Data Parsing using Tier Grammars

Alexander Sakharov[1] and Timothy Sakharov[2]
[1]*Verizon, Waltham, Massachusetts, U.S.A.*
[2]*Northeastern University, Boston, Massachusetts, U.S.A.*

Keywords: Data Preprocessing, Unstructured Data, Data Languages, LL(1) Grammars, Predictive Parsing.

Abstract: Parsing turns unstructured data into structured data suitable for knowledge discovery and querying. The complexity of grammar notations and the difficulty of grammar debugging limit the availability of data parsers. Tier grammars are defined by simply dividing terminals into predefined classes and then splitting elements of some classes into multiple layered sub-groups. The set of predefined terminal classes can be easily extended. Tier grammars and their extensions are LL(1) grammars. Tier grammars are a tool for big data preprocessing.

1 INTRODUCTION

Knowledge discovery methods focus on structured data such as databases, semi-structured data such as XML, and natural language (NL) documents. Information retrieval is also well researched for structured data (SQL), XML (XQuery), and for NL (search engines). In reality, plenty of data are unstructured, and they are not precisely NL documents. Examples of such unstructured data include log files, dump files, documents combining NL, codes/abbreviations, references, and numeric data. NL processing methods cannot be efficiently used for these data because the NL that they contain is usually short and mixed with numeric and encoded values. These unstructured data need to be preprocessed to become usable for knowledge discovery or search.

Parsing turns unstructured data into structured data and can also serve as an information extraction utility. Hard-coded parsers are typically used for processing unstructured data. Their implementation is costly and error-prone. These parsers require software updates with every change in data format. Due to these implementation problems, documents combining NL with other kinds of data may even be treated as NL, and the other data including numbers become noise. The declarative programming of parsers using grammars partially solves these problems and is a good fit for data preprocessing.

The output of grammar-based parsers is an abstract syntax tree (AST) (Aho et al., 2006) which contains syntactic information extracted from the source. ASTs can be represented as DOM trees or can be converted to XML or JSON. The XML or JSON generated from ASTs is structured data because the set of node tags and the schema are predefined. For the same reason, ASTs can be loaded into relational database tables. Following parsing, knowledge can be extracted from DOM, database tables, XML, or JSON, and queries can be executed. ASTs may provide leverage for information extraction (Tari et al., 2012). Note that fielded search (`http://lucene.apache.org`) and XQuery and XPath Full Text (`http://www.w3.org/TR/xpath-full-text`) search can be applied to the transformed data originating from documents including NL fragments.

Context-free grammars (CFG) are an excellent mechanism for specifying the syntax of and parsing programming languages (Aho et al., 2006) but they are rarely used for data parsing because of the complexity of their notation. Few software developers have experience with CFGs. Creating an unambiguous CFG is a challenge even for experts. The power of available grammar inference methods (Sakakibara, 1997) is not sufficient to handle real-world problems. Note also that the inferred grammars have to be analyzed and their nonterminals have to be mapped to meaningful constructs for further data processing, which is a non-trivial task.

There exist ample differences between programming languages and data languages. In contrast to programming languages, data languages normally have a limited variety of constructs. Data languages mostly consist of aggregation constructs and references. The former represent structures with named fields or sets including maps, i.e. key-value pairs. Data languages are less constraining and strict than programming languages. Almost always, some portions of data somehow diverge from any given standard. Therefore, grammars for defining the syntax of

Sakharov, A. and Sakharov, T..
Data Parsing using Tier Grammars.
In *Proceedings of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015) - Volume 1: KDIR*, pages 463-468
ISBN: 978-989-758-158-8

data should be inclusive in order to avoid undesirable exceptions when processing these data. In contrast to programming languages, data formats are plentiful and evolve all the time. It is important, especially for big data, to be able to easily modify data grammars without the danger of compromising their properties. It is also important to be able to parse data using an incomplete grammar because the exact syntax of big data may not be known. Moreover, big data are often syntactically incoherent, and grammars apply to data fragments only. Therefore, a family of tiny grammars may be needed to specify the syntax of data.

