# Wrapper Induction for Semistructured, Web-based Information Sources *

**Ion Muslea, Steve Minton, Craig Knoblock**
Information Sciences Institute, the Integrated Media Systems Center, and Computer Science Department
University of Southern California
{muslea, minton, knoblock}@isi.edu

## Abstract

Central to any information mediator that accesses Web-based information sources is a set of *wrappers* that extract relevant data from Web pages. We introduce STALKER, a wrapper-induction algorithm that generates extraction rules for semistructured, Web-based information sources. STALKER generates extraction rules that are expressed as simple landmark grammars, which are a class of finite automata that is more expressive than the existing extraction languages. Based on just a few training examples, STALKER learns extraction rules for sources that can not be wrapped by existing techniques.

## 1 Introduction

With the expansion of the Web, computer users gained access to a large variety of comprehensive information repositories. However, the Web is based on a browsing paradigm that makes it difficult to retrieve and integrate data from multiple sources. The most recent generation of *information mediators* (e.g., Ariadne (Knoblock *et al.* 1998), TSIMMIS (Chawathe *et al.* 1994), or Internet Softbots (Doorenbos, Etzioni, & Weld 1997)) address this problem by enabling information from pre-specified sets of Web sites to be accessed via database-like queries. For instance, consider the query "What seafood restaurants in L.A. have prices below $20, and accept the Visa credit-card?". Assume that we have a mediator that can access two sources that provide information about LA restaurants: the Zagat Guide and LA Weekly (see Figure 1). To answer this query, the mediator could use Zagat's to identify seafood restaurants under $20 and then use LA Weekly to check which of these accept Visa.

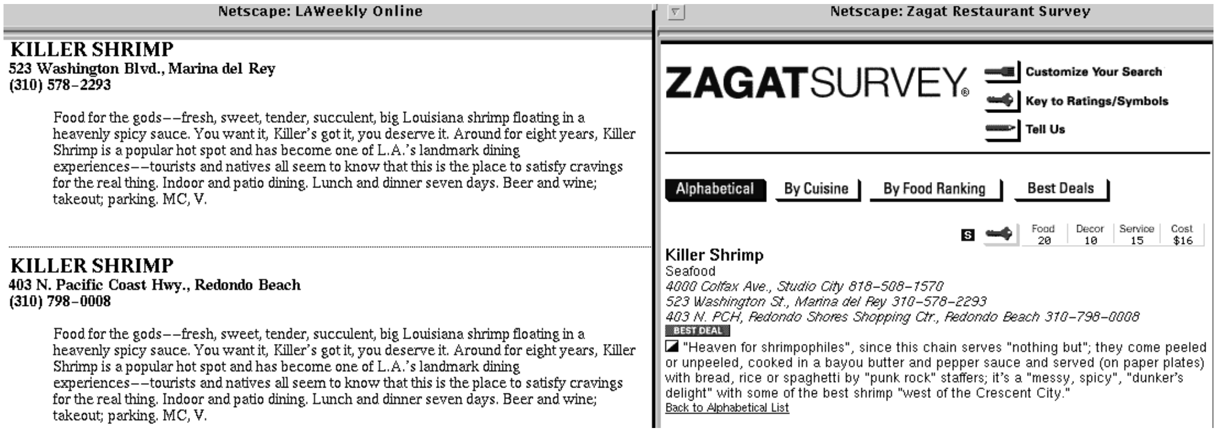
The mediators above rely on *wrappers* that can extract information from *semistructured* Web pages (a page is semistructured if the desired information can be located using a concise, formal grammar). Some mediators, such as TSIMMIS (Chawathe *et al.* 1994) and ARANEUS (Atzeni, Mecca, & Merialdo 1997) depend on humans to create the grammar rules required to extract information from a page. However, there are several reasons why this is undesirable. Writing extraction rules is tedious, time consuming and requires a high level of expertise. These difficulties are multiplied when an application domain involves a large number of existing sources or the sources change over time.

