9 Wrappers

Wrappers are the components of a data integration system that communicate with the data sources. The wrapper's task involves sending queries from the higher levels of the data integration system to the sources and then converting the replies to a format that can be manipulated by the query processor. The complexity of the wrapper depends on the nature of the data source. In the simplest case, if the data source is a relational database system, then the task of the wrapper is rather simple and can involve merely interacting with a JDBC driver (which, admittedly, can often be harder than you would expect!). In more complex cases, the wrapper needs to parse semistructured data such as those found on HTML pages and transform them into a set of tuples. This chapter will focus on the latter case — efficiently building wrappers that convert semistructured data into tuples. We introduce the problems that are faced by wrappers and then discuss the different solutions that have been proposed in the literature.

9.1 Introduction

We illustrate the ideas underlying wrappers by considering data sources that consist of a set of Web pages. For each source S, we assume that each Web page displays structured data using a *schema* T_S and a *format* F_S, which are common across all pages of the source. It is important to keep in mind that the schema is not explicitly declared.

Example 9.1

Figures 9.1(a)–(c) describe three such data sources. Source *countries.com* describes basic information about countries, and Figure 9.1(a) shows a page that describes Germany. This page uses a relational schema that consists of a single table with the attributes country, capital, population, and continent. It displays country first, fully capitalized, then capital, population, and continent, prefixed with "Capital:", "Population:", and "Continent:", respectively.

Source *easycalls.com* is about calling codes, and Figure 9.1(b) shows a page that displays a list of tuples (country, code). Each tuple is displayed on a single line, with country in bold font, followed by code in italic. Finally, Figure 9.1(c) shows a page from *greatbooks.com*, which displays a book title in bold font, one or more authors, which are underlined, then price and publisher, which are prefixed with "Price:" and "Publisher:", respectively.

These kinds of pages are very common on the World Wide Web. They are created in sites that are powered by database systems. When a query is posed by a user, it is sent to

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00009-0

(a) *countries.com* (b) *easycalls.com* (c) *greatbooks.com*

FIGURE 9.1 Examples of data sources, each of which displays data using a schema and a format that are common across all pages of the source.

the back-end database, which responds with a set of tuples. At that point, a Perl script or another scripting program will create HTML that looks presentable to the user.

9.1.1 The Wrapper Construction Problem

Given a source S as described above, a *wrapper* W extracts structured data from the pages of S. Formally, W is a tuple (T_W, E_W), where T_W is a *target schema*, and E_W is an *extraction program* that uses the format F_S to extract from each page a data instance conforming to T_W. The target schema E_W need not be the same as the schema used on the page, since we may want to rename the attributes or only include a subset of them in our output. The following two examples illustrate the operation of wrappers.

Example 9.2

Consider a wrapper that extracts all attributes from the pages of *countries.com* (Figure 9.1(a)). The target schema T_W is the source schema T_S = (country, capital, population, continent). The extraction program E_W may specify that, when given a page P from the source, return the first fully capitalized string as country, then the string immediately following "Capital:" as capital, and so on.

Example 9.3

Consider a wrapper that extracts only title and price from source *greatbooks.com* (Figure 9.1(c)). Here the target schema T_W is the relational schema (title, price), and the extraction program E_W returns the string in bold font as title, then the number following "$" as price.

The *wrapper construction* problem is to quickly create the pair (T_W, E_W) by inspecting the pages of the source S. Since much of the effort in this realm has focused on developing

machine learning algorithms for constructing wrappers, the problem is often referred to as *wrapper learning*. Two main variants of this problem exist. In the first variant we want to learn the source schema T_S as well as a program E_W that extracts data conforming to T_S. Thus the wrapper to be constructed is $W = (T_S, E_W)$. In this case the source schema T_S is also the target schema T_W. For example, if we do not know the schema of *countries.com*, we may want to construct the wrapper in Example 9.2, in particular we want to construct both the target schema and the extraction program.

In the second variant, we want to extract only a subset of the attributes of the source schema T_S, and we already know this subset (e.g., by manually examining a set of pages of S). Then we define this subset to be the target schema T_W, and our goal is to construct only the extraction program E_W. For example, if we want to extract only title and price from *greatbooks.com*, then we may want to build the wrapper in Example 9.3, where we already know the target schema.

9.1.2 Challenges of Wrapper Construction

The above two variants of wrapper construction have been studied extensively and have proven quite challenging, for the following reasons.

LEARNING THE SOURCE SCHEMA

First, learning the source schema T_S turns out to be quite difficult. A common way to do this is to imagine that each page of S is a string generated by a grammar G. We then learn G from a set of pages and use G to infer T_S. For example, after examining a set of pages of *countries.com*, we may learn that they are generated by the following regular expression (which encodes a regular grammar):

$$R = \text{<html>(.+?)<hr>
(.+?)
Capital: (.+?)
Population:(.+?)}\\ \text{
Continent: (.+?)< /body>< /html>}$$

From this regular expression we can infer the source schema (country, capital, population, continent).

Unfortunately, inferring a grammar from positive examples (i.e., the pages of S in this case) is well known to be difficult. For example, we know that regular grammars cannot be correctly identified from positive examples alone. Even with both positive and negative examples, there is no efficient algorithm to identify a reasonable grammar (e.g., to identify the minimum-state deterministic finite state automaton that is consistent with an arbitrary set of examples).

Given these limitations, current solutions consider only relatively simple regular grammars that encode either flat tuple or nested tuple schemas. But even learning simple schemas has proven quite difficult. The general approach is to use various heuristics for searching a large space of candidate schemas. However, the correctness of the discovered schemas heavily depends on the heuristics employed, as incorrect heuristics often lead to incorrect schemas. Furthermore, increasing the complexity of the schema even slightly can lead to an exponential increase in the size of the search space, resulting in an intractable learning process.

LEARNING THE EXTRACTION PROGRAM

Learning the extraction program E_W has also proven quite difficult. Ideally, E_W should be Turing-complete (e.g., as a Perl script) to have the maximal expressive power. But learning such programs is clearly impractical. Consequently, we often impose a far more restricted "computational model" on E_W, then learn only the limited set of "parameters" of the model.

Consider, for example, learning to extract country and capital from pages such as the Germany page in Figure 9.1(a). We may assume that the program E_W is specified by a tuple (s_1, e_1, s_2, e_2), whose meaning is that E_W always extracts the first string between s_1 and e_1 as country and the first string between s_2 and e_2 as capital. In this case, learning E_W reduces to learning the four parameters s_1, e_1, s_2, and e_2, and we may learn that s_1 = `<hr><br>`, e_1 = `<br>`, s_2 = `Capital:`, and e_2 = `<br>`.

Even learning just parameters like these has proven quite difficult. Like the case of learning the source schema, learning the parameters often involves a search in a space of possible values, guided by heuristics. Incorrect heuristics often lead to incorrect parameter values, and the search space is often vast, making the search process time intensive and easy to go wrong.

COPING WITH EXCEPTIONS

The third reason wrapper construction is difficult is that there are often many exceptions in how data are laid out and formatted. For example, the data may normally be laid out as a tuple, say (title, author, price). However, in some cases certain attributes (e.g., price) may be missing, attribute order may be reversed (e.g., listing author before title, if the author is well known), or attribute format may be changed (e.g., price is normally listed in black font, but will be listed in red font if it is below $2).

Such exceptions are common in practice and not always apparent if we only inspect a small number of pages when we begin to create the wrapper. As such, exceptions cause numerous problems. They can invalidate assumptions regarding the schema and data format, thus producing incorrect wrappers. They can also force us to revise considerably the source schema T_S and the extraction program T_W. For instance, in the above book example, to accommodate missing attributes and different attribute orders, we must revise T_S from a flat tuple schema into a nested tuple schema with disjunction. We must also revise the program E_W to handle the many ways that price can be formatted. These revisions blow up the search space, making finding T_s and E_W far more difficult.

9.1.3 Categories of Solutions

Current approaches to constructing wrappers fall into four main groups: manual, learning, automatic, and interactive. In the *manual* approach, a developer examines a set of Web pages, then manually creates the target schema T_W and the extraction program E_W. The program E_W is typically written in a procedural language (e.g., Perl, Java) or a specialized declarative language. This approach is relatively easy to understand, implement, and debug. It also produces highly accurate wrappers. Thus, it is very commonly used

in practice. However, the manual approach is labor intensive and requires highly trained developers.

In the *learning* approach, the developer creates the target schema T_W and highlights the attributes of T_W in a set of Web pages (typically using a graphical user interface that was designed for this purpose). The developer then applies a learning algorithm to these highlighted examples to automatically learn the extraction program E_W. This approach requires less work than the manual approach and can be used by users with less technical skills. However, highlighting attributes still incurs a nontrivial amount of work, and the learned program E_W is often brittle, requiring significant post-processing effort.

The *automatic* approach examines a set of Web pages to automatically infer a grammar that encodes the source schema T_S and a program E_W that extracts data conforming to T_S. Learning arbitrary grammars is impractical, as mentioned earlier. Thus this approach considers only grammars that are restricted forms of regular expressions. The approach requires virtually no developer effort and can be used by technically naive users. But like the learning approach, this approach often produces brittle wrappers, which require significant post-processing effort.

