Interactive Parser Synthesis by Example

Alan Leung John Sarracino Sorin Lerner

University of California, San Diego
{aleung,jsarraci,lerner}@cs.ucsd.edu

Abstract

Despite decades of research on parsing, the construction of parsers remains a painstaking, manual process prone to subtle bugs and pitfalls. We present a programming-by-example framework called Parsify that is able to synthesize a parser from input/output examples. The user does not write a single line of code. To achieve this, Parsify provides: (a) an iterative algorithm for synthesizing and refining a grammar one example at a time, (b) an interface that provides immediate visual feedback in response to changes in the grammar being refined, and (c) a graphical mechanism for specifying example parse trees using only textual selections. We empirically demonstrate the viability of our approach by using Parsify to construct parsers for source code drawn from Verilog, SQL, Apache, and Tiger.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.4 [*Programming Languages*]: Processors – Parsing; H.5.2 [*Information Interfaces and Presentation*]: User Interfaces

General Terms Algorithms, Human Factors, Languages

Keywords Parsing, Programming by Example, Program Synthesis

1. Introduction

> *Bison parsers are shift/reduce automata. In some cases (much more frequent than one would hope), looking at this automaton is required to tune or simply fix a parser.*
> (Bison 3.0.2 User Manual)

Parsing is a pervasive programming task that is still difficult for non-experts, despite decades of research into the subject. Mainstream parser generators like Bison offer high performance but at the cost of a steep learning curve: as Bison's developers admit themselves, an understanding of shift/reduce automata theory is a necessary prerequisite. More modern incantations of parser technologies such as ANTLR [32] and Packrat parsing [11, 13] pave the way for more user-friendly syntax specifications, but even so are subject to subtle gotchas requiring an understanding of their underlying parsing strategies. For instance, ANTLR and other LL-based top-down parsers disallow use of mutually left-recursive productions such as $E \to T$ and $T \to E + T$, which arise naturally when specifying the form of binary expressions and other recursive forms. Although it is possible to rewrite such productions to avoid left recursion, the standard algorithm for doing so leads to an explosion in the grammar [31]. Thus, in practice, parser writers must search for a more concise refactoring – finding such refactorings can be an art, as better algorithms are not known.

Packrat parsers seek to simplify parsing by eliminating ambiguity via *ordered choice*: the first alternate to match a string is always chosen. Unfortunately, use of ordered choice introduces a particularly subtle quirk: a production such as $A \to \texttt{a}|\texttt{ab}$, which we might naturally expect to match either the string `a` or `ab`, cannot in fact ever match `ab` because `a` is a prefix of `ab`. Although a contrived example, this situation arises in practice whenever one alternate can match the prefix of another, such as when matching *if* and *if-else* blocks. Finally, the advent of efficient, generalized parsing strategies such as GLL [34] and GLR [37] promise the ability to use any context-free grammar without restriction, seemingly solving all our problems. Unfortunately, the price of using a generalized parser is the freedom to specify grammars rife with ambiguities if left unchecked. The user is left with the unenviable task of sifting through the resulting *parse forests*. Detecting ambiguities, let alone fixing them, can be a difficult undertaking as the general problem is undecidable. Given the numerous options, perhaps the most daunting task a non-expert programmer must face is the decision of what parsing technology to even choose in the first place: each has its own dark corners, and there is no clear-cut winner.

The difficulty of constructing parsers has given rise to a particularly troubling phenomenon dubbed "cargo cult parsing," [28] wherein programmers eschew established parsing technologies in favor of ad-hoc regular expressions, often copied directly from web search results. Clearly, there is a need for tools to bridge the gap between established parsing theory and actual practice.

Programming-by-example (PBE) is a promising approach to bridging that gap. PBE is a programming paradigm in which end users synthesize a program by providing sample inputs and outputs demonstrating the result of an intended computation. PBE has been applied to problems from diverse domains including text editing [22], spreadsheet table transformations [17], and data extraction from ad-hoc logs [10]. PBE presents an attractive option for situations in which it is much easier to demonstrate correct behavior (e.g., the correct parse of an example string), than to provide a specification of that behavior (e.g., a formal grammar specification accepted by a parser generator). Although much progress has been made towards synthesis of programs for analyzing unstructured or semi-structured strings, these results have thus far not extended to construction of parsers for context-free languages.

