% of them daily [21]. 84% of research undertaken by college students begins with a search engine query [3], and the same trend is observed for faculty and professional researchers [11]. Search engines dictate our habits of knowledge discovery on the Web.

Even though many of the queries submitted to search engines represent one-off tasks, a significant percent of them are repetitive. First, end-users often have to *process batch data* that requires making a series of similar queries to a search engine. A typical example is finding a similar piece of information about every item in a given list, e.g. acquiring contact information about a list of people, getting BibTeX entries for a list of articles, etc.

Second, so called *factoid questions* ("Who invented radio?", "What is the population of Germany?", etc.) still constitute a significant portion of all user queries in search engine logs [23]. Search engines have recognized the importance of factoid questions, and introduced a notion of a *micro-segment* of queries – a specific category of questions, for which the search engine shows the instant answer underneath the search bar, along with the list of search results. The source of data for answering micro-segment questions is typically a structured knowledge database, such as FreeBase. This implies that (a) the information presented in an answer is limited to the content of the database; (b) answer extraction code is hard-coded for every micro-segment; (c) time-sensitive information is not tracked consistently. In constrast, any desired knowledge is usually present in the free-text Web data, but it is usually unstructured or semi-structured (i.e., partially labeled to be accessible and recognizable by humans, but not by data collection algorithms).

Even when a search engine provides instant response, it may not be exactly what the end-user wants. According to [24], people prefer to locate their target in iterations, by observing the context of an answer and refining the query as necessary. They mostly look at multiple pages related to their desired answer, although seldom at more than ten [13]. Moreover, there may be multiple possible "answers" to a user's query, and the user is often interested in exploring the context related to each of them. For this, end-users like to explore the list of search results manually, since the task of *extracting and ranking multiple answer candidates along with their context* lies beyond the capabilities of current micro-segment search. This is the problem addressed in this paper.

All mentioned repeatable search tasks share the same generic features. A repeatable search task is a parameterized Web query that takes a tuple of string *arguments*, and returns a set of string *answers*. Essentially, a parameterized Web query is a *function* over the Web data, which is defined once, and later executed multiple times for various specific

*Work done during an internship at Microsoft Research.

KDD'14, August 24–27, 2014, New York, NY, USA.

ACM 978-1-4503-2956-9/14/08…$15.00.
http://dx.doi.org/10.1145/2623330.2623761 .

arguments. Currently, the only automated component in the execution of such a function is a search engine. The rest of the execution is defined by end-users' search strategies over the search engine results, and is entirely manual.

We observe that for any repeatable search task people follow similar strategies in extracting knowledge from the Web. They compose a search query aimed to bring a list of results with high recall, but not necessarily high precision (i.e. the desired knowledge is present among the suggested links, but it may not be ranked high enough by the search engine). People then navigate to the suggested links, and explore the content of webpages to build a list of answer candidates and associated context. In order to locate an answer candidate on a webpage, they use a set of patterns that form a basis of their *search strategy*. These patterns include:

- Linguistic patterns: sentence structure, semantic information, textual content of the webpage.
- Structural patterns: any relational or otherwise semi-structured information on the webpage, tables, lists.
- Visual patterns: layout and styling of the webpage, presentational attributes of its elements, emphasis.

These patterns serve as a way to create structure and add semantics on top of the semi-structured content of the Web.

In this paper, we present LaSEWeb[1] – a system for automating repetitive search tasks from end-users' descriptions of their search strategies involving the above-mentioned patterns. LaSEWeb consists of a novel domain-specific language (DSL) for programming search strategies, and an interpreter for it, built on top of the search engine Bing. A LaSEWeb program takes a tuple of user query arguments, and returns a set of answer strings from the Web, ranked by their confidence scores, along with their corresponding source URLs. It does so by (a) exploring the list of search results returned by Bing for our *seed query*, (b) executing a LaSEWeb *query* on each webpage in the list to extract multiple answer string representations from it, and (c) clustering these representations using an application-specific *similarity function*. The language of LaSEWeb queries includes textual, structural, and visual search patterns that we collected from our observations of typical search strategies. The interpreter for this language makes use of semi-structured content of a webpage and CSS attributes of HTML nodes along with state-of-the-art natural language processing (NLP) algorithms and programming by example (PBE) algorithms. We intend LaSEWeb to be used by both end-users and micro-segment developers.

We evaluated LaSEWeb on two different applications of Web programming: (a) 7 factoid micro-segments, represented by 100,000+ user queries from Bing logs, and (b) 5 repeatable academic search tasks, related to PLDI-2014 committee. The system achieved 95%+ precision in all micro-segments, and 73% precision in repeatable search, with an average recall of 71%.

This paper makes the following contributions:

- We define and motivate the problem of Web programming with two different applications (§2).
- We introduce LaSEWeb, our DSL for writing Web programs that incorporate humans' search strategies in a declarative format (§3).
- We provide an efficient interpreter for our DSL (§4). Besides other things, it makes use of novel *logical table parsing* techniques that utilize textual, structural, and visual features of a webpage to extract structural information from semi-structured data (§4.1.2).
- We evaluate LaSEWeb on two applications of Web programming: 7 factoid micro-segments and 5 batch search tasks (§5).

[1] **La**nguage for **S**tructure **E**xtraction.

2. MOTIVATION

2.1 Examples

We begin with two scenarios that illustrate the problem of Web programming. These scenarios are: factoid micro-segments in search engines, and repetitive Web searches.

Scenario 1. Alice is a software developer in a search engine department. She is responsible for a "micro-segment" search: providing instant answers to simple factoid questions underneath the search bar, along with a list of usual search results. She wants to implement a new micro-segment that can answer questions about inventors/creators of various products or discoveries. A fraction of this information exists in FreeBase and similar databases, but they lack data about (a) less fundamental inventions, like "paper clip", and (b) ambiguous attributions, like "radio" (N. Tesla vs. G. Marconi vs. A. Popov, etc.) Such information is available on the Web, in the list of provided search results. However, in order to discover any contextual information, a user has to navigate to every webpage in the list manually.

Instead of hard-coding FreeBase information, Alice might want to implement the following pseudocode, which represents her own search strategy in locating inventor infomation on the Web. This pseudocode describes a set of linguistic patterns that match sentences about inventors on webpages, returned by a search engine.

$w \leftarrow$ product name
$q \leftarrow w \circ$ "`inventor`"
$U \leftarrow$ search results for q
for all urls $u_j \in U$ **do**
  $\mathcal{Q}_1 \leftarrow$ pattern "$\mathsf{Entity(Person)} \sqcup * \sqcup \mathsf{Syn}(\texttt{invented}) \sqcup * \sqcup w$"
  $S \leftarrow \{$strings s within the content of u_j | s is an $\mathsf{Entity(Person)}$ component in a match of $\mathcal{Q}_1$ $\}$
  $\mathcal{Q}_2 \leftarrow$ pattern "$\mathsf{Entity(Person)} \sqcup * \sqcup \mathsf{Syn}(\texttt{creator}) \sqcup \mathsf{POS(Prep)} \sqcup w$"
  $S \leftarrow S \cup \{$strings s within the content of u_j | s is a $\mathsf{Entity(Person)}$ component in a match of $\mathcal{Q}_2$ $\}$
  ...
  Assume more confidence in the more frequent results in S
return all collected results with confidence scores and URLs

In this pseudocode $\mathsf{Syn}(s)$ matches any word that is synonymic to s, $*$ matches any number of words, $\mathsf{POS(Prep)}$ matches a preposition, $\mathsf{Entity(Person)}$ matches several words that constitute a person name (such as "John Atanasoff"). $\circ$ denotes string concatenation.