The *interactive* approach combines aspects of the learning and automatic approaches. It examines Web pages and some initial user feedback to infer a set $\mathcal{E}$ of possible extraction programs. It then interactively solicits feedback to refine and narrow $\mathcal{E}$. Example feedback includes highlighting attributes on Web pages, identifying correct extraction results, and visually guiding the process of creating extraction rules. Unlike the learning approach, in which users spend a considerable amount of effort *in advance* to highlight attributes, this approach asks for feedback only when judged necessary to make progress. Thus, it often requires less manual effort. Furthermore, it uses user feedback to guide the search process and thus is more robust than both learning and automatic approaches.

In the rest of this chapter we describe the above four approaches. For ease of exposition, we will use the phrases "wrapper" and "extraction program" as well as "developer" and "user" interchangeably, when there is no ambiguity.

9.2 Manual Wrapper Construction

Manual wrapper construction involves a developer examining a set of Web pages and then creating the target schema T_W and the extraction program E_W. The developer often writes E_W using a procedural language such as Perl. For example, given the Web page in Figure 9.2(a), Figure 9.2(b) shows a Perl program that extracts (country, code) tuples, which in this case are (Australia, 61), (East Timor, 670), and (Papua New Guinea, 675).

In the above example, the Perl program models the page as a long string. An alternative model is to consider the DOM tree of a page. For example, Figure 9.3(a) shows a DOM tree that captures the HTML structure of a movie Web page. With the DOM model, the developer can write the wrapper in Figure 9.3(b) using the XPath language. The first rule of this wrapper

$$title = /html/body/div[1]/table/td[2]/text()$$

is an XPath rule that starts from the root, and then travels through the *body* child, the first *div* child, and *table* before arriving at the second *td* child of *table* and extracting the text value of this child as a movie title. The second rule extracts rating, and the third rule extracts run time in a similar fashion.

Another option is for the developer to employ a visual model of the pages. For example, the page in Figure 9.2(a) can be visually modeled with three blocks: the header "Countries in Australia (Continent)," the region of (country, code) tuples, and the footer "Copyright easycalls.com." Given a page, the developer can use this visual model to locate the second block, then parse this block using a string model to extract (country, code) tuples. In general, visual models often provide effective ways to remove headers, footers, and ads and to locate data regions. String models then provide effective ways to parse data regions to extract the desired data.

Regardless of the page model employed, using a low-level procedural language to write extraction programs can be very laborious. In response, several high-level wrapper languages have been proposed. For example, the HLRT language that we describe in detail

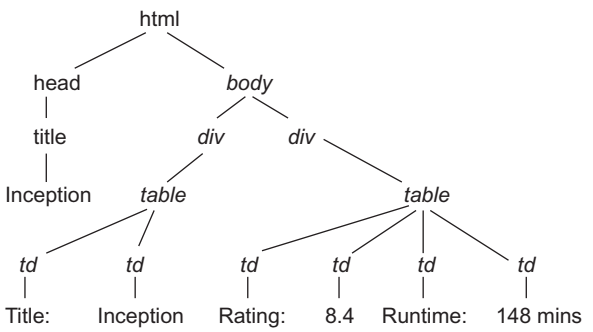
(a)

```
#!/usr/bin/perl -w

open(INFILE, $ARGV[0]) or die "can't open file\n";
while ($line = <INFILE>) {
  if ($line =~ m/<B\>(.+?)\<VB\>\s+?\<I\>(\d+?)\<VI\>\<BR\>/) {
     print "($1,$2)\n";
  }
}
close(INFILE);
```

(b)

FIGURE 9.2 (a) An example page from source *easycalls.com* and (b) a manually constructed wrapper (a Perl program in this case) that extracts data from such pages.



(a)

```
title   = /html/body/div[1]/table/td[2]/text()
rating  = /html/body/div[2]/table/td[2]/text()
runtime = /html/body/div[2]/table/td[4]/text()
```

(b)

FIGURE 9.3 (a) The DOM tree of a Web page about a movie and (b) a wrapper that uses the DOM tree to extract the title, rating, and run time of the movie.

in the next section uses a tuple of $2n+2$ values of the form $(h, t, l_1, r_1, \ldots, l_n, r_n)$ to specify a program for extracting n attributes. Given this tuple, a program E_W extracts the region between the *first* occurrence of h and the *last* occurrence of t as the data region. It then parses this region to extract strings between l_1 and r_1 as the values of the first attribute, between l_2 and r_2 as the values of the second attribute, and so on. If a developer determines that HLRT suffices for his or her needs, such as extracting (country, code) tuples from *easycalls.com*, then writing the wrapper reduces to specifying $2n+2$ parameter values. Section 9.5 discusses other high-level wrapper languages, including datalog variants. Besides saving developer effort, wrappers written in such languages are often easier to understand, debug, and maintain than those in low-level procedural languages.

9.3 Learning-Based Wrapper Construction

While manually constructed wrappers can be very powerful, they often incur labor cost that is impractical when dealing with a large number of data sources. Learning approaches, in contrast, consider only limited wrapper types, but can automatically learn these using training examples. Providing such examples, typically by marking up a set of Web pages, can be done by technically naive users and requires far less work than manually writing the wrappers themselves.

This section explains learning approaches. We use HLRT, a simple wrapper learner, to explain the key ideas underlying wrapper learning. We then describe Stalker, a more complex wrapper learner, and use it to illustrate the full range of complexity in wrapper learning.

9.3.1 HLRT Wrappers

HLRT wrappers use string delimiters to specify how to extract relational tuples. To explain, consider again the page "Countries in Australia (Continent)," reproduced in Figure 9.4(a). Figure 9.4(b) shows the HTML text, which consists of a "head" ending with `<P>`, a "tail" starting with `<HR>`, and a data region in between, which lists (country, code) tuples, with country between `<B>` and `</B>`, and code between `<I>` and `</I>`.

To extract (country, code) tuples from such pages, we can write a simple wrapper that would "chop off" the head using the delimiter `<P>`, "chop off" the tail using `<HR>`, then scan the data region to extract strings between `<B>` and `</B>` as countries, and between `<I>` and `</I>` as codes. Since the head and tail may also contain strings between `<B>` and `</B>`, such as "Countries in Australia (Continent)," which is clearly not a country name, it is necessary to chop them off before scanning for countries and codes.

In HLRT (standing for "head-left-right-tail"), the above wrapper can be represented with a tuple of six strings (`<P>`, `<HR>`, `<B>`, `</B>`, `<I>`, `</I>`). Formally, an HLRT wrapper that extracts n attributes $a_1, \ldots, a_n$ is a tuple of $(2n+2)$ strings $(h, t, l_1, r_1, \ldots, l_n, r_n)$, where h marks the end of the head, t marks the start of the tail, and l_i and r_i delimit

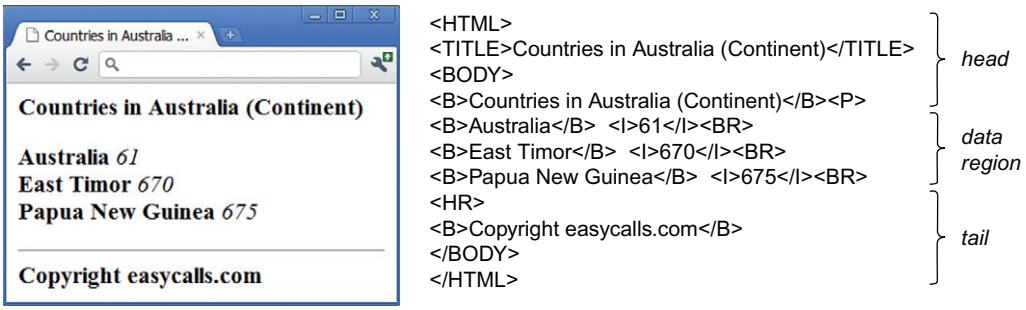


FIGURE 9.4 An example of a Web page from which the **HLRT** wrapper model can use string delimiters to extract relational tuples (**country, code**).

attribute a_i. Such a wrapper does not have to extract *all* attributes present in a page. For example, the HLRT wrapper (`<P>`, `<HR>`, `<B>`, `</B>`) extracts only countries from the page in Figure 9.4(a).

Learning HLRT Wrappers

We now describe how to learn an HLRT wrapper for a data source S. Suppose a developer wants to extract attributes $a_1, \ldots, a_n$ from S. Suppose that after examining some pages of S the developer has established that an HLRT wrapper $W = (h, t, l_1, r_1, \ldots, l_n, r_n)$ will accurately extract these attributes.

Our goal is to learn the $(2n+2)$ parameters $h, t, l_1, r_1, \ldots, l_n, r_n$. To do this, we label a set of pages $T = \{p_1, \ldots, p_m\}$ from source S. Labeling a page p_i means identifying in p_i the start and end positions of *all* values of the attributes $a_1, \ldots, a_n$ and is typically done using a specialized graphical user interface. For example, labeling the page in Figure 9.4(a) involves specifying that "Australia" is a country name, which starts at position 108 and ends at position 116, that "61" is a code, which starts at position 125 and ends at position 126, and so on.

The developer then feeds the labeled pages $p_1, \ldots, p_m$ into a learning module. The simplest kind of learning module systematically searches the space of all possible HLRT wrappers that are consistent with the labeled pages, as follows.