In this paper, we introduce a new machine learning method for wrapper construction, and, based on our approach, unsophisticated users can painlessly turn Web pages into relational information sources. The next section presents a formalism describing semistructured Web documents, and then Sections 3 and 4 present a domain-independent information extractor that we use as a skeleton for all our wrappers. Section 5 describes STALKER, a supervised learning algorithm for inducing extraction rules, and Section 6 presents a detailed example. The final sections describe our experimental results, related work and conclusions.

## 2 Describing the Content of a Page

As Web pages are intended to be human readable, there are some common conventions that are used to structure HTML pages. For instance, the information on a page often exhibits some hierarchical structure; furthermore, semistructured information is often presented in the form of lists or tuples, with explicit separators used to distinguish the different elements. Consequently, we have developed the *embedded catalog* ($\mathcal{EC}$) formalism, which can be used to describes the structure of most semistructured documents.

*This work is supported in part by the University of Southern California Integrated Media Systems Center (IMSC) - a National Science Foundation Engineering Research Center, by the United States Air Force under contract number F49620-98-1-0046, by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-97-2-0352, by the Defense Logistics Agency, DARPA, and Fort Huachuca under contract number DABT63-96-C-0066, and by a research grant from General Dynamics Information Systems. The views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of any of the above organizations or any person connected with them.



**KILLER SHRIMP**
**523 Washington Blvd., Marina del Rey**
**(310) 578–2293**

Food for the gods––fresh, sweet, tender, succulent, big Louisiana shrimp floating in a heavenly spicy sauce. You want it, Killer's got it, you deserve it. Around for eight years, Killer Shrimp is a popular hot spot and has become one of L.A.'s landmark dining experiences––tourists and natives all seem to know that this is the place to satisfy cravings for the real thing. Indoor and patio dining. Lunch and dinner seven days. Beer and wine; takeout; parking. MC, V.

**KILLER SHRIMP**
**403 N. Pacific Coast Hwy., Redondo Beach**
**(310) 798–0008**

Food for the gods––fresh, sweet, tender, succulent, big Louisiana shrimp floating in a heavenly spicy sauce. You want it, Killer's got it, you deserve it. Around for eight years, Killer Shrimp is a popular hot spot and has become one of L.A.'s landmark dining experiences––tourists and natives all seem to know that this is the place to satisfy cravings for the real thing. Indoor and patio dining. Lunch and dinner seven days. Beer and wine; takeout; parking. MC, V.

| S | | Food | Decor | Service | Cost |
|---|---|---|---|---|---|
| | | 20 | 10 | 15 | $16 |

**Killer Shrimp**
Seafood
*4000 Colfax Ave., Studio City 818–508–1570*
*523 Washington St., Marina del Rey 310–578–2293*
*403 N. PCH, Redondo Shores Shopping Ctr., Redondo Beach 310–798–0008*

"Heaven for shrimpophiles", since this chain serves "nothing but"; they come peeled or unpeeled, cooked in a bayou butter and pepper sauce and served (on paper plates) with bread, rice or spaghetti by "punk rock" staffers; it's a "messy, spicy", "dunker's delight" with some of the best shrimp "west of the Crescent City."

Figure 1: **Restaurant Descriptions provided by Zagat Guide and LA-Weekly**

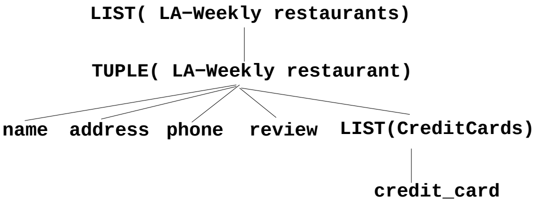


Figure 2: $\mathcal{EC}$ **description of LA-Weekly pages.**

The $\mathcal{EC}$ description of a page is a tree-like structure in which the leaves represent the data to be extracted. Each internal node of the $\mathcal{EC}$ tree denotes either a homogeneous list (such as a list of names) or a heterogeneous tuple (e.g., a 2-tuple consisting of a name and an address). We use the terms *items* and *primitive items* to designate the nodes and the leaves of the tree, respectively. For instance, Figure 2 displays the $\mathcal{EC}$ description of the LA-Weekly pages. At the top level, each page is a list of restaurant descriptions. Each restaurant description is a 5-tuple that contains four primitive items (i.e., name, address, phone, and review) and an embedded list of credit cards.

