

Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers

Alan Leung, Sorin Lerner
University of California, San Diego, USA
{aleung, lerner}@cs.ucsd.edu

Abstract—We present Parsimony, a programming-by-example development environment for synthesizing lexers and parsers by example. Parsimony provides a graphical interface in which the user presents examples simply by selecting and labeling sample text in a text editor. An underlying synthesis engine then constructs syntactic rules to solve the system of constraints induced by the supplied examples. Parsimony is more expressive and usable than prior programming-by-example systems for parsers in several ways: Parsimony can (1) synthesize lexer rules in addition to productions, (2) solve for much larger constraint systems over multiple examples, rather than handling examples one-at-a-time, and (3) infer much more complex sets of productions, such as entire algebraic expression grammars, by detecting instances of well-known grammar design patterns. The results of a controlled user study across 18 participants show that users are able to perform lexing and parsing tasks faster and with fewer mistakes when using Parsimony as compared to a traditional parsing workflow.

Index Terms—Lexer, parser, program synthesis, programming-by-example.

I. INTRODUCTION

Despite over four decades of research on parsing, building parsers remains a difficult task. Widely used parser generators such as Bison demand detailed understanding of their underlying algorithms for effective use. More modern tools such as ANTLR [1] and Packrat parsers [2] are arguably more user friendly, but even still have their own subtle gotchas, such as restrictions against prefix matches or left recursion [3]. Even generalized parsers [4], which allow specification of any context-free grammar, present subtle difficulties as they allow specification of ambiguous grammars.

Programming-by-example (PBE) is a programming paradigm that improves user-friendliness by allowing users to construct programs by supplying examples demonstrating the result of an intended computation, rather than writing code manually. In this paper we present Parsimony, a novel application of PBE for constructing parsers: the user selects text in a file the user wishes to parse, then supplies a label for that selection. Under the hood, Parsimony infers productions to parse those selections. As the user provides more examples in this way, Parsimony successively constructs a more complete implementation.

As will be discussed in detail in related work, compared to Parsify, our prior work on PBE for parsers [5], Parsimony is more expressive, more usable, and has been evaluated more extensively – in particular, through an end-user study. The key to Parsimony’s expressiveness and usability lies in several novel technical contributions, which we now discuss.

Lexer by Example: Parsimony provides new facilities for synthesizing lexer definitions from example tokens. Our key insight is to exploit a large corpus of useful, curated regular expressions (regexes) that already exist: the lexers for existing programming languages. We frame the task of synthesizing lexers as queries to a data structure called an R-DAG built from this corpus. In contrast to previous approaches [6], our approach guarantees that any synthesized rule will be *realistic* rather than *synthetic*, in the sense that it is known to be useful in the context of a real-world language implementation.

Insensitivity to Order: To make Parsimony insensitive to the order in which the user presents examples, we develop a fresh perspective on a classical parsing algorithm first described over 50 years ago, the Cocke-Younger-Kasami (CYK) parsing algorithm [7]. We propose a novel graph data structure built from CYK tables called the *CYK automaton* that efficiently tracks a large set of candidate productions, rather than just one. Crucially, because the number of candidates can explode with the length of the example, CYK automata efficiently encode an exponential number of candidates with space only quadratic in the length of the example. We then frame synthesis as graph transformations on automata in which candidates are only removed from consideration when no longer applicable, thus avoiding premature loss of candidates.

Principled Generalization: To allow Parsimony to generalize from examples, we make the key insight that many important design patterns can be encoded by specially constructed CYK automata. Detecting an instance of a design pattern then reduces to a standard graph intersection between two CYK automata: one representing the pattern and one representing the example. We demonstrate the generality of this approach by implementing the repetition patterns from Parsify along with new patterns (such as infix algebraic expressions), simply by defining an automaton for each pattern, along with a schema for the productions to generate based on that pattern.

Finally, we perform a thorough evaluation of Parsimony’s effectiveness via a controlled study in which 18 programmers previously unfamiliar with Parsimony performed a series of tasks using experimental and control variants of Parsimony. Our results show that Parsimony improves user outcomes as measured by both time to completion and number of mistakes.

In summary, we make the following contributions:

- (1) A novel data structure, the R-DAG, for synthesizing lexer rules by example (Section III).

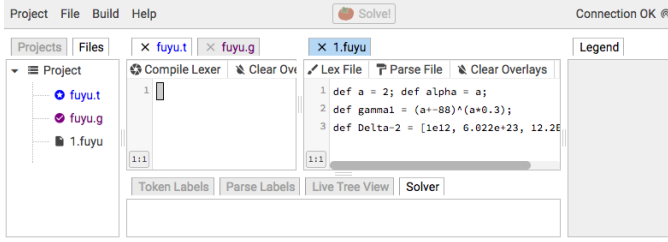


Fig. 1. The Parsimony user interface.

- (2) Another novel data structure, the CYK automaton, and corresponding algorithms for synthesizing productions by example (Section IV).
- (3) An extensible framework for generalizing from examples of common parser design patterns (Section IV).
- (4) The results of a controlled user study demonstrating the effectiveness of our approach (Section V).

II. OVERVIEW

Parsimony’s user interface is shown in Figure 1. Its basic functionality includes standard features ubiquitous amongst integrated development environments: (1) a customizable workspace consisting of resizable panes and tabs, (2) a file browser for managing project contents, and (3) text editors for viewing and editing files. Parsimony also borrows graphical features from Parsify, such as tree visualizations and coloring of text files based on grammar changes. Unique to Parsimony, however, are two tabs for the lexer and parser synthesis engines:

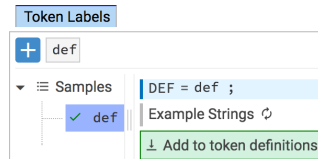
- (1) *The Token Labels Tab* allows the user to synthesize lexer rules from example tokens.
- (2) *The Solver Tab* provides rich functionality for synthesizing and previewing grammar productions derived from strings labeled with syntactic categories (i.e., nonterminals).

In the remainder of this section, we illustrate the Parsimony workflow and its salient features by walking through a series of scenarios demonstrating how we might employ Parsimony to develop the lexer and parser for a toy language called Fuyu. A sample Fuyu program, 1.fuyu, is shown below.

```
def a = 2; def alpha = a;
def gamma1 = (a+-88)^(a*0.3);
def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

Keywords: We start by synthesizing a lexer rule for the `def` keyword via following three steps: (1) click the Token Labels Tab to activate it, (2) select the substring “def” in 1.fuyu, then (3) add it as an example token by clicking the blue plus sign that appears.

The Token Labels Tab then updates its contents, as shown at right. In particular, the left-hand drop-down shows the list of examples added (only one so far), and the right-hand side shows a candidate rule that Parsimony has inferred from that example: `DEF = def`. This rule meets our requirements, so we add it to our lexer definition by clicking the button labeled *Add to token definitions*.



Parsimony immediately recompiles the lexer and colors 1.fuyu in response. The coloring, shown at right, tells us two important facts.

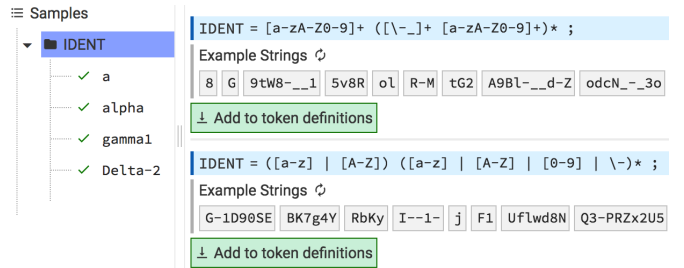
First, the colored box surrounding “def”

tells us that “def” matches the lexer rule we just defined, as indicated by the Legend. Just like a chart legend, the Legend gives the correspondence

between colors and names. Second, the red error box tells us that no lexer rule yet matches the character “a”. Intuitively, the error box’s location tells us how far into the file the lexer was able to construct the token stream. To fix this error, we will need to define a rule for identifiers like “a”.

Identifiers: The `DEF` rule we have just defined is the simplest sort of rule – it matches exactly the string “def”, which is easy enough to write without synthesizing it. Identifiers, however, are a more challenging sort of token. Suppose our language specification dictates that identifiers consist only of alphanumeric characters and hyphens; additionally, the first character must be alphabetical. We want Parsimony to automatically synthesize a lexer rule that meets that specification. We start by adding the example identifier “a” to the Token Labels Tab. Based on this single example, Parsimony infers the candidate rule `ALC = a`, which is disappointingly specific because one example is not enough for Parsimony to make a good inference.

To ask Parsimony to infer a rule from a *group of examples*, rather than just one, we drag and drop multiple samples into their own folder. Shown below is a folder labeled `IDENT` into which we have added the four identifiers from 1.fuyu; the right-hand side of the figure shows that Parsimony has inferred two different candidates from the examples in that folder.



To gain more intuition, we click the *Example Strings* buttons to ask Parsimony to show us examples of strings matched by each rule. Parsimony then updates the view with the strings shown in gray. It is clear from them that the first rule permits underscores, which violates our specification. The second rule, however, seems correct by inspection of the example strings and the rule’s definition, so we accept the inference. As before, Parsimony recompiles the lexer and colors 1.fuyu.

At this point, based on the coloring we can proceed as before: synthesize a new rule for the next failing token, which happens to be the “=” token. Since the basic scenario is the same as for keywords, let us assume for the sake of exposition that lexer rules for the remaining basic symbols (e.g., “+”, “*”, “{”, etc.) have been defined in the remainder of this section.

Numeric Literals: Numeric literals in Fuyu take the form of integers or floating point numbers specified in decimal or scientific notation. The desired lexer should assign such literals the token name NUMBER. The six numeric literals from 1.fuyu are 2, -88, 0.3, 1e12, 6.022e+23, and 12.2E-10.

Numeric literals are the most complex lexical forms in Fuyu. It would likely take a seasoned veteran of lexical analysis to correctly implement its regex on the first try:

```
\-?(0|[1-9][0-9]*)\.[0-9]+)?([Ee][+-]?([0-9][0-9]*))?
```

Parsimony synthesizes this regex from just those six examples. In fact, it is the only candidate Parsimony shows the user because there exists no other expression of equivalent or better quality in its corpus of training data. Parsimony uses a notion of quality based on how specifically the candidate matches the examples: for instance, the pattern `.*` also matches the examples, but it is much more general and thus inferior – we define this notion of quality formally in Section III. After accepting the inference, our lexer is complete. 1.fuyu, with all tokens properly colored, is shown below.

```
1 def a = 2; def alpha = a;
2 def gamma1 = (a+-88)^(a*0.3);
3 def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

With lexer in hand, we proceed to the parser. In this section, we successively augment fuyu.g with productions for the various syntactic constructs of Fuyu.

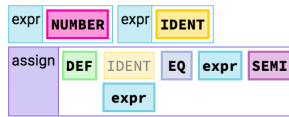
Simple Assignments: We start the process by posing an example of an assignment statement. We do this by 1) selecting "def a = 2;"; 2) typing "assign" into the text box that appears, then 3) clicking the Solve button. The Solver Tab responds by presenting the following stylized candidate production that Parsimony has synthesized from the example:



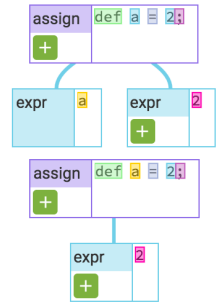
The production is close to correct, but it has the token NUMBER hardcoded in the fourth position, which precludes other kinds of non-numeric expressions. To fix this, suppose we pose another example: "def alpha = a;". The Solver Tab now responds with a pair of candidates:



At this point, it should be clear that Parsimony needs to be taught that NUMBER and IDENT are instances of a common syntactic category (nonterminal) representing expressions: `expr`. To do this, we pose to the Solver both "2" and "a" as examples of `expr`. The result is a set of three candidates, shown at right. The first two candidates match our expectation: to parse NUMBER and IDENT tokens as expressions, add the two productions `expr → NUMBER` and `expr → IDENT`. The third candidate has a special form. It indicates that we can choose between two options for the second position: either IDENT or `expr`. Parsimony gives us this option because it has determined that either choice is consistent with the examples we have provided.



To help us make a decision, Parsimony shows us parse tree visualizations corresponding to each option, shown at right. In particular, if we choose `expr`, we will get the top parse tree. If we choose IDENT, we will get the bottom parse tree. Suppose that according to our specification, only variable names (i.e., IDENT tokens), can appear on the left-hand side of an assignment. To achieve this, we choose the IDENT option before accepting the solution. All our interactions with the Solver thus far have the net effect of augmenting fuyu.g with the three productions `expr → NUMBER`, `expr → IDENT`, and `expr → DEF IDENT EQ expr SEMI`. Parsimony automatically recompiles the parser, then colors 1.fuyu accordingly: the result is shown below. Note that the first two assignments are now surrounded by colored boxes corresponding to the nonterminal `assign`.



```
1 def a = 2; def alpha = a;
2 def gamma1 = (a+-88)^(a*0.3);
3 def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

Algebraic Expressions: From the appearance of line 2, we know that our parser cannot yet handle the right-hand side of the assignment to `gamma1`, so we pose a new `expr` example: "(a+-88)^(a*0.3)". Because algebraic expressions are ubiquitous in programming languages, Parsimony contains a powerful heuristic mechanism for detecting such syntactic constructs. Based on this heuristic, Parsimony presents the user with a graphical wizard that asks 1) if this is indeed an algebraic expression, 2) for each operator (+, *, ^) whether that operator is left- or right-associative, and 3) what should be the order of precedence for those operators. Based on the answers to these questions, Parsimony constructs an idiomatic subgrammar for parsing expressions of this kind. Suppose we answer using the standard mathematical order of operations. The synthesized subgrammar then comprises four productions with associativity annotations: `expr → expr + expr {left}`, `expr → expr * expr {left}`, `expr → expr ^ expr {right}`, and `expr → (expr)`. Additionally, Parsimony synthesizes the following precedence annotation: `priorities { expr → expr ^ expr > expr → expr * expr ; expr → * expr > expr → expr + expr ; }`. Under the hood, these annotations compile to *disambiguating filters* [8], [9] that enforce a policy in the parser such that parse trees obey the specified associativity and precedence hierarchy. Parsimony recolors 1.fuyu in accordance with this new set of inferences:

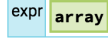
```
1 def a = 2; def alpha = a;
2 def gamma1 = (a+-88)^(a*0.3);
3 def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

Array Literals: The last syntax left to handle is the array literal, shown on line 3. We pose "[1e12, 6.022e+23, 12.2E-10]" as an example of an array. Parsimony contains a built-in heuristic to detect delimited repetitions, another ubiquitous language design pattern. Based on this heuristic,

Parsimony presents a graphical wizard confirming whether 1) each element of the list is a `NUMBER` or `expr`, 2) the separator between elements is a comma, and 3) the list is surrounded by a pair of square brackets. After confirming, Parsimony synthesizes an idiomatic subgrammar for delimited lists of `exprs`: `array` \rightarrow `[-array-inner]`, `-array-inner` \rightarrow `expr`, `-array-inner` \rightarrow `expr COMMA -array-inner`.

The new coloring shows that the array literal parses correctly. However, the parent assignment is still not surrounded by a box for `assign`: we haven't told Parsimony that an array literal is also a form of `expr`.

The fix is simple: we pose "[1e12, 6.022e+23, 12.2E-10]" as an example of an `expr`, then accept the inference `expr` \rightarrow `array`, shown at right.



Fuyu Program: Finally, we define a start symbol for the parser. We simply pose the entirety of 1.fuyu as an example of a program. Parsimony detects that this is yet another example of a ubiquitous pattern – this time, an undelimited list of `assign` instances. When we confirm this inference, Parsimony generates two productions: `program` \rightarrow `assign` and `program` \rightarrow `assign program`. Our parser is now complete.

III. LEXER SYNTHESIS

In this section we formalize Parsimony's algorithm for synthesizing lexers. The core of our approach is the R-DAG, a novel data structure for representing sets of regexes.

We first define the R-DAG*, the precursor to an R-DAG. An R-DAG* is a poset (R, \sqsubseteq^*) such that

- (1) R is a set of regexes,
- (2) $\sqsubseteq^* \subseteq (R \times R)$ is the language containment relation over R such that $\forall r_1, r_2 \in R. \mathcal{L}(r_1) \subset \mathcal{L}(r_2) \Leftrightarrow (r_1, r_2) \in \sqsubseteq^*$,
- (3) R contains a designated regex \top_D such that $\mathcal{L}(\top_D)$ is the set of all strings, and
- (4) $\forall r_1, r_2 \in R. \mathcal{L}(r_1) = \mathcal{L}(r_2) \Rightarrow r_1 = r_2$.

