Understanding Web Query Interfaces: Best-Effort Parsing with Hidden Syntax*

Zhen Zhang, Bin He, Kevin Chen-Chuan Chang
Computer Science Department
University of Illinois at Urbana-Champaign
{zhang2, binhe}@uiuc.edu, kcchang@cs.uiuc.edu

ABSTRACT

Recently, the Web has been rapidly "deepened" by many searchable databases online, where data are hidden behind query forms. For modelling and integrating Web databases, the very first challenge is to understand what a query interface says– or what *query capabilities* a source supports. Such automatic extraction of interface semantics is challenging, as query forms are created autonomously. Our approach builds on the observation that, across myriad sources, query forms seem to reveal some "concerted structure," by sharing common building blocks. Toward this insight, we hypothesize the existence of a *hidden syntax* that guides the creation of query interfaces, albeit from different sources. This hypothesis effectively transforms query interfaces into a *visual language* with a *non-prescribed* grammar– and, thus, their semantic understanding a *parsing* problem. Such a paradigm enables principled solutions for both declaratively representing common patterns, by a derived grammar, and systematically interpreting query forms, by a global parsing mechanism. To realize this paradigm, we must address the challenges of a hypothetical syntax– that it is to be derived, and that it is secondary to the input. At the heart of our form extractor, we thus develop a *2P grammar* and a *best-effort parser*, which together realize a parsing mechanism for a hypothetical syntax. Our experiments show the promise of this approach– it achieves above 85% accuracy for extracting query conditions across random sources.

1. INTRODUCTION

In the recent years, the Web has been rapidly "deepened" by many searchable databases online. Unlike the surface Web providing link-based navigation, these "deep Web" sources support query-based access– Data are thus hidden behind their *query interfaces*. With the myriad databases online, at the order of 10^5 [3, 4], the deep Web has clearly rendered large-scale integration a real necessity and a real challenge.

Guarding data behind them, such query interfaces are the "entrance" to the deep Web. These interfaces, or HTML *query forms*, express *query conditions* for accessing objects from databases behind. Each condition, in general, specifies an *attribute*, one or more supported *operators* (or modifiers), and a *domain* of allowed values. A condition is thus a three-tuple [attribute; operators; domain], *e.g.*, c_{author} = [author; {"first name...", "start...", "exact name"}; text] in interface Q_{am} (Figure 3(a)). Users can then use the condition to formulate a specific constraint (*e.g.*, [author = "tom clancy"] by selecting an operator (*e.g.*, "exact name") and filling in a value (*e.g.*, "tom clancy").

For modelling and integrating Web databases, the very first step is to "understand" what a query interface says– *i.e.*, what *query capabilities* a source supports through its interface, in terms of specifiable conditions. For instance, amazon.com (Figure 3(a)) supports a set of five conditions (on author, title, ..., publisher). Such query conditions establish the *semantic model* underlying the Web query interface. This paper studies this "form understanding" problem: to extract such form semantics.

Automatic capability extraction is critical for large-scale integration. Any mediation task generally relies on such *source descriptions* that characterize sources. Such descriptions, largely constructed by hands today, have been identified as a major obstacle to scale up integration scenarios [14]. For the massive and ever-changing sources on the Web, automatic capability extraction is essential for many tasks: *e.g.*, to model Web databases by their interfaces, to classify or cluster query interfaces [12], to match query interfaces [11] or to build unified query interfaces.

Such form understanding essentially requires both *grouping* elements hierarchically and *tagging* their semantic roles: First, grouping associates semantically related HTML elements into one construct. For instance, c_{author} in Q_{am} is a group of 8 elements: a *text* "author", a *textbox*, three *radio buttons* and their associated *text*'s. Such grouping is hierarchical with nested subgroups (*e.g.*, each radio button is first associated with the text to its right, before further grouping). Second, tagging assigns the semantic roles to each element (*e.g.*, in c_{author}, "author" has the role of an *attribute*, and the textbox an *input domain*.)

Such extraction is challenging, since query forms are created autonomously. This task seems to be rather "heuristic" in nature: with no clear criteria but only a few fuzzy *heuristics*– as well as *exceptions*. *First*, grouping is hard, because a condition is generally n-ary, with various numbers of el-

*This material is based upon work partially supported by NSF Grants IIS-0133199 and IIS-0313260. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

SIGMOD 2004 June 13-18, 2004, Paris, France.

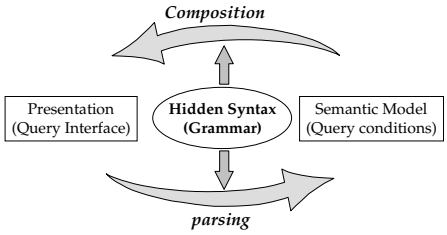


Figure 1: Hidden-syntax hypothesis.

ements nested in different ways. ([*heuristics*]: Pair closest elements by spatial proximity. [*exception*]: Grouping is often not pairwise.) *Second*, tagging is also hard– There is no semantic labelling in HTML forms. ([*heuristics*]: A text element closest to a textbox field is its attribute. [*exception*]: Such an element can instead be an operator of this or next field.) *Finally*, with various form designs, their extraction can be inherently confusing – The infamous Florida "butterfly" ballots in US Election 2000 indicates that ill-designed "forms" can be difficult, even for human voters, to simply associate candidates with their punch holes. This incident in fact generated discussions[1] on Web-form designs.

Our approach builds on the observation that, across myriad sources, query forms seem to reveal some "concerted structure": They appear to be "modularly" constructed upon a small set of building blocks. Those *condition patterns* present query conditions in certain visual arrangement– Figure 3(c) shows several examples. For instance, pattern 1 represents a common format for conditions of the form [attribute; {contains}; text], by arranging attribute to the *left* of a textbox. Such conditions represent keyword search (by an implicit contains operator) on a textual attribute (*e.g.*, author).

To capture this insight, we hypothesize the existence of a *hidden syntax* behind Web query interfaces, across different sources. As Figure 1 illustrates, we rationalize the phenomenon of concerted-structure by asserting query-form creation as guided by such a hypothetical syntax, which connects *semantics* to *presentations*. This hypothesis effectively transforms the problem into a new paradigm: We view query interfaces as a *visual language*, whose composition conforms to a hidden, *i.e.*, *non-prescribed*, grammar. Their semantic understanding, as the inverse, is thus a *parsing* problem.

This "language" paradigm enables principled solutions– to a problem that at first appears heuristic in nature– with the essential notions of a grammar and a parser:

- For *pattern specification*, the *grammar* provides a *declarative* mechanism. Such patterns (*e.g.*, Figure 3(c)) are simply declared by *productions* (*i.e.*, grammar rules) that encode their visual characteristics.
- For *pattern recognition*, the *parser* provides a *global* mechanism for systematically constructing a *parse tree* as a coherent interpretation of the *entire* query interface. Such a parse naturally groups elements (nested in subtrees) and tags their semantic roles (by grammar *symbols*), thus achieving form understanding.

However, the hypothesis itself entails challenges in its realization: First, as the hidden syntax is hypothetical, we must derive a grammar in its place– What should such a *derived grammar* encode, for capturing this "hidden syntax"? Further, any derived grammar will be inherently *incomplete*

[1] *e.g.*, www.larrysworld.com/articles/ups_ballot.htm.

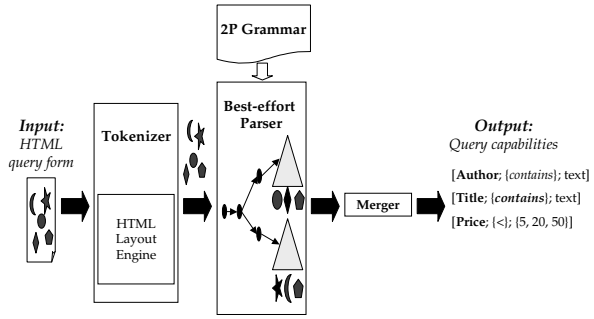


Figure 2: Form extractor for Web query interfaces.

(with uncaptured patterns) and *ambiguous* (with conflicting patterns). Thus, a derived grammar is only secondary to any input – Unlike traditional parsing, our parser cannot reject any input query form, even if not fully parsed, as "illegal." To work with a hypothetical syntax, what should be the semantics and machinery for such a *soft parser*?