## 3 Extracting Data from a Document

In our framework, a document is a sequence of tokens (e.g., words, numbers HTML tags, etc.). Given an $\mathcal{EC}$ description of a document, a query specifying a set of items to be extracted from that document, and a set of extraction rules for each item in $\mathcal{EC}$, a wrapper must generate the answer to the input query. A key idea underlying our work is that the extraction rules can be based on "landmarks" that enable a wrapper to locate a item $x$ within the content of its parent item $p$.

For instance, let us consider the two restaurant descriptions presented in Figure 3. In order to identify the beginning of the restaurant name within a restaurant description, we can use the rule

**R1** = *SkipTo*(<BR> <BR>) *SkipTo*(<B>)

The rule **R1** has the following meaning: start from the beginning of the restaurant description and skip everything until you find two consecutive <BR> tags. Then, again, skip everything until you find a <B> tag. An alternative way to find the start of the restaurant name is the rule *SkipTo*(<BR> _HtmlTag_) *SkipTo*(<B>), which uses the wildcard _HtmlTag_ as a placeholder for any HTML tag. Similarly, the rules *SkipTo*(<BR> <B>) and *SkipTo*(_HtmlTag_ <B>) identify the address, while the rule *SkipTo*(<BLOCKQUOTE>) finds the beginning of the review. We say that the rules above *match* the examples in Figure 3 because all the arguments of the *SkipTo()* functions are found within the restaurant description. By contrast, the rule *SkipTo*(<HTML>) is said to *reject* both examples because the restaurant descriptions end before the rule matches.

The phone-number item is more difficult to extract because the information source uses two different formats: based on the phone number availability, the phone information is preceded either by <p>, or by <i>. In this case, the beginning of the phone-number item can be identified by the *disjunctive rule* EITHER *SkipTo*(<p>) OR *SkipTo*(<i>) (a disjunctive rule matches if any of its disjunct matches[1]).

The examples above describe rules that are used to locate an item $x$ within its parent $p$. If the item $x$ is a list, then a second step is necessary to break out the individual members of the list. For example, to extract the credit card information, we first apply an extraction rule to get the list of credit cards MC, V, AE, DIS; then we repeatedly use the *iteration rule* *SkipTo*(,) to reach the start of each credit card. More formally, an iteration rule can be repeatedly applied to a list $L$ in order to identify each element of $L$.

Our approach to information extraction is based on

[1] If more than one disjunct matches, and the two disjuncts disagree on the starting position, then we choose nondeterministically among the matching disjuncts.

```
<TITLE>LA Weekly Restaurants</TITLE><CENTER><BR>
Name: <B> killer shrimp </B> Location: <B><FONT>
Any </FONT> </B> <BR><BR> Cuisine: any <B>
KILLER SHRIMP </FONT></B><BR><B> 523 Washington
, Marina del Rey <BR><p> (310) 578-2293 </B>
<BLOCKQUOTE> Food for the gods--fresh, sweet ...
Lunch and dinner seven days. Beer and wine;
takeout; no reservations. MC, V, AE, DIS.
```

```
<TITLE>LA Weekly Restaurants</TITLE><CENTER><BR>
Name: <B> Hop Li</B> Location: <B><FONT> LA </FONT>
</B> <BR><BR> Cuisine: chinese <B>
HOP LI </FONT></B><BR><B> 10974 Pico Blvd., West
L.A. <BR></B><i> No phone number.</i><BLOCKQUOTE>
This branch of a venerable Chinatown seafood house
brings Hong Kong-style seafood to the Westside ...
No alcohol; takeout; reservations accepted. V, AE.
```

Figure 3: **Fragments of two LA-Weekly answer-pages.**

a very simple idea: in order to extract a set of items $\mathcal{X}$ from a page, the wrapper starts from the root and goes iteratively deeper into the $\mathcal{EC}$ tree. For each level $l$ in the tree, the wrapper extracts each item that is an *ancestor* of an item from $\mathcal{X}$. Our approach has two important advantages. First, the *hierarchical* extraction (i.e., each item $x$ is extracted from its parent $p$) is beneficial whenever the structure of the page has some variations and the extraction of $p$ from $Parent(p)$ requires a disjunctive rule. Under such a scenario, the process of extracting an item $x$ from $p$ is not affected by the complexity of the process of extracting $p$ from $Parent(p)$. Second, our approach does not rely on there being a fixed ordering of items within a tuple. We decided against relying on item ordering because, in our experience, about 40% of the Web sources we examined either had missing items or allowed items to occur in different orders.