An adequate notation for defining the syntax of data languages should be on par with regular expressions in terms of simplicity and comprehensibility. Unfortunately, regular expressions themselves are not a good choice for defining data languages because of their limited expressiveness and because they do not help build informative ASTs. The use of such notation should not require sophisticated tools for parser generation, and parsing should be feasible in linear time. We introduce a grammar notation that satisfies the above criteria.

This notation has no nonterminals, no grammar productions, and no formulas. A language is defined in this notation by simply dividing terminals into predefined classes. Each class has its role. There could be multiple layered sub-groups within a class. Note that the choice of terminal classes in this notation is not motivated by theoretical considerations but rather is driven by the intent to cover more constructs used in practice, while maintaining a clear meaning for every terminal class.

Our notation defines a subset of LL(1) languages, which makes predictive parsing possible (Aho et al., 2006). These languages are unambiguous, and are devised to be very inclusive. We give a simple characterization of strings belonging to these languages. This notation is rich enough for specifying data formats of various kinds of documents, including machine-generated documents. Our notation facilitates the definition of constructs representing data aggregates and references. This notation is especially beneficial for big data tasks because it enables the quick and easy specification of multiple data formats as well as the modification and augmentation of these specifications. We call this notation tier grammars because their constructs stack according to the priorities of layered terminals groups. Tier grammars can be easily combined and extended in a variety of ways without compromising the LL(1) property.

2 DEFINITION OF TIER LANGUAGES

Following the tradition for programming languages, it is assumed that lexical analysis using regular expressions is done before parsing. The output of lexical analysis is a sequence of tokens whose names are terminals for parsing. As usual, the longest lexeme is selected in case of conflicts (Aho et al., 2006). If the syntax is known for portions of the input, then regular expressions are also used to select these fragments before parsing them.

Suppose the set of terminals T is a union of disjoint sets T_1, T_2, T_3, T_4, T_5, T_6, T_7. T_1 is the set of base terminals. Terminals from T_2 and T_3 define bracketed constructs. Terminals from T_2 are opening brackets, and terminals from T_3 are closing ones. Terminals from T_4 are called markers. These terminals are split into disjoint groups by their priority. Their role is to serve as delimiters that combine items to the left and right of them in groups.

Terminals from T_5 are called postfixes, and act as postfix operators. Terminals from T_6 are called prefixes; these are unary prefix operators. Terminals from T_7 are connectives that serve either as binary operators in expressions or as separators, such as in the comma-separated values format. Prefixes, postfixes, and connectives are also split into disjoint groups by their priority. They share the range of priorities but only one kind of terminals is allowed for a given priority. Let q be the highest priority for markers and k be the highest priority for postfixes, prefixes, and connectives. We use $\underline{i}$ to denote the number of distinct markers, postfixes, prefixes, or connectives of priority i.

The tier language $\Lambda(T)$ for $T = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$ is defined recursively by the following rules. Understanding these rules does not require any knowledge of CFGs, but tier languages can still be expressed via CFGs. We give CFG productions along with the rules in order to demonstrate how the rules map to them. S will denote the start nonterminal of the corresponding CFG. Symbol ε will denote the empty string. $T_{4i}, T_{5i}, T_{6i}, T_{7i}$ will denote the respective terminals of priority i. Note that only one of T_{5i}, T_{6i}, T_{7i} may be non-empty for any i.

