Saggitarius: A DSL for Specifying Grammatical Domains

ANDERS MILTNER, Simon Fraser University, Canada
DEVON LOEHR, Princeton University, USA
ARNOLD MONG, Princeton University, USA
KATHLEEN FISHER, Tufts University, USA
DAVID WALKER, Princeton University, USA

Common data types like dates, addresses, phone numbers and tables can have multiple textual representations, and many heavily-used languages, such as SQL, come in several dialects. These variations can cause data to be misinterpreted, leading to silent data corruption, failure of data processing systems, or even security vulnerabilities. Saggitarius is a new language and system designed to help programmers reason about the format of data, by describing *grammatical domains*—that is, sets of context-free grammars that describe the many possible representations of a datatype. We describe the design of Saggitarius via example and provide a relational semantics. We show how Saggitarius may be used to analyze a data set: given example data, it uses an algorithm based on semi-ring parsing and MaxSAT to infer which grammar in a given domain best matches that data. We evaluate the effectiveness of the algorithm on a benchmark suite of 110 example problems, and we demonstrate that our system typically returns a satisfying grammar within a few seconds with only a small number of examples. We also delve deeper into a more extensive case study on using Saggitarius for CSV dialect detection. Despite being general-purpose, we find that Saggitarius offers comparable results to hand-tuned, specialized tools; in the case of CSV, it infers grammars for 84% of benchmarks within 60 seconds, and has comparable accuracy to custom-built dialect detection tools.

ACM Reference Format:
Anders Miltner, Devon Loehr, Arnold Mong, Kathleen Fisher, and David Walker. 2023. Saggitarius: A DSL for Specifying Grammatical Domains. 1, 1 (August 2023), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data transformation, cleaning, and processing is a tedious and difficult task that stands in the way of getting important *information* out of the raw text all around us. One particularly thorny problem is that the same information is often formatted differently when drawn from different sources. Correctly parsing even a single data source in the face of such ambiguities can be difficult; if data is drawn from multiple sources, the chance of formatting inconsistencies and errors skyrockets. This lack of standardization can have real costs—if a file is incorrectly parsed, it may lead to processing system failures, silent data corruption that pollutes critical data bases, or even security vulnerabilities [Bratus 2020].

For example, consider processing a data file that contains a collection of dates—those dates can assume a range of different formats, and some of those formats are ambiguous. If a data processing system assumes a DD/MM/YY format when MM/DD/YY appear in the data, the system may crash or corrupt back-end databases. Domain specific languages for data processing, such as PADS [Fisher and Walker 2011], aim to help users solve such problems by providing a high-level language that allows users to describe their data *exactly*. However, doing so is not without its own difficulties. For example, the PADS Project's standard library [Fisher 2009] provides 108 (!) different functions *just for reading dates, times, and timestamps* (some due to differing data encodings such as ASCII vs binary). Other common basic types found in the "ad hoc" data formats processed by systems like

Authors' addresses: Anders Miltner, Simon Fraser University, Burnaby, BC, Canada, amiltner@cs.princeton.edu; Devon Loehr, Princeton University, Princeton, NJ, USA, dloehr@princeton.edu; Arnold Mong, Princeton University, Princeton, NJ, USA, among@alumni.princeton.edu; Kathleen Fisher, Tufts University, Medford, MA, USA, kfisher@cs.tufts.edu; David Walker, Princeton University, Princeton, NJ, USA, dpw@cs.princeton.edu.

PADS include phone numbers, addresses, times, names, postal codes, and abbreviations (such as for states or provinces). Such data also has variations, especially when that data may involve different national conventions [Wikipedia contributors 2023a,b]. One can see why a programmer new to a system like PADS would have difficulty selecting among options to craft a format entirely by hand. Indeed, even experts can face problems: Fisher et al. [2008] describe an experience working with PADS at AT&T where it took roughly three weeks to craft a description of a data format, in part because the format was not consistent, shifting part-way through.

The problem is not limited to "ad hoc" formats like those addressed by the PADS system. For instance, one might think it safe to assume that a "comma-separated-value" (CSV) file will be formatted as a series of values separated by commas. For instance, one might think it safe to assume that a "comma-separated-value" (CSV) file will be formatted as a series of *values* separated by *commas*. One would be wrong: tabs, vertical bars, semi-colons, spaces, or carats are all sometimes used instead of commas to separate fields. Furthermore, some files use quotes to delimit strings and related symbols, while others use tildes or single quotes. And of course, some CSV files are "typed," with, for example, one column expected to contain integers, and another strings.

As a result, the appropriate way to parse a CSV file can be ambiguous [van den Burg et al. 2019b]; to avoid incorrectly interpreting it, one must determine the correct CSV grammar (or "dialect") to use for a given data set. In the case of CSV, there are some tools to help users. For instance, Microsoft Excel provides a wizard that allows a user to select the kind of delimiter to use when importing a CSV file as a spreadsheet; unfortunately, the process is a manual one. To try to deal with such variations in format, Python has libraries that implement "sniffers" to detect CSV dialects. However, the range of ambiguities make such tools difficult to develop. Indeed, van den Burg *et al.* provide a host of examples of the kinds of ambiguities they found when analyzing CSV files in the wild. In attempt to improve the state of the art (just for CSV), they built their own custom CSV sniffer.

CSV is not the only domain suffering problems due to ambiguity. While the PDF format has been standardized, different tools implement different subsets (or even supersets!) of the standard. Moreover, some of these PDF variants, and the tools used to process them, contain vulnerabilities [Carmony et al. 2016; Liu 2017]. Hence, ambiguities in how to process PDF lead not just to bugs, but possibly to security vulnerabilities. The DARPA SafeDocs program [Bratus 2020] is currently exploring ways to define safe subsets of PDF to limit tool vulnerabilities.

The problem of determining the format of a file (or group of files) is called *grammar induction*. Historically, grammar induction has been an exceptionally difficult problem studied by countless researchers (see [Angluin 1978, 1987; Firoiu et al. 1998; Garcia and Vidal 1990; Gold 1967; Oncina and García 1992; Rivest and Schapire 1989; Vidal 1994] for just a few examples from the literature). Without any prior information, the space of grammars that might describe a data set is enormous. Consequently, modern grammar induction tools either require a large volume of examples, are specialized to particular tasks, and/or do not scale particularly well. Indeed, even the restricted case of regular expression inference is a challenge: Chen et al. [2020] observe that Angluin's classic L^* algorithm makes 679 queries just to infer the simple regular expression [a-zA-Z]+, and Lee et al. [2016] show that the search space for regular expressions grows at a rate of c^{2^d-1} where d is the depth of the regular expression and c is the number of regular expression operators (c is 7 when working with a binary alphabet). While recent research [Chen et al. 2020; Lee et al. 2016] has taken impressive steps forward in the special case of regular languages, scaling to larger data formats remains a challenge. For instance, AlphaRegex still classifies some regular expressions over a binary alphabet with 10-20 symbols as "hard" [Lee et al. 2016, Table 3]. Scaling to richer classes of grammars, such as context-free languages, is even more daunting.

Syntax-Guided Grammar Induction. In this paper, we explore a new kind of grammar inference strategy, which we term *syntax-guided grammar induction*. Our work was heavily inspired by another, closely related problem: program synthesis. Progress on that problem was made recently through the use of syntax-guided (i.e., SYGUS [Alur et al. 2013]) methods and tools. Just as SYGUS tools constrain programs syntactically to cut down the search space and speed program synthesis, we constrain *grammars* syntactically to cut down the search space and speed grammar synthesis.

Central to our strategy is a new way of thinking about the formatting options for a datatype, centered around the idea of a *grammatical domain*. A grammatical domain is a *set* of formal grammars that describe the many possible representations of a datatype. For instance, the *date grammatical domain* would contain one grammar describing the DD/MM/YY format, another grammar describing the MM/DD/YY format, and many other grammars describing other formats (e.g., MM-DD-YY, YYYY.MM.DD, and so on).

To rigorously specify and manage such domains, we develop a new language, called SAGGITARIUS. Such a language is a first step towards developing more robust data processing tools. Roughly speaking, SAGGITARIUS may be viewed as an extension of a standard YACC-based parser generator. However, whereas YACC defines a single grammar and outputs a parser, SAGGITARIUS defines a *set* of possible grammars—*i.e.,* a grammatical domain—and outputs a single grammar. To represent a set of grammars, SAGGITARIUS allows grammar engineers to specify certain grammatical productions as optional. Grammars within the domain are defined by the subset of optional productions that they include. SAGGITARIUS also has features that allow grammar engineers to declare constraints that force certain combinations of productions to appear, or not appear, and hence provides fine control over the grammars of the domain in question.

Once a grammatical domain is defined in SAGGITARIUS, it may be *applied* to example data. To apply a SAGGITARIUS program, the user provides a set of example data files which are marked as either positive or negative. Then, our *grammar induction algorithm* will attempt to find a matching element of the grammatical domain. That is, it will select productions that allow the resulting grammar to parse all positive examples and none of the negative ones. Because more than one grammar from the domain may satisfy the provided examples, SAGGITARIUS allows users to specify preferences that rank the generated grammars. For example, a preference might state that grammars with fewer productions should be preferred, or that certain productions are preferred over others.

We note that the ones *defining* the domain and those *using* the domain may (and frequently will) be different. We call the person who defines the domain the *grammar engineer* and the one who uses the grammar the *instance engineer*. These two roles are separate because we expect heavy reuse of grammatical domains. For instance, the date domain need only be defined once by an expert. It can then be used countless times by instance engineers developing specific data processing tools for formats that contain dates. While grammar engineers require sophisticated knowledge of a grammatical domain and the SAGGITARIUS tool, instance engineers need only supply appropriate examples from the domain in question.

A key strength of the language and system design we propose is that it allows programmers to leverage their domain knowledge, which is often substantial. For example, if an engineer expects their data to be formatted roughly as CSV files containing dates, they can write or re-use CSV and date metagrammars to heavily restrict the search space. In other words, we provide a linguistic framework that allows programmers to apply their prior knowledge about the data, and reduce the general grammar induction problem to a more specialized and tractable one.