## 4 Extraction Rules as Finite Automata

We now introduce two key concepts that can be used to define extraction rules: *landmarks* and *landmark automata*. In the rules described in the previous section, each argument of a *SkipTo*() function is a *landmark*, while a group of *SkipTo*()s that must be applied in a pre-established order represents a landmark automaton. In other words, any extraction rule from the previous section is a landmark automaton.

In this paper, we focus on a particular type of landmark: the *linear landmark*. A linear landmark is described by a sequence of tokens and wildcards (a wildcard represents a class of tokens, as illustrated in the previous section, where we used the wildcard *_HtmlTag_*). Linear landmarks are interesting for two reasons: on one hand, they are sufficiently expressive to allow efficient navigation within the $\mathcal{EC}$ structure of the documents, and, on the other hand, as we will see towards the end of this section, there is a simple way to generate and refine landmarks during learning.

*Landmark automata* ($\mathcal{LA}$s) are nondeterministic finite automata in which each transition $S_i \rightsquigarrow S_j$ ($i \neq j$) is labeled by a landmark $l_{i,j}$; that is, the transition $S_i \rightsquigarrow S_j$ takes place if and only if in the state $S_i$ the input is a string $s$ that is accepted by the landmark $l_{i,j}$. The *Simple Landmark Grammars* ($\mathcal{SLG}$s) are the class of $\mathcal{LA}$s that correspond to the *disjunctive rules* introduced in the previous section. Any $\mathcal{SLG}$ has the following properties:

- the initial state $S_0$ has a branching-factor of $k$;
- it has exactly $k$ accepting states (one per branch);
- all $k$ branches that leave the $S_0$ are *sequential* $\mathcal{LA}$s (i.e., from each non-accepting state $S_i$, there are exactly two possible transitions: a loop to itself, and a transition to the next state.
- *linear landmarks* label each non-looping transition.

Now that we introduced the $\mathcal{SLG}$ formalism, let us consider the four simplified restaurant descriptions presented in Figure 4. It is easy to see that they have a similar structure: available drinks, optionally followed by parking information, followed by reservation and credit card information. Despite the similarity of the descriptions, writing an $\mathcal{SLG}$ that extracts the credit card data is a non-trivial task for several reasons. First, the formats for the restaurants that do and do not accept credit cards is slightly different. Second, there is no unique landmark that immediately precedes the credit card item in all four descriptions. Consequently, even though the semicolon is the primary item separator, there is no simple, semicolon-based extraction rule for the credit cards. However, we can extract the credit card data by using the disjunctive rule EITHER *SkipTo*(`<i>`) OR *SkipTo*(`reservations`)*SkipTo*(;)

D1: wine; parking; reservations suggested; *Visa*
D2: full bar; no reservations; *major* cards
D3: full bar; reservations accepted <i> *no* credit
D4: beer; parking; reservations suggested <i>*cash*

Figure 4: **Examples of restaurant descriptions.**

## 5 Learning Extraction Rules

In this section, we present an induction algorithm that generates an $\mathcal{SLG}$ rule identifying the start (or end) of an item $x$ within its parent $p$. The input to the algorithm is a set of training examples, where each training example consists of a sequence of tokens $p$ and an index indicating where $x$ starts (or ends) in $p$.