1. If $b \in T_1 = \{b_1, ..., b_{\underline{b}}\}$, then $b \in \Lambda(T)$.
$A \to b_1|...|b_{\underline{b}}$

2. If $a \in \Lambda(T), r \in T_2 = \{r_1, ..., r_{\underline{r}}\}, e \in T_3 = \{e_1, ..., e_{\underline{e}}\}$, then $rae \in \Lambda(T)$.
$B \to FSH$; $F \to r_1|...|r_{\underline{r}}$; $H \to e_1|...|e_{\underline{e}}$

3. Let either $c_1, ..., c_n \in T_{7i} = \{c_{i1}, ..., c_{i\underline{i}}\}$ (connective), $p \in T_{6i} = \{p_{i1}, ..., p_{i\underline{i}}\}$ (prefix), or $s \in T_{5i} = \{s_{i1}, ..., s_{i\underline{i}}\}$ (postfix). If $a_1, ..., a_n, a_{n+1} \in \Lambda(T)$,

$a_1, \ldots a_n, a_{n+1}$ are defined by rules 1, 2, or this rule for terminals of higher priority, $a_0 \in \Lambda(T), a_0$ is defined by rules 1, 2, or this rule for terminals of the same or higher priority, then $a_1 c_1 a_2 c_2 \ldots a_n c_n a_{n+1} \in \Lambda(T)$, $pa_0 \in \Lambda(T)$, or $a_1 s \in \Lambda(T)$.
$C \to A|B$

postfix:
$E_i \to E_{i+1} G_i$ (for $i = 1, \ldots, k-1$); $E_k \to C G_k$
$G_i \to \varepsilon | s_{i1} | \ldots | s_{i\underline{i}}$

prefix:
$E_i \to E_{i+1} | p_{i1} E_i | \ldots | p_{i\underline{i}} E_i$ (for $i = 1, \ldots, k-1$)
$E_k \to C | p_{k1} E_k | \ldots | p_{k\underline{k}} E_k$

connective:
$E_i \to E_{i+1} L_i$ (for $i = 1, \ldots, k-1$); $E_k \to C L_k$
$L_i \to \varepsilon | c_{i1} E_{i+1} L_i | \ldots | c_{i\underline{i}} E_{i+1} L_i$ (for $i = 1, \ldots, k-1$)
$L_k \to \varepsilon | c_{k1} C L_k | \ldots | c_{k\underline{k}} C L_k$

4. If $m_1, \ldots, m_n \in T_{4i} = \{m_{i1}, \ldots, m_{i\underline{i}}\}$, $a_0, a_1, \ldots, a_n \in \Lambda(T)$, then $\varepsilon \in \Lambda(T)$, $a_1 \ldots a_n \in \Lambda(T)$, $a_0 m_1 a_1 \ldots m_n a_n \in \Lambda(T)$ provided that this string follows the beginning of the input string, a terminal from T_2, or a marker of lower priority, and precedes the end of the input string, a terminal from T_3, or a marker of lower priority.
$Q_i \to Q_{i+1} R_i$ (for $i = 1, \ldots, q-1$); $Q_q \to D R_q$
$R_i \to \varepsilon | m_{i1} Q_{i+1} R_i | \ldots | m_{i\underline{i}} Q_{i+1} R_i$ (for $i = 1, \ldots, q-1$)
$R_q \to \varepsilon | m_{q1} D R_q | \ldots | m_{q\underline{q}} D R_q$
$D \to \varepsilon | E_1 D$

Now we only need to add one more production to complete the definition of the corresponding CFG: $S \to Q_1$. The above context-free productions have to be slightly modified when some terminal sets are empty. In case the sets of connectives, postfixes, and prefixes are all empty: $E_1 \to C$. In case the set of markers is empty: $Q_1 \to D$.

These terminal classes are suitable for various representations of data aggregates and references: prefixes, postfixes, and connectives for named fields in structures; brackets for structures, including recursive ones; markers and connectives for sets, including multi-dimensional arrays; connectives for key/value pairs; prefixes and brackets for references. Rule applications define parse trees for tier languages. Applications of rule 1 constitute the terminal nodes of these parse trees. Every application of all other rules corresponds to a nonterminal node of the parse tree.