When executed against a user query $w =$ "`computer`", this program produces a set of results, similar to the following:

Answer	Confidence	Sources

John Atanasoff	14.5%	`http://www.computerhope.com`
		`http://www.ehow.com`
		`http://inventors.about.com`

Charles Babbage	10.2%	`http://www.buzzle.com` `http://www.ask.com`
...	...	...

We note that with this approach, *every* micro-segment turns into a "repetitive search": a parameterized search task that is defined once by a search engine developer, and executed multiple times with different arguments by end-users. The input of this search task are micro-segment arguments –

Figure 1: Screenshot of the personal webpage of Dr. Shaz Qadeer[2] (retrieved in October 2013).

for example, for "inventor" micro-segment this is the product name. The output of the task are answers to the factoid question of the micro-segment, ranked by confidence scores, along with their source URLs for additional context.

Scenario 2. Sofus has a table of KDD 2014 committee member names, which he wants to fill in with their corresponding phone numbers. Phone numbers can be extracted from their personal webpages automatically via regular expressions, but there are two challenges: (a) these webpages have to be identified first; (b) regular expressions may have false positives. Consider a webpage in Fig. 1. A simple regular expression for phone numbers will match both "Phone" and "Fax" lines on the page. Moreover, on different webpages it might also match a secretary's phone number, lab numbers, university numbers, etc. Also, retrieval of all personal webpages is still manual and tedious.

Without any automation, a human typically looks for a phone number on a personal webpage using visual and linguistic cues: proximity to header, words like "Phone" or "Fax", string pattern (regular expressions), table-like structure "⟨field name⟩: ⟨value⟩". The following pseudocode describes this manual search strategy:

$w_i \leftarrow i^{\text{th}}$ committee member's name
$q_i \leftarrow w_i \circ$ "`phone number`"
$U \leftarrow$ search results for q_i
for all urls $u_j \in U$ **do**
  $S \leftarrow$ {strings s within the content of u_j | s appears close to w_i or a picture $\wedge$ w_i is emphasized (bold/colored) $\wedge s$ is a value in a table-like structure where key is similar to "Phone"/"Voice" $\wedge$ s matches a phone number }
  Assume more confidence in the more frequent results in S
return all collected results with confidence scores and URLs

We note that the information on a webpage could be presented in an entirely different format: as a series of lines with a separator other than a colon, as an HTML table, with different layout, etc. However, people interpret any of these formats as a generic "table-like" structure, and locate information in them in a similar manner.

In this scenario, the repetitive search task lies in collecting similar knowledge for each item in the list. The input is a person's name, and the output is the set of found phone numbers. There may be multiple "correct" answers (e.g., home/cell phone number).

2.2 Observations

Examples in §2.1 show two different applications of parameterized Web queries, and expose several features that most people share in their search strategies. We summarize the list of our observations about these features below. They form the foundation for the definition of the problem of Web programming (which we tackle in this paper), and the LaSEWeb system (which is our approach to solving it).

2.2.1 Problem definition

Repetitive search tasks are parameterized. People often need to use the same strategy for a series of similar Web search tasks with different parameters. Scenario 2 showed a case of an end-user dealing with a single problem of finding a phone number, applied to multiple people. In Scenario 1 the task is defined by a micro-segment developer, and later used multiple times by different end-users.

Search tasks have multiple answers. Depending on the nature of the search task, there may be multiple pieces of information that should be extracted from the Web as possible "answers". A user is often interested in (a) exploring different answers ranked by some measure of their respective confidence/relevance, and (b) exploring some context related to the various answers.

These observations motivate our problem definition:

Definition 1. A *Web program* is a process that takes a tuple of user *query arguments*, and returns a set of *answer strings*, ranked by their confidence, along with corresponding *source URLs* for every answer string.

A Web program is essentially defined as a *function* from strings to sets of strings. The *execution* of such a function happens over data on the Web. The *definition* of such a function consists of various typical patterns that constitute humans' *search strategy*. Both definition and execution of Web programs are currently entirely manual, with the search engine being the only automatic bridge between the user's goal and the Web data; our LaSEWeb system automates the execution of such Web programs.

2.2.2 Our approach

We propose LaSEWeb, our approach to Web programming. It implements a Web program by automating a typical human's search strategy for it. The automation is based on our observations about common features of such search strategies, summarized below.

Exploring multiple webpages. When there are multiple "correct" answers to the question, people explore multiple webpages in order to adjust their confidence in some of these answers. In order to retrieve a list of relevant webpages to explore, an end-user typically starts with a query to a search engine. This initial query is expected to yield a set of results with high recall but possibly low precision. A simple combination of arguments does not necessarily constitute a good initial query for a search engine: based on the application, the user might want to use additional keywords or some features of a search engine front-end in the query.

In our approach, every LaSEWeb program contains a *seed query builder* – a function that composes the initial query to a search engine (§3.1).

Answers have multiple representations. A single answer can be found on the Web in multiple string representations. For example, a person's name can be written with or without middle initial, with given name or family name first, with or without proper capitalization, etc. The nature of such multiple representations depends on the application.

[2] http://research.microsoft.com/en-us/people/qadeer/

We *cluster* multiple representations of the same answer together using a *similarity function* that defines an application-specific logic of answer similarity (§3.1). We then rank the resulting clusters by our measure of answer confidence, and choose a *representative answer* for every cluster (§4.2).

People look for patterns. When looking for an answer on a particular webpage, people use a set of patterns that locate relevant information within the content of the webpage. These patterns can be categorized as follows:

1. **Linguistic patterns**, as in Scenario 1, use syntax and semantic characteristics of text to find appropriate matches within the content of a webpage. The required answer of a LaSEWeb query typically exists at this lowest level surrounded by text that we can fuzzily describe using linguistic patterns. End-users usually do not build such patterns consciously, but domain experts can easily extrapolate their search strategy into simple linguistic patterns, as was shown in Scenario 1.
2. **Structural patterns**, as in Scenario 2, use structured or semi-structured content of the webpage to extract required information according to the implicit schema. The majority of text content on the Web is not structured, but rather semi-structured: tables are not normalized, and often not even marked as a table. However, people identify any of the following as a "table":
   - HTML `<table>`, `<ul>`, or `<ol>`.
   - plain text with separators (e.g. colon).
   - block elements, spatially aligned in a table-like structure.
3. **Visual patterns**, as in Scenario 2, use spatial layout of the webpage, colors, proximity and other stylistic features to locate relevant information. The exact instantiation of these patterns (i.e., specific colors or layout) differs among webpages, but human brain is capable of checking them in a generic manner.

We present an appropriate query language that includes linguistic, structural, and visual patterns of search strategies (§3.2). This language is the main component of LaSEWeb.

3. LASEWEB LANGUAGE

In this section, we present the language of LaSEWeb. §3.1 introduces LaSEWeb *programs* – our representation of parameterized queries, described and motivated before in §2. We describe the syntax and semantics of LaSEWeb *queries*, the main component of LaSEWeb programs, in §3.2. The implementation of our interpreter for LaSEWeb programs is discussed in §4.

3.1 LaSEWeb programs

Every LaSEWeb *program* $\mathcal{P}$ represents a single parameterized query that takes a tuple of user query arguments $\vec{v}$ and returns a set of answer strings, annotated with their confidence scores along with their source URLs. The execution of a LaSEWeb program $\mathcal{P}$ on a tuple of user query arguments $\vec{v}$ is defined as $\mathcal{P}\colon \vec{v} \mapsto \{\langle a_i, \beta_i, U_i\rangle\}$, where a_i is the i^{th} *answer string*, β_i is its *confidence score*, and U_i is the set of its *source URLs*. Higher confidence scores correspond to more relevant answers.