We present Parsify, a graphical, interactive system for incrementally synthesizing and testing parsers. In contrast to previous PBE approaches that attempt to continuously maintain a (potentially large) set of programs consistent with an increasing set of examples, Parsify presents an *exploratory* interface in which the user poses examples that induce the shape of production rules in the underlying grammar being inferred. These examples are presented in the form

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI'15, June 13–17, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/

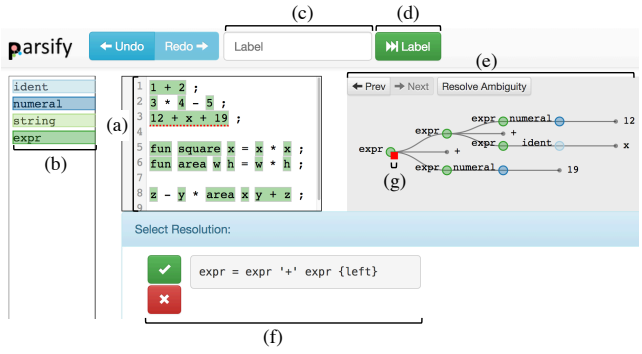

Figure 1. The Parsify user interface: (a) File View, (b) Legend, (c) Label Box, (d) Label Button, (e) Parse Tree Pane, (f) Resolution Pane, and (g) Negative Label

of text selections (i.e., using a mouse) in an editor window presenting the file the user wishes to parse. The user then examines the result of each inference step visually, as Parsify presents an overlay with colored regions (and corresponding parse trees) that are continuously updated to reflect each inference. In particular, ambiguities are presented immediately after the offending production has been inferred, allowing the user to either (a) ask Parsify to automatically synthesize a disambiguating strategy, or (b) undo the last inference and proceed along another path to avoid the ambiguity.

Contributions

1. We present a novel application of programming-by-example to the domain of parsers for context-free languages (Sections 2–3).
2. We present techniques and algorithms for efficiently visualizing progress, inferring production rules, and synthesizing disambiguating filters by example, including novel uses of generalized parsers and A^*-search (Section 3).
3. We evaluate our approach's effectiveness via case studies: we have generated parsers for a test suite consisting of Verilog modules, Apache logs, Tiger programs, and SQL queries (Section 4).

2. Overview

We begin with a series of examples illustrating how a user might employ Parsify to implement the parser for a small untyped functional language. As we layer additional features into the concrete syntax we will see how Parsify is able to handle various practical difficulties that arise during parser development. The example is kept very simple for the purpose of exposition. Our tool can actually handle much more complicated languages and grammars, as described in our evaluation in Section 4. Also note that for the purpose of exposition, we describe features using textual descriptions where possible, although the actual implementation of Parsify provides a fully graphical user interface.

2.1 User Interface Overview

Figure 1 shows a sample view of the Parsify user interface with several features highlighted. We now briefly describe each feature. The File View (a) and Legend (b) together show parsing progress on the current file: each color represents a syntactic category (i.e., nonterminal), as described shortly. The Label Box (c) and Label Button (d) allow the user to annotate substrings in the file with labels. The Parse Tree Pane (e) shows the parse tree at the current cursor location in the File View. The Resolution Pane (f) is used to resolve ambiguities. As we progress through our example, each feature will be explained in further detail.