Parse trees for tier grammars are similar to the ASTs of the underlying CFG. One difference is that one node in a tier grammar parse tree combines all associated connectives or markers. Tier grammar parse trees are very compact, which is especially important for big data. This translates into compact XML or database representations with simple schemas. XPath is widely used for expressing queries and information extraction wrappers (Dalvi et al., 2011) for HTML. It can fulfill the same purposes for tier grammar parse trees due to their simplicity.

3 EXAMPLES

Typical data dump formats such as CSV and other formats for multidimensional arrays can be easily specified as tier grammars. The same applies to the output of many Unix commands and of many command-line tools. Here are a couple of other simple examples of data formats that can be parsed using tier grammars. In these examples, \b denotes a space and \n denotes a new line character.

1. BibTex format (please see its specification at `http://www.bibtex.org`)
Base terminals: words and quoted strings
Brackets: { }
Prefix: words starting with @ (priority 4)
Connectives: # (priority 3) = (priority 2) , (priority 1)

2. Documents with numbered sections
Markers: .\b ?\b !\b .\n ?\n !\n (priority 3); \n\n (priority 2); section numbers defined as lexemes \n[0-9]* (priority 1)

Machine-generated human-readable files are the main source of examples of tier languages. The output of Apache's `ReflectionToStringBuilder` (`http://commons.apache.org`) is one example. Let us look at some code fragments that generate log files. The following code patterns demonstrate why log files or some parts contained therein are usually tier languages.

```
print(<opening bracket>);
   loop: { ... print(<data>); ... }
   print(<closing bracket>);
function f(...){ print(<opening bracket>);
   ... f(...); ... print(<closing bracket>);
   return; }
loop: { ... case ...: print(<prefix>);
   print(<data>); ... }
loop: { ... print(<data>); if ( ... )
   print(<postfix>); ... }
loop: { ... print(<data>);
   print(<connective>); print(<data>); ... }
loop: { if ( !first ) print(<connective>);
   ... print(<data>); ... }
loop: { loop: { loop: { ... print(<data>);
   ... } ... print(<high priority marker>);
   ... } ... print(<low priority marker>);
   ... }
```

4 ANALYSIS

Proposition 1. *Tier grammars define LL(1) languages.*

The availability of matching LL(1) grammars makes table-driven predictive parsing (Aho et al., 2006) possible for tier languages. Predictive parsing has a linear time complexity. The uniformity of tier languages with respect to predictive parsing is an essential benefit because most questions about CFGs are undecidable. Note that $S \Rightarrow^* N$ for every nonterminal N from tier grammar parse trees. This is an indication of the inclusiveness of tier grammars. Tier languages are unambiguous.

LL(1) parsing does not require any parser generator tools. A parser can be implemented as a couple of library functions. One of them builds a parsing table, and the other parses the input. In the case of gigantic documents, parsing can be implemented via callbacks like it is done in the SAX API for XML in Java

(`http://www.saxproject.org`):

```
void parse(LexemeStream stream,
   EventHandler handler);
```

where class EventHandler has callback methods for terminals from T_1, T_2, T_3, as well as for prefixes, postfixes, connectives, markers. The latter methods are called when the corresponding nonterminal is popped from the stack.

The set of tier languages is a proper subset of LL(1) languages. It includes languages that are not regular. For instance, the language $\{a^n b^n | n \geq 0\}$ is one such example. Since tier languages are designed to be as inclusive as possible, they do not even include some restrictive regular languages. For instance, the language defined by regular expression $(ab)^*$ and any language with a finite set of distinct strings are not tier languages. If a tier grammar does not have brackets, then it defines a regular language.

Since the tier grammar notation does not involve any kind of formulas, terminals can only serve as tags giving a particular syntactic meaning to neighboring items or to strings starting or ending with them. Prefixes give a syntactic meaning to the item to the right. Postfixes do the same for the item to the left. A connective glues together the two items adjacent to it. Markers group items on the left and on the right. Brackets define construct borders. Altogether, they cover more important cases.