Furthermore, our approach leads to a simple-yet-effective algorithm for grammar induction that nonetheless requires very few examples. We study the performance of our grammar induction algorithm on a set of ten different induction tasks, considering the performance of the algorithm on each task under a range of different conditions. We find that SAGGITARIUS frequently induces grammars in under 10 seconds, and we demonstrate that Saggitarius can often learn the expected grammar with a mere handful of examples.

Use Cases. Saggitarius is a broad tool, with a variety of potential applications. For example,

- Alice maintains a large cloud service that allows clients to access a database using SQL queries. She has found that a common source of errors is when users copy-paste code meant for a different SQL dialect. Alice could use a metagrammar to represent the different dialects of SQL, and use Saggitarius to produce a useful, automated error message that informs the user what dialect they're using, so they can easily adjust their code.
- Bob is in charge of aggregating data from a variety of sources, all represented as CSV files. However, different sources used different tools to generate their data, so each group of files potentially comes in its own unique variation of the CSV format. Bob can use the CSV metagrammar and apply it to each group of files individually to pick out the important features (separators, delimiters, etc), which can then be passed to an automated CSV parsing tool.
- Charlie is an IT professional at a high-security government facility. They wish to integrate a new, third-party PDF processing tool into their office's toolchain, but are worried that vulnerabilities in the tools may leak classified information if given certain inputs [Bratus 2020]. Charlie could use the PDF metagrammar to create a "shield" by finding a PDF benchmark suite, determining which benchmarks cause failures, and using Saggitarius to infer a grammar which matches only the successful benchmarks. They could then automatically compare inputs to this grammar before invoking the tool, ensuring it is only run on "good" inputs.

In each of these cases, Saggitarius allows users to rapidly develop a fitting tool for the domain at hand. Rather than manually writing a dialect recognizer, or a CSV sniffer, one need only write a metagrammar expressing their knowledge of the domain. When metagrammar components for the task already exist (dates, phone numbers, addresses, etc) one's task is reduced further. In the same way that it is easier to write a YACC file than to write a parser implementation, it is easier to write a metagrammar than a grammar induction engine from scratch.

Finally, Saggitarius *works* – despite its generality, it can achieve comparable performance to specialized, hand-tuned tools. We demonstrate in our evaluation section that when Saggitarius is used to detect the format of CSV files, it produces similar results to custom-built CSV sniffers.

To summarize, this paper makes the following contributions.

- We introduce the concept of *grammatical domains* and demonstrate that a variety of grammatical domains exist in the wild.
- We design the first user-facing language, Saggitarius, for specifying grammatical domains precisely, and supply an algorithm for inferring a candidate grammar from the domain.
- We create a benchmark suite for evaluating syntax-guided grammar induction algorithms and evaluate the performance of Saggitarius on that suite, showing that it often returns a satisfying grammar within a few seconds, using few examples.
- We demonstrate practical use cases of Saggitarius through a case study of the CSV metagrammar. We compare Saggitarius's performance to custom-built CSV sniffers, and find it yields comparable results. Additionally, in the full version of the paper [Miltner et al. 2023] we provide two additional case studies on metagrammars for XML and SQL. In the first, we demonstrate how the structure of metagrammars can be used to control the search space, speeding induction tasks. In the second, we show how Saggitarius can be used in practice to provide helpful feedback to users of query engines.

2 MOTIVATING EXAMPLES

Grammatical domains appear in many different contexts. In this section, we show how to use Saggitarius to define two useful domains: the domain of calendar dates and the domain of comma-separated-value (CSV) formats.

2.1 Example 1: Calendar Dates

Dates are formatted in many different ways. Because the various formats are ambiguous (causing confusion as to whether one is reading a MM/DD or DD/MM format, for instance), date parsers must be specialized to a particular data set. Said another way, dates formats form a natural grammatical domain, and different data sets adhere to different grammars within that domain.

Saggitarius programs specify a grammatical domain through the use of a *metagrammar*, which is a set of *candidate productions* (a.k.a. *candidate rules*) together with (a) constraints that limit which combinations of productions may appear, and (b) preferences that rank the grammars, for breaking ties when multiple grammars are applicable.

The simplest Saggitarius components specify productions using a YACC-like syntax with the form `N -> RHS.` Here, `N` is a non-terminal and `RHS` is a regular expression over terminals and non-terminals. For instance, to begin construction of our date grammatical domain, we can specify `Digit` and `Year` non-terminals as follows.

```
Digit -> ["0"-"9"].
Year -> Digit Digit | Digit Digit Digit Digit.
```

This first definition looks like a definition one might find in an ordinary grammar. It states that `Year` can have either two or four digits. The denotation of such a definition is a grammatical domain—in this case, a grammatical domain (a set) containing exactly one grammar.

Saggitarius is more interesting when one defines metagrammars that include optional productions. Optional productions are preceded by a "?" symbol. For instance, consider the following:

```
Digit -> ["0"-"9"].
Year -> ? Digit Digit ? Digit Digit Digit Digit.
```

The metagrammar above denotes a grammatical domain that includes four grammars:

- one grammar in which Year has no productions,
- two grammars in which Year has one production, and
- one grammar in which Year has two productions.

To extract a single grammar from this set of four grammars, one supplies the Saggitarius grammar induction engine with positive and negative example data. If no grammar parses all the data as required, the grammar induction algorithm will return "no viable grammar."

Continuing, consider the following specification for days.

```
Day -> ? ["1" - "9"]
       ? "0" ["1" - "9"]
       | ["1" - "2"] Digit | "30" | "31".
```

This metagrammar includes grammars for days ranging from 1 -> 31. It allows single digit days to be prefixed with a 0. However, it is natural to desire grammars that parse either single-digit days or 0-prefixed-days, but not both. One way to specify such a constraint is as follows.

```
constraint(|productions(Day)| = 4).
```

Here, the constraint specifies that the number of production rules for `Day` must be exactly 4. Since the three productions on the last row are always included, exactly one of the ? production candidates can be in the solution. Another option is to name productions and to use the names in constraints:

```
Day -> ? ["1" - "9"] as SingleDigitDays
       ? "0" ["1" - "9"] as ZeroPrefixDays
       | ...
constraint(SingleDigitDays XOR ZeroPrefixDays).
```

Here, we have given each of the rule candidates a name, which is used as an indicator variable in the constraint expression; each evaluates to `true` iff a given grammar includes that production. All constraints must be a boolean combination of indicator variables or integer comparisons; we have provided commonly-used operations (such as counting the number of productions) as syntactic sugar.

```
Sep -> ? ","  ? "/"  ? "-".
constraint(|Production(Sep)| = 1)

Digit -> ["0"-"9"].

Year -> ? Digit Digit
        ? Digit Digit Digit Digit.
constraint(|Productions(Year)| = 1)

Month ->  ? Digit
          ? "0" Digit
          | "10" | "11" | "12".
constraint(|Productions(Month)| = 2)

Day -> ? ["1" - "9"]
       ? "0" ["1" - "9"]
       | ["1" - "2"] Digit | "30" | "31".
constraint(|Productions(Day)| = 2).

Date -> ? Day   Sep Month Sep Year
        ? Month Sep Day   Sep Year
        ? Year  Sep Month Sep Day
        ? Year  Sep Day   Sep Month.
constraint(|Productions(Date)| = 1).

start Date
```

Fig. 1. Calendar Dates Metagrammar

Figure 1 includes the rest of the (simplified) definition of the date format, adding definitions for separators, months, and dates as a whole. Dates are also designated as the start symbol. The use of constraints is common. For instance, notice the grammar engineer who designed this particular format allowed for the possibility of several different separators, but required a single separator to be used consistently throughout a format. Hence, while a date format may use "-" or "/" as a separator, it never uses both.

To extract a particular grammar from the domain, an instance engineer will supply positive and/or negative example data. For example, one could supply U.S.-style dates `12/31/72` and `01/01/72`, marking them as positive examples. Having done so, the Saggitarius grammar induction algorithm

```
Sep -> "/".

Digit -> ["0"-"9"].

Year -> Digit Digit.

Month ->  "0" Digit | "10" | "11" | "12".

Day -> "0" ["1" - "9"]
     | ["1" - "2"] Digit
     | "30" | "31".

Date -> Month Sep Day Sep Year.

S -> Date
```

Fig. 2. Date Solution Grammar

might generate the example grammar presented in Figure 2. In this case, the solution presented in Figure 2 is unique. However, in other circumstances, multiple grammars might satisfy the given examples. To manage multiple grammars, one can rank grammars (e.g. preferring those with fewer productions) and ask for the most preferred grammar according to the ranking. This is explained further in Section 2.3.

While one might worry that a naive instance engineer could supply insufficient data and thereby underconstrain the set of possible solution grammars, such problems could likely be mitigated through a well-designed user interface that informs a user when multiple solutions are possible and presents example data to the user, asking them to choose valid and invalid instances of the format.

2.2 Example 2: CSV

Dates, phone numbers, addresses and the like are simple data types with many textual representations. Saggitarius is also capable of specifying domains that contain larger, aggregate data types. The domain of comma-separated-value (CSV) formats is a good example.

One challenge in specifying a CSV domain is that if we want the columns of the CSV format to be "typed" – one column must be integers, another strings or dates, for instance – we need to consider many, many potential grammar productions. To facilitate construction of such metagrammars succinctly, we allow grammar engineers to define indexed collections of productions. For instance, suppose we would like to specify a spreadsheet with three columns (numbered 0-2) where each column contains either only numbers or only strings. We might define the ith Cell in each row as:

```
Cell{i : [0,2]} -> ? Number ? String.
forall (i : [0,2])
  constraint(|Productions(Cell{i})| = 1)
```

This declaration defines three nonterminals simultaneously: `Cell{0}`, `Cell{1}`, and `Cell{2}` and provides the same definition for each of them. However, since each of `Cell{0}`, `Cell{1}`, and `Cell{2}` are separate non-terminals, the underlying inference engine can specialize them independently based on the supplied data. For instance, `Cell{0}` could be a string and `Cell{1}` and `Cell{2}` might both be numbers.[1]. Constraints can refer to specific indexed non-terminals as shown.

[1]Observant readers may worry that the characters "12" could be interpreted as either a number or a string if the definition of strings includes numbers. We will explain how to create preferences to disambiguate momentarily.