1. *Find all possible values for h:* Let x_i be the string from the beginning of page p_i until (but not including) the first occurrence of the very first attribute a_1. For example,

```
<HTML>
<TITLE>Countries in Australia (Continent)</TITLE>
<BODY>
<B>Countries in Australia (Continent)</B><P>
<B>
```

is such a string for the page in Figure 9.4(b). Clearly, the strings $x_1, \ldots, x_m$ contain the correct h. Thus, we take the set of all common substrings of $x_1, \ldots, x_m$ to be the candidate values for h.

2. *Find all possible values for t:* We can find all candidate values for t in a similar fashion.
3. *Find all possible values for each l_i:* Consider l_1, the left delimiter of attribute a_1. Clearly, l_1 must be a common suffix of all strings (in the labeled pages) that end right before a marked value of attribute a_1. Thus, we can take the set of all such suffixes to be the candidate values for l_1. We proceed similarly to find the candidate values for $l_2, \ldots, l_n$.
4. *Find all possible values for each r_i:* Similarly, we take the set of all common prefixes of all strings (in the labeled pages) that start right after a marked value of attribute a_i to be the candidate values for r_i.
5. *Search in the combined space of the above values:* We combine the above candidate values to form candidate wrappers. If a candidate wrapper W correctly extracts all values of $a_1, \ldots, a_n$ from a page p, we say W is consistent with p. As soon as we find a wrapper that is consistent with all labeled pages $p_1, \ldots, p_m$, we terminate and return that wrapper.

In the bibliographic notes we point to work that discusses how to optimize the above search, and how to select m, the number of pages to be labeled, to guarantee with a high probability that we produce a correct wrapper.

As described, HLRT wrappers are relatively easy to understand and implement. However, they have limited applicability. In particular, they assume a *flat tuple schema* and assume that all attributes can be reliably extracted using delimiting strings. In practice, many sources use more complex schemas, such as *nested tuple schemas*. For example, a page may describe a book as a tuple (title, authors, price), where the attribute authors is actually a list of tuples (first-name, last-name). Furthermore, we may not be able to extract attributes using delimiting strings, such as extracting zip codes from the addresses "4000 Colfax, Phoenix, AZ 85258" and "523 Vernon, Las Vegas, NV 89104." In what follows we describe Stalker wrappers that address these limitations.

9.3.2 Stalker Wrappers

We begin by defining nested tuple schemas, and then we discuss how Stalker learns wrappers that employ such schemas.

Nested Tuple Schemas

The Web page in Figure 9.5(a) is an example of a nested tuple schema. The page lists a restaurant's name, food type, and its multiple addresses. Each address in turn lists the street, city, state, zip code, and phone. Thus the page displays a single tuple (name, food, addresses), where addresses in turn contains multiple tuples (street, city, state, zip-code, phone).

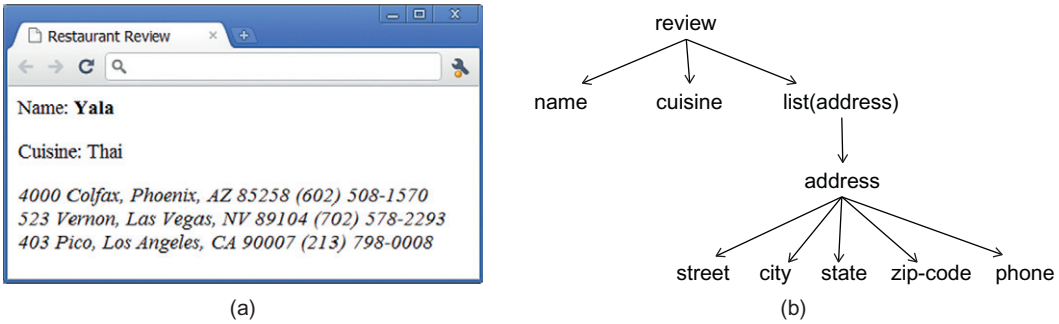


FIGURE 9.5 (a) A Web page that displays data using a nested tuple format and (b) the visualization of this format as a tree.

Nested tuples are very commonly used on Web pages because they are convenient for visual presentation. Formally, the set $\mathcal{T}$ of all nested tuple schemas satisifies the following properties:

- The schema that displays data as a single string belongs to $\mathcal{T}$.
- If $T_1, \ldots, T_n$ belong to $\mathcal{T}$, then the tuple schema $(T_1, \ldots, T_n)$, which generates tuples of the form $(t_1, \ldots, t_n)$ where t_i is an instance of T_i, $1 \leq i \leq n$, also belongs to $\mathcal{T}$.
- Finally, if T belongs to $\mathcal{T}$, then the list schema $\langle T \rangle$, which generates lists whose elements are instances of T, also belongs to $\mathcal{T}$.

A nested tuple schema can be visualized as a tree whose leaves are strings and internal nodes are either tuple nodes or list nodes. The children of a tuple node are the different components of the tuple, while a list node has a single child that describes the type of the instances of the list. Figure 9.5(b) shows the tree that visualizes the schema of the page in Figure 9.5(a). Here a leaf node such as name is a string. The internal node list(addresses) is a list of address nodes, and each address node includes leaves such as street and city.

The Stalker Wrapper Model

A Stalker wrapper specifies a nested-tuple schema in the form of a tree and assigns to each node in the tree a set of rules that show how to extract data values for that node. Figure 9.6(a) shows a Stalker wrapper for restaurant reviews. (To avoid clutter, we omit the rules at certain leaf nodes.)

We demonstrate the wrapper in Figure 9.6(a) by showing how it is executed on the page p in Figure 9.6(b). We begin by assigning p to the root node restaurant. Then for each child node — name, cuisine, and list(address) — we execute the associated rules on the string assigned to the root node to extract the appropriate data values.

Consider executing the rules of node name. The first rule, *Start: SkipTo*(), scans p *from the start forward* until reaching a token , then marks the subsequent token as the *start* of a name. Similarly, the second rule, *End: BackTo*(), scans p *from the end backward* until , then marks the token right before as the *end* of a name. This allows

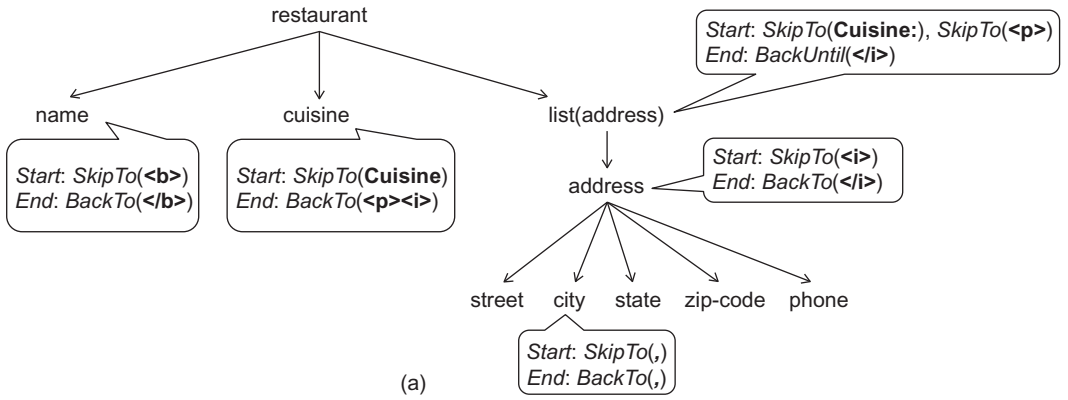


FIGURE 9.6 (a) A **Stalker** wrapper and (b) a target page.

us to pull out "Yala" as the restaurant name. We execute the rules at node cuisine similarly, to extract "Thai" as the cuisine.

Executing node list(address) is a bit more involved. Here the two rules extract the entire list, i.e., a string that contains *all* addresses. Specifically, rule *Start: SkipTo*(`Cuisine:`), *SkipTo*(`<p>`) scans *p* forward until `Cuisine:`, keeps scanning until `<p>`, then marks the next token as the start of the list. (Note that here a single *SkipTo* rule such as *Start: SkipTo*(`<p>`) will not correctly mark the start of the list. It will terminate well before that.) Similarly, rule *End: BackUntil*(`</i>`) scans *p* backward until `</i>`, then marks this token as the end of the list. Note that here *BackUntil*(`</i>`) does not consume `</i>` (unlike *BackTo*(`</i>`)). The entire list is then the string between the marks.

Next, we apply the two rules of node address to the above string to extract the addresses. Rule *Start: SkipTo*(`<i>`) scans the string until the first `<i>`, marks the next token as the start of the *first* address, then keeps scanning until the second `<i>`, marks the next token as the start of the *second* address, and so on. Similarly, rule *End: BackTo*(`</i>`) marks the ends of the addresses. We then extract the strings between the corresponding marks as the addresses.

In the next step, we apply the rules at nodes street, city, state, zip-code, and phone to each address to extract appropriate instances. For example, to extract a city, we scan an address forward until a comma, then backward until a comma, then extract the string in between.

Thus, a Stalker wrapper is executed in a top-down fashion. The root node is assigned a string (which is the Web page). Executing a child node of the root produces a set of substrings, which are then passed as input to the extraction rules of the children of this child node, and so on.

Stalker EXTRACTION RULES

We now describe the extraction rules in detail. Each rule consists of a *context* and a *sequence of commands*. Example contexts are *Start* and *End*, which we have seen earlier. Example sequences of commands are

SkipTo()
SkipTo(Cuisine:),*SkipTo*(<p>)

Each command takes as input a *landmark*, such as , Cuisine:, <p>, or the triple (Name *Punctuation HTMLTag*).