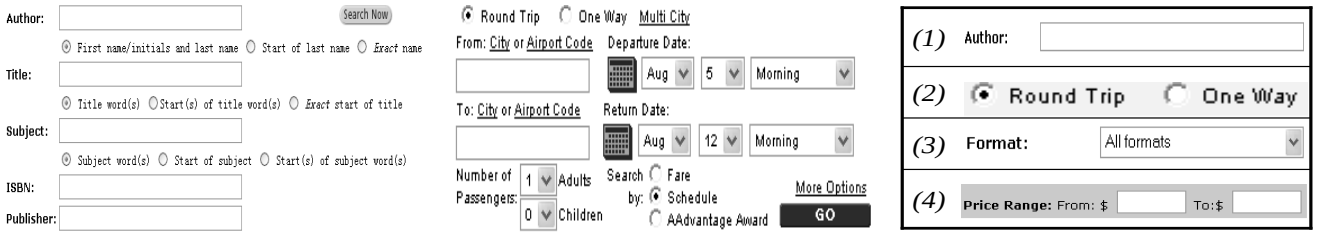
For understanding Web query interfaces, this paper presents our *form extractor*, which essentially realizes the hidden-syntax hypothesis. As Figure 2 shows, given an HTML query form, the form extractor tokenizes the page, parses the tokens, and then merges potentially multiple parse trees, to finally generate the query capabilities. At its heart, we develop a *2P grammar* and a *best-effort parser*, which together realize a non-traditional parsing mechanism for a hypothetical syntax. Our experiments show the promise of this approach– it achieves above 85% accuracy for extracting query conditions across random sources. In summary, this paper makes the following contributions:

- **Hidden-syntax hypothesis**: Motivated by the concerted structure, our key hypothesis transforms the seemingly heuristic problem into a novel paradigm (Section 3).
- **2P grammar**: For capturing the hidden syntax, we develop a grammar mechanism that encodes not only patterns but also their precedence (Section 4).
- **Best-effort parser**: Coping with the derived grammar that is inherently ambiguous and incomplete, we develop a best-effort parsing algorithm (Section 5).
- **Experimental evaluation**: We extensively evaluate the effectiveness of the framework (Section 6).

2. RELATED WORK

While understanding query interfaces automatically is an important problem for many applications (*e.g.*, integration of Web databases), the efforts invested are very limited. In this section, we relate our work to others from three aspects: works with similar basis on structure regularity, works addressing the same problem of interface understanding, and works developing techniques in visual language parsing.

First, many works that address extracting the underlying structure of Web pages rely on the regularities of such structures, among which *wrapper induction* [2, 8, 17] has been extensively studied. While those works deal with result pages as responses to a submitted query, our work focus on query interfaces - the entrance to the database. The work by Crescenzi [2] and Arasu [17] are closest to ours in that we both view Web pages as generated from a grammar. However, in their settings, the existence of a grammar is not an assumption but reality, because the collection of Web pages studied are homogeneously generated from the same



(a) Query interface Q_{am}: amazon.com. **(b)** Query interface Q_{aa}: aa.com. **(c)** Examples of condition patterns.

Figure 3: Query interfaces examples.

background template. In our problem, the syntax is only hypothetical based on the observations of heterogeneous query interfaces. Further, their works are essentially addressing a grammar derivation problem, while ours a parsing problem with the hypothesis of a hidden syntax.

Second, the problem of understanding query interfaces is mentioned, but not as focus, in several works [6, 13, 21]. Quite a few works that rely on automatic form filling [6, 9, 18] either only deal with simple keyword query forms or make use of only selection lists for easy interaction. In particular, reference [21] proposed to use simple heuristics such as proximity and alignment to associate pairwise elements and texts in the forms, while we explore a parsing paradigm with a hidden syntax to derive a global interpretation for the input, which can generally capture not only complex compositions but also sophisticated features other than proximity or alignment.

Third, by abstracting query interfaces as a visual language, we are able to leverage the formalisms [10, 19, 22] and techniques [10, 15, 16] developed in visual languages, meanwhile still develop our own strategy to address specific challenges in our problem, as we will see in the following sections.

3. TOWARDS PARSING PARADIGM

This section develops our key insight of the hidden-syntax hypothesis and the approach it enables. We first report our motivating observations (Section 3.1), which leads to the key hypothesis (Section 3.2). As the hypothesis brings forward, we must also address the challenges entailed by a hypothetical syntax (Section 3.3). Overall, our solutions develop a language-parsing framework for building an automatic form extractor, which Section 3.4 gives an overview.

3.1 Observations: Concerted Structure

As query interfaces are created autonomously, automatic extraction of form semantics is clearly challenging. Is such "form understanding" even possible? As Section 1 hinted, there seems to be some common "patterns" emerging from heterogeneous query forms. This impression suggests that Web forms are not entirely chaotic (which, if so, would render automatic extraction unlikely). Consider these patterns as the building blocks, or *vocabulary*, for constructing query forms. We ask: What is this vocabulary? How large is it?

To answer these puzzles, we performed an informal survey: Using search engines (e.g., google.com) and Web directories (*e.g.*, invisibleweb.com), we collected a total of 150 sources, which we call the Basic dataset, with 50 in each of Books, Automobiles, and Airfares domains. (Many sources are familiar ones, *e.g.*, amazon.com and aa.com in Figure 3.) We chose these domains as they are schematically dissimilar and semantically unrelated– and thus constitute a diverse "sample" of Web sources.

Our survey found that the query interfaces reveal some concerted structure: there are only 25 condition patterns overall– which is surprisingly small as a vocabulary for online queries. Figure 4(a) summarizes the occurrences of 21 "more-than-once" patterns. The figure marks (x, y) with a "+" if pattern y occurs in source x. As more sources are seen (along the x-axis), the growth (along y) of the vocabulary slows down and thus the curve flattens rapidly. Further, we observe that the convergence generally spans across different domains (e.g., Automobiles and Airfares are mostly reusing the patterns from Books), which indicates that most condition patterns are quite generic and not domain specific.

Second, we observe that the distribution is extremely non-uniform: Figure 4(b) ranks these 21 patterns according to their frequencies, for each domain and overall. We observe a characteristic Zipf-distribution, which means that a small set of top-ranked patterns is very frequently used.

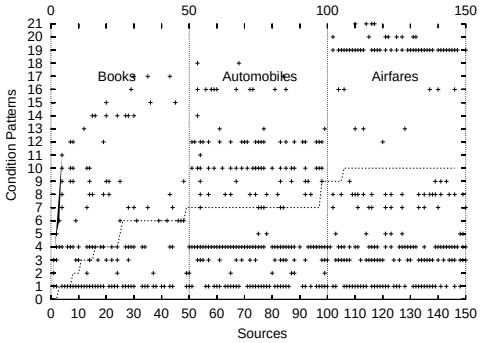
As implications, first, the small and converging vocabulary, which occurs across autonomous sources and even across diverse domains, indicates that there are *conventions* (or "design patterns") emerging among Web query forms. While each form is different, together they seem to share a relatively small set of vocabulary. Second, the non-uniform distribution of patterns suggests that, to leverage such conventions, even if we can not exhaustively cover all patterns, a few frequent ones will likely pay off significantly.

3.2 Hypothesis: Hidden Syntax

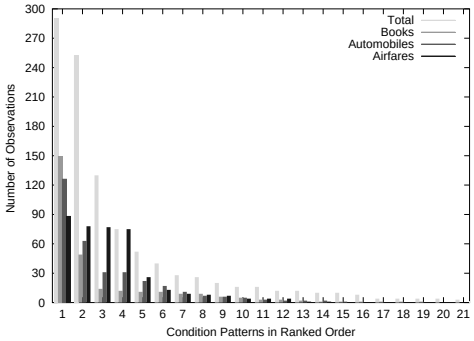
The concerted-structure hints that form understanding can be promising, by leveraging presentation conventions. Intuitively, given a query form, we may build our understanding of it by decomposing into some known patterns, each of which we have seen before– Thus, we assemble an interpretation of an interface unseen before, by the patterns we know of. This "divide-and-conquer" approach seems promising, since we have observed a small vocabulary of such patterns shared across diverse query forms.

While this approach is intuitively appealing, to realize it, what would be a principled computation paradigm? The task seems to be *heuristic* in nature– To use these layout patterns (as previous works also explored; Section 2), it is tempting to "simply" code up each pattern as a rule-of-thumb, *e.g.*, the pairwise-proximity grouping heuristic (Section 1). To *specify* these patterns, such procedural description will involve convoluted code, lacking both generality and extensibility. Further, to *recognize* these patterns, it is far from clear, beyond individual heuristics, how they together form a coherent interpretation of the query form.