An R-DAG* can be viewed equivalently as a directed acyclic graph such that R is its vertex set and \sqsubseteq^* is its edge set. For ease of exposition we will view R-DAG*s as graphs or posets interchangeably as is convenient in the sequel.

We now define the R-DAG via reduction from an R-DAG*. Let $\mathcal{D}^* = (R, \sqsubseteq^*)$ be an R-DAG*. We define its corresponding R-DAG $\mathcal{D} = (R, \sqsubseteq)$ to be the transitive reduction of \mathcal{D}^* . That is, \mathcal{D} is the graph with the same vertex set R as \mathcal{D}^* , but with edge set \sqsubseteq , the unique minimum size relation whose transitive closure is \sqsubseteq^* .

From a practical perspective, we can view an R-DAG as a database of regexes such that the language containment relationship between regexes is stored explicitly in the form of graph edges. In our implementation, this database contains ~3K regexes scraped from open source lexer implementations [10]. We exploit this structure via a specially designed graph query to answer the question "What is the *most specific* set of regexes that match strings s_1, s_2, \dots ?" In this instance, "most specific" informally means that there exists no other regex (in the database) with smaller language that could also match those

strings. We additionally would like this set to be the *largest* set with this property, so we know we are not missing out. The remainder of this section discusses the query algorithm.

A. Regular Expression Inference via R-DAG Queries

We now define the HORIZON query on R-DAGs: the purpose of this query is to discover the *largest*, yet *most specific* set of regexes that match a set of example strings.

We start with some intuition. Suppose we have an R-DAG $\mathcal{D} = (R, \sqsubseteq)$ and a string s . First, we wish to find a set of regexes $\mathcal{H} \subseteq R$ such that every regex in \mathcal{H} matches s . Second, we require \mathcal{H} to be succinct: no two regexes in \mathcal{H} should be related by \sqsubseteq^* . Third, we require \mathcal{H} to be as large as possible without compromising quality: adding any regex would violate succinctness, and replacing any regex would make it worse (i.e., closer to \top_D). These three conditions are captured by the notions of *consistency*, *succinctness*, and *maximality*, which we now define formally.

Let S be a set of strings and let $\mathcal{D} = (R, \sqsubseteq)$. \mathcal{H} is consistent with S iff $\forall r \in \mathcal{H}, s \in S. s \in \mathcal{L}(r)$. \mathcal{H} is succinct with respect to \mathcal{D} iff $\forall r, r' \in \mathcal{H}. r \not\sqsubseteq^* r'$. \mathcal{H} is maximal with respect to (\mathcal{D}, S) iff no regex $r \in R$ exists such that: (1) $r \notin \mathcal{H}$, (2) $\forall s \in S. s \in \mathcal{L}(r)$, and (3) $\forall r' \in \mathcal{H}. (r \sqsubseteq^* r' \vee r' \not\sqsubseteq^* r)$. \mathcal{H} is a horizon of (\mathcal{D}, S) iff \mathcal{H} is consistent with S , succinct with respect to \mathcal{D} , and maximal with respect to (\mathcal{D}, S) .

We now define the query $\text{HORIZON}(\mathcal{D}, S)$, which computes the horizon of (\mathcal{D}, S) :

```

1 function HORIZON( $\mathcal{D}, S$ )
2    $\mathcal{W} \leftarrow \{\top_D\}; \mathcal{H} \leftarrow \emptyset$ 
3   while  $|\mathcal{W}| > 0$ 
4     let  $r = \text{removeAny}(\mathcal{W})$ 
5     let  $R_p = \{r' \in \text{predecessors}(\mathcal{D}, r) \mid \forall s \in S. s \in \mathcal{L}(r')\}$ 
6     if  $|R_p| > 0$ 
7        $\mathcal{W} \leftarrow (\mathcal{W} - r) \cup R_p$ 
8     else  $\mathcal{W} \leftarrow \mathcal{W} - r; \mathcal{H} \leftarrow \mathcal{H} \cup \{r\}$ 
9   return  $\mathcal{H}$ 

```

Intuitively, HORIZON maintains a worklist \mathcal{W} of vertices to inspect. The worklist initially contains only \top_D , the topmost regex that matches any string, and is thus guaranteed to be reachable from any other vertex of \mathcal{D} . In each iteration, we remove a regex r from the worklist and compute the set of predecessors of r that match all strings in S . If such predecessors are found, we then add them to the worklist and proceed to the next iteration. However, if no such predecessor exists, then we have gone as far down the graph as possible (i.e., we have found the *most specific* regex), so we add the current vertex to output set \mathcal{H} . We continue this process until the worklist is exhausted. At completion, \mathcal{H} is a set of regexes that is consistent with S , succinct with respect to \mathcal{D} , and maximal with respect to (\mathcal{D}, S) . In other words, it is the horizon of (\mathcal{D}, S) . Because each constituent regex is known to match every string in S , it is a candidate for the body of a lexer rule for S . Because \mathcal{H} is succinct with respect to \mathcal{D} and maximal with respect to (\mathcal{D}, S) , we know there are no "better" candidates that we have missed.

IV. PARSER SYNTHESIS

In this section we formalize Parsimony's algorithms for synthesizing parsers. We begin with a preliminary overview of context-free grammars and parsers.

A. Preliminaries

A *context-free grammar* is a tuple $G = (N, \Sigma, P, S)$, where N is a set of nonterminals, Σ is a set of terminals, S is a designated start nonterminal, and $P \subseteq (N \times V^*)$ is a set of productions where V is the set $N \cup \Sigma$ called the *vocabulary* of G . For convenience, productions $(A, \beta) \in P$ are written equivalently as $A \rightarrow \beta$.

String Indexing: String indices begin at 0. We write $\alpha_{[i]}$ to mean the symbol at index i of string α . By $\alpha_{[i...]}$ we denote the suffix of α starting at index i , and by $\alpha_{[i...j]}$ we denote the substring of α starting at index i with length $j - i$. We write $|\alpha|$ to denote the length of α , and we write $\alpha\beta$ for concatenation of α and β .

Derivations: We say α derives β in a single step, written $\alpha \Rightarrow \beta$, if P contains a production $A \rightarrow \mu$ such that $\alpha = \gamma A \delta$ and $\beta = \gamma \mu \delta$. Equivalently, \Rightarrow is the single-step derivation relation such that $(\alpha, \beta) \in \Rightarrow$ iff α derives β in a single step. Let \Rightarrow^* be the transitive closure of \Rightarrow . We say α derives β iff $\alpha \Rightarrow^* \beta$, and a *derivation* of β from α is a sequence $\alpha \Rightarrow \dots \Rightarrow \beta$ that witnesses $\alpha \Rightarrow^* \beta$. A *sentential form* is a string $\alpha \in V^*$ such that $S \Rightarrow^* \alpha$, a *sentence* is a sentential form containing only terminals, and $\mathcal{L}(G)$, the language of G , is the set of all its sentences.

CYK Algorithm: The Cocke-Younger-Kasami (CYK) parsing algorithm [7] is a classical dynamic programming algorithm for computing whether a string τ is derivable via a grammar G . For our purposes, the crucial feature of the algorithm is that it constructs a 2D table M , called a CYK table, that records for every substring τ' of τ the set of nonterminals that derive τ' : $A \in M_{i,l} \iff A \Rightarrow^* \tau_{[i...i+l]}$. For a description of the algorithm itself, we refer the reader to Grune and Jacobs [11]. In the remainder of this paper, we assume without definition that we have a function $\text{CYK}(G, \tau)$ that returns a CYK table given a grammar G and string τ .

B. Parser Synthesis Constraint Systems

We now make precise the parser synthesis problem by framing it as constraint satisfaction. A *parser synthesis constraint system* is a tuple $\mathbb{C} = (G, \mathbb{F}, \mathbb{L})$ where

- (1) G is a grammar,
- (2) $\mathbb{F} = \{\tau^{f_1}, \tau^{f_2}, \dots, \tau^{f_k}\}$ is a set of strings called *files* with unique labels f_1, f_2, \dots, f_k called *file names*, and
- (3) \mathbb{L} is a set of *parse constraints* of form $\langle A, i, l \rangle^f$ denoting a length l selection starting at index i into file $\tau^f \in \mathbb{F}$ labeled with nonterminal A .

Let the notation $G \uplus P$ mean the grammar G augmented with additional productions P . A solution to constraint system \mathbb{C} is a set of productions P that satisfies the formula:

$$\forall \langle A, i, l \rangle^f \in \mathbb{L}. M = \text{CYK}(G \uplus P, \tau_{[i...i+l]}^f) \wedge A \in M_{i,l}$$

If P satisfies the above formula, we say P satisfies \mathbb{C} . Intuitively, then, the parser synthesis problem is the task of finding P , a set of productions that allow us to derive every constrained substring encoded by \mathbb{L} .