2.2 Basic Inference

To start defining a parser for a simple functional language, the user first constructs a file with the following arithmetic expressions:

```
1 + 2 ;
3 * 4 - 5 ;
12 + x + 19 ;
```

Every Parsify session begins with default predefined production rules that encode basic tokens such as integers and alphanumeric identifiers, along with the standard assumption that whitespace is a discarded token separator. Using these default rules, Parsify colorizes various substrings that can be derived from the built-in ***ident*** or **numeral** rules, where uncolored regions represent substrings that cannot yet be derived from any rule in the grammar:

```
1 + 2 ;
3 * 4 - 5 ;
12 + x + 19 ;
```

The above colorization is our textual representation of the UI mechanism shown in the File View of Figure 1. For the paper to be readable in black-and-white, we also use **bold-face** font for red, ***overline_italics*** for blue, and underline for green. The user's first interaction is to "teach" Parsify that a literal numeral is a form of expression. To do this, the user selects the substring `2` in the File View, types `expr` in the Label Box, and then clicks on the Label Button. This instructs Parsify to apply the label **expr** to the selected string `2`, which causes Parsify to infer a new production rule to add to the grammar: `expr` → `numeral`. Whenever making an inference, Parsify immediately recolors its output to represent the change. In particular, the first line now becomes `1 + 2` to reflect the fact that the substrings `1` and `2` can be derived from the new rule for **expr**. Notice that there now exist two valid colorizations for the substring `1` (and likewise `2`), as they can be derived from either of rules **expr** and **numeral** – in such cases, Parsify prefers **expr** as it corresponds to the "more general" production (we informally define the more general production as the one higher in the parse hierarchy – we defer to Section 3 a formal definition). Following the same procedure, the user can likewise select the identifier ***x*** on the third line and apply label **expr**, from which Parsify infers `expr` → `ident` and produces the following colorization in which all identifier and numerals are correctly parsed as expressions.

```
1 + 2 ;
3 * 4 - 5 ;
12 + x + 19 ;
```

2.3 Infix Expressions

The user now proceeds to binary infix expressions by applying label **expr** to the substring `1 + 2`, which allows Parsify to infer the new production rule `expr` → `expr + expr`.

Associativity At this point, the user has unwittingly introduced an ambiguity into the grammar, a concrete example of which is found on line 3. Parsify immediately detects that line 3 has an ambiguous parse and visually depicts it with a red dashed underline: `12 + x + 19`. The ambiguity results because the `expr` rule just introduced allows for both left-associative and right-associative parses: `[12 + x] + 19` and `12 + [x + 19]`. It is in this situation that existing parser generators such as Bison or ANTLR would require some modification to the syntax specification to remove the offending construct. Parsify shields the user from performing such modifications manually by simply presenting both parse trees visually and asking the user to choose the correct one. The different parse trees are shown one at a time in the Parse Tree Pane, as shown in Figure 1, with Next and Prev buttons to browse through the different trees, and a Resolve Ambiguity button to resolve the ambiguity

using the currently displayed tree. The Parse Tree Pane of Figure 1 is precisely in the middle of such an ambiguity resolution phase. Let's assume the user intends + to be a left-associative operator, thus choosing the left-associative parse tree. To implement this preference, Parsify makes use of so-called *disambiguating filters* [20], which can be used to remove unwanted parses. More specifically, in this case Parsify automatically synthesizes an *associativity filter*, written `expr → expr + expr {left}`, which disallows derivation of trees with form `expr + [expr + expr]`. The filter that would be used to resolve the conflict is displayed in the Resolution Pane at the bottom of the UI, as shown in Figure 1.

Priority Now the user proceeds similarly for the * and – operators, teaching Parsify the production rules `expr → expr * expr` and `expr → expr - expr` by applying the label **expr** to substrings `3 * 4` and `3 * 4 - 5` in sequence. This exposes a new ambiguity evidenced on line 2, `3 * 4 - 5`, due to the fact that no precedence has been specified between the * and – operators. Parsify presents two valid parse trees, `[3 * 4] - 5` and `3 * [4 - 5]`, from which the user chooses the first: the * operator should bind tighter than the – operator. Parsify is able to synthesize a *priority filter* that disallows derivation of trees that give – higher precedence than the * operator: `expr → expr * expr > expr → expr - expr`. Note that this is not the only filter that discriminates between the two parses. Another valid filter that disallows the second parse tree would be `left { expr → expr * expr ; expr → expr - expr }`, which specifies that the * and – operators are left-associative with respect to one another. This is the reason for displaying the actual filter at the bottom of the UI in the Resolution Pane, as shown in Figure 1. If the proposed filter is not the one the user intended, Parsify provides the option of rejecting the suggested filter by clicking the red box marked X. The synthesis algorithm then proceeds to search for another filter that also satisfies the user's preference of parse tree. In the above example, Parsify would first present the priority filter. If the user rejects this filter, Parsify's next suggestion would be the left-associative filter.