Toward our insight, we hypothesize the existence of a *hidden syntax* behind Web query interfaces, across different sources. This hypothesis rationalizes the observed con-



(a) Vocabulary growth over sources.



(b) Frequencies over ranks.

Figure 4: Query vocabulary: condition patterns as building blocks for query interfaces.

certed structure. As Figure 1 illustrates, we view query-form creation as guided by such a hypothetical syntax, which connects *semantics* (*i.e.*, query conditions) to *presentations* (*i.e.*, query forms). Such a hidden syntax represents the presentation conventions across Web forms. Unlike traditional string languages (*e.g.*, programming languages), this syntax uses visual effects to express the embedded semantics (*e.g.*, pattern 1 in Figure 3(c) arranges the attribute to be *left-adjacent* and *bottom-aligned* to the input field).

This hypothesis brings forward a new paradigm: We now view query interfaces as a *formal* language, and in particular, a *visual language*, whose composition conforms to a hidden, *i.e.*, *non-prescribed*, grammar. Their semantic understanding, as the inverse, is thus a *parsing* problem.

This "language" paradigm further enables a principled algorithmic framework for form understanding– a task that appears inherently heuristic at first. By the hidden-syntax hypothesis, we can now resort to a formal framework for languages: The dual notions of a grammar and a parser together provide a systematic framework for both specifying and recognizing common patterns.

For *pattern specification*, the *grammar* provides a *declarative* mechanism. Such patterns (*e.g.*, Figure 3(c)) are simply declared by *productions* (*i.e.*, grammar rules) that encode their visual characteristics. The specification of patterns is thus declarative, fully separated from and independent of how they are recognized individually and assembled globally– by the parser. It is *general*: By incorporating arbitrary spatial relations (instead of, say, only proximity), we can describe complex visual patterns. By building productions upon productions, we can describe patterns of different "orders." It is also *extensible*: We simply augment the grammar to add new patterns, leaving parsing untouched.

For *pattern recognition*, the *parser* provides a *global* mechanism for systematically constructing a *parse tree* as a coherent interpretation of the *entire* query interface. Such a parse naturally structures elements in nested subtrees, thus satisfying the grouping requirement (Section 1). Further, it assigns grammatical alphabet symbols (terminals and nonterminals) to each construct, thus satisfying the tagging requirement. Finally, we stress that, such parsing leverages not only individual patterns but also their coherent assembly into an entire query form, thus resolving local conflicts by a global context. Parsing thus systematically realizes the intuitive "divide-and-conquer" approach.

3.3 Challenges: Hypothetical Syntax

As the hidden syntax is simply hypothetical, while it enables a new paradigm, we must address the challenges it entails: As this hypothetical nature implies, our grammar is *non-prescribed*: Instead of being prescribed before query forms are created, it is simply derived from whatever conventions naturally emerging. Further, our grammar is thus *secondary* to any language instance: Instead of dictating form creation, it relies on the language's natural convergence to derive any convention. Our challenges are thus two-folded:

First, for capturing this hypothetical syntax, what should such a *derived grammar* encode? Such a syntax represents "conventions" used for Web form presentation. What types of conventions are necessary to enable parsing? While we want ideally to capture *all* patterns across many forms, unlike in a carefully-orchestrated grammar, these patterns may not be mutually "compatible." We must thus rethink the right mechanism for such a derived grammar, to capture necessary conventions for enabling parsing.

Second, for working with a hypothetical syntax, what should be the semantics and machinery for a *soft parser* ? A derived grammar will be inherently *incomplete* (with uncaptured patterns) and *ambiguous* (with conflicting patterns). Thus, such a grammar is only secondary to any input: Unlike traditional parsing, our parser cannot reject any input query form, even if not fully parsed, as "illegal." That is, our parser is no longer a language "police" for checking and enforcing grammar rules. It must now be a "soft" parser that accepts any input. We must thus rethink the right semantics for such a soft parser, and further, its realization.

3.4 Solutions: 2P Grammar & Best-effort Parser

Our solutions build upon the traditional language framework, dealing with the specific challenges outlined above. First, as a derived grammar for capturing the hypothetical syntax, the 2P grammar encodes not only "patterns" but also their "precedence." Second, as a soft-parser directed by a hypothetical syntax, when a single perfect parse does not exist, the best-effort parser resolves ambiguities as much as possible and constructs parse trees as large as possible.

2P Grammar: To capture the hidden syntax, we develop our grammar to encode two complementary types of presentation conventions. On one hand, we must ideally capture all conventional *patterns*. On the other hand, however, by capturing many patterns, some will conflict, and we must also capture their conventional *precedence* (or "priorities").

Our *2P grammar* mechanism, as Definition 1 specifies, encodes both conventions by *productions* and *preferences* respectively (and thus the name). That is, it captures knowledge for both pattern construction (by productions) and ambiguity resolution (by preferences), as Section 4 will discuss.

Definition 1 (2P Grammar): A *2P grammar* is a 5-tuple $\langle \Sigma, N, s, P_d, P_f \rangle$: Σ is a set of *terminal* symbols. N is a set of *nonterminal* symbols. $s \in N$ is a *start* symbol. P_d is a set of *production* rules. P_f is a set of *preference* rules. ∎

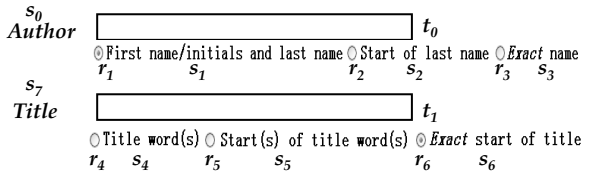
In our framework, we use this 2P grammar mechanism to express the hypothetical syntax. Such a grammar is to be derived, from analyzing and abstracting common patterns. In our implementation, we use a "global" grammar derived from query interfaces in the Basic dataset– which thus essentially captures the observations that Section 3.1 reported. We discuss this grammar and the related issues in Section 6.

Best-effort Parser: To work with a hypothetical syntax, we develop our parser to perform "best-effort." As explained earlier, a derived grammar will be inherently ambiguous and incomplete. We thus need a "soft parsing" semantic– The parser will assemble parse trees that may be *multiple* (because of ambiguities) and *partial* (because of incompleteness), instead of insisting on a single perfect parse. First, it will prune ambiguities, as much (and as early) as possible, by employing preferences (as in the 2P grammar). Second, it will recognize the structure (by applying productions) of the input form, as much as possible, by maximizing partial results. Our parser pursues this new "philosophy" of *best-effort* parsing, which Section 5 will present.

Overall Framework– The Form Extractor: Upon the core components of the 2P grammar and the best-effort parser, we build our *form extractor* in a language-parsing framework, as Figure 2 shows. Given an input HTML query form, the form extractor outputs its semantic model (or the query capabilities) of the form. At the heart, the best-effort parser works with a derived 2P-grammar to construct multiple and partial parse trees. As preprocessing, the *tokenizer* prepares the input to the core parser, by converting the input HTML form into a set of basic *tokens*, which are the atomic units in the visual grammatical composition. As post-processing, the *merger* integrates the output of the parser to generate the final semantic model.

At the front-end, the tokenizer converts an HTML query form (in a Web page) into a set of *tokens*, each representing an atomic visual element on the form. Note that these tokens are instances of the terminals Σ as the 2P grammar defines (Definition 1). Each token thus has a terminal *type* and some *attributes* recording properties necessary for parsing. For instance, given the HTML fragment (as part of interface Q_{am}), as Figure 5 shows, the tokenizer extracts a set $\mathcal{T}$ of 16 tokens. In particular, token s_0 is a `text` terminal, with attributes $sval =$ `"Author"` (its string value) and $pos = (10, 40, 10, 20)$ (its bounding-box coordinates). Although different terminals have different attributes, this *pos* attribute is universal, as our grammar captures two dimensional layout. Such a tokenizer thus essentially builds on a *layout engine* for rendering HTML into its visual presentation. In particular, our tokenizer uses the HTML DOM API (available in browsers, *e.g.*, Internet Explorer), which provides access to HTML tags and their positions.