The syntax and semantics of LaSEWeb programs is based on our observations, summarized in §2.2. A LaSEWeb program $\mathcal{P}$ first constructs a *seed query* for Bing, executes a LaSEWeb *query* on each of the returned search results, extracts the desired *answer strings* from the matches of the LaSEWeb query, and clusters these answer strings together, according to the *similarity function*.

Definition 2. A LaSEWeb program $\mathcal{P}$ is parameterized by $\vec{v}$ that is replaced by a tuple of user-provided arguments at runtime for every specific search invocation. It is defined as a tuple $\langle q, \sigma, \mathcal{Q}, \ell_a\rangle$, where:

- q is a *seed query builder* function, which builds an initial query for Bing from $\vec{v}$. This query should be likely to produce a set of relevant links with high recall.
- σ is a *similarity function*, which compares two answer strings from the Web, and determines whether they describe the same "answer" in the application logic.
- $\mathcal{Q}$ is a LaSEWeb *query*, which describes a set of patterns that humans use to find this kind of information on the Web. As a result of its execution, $\mathcal{Q}$ matches none or many answer strings on a single webpage.
- ℓ_a is a label in $\mathcal{Q}$ that specifies what subexpression of the match is extracted as an "answer" (this is similar to named capturing groups in regular expressions).

Example 1. *In Scenario 1 the "inventor" micro-segment is described with* LaSEWeb *program* $\mathcal{P} = \langle q, \sigma, \mathcal{Q}, \ell_a\rangle$*, where:*

- $q(w) = w \circ$ "*inventor*"
- $\sigma(w_1, w_2) = \mathsf{true}$ *iff* w_1 *and* w_2 *describe the same person name (e.g., "Charles Babbage" and "C. Babbage").*
- $\mathcal{Q}$ *is the* LaSEWeb *query equivalent to* $\mathcal{Q}_1 \vee \mathcal{Q}_2$ *in Scenario 1. We show this query in Example 4.*
- ℓ_a *is a label of the subexpression* $\mathsf{Entity}(\mathsf{Person})$ *in* $\mathcal{Q}$.

Specific search invocations of $\mathcal{P}$ *apply it on specific user parameters, such as* $\vec{v} =$ ("*computer*")*, or* $\vec{v} =$ ("*paper clip*").

The rest of this section focuses on the syntax and semantics of the main component of a LaSEWeb program – a LaSEWeb query $\mathcal{Q}$. Our interpreter for LaSEWeb programs, which makes use of the other components as well (q, σ, and ℓ_a), is described in §4.

3.2 LaSEWeb queries

A LaSEWeb query $\mathcal{Q}$ is executed against a webpage. A *webpage* is a tree of HTML nodes $\mathcal{N}$. For each HTML node $\mathcal{N}$ we define two auxiliary functions $\mathsf{BBox}(\mathcal{N})$ and $\mathsf{Text}(\mathcal{N})$, which are used by LaSEWeb to determine the result of the execution. $\mathsf{BBox}(\mathcal{N})$ returns a rectangle b that is the smallest bounding box of node $\mathcal{N}$ on the page, when rendered by a browser (assuming a fixed resolution). $\mathsf{Text}(\mathcal{N})$ is a string that is a displayed textual content of $\mathcal{N}$, with all HTML tags stripped off. The result of executing a LaSEWeb query $\mathcal{Q}$ against such a webpage is a multi-set of possible *answer strings*, each labeled with some label ℓ – the subexpression of $\mathcal{Q}$ that matched this answer string. A subset of this multi-set that is labeled with the "answer label" ℓ_a in the definition of $\mathcal{P}$, is selected for the final answer set.

Fig. 2 shows the syntax of LaSEWeb. A LaSEWeb query $\mathcal{Q}$ unites three types of expressions: *visual expressions* $\mathcal{B}$, *structural expressions* $\mathcal{S}$, and *linguistic expressions* $\mathcal{L}$. They are aligned in a *hierarchical structure* of three layers that match separate webpage elements:

- The topmost layer of visual expressions describes webpage elements with particular stylistic propeties. They make use of presentational attributes of an HTML node, its CSS, position, and bounding box. A visual expression $\mathcal{B}$ is executed against a *bounding box* b.

(a)

$$
\begin{array}{rl}
\text{Linguistic expression } \mathcal{L} & := \mathsf{Ling}(\mathcal{E}, \Phi) \mid \mathcal{L}_1 \vee \mathcal{L}_2 \\
\text{Linguistic pattern } \mathcal{E} & := \mathcal{E}^+ \mid \mathcal{E}^* \mid \mathcal{E}? \mid \mathcal{E}_1\ \mathcal{E}_2 \mid \ell\colon \mathcal{E} \mid \mathsf{Word} \\
& \quad \mid \mathsf{ConstWord}(s) \mid \mathsf{ConstPhrase}(s_1, \ldots, s_k) \\
& \quad \mid \mathsf{Syn}(s) \mid \mathsf{POS}(p) \mid \mathsf{Entity}(e) \mid \mathsf{NP} \mid \ldots \\
\text{Linguistic constraint } \Phi & := \mathsf{SameSentence}(\ell_1, \ell_2) \mid \mathsf{Regex}(\ell, s) \mid \ldots \\
& \quad \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi \mid \mathsf{true} \mid \mathsf{false} \\
\text{String } s & := w \mid v_k \\
\text{ID labels } \ell, \eta & := w \\
\text{Part of speech } p & \in \{\mathsf{Noun}, \mathsf{Verb}, \mathsf{Prep} \ldots\} \\
\text{Entity type } e & \in \{\mathsf{Person}, \mathsf{Org}, \mathsf{Place} \ldots\}
\end{array}
$$

$$
\begin{array}{rl}
\text{LaSEWeb query } \mathcal{Q} & := \mathsf{FW}(\mathcal{B}, \Psi) \mid \mathcal{Q}_1 \vee \mathcal{Q}_2 \\
\text{Visual expression } \mathcal{B} & := \mathcal{S} \mid \mathsf{Union}(\mathcal{B}_1, \mathcal{B}_2) \mid \eta\colon \mathcal{B} \\
\text{Visual constraint } \Psi & := \mathsf{Nearby}(\eta_1, \eta_2) \mid \mathsf{Emphasized}(\eta) \mid \mathsf{Layout}(\eta_1, \eta_2, d) \mid \ldots \\
& \quad \mid \Psi_1 \wedge \Psi_2 \mid \Psi_1 \vee \Psi_2 \mid \neg\Psi \mid \mathsf{true} \mid \mathsf{false} \\
\text{Structural expression } \mathcal{S} & := \mathsf{Leaf}(\mathcal{L}) \mid \mathsf{VLOOKUP}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3) \mid \mathsf{AttrLookup}(\mathcal{L}_1, \mathcal{L}_2) \\
\text{Direction } d & \in \{\mathsf{Up}, \mathsf{Down}, \mathsf{Left}, \mathsf{Right}\}
\end{array}
$$

(b)

$$
\mathsf{Ling}(\mathcal{E}, \mathsf{true}) \equiv \mathcal{E} \qquad \mathsf{FW}(\mathcal{B}, \mathsf{true}) \equiv \mathcal{B} \qquad \ell\colon \mathsf{Word} \equiv \ell \qquad \mathsf{Word}^* \equiv *
$$

$$
\mathsf{ConstWord}(s) \equiv \text{“}s\text{”} \qquad \mathsf{ConstPhrase}(s_1, \ldots, s_k) \equiv \text{“}s_1 \sqcup \ldots \sqcup s_k\text{”}
$$

Figure 2: (a) LaSEWeb query language syntax. v_k denotes the k^{th} argument in a tuple of user query arguments $\vec{v}$, k denotes an integer constant, and w denotes a string constant. (b) Common shorthands in LaSEWeb, used for brevity.