The domain of 3-column CSVs is likely a rare one! Fortunately, users can also declare collections of indexed non-terminals with an arbitrary natural-number size (e.g. `Cell{i:nat} -> ...`).

It is possible to restrict the possible productions based on an index. Below, we define `Row{i}`, a non-terminal for a row containing cells `Cell{0}` through `Cell{i}`. The use of normal context-free definitions allows `Row{i}` to refer to `Row{i-1}`. Notice that separators (`Sep`) are not indexed, so that one separator definition that is used consistently throughout the entire grammar.

```
Row{i : nat} ->
  ? if (i = 0) then Cell{i}
  ? if (i > 0) then Row{i-1} Sep Cell{i}.

Sep -> ? "," ? "|" ? ";".
  constraint(|Productions(Sep)| = 1).

Table{i : nat} -> MyRow{i} ("\n" MyRow{i})*.
```

`Row{i}` represents a single row with `i` `Cell`s, and similarly `Table{i}` uses the standard Kleene star to represent a table with an arbitrary numbers of rows of length `i` (we could equally well have written the usual recursive, context-free definition instead). The only difficulty that remains is the fact that, while we expect each CSV file to have a fixed number of columns, we cannot know that number in advance. Saggitarius allows users to represent such unknown quantities by declaring *existential variables* and using them in rules. For example, we might use an existential variable to define a CSV format by adding the following code:

```
exists rowlen : nat
start Table{rowlen}.
```

This CSV specification represents grammars in which all rows have a single, fixed length (e.g. a grammar of 5-column CSV files). We might want a more flexible grammar that permits any row length up to some maximum, so long as all rows in each file are the same length. We could represent this using a *rule comprehension*, as below:

```
exists maxlen : nat
S -> [? Row{len} for len = 0 to maxlen].
start S.
```

Here, the brackets `[? ... for ... to ...]` act similarly to a list comprehension. They define one rule (`? Row{len}`) for each possible length up to `maxlen`.

Figure 3 presents our progress so far on defining a metagrammar for simple CSV formats. At the top, we have "imported" a couple of useful non-terminal definitions—definitions for `String` and `Number`. Users can write such definitions from scratch, but we have developed a modest library of them to facilitate quick construction of parsers for ad hoc data formats.

2.3 Ranking Grammars

Consider the following example data.

```
0,1,15,Hello world!
1,2,23,Programming
0,3,-2,rocks!
```

A human would probably claim that the first three columns should contain `Number`s, and the last one contains `String`s. However, if `Number`s can be `String`s then the column types could be `Number/String/String/String`, or some other combination. Without guidance, an algorithm will not know how to choose between potential grammars.

```
import String, Number

exists rowlen : nat.

S -> Table{rowlen}.

Table{len : nat} -> Row{len} ("\n" Row{len})*.

Sep -> ? ","  ? "|" ? ";".
  constraint(|Productions(Sep)| = 1).

Row{len : nat} ->
  ? if (len = 0) then Cell{len}
  ? if (len > 0) then Row{len-1} Sep Cell{len}.

Cell{i : nat} -> ? Number ? String.
forall (i:nat) :
  constraint(|Productions(Cell{i})| = 1)
```

Fig. 3. CSV Metagrammar, Version 1

```
Cell{i : nat} ->
  ? Number as Num{i}
  ? String as Str{i}.
forall (i:nat) :
  constraint (|productions(Cell{i}| = 1).

forall (i:nat) : prefer 1 Num{i}.
```

Fig. 4. A Metagrammar with Preferences

Saggitarius allows users to steer the underlying grammar induction algorithm towards the grammar of choice by expressing preferences for one grammar over another. Such preferences are expressed using `prefer` clauses, which have a similar syntax to `constraint` clauses. `Prefer` clauses assign integer *weights* to boolean formulas; the ranking of a synthesized grammar is the sum of the weights of all satisfied boolean formulas.

Figure 4 illustrates the use of preferences to force Saggitarius to infer `Number` cells when possible and `String` cells otherwise. For each i, the rank of the grammar is increased by 1 if the production `Num{i}` is included in the grammar. Since we have constrained each `Cell{i}` to have only one production, this leads Saggitarius to choose the `Number` rule whenever possible.

2.4 Emitting Grammar Information

Sometimes, users will make use of the entire inferred grammar (e.g. using it to build a parser or recognizer). In others, they may simply care about certain *features* of the grammar. For example, using a built-in CSV parsing library might require the user to describe the type of each column. We could configure Saggitarius to print such information based on boolean conditions using the `emit` syntax illustrated in Figure 5. This allows Saggitarius to be used as either a grammar inducer or a classifier (or both!) as needed.

```
Cell{i : nat} ->
  ? Number as Num{i}
  ? String as Str{i}.
forall (i:nat) :
  constraint (|productions(Cell{i}| = 1).

forall (i:nat) : emit "Row"+str(i)+":Num" if Num{i}.
forall (i:nat) : emit "Row"+str(i)+":String" if Str{i}.
```

Fig. 5. Example Emit Statements

3 FORMAL LANGUAGE

In this section, we formally describe the metagrammar specification language of Saggitarius, and describe how to interpret it using a relational semantics. The formal syntax is laid out in Figure 6a. We describe how to compile the full language introduced in §2 to this core calculus in §3.3.

```
Int Exp.      e  := nat | id | e - 1

Bool Exp.     b  := T | F | e = e
                   | e < e  | b && b
                   | ...

Nonterminal   N  := id{e}

Production    p  := N | str | p p

Cond. Prod.   c  := if b then p

Rule body     rb := c | c rb

Existential   ex := exists id

Rule          R  := id{id} -> ex.rb

Start symbol  S  := start N

Metagrammar   MG := S | ex; MG | R; MG
```

(a) The core grammar of Saggitarius

```
Production  p   := id | str
                 | p p

Productions ps  := [] | p ps

Rule        R   := id -> ps

Start       S   := start id

Grammar     G   := S | R G
```

(b) Syntax of a concrete grammar

Fig. 6

Our definition contains three built-in symbols: identifiers (`id`), natural numbers (`nat`), and strings (`str`). There are two types of expression: integer, on which the only allowed operation is subtracting 1, and boolean, which permit the standard boolean operations as well as integer comparisons.

Nonterminals N have the form described in §2.2: an identifier followed by a natural-number argument. For simplicity, we require each nonterminal to have exactly one argument, but it is straightforward to extend the system to cover any a number of arguments. Productions in Saggitarius are simply a sequence of terminals (i.e., constant strings) and nonterminals.

The body of each rule is a sequence of conditional productions, combining productions with boolean conditions. Unlike in §2, which included both mandatory and optional productions (beginning with `|` and `?`, respectively), the core grammar only uses mandatory, conditional productions.

A rule declaration `id {id'} -> exists id".rb` declares a nonterminal identifier `id`, which may be referenced in the rest of the program, as well as a name `id'` for its argument and a local existential variable `id"`, both of which may be referenced only in the rule body. We omit the type annotations on `id'` and `id"` for simplicity. Finally, a metagrammar is a list of rule and global existential variable declarations, terminating in a declaration of the start symbol.

An astute reader may notice that grammar in Figure 6a does not make reference to the constraints or preferences mentioned in §2. We encode all constraints using a combination of local and/or global existential variables, and guards on the conditional productions. User preferences can easily be formalized as functions from sets of rules to metrics. We do not give such preference functions a formal syntax and semantics in this section, but assume such functions exist in the following section. In practice, these preference functions are easily specified through the syntax used in §2.3.

3.1 Interpreting Metagrammars

A metagrammar represents a set of *concrete* (or *candidate*) grammars, which have the form described in Figure 6b. Intuitively, this set contains all possible combinations of rules, generated by all possible instantiations of each existential variable. We formally define which which grammars are denoted by a given metagrammar in the next section, via a relational semantics. However, it is illuminating to first consider a simple example metagrammar, and determine which grammars it corresponds to.

Consider the following stripped-down CSV grammar, which describes only a single row. Note that the variables i and y are unused, and would be omitted in an actual Saggitarius program.

```
Cell{i} -> exists x. // i unused
  | if (x = 0) then String
  | if (x <> 0) then Number.

Row{j} -> exists y. // y unused
  | if (j = 0) then Cell{j}
  | if (j > 0) then Row{j-1} Sep Cell{j}.

exists len.
start Row{len}.
```

Intuitively, obtaining a grammar from a metagrammar has three steps. Each step involves several *choices*, and different choices will result in different (though possibly equivalent) grammars. The first step is straightforward: we choose a value for each global existential variable.

The second step is to remove arguments to nonterminals by duplicating each nonterminal definition once for each possible argument. For example, we turn the definition of `Row{i}` into definitions for new nonterminals `Row_0`, `Row_1`, and so on. The choice made in this step is the number m of times we copy each definition. If we chose `len` $= 2$ and $m = 3$, we would transform the above grammar into the following:

```
Cell_0 -> exists x.
  | if (x = 0) then String
  | if (x <> 0) then Number.
Cell_1 -> // Same as Cell_0
Cell_2 -> // Same as Cell_0

Row_0 -> Cell_0
Row_1 -> Row_0 Sep Cell_1
Row_2 -> Row_1 Sep Cell_2

start Row_2.
```

Notice that m must be strictly greater than the value chosen for `len`, since `len` is used as an argument to Row. If we chose `len` $= m = 2$, we would end up with a nonexistent start symbol Row_2. In general, m must be strictly larger than any of the existential values chosen previously.

The final step is to choose values for each local existential variable; again, these values must be strictly less than m. In the example above, if we chose values $0, 2, 1$ for the existentials inside `Cell_0`, `Cell_1`, and `Cell_2` respectively, we would end with the following concrete grammar:

```
Cell_0 -> String
Cell_1 -> Number
Cell_2 -> Number

Row_0 -> Cell_0
Row_1 -> Row_0 Sep Cell_1
Row_2 -> Row_1 Sep Cell_2

start Row_2.
```

By making every possible choice in each of these three steps, we generate every grammar that is denoted by the metagrammar. This process is formalized in the next section.