The induction algorithm, called STALKER, is a covering algorithm. It begins by generating a simple rule

```
STALKER( $TrainingExs$ )
- WHILE $TrainingExs \neq \emptyset$
  - $Terminals$ =GetTokens($TrainingExs$) $\cup$ $Wildcards$
  - $Disj$ =LearnDisjunct($TrainingExs, Terminals$)
  - remove $TrainingExs$ covered by $Disj$
- return $\mathcal{SLG}$( all learned disjuncts )

LearnDisjunct( $TrainingExs$, $Terminals$ )
- $Candidates$ = GetInitialCandidates( $TrainingExs$ )
- WHILE $Candidates \neq \emptyset$ DO
  - $D$ = BestDisjunct($Candidates$)
  - IF $D$ is a perfect disjunct THEN return $D$
  - replace $D$ by $\bigcup_{t \in Terminals}$RefineDisjunct($D, t$)
- return best disjunct
```

Figure 5: **The STALKER algorithm.**

that covers as many of the positive examples as possible. Then it tries to create a new rule for the remaining examples, and so on. For instance, let us consider the training examples $D1$, $D2$, $D3$, and $D4$ from Figure 4 (in each training example, the italicized token denotes the start of the credit card information). As there is no unique token that immediately precedes the credit card information in all four examples, the algorithm generates first the rule **R1**::= *SkipTo*(`<i>`), which has two important properties:

- it accepts the positive examples in $D3$ and $D4$;
- it rejects both $D1$ and $D2$ (i.e., **R1** does not generate a false positive for $D1$ and $D2$).

During a second iteration, the algorithm considers only the uncovered examples $D1$ and $D2$, and it generates the rule **R2**::= *SkipTo*(`reservations`) *SkipTo*(;). In Figure 6, we show the *disjunctive rule* EITHER **R1** OR **R2**, together with a more general rule that replaces the landmark `<i>` by the wildcard *_HtmlTag_*.

Figure 5 presents the STALKER learning algorithm. The figure actually describes only how STALKER identifies the beginning of $x$ within $p$. The complementary task of finding the end of $x$ can be easily solved based on the symmetry of the substring identification problem. That is, given a substring $X$ of a string $P$, finding a rule that identifies the start of $X$ by looking leftward from the beginning of $P$ is similar to finding a rule that identifies the end of $X$ by looking rightward from the end of $P$. If we think of the strings $X$ and $P$ as the contents of the items of $x$ and $p$, it follows that by simply "reversing" the strings $X$ and $P$, we can use STALKER to find a rule that identifies the end of $x$ within $p$. For instance, the end of the address in Figure 3 can be found by applying the rule *SkipTo*(`<BR>`) to the "reversed" restaurant descriptions (i.e., the last token becomes the first one, and vice-versa).

In order to create an extraction rule, STALKER has to induce both the topology of the $\mathcal{SLG}$ (i.e., the number of disjuncts, and the number of landmarks in each disjunct) and the landmarks that label the $\mathcal{SLG}$'s transitions. STALKER's input consists of pairs ($Tokens_i$, $Start_i$), where $Tokens_i$ is the sequence of strings obtained by *tokenizing* $P_i$ (i.e., the $i$-th instance of $p$), and $Tokens_i[Start_i]$ is the token that represents the start of $x$ within $P_i$. A sequence $S ::= Tokens_i[1]$, $Tokens_i[2]$, ..., $Tokens_i[Start_i - 1]$ (i.e., *all* tokens that precede the start of $x$ in $P_i$) represents a positive example, while any other sub-sequence or super-sequence of $S$ represents a negative example. STALKER tries to generate an $\mathcal{SLG}$ that accepts all positive examples and rejects all negative ones.

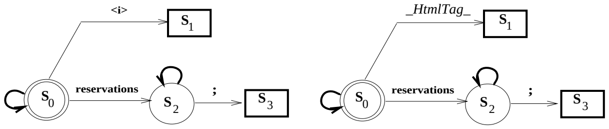


Figure 6: **Two 2-disjunctive rules.**

STALKER is a typical *sequential covering* algorithm: as long as there are some uncovered positive examples, it tries to learn a *perfect disjunct* (i.e., a sequential $\mathcal{LA}$ that accepts only true positives). When all the positive examples are covered, STALKER returns the solution, which consists of an $\mathcal{SLG}$ in which each branch corresponds to a *perfect disjunct*.