The following simple characterization of tier languages shows that every tier language includes a wide variety of strings. This helps avoid parsing exceptions.

Proposition 2. *A string belongs to a given tier language if and only if the following conditions hold:*

- brackets are balanced, i.e. the number of opening brackets in the string is equal to the number of closing brackets, and the number of opening brackets is greater or equal to the number of closing brackets in any prefix substring

- every postfix follows a base token, closing bracket, or another postfix of a higher priority

- every prefix precedes a base token, opening bracket, or prefix of the same or higher priority

- every connective follows a base token, closing bracket, or postfix of a higher priority and precedes a base token, opening bracket, or prefix of a higher priority

Corollary 1. *If T'_2, T'_3, T'_4, T'_5, T'_6, T'_7 are subsets of T_2, T_3, T_4, T_5, T_6, T_7, respectively, $s \in \Lambda(\{T_1, T_2, T_3, T_4, T_5, T_6, T_7\})$, and terminals from $T_2 \setminus T'_2$ are balanced with terminals from $T_3 \setminus T'_3$ in s, then $s \in \Lambda(\{T_1 \cup T'_2 \cup T'_3 \cup T'_4 \cup T'_5 \cup T'_6 \cup T'_7, T_2 \setminus T'_2, T_3 \setminus T'_3, T_4 \setminus T'_4, T_5 \setminus T'_5, T_6 \setminus T'_6, T_7 \setminus T'_7\})$.*

This corollary guarantees that parsing with incomplete syntax will work. The extension of syntax usually amounts to assigning other roles to some of the base terminals. Another corollary of Proposition 2 is that all strings belong to every tier language containing only base terminals and markers.

5 EXTENDING AND COMBINING TIER GRAMMARS

If the expressiveness of tier grammars is not sufficient, they can be easily extended. One extension is the addition of prefixes of arity more than one. This extension is introduced by the following context-free production: $E_i \rightarrow E_{i+1} | pE_i ... E_i$ where the number of E_i is the arity of p. Another typical extension is a construct defined by a terminal pair: $E_i \rightarrow E_{i+1} | p_1 E_i p_2 E_i$. It may also be useful to add other types of prefixes. These other prefixes share priorities with markers, as opposed to connectives and postfixes. They are introduced by the following production: $Q_i \rightarrow Q_{i+1} | p_1 Q_i | ... | p_n Q_i$. One more example is a construct with three constituents, where the third one is optional. This construct is defined by the following productions: $E_i \rightarrow p_1 E_i p_2 E_{i+1} L_i$, $L_i \rightarrow \varepsilon | p_3 E_{i+1}$.

We specify a class of productions that can be added to tier grammars to form extensions. All aforementioned examples belong to this class. The four following types of productions guarantee that any extension defined by them is a LL(1) grammar. Let α_j denote a string of terminals and/or nonterminals.

1. $E_i \rightarrow \alpha_1 | ... | \alpha_n$

where E_i, E_{i+1}, and S are the only nonterminals that may occur in any α_j, all α_j start with a terminal or E_{i+1}, not more than one α_j starts with E_{i+1}, and ev-

ery occurrence of S in α_j should be preceded and followed by terminals.
2. $E_i \to \alpha_0 L_i$
$L_i \to \varepsilon|\alpha_1 L_i|...|\alpha_n L_i$ (or $L_i \to \varepsilon|\alpha_1|...|\alpha_n$)
where α_0 starts with E_{i+1} or a terminal, other α_j start with a terminal, E_i, E_{i+1}, and S are the only nonterminals that can occur in α_j, and every occurrence of S and E_i in any α_j should be preceded and followed by terminals.
3. $Q_i \to \alpha_1|...|\alpha_n$
where Q_i, Q_{i+1}, and S are the only nonterminals that may occur in α_j, all α_j start with a terminal or Q_{i+1}, not more than one α_j starts with Q_{i+1}, every occurrence of S and Q_i in any α_j should be preceded and followed by terminals, and every two consecutive occurrences of Q_{i+1} in any α_j should be separated by a terminal.
4. $Q_i \to \alpha_0 R_i$
$R_i \to \varepsilon|\alpha_1 R_i|...|\alpha_n R_i$ (or $R_i \to \varepsilon|\alpha_1|...|\alpha_n$)
where α_0 starts with Q_{i+1} or a terminal, other α_j start with a terminal, Q_i, Q_{i+1}, and S are the only nonterminals that can occur in α_j, every occurrence of S and Q_i in any α_j should be preceded and followed by terminals, and every two consecutive occurrences of Q_{i+1} in any α_j should be separated by a terminal.