C. A Data Structure for Large Sets of Candidate Productions

In this section we describe the *CYK automaton*, a data structure for efficiently representing large sets of candidate productions. This data structure is a central component of Parsimony's parser synthesis engine.

Intuition: Intuitively, a CYK table is simply a static record of the nonterminals that derive each piece of a string τ being parsed. For example, $M_{2,5}$ is the set of nonterminals that derive the substring $\tau_{[2...7]}$. However, this is only one interpretation of the table. An alternate perspective is that the table contains predictions about the set of productions that we might add to our grammar to grow the language. Consider, for instance, that we have the string ab and grammar with productions $A \rightarrow a$ and $B \rightarrow b$. We would then have CYK table M such that $M_{0,1} = \{A\}$, $M_{1,1} = \{B\}$, and $M_{0,2} = \emptyset$. Suppose that we wish for ab to also belong to the language we are designing. What production should we add to make it so? The CYK table has almost all the information we need to answer that question. We could combine one element of $M_{0,1}$ with one element of $M_{1,1}$ to create the production body AB . If we add the production $S \rightarrow AB$, we will have augmented the language to include exactly the string ab . Note, however, that there are other productions we could have added instead: $S \rightarrow aB$, $S \rightarrow Ab$, or $S \rightarrow ab$. Even in this trivial case, we see that there can be many such candidate productions. A CYK automaton is a data structure for making explicit what the candidates are and for providing efficient queries to compute those candidates.

Definition: A CYK automaton is a directed graph $Y = (I, E, i_s, I_f, U, \Lambda)$ where $I \subseteq \mathbb{Z}^*$ is a set of vertices, $E \subseteq I \times I$ is a set of edges, $i_s \in I$ is a designated start vertex, $I_f \subseteq I$ is a designated set of final vertices, U is a set of symbols, and Λ is a map from edges in E to sets of symbols in U . We use the notation $\Lambda[e \mapsto X]$ for the map λx . if $x = e$ then X else $\Lambda(x)$.

Given a grammar G and string τ , we construct a CYK automaton via algorithm BUILD-CYK-AUTOMATON below.

```

1 function BUILD-CYK-AUTOMATON( $G, \tau, i_s, i_f$ )
2   let  $(N, \Sigma, \cdot, \cdot) = G$ 
3   let  $M = \text{CYK}(G, \tau)$ 
4    $(I, E, \Lambda) \leftarrow (\{0 \leq i \leq |\tau|\}, \emptyset, \lambda x. \emptyset)$ 
5   for  $i$  in  $[i_s, i_f]$ 
6      $E \leftarrow E \cup \{(i, i+1)\}$ 
7      $\Lambda \leftarrow \Lambda[(i, i+1) \mapsto \{\tau_{[i]}\}]$ 
8   for  $i$  in  $[i_s, i_f]$ 
9     for  $l$  in  $[1, i_f - i]$ 
10      if  $M_{i,l} \neq \emptyset$ 
11         $E \leftarrow E \cup \{(i, i+l)\}$ 
12         $\Lambda \leftarrow \Lambda[(i, i+l) \mapsto \Lambda(i, i+l) \cup M_{i,l}]$ 
13   return  $(I, E, i_s, \{i_f\}, N \cup \Sigma, \Lambda)$ 

```

Intuitively, each vertex of a CYK automaton corresponds to a position *between tokens* (e.g., 1 indicates the position between the 0th and 1st token). An edge between vertices j and k corresponds to the CYK table entry $M_{j,k-j}$: that is, the set of nonterminals of G that derive the substring $\tau_{[j...k]}$. This

set is recorded via map entry $\Lambda(j, k)$. Additionally, for every singleton edge $(j, j + 1)$ (i.e., those that correspond to length 1 substrings), we also add to Λ the terminal $\tau_{[j]}$ occurring at that position. By construction, any path between vertex 0 and vertex $|\tau|$ corresponds to a set of candidate production bodies that derive τ , as will be illustrated in the following section.

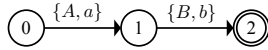
D. Parser Synthesis via CYK Automata

In this section, we progress through several descriptions of successively more sophisticated mechanisms for solving parser synthesis constraint systems via CYK automata, building from simple cases up to more complex cases. We motivate each augmentation with an example demonstrating the limitation it overcomes. At the end of this section, we will have arrived at the full algorithm used by Parsimony, dubbed PARSYNTH.

Case 1 – One Parse Constraint: We first consider synthesis constraint systems with only one parse constraint. Let us revisit the example from Section IV-C, in which we have the string ab , a partially implemented grammar G_1 with productions $A \rightarrow a$ and $B \rightarrow b$, but wish for ab to derive from a new nonterminal S that has yet to be implemented. As we saw in Section II, to do this using Parsimony we simply highlight the text ab , type the label S into the Solver text box, then run the Solver. Under the hood, this sequence of user operations constructs the following parser synthesis constraint system:

$$\mathbb{C}_1 = (G_1, \mathbb{F}_1, \mathbb{L}_1), \quad \mathbb{F}_1 = \{ab^{f_1}\}, \quad \mathbb{L}_1 = \{\langle S, 0, 2 \rangle^{f_1}\}$$

To solve this constraint system, our strategy will be to construct a CYK automaton for the parse constraint in \mathbb{L}_1 , then generate productions corresponding to the shortest path through the automaton. Specifically, we construct the automaton for constraint $\langle S, 0, 2 \rangle^{f_1}$ with parameters $\tau = ab, i_s = 0, I_f = \{2\}$:



The shortest (and only) path is 0, 1, 2. Taking the n -ary Cartesian product of edge attributes along the path, we have $\{A, a\} \times \{B, b\} = \{(A, B), (A, b), (a, B), (a, b)\}$. Each constituent tuple, when read from left to right, is the body of a production for S that derives ab . That is, any such production is a solution to \mathbb{C}_1 . To succinctly represent the space of choices, Parsimony uses a graphical representation called a *candidate matrix*, an example of which is shown below. The semantics of a candidate matrix is straightforward: the k^{th} column of the matrix shows all the symbols that may possibly occur at position k in the corresponding production body. The user must enable exactly one such symbol per column. The sequence of enabled symbols, when read from left to right, gives us the corresponding production. A candidate matrix succinctly visualizes a potentially large set of productions that grows exponentially in the number of columns: a candidate matrix with k columns and n symbols per column encodes n^k productions. We denote by $X[[N_1 N_2 \dots N_k]]$ the candidate matrix with k columns such that N_j is the set of symbols in column j , and X is the left-hand side symbol.



Each production encoded by a candidate matrix \mathbb{M} is called a *valuation* of \mathbb{M} . To construct candidate matrices, we define the following *constructor function* Ψ_{Λ}^X , which given a path through a CYK automaton, constructs the corresponding candidate matrix: $\Psi_{\Lambda}^X(i_0, \dots, i_n) = X[[\Lambda(i_0, i_1) \dots \Lambda(i_{n-1}, i_n)]]$.

Case 2 – Non-Overlapping Parse Constraints: We now consider systems of multiple parse constraints. As a first step, we consider the simplest case in which no two parse constraints overlap (i.e., they reference disjoint substrings).

To handle this in the straightforward way, we could construct one candidate matrix per parse constraint. Suppose we have the following constraint system \mathbb{C}_2 , which models a grammar where identifiers id and numbers 1 are forms of expressions E , and the user has selected and labeled two substrings " $\text{id} = \text{id}$ " and " $\text{id} = 1$ " with the nonterminal S (statements). The synthesis task is to infer one or more productions for S .

$$\begin{aligned} G_2 &= (\{E, S\}, \{\text{id}, 1, =\}, \{E \rightarrow \text{id}, E \rightarrow 1\}, S) \\ \mathbb{C}_2 &= (G_2, \mathbb{F}_2, \mathbb{L}_2) \quad \mathbb{F}_2 = \{\text{id} = \text{id}^{f_1}, \text{id} = 1^{f_2}\} \\ \mathbb{L}_2 &= \{\langle S, 0, 3 \rangle^{f_1}, \langle S, 0, 3 \rangle^{f_2}\} \end{aligned}$$

The computed solution is the set of two candidate matrices $\{\mathbb{M}_1, \mathbb{M}_2\}$ where $\mathbb{M}_1 = S[[\{E, \text{id}\} \{=\} \{E, \text{id}\}]]$ and $\mathbb{M}_2 = S[[\{E, \text{id}\} \{=\} \{E, 1\}]]$. In this situation, Parsimony would display both candidate matrices in the Solver Tab and allow the user to interact with each. There are two problems in this scenario: (a) Since the user provided parse constraints for only one kind of syntactic construct (namely, statements S), it may be confusing for the user to see two distinct candidate matrices when only one was expected, and (b) it may lead the user to accept a solution of two productions (one for \mathbb{M}_1 and one for \mathbb{M}_2), which is subpar because the more economical solution to \mathbb{C}_2 consists of only a single production: namely $S \rightarrow \text{id} = E$. Clearly, our algorithm needs to be improved to handle such cases and avoid computing more candidate matrices than necessary.