The function *LearnDisjunct*() is a *greedy* algorithm for learning *perfect disjuncts*: it generates an initial set of *candidates* and repeatedly selects and refines the *best candidate* until either it finds a *perfect disjunct*, or it runs out of candidates. To find the best disjunct in *Candidates*, STALKER looks for a disjunct $D$ that accepts the largest number of positive examples. In case of a tie, the best disjunct is the one that accepts fewer false positives.

Each initial candidate is a 2-state landmark automaton in which the transition $S_0 \rightsquigarrow S_1$ is labeled by a terminal in the set of initial landmarks $IL = \{t | \exists i, Tokens_i[Start_i - 1] = t\} \bigcup \{w \in Wildcards | \exists t \in IL, Matches(w, t)\}$. That is, a member of $IL$ is either a token $t$ that immediately precedes the beginning of $x$, or a wildcard that "matches" such a $t$. The rationale behind the choice of $IL$ is straightforward: as disjuncts that are not ended by $IL$ terminals cannot accept positive examples, it follows that STALKER can safely ignore them.

In order to perform the landmark induction, STALKER uses two types of terminals: tokens and wildcards. The former are the tokens that occur *at least once* in *each* uncovered positive example, while the latter are one of the following placeholders: *Numeric, AlphaNumeric, Alphabetic, Capitalized, AllCaps, HtmlTag,* and *Symbol* (i.e., any 1-character token that is not a letter or a number). The terminals must be recomputed after each invocation of *LearnDisjunct*() because the cardinality of the uncovered cases decreases, and, consequently, the number of tokens that occur in each uncovered positive example may increase.

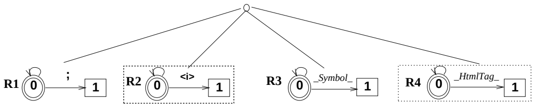


Figure 7: **Initial candidates (first iteration).**

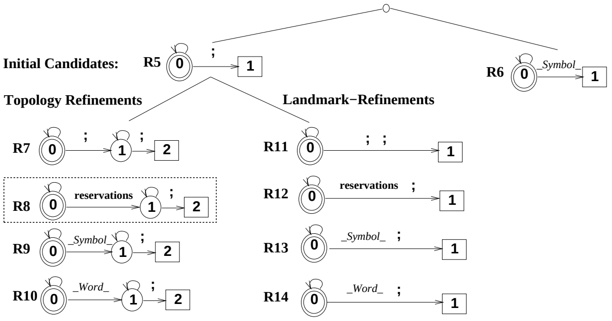


Figure 8: **Rule induction (second iteration).**

Intuitively, *RefineDisjunct*() tries to obtain (potentially) better disjuncts either by making its landmarks more specific (*landmark refinements*), or by adding new states in the automaton (*topology refinements*). Given a disjunct $D$, a landmark $l$ from $D$, and a terminal $t$, a *landmark refinement* makes $l$ more specific by concatenating $t$ either at the beginning or at the end of $l$. By contrast, a *topology refinement* leaves the existing landmarks unchanged and modifies only the number of states. That is, if $D$ has a transition $A \rightsquigarrow B$ labeled with the landmark $l$, a *topology refinement* adds a state $S$ by replacing $A \rightsquigarrow B$ with $A \rightsquigarrow S$ (labeled by $t$), and $S \rightsquigarrow B$ (labeled by $l$).

## 6 Example of Rule Induction

Let us consider again the restaurant descriptions from Figure 4. In order to generate extraction rules for the start of the credit card information, we invoke STALKER with the training examples {$D1$, $D2$, $D3$, $D4$}. During the first iteration, *GetTokens*() returns {`; reservations`} because only the semicolon and the word `reservations` occur at least once in each positive example. The function *LearnDisjunct*() is invoked with two parameters: the training examples and the terminals {`; reservations` *_Symbol_ _Word_*} (i.e., the two tokens above together with all the wildcards that "match" them). *LearnDisjunct*() computes the set $IL$ = {`; <i>` *_Symbol_ _HtmlTag_*} , which consists of the tokens that *immediately precede* the start of the credit card item and all the wildcards that "match" them (the corresponding initial candidates are showed in Figure 7). The rules R2 and R4 represent *perfect disjuncts*: they recognize the start of the credit card item in $D3$ and $D4$ and reject $D1$ and $D2$. Consequently, *LearnDisjunct*() returns the disjuncts R2 and R4.