All terminals from α_j are distinct from terminals from T. No terminal may occur more than once in all productions. As with basic tier grammars, one extension production defines a class of terminals. This production can be used as a template for the introduction of multiple instances of this production, each having distinct terminals and a priority. The first two types of extension productions add new priorities to those of postfixes, prefixes, and connectives. The last two types add new priorities to the priorities of markers. The priorities of the original tier grammar should be shifted accordingly.

Proposition 3. *Extended tier grammars define LL(1) languages.*

If the flexibility of a single tier grammar is not sufficient, then multiple tier grammars can be combined so that every source grammar applies only to a relevant portion of a document. The advantage of combining multiple tier grammars vs CFGs is that the simplicity of the notation is not compromised. Note that the terminals of combined grammars may intersect. If the set of terminals of tier grammar Γ_1 does not include $e_{\underline{e}+1}$, then Γ_1 can be combined with Γ by modifying the B production of Γ to the following:
$B \to FSH|r_{\underline{r}+1}S_1 e_{\underline{e}+1}$
where S_1 is the start nonterminal of Γ_1. If the set of terminals of Γ_1 is disjoint with $\{T_3, T_{41}, ..., T_{4i-1}\}$ for Γ, then Γ_1 can be combined with Γ by adding a marker tier. Here is the R_i production for this tier:
$R_i \to \varepsilon|m_{i1}S_1$
The combined grammars define LL(1) languages.

6 RELATED WORK

Several alternatives to the notation of CFGs have been developed (Ford, 2004; Aho et al., 2006; Berstel and Boasson, 2002). With the exception of regular expressions, none of these alternatives really simplified the task of creating and debugging grammars. Stochastic CFG parsers (Chappelier and Rajman, 1998) have a prohibitive time complexity for data that may be much bigger than programs.

Despite the remarkable research in the area of formal grammars, its applications to data parsing are few and far between (Underwood, 2012; McCann and Chandra, 2000; Back, 2002; Fisher and Gruber, 2005; Xi and Walker, 2010; Powell et al., 2011). An overview of data description languages can be found in (Fisher et al., 2006). None of these data description languages are on par with tier grammars in terms of the simplicity of specification for data formats.

Regular expressions have been used for information extraction tasks (Appelt and Onyshkevych, 1998), particularly for entity recognition. Google uses regular expressions and LL(1) grammars for entity recognition in their Search Appliance. There exist techniques for learning regular expressions and CFGs utilized in entity recognition (Li et al., 2008; Viola and Narasimhan, 2005). Grammars for the purpose of entity recognition should be strict, unlike tier grammars. Tier grammars capture the overall syntactic structure.

Grammar inference methods are basically limited to regular languages and other simple languages (Sakakibara, 1997). RoadRunner (Crescenzi and Mecca, 2004) infers union-free regular grammars that are used to extract information from large web sites. A method of learning CFG productions that specify the syntax of web server access logs is presented in (Thakur et al., 2013). The log format considered in this paper is a very simple regular language. It is not clear if this inference method will work for more complex languages.