3.2 Interpreting Metagrammars, but formally this time

Building upon the intuition from the previous section, we formally define the grammars G represented by a metagrammar MG using a relation $MG \Rightarrow_m G$. As before, the parameter m denotes the number of times each rule is copied. We can then define the set of grammars denoted by MG as

$$[\![MG]\!] = \bigcup_{m=1}^{\infty} \{G | MG \Rightarrow_m G\}.$$

The three rules of the $\Rightarrow_m$ relation follow much the same steps as the previous section; the only difference is that we fix m in advance, and use it as an upper bound for the value of *all* existential variables, global or local. We make use of an injective function $\lfloor\cdot\rfloor$ to translate the parameterized identifiers of the meta-grammar into the non-parameterized nonterminal identifiers of the candidate grammars. For example, one implementation of this function would yield $\lfloor$`foo{0}`$\rfloor$ = `foo_0`. We leave $\lfloor$`id{`e`}`$\rfloor$ undefined if e is not an integer value, and lift the function $\lfloor\cdot\rfloor$ to productions p by applying it individually to each identifier within p.

We define the relation $\Rightarrow_m$ below. In the process, we define a subordinate relation `rb` $\Rightarrow$ `ps` which relates rule bodies `rb` to production lists `ps`. We use the @ operator to denote list concatenation, and use the notation $[n/$`id`$]$ to denote capture-avoiding substitution of n for `id`. Finally, we use $\equiv$ to denote equivalence of expressions according to normal boolean and arithmetic rules.

$$\textsc{Start}\ \frac{e \equiv n \qquad \mathtt{id}' = \lfloor \mathtt{id}\{n\} \rfloor}{\mathtt{start}\ \mathtt{id}\{e\} \Rightarrow_m \mathtt{start}\ \mathtt{id}'} \qquad \textsc{Exist}\ \frac{n < m \qquad MG[n/\mathtt{id}] \Rightarrow_m G}{\mathtt{exists}\ \mathtt{id};\ MG \Rightarrow_m G}$$

$$\textsc{Unrolling}\ \frac{\begin{array}{c} n_0 \ldots, n_{m-1} < m \qquad MG \Rightarrow_m G \\ \mathtt{rb}[0/\mathtt{id}'][n_0/\mathtt{id}''] \Rightarrow ps_0 \quad \cdots \quad \mathtt{rb}[m-1/\mathtt{id}'][n_{m-1}/\mathtt{id}''] \Rightarrow ps_{m-1} \\ \mathtt{id}_0 = \lfloor \mathtt{id}\{0\} \rfloor \quad \cdots \quad \mathtt{id}_{m-1} = \lfloor \mathtt{id}\{m-1\} \rfloor \end{array}}{\mathtt{id}\{\mathtt{id}'\} \longrightarrow \mathtt{exists}\ \mathtt{id}''.\mathtt{rb}; MG \Rightarrow_m \mathtt{id}_0 \longrightarrow ps_0; \ldots; \mathtt{id}_{m-1} \longrightarrow ps_{m-1}; G}$$

$$\textsc{Cond-True}\ \frac{b \equiv \mathtt{True} \qquad \mathtt{p}' = \lfloor \mathtt{p} \rfloor \qquad e \equiv n \qquad \mathtt{id}' = \lfloor \mathtt{id}\{n\} \rfloor}{\mathtt{if}\ b\ \mathtt{then}\ \mathtt{p}\ \mathtt{as}\ \mathtt{id}\{e\} \Rightarrow [\mathtt{p}']} \qquad \textsc{Cond-False}\ \frac{b \equiv \mathtt{False}}{\mathtt{if}\ b\ \mathtt{then}\ \mathtt{p}\ \mathtt{as}\ \mathtt{id}\{e\} \Rightarrow []}$$

$$\textsc{Body}\ \frac{\mathtt{rb} \Rightarrow \mathtt{ps} \qquad \mathtt{c} \Rightarrow \mathtt{ps}'}{\mathtt{c}\ \mathtt{rb} \Rightarrow \mathtt{ps}\ @\ \mathtt{ps}'}$$

The START rule is the simplest; it simply checks that e is a value, and applies $\lfloor\cdot\rfloor$ to translate the metagrammar nonterminal to a concrete-grammar nonterminal. The EXIST rule is almost as simple; it chooses a particular value for an existential variable, and substitutes that value throughout the rest of the metagrammar.

Most of the complexity lies in the UNROLLING rule. For each argument i from 0 to m, we choose a value n_i for the local existential variable, and substitute both the the index i and n_i into the rule body. For each substituted body, we use the $\Rightarrow$ relation to obtain the list of corresponding productions $\mathtt{ps}_i$ and add the rule $\lfloor \mathtt{id}\{i\} \rfloor \longrightarrow \mathtt{ps}_i$ to the output grammar.

The $\Rightarrow$ relation is relatively simple: for each conditional production, we return either a singleton list `[p']` if the condition is true, or an empty list if the condition is false. For rule bodies with multiple productions, we concatenate these lists, ultimately resulting in a list containing exactly those productions whose conditions are true.

3.3 Syntactic Sugar by Example

As demonstrated in §2, Saggitarius includes a number of features to enable rapid development of complex metagrammars that do not appear in the core calculus, such as optional productions, constraints, and regular expressions. We informally illustrate here how to encode each of these features in our core calculus.

Optional Productions. Optional productions can be represented using local existential variables. For example, `A -> ? B ? C` can be represented as the following rule

```
A -> exists b.
     | if (b = 0 || b = 1) then B
     | if (b = 0 || b = 2) then C
```

Notice that depending on the choice of b, any combination of the two rules may be included in the grammar.

Regular Expression Productions. We compile regular expression syntax to context-free grammar rules in the usual way, by introducing new nonterminals and rules as appropriate. Collections of rules generated from the right-hand side of a single meta-grammar rule can be included or excluded as a group using global existential variables. For example, `A -> ? "b"*` may be compiled as follows, where g and Temp are fresh variables:

```
exists g.
A -> if (g = 0) then Temp.
Temp -> if (g = 0) then "".
Temp -> if (g = 0) then "b" Temp.
```

Constraints. The surface language of Saggitarius includes several different types of constraints, but all of them can essentially be encoding by enumerating the possible combinations of rules. For example, imagine we have the program

```
A -> ? B as r1                  D -> ? E as r3
     ? C as r2                       ? F as r4
                                     ? G as r5
```

We could encode the constraints that `A` has exactly one production, and `D` has at least two productions by creating local existential variables `b1` and `b2`, and enumerating the possible instantiations:

```
A -> exists b1.                 D -> exists b2.
     | if (b1 = 0)  then B           | if (b2 <> 0) then E
     | if (b1 <> 0) then C           | if (b2 <> 1) then F
                                     | if (b2 <> 2) then G
```

For something more complicated, consider the constraint `r1 => (r3 || r5)` which specifies that either `r3` or `r5` must be included whenever `r1` is. We can represent this cross-rule constraint using a *global* existential variable b, and again simply enumerate the cases:

```
exists b.                       D -> ? E ? F ? G
A -> ? C                             | if (b = 0 || b = 1) then E
     | if (b < 3) then B             | if (b = 0 || b = 2) then G
```

4 GRAMMAR INDUCTION ALGORITHM

The goal of Saggitarius' grammar induction algorithm is, given a set of positive and negative examples, a metagrammar MG and a ranking function $p : [[MG]] \to \mathbb{Z}$, to return the highest-ranked grammar contained in $[[MG]]$ which parses all the positive examples and none of the negative examples[2]. We have experimented with several possible algorithms that interleave parsing and constraint solving in different ways, and outline the most successful algorithm here.

Our grammar induction algorithm is formalized in Algorithm 1. At a high level, our strategy is rather simple. First, we concretize the metagrammar, removing arguments to nonterminals along with existential variables. Next, we determine all possible combinations of rules that successfully parse each example. We use these to create a logical formula asserting either that one of these combinations is included in the final grammar (for a positive example), or that none are included (for a negative example). For space, we elide the formal definitions of Concretize and Constraints and the proof of the following theorem. The interested reader can find them in the full version of the paper [Miltner et al. 2023].

Theorem 4.1. *If there is a Grammar G such that $G \in [[MG]]$, where $Ex^+ \subseteq \mathcal{L}(G)$ and $Ex^- \subseteq \overline{\mathcal{L}(G)}$, then* InduceGrammar$(MG, Ex^+, Ex^-, 0)$ *will return such a grammar.*

In Saggitarius, the ranking function p is constructed piecewise from various `prefer` statements in the surface language, each consisting of an integer weight and a boolean combination of rules. The rank of a grammar is simply the sum of all weights whose boolean combinations are satisfied. These

[2]If several of these grammars have the same rank, any of them may be returned.

```
procedure InduceGrammar($MG, Ex^+, Ex^-, k$)
    $G \leftarrow$ Concretize($MG, k$)
    $\phi \leftarrow$ Constraints($MG, k$)
    $\phi^+ \leftarrow \bigwedge_{ex \in Ex^+}$ SemiringParse($G, ex$)
    $\phi^- \leftarrow \bigwedge_{ex \in Ex^+} \neg$SemiringParse($G, ex$)
    $ret \leftarrow$ SMT($\phi^+ \wedge \phi^- \wedge \phi$)
    match $ret$ with
        | Some($vs$) $\longrightarrow$ return $MG[vs]$
        | None $\longrightarrow$ return InduceGrammar($MG, Ex^+, Ex^-, k + 1$)
```

Algorithm 1. Syntax-Guided Grammar Induction Algorithm. MG is the provided metagrammar, Ex^+ is the provided positive examples, Ex^- is the provided negative examples, and k is the search depth (initially 0).

preferences are translated into logical formulas, combined with the formulas for each example, and shipped to Z3 [de Moura and Bjørner 2008], which uses its MaxSMT algorithms to find a preference-optimal solution.

For simplicity, we assume there are no variables in our start symbol.

4.1 Concretizing Metagrammars

Before we can apply a conventional parsing algorithm to determine which combinations of rules parse a given example, we must first eliminate Saggitarius' unconventional constructs – namely, nonterminals with arguments, and existential variables. This process closely mirrors the one described in §3.1. We begin by choosing a maximum value for each global variable (corresponding to a particular choice of m). We then copy each rule that number of times, and expand out *all possible* values for each global and local existential variable.