Case 3 – Non-Overlapping Parse Constraints with Sharing: As we have just seen, \mathbb{M}_1 and \mathbb{M}_2 are redundant – we need only one of the two since both share the desired valuation $S \rightarrow \text{id} = E$. To eliminate such redundancies, our strategy is to find a way to *partition* the constraints \mathbb{L}_2 into disjoint sets, called classes, such that the constraints in each class can be satisfied by the same productions. By producing as few classes as we can, then computing only a single candidate matrix for each such class, we seek to produce an economical solution. To do this we will first need to define an operation for the intersection of two CYK automata.

Intersection: Let $Y = (I, E, i_s, I_f, U, \Lambda)$ and $Y' = (I', E', i'_s, I'_f, U', \Lambda')$. The intersection of Y and Y' , written $Y \tilde{\cap} Y'$, is defined as follows:

$$\begin{aligned} Y \tilde{\cap} Y' &= (I \times I', E^{\cap}, (i_s, i'_s), I_f \times I'_f, U \cap U', \Lambda^{\cap}) \\ \Lambda^{\cap} &= \lambda((x, x'), (y, y')) . \Lambda((x, y)) \cap \Lambda'((x', y')) \\ E^* &= \{((x, x'), (y, y')) \mid (x, y) \in E \wedge (x', y') \in E'\} \\ E^{\cap} &= \{e \in E^* \mid \Lambda^{\cap}(e) \neq \emptyset\} \end{aligned}$$

We say that CYK automata Y and Y' are compatible, written $\text{COMPATIBLE}(Y, Y')$, if and only if the intersection $Y \tilde{\cap} Y'$

```

1 function PARTITION( $\mathbb{Y}$ )
2   let  $\text{ays} = \{((A_1, Y_1), (A_2, Y_2)) \mid ((A_1, Y_1), (A_2, Y_2)) \in \mathbb{Y}^2 \wedge$ 
3      $A_1 = A_2 \wedge \text{COMPATIBLE}(Y_1, Y_2)\}$ 
4   if ( $\text{ays} \neq \emptyset$ )
5     let  $(\text{ay}_1, \text{ay}_2) = \text{first}(\text{sortDescBy}(\text{SCORE}_{\mathbb{Y}}, \text{ays}))$ 
6     let  $((A_1, Y_1), (A_2, Y_2)) = (\text{ay}_1, \text{ay}_2)$ 
7     let  $Y_{12} = Y_1 \cap Y_2$ 
8     return PARTITION( $\mathbb{Y} - \{\text{ay}_1, \text{ay}_2\} \cup \{(A_1, Y_{12})\}$ )
9   else return  $\mathbb{Y}$ 
10 function SCORE $_{\mathbb{Y}}((A_1, Y_1), (A_2, Y_2))$ 
11 if ( $A_1 \neq A_2$ ) return 0
12 return  $\sum_{(A_3, Y_3) \in \mathbb{Y} \text{ s.t. } A_3 = A_1} \text{SCORE-ONE-TRIPLET}(Y_1, Y_2, Y_3)$ 
13 function SCORE-ONE-TRIPLET( $Y_1, Y_2, Y_3$ )
14 if ( $\text{COMPATIBLE}(Y_1, Y_3) \wedge \text{COMPATIBLE}(Y_2, Y_3) \wedge$ 
15    $\text{COMPATIBLE}(Y_1 \cap Y_2, Y_3)$ ) return 1
16 else return 0

```

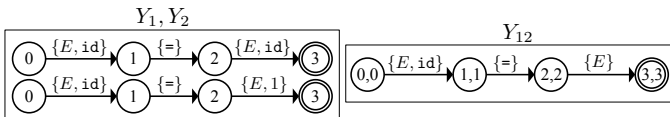
Fig. 2. Algorithm PARTITION. \mathbb{Y} is a set of pairs (A, Y) where Y is a CYK automaton and A is its corresponding nonterminal.

contains a path from the start vertex (i_s, i'_s) to a final vertex in $I_f \times I'_f$.

Intersection of CYK automata is similar to the standard product construction for intersection of finite automata; however, we additionally intersect edge attributes such that each resulting edge attribute is the set of symbols shared by *both* originating edges. With this construction, any path through the intersection $Y \cap Y'$ corresponds to a common set of candidates shared by both Y and Y' . If $\neg \text{COMPATIBLE}(Y, Y')$, then there exists no such shared candidate.

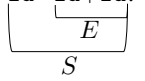
Partitioning: Our partitioning algorithm is shown in Figure 2. We describe the algorithm informally here. We first compute for each parse constraint a tuple (A, Y) where A is the nonterminal of the constraint, and Y is the CYK automaton constructed from that constraint. This set of tuples, denoted \mathbb{Y} , serves as the input to PARTITION. We then iteratively intersect automata until no more intersection is possible. In each iteration, we greedily intersect only the highest scoring pair of automata, where our scoring function $\text{SCORE}_{\mathbb{Y}}$ gives preference to the pair that is maximally compatible with all the other automata. The idea is to intersect those pairs whose intersection has the most opportunity to intersect again in a future iteration. At termination, the output of PARTITION should be a set \mathbb{Y}' such that $|\mathbb{Y}'| \leq |\mathbb{Y}|$, and each constituent tuple $(A', Y') \in \mathbb{Y}'$ contains a CYK automaton Y' that is possibly the intersection of multiple automata from the original input \mathbb{Y} . Most importantly, any path in Y' from start to final vertex gives us a solution to all the parse constraints that gave rise to Y' . In other words, each element of \mathbb{Y}' corresponds to the *class of parse constraints that it solves*.

For illustration, consider the constraint system \mathbb{C}_2 from Case 2. The CYK automata before and after partitioning are shown below, where Y_1 and Y_2 correspond to the two constraints in \mathbb{L}_2 , and Y_{12} is the intersection of Y_1 and Y_2 due to partitioning.



By incorporating partitioning, the computed solution becomes the single candidate matrix $\mathbb{M}_3 = S[\{E, \text{id}\} \{=\} \{E\}]$. There are two key features of this solution to note. First, there is only one candidate matrix, not two. Second, the last column of \mathbb{M}_3 contains only E , not id or 1, because id and 1 were excluded from edge $((2, 2), (3, 3))$ in Y_{12} during intersection. The two possible valuations of \mathbb{M}_3 are $S \rightarrow E = E$ and $S \rightarrow \text{id} = E$. In fact, these are the only possibilities: there exists no other single production that would also satisfy \mathbb{C}_2 . In this sense, this computed solution is as good as possible.

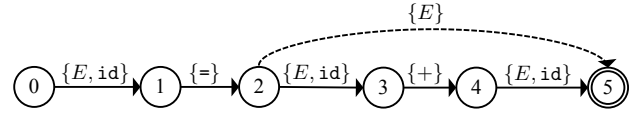
Case 4 – Overlapping Parse Constraints: A complication occurs when constraints can overlap. Suppose we have the following constraint system \mathbb{C}_4 , which represents a situation in which we have the same grammar G_2 as before, but the user has created overlapping parse constraints like so: $\text{id}=\text{id}+\text{id}$.



$$\mathbb{C}_4 = (G_2, \mathbb{F}_4, \mathbb{L}_4)$$

$$\mathbb{F}_4 = \{\text{id} = \text{id} + \text{id}^{f_1}\} \quad \mathbb{L}_4 = \{\langle S, 0, 5 \rangle^{f_1}, \langle E, 2, 3 \rangle^{f_1}\}$$

The user's intention is to specify that the enclosing context $\text{id}=\text{id}+\text{id}$ is an example of a statement S , and that nested within it $\text{id}+\text{id}$ is an example of an expression E . Unfortunately, our algorithm sketched so far ignores this nested relationship, and would compute the solution $\{\mathbb{M}_4, \mathbb{M}_5\}$ where $\mathbb{M}_4 = S[\{E, \text{id}\} \{=\} \{E, \text{id}\} \{+\} \{E, \text{id}\}]$ and $\mathbb{M}_5 = E[\{E, \text{id}\} \{+\} \{E, \text{id}\}]$. To see the problem, examine the CYK automaton for \mathbb{M}_4 (ignore the dotted edge for now):



The subpath 2,3,4,5 corresponds to the substring $\text{id}+\text{id}$ that the user has constrained with nonterminal E , but the automaton ignores that constraint and faithfully retains the underlying CYK table information for edges $(2, 3)$, $(3, 4)$, and $(4, 5)$. Thus, the synthesized candidate matrix \mathbb{M}_4 is overly specific and contains columns corresponding to those edges.