2.4 Function Definitions

The user now adds functions to the language. To begin, the user appends to the current file some examples of function definitions.

```
fun square x = x * x ;
fun area w h = w * h ;
```

Notice that some of the colors are incorrect: in particular, it appears the `fun` keywords are incorrectly identified as **expr**s. Of course, this is to be expected because we have not given Parsify any indication that the sequence of characters "`fun`" should be treated any differently than other identifiers.

Negative labels To inform Parsify of the mistake, the user can apply *negative labels* in the Parse Tree Pane against the labeling on both keywords. The user does this by clicking on the nodes in the parse tree whose labeling is wrong, and then clicking on the red box that appears next to the node. An example of this mechanism being applied to an **expr** label is shown at (g) in Figure 1. In response to the negative labels, Parsify now refines its output:

```
fun square x = x * x ;
fun area w h = w * h ;
```

This is almost, but not quite what the user wants – the function names and formal parameters have been identified as **expr**s as well, but the desired syntax restricts them to be bare identifiers. Thus we apply negative labels against the **expr** label on all offending substrings, resulting in the following:

```
fun square x = x * x ;
fun area w h = w * h ;
```

Generalization Now the user applies the label `fundef` to each of the two lines above. If Parsify follows the process described so far, it would produce two basic production rules of the form

```
fundef → 'fun' ident ident = expr ;
fundef → 'fun' ident ident ident = expr ;
```

Although these rules parse the given program, they preclude function definitions that have greater than 2 parameters. Thus, Parsify detects such redundant productions and infers the generalization

```
fundef → 'fun' ident+ = expr ;
```

in which an arbitrary number of parameters is permissible.

2.5 Function Calls

Now let us turn our attention to the last feature the user will add: syntax for function calls. As in OCaml or Haskell, the user wishes a function call to take the form of expressions separated by whitespace. The user adds one last example to the file, an expression containing a function call:

```
y * area x y + y - z ;
```

Unfortunately, this colorization is quite far from the user's desire. It seems `y * area` has been parsed as an **expr** even though `area` should be the beginning of the function call `area x y`. The root cause, of course, is that we have not provided an example of a function call to Parsify, so it does not know to treat **area x y** as a new kind of expression.

The user provides 3 negative labels to tell Parsify that it has incorrectly colored various parts of our expression:

```
y * area x y + y - z ; → negate y * area
y * area x y + y - z ; → negate y + y - z
y * area x y + y - z ; → negate y + y
y * area x y + y - z ;
```

Now the user selects `area x y` and applies label call, then **expr**, to reflect that a call is a kind of expression.

```
y * area x y + y - z; → apply call: area x y
y * area x y + y - z; → apply expr: area x y
```

At this point, Parsify correctly colors the full expression but also reveals an ambiguity: `y * area x y + y - z`. There are 9 valid parse trees for this string, but the intended parse groups subexpressions to the left and gives function calls highest precedence:

```
[[[y * [area x y]] + y] - z]
```

After the user chooses the intended parse tree, Parsify is able to synthesize the following set of disambiguating filters,

```
left {
  expr → expr + expr
  expr → expr - expr
}
expr → call > expr → expr * expr
```

which specify that the + and – operators are left-associative with respect to one another, and that function calls have higher precedence than *, as expected.

2.6 Challenges

To achieve this level of interaction, we address several challenges:

1. *What is a concise, natural way of presenting partial progress to the user?* Although we experimented with many representations, we found the most natural representation was that of a *coloring* in which different nonterminals of the grammar correspond to different colors, and colored regions are "as big as possible."