At the back-end, the merger combines the multiple partial parse trees that the parser outputs, to compile the semantic model and report potential errors (if any). Since our parser is rather generic, this step applies application (*i.e.*, query form) specific processing. First, since our goal is to identify all the query conditions, the merger combines multiple parse trees by taking the union of their extracted conditions. As each parse covers different parts of the form, this union



ID	Type	<Attribute, value>	ID	Type	<Attribute, value>
s_0	text	**<sval, *Author*>** **<pos, *(10,40, 10,20)*>**	t_0	textbox	**<name, *query-0*>** **<pos, *(50,100,10,20)*>**
r_1	radio	**<name, *field-0*>** **<pos, *(50,55,22,32)*>**	s_1	text	**<sval, *first name/initial and last name*>** **<pos, *(55,85, 22,32)*>**
...	...			...	...

Figure 5: Tokens $\mathcal{T}$ in fragment of interface Q_{am}.

enhances the coverage of the final model constructed. For example, given a fragment of interface Q_{aa}, as Figure 14 shows, the parser will generate three partial parses (trees 2, 3, 4 in the figure) (Section 5.3). Their union covers the entire interface and generates all the conditions.

The merger also reports errors, which will be useful for further error handling by the "client" of the form extractor. It reports two types of errors: First, a *conflict* occurs if the same token is used by different conditions. In Figure 14, tree 2 associates the number selection list with `number of passengers`, while tree 3 with `adults`– and thus they conflict by competing for the number selection. (In this case, tree 3 is the correct association.) Second, a *missing element* is a token not covered by any parse tree. The merger reports both types of errors for further client-side handling.

In summary, for automatic form understanding, we make the key hypothesis of hidden syntax and thus pursue a parsing framework for the form extractor. As the focus of this paper, we will now concentrate on the dual cores of this framework– the 2P grammar and the best-effort parser.

4. 2P GRAMMAR

As the key component in the parsing framework, 2P grammar captures presentation conventions of Web interfaces. Specifically, the 2P grammar *declaratively* and *comprehensively* specifies both condition patterns and their precedence, as a principled way to express a derived syntax and to resolve potential ambiguities. In particular, *productions* formally specify common condition patterns and *preferences* their relative precedence, as Section 4.1 and Section 4.2 will present respectively.

4.1 Productions: Capturing Patterns

Since the condition patterns establish a small set of building blocks for Web interfaces, we need to explore appropriate presentational characteristics to capture those condition patterns as productions. In particular, in query interfaces, visual effects such as *topology* (e.g., alignment, adjacency) and *proximity* (e.g., closeness) are frequently used for expressing semantically related components and thus are the candidates to be captured by productions. We find that some features such as proximity work well for simple interfaces; however, it is hard to be extended to complex interfaces and can often result in incorrect interpretations. On the other hand, we observe that the topology features such as alignment and adjacency (*e.g.*, *left*, *above*) accurately indicate the semantic relationships among the components in query interfaces. Therefore, in the 2P grammar, we exten-

#	Production Rules	Visual Patterns
P1	**QI**← HQI | Above(QI, HQI)	
P2	**HQI**← CP | Left(HQI, CP)	
P3	**CP**← TextVal | TextOp | EnumRB	
P4	**TextVal**← Left(Attr,Val) | Above(Attr,Val) | Below(Attr, Val)	Attr Val
P5	**TextOp**← Left(Attr, Val) ∧ Below(Op, Val)	Attr Val Op
P6	**Op**← RBList	
P7	**EnumRB**← RBList	RBList
P8	**RBList** ← RBU | Left(RBList RBU)	RBList RBU
P9	**RBU**← Left(radiobutton, text)	radiobutton text
P10	**Attr**←text	
P11	**Val**←textbox	

Figure 6: Productions of the 2P grammar.

sively explore such topological information, in the productions, to capture condition patterns.

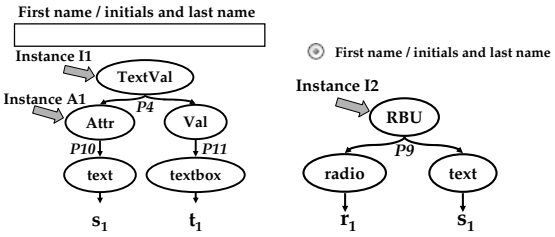
Many *two-dimensional grammars* have been proposed in visual languages to realize such specifications of visual patterns, *e.g.*, relational grammar [22], constraint multiset grammar [19], positional grammar [7]. Our 2P grammar (without considering the preferences), is a special instance of *attributed multiset grammar* [10], where a set of spatial relations capturing topological information (*e.g.*, *left*, *right*) are used in productions.

The main extension of two dimensional grammars from string grammars (e.g., for programming languages) is to support general constraints. In two dimensional grammars, productions need to capture spatial relations, which essentially are constraints to be verified on the constructs. For example, consider production $P5$ in Figure 6. To capture the pattern TextOp (used by author in interface Q_{am}), we specify that Attr is *left* to Val and Op *below* to Val. (Note that, in the 2P Grammar, *adjacency* is implied in all spatial relations and thus omitted in the constraint names). In contrast, productions in string grammars only use one constraint, the *sequentiality*, among components.

As a consequence, such extension leads to adaptations in other aspects of the productions. Specifically, to support the general constraints, each symbol has a set of attributes (*e.g.*, *pos* of Attr, Op and Val), which stores the information used in constraints evaluation (*e.g.*, *left*, *below*). Further, each production has a *constructor*, which defines how to instantiate an *instance* of the head symbol from the components. For example, after applying the production $P5$ to generate a new TextOp instance i, the constructor computes i's position from its components. Formally, we define the production as:

Definition 2 (Production): A *production* P in a 2P grammar $G = \langle \Sigma, N, s, P_d, P_f \rangle$ is a four-tuple $\langle H,\ M,\ C,\ F \rangle$: *Head* $H \in N$ is a nonterminal symbol. *Components* $M \subseteq \Sigma \cup N$ is a multiset of symbols. *Constraint* C is a boolean expression defined on M. *Constructor* F is a function defined on M, returning an instance of H. ∎

Example 1 (Grammar $\mathcal{G}$): We show a simple grammar $\mathcal{G}$ (without the component P_r) that we will use in later explanations. As Figure 6 shows, grammar $\mathcal{G}$ specifies 11 productions labelled from $P1$ to $P11$. Each production defines a nonterminal (*e.g.*, TextOp and EnumRB) as its head. The start symbol is QI and the terminal symbols are text, textbox and radiobutton. Note that, to simplify the illustration, we omit the production constructors in Figure 6.



(a) TextVal instance I1 **(b)** RBU instance I2

Figure 7: Two interpretations for text s_1.

Productions $P3$ to $P11$ capture three patterns (patterns 1 and 2 in Figure 3(c) in addition to TextOp introduced above). Productions $P1$ and $P2$ capture the *form pattern* by which condition patterns are arranged into query interfaces. In particular, we consider a query interface QI as composing of vertically aligned "rows" HQI, where each HQI further composes of horizontally aligned condition patterns CP. ∎

Our productions provide a quite general and extensible mechanism for describing patterns. First, it can express patterns of different "orders": Complex patterns are built upon simpler ones. For example, pattern TextOp is constructed from simpler patterns Attr, Op and Val, and in turn serves as the basis of higher order patterns such as QI. Second, it is very extensible to incorporate new patterns and new constraints, while leaving the parsing algorithm untouched. As we will discuss in Section 7, by changing the grammar, exactly the same parsing framework can be used for other applications.

4.2 Preferences: Capturing Precedence

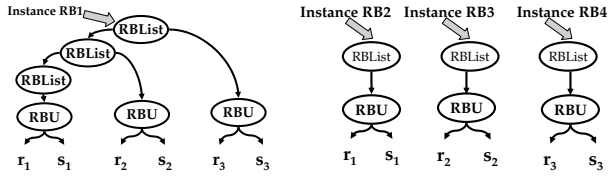
For derived grammars, precedence is essential for resolving the conflicts among patterns and thus an integral component of the 2P grammar. While our grammar intends to capture as many common (but non-prescribed) patterns as possible, those patterns may not be "compatible," which results in significant ambiguities (Section 4.2.1). To resolve those ambiguities, we explore a preference framework, which captures the conventional precedence among condition patterns (Section 4.2.2).

4.2.1 Inherent ambiguities

Ambiguity happens when there exist multiple interpretations for the same token, and therefore these interpretations *conflict* on such a token. Example 2 shows an example of a conflicting situation.