This process generates a finite subset of $[\![MG]\!]$, obtained by taking the union only up to our chosen value of m. We then apply the concrete parsing algorithm described below; if it fails to find a suitable grammar, we begin the process anew with a larger value of m. In general, this process is not guaranteed to terminate, but termination *can* be guaranteed for certain types of grammar.

Optimizations. Our construction of $[\![MG]\!]$ contains significant redundancy. For one, different values of existential variables may not actually lead to different grammars. Similarly, we need not copy *every* rule the same number of times, since this may result in rules which are unreachable from the start symbol.

Saggitarius leverages these observations by allowing users to declare variables with a range type instead of type `nat`; for example, the line `exists x : [0, 4]` declares an existential variable whose value is constrained between 0 and 4, inclusive. By using these types, users can both express their intent for variables to be restricted, and provide the compiler with hints to avoid checking redundant grammars. Furthermore, Saggitarius copies rules only as much as necessary, ensuring that no unreachable rules appear in the concretized metagrammar.

Notice that metagrammars in which all variables are bounded are *finite*, and hence permit an exhaustive search of candidate grammars. This allows us to strengthen our termination conditions: **Induction for Saggitarius metagrammars is guaranteed to terminate whenever all variables in the program are bounded.**

4.2 Grammar Induction Via Semiring Parsing

Once we have generated a large, ambiguous grammar from a metagrammar, we must identify a subset of rules in that grammar that parse all positive examples, do not parse any negative examples, and satisfy the user-provided constraints.

For example, consider the following grammar:

```
{x_1} X -> x
{x_2} X -> x
{y_1} Y -> y
{y_2} Y -> y
{z_1} Z -> z
{z_3} Z -> z

G -> XG
G -> YG
G -> Z
```

Consider the example string "xyz". This could be parsed using rules x_1, y_1, z_1 or x_1, y_1, z_2 or x_1, y_2, z_1, and so on. A naïve algorithm could generate all combination of rules, filter out those that do not satisfy the user-provided constraints, and iteratively test them until the it finds a set of rules that accept the positive examples but reject the negative examples. Unfortunately such an approach is not tractable in practice – as there are can be an exponential number of different rule combinations for parsing the same input string. Even in this simple example there are 2^3 possible combinations for only a single string.

We address this problem by representing the possible rule combinations as a logical formula. For the example string above, the possible rule sets are represented by the formula $(x_1 \vee x_2) \wedge (y_1 \vee y_2) \wedge (z_1 \vee z_2)$. Rule names $x_1 \dots z_1$ are interpreted as boolean variables; satisfying assignments correspond to collections of those rules whose variable is `true`. Any satisfying assignment for this formula will yield a set of rules which suffice to parse the example.

More broadly, we encode the possible parse trees for every example as such logical formulas. We then create a single formula as a conjunction of the formulas for each positive example, the negation of the parse formulas for each negative examples, and the user-provided constraints. Finally, we pass this formula to an off-the-shelf SAT solver. As with many practical problems, SAT solvers can typically find satisfying assignments for these formulas in much less than exponential time.

Generating The Formulas. The process of generating formulas can be implemented using semiring parsing [Goodman 1998, 1999]. In particular, we consider logical formulas to be elements in a conditional-table semiring [Green et al. 2007], where logical disjunctions and conjunctions correspond to the semiring's + and × operators.

Satisfying the Formulas. Once provided to an SMT solver, the exponential number of possible rules does not diminish. However, this is the bread-and-butter of an SMT solver. In the past, most DNF formulas would incur an exponential blowup when given to an SMT solver; however, modern SMT solvers can efficiently find satisfying assignments for such formulas.

Figure 7 illustrates our process on a simple example. In the upper left, we present five candidate rules (numbered 0-4) that define nonterminals A, B and C. Rules 1 and 2 are identical in this example; they help illustrate the ambiguity the algorithm must handle. In the upper right, Figure 7 presents two example parse trees, one for the example “abbb”, and one for “abbc”. Each edge of the parse

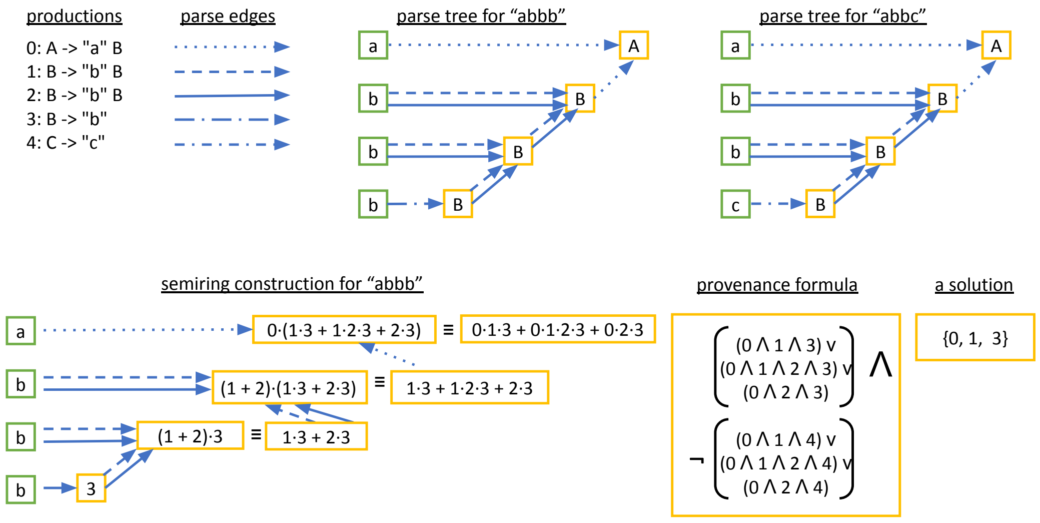


Fig. 7. Semiring parsing and provenance formula construction

tree is associated with a particular rule; for instance, rule 1 is denoted by a dashed edge and rule 2 is denoted by a solid edge.

On the lower left-hand side, Figure 7 presents the strategy that the parser implements to compute a semiring expression for the example "abbb" (the computation for "abbc" is similar). These expressions represent the possible sets of rules that can parse each example. To begin, the final "b" of "abbb" can only be obtained via rule 3, which corresponds to the algebraic expression 3. The second-to-last "b" character could be obtained either with rule 1 or rule 2, represented as the expression $1 + 2$. Hence, to produce a string ending in "bb", we must have either rule 1 or 2, and also rule 3. We can represent this by conjoining our two expressions to get $(1 + 2) \cdot 3$, or equivalently $1 \cdot 2 + 1 \cdot 3$.

Continuing on, the third-to-last "b" could also be obtained via rule 1 or rule 2, so we once again conjoin the expression $1 + 2$ with our accumulator expression to get $(1 + 2) \cdot (1 \cdot 3 + 2 \cdot 3)$. Since the conditional table semiring is commutative and idempotent[3], this is equivalent to $1 \cdot 3 + 1 \cdot 2 \cdot 3 + 1 \cdot 2$. Finally, the first "a" character could only be obtained from the start symbol A using rule 0, so we end up with a final expression of $0 \cdot (1 \cdot 3 + 1 \cdot 2 \cdot 3 + 2 \cdot 3)$, or equivalently $0 \cdot 1 \cdot 3 + 0 \cdot 1 \cdot 2 \cdot 3 + 0 \cdot 2 \cdot 3$.

To finish the process, we must translate our example semiring expressions into logical formulas. Fortunately, this is easy—our intuition for the semiring expressions already told us that + corresponds to disjunction and $\cdot$ to conjunction. Simply replacing the connectives transforms our semiring expression into a logical one.

If we take "abbb" to be a positive example and "abbc" to be a negative example, we can obtain an overall provenance formula by translating both examples into semiring expressions, then to logical formulas, and conjoining them. Since "abbc" is negative, we negate the corresponding conjunct, indicating that *none* of the sets of rules that can parse "abbc" should be included. We then pass the provenance formula to an SMT solver (specifically, Z3), to obtain a solution; in this case, one possible solution is the set of rules $\{0, 1, 3\}$. If we had included preferences in our example, we would use Z3's MaxSMT algorithms to obtain the solution with the highest rank.

[3]Intuitively, the expression $1 \cdot 1 \cdot 3$ represents the set containing rule 1, rule 1, and rule 3; this is of course equal to the set containing rule 1 and rule 3, represented by $1 \cdot 3$. For similar reasons, $1 \cdot 2 \cdot 3 \equiv 2 \cdot 1 \cdot 3$.

SGI Benchmark Suite				
Name	Nonterms.	Prods.	AST Nodes	Description
States	6	9	371	US State identifiers. Permits acronyms, full names, and abbreviations.
Phone #s	11	44	260	Phone numbers. Permits local and international phone numbers.
Times	12	21	204	Time of day in a variety of formats.
Floats	12	18	113	Floating-point numbers. Includes grammars for different scientific notations and standard decimal form.
Emails	15	105	493	Email addresses. Includes grammars that accept emails from specific or arbitrary domains.
Names	11	32	142	Human identifiers. Includes grammars specifying salutations, post-nominal titles, and acronyms.
Streets	14	40	167	US street identifiers. Includes grammars that demand specific suffixes and directions.
Dates	18	37	314	Calendar dates. Includes month-first, day-first, and year-first formats.
Addresses	28	58	597	US street addresses. Uses the States and Streets metagrammars to identify those portions of the address.
XML	21	91	523	XML Files. Permits 10 classes of XML elements. It discovers the identifiers for element classes and the recursive schemes. In effect, it imputes the structural component of a schema definition.
SQL	25	31	357	SQL: SQL SELECT statements. Supports nested joins and extra keyword clauses like WHERE and LIMIT.
IdealCSV	38	61	506	CSV: Idealized version of CSV. Based on RFC 4180 [Network Working Group 2005], but also admits common kinds of separators including tab and semi-colon. Automatically infers cell type.

Fig. 8. Information on metagrammars for the SGI benchmark suite. For each, we include the number of nonterminal definitions, the total number of productions, the total number of AST nodes.