2. *How do we achieve performance capable of supporting interactive use?* The interface would be unusable if the user were forced to wait long periods of time between colorings. Our solution employs a greedy algorithm for generating colored labels based on ranking of *partial parses* generated by a GLL parser.
3. *How can we synthesize disambiguating filters in a more principled way than brute force?* Even with a small parse tree, the number of possible disambiguations can grow exponentially (via subset construction). Our solution formulates synthesis as an instantiation of A^*-search to avoid unlikely candidates.

Section 3 details our solutions to these challenges.

3. Algorithm

In this section we formalize the core algorithms employed by Parsify for inferring context-free grammars. We begin with a preliminary overview of context-free grammars, parsing, and disambiguating filters, but point the reader to [1, 14, 20] for thorough coverage.

3.1 Preliminaries: Context-Free Grammars

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 $V = N \cup \Sigma$ is the set of symbols called the *vocabulary* of G. Unless otherwise stated we will use the following notational conventions throughout this paper: the upper case letters A, B, C are nonterminals, the lower case letters a, b, c are terminals, the lowercase Greek letters $\alpha, \beta, \gamma, \mu$ are (possibly empty) strings of symbols in V, the letter w is a (possibly empty) string of terminals in Σ, and productions $(A, \beta) \in P$ are written equivalently as $A \to \beta$. We write $G(A)$ to mean the grammar G with start nonterminal replaced by A.

String Indexing String indices begin at 0. We write $\alpha[i]$ to mean the symbol at index i of string α. The notation $\alpha[i...]$ denotes the suffix of α starting at index i. We write $|\alpha|$ to mean the length of α and $\alpha \cdot \beta$ 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 $A \to \mu$ 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 $(\alpha, \beta) \in \Rightarrow^*$, and a *derivation of* β *from* α is a sequence $\alpha \Rightarrow ... \Rightarrow \beta$ 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 the language of G, written $L(G)$, is the set of all its sentences. A *terminated derivation* is a derivation whose final element is a sentence. A *full derivation* is a terminated derivation whose initial element is the start nonterminal S. We define $D_G(w)$ to be the (possibly empty) set of all full derivations of sentence w with respect to G. We write $D(w)$ when G is clear from context.

Parse Trees A *parse tree* is a tree t such that: (a) every internal node is labeled with a nonterminal in N, (b) every leaf node is labeled with a terminal in Σ, (c) for every internal node with label A and children with labels $v_1, ..., v_n \in V$, there exists a production $A \to v_1 ... v_n \in P$. For ease of textual representation, we depict trees as nested bracketed forms $[L\ t_1\ ...\ t_n]$, where L is the node label, and each of $t_1, ..., t_n$ are themselves either bracketed forms or leaf labels. The *yield* of t, written $yield(t)$, is the string formed by concatenating its leaf node labels (e.g., $yield([A\ [Ba]b]) = ab$). The *signature* of t, written $sig(t)$, is the production corresponding to the root of t (e.g., $sig([A\ [Ba]b]) = A \to Bb$). The *root symbol* of t, written $rootSymbol(t)$, is the symbol at the root node of t (e.g., $rootSymbol([A\ [Ba]b]) = A$).

For any derivation d we can construct its corresponding parse tree t by induction on the elements of d. Let $tree(d)$ be the map from derivations to their parse trees. We can now define the set of parse trees of a sentence w as follows: $trees(w) = \{tree(d) \mid d \in D(w)\}$ which we extend to grammars naturally: $trees(G) = \bigcup_{w \in L(G)} trees(w)$.

Index Trees The *index tree* $\hat{t}$ of t is the tree isomorphic to t, with identical internal node labels, but with its leaf labels replaced from left to right by consecutively increasing integers starting from 0. For example, the index tree of $[A\ [Ba]b]$ is $[A\ [B\ 0]\ 1]$. We define $index(t)$ to be the map from parse trees to their index trees. The *span* of index tree $\hat{t}$, written $span(\hat{t})$, is the pair (i, j) of the labels of its leftmost and rightmost leaves, respectively.