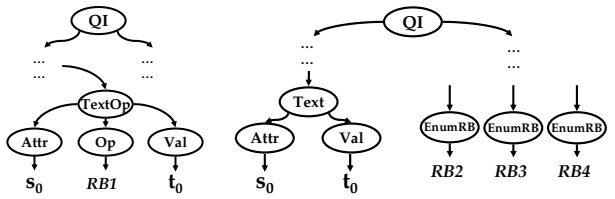
Example 2 (Ambiguity): To capture the condition pattern TextVal used by from condition in Q_{aa} and pattern RBU used in Q_{am}, we define productions $P4$ and $P9$ respectively. However, such generality brings ambiguities, allowing a token to be interpreted differently by different patterns. Consider the text token s_1 (i.e.,"first name/initial and last name") in Figure 5, pattern TextVal($P4$) and RBU($P9$) have different interpretations on s_1, as Figure 7 shows. In particular, TextVal interprets it as an Attr instance A1 in a TextVal instance I1 (Figure 7(a)). In contrast, RBU interprets it as the text of a RBU instance I2 (Figure 7(b)). Since conflicting on s_1, I1 and I2 cannot appear in the same parse tree. ∎

The existence of ambiguities may cause parsing inefficient and inaccurate. It is inefficient because of *local ambiguities*: The parser may generate "temporary instances" that will not appear in any complete parse tree. An ambiguity



(a) Derivation tree 1 **(b)** Derivation tree 2

Figure 8: Two interpretations for radio button list.



(a) Parse tree 1 **(b)** Parse tree 2

Figure 9: Two parse trees for query interface Q_1.

between two instances is *local* if at least one of them is a temporary instance. Consider Example 2, I1 is a temporary instance, since we cannot further derive a complete parse tree from I1. In contrast, we can derive complete parse trees from I2 (as Figure 9 shows two). Hence, such ambiguity is *local* because it can eventually be resolved at the end of parsing. As we will show in Section 5, the parsers used in visual language generally follow a bottom-up exhaustive approach which explores all possible interpretations. Therefore, the existence of local ambiguities makes parsing very inefficient due to the generation of many "temporary instances."

In contrast, *global* ambiguities make the parsing results inaccurate: The parsing may generate more parse trees than the semantically correct one. An ambiguity between two instances is *global* if they lead into different parse trees, and thus cannot be resolved even at the end of parsing.

Example 3 (Global ambiguity): To capture radio button lists of arbitrary length, production $P8$ is defined in a recursive way. As a result, a radio button list of length three can have four interpretations, depending on how they are grouped. Figure 8 shows such two - (a) as a single list or (b) as three individual lists with each of length one. The ambiguity between these two interpretations is global, because they eventually lead to two different parse trees, as Figure 9 shows. The first one takes the entire list as an operator of author, while the second takes each list (of length 1) as a condition pattern EnumRB. ∎

The effect of the inherent ambiguities is significant. For instance, the simple query interface in Figure 5 has one correct parse tree containing 42 instances (26 non-terminals and 16 terminals). However, applying the basic parsing approach (to be discussed in Section 5) that exhausts all possible interpretations by "brute-force," we get 25 parse trees and totally 773 instances (645 temporary instances and 128 non temporary ones). The reason to have such a significant amount of ambiguities is that conflicting instances may further participate in generating other instances, which in turn conflict. Such exponential aggregation makes ambiguity a significant problem in parsing.

4.2.2 Preference

To resolve the significant ambiguities among condition patterns, it is essential for a derived grammar to prioritize these patterns. The derived nature of our hidden syntax implies that such precedence comes from "hidden priority conventions" across patterns. In predefined grammars, the creation of a grammar is prior to that of the corresponding language, therefore how to resolve ambiguity is determined apriori. However, in derived grammars, the precedence itself is part of conventions to be derived from the language, and thus cannot be arbitrarily decided. In this paper, we propose to explore *preference* to encode such conventional precedence across patterns.

Example 4 (Preference): Consider the two conflicting instances, A1 and I2, in Example 2. Is there any convention that indicates which one is better? We observe that text and its preceding radio button are usually tightly bounded together, therefore when conflicting, I2 is more likely to have a higher priority than A1. Such convention of the precedence between patterns establishes our knowledge to resolve the ambiguities. In particular, we encode such precedence convention as a "preference" R_1: When an RBU instance and an Attr instance conflict on a text token, we arbitrate unconditionally the former as the winner.

In general, a convention may also carry a criterion for picking the winner: For example, for the ambiguity in Example 3, we observe that a row of radio buttons is usually used as a single longer list rather than separate shorter ones. Therefore, we define a preference R_2: When two RBList instances conflict, and if one subsumes the other, we pick the *longer* one as the winner. ∎

Specifically, each *preference* resolves a particular ambiguity between two types of conflicting instances by giving priority to one over the other. As Example 4 motivates, such a preference needs to specify the situation and the resolution. The situation indicates the type of *conflicting instances* (*e.g.*, RBList in preference R_2) and the *conflicting condition* (*e.g.*, subsume). The resolution describes the *criteria* that the winner instance should satisfy (*e.g.*, longer). Formally, we define the *preference* as:

Definition 3 (Preference): A *Preference* R in a 2P grammar $G = \langle \Sigma, N, s, P_d, P_f \rangle$ is a three-tuple $\langle I, U, W \rangle$:

- *Conflicting instances* $I = \langle v_1 : A, v_2 : B \rangle$, where $A, B \in T \cup \Sigma$, identifies the types of instances v_1 and v_2 respectively.
- *Conflicting condition* U: a boolean expression on v_1, v_2 that specifies a conflicting situation to be handled.
- *Winning criteria* W: a boolean expression on v_1, v_2 that specifies the criteria to pick v_1 as winner. ∎

There are several advantages of using preferences to resolve ambiguities. First, as the specification of precedence, preferences are simple and effective mechanism to encode the precedence conventions deterministically. Such simplicity helps the parser efficiently resolve ambiguities, compared with other approaches, *e.g.*, probabilistic model for enabling precedence. As we will discuss in Section 7, although our preferences only encode a "flat set" of precedence and thus may be nondeterministic when preferences themselves conflict, in practice we never have such a situation, because semantically meaningful preferences are consistent.

Second, as a mechanism for ambiguity resolution, preferences are particularly suitable to derived grammars. Traditional techniques, such as *lookahead prediction* [1] and *grammar transformation* [7], impose significant restrictions on

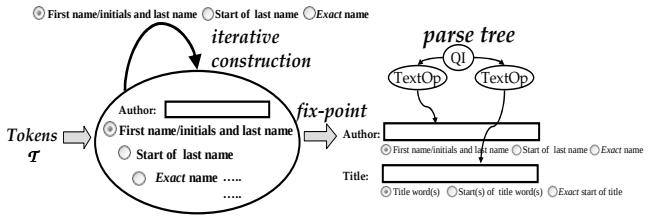


Figure 10: Fix-point parsing process.

the productions, which are difficult to meet for a derived grammar. In contrast, preferences can be specified independently from the productions without any specific constraint on the grammar. Further, preferences uniformly deal with both local and global ambiguities by favoring promising interpretations. Such uniform treatment is especially desirable for a derived grammar because its potential incompleteness blurs the distinction between local and global ambiguities.

5. BEST-EFFORT PARSER

With 2P grammar capturing the conventions of condition patterns and their precedences, this section presents a best-effort parsing algorithm that on one hand makes use of preferences to prune the wrong interpretations in a timely fashion, and on the other hand handles partial results to achieve maximum interpretations for the input. We start with outlining the best-effort parsing algorithm *2PParser* (Section 5.1), then zoom into two essential components: *just-in-time pruning* (Section 5.2) and *partial tree maximization* (Section 5.3).

5.1 Overview

With potential ambiguities and incompleteness, our best effort parser operates on a basic framework, the fix-point evaluation [10, 15, 23], that progressively and concurrently develops multiple parse trees. The essential idea is to continuously generate new instances by applying productions until reaching a fix-point when no new instance can be generated. For example, as Figure 10 conceptually shows, the parser starts from a set of tokens $\mathcal{T}$ (Figure 5), iteratively constructs new instances and finally outputs parse trees. In particular, by applying the production $P9$, we can generate an RBU instance from the text token s_1 and radiobutton r_1. Further, with the production $P8$, the RBUs in a row together generate an RBList instance. Continuing this process, we eventually reach the fix-point. A complete parse tree corresponds to a unique instance of the start symbol QI that covers all tokens, as Figure 10 conceptually shows one. However, due to the potential ambiguities and incompleteness, the parser may not derive any complete parse tree and only end up with multiple *partial* parse trees.

Upon this framework, we realize the "best-effort" philosophy by essentially: First, *just-in-time pruning* to prune the parse trees with wrong interpretations as much and as early as possible; Second, *partial tree maximization* to favor the parse trees that interpret an input as much as possible.