Our strategy is to replace such subpaths with *summary edges* that summarize the effect of nested constraints. For example, we replace the subpath 2,3,4,5 with a single edge $(2, 5)$ whose edge attribute $\{E\}$ references the nonterminal of the nested constraint. The dotted edge $(2, 5)$ is such a summary edge. After this transformation, the revised candidate matrix $\mathbb{M}'_4 = S[\{E, \text{id}\} \{=\} \{E\}]$ correctly encodes only those productions where the right hand side of the assignment statement S must be an expression E . The candidate matrix \mathbb{M}_5 remains untouched. Together, \mathbb{M}'_4 and \mathbb{M}_5 represent a solution of two interrelated productions (one for E and one for S that references E). For example, one possible valuation for \mathbb{M}'_4 and \mathbb{M}_5 is $S \rightarrow \text{id} = E$ and $E \rightarrow E + E$.

The algorithm for inserting summary edges is shown in Figure 3. The function APPLY-NESTING takes as input constraint system \mathbb{C} and returns a set \mathbb{Y} with summary edges inserted.

```

1 function APPLY-NESTING( $\mathbb{C}$ )
2   let  $(G, \{\tau^{f_j}\}_{j=0}^n, \mathbb{L}) = \mathbb{C}$ 
3    $\mathbb{Y} \leftarrow \emptyset$ 
4   for  $L = \langle A, i, l \rangle^{f_k} \in \mathbb{L}$ 
5     let  $\mathbb{L}' = \{L' \in \mathbb{L} \mid L \text{ contains } L'\}$ 
6      $Y' \leftarrow \text{BUILD-CYK-AUTOMATON}(G, \tau^{f_k}, i, i + l)$ 
7     for  $L' \in \mathbb{L}'$ 
8        $Y \leftarrow \text{SUMMARIZE}(Y, L')$ 
9    $\mathbb{Y} \leftarrow \mathbb{Y} \cup (A, Y)$ 
10  return  $\mathbb{Y}$ 

```

Fig. 3. Algorithm APPLY-NESTING. SUMMARIZE takes an automaton and parse constraint and returns the automaton with constrained paths replaced by a summary edge.

Case 5 – Constrained Patterns: Consider the constraint system \mathbb{C}_5 , which models the scenario in which the user has selected and labeled the string $[1, \text{id}, 1]$ with the nonterminal A (arrays).

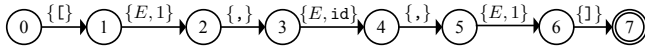
$$G_5 = (\{E\}, \{\llbracket, \rrbracket, \cdot, \cdot, 1, \text{id}\}, \{E \rightarrow 1, E \rightarrow \text{id}\}, E)$$

$$\mathbb{C}_5 = (G_5, \mathbb{F}_5, \mathbb{L}_5) \quad \mathbb{F}_5 = \{[1, \text{id}, 1]^{f_1}\} \quad \mathbb{L}_5 = \{\langle A, 0, 7 \rangle^{f_1}\}$$

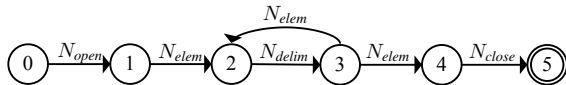
Plausibly, the user’s intention is that the constrained substring is an example of literal array syntax permitting repetition of the elements within square brackets. However, our algorithm sketched so far has no special handling of such patterns. It simply infers that the literal array must contain exactly 3 elements: $\mathbb{M}_6 = A[\llbracket \{ \} \{E, 1\} \{ \cdot, \cdot \} \{E, \text{id}\} \{ \cdot, \cdot \} \{E, 1\} \{ \} \rrbracket]$.

Our strategy for handling this case is two-fold. First, we detect instances of common grammar design patterns using the machinery for intersection of CYK automata. Second, for each pattern we predefine carefully crafted schema for productions; the schema contain holes to be instantiated with symbols that have been resolved during pattern detection.

Pattern Detection: Suppose Y_6 is the CYK automaton shown below from which we computed \mathbb{M}_6 .



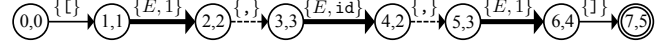
We wish to detect whether Y_6 represents an enclosed, delimited repetition: that is, the repetition of two or more instances of a symbol, each separated by a delimiter symbol, and surrounded by a matching pair of enclosing symbols. Our key insight is that it is possible to precisely specify such a pattern with a specially constructed CYK automaton Y_{edlist}^\bullet :



where $N_{open}, N_{close}, N_{elem}, N_{delim}$ are sets of opening enclosers, closing enclosers, element symbols, and delimiter symbols, respectively. We set N_{open} and N_{close} to statically predefined values for common enclosure terminals ($\llbracket, \rrbracket, (,)$), while we set N_{elem} and N_{delim} to be the vocabulary and terminals of the current grammar, respectively. Then, to detect whether Y_6 matches the pattern, we simply intersect Y_6 and Y_{edlist}^\bullet . If $\text{COMPATIBLE}(Y_6, Y_{edlist}^\bullet)$, then we have detected a match.

The intersection $Y_6 \cap Y_{edlist}^\bullet$ is shown below, where edges corresponding to array elements (N_{elem}) are bold, and edges

corresponding to delimiters (N_{delim}) are dashed.



The set intersection of all dashed edge attributes is $\{ \cdot, \cdot \}$ and gives us the set of possible delimiters N_{delim}^\cap . Analogously, the set intersection of all bolded edge attributes is $\{E\}$ and gives us the set of possible elements N_{elem}^\cap . Finally, the first and last edge attributes are $\{ \llbracket \rrbracket$ and $\{ \} \}$, which give us the sets of possible open and closing enclosers N_{open}^\cap and N_{close}^\cap .

Schema Instantiation: The last step is to instantiate productions. Parsimony contains the following predefined schema, named P_{edlist}^\bullet , for enclosed, delimited lists, where each hole (subscripted \bullet) is a placeholder to be instantiated by a nonterminal or terminal, and A_{fresh} is a fresh nonterminal.

$$P_{edlist}^\bullet = \left\{ \begin{array}{l} \bullet_{lhs} \rightarrow \bullet_{open} A_{fresh} \bullet_{close} \\ A_{fresh} \rightarrow \bullet_{elem} \ , \ A_{fresh} \rightarrow \bullet_{elem} \bullet_{delim} A_{fresh} \end{array} \right\}$$

The valid instantiations for each placeholder are restricted to the symbols (edge attributes) captured during intersection:

$$\bullet_{open} \in N_{open}^\cap \quad \bullet_{close} \in N_{close}^\cap \quad \bullet_{elem} \in N_{elem}^\cap \quad \bullet_{delim} \in N_{delim}^\cap$$

Additionally, \bullet_{lhs} is a special placeholder instantiated with A , the nonterminal from the originating parse constraint $\langle A, 0, 7 \rangle^{f_1}$. Fully instantiated, our solution is $\{A \rightarrow [A_{fresh}], A_{fresh} \rightarrow E, A_{fresh} \rightarrow E, A_{fresh} \rightarrow E\}$.

As already discussed in Section II, Parsimony’s interface for pattern detection and schema instantiation comes in the form of a wizard in the Solver Tab – the user can graphically select instantiations for each hole, or reject the inference altogether if the detected pattern is spurious. Parsimony also implements pattern detection and schema instantiation for infix algebraic expressions, undelimited lists, and unenclosed lists. In each case, we simply define a CYK automaton paired with a corresponding production schema. Their specification is similar in principle to that already shown, so we omit their details here. We encapsulate pattern detection and schema instantiation in procedure $\text{HEURISTIC}(\mathbb{Y})$, which returns a tuple (P, \mathbb{L}) where P is a set of instantiated production schemas (i.e., a set of productions) and \mathbb{L} is the set of originating parse constraints.