To explain landmarks, we note that a page is viewed as a sequence of *tokens*, which are punctuation symbols, HTML tags, and alphanumeric strings (that are delimited by space, punctuation symbols, or HTML tags). A landmark is then a sequence of tokens and *wildcards*, where each wildcard such as *Punctuation* or *HTMLTag* refers to a class of tokens. A landmark can be viewed as a restricted kind of regular expression that can be matched to the page content. For example, the landmark

Name Punctuation HTMLTag

will match the string Name: .

Executing a command proceeds by consuming text until reaching a string that matches the input landmark. Executing a sequence of commands means executing the commands in the listed order, with the next command starting where the previous command stopped.

To handle variations in the page format, Stalker also considers extraction rules that contain a *disjunction* of sequences of commands. For example, if restaurant names appear in bold for recommended ones and in italic otherwise, then we can use the following rule to mark the start of a restaurant:

Start : either *SkipTo*() or *SkipTo*()

This rule stops when we have consumed a token or an token. Similarly, the following rule marks the end of a restaurant:

End : either *BackTo*() or *BackTo*(Cuisine), *BackTo*()

In general, a disjunctive rule specifies an ordered list of sequences of commands. To apply the rule, we apply the sequences in order until we find a sequence that matches.

Learning Stalker Wrappers

We now describe how to learn a Stalker wrapper. The learner will take as input a nested tuple schema specified by the developer and a set of pages in which the instances of the nodes have been marked up.

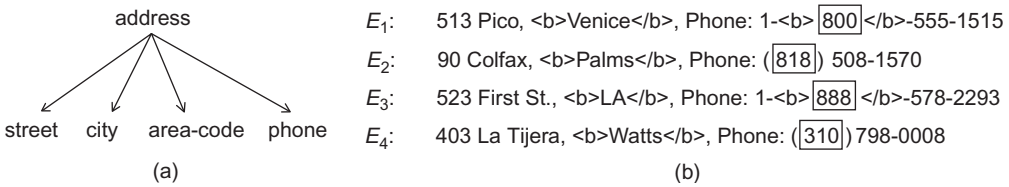


FIGURE 9.7 (a) A simple address schema and (b) four addresses where the occurrences of **area-code** have been marked up.

Our goal is to use the marked-up pages to learn the rules for the nodes of the tree. Specifically, for each leaf node, such as name, we learn a start rule and an end rule, such as the two rules shown in Figure 9.6(a). For each internal node, such as list(address), we learn a start rule and an end rule for extracting the entire list. In what follows we illustrate the learning process by discussing how to learn a start rule for a leaf node.

For ease of exposition, we will consider the simple address schema in Figure 9.7(a) and the four pages E_1–E_4 in Figure 9.7(b), which shows the area codes marked up. Our goal is to use these marked-up examples to learn a start rule for area-code.

We apply a learning technique called *sequential covering*, which proceeds iteratively. The first iteration finds a rule that *covers* (i.e., correctly matches) a subset of the training examples. The second iteration finds a rule that covers a subset of the remaining training examples, and so on, until we have covered all training examples. The final rule is then a disjunction of all the rules found so far.

Continuing with the example in Figure 9.7, we first define the *prefix of an example* to be the string from the start of the example to the start of the area code. For example, the prefix of E_1 is "513 Pico, Venice, Phone: 1-." Next, we select the example with the shortest prefix, which is E_2 in this case. The last token of the prefix is "(", which also matches two wildcards: *Punctuation* and *Anything*. (See Figure 9.8 for sample wildcards that we use in this learning scenario.) So we create three initial candidate rules:

$$R_1 = SkipTo(()), R_2 = SkipTo(Punctuation), R_3 = SkipTo(Anything)$$

Rule R_1 *covers* E_2 and E_4, in that it stops right before an area code. In contrast, rules R_2 and R_3 do not cover any training example. Hence, the first iteration returns the rule R_1.

The second iteration considers the remaining examples: E_1 and E_3. E_1 has a shorter prefix, so we select E_1 and create three initial candidate rules:

$$R_4 = SkipTo(\texttt{<b>}), R_5 = SkipTo(HTMLTag), R_6 = SkipTo(Anything)$$

None of these rules covers any training example, so we select one rule to refine. We select R_4 because it uses no wildcards in the landmark. Refining R_4 produces the 18 candidate rules shown in Figure 9.8. Out of these rules, rules $R_7, R_{11}, R_{12}, R_{13}, R_{15}, R_{16}$, and R_{19} cover all the remaining examples, E_1 and E_3. Hence, we select one rule from these to return. We

Wildcards: *Anything*, *Numeric*, *AlphaNumeric*, *Alphabetic*, *Capitalized*, *AllCaps*, *HTMLTag*, *nonHTML*, *Punctuation*

R_7: *SkipTo*(**- **)
R_8: *SkipTo*(*Punctuation* ****)
R_9: *SkipTo*(*Anything* ****)
R_{10}: *SkipTo*(**Venice**) *SkipTo*(****)
R_{11}: *SkipTo*(****) *SkipTo*(****)
R_{12}: *SkipTo*(**:**) *SkipTo*(****)
R_{13}: *SkipTo*(**-**) *SkipTo*(****)
R_{14}: *SkipTo*(**,**) *SkipTo*(****)
R_{15}: *SkipTo*(**Phone**) *SkipTo*(****)
R_{16}: *SkipTo*(**1**) *SkipTo*(****)
R_{17}: *SkipTo*(*Numeric*) *SkipTo*(****)
R_{18}: *SkipTo*(*Punctuation*) *SkipTo*(****)
R_{19}: *SkipTo*(*HTMLTag*) *SkipTo*(****)
R_{20}: *SkipTo*(*AlphaNum*) *SkipTo*(****)
R_{21}: *SkipTo*(*Alphabetic*) *SkipTo*(****)
R_{22}: *SkipTo*(*Capitalized*) *SkipTo*(****)
R_{23}: *SkipTo*(*NonHTML*) *SkipTo*(****)
R_{24}: *SkipTo*(*Anything*) *SkipTo*(****)

FIGURE 9.8 A sample set of wildcards and a set of candidate rules obtained while learning a start rule for **area-code**.

end up selecting R_7 because it has the longest end landmark (all other rules end with a one-token landmark). Since there are no more uncovered examples, the algorithm terminates returning the disjunctive rule that combines R_1 and R_7 as the start rule for area-code:

Start : either *SkipTo*(() or *SkipTo*(-)

Discussion

The wrapper model of Stalker subsumes that of HLRT, and both can be viewed as modeling finite state automata. Together, HLRT and Stalker illustrate how imposing structure on the target schema language makes learning practical. This structure can be relatively simple, as in the flat tuples of HLRT, or significantly more complex, as in the tree structure of Stalker. Regardless, the structure severely restricts the target languages and transforms the general learning problem into an easier one of learning a relatively small set of parameters, such as delimiting strings in HLRT or extraction rules in Stalker.

Even with the restricted search space, HLRT and Stalker still need to use heuristics to make learning the parameters easier. For example, in each iteration Stalker selects the example with the shortest prefix to generate candidate extraction rules. Stalker then refines each rule by expanding the landmark or adding another command (see Section 9.3.2). Even with these heuristics, we often still face a vast search space. For example, the tiny scenario in Figure 9.7 already generates 18 rules in its second iteration (see Figure 9.8). Complications such as variations in page formats further blow up the search space, making learning brittle.

9.4 Wrapper Learning without Schema

The so-called *automatic approaches* to wrapper learning take as input a set of Web pages of a source S, examine the similarities and dissimilarities across the pages, and automatically infer the schema T_S of the pages and a program E_W that extracts data conforming to T_S.

(a)

AB+C?D

(b)

```
<HTML><B>#PCDATA</B><P>(<U>#PCDATA</U><BR>)+
(<I>Price:</I>#PCDATA<BR>)?<I>Publisher:</I>#PCDATA<BR></HTML>
```

(c)

A	B	C	D
The Elements of Style	William Strunk Jr.	$9.95	Longman
	E.B. White		
The Snow of Kilimanjaro	Ernest Hemingway		Scribner

(d)

FIGURE 9.9 (a) Two Web pages from the same data source, (b) the schema T_S of the pages, expressed as a regular expression, (c) the extraction program E_W, and (d) the data extracted by E_W from the above two pages.

As an example, given the two pages that describe books in Figure 9.9(a), an automatic approach may return the schema T_S in Figure 9.9(b) and the extraction program E_W in Figure 9.9(c), both written in a regular-expression language variant. The schema T_S shows that each page lists an attribute A, one or more values of attribute B, optionally attribute C, then attribute D. The extraction program E_W shows how to parse a page to extract the data of the attributes A–D (encoded as `#PCDATA` in the program). Applying E_W to the two pages in Figure 9.9(a) produces the table of extracted data in Figure 9.9(d).

Thus, in contrast to the methods described in the previous sections, automatic methods do not take the target schema as input. Consequently, they cannot assign meaningful names (e.g., author, title) to the attributes of the schema they learn but only generic ones (e.g., A and B). The main advantage of automatic wrapper learners is that they require no human intervention.