Figure 11 shows the best-effort parsing algorithm 2PParser. Corresponding to the above two components, the algorithm has two phases: first, parse construction with just-in-time pruning, and second, partial tree maximization at the end of parsing. To achieve just-in-time pruning, we schedule the symbols (by procedure BldSchdGraph) in a proper order so that false instances are pruned timely before further causing more ambiguities. According to the scheduled order, we instantiate the symbols one by one with a fix-point process (by instantiate). Preferences are enforced at the end of each iteration (by enforce) to detect and remove the false instances in this round. When an instance is invalidated, we need to erase its negative effect: false instances may participate in further instantiations and in turn generate more false parents. Procedure rollback is used to remove all those false ancestors to avoid further ambiguity aggregation. Finally, after parse construction phase, PRHandler chooses the maximum parse trees generated in the parse construction phase and outputs them.

```
Proc 2PParser(TS, G):
Input: Token set TS, grammar G
Output: Maximum partial trees res
begin:
  Y = BldSchldGraph(G)
  find a topological order of symbols in Y
  for each symbol A in order:
      I += instantiate(A)
      for each preference R involving A:
          F = enforce(R)
          for each invalidated instance i ∈ F
              Rollback(i)
  res = PRHandler()
end
Proc instantiate(A):
Input: Symbol A
Output: Instances of A inst
begin
  inst = ∅
  repeat
      for each production p with head being A:
          inst += apply(p)
  until no new instance added into inst
end
```

Figure 11: Parser for 2P grammar.

Inherently, visual language parsing is a "hard" problem. The complexity of the *membership* problem (*i.e.*, given grammar G, a sentence S, to determine whether $S \in L(G)$) for visual languages is NP-complete [20]. Therefore, the algorithm runs in exponential time with respect to the number of tokens. However, in practice, the use of preferences gives reasonably good performance. Our implementation[2] shows that, given a query interface of size about 25 (number of tokens), parsing takes about 1 second. Parsing 120 query interfaces with average size 22 takes less than 100 seconds. (The time measured here only includes the parsing time without tokenization and merger.)

In the next two sections, we zoom into the just-in-time pruning technique for ambiguity resolution and the partial tree maximization technique for partial results handling in more details.

5.2 Just-in-time Pruning

To prune false instances as much and as early as possible, we need to find a good timing for enforcing the preferences. Such timing would guarantee that any false instance is removed before participating in further instantiations, therefore no rollback is necessary. However, applying preferences whenever a new instance is generated in the basic fix-point algorithm cannot achieve so.

[2]The experiment is conducted with a Pentium IV 1.8GHz PC with 512MB RAM.

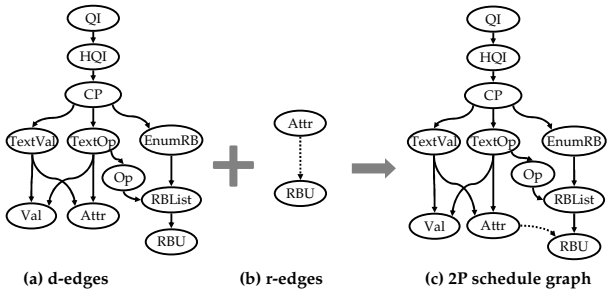


Figure 12: The 2P schedule graph for grammar $\mathcal{G}$.

Example 5: With the preference R_1 (defined in Example 4) which resolves the local ambiguity in Example 2, the Attr instance A1 should be removed by the RBU instance I2. What if A1 is generated at the very beginning of parsing, while I2 at the end? A1 will still instantiate instance I1 (and possibly others), and only be removed at the end of parsing (when I2 is generated). This "late pruning" makes the preference R_1 ineffective in controlling ambiguity aggregation. ∎

To address the problem, we want to generate the winner instance (*e.g.*, I2) before the loser (*e.g.*, A1) so that the loser can be detected and pruned whenever it is generated. Essentially, we want to schedule the instance generation in some desired order consistent with the preferences. As preferences are defined on symbols, to guarantee the order on particular instances, we enforce such an order on symbols so that the winner symbol produces all its instances before the loser does. Therefore, such symbol-by-symbol instantiation and winner-then-loser order can guarantee that the instances are produced in a desired order to ensure just-in-time pruning.

To realize the symbol-by-symbol instantiation, the symbols have to be processed in a "children-parent" direction defined by the productions. For example, consider symbol TextOp, as the production $P5$ defines, the symbols that contribute to the instantiation of TextOp are Attr, Op and Val. Before we can process TextOp, those children symbols must be processed first. Further, to realize the winner-then-loser order, the winner symbol (*e.g.*, RBU in Example 5) must be scheduled before the loser (*e.g.*, Attr).

To schedule the symbols by the above two orders, we build a *2P schedule graph*. The graph consists of the symbols as nodes and two types of edges - *d-edges* to capture the "children-parent" order defined by the productions and *r-edges* to capture the winner-then-loser order defined by the preferences.

Example 6 (2P schedule graph Y): Figure 12(c) shows the 2P schedule graph Y for the Grammar $\mathcal{G}$ (defined in Example 1), by merging *d-edges* (Figure 12(a)) and *r-edges* (Figure 12(b)). Y has a *d-edge* $A \rightarrow B$ if the grammar has a production with head symbol A and component symbols containing B (*i.e.*, A is a parent of B). Y has an *r-edge* $C \dashrightarrow D$ if the grammar has a preference D over C (*i.e.*, D is the winner and C is the loser). We omit the self-cycles because they do not affect the scheduling. (More precisely, we also omit the terminals, as they do not affect the schedulability in this example.) By merging these two types of edges, we get the 2P schedule graph Y, with solid edges denoting *d-edges* and dashed *r-edges*. ∎

By enforcing a topological order on symbol instantiations, this 2P schedule graph captures the two requirements needed for just-in-time pruning. If the graph is acyclic, any topological order achieves such a goal. For example, as our schedule graph Y (Example 6) is acyclic, we schedule RBU before Attr. Thus, instance I2 is generated before A1, which then is pruned promptly when generated. More precisely, as preferences are enforced at the end of each symbol instantiation to avoid repeated calls for every instance, ambiguities may aggregate during the instantiation of the symbol, which is minimal.

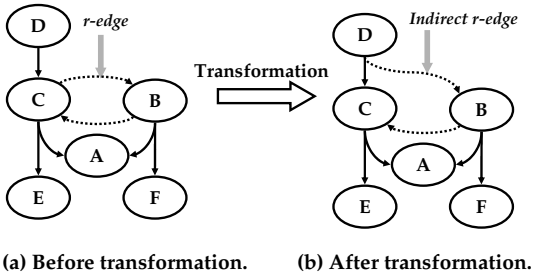


Figure 13: Transformation of an *r-edge*.

However, a 2P schedule graph may be cyclic. For example, suppose we have two symbols B and C, which share a construct A, as Figure 13 illustrates. Let a, b and c denote instances of A, B and C respectively. Instances b and c may potentially conflict on a. To resolve the ambiguity, suppose we define 2 preferences $R_{C \dashrightarrow B}$ and $R_{B \dashrightarrow C}$. The former specifies that we prefer b to c if, say, the inter-component distance of b is smaller than that of c. Similarly, the later states that we prefer c to b if such distance of c is smaller than that of b. In other words, the two preferences define that the winner instance, which can be either b or c, is the one with smaller inter-component distance. The two *r-edges* thus form a cycle $C \dashrightarrow B \dashrightarrow C$.

When the graph is cyclic, there is no such an order that satisfies both scheduling requirements (i.e., the symbol-by-symbol and winner-then-loser orders). Therefore, some edges have to be "relaxed." The *d-edge* must be enforced because symbol-by-symbol instantiation is the prerequisite of winner-then-loser order. (This implies that we require the *d-edges* form an acyclic graph.) In contrast, since the *r-edge* is to enhance efficiency by removing false instances early, it is mainly an "optimization." Even if we remove the *r-edge*, the negative effect is to introduce more false instances. Such negative effect can be erased by the rollback step, although it incurs some overhead in efficiency. In fact, removal of the *r-edge* is not the only way of relaxation. *Transformation*, as we will discuss next, may relax an *r-edge* without causing the negative effect of ambiguity aggregation.