- The middle layer of structural expressions describes a structural pattern of information within any *logical tables* in a webpage. A structural expresion $\mathcal{S}$ is executed against an HTML node. $\mathcal{S}$ is the simplest form of a visual expression: it matches a bounding box b if $\mathcal{S}$ matches a node $\mathcal{N}$, encompassed by b.
- The bottom layer of linguistic expressions $\mathcal{L}$ describes a string pattern using syntactic and semantic matching constructors. $\mathcal{L}$ is the simplest form of a structural expresion. It is executed against an input string t, which is the content $\mathsf{Text}(\mathcal{N})$ of some HTML node $\mathcal{N}$.

Together these three expression types cover the entire range of typical patterns of search strategies, defined in §2.2.

A LaSEWeb query $\mathcal{Q}$ is parameterized with a tuple of parameters $\vec{v}$, which are replaced by user-provided query arguments v_k (before its execution on a webpage). Thus, in Fig. 2 every string (which is not a label name) can be either a constant string literal w, or a user query argument v_k.

We begin with an explanation of linguistic expressions in §3.2.1, and proceed with structural expression in §3.2.2, and visual expressions in §3.2.3. Examples 4 and 5 give examples of full LaSEWeb queries on applications from §2.

3.2.1 Linguistic expressions

A *linguistic expression* $\mathcal{L}$ consists of a *linguistic pattern* $\mathcal{E}$ and a *linguistic constraint* Φ. Here linguistic patterns, when executed against an input string t, collect multiple matches M, and linguistic constraints filter out matches with undesired properties.

A linguistic pattern $\mathcal{E}$ is executed against an input string t. In our system, we consider strings as lists of *tokens*. A token is a natural language primitive (word, punctuation, etc.).

The result of the execution of $\mathcal{E}$ on t is a set W of *linguistic matches* M. Every linguistic match M is a mapping from ID labels ℓ of $\mathcal{E}$'s subexpressions to *matched substrings* of t. Intuitively, $\mathcal{E}$ acts similarly to a regular expression: when $\mathcal{E}$ finds a succesful match within t, every subexpression of $\mathcal{E}$ corresponds to some substring t' of t. Some of these subexpressions $\ell\colon \mathcal{E}$ are explicitly marked with labels ℓ; this is similar to named capturing groups in regular expressions.

Linguistic patterns are similar to regular expressions in a sense of their composition methods. They are divided into three kinds as described below.

Primitive. A linguistic pattern Word matches any single token. We usually denote $\ell\colon \mathsf{Word}$ simply as ℓ, since label ℓ is essentially a variable that captures a word match in the mapping M. Linguistic patterns $\mathsf{ConstWord}(s)$ and $\mathsf{ConstPhrase}(s_1, \ldots, s_k)$ match a fixed token s or a fixed sequence of tokens "$s_1 \sqcup \ldots \sqcup s_k$", respectively, in t.

Composite. Operators $+$, ?, and $*$ (Kleene star) borrow their semantics from regular expressions. Each of them has a single linguistic pattern as a subexpression, and their resulting mappings M are composed from the matches found in subexpressions. For $+$ and $*$, whose subexpression matches multiple times in the substring, we put all the found mappings in the final linguistic match M. Thus, the same label ℓ may occur multiple times in the linguistic match M.

Linguistic predicates. Most of the forms of linguistic patterns are *predicates*, which match a token or a sequence of tokens only if it satisfies some linguistic property. We show several representatives of linguistic predicates in Fig. 2; more can be implemented, assuming the existence of corresponding NLP algorithms. In this paper we use the following representative predicates:

- $\mathsf{POS}(p)$: matches a token if its part of speech is p.
- $\mathsf{Entity}(e)$: matches a sequence of tokens, classified as a *named entity* of type e (person name, place, etc.).
- NP: matches a sequence of tokens if it constitutes a *noun phrase* in the parse tree of the sentence.
- $\mathsf{Syn}(s)$: matches a single token if it is synonymic to s.

We assume the existence of corresponding functions $\mathsf{IsNP}(s)$, $\mathsf{PosValue}(s)$, $\mathsf{AreSynonyms}(s_1, s_2)$, $\mathsf{EntityValue}(s)$, which implement required NLP algorithms on strings. §4 discusses our implementation of these functions.

Linguistic constraints. Every match M, returned by a linguistic pattern $\mathcal{E}$, is filtered through a linguistic constraint Φ. The final set of matches W, returned by a linguistic expression $\mathcal{L}$, consists only of those matches M that haven't been filtered out by Φ.

We present two representatives of linguistic constraints:

- $\mathsf{SameSentence}(\ell_1, \ell_2)$ returns true iff matches captured by ℓ_1 and ℓ_2 belong to the same sentence within t.
- $\mathsf{Regex}(\ell, s)$ returns true iff the match captured by ℓ satisfies the regular expression s.

Example 2. *The "inventor" micro-segment in Scenario 1 can be described with the following linguistic expression:*

$\mathcal{L} = \mathsf{Ling}(\mathcal{E}_1, \Phi) \vee \mathsf{Ling}(\mathcal{E}_2, \Phi)$ *where:*

$\mathcal{E}_1 = (\ell_{ans}\colon \mathsf{Entity}(\mathsf{Person})) \sqcup * \sqcup \mathsf{Syn}(\text{“}\texttt{invent}\text{”}) \sqcup * \sqcup (\ell_q\colon v_1)$

$\mathcal{E}_2 = (\ell_q\colon v_1) \sqcup \mathsf{Syn}(\text{“}\texttt{created}\text{”}) \sqcup \mathsf{POS}(\mathsf{Prep}) \sqcup * \sqcup (\ell_{ans}\colon \mathsf{Entity}(\mathsf{Person}))$

$\Phi = \mathsf{SameSentence}(\ell_{ans}, \ell_q)$

Suppose the specific search invocation looks for the inventor of typewriters. We set $\vec{v} = (\text{“}\texttt{typewriter}\text{”})$ *and apply the resulting query to strings found on the pages returned by Bing. One of these strings is* $t =$ "In 1714, a patent to something like a typewriter was granted to a man named Henry Mill in England, but no example of Mills' invention survives. Finally, in 1867, a Milwaukee, Wisconsin printer-publisher-

politician named Christopher Latham Sholes, with assistance from Carlos Glidden and Samuel Soule, patented what was to be the first useful typewriter."[3]

$\mathcal{E}_2$ will capture no matches. $\mathcal{E}_1$, however, will capture 4 matches. In each of them ℓ_q will capture the word "typewriter", and ℓ_{ans} will capture named entities "Henry Mill", "Christopher Latham Sholes", "Carlos Glidden", "Samuel Soule", respectively. All of them are possible matches of $\mathcal{E}_1$, because there is a word "patented", the synonym of "invent", between them and the last occurence of "typewriter" in t. However, the first match will be eliminated by Φ, because captures of ℓ_{ans} and ℓ_q occur in different sentences. The final result of $\mathcal{L}$ is a multi-set W of three matches. Out of them, all three can be considered "inventors" of the typewriter, although one (Christopher Latham Sholes) is arguably more relevant than two others.