We now describe RoadRunner, a representative automatic approach. We illustrate how RoadRunner models the target schema T_S and the extraction program E_W, then how it infers T_S and E_W from a set of Web pages.

9.4.1 Modeling Schema T_S and Program E_W

The Web pages of source S use the schema T_S to display data. RoadRunner models T_S as a nested-tuple schema (see Section 9.3.2). Recall that such a schema nests tuples and lists and allows certain kinds of *optionals* and *disjunctions*. RoadRunner allows *optionals* in that

certain attributes (e.g., attribute C in Figure 9.9(d)) can be missing from a page. But it does not allow disjunctions. Thus, T_S can be expressed as a *union-free regular expression*, such as the one shown in Figure 9.9(b).

RoadRunner models the extraction program E_W as a regular expression that when evaluated on a Web page will extract the attributes of T_S. Figure 9.9(c) shows such a regular expression R. Given a Web page, R matches `<HTML><B>` from the start of the page, then matches and returns the string up to the first `</B>` as the value of attribute A, and so on. On the first Web page of Figure 9.9(a), for example, R would return "The Elements of Style" as the value of A. In general, the fields `#PCDATA` of R are the "slots" for the values of the attributes of T_S, and these values are not supposed to contain any HTML tag (such as `<B>`).

The assumptions of no disjunction in T_S and no HTML tag in attribute values reduce the complexity of finding T_S and E_W. But they do limit the applicability of RoadRunner. The bibliographic notes discuss works that relax these assumptions.

9.4.2 Inferring Schema T_S and Program E_W

Given a set of Web pages $\mathcal{P} = \{p_1, \ldots, p_n\}$ from the source S, RoadRunner examines $\mathcal{P}$ to infer the extraction program E_W and to infer the schema T_S from E_W. In the rest of this section we focus on the first step of inferring E_W; the second step of inferring T_S from E_W is relatively straightforward.

To infer the extraction program E_W, RoadRunner proceeds iteratively. It begins by initializing E_W to be a page from $\mathcal{P}$, say p_1. Page p_1 can clearly be viewed as a regular expression that matches only p_1. Thus at this point E_W matches only p_1. RoadRunner then takes another page from $\mathcal{P}$, say p_2, and attempts to generalize E_W so that E_W can also match p_2. Continuing in this way, in the end RoadRunner returns an E_W that has been generalized (in a minimal fashion) to match all pages in $\mathcal{P}$.

We now focus on the generalization step. To illustrate, we will use the example in Figure 9.10, where E_W has just been initialized to be the page p_1, and we have to generalize it to match a target page p_2. We proceed with the following steps.

Tokenizing the Target Page

We first convert the target page p_2 into a sequence of tokens, where each token is an HTML tag or a string (that does not contain any HTML tag). The right side of Figure 9.10 shows how p_2 has been converted into 27 tokens (one per line, except lines 09-11, 15-17, and 21-23, which contain three tokens each).

Generalizing Program E_W to Match the Target Page

Next, we apply E_W to match p_2. Continuing with our example, currently E_W is just the regular expression represented by the page p_1. To facilitate matching p_1 with p_2, we also display p_1 one token per line, as shown in the left side of Figure 9.10 (except lines 08-10 and 14-16, which contain three tokens each).

We now match p_1 with p_2 line by line, from the top down. If we reach the end of p_2, then E_W has successfully matched p_2 and does not have to be generalized. Otherwise, there is

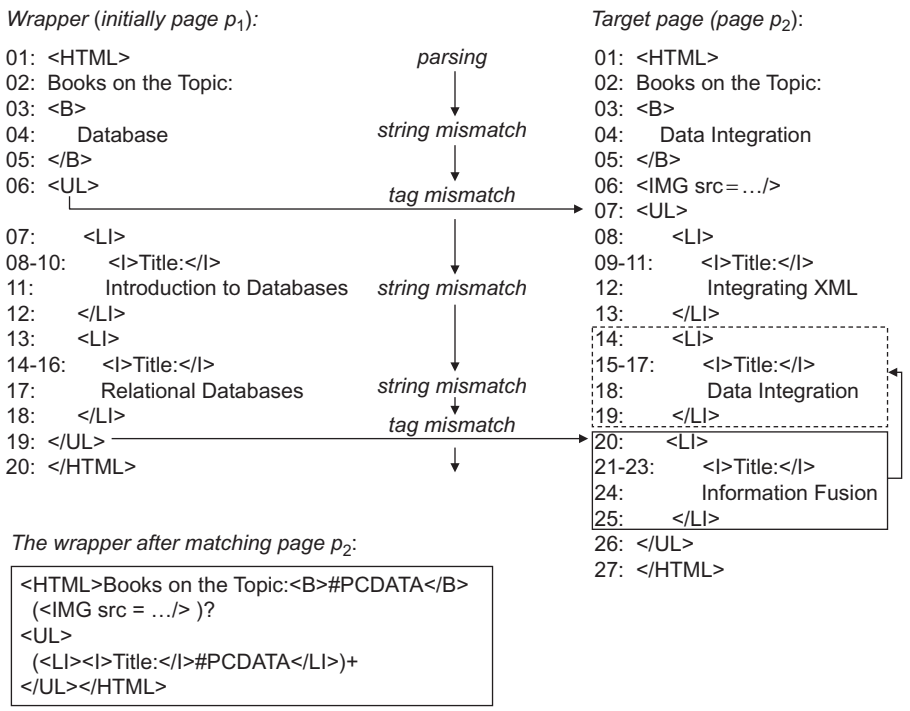


FIGURE 9.10 Generalizing the extraction program E_W, which is currently just page p_1, to also match page p_2.

a *mismatch*. A *string mismatch* involves two strings, such as "Database" on line 04 of p_1 versus "Data Integration" on line 04 of p_2. A *tag mismatch* involves an HTML tag and a string, or two tags, such as `<UL>` on line 06 of p_1 versus `<IMG src=.../>` on line 06 of p_2. We must generalize the extraction program E_W to resolve these mismatches.

It is not difficult to show that if a string mismatch happens, we have just discovered a new attribute, and the two strings are two different values of that attribute. To resolve this mismatch, we generalize E_W by adding a new `#PCDATA` slot to capture the new attribute. For example, after detecting the string mismatch "Database" versus "Data Integration" on line 04 of p_1 and p_2, we generalize the initial part of E_W from

```
<HTML>Books on the Topic:<B>Database</B>
```

to

```
<HTML>Books on the Topic:<B>\#PCDATA</B>
```

Resolving a tag mismatch is far more difficult. Such a mismatch happens due to either an *iterator* or an *optional*. For example, the mismatch `<UL>` versus `<IMG src=.../>` on line 06 is due to an optional image on page p_2. The mismatch `</UL>` on line 19 of p_1 versus `<LI>` on line 20 of p_2, on the other hand, is due to an iterator, and it comes from the different lengths of the book lists (two books in p_1 versus three books in p_2).

When a tag mismatch happens, we first try to find if it is due to an iterator. If it is, we generalize E_W to incorporate the iterator. Otherwise, we assume it is due to an optional and generalize E_W accordingly (later we explain why we look for iterators before optionals). We now discuss the two cases, starting with the case of optional.

RESOLVING AN OPTIONAL MISMATCH

We begin by detecting which page includes the optional by searching for the mismatched strings in the pages. Consider again the mismatch `<UL>` versus `<IMG src=.../>` on line 06 of pages p_1 and p_2. There are two cases.

1. String `<UL>` is the optional. Then after skipping it we should be able to continue by matching `<IMG src=.../>` of page p_2 with the first occurrence of `<IMG src=.../>` in the rest of p_1.
2. String `<IMG src=.../>` is the optional. Then after skipping it we can continue by matching `<UL>` of page p_1 with the first occurrence of `<UL>` in the rest of p_2.

Since `<UL>` occurs in the rest of p_2 (after `<IMG src=.../>`), and `<IMG src=.../>` does not occur in the rest of p_1 (after `<UL>`), it is clear that we are in Case 2, that is, `<IMG src=.../>` is the optional.

Once we have found which string is the optional, it is relatively straightforward to generalize E_W. Continuing with the above example, we generalize E_W by introducing the pattern `(<IMG src=.../>)?`, then resume matching at tokens `<UL>` on line 06 of p_1 and line 07 of p_2, respectively.

RESOLVING AN ITERATOR MISMATCH

An iterator repeats a pattern, which we will refer to as a *square*. For example, page p_1 contains a list of two books, where each book description is a square of the form `<LI><I>Title:</I> ... </LI>`. An iterator mismatch happens when the two lists differ in their numbers of squares. For example, pages p_1 and p_2 contain lists of two and three books, respectively. This causes an iterator mismatch at line 19 of p_1 and line 20 of p_2 (tokens `</UL>` versus `<LI>`).

To resolve such a mismatch, we must first find the squares, then use them to find the lists, then generalize E_W to account for the lists. Let the two lists be $U = u_1 u_2 \ldots u_n$ and $V = v_1 v_2 \ldots v_m$, where the u's and v's are squares. Suppose $n < m$; then by the time we run into an iterator mismatch, we can conclude that we have successfully matched u_1 with v_1, u_2 with v_2, and so on, until u_n with v_n. The mismatch happens the moment we move on to the first token of v_{n+1}.