During the second iteration, *LearnDisjunct*() is invoked with the following parameters: the uncovered training examples {$D1$, $D2$} and the terminals {`; reservations` *_Symbol_ _Word_*} . *LearnDisjunct*() finds the set $IL$ = {*; _Symbol_*} and generates the initial candidate-rules R5 and R6 (see Figure 8). As both candidates accept the same false positives ("wine ;" and "full bar ;", respectively), *LearnDisjunct*() randomly selects the rule to be refined first - say R5. By refining R5, STALKER creates the topological refinements R7÷R10, together with the landmark refinements R11÷R14. As R8 is a perfect disjunct (i.e., it covers both $D1$ and $D2$), there is no need for additional iterations. Finally, STALKER completes its execution by returning two equivalent disjunctive rules: EITHER R2 OR R8, and EITHER R4 OR R8 (see Figure 6).

| Source | $PI$ | Miss | Unord. | WIEN | STALKER |
|---|---|---|---|---|---|
| Okra | 4 | - | - | √ | √ |
| BigBook | 6 | - | - | √ | √ |
| Addr Finder | 6 | √ | √ | - | √ |
| Quote Serv. | 18 | - | √ | - | √ |
| LA Weekly | 5 | √ | - | - | √ |

Table 1: **Illustrative information sources.**

| Source | $TE$ | Refs | CPU time |
|---|---|---|---|
| Okra | 2 | 4892 | 30 |
| BigBook | 4 | 4026 | 2:25 |
| Address Finder | 5 | 10946 | 9:02 |
| Quote Server | 4 | 334349 | 39:10 |
| LA Weekly | 4 | 4946 | 1:08 |

Table 2: **Experimetal data.**

## 7 Experimental Results

In Table 1, we present five illustrative sources that were successfully wrapped based on the extraction rules learned by STALKER. The first four sources were taken from Kushmerick's thesis, for purposes of comparison. These four sources were relatively hard for the WIEN system (Kushmerick 1997). Specifically, the Address Finder and the Quote Server were beyond WIEN's wrapping capabilities, while Okra and BigBook required the largest number of training cases. We also show our results for LA Weekly, which can not be wrapped by WIEN because it has both missing items and multiple levels of embedding. For each source, Table 1 provides the following information: the number $PI$ of primitive items per page, and whether it allows missing or unordered items.

In Table 2 we present the experimental data collected while using STALKER to learn the extraction rules for the sources above. For each source, we provide the following information: the number $TE$ of training examples, the total number of refinements performed during the learning process, and the CPU time required to learn the rules. Based on the data in Table 2, we can make several observations. First, STALKER needs only a few training examples to wrap Okra and

BigBook, as opposed to WIEN, which requires almost two orders of magnitude more examples. Second, even for hard sources like the Address Finder and the Quote Server, STALKER requires extremely few training cases (5 and 4, respectively).

## 8 Related Work

With the recently increasing interest in accessing Web-based information sources, a significant number of research projects deal with the information extraction problem: Ariadne (Knoblock *et al.* 1998), WIEN (Kushmerick 1997), TSIMMIS (Chawathe *et al.* 1994), ARANEUS (Atzeni & Mecca 1997) and Knowledge Broker (Chidlovskii, Borghoff, & Chevalier 1997). All these research efforts rely on one of the following wrapper-generation techniques: manual, expert system based, or inductive. There is a wide variety of manually-generated wrappers (i.e., the extraction rules are written by a human expert), from procedural languages (Atzeni & Mecca 1997) to pattern matching (Chawathe *et al.* 1994) to LL(k) grammars (Chidlovskii, Borghoff, & Chevalier 1997). Even though such systems offer fairly expressive extraction languages, the manual wrapper generation is a tedious, time consuming task that requires a high level of expertise; furthermore, the rules have to be rewritten whenever the sources suffer format changes.