Ambiguity A sentence w is *ambiguous with respect to* G iff $\exists\ d_1, d_2 \in D_G(w).\ tree(d_1) \neq tree(d_2)$. For brevity, we say w is *ambiguous* when G is clear from context. A grammar G is ambiguous iff $L(G)$ contains an ambiguous sentence.

Parsers A *parser* p is a function of type $G \times \Sigma^* \to \mathcal{P}(trees(G) \times \Sigma^*)$ that given a grammar G and string w, returns a set of tuples, called *parses*, with two elements: a parse tree t for some non-empty prefix of w, and a string suffix w' such that $w = yield(t) \cdot w'$. In other words, a parser does not necessarily consume its entire input string, and thus returns the unconsumed portion. A *full parser* $\overline{p}$ is a parser that always consumes its entire input or not at all: the second element of each parse must be the empty string. (Note that this does not mean a full parser always produces a parse tree for any string: if a string $w \notin L(G)$ then $\overline{p}(G, w) = \emptyset$.)

Generalized Parsers A parser p is generalized iff $\forall\, w \in L(G)$, $w' \in \Sigma^*.\ p(G, w \cdot w') \supseteq trees(w) \times w'$. In other words, generalized parsers produce all possible parse trees for all inputs. Real-world examples of generalized parsers are GLL or GLR-based parsers such as instaparse [19] or Elkhound [26], respectively. In the remainder of this paper, let p_{GLL} be such a generalized parser (our implementation uses the instaparse GLL parser).

3.2 Preliminaries: Disambiguating Filters

We formalize disambiguating filters as predicates on trees: the intuition is that whenever the predicate evaluates to true, we say that the tree is invalid and removed from the parser's output set. Disambiguating filters provide a declarative approach to removing ambiguity from syntax specifications without resorting to rewriting the grammar's productions.

More formally, let $\pi(t)$ be a predicate on trees, and let p be a parser. The *disambiguation of* p *with respect to* π, written $p|_\pi$ is defined as follows:

$$p|_\pi(G, w) = \{(t, w') \mid \neg\pi(t) \wedge (t, w') \in p(G, w)\}$$

The composition of two filters is simply disjunction: $(\pi_1 \circ \pi_2)(t) = \pi_1(t) \vee \pi_2(t)$; we lift disambiguations to sets of filters Π naturally:

$$p|_\Pi = \left\{ (t, w') \mid \neg(\bigvee_{\pi \in \Pi} \pi(t)) \wedge (t, w') \in p(G, w) \right\}$$

Note that a naive implementation of the above would simply discard parse trees as a postprocessing step after parsing, which can be extremely expensive if the number of parse trees is high. Our actual implementation embeds filters into the parser's internals such that trees are discarded early.

Associativity Filters Associativity filters rule out a common form of ambiguity that arises when there exists a sentential form $A \otimes A$ in which two valid derivations are $A \Rightarrow^* A \otimes A \Rightarrow^* \alpha \otimes \beta \otimes A \Rightarrow^* \alpha \otimes \beta \otimes \gamma$ and $A \Rightarrow^* A \otimes A \Rightarrow^* A \otimes \beta \otimes \gamma \Rightarrow^* \alpha \otimes \beta \otimes \gamma$.

The underlying problem is that the two derivations induce trees of different shape: $[A\ [A\ \alpha \otimes \beta] \otimes [A\ \gamma]]$ and $[A\ [A\ \alpha] \otimes [A\ \beta \otimes \gamma]]$, respectively. We define here a constructor for left-associativity filters that given a set of productions R, returns a filter that rejects trees containing non-left-associative uses of any production in R:

$$f_{\pi_L}(R) = \lambda t.\exists\, t' \in nodes(t).$$
$$\begin{cases} \bigvee_{i=1}^{n} (sig(t') \in R \wedge sig(t_i) \in R) & \text{if } t' = [A\ t_0 ... t_n] \\ false & \text{otherwise} \end{cases}$$

In words, left-associativity filters reject any tree in which a parent and a child in non-leftmost position each have a signature in R. Analogously, right-associativity filters do so for non-rightmost positions, and non-associativity filters simply disallow any child from sharing its parent production. We omit formal definitions of right- and non-associativity filter constructors f_{π_R} and f_{π_N} as they are immediately analogous.