We note several features of LaSEWeb in this example:

- The optimal structure of $\mathcal{E}$ can be easily deduced from a single look at an example of a desired answer on the Web, and refined later in 1-2 iterations, based on the execution results.
- Most of the features of the desired answer strings can be generally described with just 2-3 linguistic predicates in LaSEWeb.
- Multiple ways to extract an intended answer can be expressed as a disjunction of linguistic patterns.
- Linguistic constraints Φ and linguistic predicates within $\mathcal{E}$ filter out false positives. Without them, more erroneous matches would be found by $\mathcal{L}$ on the webpage.
- Not all strings in the resulting answer set are equally likely to serve as "the answer" to the question. Therefore, our interpreter for LaSEWeb programs, described in §4, assigns confidence scores to answer strings based on their relative frequency on the Web.

3.2.2 Structural expressions

A structural expression $\mathcal{S}$ is executed against an HTML node $\mathcal{N}$ and returns an answer set W. Two representative constructors of structural expressions, $\mathsf{AttrLookup}(\mathcal{L}_1, \mathcal{L}_2)$ and $\mathsf{VLOOKUP}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$, describe two structural patterns that arise in the real-word webpages. A structural pattern is a particular alignment of structured information on a webpage, such as a relational table or a list of attributes. For such structured forms we can use any relational techniques (of which VLOOKUP and AttrLookup are two examples) to select the required information from the structure.

In real-world webpages, the information is seldom present in a clean tabular format. However, the information is often *semi-structured*: it follows a recognizable pattern, from which the implicit tabular structure can be recovered. For example, consider a list of "⟨attribute⟩: ⟨value⟩" lines, each presenting an attribute and its value, separated by a colon. Such a list can be interpreted as a 2-column table of attributes along with their corresponding values. We call such implicit tables *logical tables* and assume the existence of a $\mathsf{Tables}(\mathcal{N})$ function that returns a set of logical tables present in the HTML node $\mathcal{N}$. Each table T is indexed from 1 through the number of rows/columns. Each cell $T[j,k]$ is the textual content of the corresponding logical cell (e.g. the content of a `<td>` node, or a substring of an "⟨attribute⟩: ⟨value⟩" line in a paragraph, etc.). We discuss our implementation of $\mathsf{Tables}(\mathcal{N})$ in §4.

[3] http://ideafinder.com/history/inventions/typrwriter.htm

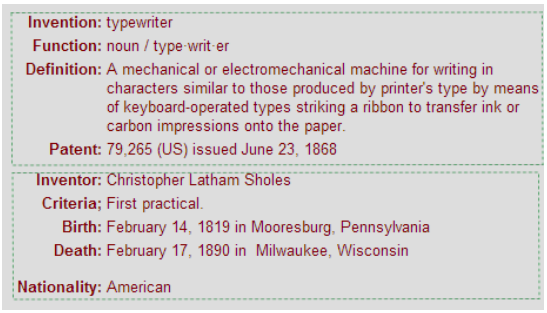

Figure 3: Textual attribute table about the typewriter inventor. Two separate HTML `<table>`s are highlighted.

Constructor $\mathsf{VLOOKUP}(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$ represents a Microsoft Excel `VLOOKUP` operation. Its arguments are the *key expression* $\mathcal{L}_1$, the *header expression* $\mathcal{L}_2$, and the *content expression* $\mathcal{L}_3$. Given a logical table T, if $\mathcal{L}_1$ matches a cell $T[j,k']$ in j^{th} row, and $\mathcal{L}_2$ matches a header cell $T[1,k]$ in k^{th} column of the first row, LaSEWeb executes the content expression $\mathcal{L}_3$ on the intersection cell $T[j,k]$, and returns the result.

Constructor $\mathsf{AttrLookup}(\mathcal{L}_1, \mathcal{L}_2)$ represents a *two-column attribute lookup* operation. Its arguments are the *attribute expression* $\mathcal{L}_1$ and the *content expression* $\mathcal{L}_2$. Given a logical table T of ≥ 2 columns where $\mathcal{L}_1$ matches any cell $T[j,1]$ in the j^{th} row of the first column (attribute name), LaSEWeb executes the content expression $\mathcal{L}_2$ on the neighboring cell $T[j,2]$, and returns the result.

Example 3. *Fig. 3 shows an example of a logical table about the inventor of a typewriter, taken from the Web. This table is not a 2-column `<table>` tag, as it might appear to the viewer. Instead, it is represented as two `<table>` tags (highlighted in green), with pieces of plain text information within. Nevertheless, if $\mathcal{N}$ is a `<div>` containing both `<table>`s, our function $\mathsf{Tables}(\mathcal{N})$ extracts a single 2-column attribute table from it.*

The following structural expression is used to extract information from such logical table:

$$\mathcal{S} = \mathsf{AttrLookup}(\mathsf{Syn}(\text{“}\textit{inventor}\text{”}),\ \mathsf{Entity}(\mathsf{Person}))$$

*This expression is not parameterized with any user arguments, it extracts the attribute value for **any** "inventor" row. In order to bind $\mathcal{S}$ to the particular invention ("typewriter"), we introduce additional constraints on it in Example 4.*

3.2.3 Visual expressions

A visual expression $\mathcal{B}$ is executed against a *bounding box* b. We use bounding boxes instead of HTML nodes here, because not every visually distinguishable webpage element is represented by a single HTML node. We made LaSEWeb accessible to end-users, according to our observations of their search strategies. As §2.2 shows, they often make use of the stylistic properties of the webpage, *as seen by end-users*.

The result of the execution of $\mathcal{B}$ is a tuple of a linguistic answer set W, matched by the lower subexpressions, and a *visual match* V. A visual match is a mapping of ID labels of visual expressions η to bounding boxes b.

Constructor $\mathsf{Union}(\mathcal{B}_1, \mathcal{B}_2)$ matches a union of two bounding boxes $\mathsf{unite}(b_1, b_2)$ if its subexpressions $\mathcal{B}_1$ and $\mathcal{B}_2$ match b_1 and b_2, respectively. The union of two bounding boxes is the smallest rectangle that contains both b_1 and b_2.

Visual constraints. The top-level LaSEWeb query $\mathcal{Q}$ consists of a visual expression $\mathcal{B}$ with a *visual constraint* Ψ. Similarly to linguistic constraints, visual constraints filter

out false positive matches V, returned by the execution of $\mathcal{B}$. We present three representatives of visual constraints here:

- Constraint $\mathsf{Nearby}(\eta_1, \eta_2)$ checks the bounding boxes $V[\eta_1]$ and $V[\eta_2]$ for proximity. It compares the distance between them with a predefined relative *proximity threshold* δ.
- Constraint $\mathsf{Layout}(\eta_1, \eta_2, d)$ checks whether bounding boxes $b_1 = V[\eta_1]$ and $b_2 = V[\eta_2]$ are aligned according to the layout d. In other words, b_1 should lie within a sector defined by the center of b_2 and the two corners of b_2 in the direction d.
- Constraint $\mathsf{Emphasized}(\eta)$ checks whether the content of the bounding box $V[\eta]$ is emphasized with respect to its surrounding elements. This includes colors, fonts, font sizes, headers (`<h1>` through `<h6>`), etc.