7 CONCLUSION AND FUTURE WORK

Grammars enable the declarative programming of data preprocessors that extract syntactic information from unstructured sources and generate structured data that, in turn, serve as input for knowledge discovery and querying. Specifying a grammar by splitting terminals into meaningful disjoint subsets is one of the easiest ways to describe syntax. It is even simpler than regular expressions. The family of tier grammars presented and investigated here has sufficient expressive power to describe the syntax of many data languages. Tier grammars can be extended and combined, and predictive parsing is possible for all of them. Tier grammars have the qualities that are important for data parsing, particularly for parsing big data. The idea behind tier grammars that leads to LL(1) conditions is considering nonterminals as an ordered set and limiting productions to the forms in which forward references in the right-hand sides are always to the next nonterminal and backward references are bracketed by terminals.

Tier grammars can be embedded into LL(1) grammars. This gives a mechanism for defining multiple variants of syntactically complex languages. The LL(1) grammar part takes care of the syntactic difficulties whereas the tier part enables easy syntax modifications with the guarantee of predictive parsing. Defining stochastic tier grammars is easier than defining stochastic CFGs. Probabilities are given for terminal membership in classes/sub-groups rather than for productions. Tier grammar inference from positive examples can be formulated as a discrete optimization problem. Further investigation of all these topics is beyond the scope of this paper.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Appelt, D. E. and Onyshkevych, B. (1998). The common pattern specification language. In *Proceedings of a Workshop on Held at Baltimore, Maryland: October 13-15, 1998*, TIPSTER '98, pages 23–30, Stroudsburg, PA, USA. Association for Computational Linguistics.

Back, G. (2002). Datascript - a specification and scripting language for binary data. In *In Generative Programming and Component Engineering*, pages 66–77. Springer.

Berstel, J. and Boasson, L. (2002). Balanced grammars and their languages. In *Formal and Natural Computing - Essays Dedicated to Grzegorz Rozenberg [on occasion of his 60th birthday, March 14, 2002]*, pages 3–25.

Chappelier, J.-C. and Rajman, M. (1998). A generalized cyk algorithm for parsing stochastic cfg. In *Proceedings of Tabulation in Parsing and Deduction (TAPD'98)*, pages 133–137, Paris, France.

Crescenzi, V. and Mecca, G. (2004). Automatic information extraction from large websites. *J. ACM*, 51(5):731–779.

Dalvi, N., Kumar, R., and Soliman, M. (2011). Automatic wrappers for large scale web extraction. *Proc. VLDB Endow.*, 4(4):219–230.

Fisher, K. and Gruber, R. (2005). Pads: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 295–304, New York, NY, USA. ACM.

Fisher, K., Mandelbaum, Y., and Walker, D. (2006). The next 700 data description languages. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 2–15, New York, NY, USA. ACM.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA. ACM.

Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., and Jagadish, H. (2008). Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30. Association for Computational Linguistics.

McCann, P. J. and Chandra, S. (2000). Packet types: Abstract specification of network protocol messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA. ACM.

Powell, A., Beckerle, M., and Hanson, S. (2011). Data format description language (dfdl). Technical report, Open Grid Forum.

Sakakibara, Y. (1997). Recent advances of grammatical inference. *Theoretical Computer Science'*, 185(1):15–45.

Tari, L., Tu, P. H., Hakenberg, J., Chen, Y., Son, T. C., Gonzalez, G., and Baral, C. (2012). Parse tree database for information extraction. *IEEE Transactions on Knowledge and Data Engineering*, 24(1):86–99.

Thakur, R., Jain, S., and Chaudhari, N. S. (2013). User behavior analysis using alignment based grammatical inference from web server access log. *International Journal of Future Computer and Communication*, 2(6):543.

Underwood, W. (2012). Grammar-based specification and parsing of binary file formats. *International Journal of Digital Curation*, 7(1):95–106.

Viola, P. and Narasimhan, M. (2005). Learning to extract information from semi-structured text using a discriminative context free grammar. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 330–337. ACM.

Xi, Q. and Walker, D. (2010). A context-free markup language for semi-structured text. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 221–232, New York, NY, USA. ACM.