Is it possible to relax an *r-edge* while still achieving just-in-time pruning? The answer is yes, because the requirement imposed by an *r-edge* is sufficient but not necessary to achieve our goal. Consider the *r-edge* $C \dashrightarrow B$, assume we remove this *r-edge* to break the cycle. We notice that if we can schedule B before C's parent symbol D, the winner B can still prevent any further false instances, which is D here, to be generated from the loser C.

Therefore, as Figure 13 illustrates, in case of cycle, we transform an original *r-edge* $C \dashrightarrow B$ to a set of *indirect r-edges* by, for each C's parent D (defined by a *d-edge*), adding an indirect *r-edge* $D \dashrightarrow B$. By doing so, the cycle in Figure 13(a) is broken. We thus have a scheduling, *e.g.*, E, F, A, C, B, D, that achieves just-in-time pruning. In this schedule, since C is scheduled before B, any false instance of B is pruned promptly at generation. On the other hand, although C is scheduled before B, the false instances

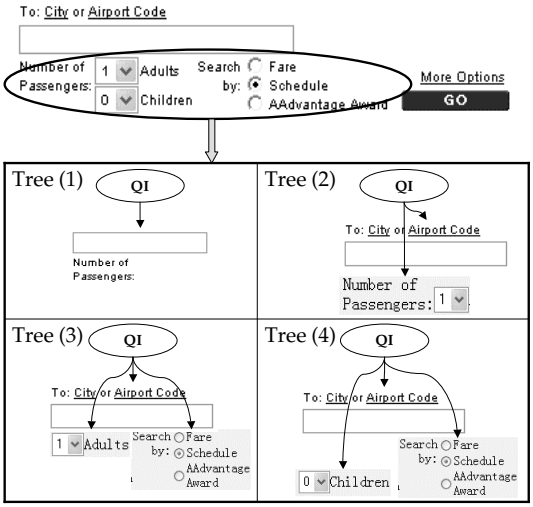


Figure 14: Partial trees for interface Q_{aa} fragment.

of C are removed when the instances of the winner B are generated, which is still before any instance of C's parent, D, is processed. Therefore, by transformation, we achieve just-in-time pruning. Although such transformation cannot guarantee to break all cycles in a 2P schedule graph, in practice, it is very effective to handle all the cycles in our 2P grammar.

If transformations still cannot break cycles, we have to remove *r-edges* and use rollback to offset the negative effects caused by late pruning. As cycles are rare situations, we employ a greedy algorithm in building 2P schedule graph to avoid cycles at first place: we add *r-edges* one by one, and if an *r-edge* causes cycle even after transformation, we simply remove it. When an *r-edge* is removed, a false instance may have chance to generate its parents. We then use *rollback* to erase those false ancestors caused by late pruning.

An interesting issue, as we will discuss in Section 7, is the problem of preferences *consistency*. The preferences are not consistent if given a set of tokens as input, different orders of applying the preferences result in different derivation results. In such case, which order is the right one is not defined. The algorithm outlined above assumes the consistency of preferences, and therefore generates a unique result.

5.3 Partial Tree Maximization

While just-in-time pruning addresses the inherent ambiguities of the grammar, in this section we briefly discuss how to handle the partial parse trees. The parsing algorithm generates partial parse trees when the grammar is incomplete to interpret the entire query interface.

Specifically, *partial parse trees* are the derivation trees that cover a subset of tokens and can not be expanded further. For instance, when a query interface contains new condition patterns not covered by the 2P grammar, the parse construction will stop at those partial trees, since not being able to further assemble more tokens. For example, consider the query interface in Figure 14, which is a variation from the interface Q_{aa}. Grammar $\mathcal{G}$ does not completely capture the form patterns of that interface: The lower part is arranged "column by column" instead of "row by row." Therefore, the parse construction generates only partial parses, as Figure 14 shows four of them.

To maximize the understanding of query interfaces, our parser favors the *maximum* partial trees that interpret as many tokens as possible. In particular, we use *maximum subsumption* to choose parse trees that assemble a maximum set of tokens not subsumed by any other parse. For example, Tree 1 in Figure 14 is not maximum because the tokens covered by Tree 1 is subsumed by those of Tree 2. The other three, although overlapping, do not subsume each other. (It is straightforward to see that a complete parse tree is a special case of maximum partial tree.) In addition to maximizing the interpretations, such maximum parse trees also potentially achieve better interpretations, since they are looking at larger context compared with the non-maximum ones.

6. EXPERIMENTS

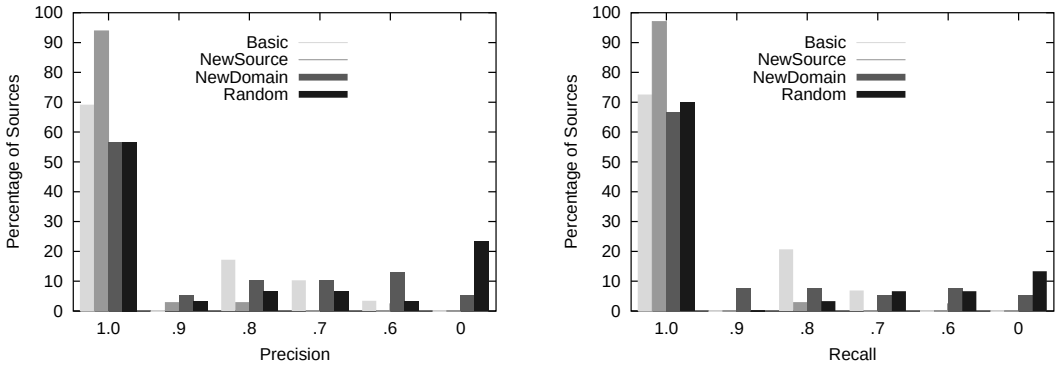
To evaluate the performance, we test our approach extensively on four datasets. Three of them are mainly selected from the TEL-8 dataset of the UIUC Web Integration Repository [5] for different purposes (as will be discussed below). The TEL-8 dataset contains about 500 deep Web sources across 8 domains. It is manually collected using Web directories (e.g., invisibleweb.com, brightplanet.com) and search engines (e.g., google.com). The last dataset we use in the experiment is collected by randomly sampling the invisible-web.net.

- Basic dataset, which we used in our motivating survey (as Section 3 reported), contains 150 sources in three domains (i.e., Airfares, Automobiles, and Books), mainly from the TEL-8 dataset.
- NewSource dataset contains 10 extra query interfaces from each of the above three domains, with a total of 30 sources.
- NewDomain dataset contains query interfaces from six different domains other than the above three (five from the TEL-8 dataset and a new RealEstates domain). Each domain has about 7 sources with a total of 42 sources.
- Random dataset contains 30 query interfaces that we randomly sampled from the Web. In particular, we choose invisible-web.net as our pool of sampling, which contains about 1,000 manually complied deep Web sources. As all the sources are sequentially numbered in the directory, we can easily draw random samples using the source IDs. The sampled sources cover 16 out of the 18 top level domains in the directory.

Our study intends to evaluate whether a single global grammar can be used for arbitrary domains. The grammar used in the experiment is derived from the Basic dataset. We manually observe the 150 query interfaces in the dataset, and summarize 21 most commonly used patterns. The derived grammar has 82 productions with 39 nonterminals and 16 terminals. The grammar is available online, as we will discuss later.

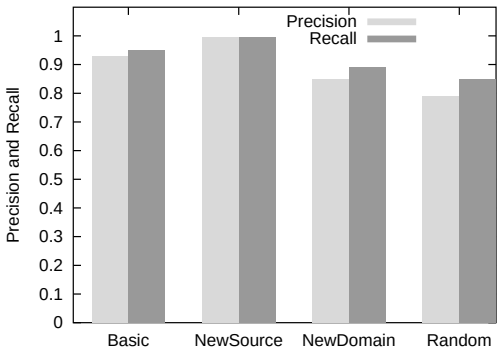
We test the derived grammar against the four datasets. The first one, as a baseline comparison, evaluates the performance over sources from which the grammar is derived. The other three test how well such a single grammar can perform over new sources (NewSource dataset), new domains (NewDomain dataset) and even random sources (Random dataset) from highly heterogeneous domains.

The experimental results (Section 6.2) show that our parsing approach can achieve good precisions and recalls (as Section 6.1 will introduce) for all these four datasets. Some sample results of the form extraction (as part of the exper-

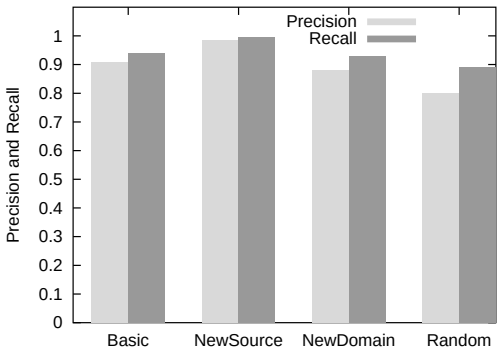


(a) Source distribution over precision.