EXAMPLE 4. *The complete* LASEWEB *query for the "inventor" micro-segment is shown below:*

$\mathcal{Q} = \mathsf{FW}(\mathsf{Union}(\eta_v : \mathsf{Leaf}(\mathsf{Syn}(v_1)),\ \eta_t : \mathcal{S}),\ \mathsf{Nearby}(\eta_v,\ \eta_t)) \vee \mathsf{Leaf}(\mathcal{L})$

where $\mathcal{L}$ *is the linguistic expression from Example 2*

$\mathcal{S}$ *is the structural expression from Example 3*

EXAMPLE 5. *Scenario 2 shows a repetitive task of collecting phone numbers from multiple personal contact pages of researchers, similar to the one shown in Fig. 1. The following* LASEWEB *query collects the desired phone number(s) from such a webpage:*

$\mathcal{Q} = \mathsf{FW}(\mathsf{Union}(\eta_t : \mathsf{Leaf}(v_1),\ \eta_b : \mathcal{S}_b),\ \Psi)$

$\Psi = \mathsf{Layout}(\eta_t, \eta_b, \mathsf{Down}) \wedge \mathsf{Nearby}(\eta_t, \eta_b) \wedge \mathsf{Emphasized}(\eta_t)$

$\mathcal{S}_b = \mathsf{AttrLookup}(\mathsf{Syn}(\text{“}\texttt{phone}\text{”}),\ \mathcal{L}_a)$

$\mathcal{L}_a = \mathsf{Ling}(\ell,\ \mathsf{Regex}(\ell,\ \text{“}\texttt{\(?\d+\)?\W\d+\W\d+}\text{”}))$

When $\mathcal{Q}$ *is executed with the arguments* $\vec{v} = (\text{“}\texttt{Shaz Qadeer}\text{”})$, *it matches any phone number in a logical attribute table such that "Shaz Qadeer" can be found above the table within the proximity threshold, and is somehow emphasized.*

4. IMPLEMENTATION

In this section, we discuss our implementation of the interpreter for LASEWEB programs. It consists of two important components: the interpreter for LASEWEB queries $\mathcal{Q}$, and the execution engine for the entire LASEWEB program $\mathcal{P}$. We discuss important aspects of the LASEWEB query interpreter implementation in §4.1, and present our LASEWEB program execution algorithm in §4.2.

4.1 Query interpreter

4.1.1 Linguistic expressions

The key ideas in our interpretation of linguistic expressions are: (a) usage of state-of-the-art NLP algorithms, and (b) our refinement of the semantics of $\mathcal{E}$ for better performance on large webpages, called *anchoring*.

NLP algorithms. To support interpretation of linguistic predicates, our implementation requires algorithms for the functions $\mathsf{IsNP}(s)$, $\mathsf{PosValue}(s)$, $\mathsf{EntityValue}(s)$. They are implemented using state-of-the-art algorithms in NLP: Stanford CoreNLP [7, 15, 26] and MSR SPLAT [22] for named entity recognition, syntactic parsing, and part-of-speech tagging. To implement the $\mathsf{AreSynonyms}(s_1, s_2)$ function, we used S. W.-t. Yih's PILSA word synonymy library [27].

Anchoring. A naïve implementation of matching a linguistic expression $\mathcal{E}$ against a string t would try to find a linguistic match for every possible starting token in t. This

(a)

(b)

```
County Name,City,Zip Code

Tarrant,Fort Worth,76114
Tarrant,Fort Worth,76115
Tarrant,Fort Worth,76116
Tarrant,Grand Prairie,75050
Tarrant,Grand Prairie,75051
Tarrant,Grand Prairie,75052
Tarrant,Grand Prairie,76018
Tarrant,Grand Prairie,76063
Tarrant,Grand Prairie,76065
Tarrant,Grapevine,76034
Tarrant,Grapevine,76051
```

Figure 4: (a) A logical table on the Amazon website, built with a grid of `<div>` elements (shown in green). (b) A logical table, built in plain text with commas as column separators.

is extremely inefficient, especially when dealing with Kleene star operator $*$. Instead, we *anchor* our matching procedure, identifying a set of constant strings (*anchors*), if any, that *must* match as particular subexpressions of $\mathcal{E}$. If anchors are given, we start the matching with a linear search for anchor occurences in text, and proceed with matching other subexpressions of $\mathcal{E}$ around every occurence. If no anchors are given, the algorithm resorts to naïve matching.

4.1.2 Structural expressions

The key idea in our interpretation of structural expressions is the $\mathsf{Tables}(\mathcal{N})$ function, which implements *logical table detection*. Its goal is to extract any information from the HTML node $\mathcal{N}$ that is likely to be *perceived* as a table by an end-user. We implement three basic strategies for detecting logical tables within an HTML node:

HTML tables. A `<table>` tag defines a logical table unless it is used for webpage layout (i.e. has multiple non-primitive pieces of content within it, possibly nested tables).

Visual tables. Any grid of `<div>`s or similar block elements constitutes a logical table if its bounding boxes are precisely aligned in a grid structure. Fig. 4a shows an example of a logical table built with `<div>` elements. We detect such tables by collecting a tree of bounding boxes of all HTML nodes within $\mathcal{N}$, and then building grids of those bounding boxes that are aligned next to each other.

Plain text tables. Often the information is presented as a table by marking it with punctuation signs and separators in plain text, instead of HTML tags. Fig. 1 and Fig. 3 show examples of such plain text tables, with colon being a separator between two columns. Fig. 4b shows another example, where the intended table is relational, and the separator between the columns is comma.

The simplest approach for detecting plain text tables would be to maintain a set of common separators (i.e. a colon, a comma, a space, etc.), and attempt to split entire paragraphs of text by them, picking the one that gives most results. However, even if the separator is "correct" (i.e., it actually splits the logical columns in the paragraph), it can also be present within some of the logical cells. For example, if Fig. 4b used whitespace instead of comma for column

function SEARCH($\mathcal{P} = \langle q, \sigma, \mathcal{Q}, \ell_a \rangle$, $\vec{v}$)

1: $U \leftarrow$ the results of Bing on $q(\vec{v})$
2: Substitute v_k in $\mathcal{Q}$ with values from $\vec{v}$
3: $\mathcal{C} \leftarrow \emptyset$ // *set of clusters,* $C_i = \{\langle s_k, \{u_j\}_{j=1}^{n_{ik}} \rangle\}_{k=1}^{m_i}$
4: **for all** URLs $u_j \in U$ **do**
5: $\mathcal{N} \leftarrow$ the `<body>` node of u_j
6: $S_j \leftarrow \{M[\ell_a] \mid M$ is the result of executing $\mathcal{Q}$ on $\mathcal{N}\}$
7: **for all** answer strings $s_k \in S_j$ **do**
8: $C_j \leftarrow \{\langle s_k, \{u_j\} \rangle\}$
9: **for all** $C \in \mathcal{C}$ such that $\exists s' \in C$: $\sigma(s_k, s')$ **do**
10: Merge C_j with C
11: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_j\}$
12: **for all** final clusters $C_i \in \mathcal{C}$ **do**
13: $a_i \leftarrow$ the most frequent string representation $s_k \in C_i$
14: $\beta_i \leftarrow \frac{1}{|U|} \sum_{j=1}^{|U|} \sum_{s \in C_i} \frac{c(s,u_j)}{|S_j|}$ where:
$c(s, u_j) = \#$ of times s was found at URL u_j
15: $U_i \leftarrow$ union of all source URLs for all $s_k \in C_i$
16: **yield return** $\langle a_i, \beta_i, U_i \rangle$

Figure 5: LASEWEB program execution algorithm. It takes a LASEWEB program $\mathcal{P}$ and tuple of user query arguments $\vec{v}$, and returns a set of triples $\langle a_i, \beta_i, U_i \rangle$ where a_i is an answer, β_i is its confidence score, U_i is a set of its source URLs.

separation, this naïve algorithm would detect four columns instead of three for "Fort Worth" and "Grand Prairie" rows.