5 EXPERIMENTAL RESULTS

To illustrate Saggitarius 's practicality, we evaluate the following properties:

- **Expressiveness** Can Saggitarius specify real-world grammatical domains?
- **Time Efficiency** How much time does Saggitarius take to induce a grammar?
- **Sample Efficiency** How many examples does Saggitarius take to induce a specific grammar?
- **Comparisons** How does Saggitarius compare to prior work?

Experimental setup. All experiments in this section were performed on a 2.5 GHz Intel Core i7 processor with 16 GB of 1600 MHz DDR3 RAM running macOS Catalina. We ran each benchmark 10 times, with a timeout of 60 seconds, and report the average time. If any of the 10 runs times out then we consider the benchmark as a whole to have timed out.

5.1 Expressiveness

We began our evaluation by identifying 12 grammatical domains to use in our experiments. We intentionally chose domains that range from simple (dates, times) to complex (XML, CSV). For each domain, we manually searched the internet to identify various formats that were used in practice. Our sources included actual data files, as well as documents containing descriptions of formats.

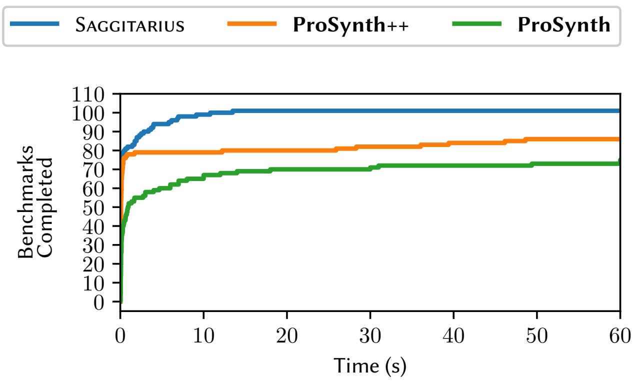


Fig. 9. Number of benchmarks that terminate in a given time in different modes. The **ProSynth** line represents using the ProSynth algorithm, originally built for learning logic programs, for grammar induction. The **ProSynth**++ line represents using the ProSynth algorithm, but with an optimization that minimizes duplicate parsing calls.

We then *manually* constructed a corpus of example files adhering to each format. As a result, our examples are somewhat artificial—we wrote them ourselves—but nonetheless represent a broad range of real-world formats.

Once we had conducted our survey of real-world data formats, we used that experience to write metagrammars for each domain. Figure 8 (in the body of the paper) describes these domains briefly, as well as the complexity of each. Qualitatively, we did not find these grammars particularly hard to write—the domains do not require many productions and most of the insight was in what preferences and constraints were necessary. On occasion, our initial domain definitions were erroneous, in the sense that they did not return the "best" grammar for certain examples. However, we found such errors were relatively easy to fix by adjusting preferences. Since we were able to successfully write metagrammars for each domain, we conclude that SAGGITARIUS *is* capable of representing real-world domains.

5.2 Time Efficiency

Benchmarks. To evaluate the efficiency of SAGGITARIUS, we developed a benchmark suite of 10 induction tasks, each of which ask SAGGITARIUS to induce a grammar given a certain number of positive examples (PEs) and negative examples (NEs). Each benchmark varies the number of PEs and NEs, as well as whether or not a suitable grammar exists in the metagrammar. Specifically, our benchmarks are to induce a grammar given:

(1) 1 PE.
(2) 1 NE.
(3) 1 PE and 1 NE.
(4) 10 PEs.
(5) 5 PEs and 5 NEs.
(6) 1 PE and 20 NEs.
(7) 20 PEs and 20 NEs.
(8) 1 PE and 1 NE, when no appropriate grammar exists.
(9) 10 PEs and 10 NEs, when no appropriate grammar exists.
(10) 20 PEs and 1 NE, when no appropriate grammar exists.

In this experiment, we are are not asking SAGGITARIUS to return a particular grammar; rather, we are evaluating how long it takes to return *some* grammar (or "no grammar exists", for the latter three tasks). We evaluate SAGGITARIUS's ability to return specific grammars in the next section.

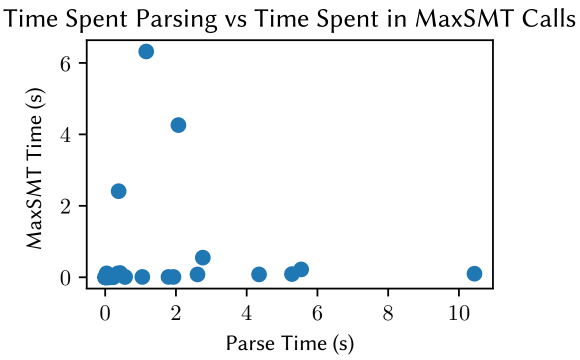
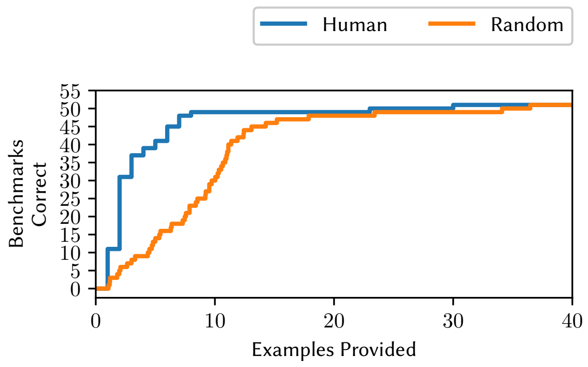


Fig. 10. The time spent during the parsing phase of the benchmark suite compared to the time spent during the MaxSMT solving phase of the benchmark suite. Aside from a few outliers, the majority of the time is spent during parsing.

For each task and each grammatical domain in Figure 8 except SQL (due to time constraints), we selected a set of examples from our corpus to create a total of 110 benchmarks. We evaluated Saggitarius by running it on these 110 benchmarks using three different algorithms, meant to simulate the performance of related tools. We recorded how long each benchmark took to finish, and summarize the results in Figure 9. This figure is also used later in Section 5.4. We find that Saggitarius solves 102 of our 110 benchmarks in under a minute, and typically does so in under 10 seconds.

Figure 10 shows how much time was spent for each benchmark in the two main phases of our algorithm: parsing and performing MaxSMT calls. In a majority of these benchmarks, parsing dominates the runtime, and all but one of the failing benchmarks hang during parsing. Saggitarius uses an Earley parser [Earley 1970; Vaillant 2020] modified to operate over semirings, though other general context-free parsing algorithms, such as a GLR algorithm [Nozohoor-Farshi 1991], would also have been suitable. We believe that further developments in semiring parsing could substantially increase the efficiency of Saggitarius on many benchmarks. For example, recent work has found that semirings with rich sets of equivalences (like the idempotence of + and × in the conditional table semiring) provide additional opportunities for memoization in the parsing algorithm [Herman 2020]. Furthermore, in some simpler domains like Dates and Phone Numbers, every grammar in the domain falls into smaller complexity classes than merely context-free, like LL(*). By integrating the faster parsing algorithms known to work on these simpler domains, induction could likely happen dramatically faster.

However, in one domain MaxSMT, rather than Early parsing, dominates the runtime – XML. For each example in the XML domain, a majority of the time is spent during the MaxSMT call, and one XML benchmark fails to complete the MaxSMT call in the allotted timeframe. This is because the MaxSMT algorithm has a difficult time proving a given model is optimal in the XML domain. If we replace the MaxSMT call by a simple SMT call (equivalent to removing preferences from our source file), these benchmarks finish near instantly. We perform a more involved case study on the XML domain in the full version of the paper. We believe that developments into more efficient MaxSMT algorithms could increase the efficiency of Saggitarius without sacrificing optimality.



(a) # Correct benchmarks vs. # examples used.

Name	Human	Random
States	2	6.5
Phone #s	3	10.3
Times	4.5	12.4
Floats	1.8	6.2
Emails	2	5.9
Names	6.5	12.5
Dates	4	10.6
Addresses	8	13.9
IdealCSV	1	3

(b) Average # examples needed per domain

Fig. 11. Sample efficiency of Saggitarius. Human examples were chosen to be as helpful as possible, in the human's judgement.

5.3 Sample Efficiency

One of the advantages of Saggitarius is its ability to successfully induce useful grammars from very few example. To demonstrate this, we designed an experiment to measure how many examples it takes to induce a particular grammar, or correctly claim that no suitable grammar exists.

Our setup is as follows: for each terminating benchmark that involves at least 5 examples, run Saggitarius on a single example. If the output is not the desired one, select another example, and run Saggitarius on both. Continue adding examples until the output matches our expectation.

Figure 11 summarizes our results using two different selection strategies. In the first, we have a human operator select examples they think will be the most useful. In the second, we select each example at random. We can clearly see from Figure 11a that nearly all benchmarks can be completed with fewer than 10 well-chosen examples. Even when examples are chosen at random, Saggitarius almost never needs the entire set of examples, settling on the desired grammar with between 10 and 20 inputs.

Figure 11b breaks down the data according to grammatical domain. Intuitively, grammatical domains where the grammars have a lot of "overlap" are generally harder to specify. For example, a state can be represented as with its full name, the two-letter abbreviation, a historic acronym, or some combination of the three. From a single example, it will not be clear if only one representation is acceptable, or if multiple types are allowed.

When domains are combined, the ambiguity of the components compounds. For example, addresses must determine not only which state representations are allowed, but whether street names are written fully (e.g. "Street"), abbreviated ("St."), or omitted entirely. The decisions for states, streets, and other components such as names must be made separately, and it is difficult to constrain all of them in a single example, particularly when your desired address format allows several possibilities for each.

We omitted the XML domain from this experiment because the XML meta-grammar can generate several syntactically different, but semantically equivalent grammars. As a consequence, determining whether the tool generates the correct grammar automatically is much more difficult than just running `diff`. Rather than attempting to implement equivalence checking, we opted to skip the XML domain for this experiment.

5.4 Comparisons

Saggitarius is relatively unique in the problem it solves; there are few existing tools which do the same thing. The closest analogue we know of is the inductive logic programming engine *ProSynth*. Since parsing via context-free grammar is a refinement of logic programming, the ProSynth algorithm may be effective. Unfortunately, the process of transforming a Saggitarius program into a ProSynth program proved impractical. Instead, we compare *strategies* by implementing two additional induction algorithms for Saggitarius, which simulate ProSynth's process. We can then compare the performance of these algorithms directly.