The Algorithm PARSYNTH

```

1 function PARSYNTH( $\mathbb{C} = (G, \mathbb{F}, \mathbb{L})$ )
2    $G' \leftarrow G$  ;  $\mathbb{L}' \leftarrow \mathbb{L}$  ;  $\tilde{\mathbb{M}} \leftarrow \emptyset$  ; change?  $\leftarrow \text{true}$ 
3   while change?
4     change?  $\leftarrow \text{false}$ 
5      $\mathbb{Y} \leftarrow \text{PARTITION}(\text{APPLY-NESTING}((G', \mathbb{F}, \mathbb{L})))$ 
6     let  $(P, \mathbb{L}'') = \text{HEURISTIC}(\mathbb{Y})$ 
7     if  $P \neq \emptyset$ 
8        $G' \leftarrow G' \uplus P$  ;  $\mathbb{L}' \leftarrow \mathbb{L}' - \mathbb{L}''$  ; change?  $\leftarrow \text{true}$ 
9   for  $i$  in  $[0, 1]$ 
10     $\mathbb{Y} \leftarrow \text{PARTITION}(\text{APPLY-NESTING}((G' \uplus \tilde{\mathbb{M}}, \mathbb{F}, \mathbb{L}')))$  ;  $\tilde{\mathbb{M}} \leftarrow \emptyset$ 
11    for  $(A, Y = (\cdot, \cdot, i_s, \{i_f\}, \cdot, \Lambda)) \in \mathbb{Y}$ 
12       $\tilde{\mathbb{M}} \leftarrow \tilde{\mathbb{M}} \cup \Psi_\Lambda^A(\text{SHORTEST-PATH}(Y, i_s, i_f))$ 
13  return  $(G', \tilde{\mathbb{M}})$ 

```

Lines 3-8 repeatedly attempt to match patterns, accumulating all instantiated schema until no more matches are found. Lines 11-12 accumulate candidate matrices from any parse constraints not handled by the first loop. We perform two passes of the

loop such that solutions computed in the second pass take advantage of those produced by the first. In particular, the operation $G' \tilde{\cup} \tilde{M}$ on line 10 inserts into G' productions of the form $X \rightarrow (A_1 | \dots | A_m) \dots (Z_1 | \dots | Z_n)$ for each candidate matrix $X[\{A_1, \dots, A_m\}, \dots, \{Z_1, \dots, Z_n\}]$ in \tilde{M} . To see why this is valuable, recall Section II, in which we synthesized the following candidate matrices: $\text{expr}[\{\text{IDENT}\}]$, $\text{expr}[\{\text{NUMBER}\}]$, and $\text{assign}[\{\text{DEF}\}\{\text{expr}, \text{IDENT}\}\{\text{EQ}\}\{\text{expr}\}\{\text{SEMI}\}]$. The underlined nonterminal expr is computed in the second pass by making use of the candidate production $\text{expr} \rightarrow \text{IDENT}$, which was computed in the first pass.

V. EVALUATION

To evaluate the effectiveness of Parsimony, we conducted a user study in which 18 subjects without previous experience using Parsimony were asked to complete a series of tasks using one of two interfaces: either Parsimony with all its features enabled, or a stripped-down version with no synthesis or visualization features at all. We test the following hypotheses:

- (1) **Hypothesis 1.** Parsimony helps users construct parsers more quickly than with a traditional parsing workflow.
- (2) **Hypothesis 2.** Parsimony leads users to make fewer mistakes than with a traditional parsing workflow.

We targeted our user study at programmers who had some familiarity with parsing, but who were not experts as determined by self-rating. Our sample pool consisted of 18 Computer Science students who were split by random assignment into control and experimental groups. The interface seen by the experimental group contained all features described in this paper. The interface seen by the control group retained only a workspace with file browser and text editors, but no synthesis or visualization features – the interface modeled a traditional parsing workflow in which the user employs a text editor and relies on command-line compiler feedback.

Both groups first read a brief introduction, then followed a tutorial introducing them to the features of their interface. Subjects then had two hours to complete 9 tasks asking them to implement lexers and parsers for two toy languages, Fuyu and Hachiya, designed specifically for the experiment. We chose synthetic languages to prevent bias that might stem from users' prior familiarity with existing languages – although synthetic, these languages contain syntactic constructs that occur commonly in real languages, such as literal primitives and data structures, loops, branches, and various statements. Due to lack of space, we provide reference definitions online at <https://github.com/parsimony-ide/ase2017-extra>.

The first seven tasks asked subjects to implement the lexer and parser for the Fuyu language. Each task asked the subject to implement one syntactic construct in isolation (e.g., numeric literal or expression) – in each case, the subject was given a file with positive examples of that construct. The final two tasks asked subjects to write a parser for Hachiya given a sample source file and informal language specification – subjects were free to implement syntactic constructs in any order they wished. To finish any task, subjects ran a test suite (provided by us) to

check their solution. Subjects were not allowed to skip tasks or to proceed without passing all tests.

Hypothesis 1: We find that Parsimony significantly improves users' speed at constructing parsers. We focus on two measures of time-based performance: average completion time per task, and total progress made. We discuss total progress first. Figure 4 (left) shows the number of subjects completing each of tasks 1-9. In the time allotted, five experimental subjects progressed through all 9 tasks, but only two control subjects completed all tasks. The dropoff in the control group begins at task 4, which was one of the more complex tasks, as it involved construction of a grammar for algebraic expressions built from variables and literals. By contrast, the dropoff in the experimental group begins much later at task 7.

Figure 4 (center) shows average completion time for each task. We average across only subjects who successfully completed that task. The figure shows that Parsimony either matches or improves performance in the three most difficult tasks for the Fuyu language: 2, 4, and 5. Task 2 required constructing a lexer rule for numeric literals (recall Section II). Task 4, as already mentioned, covered algebraic expressions. Task 5 required writing productions for literal arrays. In all three cases, control subjects took approximately twice as long. We do note, however, that our results appear to show no significant speed advantage in tasks 8 and 9 – the averages for those tasks are biased toward the control group as they contain only a high-performing minority of the control group, compared against a larger subset of experimental group.

Hypothesis 2: We find that Parsimony significantly reduces the number of mistakes made. We measure two kinds of mistakes: compile errors (when the user specifies a bad rule that causes a compile failure); and reasoning errors (when the user specifies a rule that is later deleted or modified).

Compile Errors: Figure 4 (right) shows the per-user average number of compile errors. In lexer tasks (1-3), the experimental group significantly outperformed the control group, as evidenced by the first three columns of the figure. The experimental group also significantly outperformed the control group in parsing tasks (4-9). To more finely distinguish the contributing factors for this trend, we classify parser compile errors into two kinds: syntax and semantic errors. Syntax errors are self-explanatory. Semantic errors occur when the compiler fails a semantic check: (a) the user specifies a production that references an undefined symbol or introduces a non-productive cycle (i.e., groups of productions that permit infinite derivation), or (b) the user specifies disambiguating filters that are inconsistent with one another (e.g., the same production is both left and right associative). Control users averaged 8.3 syntax errors and 7.7 semantic errors, while experimental users averaged 2.4 and 4.4, respectively, a significant reduction in both categories. This indicates that Parsimony helps users not only avoid simple errors, such as typos, but also helps them reason about the relationships between productions to avoid conceptual mistakes.

Reasoning Errors: To measure reasoning errors in lexing tasks, we recorded every lexer rule written by the user. We call

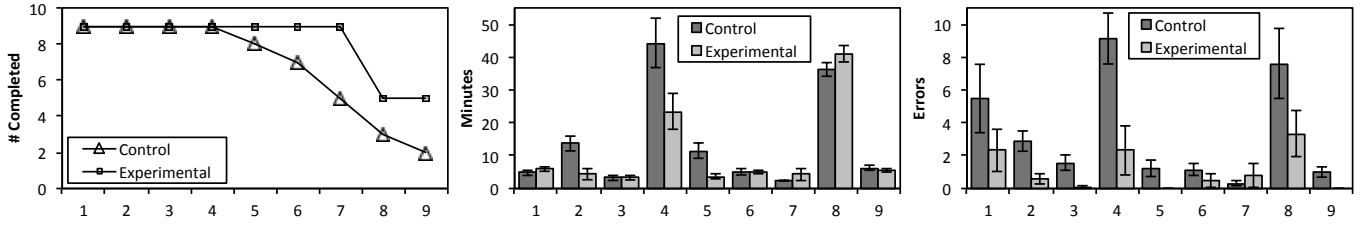


Fig. 4. (left) # subjects completing each task. (center) Mean time per task. (right) Mean compile errors per task. Error bars show standard error. X-axes show tasks numbered sequentially in time.

this the cumulative lexer rule set C_{lex} . We also recorded the set of rules appearing in the subject’s final answer. We call this the final lexer rule set F_{lex} . The ratio $|C_{lex}|/|F_{lex}|$ approximates the amount of *churn*: how much the subject edited their result in response to errors in reasoning. The experimental and control groups saw an average churn ratio of 1.2 and 3.1, respectively. In other words, the control group kept only 32% of rules and threw away more than two-thirds of attempted rules, while the experimental group kept nearly 81% of attempts.