This implies that (a) the last token right before the mismatch must be the last token of u_n and v_n, that is, *the last token of a square*, and (b) one of the mismatched tokens must be the first token of v_{n+1}, that is, *the first token of a square*. This allows us to know the overall form of the square. For example, consider the mismatch at line 19 of p_1 and line 20 of p_2. The last token right before the mismatch is `</LI>`, and the mismatched tokens are `</UL>` and `<LI>`. Thus, the square is either of the form `</UL> ... </LI>` or `<LI> ...`

`</LI>`. Next, we search the pages p_1 and p_2 (only the portions after the mismatch point) for square candidates of these forms. It is easy to see that there is only one square candidate, `<LI> ... </LI>`, spanning lines 20-25 of page p_2.

Once we have found a square candidate s, we "double check" that s is indeed a square, by matching it against the square immediately above it, in a backward fashion. In the above example, we start by matching line 25 of page p_2 with line 19 of the same page, then line 24 with line 18, and so on. If the match is successful, we declare s a true square.

Next, we generalize the extraction program E_W, by searching for contiguous repeated occurrences of s around the mismatch region, then replacing those with $(s)+$. For example, we replace squares `<LI> ... </LI>` with

```
<UL>
   (<LI><I>Title:</I>#PCDATA</LI>)+
</UL>
```

as shown in Figure 9.10.

We can now describe the entire process of matching pages p_1 and p_2 in Figure 9.10. The first mismatch at line 04 is a string mismatch. This is resolved by adding `#PCDATA` to E_W. The next mismatch at line 06 is a tag mismatch. To resolve this, we first assume that this is an iterator mismatch. This produces two square candidates: `<UL> ... </B>` and `<IMG src=.../> ... </B>`. We can quickly see that neither candidate is a true square, because the rest of page p_1 and the rest of page p_2 (after line 06) do not contain `</B>`. Thus, this is not an iterator mismatch. Next, we assume this is an optional mismatch, and resolve it with `(<IMG src=.../>)?`.

We then resume matching line 07 of p_1 with line 08 of p_2. The string mismatches at lines 11 versus 12 and lines 17 versus 18 are resolved with `#PCDATA`. The next mismatch at lines 19 versus 20 is a tag mismatch. We have described earlier how to resolve this as an iterator mismatch. After this, matching resumes at lines 19 versus 26 and ends at the last tokens of p_1 and p_2. At this point, the original program E_W, which is page p_1, has been successfully generalized into the program in the bottom of Figure 9.10, to match both p_1 and p_2.

The above example also makes clear why in the case of a tag mismatch, we want to look for an iterator first. Consider the tag mismatch at lines 19 versus 20 (`</UL>` versus `<LI>`). If we look for an optional first, we will generalize E_W so that it assumes each page contains two books, with a third book being optional. It will miss the list of books entirely and thus is clearly incorrect.

Finally, we note that resolving an iterator mismatch often involves *recursion*. To see this, consider a simple example in which we have found the square candidate in Figure 9.11(a). To verify that it is a true square, we want to match it against the square in Figure 9.11(b), in a backward fashion. We start out matching `</LI>` with `</LI>`, then `<B>Jane Lee</B>` with `<B>James Madison</B>`. Then we have a tag mismatch `</B>` versus `Data Integration`. This happens because each book square contains a list of authors, and the two squares in the above figure differ in the number of authors, causing the mismatch. Thus, while resolving an outer mismatch, we may run into an inner mismatch, which in turn may cause

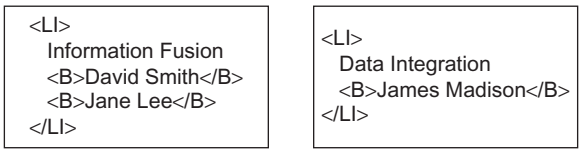


FIGURE 9.11 An example to illustrate that resolving an iterator mismatch often involves recursion.

further mismatches, and so on. Clearly, the mismatches must be resolved from inside out, in a recursive fashion.

Reducing Runtime Complexity

As described, to generalize the extraction program E_W to match a page p, we must consider the following:

- We must detect and resolve all mismatches.
- For each mismatch, we must decide if it is a string mismatch (thus introducing a new attribute), an iterator mismatch, or an optional mismatch.
- For an iterator or optional mismatch, we can search on either the side of the program E_W or the side of the target page p. For example, in the case of an optional mismatch, the optional can be either on E_W or on p.
- For an iterator or optional mismatch, even when we limit the search to just one side, there are often multiple square candidates and optional candidates to consider.
- To resolve an iterator mismatch, it may be necessary to recursively resolve multiple inner mismatches first.

As a consequence of the above points, we are typically faced with a rather large search space, with multiple options at each decision point, and when we "dead end," we must backtrack to the closest decision point and try another option. In fact, it can be shown that the above generalization algorithm incurs exponential run time with respect to the length of the inputs.

Consequently, RoadRunner employs three heuristics to reduce the run time. First, it limits the number of options at each decision point by ranking and retaining only the top k options. For example, it ranks optional candidates based on their lengths, then considers only the top four.

Second, RoadRunner does not allow backtracking at decision points where it thinks the chance that it has been wrong is very low. For example, at a decision point that considers whether the mismatch is due to an iterator or an optional, if RoadRunner has found an iterator, then it disables further backtracking. That is, it will never revisit this decision point and explore the option that the mismatch may be due to an optional.

Finally, RoadRunner ignores certain iterator and optional patterns judged to be highly unlikely. For example, it does not consider any iterator or optional pattern that is delimited on either side by an optional pattern, such as `((<HR>)?<LI>#PCDATA</LI>)` and `(<BR>)?(<HR>)?`.

9.5 Interactive Wrapper Construction

Learning and automatic approaches to wrapper construction often use heuristics to reduce the time it takes to search the huge space of wrapper candidates. Such heuristics are not perfect, and as a result, these approaches have often been brittle: sometimes they produce correct wrappers and sometimes they do not, but we cannot use them blindly because we do not know when they are correct.

Interactive approaches address this problem by injecting user feedback into the search process. They start with little or no input from the wrapper developer and search the space of wrappers until uncertainty arises. At that point they ask for user feedback, then resume searching, until they converge on a wrapper that satisfies the user. In these systems, user feedback can come in multiple forms: users can label a new Web page, identify the correct extraction result, visually create extraction rules, answer questions posed by the system, or identify page patterns. The main challenge faced by these systems is to decide *when* to solicit feedback from the user and *what* question to pose to them.

In what follows we describe three representative works using the interactive approach. We describe only the basic ideas underlying the works and point to more elaborate descriptions in the bibliographic notes.

9.5.1 Interactive Labeling of Pages with Stalker

Recall that Stalker asks the user to label a set of Web pages, then uses these pages to search for a wrapper. We can modify Stalker to be interactive, in that it asks the user to label pages *during* the search process. Specifically, Stalker asks the developer to label a page (or a few pages) and uses this page to build an initial wrapper. Then Stalker can interleave search with soliciting user feedback until it finds a satisfactory wrapper. To decide which page to ask the user to label next, Stalker maintains two candidate wrappers and finds pages on which the two wrappers disagree. It then asks the user to label one of these "problematic" pages.

Stalker employs a form of active learning called *co-testing*, in which alternative hypotheses (wrappers in this case) are co-tested on a new page to see if they disagree. We now describe the interactive Stalker and its co-testing mechanism in detail.

1. **Initialization:** The user labels one or several Web pages. In our example, to begin extracting phone numbers of restaurants, the user would label the phone numbers in Figure 9.12.

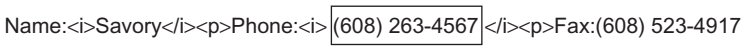


FIGURE 9.12 To start the process of wrapper construction, the interactive **Stalker** may ask the user to label a phone number on a restaurant address.

2. **Learning two wrappers:** Next, we learn to extract phone numbers from the labeled pages. This means learning to mark the start and end of a phone number (see Section 9.3.2). For simplicity, let us focus on learning to mark the *start* of a phone number. In this case, we may learn a rule such as

$$R_1 : SkipTo(\texttt{Phone:<i>})$$

   This is known as a *forward* rule in that it consumes the page forward from the start, until reaching the end of `Phone:<i>`. Alternatively, we can also learn a *backward* rule such as

$$R_2 : BackTo(\texttt{Fax}), BackTo(())$$

   which consumes the page backward from the end (see Section 9.3.2). Both rules mark the start of phone numbers. The traditional Stalker learns just one of these rules. But the interactive Stalker will learn both, thus in a sense learning two alternative wrappers.
3. **Applying the wrappers to find a problematic page:** Next, let P be a large set of unlabeled pages available to Stalker. Stalker finds all pages in P on which the two wrappers disagree. In the above example, this means pages on which the forward and backward rules identify two different sets of the start of phone numbers. Stalker can either randomly select a page or select one with the most disagreements.
4. **Soliciting feedback and relearning:** Since the rules disagree on the selected page p, at least one of them must be wrong, and user feedback can help us identify which one. Thus, we remove p from the set of unlabeled pages P, ask the user to label it, add it to the set of labeled pages, then go back to Step 2 to relearn the rules. We repeat steps 2-4 until there are no more problematic pages in P. (If we have exhausted all pages in P, we can expand P by adding more unlabeled pages.) In the end we have two wrappers and can return as the output one or both wrappers.