However, there exists an approach to figure out correct splits automatically. Fix a candidate separator from the common list. For this separator and a fixed HTML node $\mathcal{N}$, we first build a list of lines in $\mathsf{Text}(\mathcal{N})$ that have consistent splitting with respect to this separator (i.e. the same number of columns > 1). The rest of the lines are passed to the PBE system FlashFill [9], which automatically figures out likely syntactic string transformations from few examples by using sophisticated ranking schemes. We use their publicly available implementation from the Microsoft Excel spreadsheet system. The list of consistent lines is used as a list of positive examples for FlashFill. The output of FlashFill is the list of correct outputs (splits) for the rest of the lines, or a failure signal. Our LASEWEB program interpreter automatically checks the correctness of this output later, by comparing it with other answer candidates during clustering.

4.1.3 Visual expressions

The key idea in interpretation of visual expressions is usage of runtime presentational attributes of HTML nodes $\mathcal{N}$. Specifically, we ask a browser to render each webpage in memory, on a virtual canvas of 1920×1080 pixels. After rendering, we collect the information about the bounding box of each HTML node $\mathcal{N}$, and use it as a value of $\mathsf{BBox}(\mathcal{N})$. Similarly, we collect the run-time values of CSS attributes, and use them for evaluating visual constraints Ψ.

4.2 Program execution

Fig. 5 shows the algorithm of LASEWEB program execution. It takes a LASEWEB *program* $\mathcal{P}$ and a tuple of user query arguments $\vec{v}$, and returns a set of *answer strings*, augmented with their *confidence scores* and *source URLs*.

The execution algorithm is shown as SEARCH($\mathcal{P}, \vec{v}$) function. It starts with querying the search engine for URLs with a seed query (line 1) and filling in user arguments in the LASEWEB query $\mathcal{Q}$ (line 2). After that it proceeds with matching the query $\mathcal{Q}$ with every URL in the obtained list of search results U.

During matching, the algorithm maintains a set of clusters $\mathcal{C}$. Every cluster $C_i \in \mathcal{C}$ represents a single logical answer. A cluster is a multi-set of string representations s_k, augmented with sets of their source URLs. A cluster can contain multiple occurences of the same answer string s_k, and the same answer string may occur multiple times on a single webpage (URL). Every time the algorithm finds a new answer string s_k on some webpage u_j, it constructs a new singleton cluster $\{\langle s_k, \{u_j\} \rangle\}$ and merges it with all existing clusters in $\mathcal{C}$ that contain strings similar to s_k, according to the similarity function σ (lines 7-11).

After matching, the algorithm extracts logical answers and their confidence scores from the collected set of clusters $\mathcal{C}$ (lines 12-16). For each cluster $C_i \in \mathcal{C}$, it considers the most frequent answer string in it as a *representative answer* of this cluster. Its confidence score is calculated with Bayes Law:

$$\beta_i = \frac{1}{|U|} \sum_{j=1}^{|U|} p(C_i \mid u_j) \qquad p(C_i \mid u_j) = \sum_{s \in C_i} \frac{c(s, u_j)}{|S_j|}$$

where $|S_j|$ is the number of answer strings extracted from u_j, and $c(s, u_j)$ is the number of times s was found in u_j.

5. EVALUATION

We implemented LASEWEB in C# programming language in approximately 5000 lines of code. In this section, we present the results of its evaluation over two applications of parametrized Web programming, shown in §2. It aims to verify the following usability requirements of LASEWEB:

1. Low number of iterations (refinements) during LASEWEB program development.
2. High total number of correctly answered real-life queries.

For the first goal, we observed that in all experiments, the number of iterations required to compose and evaluate every program on our own search strategies was at most 5. We present the detailed results of evaluating the second goal below for every application.

Micro-segments. We extracted 100,000+ user queries across 7 micro-segments from the logs of a popular search engine over the 12 month period. These micro-segments were chosen by our Bing partners as possible targets because of a substantial query traffic. Since LASEWEB does not have an NLP front-end, we used regular expressions to extract user queries that belonged to certain factoid micro-segments.

Fig. 6a reports the results of micro-segment evaluation. For each micro-segment we show the recall (fraction of answered user queries), and comparison with the existing Bing question answering module. Given the absence of ground truth data for real-life user queries, we evaluated LASEWEB precision by manually sampling 30-50 queries for each micro-segment at random. On average, the top 3 results contained "correct" answers to the query in 95% of cases.

As Fig. 6a shows, LASEWEB handles a much greater fraction of user queries than Bing question answering module. Bing module gives definite answers from structured databases, thus achieving 100% precision, but low recall. Additionally, LASEWEB found alternative answer options for many questions, providing additional context, whereas Bing provided only a single answer from the database. Low recall of "ASCII" and "Lyrics" micro-segments can be improved by additional refinement of our LASEWEB programs.

We refrained from comparison of LASEWEB with state-of-the-art Web question answering engines for two reasons. First, question answering engines mostly operate on a lin-

(a)

Micro-segment	# queries	Recall	Bing recall
ASCII code of a symbol	1,551	32.88%	0%
Calories in a food	9,207	71.80%	0%
Inventor of a product	8,994	75.91%	16.01%
Lyrics of a song	48,995	24.36%	0%
Phone number of a company	6,881	95.49%	0%
Population of a place	18,151	92.53%	57.58%
Release date of a product	12,339	97.24%	12.60%

(b)

Search task	Recall	Precision
Phone #	29/37	21/29
Affiliation	34/37	22/34
PhD institution	21/37	13/21
General chair	21/28	17/21
Invited talks	13/28	11/13
Average	**71%**	**73%**

Figure 6: (a) Evaluation results of factoid micro-segments. For each micro-segment: number of queries, fraction of queries answered by LaSEWeb (recall), fraction of queries, answered by Bing. LaSEWeb examined 50 top Bing results. (b) Evaluation results of repeatable search tasks. For each task: fraction of answered (recall), and correctly answered queries (precision). Precision was evaluated manually. LaSEWeb examined 30 top Bing results. In both applications, the running time is linearly proportional to the number of examined webpages, and a single webpage took from 0.1 to 20 sec to process.

guistic level, taking a query in natural language, and converting it into a logical representation. LaSEWeb solves an orthogonal problem: it takes a query in a logical representation, and searches for answers on the Web. Second, prior question answering approaches were evaluated on a *narrow* set of *unique* factoid questions (e.g. TREC corpus). In contrast, LaSEWeb focuses on *large micro-segments* of *similar* questions. Both of these aspects make direct comparison of approaches infeasible: LaSEWeb does not focus on user query analysis, and spending time on writing LaSEWeb programs for 500 different TREC questions was not meaningful. We note that any state-of-the-art user query analysis module can be used as a LaSEWeb front-end that converts a query into a logical representation, as discussed in §6.

The running time of LaSEWeb on a single webpage ranges from 0.1 to 20 seconds, and the time to interpret a single query is linearly proportional to the number of webpage examined. Since this performance is not sufficient for answering user queries in real-time, our engine can be used to generate answers to logged queries offline, and the results can be stored in a database to answer future queries in real-time.

Repeatable search tasks. We chose 5 categories of repeatable academic search tasks, and used them for LaSEWeb evaluation. The test data for 3 people-related tasks is a list of PLDI 2014 committee members (37 people), and the data for 2 conference-related tasks is a list of 21st century POPL and PLDI conferences (28 items). We evaluated precision and recall of all results manually.

Fig. 6b shows the results of the evaluation. Most of the desired information was easily extracted from the Web. The average precision is 73%, which is lower than that of a micro-segment search due to (a) inconsistensies in named entity recognition for organizations, and (b) name collisions for some of committee members. The recall in influenced by the recall of Bing results (for this application it is much lower that that of a factoid questions). In some of the cases we couldn't find the correct answers ourselves (e.g., for some invited talks). Also, LaSEWeb currently does not handle non-HTML formats: for example, a researcher's PhD institution is often listed in the CV, in PDF format, and Bing couldn't find any alternative sources of information.