We also measured the analogous churn ratio in parsing tasks. In the experimental condition, the average churn ratio was 1.31, which tells us that subjects kept a large majority (76%) of attempted productions in their final answer. The churn ratio of the control group was significantly higher at 1.99: nearly half of all productions they wrote were eventually discarded.

Survey: Subjects were given exit surveys to rate their experience. The averaged survey responses are summarized below. Each response is on a 1-5 scale, where 5 is best. E and C refer to experimental and control responses, respectively.

Question	E	Question	E	C
1 Coloring	4.78	7 Overall	4.78	4.00
2 Parse Tree Viz.	4.67	8 Easier to Use	4.56	4.00
3 Token Labels Tab	4.56	9 Recommend to Others	4.78	4.00
4 Solver Tab	4.78			
5 Sequence Pattern	4.56			
6 Expr. Pattern	4.67			

Questions 1-6 asked users to rate each listed experimental feature of Parsimony individually. Only the experimental group saw these questions. The last three questions were given to both groups. In order, these questions asked (7) if the tool was useful overall, (8) if the tool was easier to use than software they had previously used to construct parsers, and (9) if they would recommend the tool to others. Answers to questions 1-6 were uniformly positive, indicating agreement that every feature of Parsimony was valuable. Responses to questions 7-9 show that subjects in the experimental group favored Parsimony more than those of the control group, across all three metrics. In summary, the experimental group not only performed better, but also indicated a higher degree of satisfaction.

Threats to Validity: Subjects were UCSD CS students and thus may not be representative of programmers in general. Because the study employed toy languages, subjects’ performance may not be indicative of performance on real-world parsing tasks; to control for this, we designed the languages to model the syntax of real languages. Our experiments were single-blind, so there is a risk that subjects were influenced by

interaction with the proctor. To minimize this risk, users were given written, rather than oral instructions for each task.

VI. RELATED WORK

Parsimony is more expressive and usable than our previous system, Parsify [5], in three ways: (1) Parsimony can synthesize lexers, which Parsify cannot; (2) by making use of CYK automata to keep track of many candidate productions simultaneously, Parsimony is far less sensitive to the order in which examples are provided; and (3) whereas Parsify had ad-hoc support for a single kind of generalization, namely repetition, Parsimony has a more principled approach to generalization – extensible encoding via CYK automata. Finally, whereas Parsify had not been evaluated on end users, we perform a thorough evaluation of Parsimony with end users and report detailed statistics on the benefits of using Parsimony.

The class of algorithms for program synthesis via input/output examples is broadly termed programming-by-example (PBE) [12], [13], [14], [15]. Examples include synthesis of string transformations [16], [17], [18], spreadsheet manipulations [19], [20], number transformations [21], and data extraction from unstructured or semi-structured inputs [22].

We mention four program synthesis systems with a visual component similar to ours. The LAPIS [23] and SMARTedit [24] systems support repetitive edits by example, but export no underlying hierarchy. The iXj [25] system supports systematic edits to Java code, but only from single examples. The STEPS [26] system supports region highlighting and labeling like Parsimony, but toward the narrower aim of generating text transforms like those performed by short shell scripts. We distinguish our work from these prior systems by our broader scope: parsers for context-free languages.

Research on *grammatical inference* [27], [28] focuses on the theory of inferring structure from text. We contrast our work by our practical focus: constructing syntax specifications that not only parse the given text, but that also give rise to meaningful parse trees comprehensible to a software engineer.

VII. CONCLUSION

We have presented Parsimony, an IDE for synthesizing lexers and parsers by example. Parsimony makes use of two novel graph data structures, the R-DAG and CYK automaton, for efficiently inferring lexical rules and grammar productions, respectively. Through a controlled study, we have demonstrated Parsimony’s effectiveness at helping users complete parsing and lexing tasks more quickly and with fewer errors.

REFERENCES

- [1] T. Parr and K. Fisher, “LL(*): The foundation of the ANTLR parser generator,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 425–436.
- [2] B. Ford, “Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl,” in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’02. New York, NY, USA: ACM, 2002, pp. 36–47.
- [3] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) parsing: The power of dynamic analysis,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’14. New York, NY, USA: ACM, 2014, pp. 579–598.
- [4] E. Scott and A. Johnstone, “GLL parsing,” *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 177 – 189, 2010, proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).
- [5] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 565–574.
- [6] D. Angluin, “Learning regular sets from queries and counterexamples,” *Information and Computation*, vol. 75, no. 2, pp. 87 – 106, 1987.
- [7] D. H. Younger, “Recognition and parsing of context-free languages in time n^3 ,” *Information and Control*, vol. 10, no. 2, pp. 189–208, 1967.
- [8] P. Klint and E. Visser, “Using filters for the disambiguation of context-free grammars,” in *Proc. ASMICS Workshop on Parsing Theory*, 1994, pp. 1–20.
- [9] M. Thorup, “Disambiguating grammars by exclusion of sub-parse trees,” *Acta Informatica*, vol. 33, no. 5, pp. 511–522, 1996.
- [10] grammars v4, <https://github.com/antlr/grammars-v4>, 2016.
- [11] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Upper Saddle River, NJ, USA: Ellis Horwood, 1990.
- [12] A. Cypher, Ed., *Watch What I Do – Programming by Demonstration*. Cambridge, MA, USA: MIT Press, 1993.
- [13] S. Gulwani, “Synthesis from examples: Interaction models and algorithms,” in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, Sept 2012, pp. 8–14.
- [14] H. Lieberman, *Your Wish is My Command: Programming by Example*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.
- [15] O. Polozov and S. Gulwani, “FlashMeta: A framework for inductive program synthesis,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 107–126.
- [16] A. Arasu, S. Chaudhuri, and R. Kaushik, “Learning string transformations from examples,” *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 514–525, Aug. 2009.
- [17] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’11. New York, NY, USA: ACM, 2011, pp. 317–330.
- [18] A. K. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. T. Kalai, “A machine learning framework for programming by example,” in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML’13. JMLR.org, 2013, pp. 187–195.
- [19] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn, “Flashrelate: Extracting relational data from semi-structured spreadsheets using examples,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 218–228.
- [20] W. R. Harris and S. Gulwani, “Spreadsheet table transformations from examples,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 317–328.
- [21] R. Singh and S. Gulwani, “Synthesizing number transformations from input-output examples,” in *Proceedings of the 24th International Conference on Computer Aided Verification*, ser. CAV’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 634–651.
- [22] V. Le and S. Gulwani, “FlashExtract: A framework for data extraction by examples,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 542–553.
- [23] R. C. Miller and B. A. Myers, “Lightweight structured text processing,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 10–10.
- [24] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, “Programming by demonstration using version space algebra,” *Mach. Learn.*, vol. 53, no. 1-2, pp. 111–156, Oct. 2003.
- [25] M. Boshernitsan, S. L. Graham, and M. A. Hearst, “Aligning development tools with the way programmers think about code changes,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’07. New York, NY, USA: ACM, 2007, pp. 567–576.
- [26] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai, “A colorful approach to text processing by example,” in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’13. New York, NY, USA: ACM, 2013, pp. 495–504.
- [27] C. de la Higuera, “A bibliographical study of grammatical inference,” *Pattern Recogn.*, vol. 38, no. 9, pp. 1332–1348, Sep. 2005.
- [28] E. Vidal, “Grammatical inference: An introductory survey,” in *Grammatical Inference and Applications*, ser. Lecture Notes in Computer Science, R. Carrasco and J. Oncina, Eds. Springer Berlin Heidelberg, 1994, vol. 862, pp. 1–4.