(b) Source distribution over recall.



(c) Average precision and recall.



(d) Overall precision and recall.

Figure 15: Precision and recall over the four datasets.

iments) as well as the grammar we used are available as an online demo[3].

6.1 Metrics

Since the goal of our form extractor is to extract the query capabilities for each source, as a *set* of supported query conditions, we adopt *precision* and *recall* as our measurement of performance. For each query interface, we manually extract the set of conditions in its semantic model and compare with the ones extracted by the form extractor. We measure the results in two ways - per-source metric and overall metric.

- Per-source metric measures the result for each source. Given interface q, let $C_s(q)$ denote the set of conditions in its semantic model and $E_s(q)$ the extracted conditions by the form extractor. The following formula $P_s(q)$ and $R_s(q)$ calculate the precision and recall respectively for the interface q.

  $P_s(q) = \frac{C_s(q) \cap E_s(q)}{E_s(q)}$, $R_s(q) = \frac{C_s(q) \cap E_s(q)}{C_s(q)}$

- Overall metric measures, given a set of query interfaces w, the precision and recall over all conditions aggregated in those sources. Let $C_a(w)$ denote all the conditions in w and $E_a(w)$ the extracted conditions. The overall precision and recall over w are defined as:

  $P_a(w) = \frac{C_a(w) \cap E_a(w)}{E_a(w)}$, $R_a(w) = \frac{C_a(w) \cap E_a(w)}{C_a(w)}$

6.2 Experimental Results

Figure 15 summarizes the results of the global study: Figure 15(a) and (b) show the distribution of per-source precision and recall for the four datasets. For instance, in the Basic dataset, 69% sources have precision 1.0, and 72% sources have 1.0 recall. Figure 15(c) shows the average per-source precision and recall in the four datasets, and (d) the overall precision and recall. As we can see, for the Random dataset, the overall precision P_a(Random) achieves 0.8 and recall R_a(Random) 0.89, resulting in an accuracy (as the average of P_a and R_a) of 0.85.

It is interesting to observe that the result from the NewSource dataset has the best performance, which is even better than that of the Basic dataset. The reason may be that we might have some "bias" during the collection of data in the survey: We tend to collect complex forms with many conditions, since we want to know how complex query interfaces can be. However, when collecting the new sources, we are more "random." Therefore those query interfaces turn out to be simpler and show better accuracy.

The results from this set of experiments show that a single global grammar can achieve reasonable good performance across heterogeneous domains. The performance over the four datasets are rather even, as Figure 15(d) shows, and we do not observe significant performance drop when extending to more heterogenous sources - about 0.85 of overall precision and recall for first three datasets, and over 0.80 for randomly sampled sources.

7. CONCLUDING DISCUSSION

During our development of the system, we observe several interesting issues that deserve more future work. First, while

[3]The online demo is available at the MetaQuerier project web site at http://metaquerier.cs.uiuc.edu.

the current *merger* reports conflicting and missing elements, it is appealing to further resolve these problems by exploring other information. Specifically, to resolve the conflict in a specific query interface, we can leverage the correctly parsed conditions from other query interfaces of the same domain (e.g., using the extraction of flyairnorth.com to help the understanding of aa.com). Also, to handle missing elements, we find it promising to explore matching non-associated tokens by their textual similarity.

Second, our experiments indicate that a single global grammar seems quite acceptable for the form extraction task (as Section 6.2 reported) and thus it may not be critical to exploit other approaches to establish such a grammar. However, for other potential applications, it may be interesting to see how techniques such as machine learning can be explored to automate such grammar creation. In particular, we are interested in the selection of sources to be used in the training phase, where techniques such as active learning can be applied.

Third, as mentioned in Section 5, our preference framework is very simple: Preferences are "equal" (or "flat"), meaning there is no priority among preferences. Such simple framework assumes the consistency of the preferences, *i.e.*, different orders of applying the preferences yield the same result. In our application, the preferences defined are indeed simple and consistent. (Even for the seemingly contradictory preferences in the schedule graph of Figure 13, they are consistent in semantics.) Is such a simple framework sufficient for other applications? If consistency cannot be guaranteed in such applications, it is interesting to see how to develop and integrate a more sophisticated preference model (*e.g.*, prioritized preferences) into the parsing framework.

Fourth, can the best-effort parsing framework be used in other applications? We observe that many Web design "artifacts" follow certain concerted structure. For instance, the navigational menus listing available services are often regularly arranged at the top or left hand side of entry pages in E-commerce Web sites. Therefore, we believe, by designing a grammar that captures such structure regularities, we can employ our parsing framework to extract the services available in E-commerce Web sites.

To conclude, this paper presents a best-effort parsing framework to address the problem of understanding Web query interfaces. The concerted structure of query interfaces as we observed motivates our key hypothesis of hidden syntax: Query interfaces, although constructed autonomously, seem to be guided by a hidden syntax in the creation. Such a hypothesis transforms query interfaces as a visual language, and thus enables parsing as a principled framework to understand the semantic model of such a language. In particular, we propose a 2P grammar to declaratively and comprehensively capture the conventions of pattern presentations and their precedence. To address the challenges rendered by the derived syntax, we realize a "best-effort" parsing philosophy by a best-effort parser that on one hand prunes wrong interpretations as much and as early as possible, and on the other hand, understand the interface as much as possible. The experiments show the effectiveness of our approach.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Pub Co, 1986.

[2] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *SIGMOD Conference*, 2003.

[3] BrightPlanet.com. The deep web: Surfacing hidden value. Accessible at `http://brightplanet.com`, July 2000.

[4] K. C.-C. Chang, B. He, C. Li, and Z. Zhang. Structured databases on the web: Observations and implications. Technical Report UIUCDCS-R-2003-2321, Department of Computer Science, UIUC, Feb. 2003.

[5] K. C.-C. Chang, B. He, C. Li, and Z. Zhang. The UIUC web integration repository. Computer Science Department, University of Illinois at Urbana-Champaign. http://metaquerier.cs.uiuc.edu/repository, 2003.

[6] B. Chidlovskii and A. Bergholz. Crawling for domain-specific hidden web resources. In *Proceedings of 4thInternational Conference on Web Information Systems Engineering*, 2003.

[7] G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12):777–799, 1997.

[8] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB Conference*, pages 109–118, 2001.

[9] R. B. Doorenbos, O. Etzioni, and D. S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, 1997.

[10] E. J. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 4(2):371 – 394, 1991.

[11] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *SIGMOD Conference*, 2003.

[12] B. He, T. Tao, and K. C.-C. Chang. Clustering structured web sources: A schema-based, model-differentiation approach. In *EDBT'04 ClustWeb Workshop*, 2004.

[13] H. He, W. Meng, C. Yu, and Z. Wu. Wise-integrator: An automatic integrator of web search interfaces for e-commerce. In *VLDB Conference*, 2003.

[14] M. A. Hearst. Trends & controversies: Information integration. *IEEE Intelligent Systems*, 13(5):12–24, 1998.

[15] R. Helm, K. Marriott, and M. Odersky. Building visual language parsers. In *Proceedings on Human Factors in Computing Systems (CHI)*, pages 105–112, 1991.

[16] J. J. and G. E. Online parsing of visual languages using adjacency grammars. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, 1995.

[17] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[18] S. Liddle, S. Yau, and D. Embley. On the automatic extraction of data from the hidden web. In *Proceedings of the International Workshop on Data Semantics in Web Information Systems*, 2001.

[19] K. Marriott. Constraint multiset grammars. In *Proceedings of IEEE Symposium on Visual Languages*, pages 118–125, 1994.

[20] K. Marriott and B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):375–402, 1997.

[21] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB Conference*, 2001.

[22] K. Wittenburg and L. Weitzman. Relational grammars: Theory and practice in a visual language interface for process modeling. In *Proceedings of Workshop on Theory of Visual Languages, May 30, 1996, Gubbio, Italy*, 1996.

[23] K. Wittenburg, L. Weitzman, and J. Talley. Unification-based grammars and tabular parsing for graphical languages. *Journal of Visual Languages and Computing*, 4(2):347–370, 1991.