9.5.2 Identifying Correct Extraction Results with Poly

The second interactive wrapper learning system we describe is Poly, and it differs from Stalker in several ways. While Poly also uses co-testing, it maintains multiple candidate wrappers instead of just two. Second, instead of asking a user to label a page, it applies the wrappers to the page, then asks the user to identify the correct extraction result. Finally, instead of using the string model, Poly uses DOM tree and visual models to build wrappers.

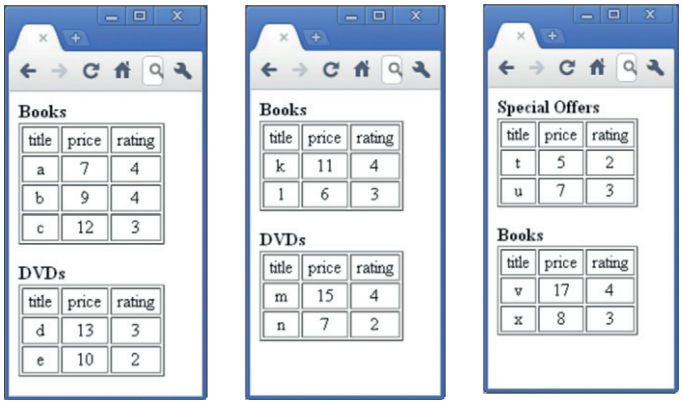


FIGURE 9.13 Poly starts by asking the user to highlight a tuple on the Web page in (a), then later uses pages in (b) and (c) to evaluate the generated wrappers.

1. **Initialization:** Poly assumes that there are multiple tuples per page, and that the user wants to extract a subset of these tuples. Thus, it starts by asking the user to label a target tuple on a page by highlighting the attributes of the tuple.

   Consider, for example, the Web page in Figure 9.13(a). Suppose we want to extract all tuples with rating 4 in table "Books." Then the user defines the attributes of the output schema to be *title*, *price*, and *rating*, and highlights these attributes in a tuple, say the first tuple $(a, 7, 4)$ in table "Books."
2. **Using the labeled tuple to generate multiple wrappers:** Next, Poly generates a set of wrappers $\mathcal{W}$, such that each wrapper extracts from the current page a set of tuples that contains the highlighted tuple. In the above example, Poly may generate wrappers that extract (1) all book and DVD tuples, (2) just book tuples, (3) book and DVD tuples with rating 4, (4) just book tuples with rating 4, (5) the first tuple of all tables, or (6) just the first tuple of the first table. All of these wrappers extract the labeled tuple $(a, 7, 4)$.
3. **Soliciting the correct extraction result:** Next, Poly shows the user the extraction results produced by wrappers in $\mathcal{W}$ on the page and asks the user to identify the correct one. Poly then removes from the set $\mathcal{W}$ all wrappers that do not produce the correct result.

   In the above example, since we want to extract book tuples with rating 4, the user would identify the set $\{(a, 7, 4), (b, 9, 4)\}$ as the correct result. This would remove wrappers such as (1) extract all book and DVD tuples, (2) just book tuples, (3) the first tuple of all tables, or (4) just the first tuple of the first table. The set $\mathcal{W}$ still contains wrappers that give correct extraction results on the current page, such as (1) all book and DVD tuples with rating 4, (2) book tuples with rating 4, and (3) all tuples with rating 4 from the first table.
4. **Evaluating the remaining wrappers on verification pages:** Next, Poly applies all wrappers in $\mathcal{W}$ to a large set of unlabeled pages Q to see if the wrappers disagree. For example, when applied to the page in Figure 9.13(b), the two wrappers "extracting all

book and DVD tuples with rating 4" and "extracting book tuples with rating 4" disagree in that they extract different sets of tuples. As another example, when applied to the page in Figure 9.13(c), the two wrappers "extracting book tuples with rating 4" and "extracting all tuples with rating 4 in the first table" disagree.

As soon as Poly finds a disagreement on a page q, it repeats Steps 3-4. That is, it asks the user to again select the correct result on q, removes from $\mathcal{W}$ all wrappers that do not produce the correct result, then evaluates $\mathcal{W}$ on pages in Q again. Poly terminates, returning $\mathcal{W}$ when all wrappers in $\mathcal{W}$ agree on all pages in Q.

We note that in Step 3 of the above algorithm, to help the user quickly find the correct result, Poly can show the results in ranked order of decreasing likelihood of being the correct result. Furthermore, if the user does not find the correct result, then he or she can edit a result shown by Poly into a correct one, or highlight another tuple on the current page.

GENERATING THE WRAPPERS

We now look in detail at how Poly generates the wrappers. Suppose the user has just highlighted the tuple $(a, 7, 4)$ on the page in Figure 9.13(a). Poly first converts the page into a DOM tree, such as the simplified tree shown in Figure 9.14. Next, Poly identifies the nodes that correspond to the highlighted attributes. They are the three nodes marked with squares in Figure 9.14. Poly finds the least common ancestor node of these squared nodes, which is the node `<tr>` with ID 8 in the figure. We refer to this node as a *tuple node*, because its subtree contains the data of a single tuple.

Next, Poly creates an XPath-like expression E that denotes the path from the root to the highlighted tuple node: $E = /html/table/tr$. It applies E to the entire page to find all potential tuple nodes, which are `<tr>` nodes with IDs 7-13 in the figure. The intuition is that the path from the root to the potential tuple nodes must be similar to the path from the root to the highlighted tuple node.

Next, Poly creates wrappers, each of which extracts a subset of the potential tuple nodes. It does this in a top-down, level-by-level fashion, and uses an XPath-like language to encode the wrappers. At level 1, all wrappers start with $/html$.

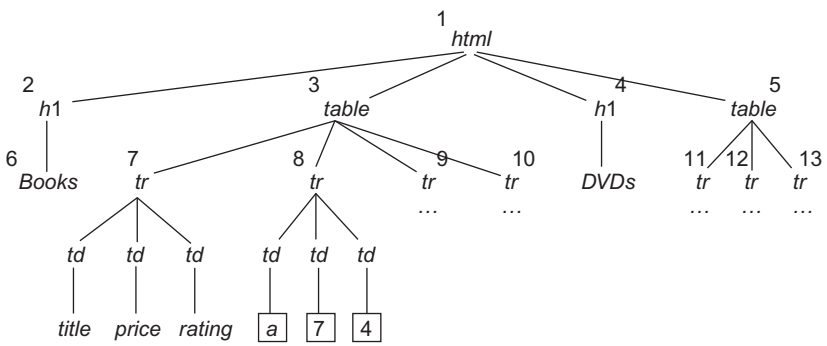


FIGURE 9.14 The DOM tree of the Web page in Figure 9.13(a).

At level 2 of the DOM tree, the wrappers must "touch" node 3 or node 5, or both of them, in order to get to the possible tuple nodes at lower levels. Thus, we refine the partial wrapper */html* into a set of wrappers: */html/table*[1] (which touches node 3), */html/table*[2] (node 5), */html/table* (both nodes), and */html/table*[*prevSibling*(*h*1, *Books*) = *true*] (node 3), among others. Here *prevSibling* is a predefined predicate, and wrapper */html/table*[*prevSibling*(*h*1, *Books*) = *true*] touches the node table that follows a sibling node whose name is "h1" and whose text content contains "Books."

At level 3, a wrapper may touch any subset of the potential tuple nodes, and Poly generates these wrappers accordingly. For example, it may generate wrapper */html/table*[1]/*tr*, which extracts all tuples of the first table, or it may generate wrapper */html/table*[*prevSibling*(*h*1, *Books*) = *true*]/*tr*, which extracts all tuples of the table "Books," and so on. Poly proceeds similarly at subsequent levels. In general, the set of wrappers that Poly generates depends on the expressiveness of the internal XPath-like language that Poly uses to encode the wrappers.

To create wrappers, Poly uses a visual model of the page in addition to the DOM tree model. The visual model helps remove incorrect tuple nodes. For example, the path */html/table/tr* may extract not just true tuple nodes, but also ad nodes, which contain large advertisement regions. To remove such ad nodes, Poly can compute the visual rectangle of the rendered page that covers a tuple set, then discard this rectangle and the associated tuple set if it exceeds a prespecified size.

9.5.3 Creating Extraction Rules with Lixto

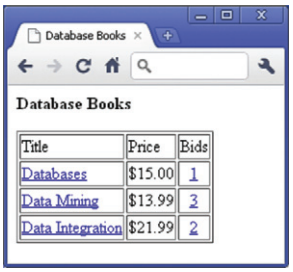
The Lixto system uses a new user interaction mode. Instead of labeling pages or selecting extraction results, users visually create extraction rules in Lixto using highlighting and dialog boxes. In addition, Lixto differs from the previous systems we described by encoding the extraction rules internally using an expressive datalog-like language, defined over both a DOM tree model and a string model of the pages.

Creating the Extraction Rules Visually

In our discussion we use the example Web page in Figure 9.15(a), which lists books being auctioned. To extract these books, a user can visually create four extraction rules. The first rule extracts the books themselves (i.e., the page regions that contain the books), and the next three rules extract the title, price, and number of bids from each book, respectively. Lixto encodes these rules internally as rules R_1–R_4 in Figure 9.15(b). We explain these rules in detail at the end of this section.