The three stratgies we evaluate are:

- Saggitarius: The algorithm described in §4.
- **ProSynth**: In this mode, Saggitarius iteratively guesses subsets the candidate rules, and checks to see whether the subset parses the positive but not the negative examples. If it fails, it uses a counter-example-guided algorithm to generate a new set of candidate rules and repeats the process. If it is able to guess an appropriate set early, it may avoid the heavy cost of parsing all data with a large ambiguous grammar.
- **ProSynth++**: In this mode, Saggitarius parses all the example data, and constructs a full provenance tree as in Saggitarius. However, instead of building a single large MaxSMT formula, it considers candidate rule sets in a counter-example-guided loop, as in **ProSynth**. This mode may avoid constructing a large SAT formula, at the cost of making many SAT calls.

Results. We can see from Figure 9 that our primary algorithm (Saggitarius) completed 101 out of 110 benchmarks in under 20 seconds. In 6 of the 9 instances the system timed out, it did so because no grammar matched the example data (by experiment design) and Saggitarius looped, forever trying larger and larger values of its existential variables. A useful improvement to the system would be to implement heuristics that can detect when no grammar in an infinite metagrammar will satisfy a data set. The remaining 3 of the 110 experiments ran long due to particularly slow MaxSMT calls. Removing all preferences drastically speeds up these tasks, allowing them to complete in under a minute (though the returned grammars are not the desired ones).

In comparison to the ProSynth modes, we found that Saggitarius solved every benchmark but one faster than both **ProSynth** and **ProSynth++**. The exception was one of the benchmarks on which Saggitarius timed out on an SMT call. Although the multiple smaller SMT calls of **ProSynth++** usually take longer in aggregate, in this one case they avoided generating a particular challenging SMT formula.

More broadly, the cost of all of the algorithms grows substantially with the number of rules in the metagrammar. The ProSynth-inspired algorithms, in particular, suffer significantly as rule sets increase because it takes them more iterations to build up a sufficiently constrained SAT call to find a solution. Saggitarius is also negatively impacted by having more rules, but in a different (and less substantial) way: more rules results in slower parsing, which is the typical bottleneck for Saggitarius (complex MaxSMT calls are most costly, but much rarer).

Across all the algorithms, we found that the *number* of examples did not affect system performance substantially. However, the *length* of an example can have a significant impact when the metagrammar is highly ambiguous. Earley parsing with large numbers of ambiguous rules is a costly enterprise that can grow cubically with the length of the input in the worst case. This parsing cost tends to dominate as examples grow in length.

5.5 Case Study: CSV Dialect Detection

As part of our comparisons, we describe our case study which compares Saggitarius to a domain-specific tool on an individual grammatical domain: CSV. We have performed additional case studies on the XML and SQL grammatical domains, but have elided them for space, the interested reader can find them in the full version of the paper. RFC 4180 provides the standardized definition of CSV, but there are many real-world data files that do not conform to the standard. Such data files may contain irregular “table-like” data in which rows have differing numbers of columns, cells are malformed (containing dangling delimiters), and the entire table is surrounded by unstructured text.

To handle such messy CSV-like data, Van den Burg *et al.* [van den Burg et al. 2019b] designed CSV Wrangler, a tool for inferring the *dialect* of messy, possibly erroneous CSV documents [4]. Unfortunately, CSV files are inherently ambiguous. To handle this ambiguity, Van den Burg *et al.* use a custom algorithm that scores potential dialects based on how consistent the resulting rows are (the *pattern score*) and how well-typed the cells are (the *type score*) [van den Burg et al. 2019b], attempting to mimic the human process of recognizing the data in a messy CSV file.

To experiment with CSV dialect detection using Saggitarius, we developed two additional CSV metagrammars, named *Strict* and *Lax*. The former is more restrictive, hewing closer to the RFC, while the latter attempts to mimic the same human priors as Van den Burg’s tool.[5] As a result, Lax is much costlier to execute. Fortunately, because Lax is more lenient than Strict, the two processes can be pipelined — we try Strict first, and fall back to Lax only if it fails. We call this pipelined system StrictLax.

Benchmarks. We measured the performance of Saggitarius on 100 ASCII-128 CSVs with human labels drawn from Van den Burg *et al.*’s GitHub repo [van den Burg et al. 2019a]. More than half of the benchmark suite (59/100) does not obey the CSV RFC. In such situations, it is difficult for humans to unambiguously identify the dialect of files, as we discuss below.

Experiments. We attempted dialect detection using 5 different tools: (1) RFC (the Saggitarius specification of the CSV RFC), (2) Strict, (3) StrictLax (Strict followed by Lax), (4) CSV Wrangler (Van den Burg’s tool), and (5) the Python Sniffer [Python Software Foundation 2020]. We found that Earley parsing was the primary bottleneck for Saggitarius system on large CSV benchmarks, so we truncated the CSV files to 20 lines prior to processing them with Strict or RFC, and to 5 lines prior to processing them with Lax[6].

Figure 12 presents the results from our tool as well as corresponding results we retrieved from Van den Burg *et al.*’s GitHub repository [van den Burg et al. 2019a] for CSV Wrangler and Python Sniffer. We find that Saggitarius performs only slightly worse than dedicated CSV sniffers – in particular, the number of incorrectly classified files is nearly identical across all the tools. Furthermore, we believe that all but 2 of our misclassifications represent reasonable behavior by Saggitarius, even if the human classifier chose a different dialect. The misclassifications are described in Figure 13.

Unlike dedicated sniffers, Saggitarius must contend with the possibility of timing out, either due to infinite metagrammars or simply slow analysis. Although we see Saggitarius timing out on a noticeable fraction of examples, profiling indicates that much of this time is spent performing Earley parsing. It is likely that we could significantly improve our performance by adopting an industrial-strength, optimized parser in place of our current, unoptimized one.

Takeaways. First, we have shown that it is possible to develop more than one specification for an ambiguous grammatical domain. Multiple generated tools can then be arranged in a pipeline

[4]A CSV dialect is a triple of a cell delimiter, a quote character (for strings), and an escape character.
[5]Specifically, Strict requires that (1) nested quote characters do not appear in any cell and (2) delimiters (commas, semicolons, tabs and vertical bars) internal to a cell must be quoted. Lax contains neither restriction.
[6]While this may seem extreme, one would expect *all* lines of a CSV file to obey the same format. Hence a very small number of lines is still likely sufficient to infer the dialect.

Detector	Yes	No	No Dialect	Timeout
RFC	41	14	44	0
Strict	67	12	8	13
StrictLax	70	14	0	16
CSV Wrangler	86	13	1	N/A
Python Sniffer	81	14	5	N/A

Fig. 12. CSV Analysis. We report the number of files on which the tool: aligns with the human label (**Yes**); does not align with the human label (**No)**; reports that no dialect exists (**No Dialect**); or times out (**Timeout**).

Id	Saggitarius behavior	Frequency
1**	CSV preferences allowed algorithm to return no delimiter character	4
2*	chose different delimiter than hand-label but file was ambiguous.	2
3*	did not find any characters because of leading metadata	2
4	did not find quote or escape character because of truncation	2
5*	did not accept space as a valid delimiter	1
6**	did not identify quote in ill-quoted file	1
7**	unquoted "," character lead to false "," delimiter	1
8*	did not find improperly used escape character	1

Fig. 13. Reasons for StrictLax misclassification. In our analysis, cases marked * are inherently ambiguous or mislabelled by the human. Cases marked ** also represent reasonable behavior by our system in our judgement.

and take advantage of time/accuracy tradeoffs. Second, real-world data is often messy. Hand-tuned, custom tools such as Python's CSV sniffer or the CSV Wrangler can make mistakes; even humans often disagree with each other. Nevertheless, Saggitarius performs comparably to these custom-built tools. The CSV sniffer and the CSV Wrangler align with outside human labeller a few more times than our tool, but the authors of this paper actually disagree with the outside labels in almost all such cases. Perhaps the labeller and tool authors know of some criteria for disambiguating CSV data that we do not; if so, we can likely add this criteria to our own specifications. When it comes to the unambiguous formats where humans agree, Saggitarius is very effective.

6 RELATED WORK

Grammar induction. Grammar induction traces back to at least the 60s when Gold [1967] began studying models for language learning and their properties. Later, Angluin [1978, 1987] developed her famous L* algorithm for learning regular languages. As mentioned earlier, however, such algorithms often require large numbers of examples even for simple regular expressions. More recently, FlashProfile [Padhi et al. 2018] has shown that regular-expression-like *patterns* can be learned from positive examples, by first clustering by syntactic similarity, and then inducing programs for given clusters.

Inference of context-free grammars is considerably more difficult, and results are limited. It has been tackled, for instance, by Stolcke and Omohundro [1994], who use probabilistic techniques to infer grammars. Fisher et al. [2008] explored inference of grammars for "ad hoc" data, such as system logs, in the context of the PADS project [Fisher and Walker 2011]. Lee et al. [2016] developed more efficient search strategies for regular languages in the context of a tool for teaching automata theory. Both these latter tools tackled restricted kinds of grammars, however; scaling to complex formats using few examples remains a challenge. A more recent approach can be seen in the Glade [Bastani et al. 2017] tool. Similarly to L^*, Glade uses an active learning algorithm, but generalizes to full context-free grammars, rather than merely regular expressions, while requiring relatively fewer membership queries.

We believe that the key contributions of this paper are largely orthogonal to these advances. In particular, we introduce the idea of using *grammatical domains*, specified via metagrammars, to restrict the set of grammars under consideration during the induction process. Doing so has the potential to improve the performance of almost any grammar induction algorithm.

Grammatical inference becomes more tractable when one can introduce bias or constraints—metagrammars are one way to introduce such bias but there are others. For instance, Chen et al. [2020] use a combination of examples and natural language to speed inference of a constrained set of regular expressions. Internally, their system generates an "h-sketch" as an intermediate result. These h-sketches are partially-defined regular expressions that may include holes for unknown regular expressions. Such h-sketches play a similar role to our metagrammars: they denote sets of possible regular expressions and constrain the search space for grammatical inference. However, our language is an extension of YACC and is designed for humans, rather than being an intermediate language. Furthermore, our metagrammars may be reused, like libraries, across data sets. In constrast, each h-sketch is generated and used only once inside a compiler pipeline.