The wrapper induction techniques used by WIEN (Kushmerick 1997) are better fit to frequent format changes because they rely on learning techniques to generate the extraction rules. While Kushmerick's approach dramatically reduces the time required to wrap a source, his extraction language is significantly less expressive than the ones provided by the manual approaches. Finally, in (Ashish & Knoblock 1997), the authors made a compromise between the manual and the inductive approaches by introducing an expert system that uses a fixed set of heuristics of the type "look for bold or italicized strings".

The STALKER-based approach presented in this paper combines the expressive power of the manual approaches with the wrapper induction idea introduced by Kushmerick. Our $\mathcal{SLG}$ language is more expressive than the one used by TSIMMIS, which is the equivalent of the 1-disjunctive $\mathcal{LA}$ that does not use wildcards. Similarly, the WIEN extraction language is a 1-disjunctive $\mathcal{LA}$ that has exactly two states and does not allow the use of wildcards. There are several other important differences between STALKER and WIEN. First, as WIEN learns the landmarks for its 2-state $\mathcal{LA}$s by searching *common prefixes* at the *character level*, it needs more training examples than STALKER. Second, given a tuple $p$ with $n$ items $a_1, a_2, \ldots, a_n$, WIEN narrows the search space by learning rules that extracts $a_{i+1}$ *starting from* the end of $a_i$. It follows that WIEN can not wrap sources in which the order of the items $a_i$ is not fixed, or the pages may have missing items (e.g., the Internet Address Finder and the Quote Server). Last but not least, STALKER can handle $\mathcal{EC}$ trees of arbitrary depths, while WIEN can generate wrappers only for sources of depth three (i.e., a single list of tuples),

## 9 Conclusions and Future Work

The primary contribution of our work is to turn a potentially hard problem – learning extraction rules – into a problem that is extremely easy in practice (i.e., typically very few examples are required). The number of required examples is small because the $\mathcal{EC}$ description of a page simplifies the problem tremendously: as the Web pages are intended to be human readable, the $\mathcal{EC}$ structure is generally reflected by actual landmarks on the page. STALKER merely has to find the landmarks, which are generally in the close proximity of the items to be extracted. In other words, given our $\mathcal{SLG}$ formalism, the extraction rules are typically very small, and, consequently, they are easy to induce.

We plan to continue our work on several directions. First, we plan to use *unsupervised learning* in order to narrow the landmark search-space. Second, we would like to investigate alternative $\mathcal{LA}$ topologies that would improve the expressiveness of the extraction language. Third, we plan to provide PAC-like guarantees for our wrapper induction algorithm.

## References

Ashish, N., and Knoblock, C. 1997. Semi-automatic wrapper generation for internet information sources. *Proceedings of Cooperative Information Systems.*

Atzeni, P., and Mecca, G. 1997. Cut and paste. *Proceedings of 16th ACM SIGMOD Symposion on Principles of Database Systems.*

Atzeni, P.; Mecca, G.; and Merialdo, P. 1997. Semi-structured and structured data in the web: going back and forth. *Proceedings of ACM SIGMOD Workshop on Management of Semi-structured Data* 1–9.

Chawathe, S.; Garcia-Molina, H.; Hammer, J.; Ireland, K.; Papakonstantinou, Y.; Ullman, J.; and Widom., J. 1994. The tsimmis project: integration of heterogeneous information sources. *10th Meeting of the Information Processing Society of Japan* 7–18.

Chidlovskii, B.; Borghoff, U.; and Chevalier, P. 1997. Towards sophisticated wrapping of web-based information repositories. *Proceedings of 5th International RIAO Conf.* 123–35.

Doorenbos, R.; Etzioni, O.; and Weld, D. 1997. A scalable comparison-shopping agent for the world wide web. *Proceedings of Autonomous Agents* 39–48.

Knoblock, C.; Minton, S.; Ambite, J.; Ashish, N.; Margulis, J.; Modi, J.; Muslea, I.; Philpot, A.; and Tejada, S. 1998. Modeling web sources for information integration. *Accepted at AAAI 1998.*

Kushmerick, N. 1997. Wrapper induction for information extraction. *PhD Thesis, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04.*