Note that in general, rules can depend on one another. For example, rule R_1 extracts books, and rule R_2 extracts titles from the result of R_1. As another example, a rule R_5 that extracts the currency (e.g., "$", "£") would depend on R_3, because R_5 would take as input prices (e.g., "$15.00"), which are the output of R_3.

We now describe how the user visually creates the rules. In the above example, the user starts by creating a rule to extract books. To do this, he or she highlights a book tuple,



(a)

R_1: *book*(*S*,*X*) ← *page*(*S*), *subelem*(*S*,.*table*,*X*)
R_2: *title*(*S*,*X*) ← *book*(-,*S*), *subelem*(*S*,(*.*td*.*.*content*,[(*href*,,*substr*)]),*X*), *notbefore*(*S*,*X*,.*td*,100)
R_3: *price*(*S*,*X*) ← *book*(-,*S*), *subelem*(*S*,(*.*td*,[(*elementtext*,*var*[*Y*].*,*regvar*)]),*X*), *isCurrency*(*Y*)
R_4: *bids*(*S*,*X*) ← *book*(-,*S*), *subelem*(*S*,*.*td*,*X*), *before*(*S*,*X*,.*td*,0,30,*Y*,-), *price*(-,*Y*)

(b)

FIGURE 9.15 (a) A Web page that lists a set of auctions and (b) an internal datalog-like program that **Lixto** generates to wrap such pages.

say the first tuple (i.e., "Databases") in Figure 9.15(a). Lixto maps this tuple into the corresponding subtree of the DOM tree of the page, then extrapolates from the subtree to create rule R_1 in Figure 9.15(b), which extracts the `<table>` subtrees as book tuples. Next, Lixto shows these newly extracted tuples (i.e., "Data Mining" and "Data Integration") to the user. These tuples are correct, and hence the user accepts R_1. (However, note that the user never sees R_1, which is kept internally by Lixto.)

To create a rule to extract titles, the user first specifies that this rule will extract from the book instances identified by rule R_1. Next, the user highlights a title, say the first title "Databases" in Figure 9.15(a). Lixto uses this title to create the internal rule

$$title(S,X) \leftarrow book(-,S), subelem(S,(*.td.*.content,[(href,,substr)]),X)$$

which says that if a data item is inside a `<td>` cell of a table and it is linked (i.e., it contains a "href" tag), then it is a title. Lixto uses this rule to extract, and shows the user all the titles in Figure 9.15(a).

The user realizes that this rule is too general because it extracts not only the titles but also the bids (which are also linked). Hence, the user restricts the rule by specifying that no other data cell should exist within 100 characters *before* a title. This condition removes all bids (because a price data cell exists before each bid) and generates rule R_2 in Figure 9.15(b), which is correct and is accepted by the user. The user then proceeds similarly to create rules R_3–R_4 that extract price and number of bids, respectively.

The user can iteratively restrict or relax a rule using a set of conditions defined by Lixto, such as the target instance (1) must appear before or after a specific element, or (2) must not be close to a specific element, or (3) must not contain a specific element. The user specifies these conditions with highlighting and dialog boxes.

In addition to highlighting and dialog boxes, the user can also write regular expressions or refer to real-world concepts defined by Lixto. For example, to extract the currency (e.g., "$", "£"), he or she may specify that the currency must be a substring of a price string (e.g., "$15.00"), and satisfy predicate *isCurrency*() supplied by Lixto. This allows Lixto to generate

the following rule (the format of which we will explain soon):

$$R_5 : \mathit{currency}(S,X) \leftarrow \mathit{price}(-,S), \mathit{subtext}(S, \backslash \mathit{var}[Y], X), \mathit{isCurrency}(Y)$$

Finally, we note that to correctly extract an attribute (or a tuple), the user often writes *multiple* rules, each covering a formatting variation. For example, if book titles are either linked or in italics, we need two separate rules. The set of titles is then the union of the output of these two.

Representing the Extraction Rules

Lixto encodes an extraction rule with datalog rules in which the head predicate specifies the instance to be extracted and the body predicates, which include predicates over the DOM tree, specify constraints that must hold true.

Consider the rule R_1 in Figure 9.15(b). The head $\mathit{book}(S,X)$ states that X is a book to be extracted and it comes from S. The body predicate $\mathit{page}(S)$ states that S is a Web page, and $\mathit{subelem}(S, .\mathit{table}, X)$ states that X is a subtree with root `<table>` in the DOM tree of S. In general, $\mathit{subelem}(S,P,X)$ states that X is a subtree of S, with the root of the subtree being a node that satisfies the XPath-like expression P.

In rule R_2, the head $\mathit{title}(S,X)$ states that X is a title to be extracted from S. The body predicate $\mathit{book}(-,S)$ states that S is a book. In predicate $\mathit{subelem}(S, (*.\mathit{td}.* .\mathit{content}, [(\mathit{href}, , \mathit{substr})]), X)$, the expression $(*.\mathit{td}.* .\mathit{content}, [(\mathit{href}, , \mathit{substr})])$ specifies paths in the DOM tree that lead to `<td>` nodes that contain a link (see the reference to "href" in the expression). Thus, the entire predicate says that X is a subtree of S with root `<td>` and that X contains a link. Finally, the predicate $\mathit{notbefore}(S,X,.\mathit{td},100)$ states that there is no subtree of S with root `<td>` that exists *before* X, within a distance of 100 (for a predefined notion of distance).

Rule R_3 in Figure 9.15(b) can be explained similarly. In this rule, the predicates $\mathit{subelem}(S, (*.\mathit{td}, [(\mathit{elementtext}, \backslash \mathit{var}[Y].*, \mathit{regvar})]), X)$ and $\mathit{isCurrency}(Y)$ jointly state that (a) the price X is a subtree of S with root `<td>` and (b) the text of this subtree matches the regular expression $\backslash \mathit{var}[Y].*$, which is a currency sign followed by zero or more characters.

Finally, in rule R_4, the predicates $\mathit{before}(S,X,.\mathit{td},0,30,Y)$ and $\mathit{price}(-,Y)$ jointly state that (a) the bid X is a subtree of S with root `<td>` and (b) a price must exist within a distance 0-30 before X.

To generate rules such as the ones above, Lixto generalizes over the DOM tree of the page. For example, when the user highlights a book tuple, Lixto maps the tuple into a subtree with root `<table>`, then generates rule R_1, which says that any subtree with root `<table>` is a book tuple. As such, Lixto generates rules in a way analogous to the way Poly generates wrappers, using the DOM structure. Unlike Poly, however, Lixto can also combine a string model of the page with the DOM tree model to create rules, such as rule R_5 to extract the currency sign.

Bibliographic Notes

Work on wrapper construction dates back to 1997. Recent surveys include [115, 362, 392]. A variety of page schemas, including flat and nested tuple schemas, are discussed in [392, Chapter 9]. Gold [256, 257] shows that learning regular languages from examples alone is very difficult. Building on this work, Grumbach and Mecca [273] and subsequent work in the RoadRunner [153, 154] and ExAlg [33] projects discuss why it is difficult to apply a grammar inference approach to learn arbitrary page schemas.

Early work on wrapper construction used manual techniques. Well-known projects include W4F [503], Minerva [152], XWRAP [394], TSIMMIS [290, 291], WebOQL [39], FLORID [401], Jedi [308], and [41, 276, 292].

Wrapper construction systems that use learning techniques include SHOPBOT [195], WIEN [358, 360], Stalker [449, 450], SoftMealy [305], WL^2 [147], and [374]. The HLRT wrappers were introduced in [360] and discussed in detail in [357, 358]. The Stalker wrappers were introduced in [450]. Several works mentioned in the learning approach, such as CRYSTAL [529], WHISK [528], RAPIER [105], and SRV [151], are designed to operate primarily on free text, such as news articles. Extracting structured data from free text is referred to as *information extraction* (see [506] for a survey) and is often viewed as a problem distinct from that of wrapper construction.

Automatic approaches to wrapper construction include IEPAD [116], RoadRunner [153, 154], ExAlg [33], DeLa [562], and DEPTA [589]. RoadRunner assumed no disjunction in the target regular expression and no HTML tag in attribute values. These assumptions were subsequently relaxed by ExAlg [33].

Interactive approaches include NoDoSe [9], interactive Stalker [451], Lixto [56, 57, 262], iFlex [519], and [322]. The Poly system described in this chapter is based on the system of [322]. The systems W4F [503] and XWRAP [394] also employ user interaction.

An ontology-based approach to wrapper construction, inspired by conceptual modeling, is described in [210], and a visual approach is described in [103]. Discovering record boundaries in multiple-record Web pages is described in [211, 393]. Once discovered, free text inside each record can be segmented into separate data fields using the technique in [94].

Several works have considered wrapper construction for multiple sites or at the Web scale. Chang et al. [139] collaboratively wrap multiple data sources all at once. Dalvi et al. [158] and Gulhane et al. [274] consider how to extract structured data at the Web scale. Cafarella et al. [102] consider how to extract millions of HTML tables on the Web.

Once built, it is also critical to maintain a wrapper over time, as the underlying data source evolves. Several works attempt to detect when a wrapper is broken (i.e., extracting incorrect data) [359, 375, 419] and to repair broken wrappers [375, 429, 441, 493]. Dalvi et al. [159] and Aditya et al. [474] consider a complementary problem: building robust wrappers that are unlikely to break as the underlying data sources evolve.