Related to the notion of grammatical inference is that of expression *repair*. RFixer [Pan et al. 2019] uses positive and negative examples to fix erroneous regular expressions. Both RFixer and Saggitarius use similar algorithms to ensure positive examples are in the generated language, and negative examples are not. Both of these tools encode these constraints as MaxSMT formulas to ensure the generated grammars are optimal. Because RFixer does not have a metagrammar to orient the search, their constraints can only help find character sets that distinguish between the grammars. Saggitarius permits any constraints that expressible in propositional logic, and the constraints can be over arbitrary productions, not merely character sets. One could see the RFixer algorithm as an instance of our algorithm, where the meta-grammar constrains sets of allowed characters.

Syntax-guided Program Synthesis. Our work was inspired by the progress on syntax-guided program synthesis over the past decade or so [Alur et al. 2013, 2018; Solar-Lezama et al. 2005, 2006]. Much of that work has focused on data transformations, including spreadsheet manipulation [Barowy et al. 2015; Gulwani 2011; Wang et al. 2017b], string transformations [Miltner et al. 2017, 2019; Wang et al. 2017a], and information extraction [Le and Gulwani 2014; Raza and Gulwani 2017]. Such problems have much in common with our work, but they have typically been set up as searches over a space of program transformation operations rather than searches over collections of context-free grammar rules. Particularly inspiring for our work was the development of FlashMeta [Polozov and Gulwani 2015]. FlashMeta is a "meta" program synthesis engine—it helps engineers design program synthesis tools for different domain-specific languages. Similarly, Saggitarius is a "meta" framework for syntax-guided grammar induction, helping users perform grammar induction in domain-specific contexts. Of course, Saggitarius and FlashMeta differ greatly when it comes to specifics of the language/system designs and the underlying search algorithms implemented.

Logic Program Synthesis. We were also inspired by work on Inductive Logic Programming [De Raedt 2008], and Logic Program Synthesis [Raghothaman et al. 2019; Si et al. 2019]. Parsing with context-free grammars is a special case of logic programming so it was natural to investigate whether inductive logic programming algorithms would work well here. ProSynth [Raghothaman et al. 2019] is a state-of-the-art algorithm in this field so we experimented with it as a tool for grammatical inference. However, we found our custom algorithm almost always outperformed ProSynth on grammatical inference tasks.

7 CONCLUSION

Grammatical domains are sets of related grammars. Such domains appear naturally whenever a common datatype like a date or a phone number has multiple textual representations. They also appear frequently when data sets are communicated via ASCII text files, as is the case for CSV files. In this paper, we introduce the concept of grammatical domains, provide a variety of examples of such domains in the wild, and design a language, called Saggitarius, for defining grammatical domains through the specification of metagrammars.

Saggitarius includes features for defining sets of candidate productions, for constraining the conditions under which candidate productions may and may not appear, and for ranking the generated grammars. We illustrate the use of Saggitarius on a variety of examples and develop a grammar induction algorithm for the system that uses semiring parsing to generate MaxSMT formulas that can be solved via an off-the-shelf theorem prover.

In the future, we look forward to developing a complete parser generator system using the ideas developed in Saggitarius. One way forward would be to add semantic actions to Saggitarius, making it much more like YACC. Another direction would involve integrating Saggitarius into a parser combinator library such as Parsec [Leijen et al. 2023].

ACKNOWLEDGMENTS

Thank you to our reviewers for their helpful comments. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under the SafeDocs program (2019-020, PRIME DARPA BAA HR001118S0054). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (nov 2018), 84–93. https://doi.org/10.1145/3208071

Dana Angluin. 1978. On the complexity of minimum inference of regular sets. *Information and Control* 39, 3 (1978), 337–350. https://doi.org/10.1016/S0019-9958(78)90683-6

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (nov 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 218–228. https://doi.org/10.1145/2737924.2737952

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Sergey Bratus. 2020. https://www.darpa.mil/program/safe-documents.

Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. https://doi.org/10.14722/ndss.2016.23483

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Luc De Raedt. 2008. Logical and Relational Learning. In *Advances in Artificial Intelligence - SBIA 2008*, Gerson Zaverucha and Augusto Loureiro da Costa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–1. https://doi.org/10.1007/978-3-540-88190-2_1

Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

Laura Firoiu, Tim Oates, and Paul R. Cohen. 1998. Learning Regular Languages from Positive Evidence. In *Twentieth Annual Conference of the Cognitive Science Society*. 350–355.

Kathleen Fisher. 2009. Pads Manual: Appendix B All Pads Base Types. https://web.archive.org/web/20230801220918/https://pads.cs.tufts.edu/doc/base_types_appendix.html

Kathleen Fisher and David Walker. 2011. The PADS Project: An Overview. In *Proceedings of the 14th International Conference on Database Theory (ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 11–17. https://doi.org/10.1145/1938551.1938556

Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 421–434. https://doi.org/10.1145/1328438.1328488

P. Garcia and E. Vidal. 1990. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12, 9 (1990), 920–925. https://doi.org/10.1109/34.57687

E Mark Gold. 1967. Language identification in the limit. *Information and Control* 10, 5 (1967), 447–474. https://doi.org/10.1016/S0019-9958(67)91165-5

Joshua Goodman. 1998. Parsing Inside-Out. , 19–28 pages. https://dash.harvard.edu/bitstream/handle/1/24829603/tr-07-98.pdf?sequence=1

Joshua Goodman. 1999. Semiring Parsing. *Comput. Linguist.* 25, 4 (Dec. 1999), 573–605.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Grzegorz Herman. 2020. Faster General Parsing through Context-Free Memoization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3385412.3386032

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

Daan Leijen, Paolo Martini, and Antoine Latter. 2023. parsec: Monadic parser combinators. https://web.archive.org/web/20230825034450/https://hackage.haskell.org/package/parsec.

Ke Liu. 2017. Dig Into the Attack Surface of PDF and Gain 100+ CVEs in 1 Year. https://web.archive.org/web/20230825034743/https://www.blackhat.com/docs/asia-17/materials/asia-17-Liu-Dig-Into-The-Attack-Surface-Of-PDF-And-Gain-100-

CVEs-In-1-Year-wp.pdf.

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* 2, POPL, Article 1 (dec 2017), 30 pages. https://doi.org/10.1145/3158089

Anders Miltner, Devon Loehr, Arnold Mong, Kathleen Fisher, and David Walker. 2023. Saggitarius: A DSL for Specifying Grammatical Domains. arXiv:cs.PL/2308.12329

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proc. ACM Program. Lang.* 3, ICFP, Article 95 (jul 2019), 28 pages. https://doi.org/10.1145/3341699

Network Working Group. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://web.archive.org/web/20230825033805/https://datatracker.ietf.org/doc/html/rfc4180. Request for Comments 4180.

Rahman Nozohoor-Farshi. 1991. *GLR Parsing for ϵ-Grammers.* Springer US, Boston, MA, 61–75. https://doi.org/10.1007/978-1-4615-4034-2_5

Jose Oncina and Pedro García. 1992. Identifying Regular Languages In Polynomial Updated Time. In *Pattern Recognition and Image Analysis*, N Perez de la Blanca, A. Sanfeliu, and E Vidal (Eds.). World Scientific, 49–61. https://doi.org/10.1142/9789812797902_0004

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276520

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 139 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360565

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Python Software Foundation. 2020. CSV File Reading and Writing. https://web.archive.org/web/20230825040257/https://docs.python.org/3/library/csv.html#csv.Sniffer.

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-Guided Synthesis of Datalog Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 62 (dec 2019), 27 pages. https://doi.org/10.1145/3371130

Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17).* AAAI Press, 882–890.

R. L. Rivest and R. E. Schapire. 1989. Inference of Finite Automata Using Homing Sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing (STOC '89).* Association for Computing Machinery, New York, NY, USA, 411–420. https://doi.org/10.1145/73007.73047

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19.* International Joint Conferences on Artificial Intelligence Organization, 6117–6124. https://doi.org/10.24963/ijcai.2019/847

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* Association for Computing Machinery, New York, NY, USA, 281–294. https://doi.org/10.1145/1065010.1065045

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII).* Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Andreas Stolcke and Stephen Omohundro. 1994. Inducing probabilistic grammars by Bayesian model merging. In *Grammatical Inference and Applications*, Rafael C. Carrasco and Jose Oncina (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 106–118. https://doi.org/10.1007/3-540-58473-0_141

Loup Vaillant. 2020. Earley Parsing Explained. https://web.archive.org/web/20230825041600/https://loup-vaillant.fr/tutorials/earley-parsing/

Gerrit J. J. van den Burg, Alfredo Nazábal, and Charles Sutton. 2019a. CSV_Wrangling. https://web.archive.org/web/20230825042110/https://github.com/alan-turing-institute/CSV_Wrangling.

Gerrit J. J. van den Burg, Alfredo Nazábal, and Charles Sutton. 2019b. Wrangling Messy CSV Files by Detecting Row and Type Patterns. *Data Min. Knowl. Discov.* 33, 6 (nov 2019), 1799–1820. https://doi.org/10.1007/s10618-019-00646-y

Enrique Vidal. 1994. Grammatical inference: An introductory survey. In *Grammatical Inference and Applications*, Rafael C. Carrasco and Jose Oncina (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–4. https://doi.org/10.1007/3-540-58473-0_131

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017a. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (dec 2017), 30 pages. https://doi.org/10.1145/3158151

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (oct 2017), 26 pages. https://doi.org/10.1145/3133886

Wikipedia contributors. 2023a. List of postal codes — Wikipedia, The Free Encyclopedia. http://web.archive.org/web/20230801230749/https://en.wikipedia.org/wiki/List_of_postal_codes

Wikipedia contributors. 2023b. National conventions for writing telephone numbers — Wikipedia, The Free Encyclopedia. https://web.archive.org/web/20230801230922/https://en.wikipedia.org/wiki/National_conventions_for_writing_telephone_numbers