6. RELATED WORK

Question answering. The problem of answering factoid questions in natural language has been studied for several decades. The most notable examples include Watson [12], KnowItAll [6], START [14], AskMSR [4], etc.

Most of these systems focus on understanding user intent, expressed as a question in natural language, and assume the existence of structured knowledge databases. In contrast, LaSEWeb extracts data from semi-structured sources on the Web, by restructuring this data according to the guidance, provided by an end-user in a form of a program in our language. These two efforts are orthogonal and can be integrated to provide a natural language front-end for extracting arguments for programs in our query language.

Some of the question answering systems [4, 6] use an automatic approach to extract answers from the Web. LaSEWeb is semi-automatic: it does not generalize the patterns automatically, mostly because it allows for specifying structural/visual patterns in addition to linguistic patterns, used by question answering systems. This allows for higher recall of LaSEWeb, but requires an end-user to construct a program manually, by exploring several examples. The integration of NLP technologies for user intent understanding will eliminate the need for manual program construction.

Repeatable search. A concept similar to our definition of repeatable search tasks are Web mashups, which are websites that combine information from public APIs into a single visualization. Ennals and Gay [5] proposed MashMaker, a GUI and a language for creating mashup websites. However, mashups use only structured information from public APIs as their data sources. In contrast, LaSEWeb allows to automate arbitrary repeatable search task over semi-structured Web data. Unlike MashMaker, LaSEWeb doesn't provide a GUI for constructing programs in our language, but we intend to explore automated synthesis of such programs from examples or natural language [10]. A similar work has been done on mashups: Vegemite [19] generates mashup expressions using programming by demonstration.

CoScripter [18] is a system that automates repetition of Web-based tasks using demonstration. It records scripts of actions on a webpages into a central public repository, from which they are later available for execution. CoScripter focuses on multiple Web-based tasks, not necessarily search-related. However, it is limited in a way in which scripts can process information on a webpage. LaSEWeb includes a wide range of webpage features, generalized from our observations of typical search strategies, which allow for a more precise capturing of the end-user's search intent.

SXPath. SXPath [20] is the language that extends popular XML query language XPath [2] towards queries upon visual layout of XML elements. It is built on the system of *visual axes* and *bouding boxes* in addition to XPath concepts.

LaSEWeb includes visual expressions as part of its query language, because they often appear in patterns that end-users use to extract information from the Web. In contrast to SXPath, LaSEWeb also includes linguistic and structural

expressions, which allow for querying a HTML document on the levels, semantically different from XML.

Table detection. An important portion of LaSEWeb is its logical table detection algorithm, that lies in the core of structural expression matching. Table detection has been studied in the past. Tengli et al. [25] develop an algorithm that extracts information from semi-structured HTML and plaintext tables using examples – marked row/column labels. Krüpl et al. [16] and, similarly, Gatterbauer et al. [8] propose techniques for using visual cues (bounding boxes) for automatic table extraction. Cafarella et al. [1] perform table extraction from linguistic cues.

LaSEWeb integrates *all* table extraction approaches (textual, visual and structural) in a *single automatic* function. Our implementation also uses a state-of-the-art PBE system [9] in a novel manner for parsing plain text tables with inconsistent separators. However, any off-the-shelf table extraction technique (e.g., [1, 16, 17, 25]) can be used to complement the logical table extraction module for LaSEWeb.

7. CONCLUSION

This paper investigated the problem of Web programming – a novel programming model, where functions are defined as repeatable search procedures, operating over semi-structured content of webpages returned by a search engine. These functions can be used multiple times with different user query arguments. We explored two of the many possible applications of Web programming, namely factoid micro-segments in search engines, and repeatable batch queries.

We present a DSL that allows end-users and software developers to express semantic patterns of their search strategies, eliminating the need for manual exploration of search engine results. Our implementation of the DSL leverages cross-disciplinary technologies: browser rendering APIs, state-of-the-art NLP tools, and programming by example technologies. Our system LaSEWeb achieves good precision and recall in practice, and is an important step towards automation of knowledge discovery on the Web.

LaSEWeb can be seen as bridging the gap between PL technologies for processing structured data formats, and numerous semi-structured data on the Web. Our programming model is a natural evolution of the prior effort on integrating structured Web queries into programming environments.

8. ACKNOWLEDGMENTS

We thank Saurabh Tiwary and the Bing team for providing us access to the valuable user search query data.

References

[1] M. J. Cafarella, C. Re, D. Suciu, O. Etzioni, and M. Banko. Structured querying of Web text. *CIDR'07*.

[2] J. Clark, S. DeRose, et al. XML path language (XPath), 1999.

[3] C. De Rosa, B. Gauder, D. Cellentani, T. Dalrymple, and L. J. Olszewski. *Perceptions of Libraries, 2010: Context and Community: a Report to the OCLC Membership*. OCLC, 2011.

[4] S. Dumais, M. Banko, E. Brill, J. Lin, and A. Ng. Web question answering: Is more always better? *SIGIR'02*.

[5] R. Ennals and D. Gay. User-friendly functional programming for web mashups. In *ICFP*, 2007.

[6] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll. In *WWW*, 2004.

[7] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by Gibbs sampling. In *ACL*, 2005.

[8] W. Gatterbauer and P. Bohunsky. Table extraction using spatial reasoning on the CSS2 visual box model. In *AAAI*, 2006.

[9] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *POPL'11*.

[10] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

[11] R. Housewright, R. C. Schonfeld, and K. Wulfson. Ithaka S+ R US Faculty Survey 2012. *April*, 8, 2013.

[12] A. Ittycheriah, M. Franz, W.-J. Zhu, A. Ratnaparkhi, and R. J. Mammone. IBM's statistical question answering system. In *TREC*, 2000.

[13] B. J. Jansen and A. Spink. How are we searching the World Wide Web? A comparison of nine search engine transaction logs. *IPM*, 42(1), 2006.

[14] B. Katz, G. Marton, G. C. Borchardt, A. Brownell, S. Felshin, D. Loreto, J. Louis-Rosenberg, B. Lu, F. Mora, S. Stiller, et al. External knowledge sources for question answering. In *TREC*, 2005.

[15] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *ACL*, 2003.

[16] B. Krüpl, M. Herzog, and W. Gatterbauer. Using visual cues for extraction of tabular data from arbitrary HTML documents. In *WWW*, 2005.

[17] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *PLDI*, 2014.

[18] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *CHI*, 2008.

[19] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *IUI*, 2009.

[20] E. Oro, M. Ruffolo, and S. Staab. SXPath: extending XPath towards spatial querying on web documents. *VLDB*, 4(2), 2010.

[21] K. Purcell, J. Brenner, and L. Rainie. *Search engine use 2012*. Pew Internet & American Life Project, 2012.

[22] C. Quirk, P. Choudhury, J. Gao, H. Suzuki, K. Toutanova, M. Gamon, W.-t. Yih, L. Vanderwende, and C. Cherry. MSR SPLAT, a language analysis toolkit. In *ACL*, 2012.

[23] A. Spink, H. C. Ozmutlu, E. Aversa, and C. Manley. What do people ask for on the web and how do they ask it: Ask Jeeves query analysis. 2001.

[24] J. Teevan, C. Alvarado, M. S. Ackerman, and D. R. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In *CHI*, 2004.

[25] A. Tengli, Y. Yang, and N. L. Ma. Learning table extraction from examples. In *ACL*, 2004.

[26] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *HLT-NAACL*, 2003.

[27] W.-t. Yih, G. Zweig, and J. C. Platt. Polarity inducing latent semantic analysis. In *ACL*, 2012.