Priority Filters We now define simple priority filters based on a relative priority between two productions.

$$f_{\pi_>}(r_h, r_l) = \lambda t.\exists\, t' \in nodes(t).$$
$$\begin{cases} sig(t') = r_h \wedge \bigvee_{i=0}^{n} sig(t_i) = r_l & \text{if } t' = [A\ t_0 ... t_n] \\ false & \text{otherwise} \end{cases}$$

Priority filters reject any tree in which a child's production has lower priority than its parent's.

Consistency Because filters are simply predicates on trees, it is possible that a composition of filters gives rise to a trivially satisfiable predicate $(\pi(t) \vee \pi'(t)) \leftrightarrow true$. Such a composition rejects all trees (e.g., consider the composition of two filters that specify left- and right-associativity of the same operator). In this case, we say the set of filters is inconsistent, and otherwise *consistent*.

3.3 Interaction

We now formalize our model of user interaction as a core set of user operations that transition between *session states*.

3.3.1 Session State

Intuitively, a *session state* encapsulates a hypothesis for the grammar and set of disambiguating filters inferred from examples seen so far. After any user operation, this hypothesis is updated to reflect new information. A session state σ is a tuple $(G, \Pi, M, w, \mathcal{C})$ where G is a grammar, Π is a set of disambiguating filters, M is a set of labels (the *negative labels* of σ), w is the text we wish to parse (a string of terminals in the alphabet of G), and $\mathcal{C}$ is a *coloring* on w.

A *label* is a tuple $(A, i, j) \in N \times \mathbb{N} \times \mathbb{N}$. That is, a label contains a nonterminal together with a start index (inclusive) and end index (exclusive) that index into the string w. The set of all labels is $\mathbb{L}$. A *coloring* $\mathcal{C} \subseteq \mathbb{L}$ is simply a set of labels. The intuition is that each label in a coloring corresponds 1-to-1 with a single colored region in the interface: our interface graphically presents a different color for each nonterminal. For example, if $(\texttt{expr}, 3, 10) \in \mathcal{C}$, then Parsify colors the seven character substring, starting at index 3, with the color corresponding to `expr`.

3.3.2 Operations

The following 5 atomic operations comprise the building blocks for user-facing interactions in Parsify:

1. DRAW: compute a new coloring.
2. ANNOTATE: accept a new label.
3. GENERALIZE: generalize an existing production.
4. NEGATE: reject an existing label.
5. RESOLVE: synthesize a new disambiguating filter.

In particular, each action performed by the user maps to a *sequence of operations* as follows:

1. *Apply Label:* ANNOTATE → GENERALIZE → DRAW
2. *Reject Label:* NEGATE → DRAW
3. *Disambiguate:* RESOLVE → DRAW

Note that we intentionally omit two auxiliary features of our interface from the formalism: (a) visualizations of parse trees, which are just visual sugar for the underlying parse trees, and (b) red dashed underlines under ambiguous regions, which are applied as a postprocessing step on the editor view after generating a coloring.

We now define the semantics of each atomic operation as functions from session state to session state. We use the notation $[\![O]\!]\sigma$ to denote the result of executing operation O on state σ. We define the initial session state to be $\sigma_0 = (G_0, \emptyset, \emptyset, w, \emptyset)$, where w is the string being parsed and G_0 is an initial grammar containing only predefined, basic productions for tokens such as identifiers and numbers. To ease exposition, we make the simplifying assumption that inputs contain no contiguous region of more than one whitespace symbol, although as previously mentioned, our actual implementation handles arbitrary whitespace by discarding whitespace at token boundaries.

Draw Our system relies crucially on presenting colorings that correspond to likely sentential forms in the language being parsed. To do this, we define a comparison function $better$ that prefers parses according to the following metric: (a) prefer parses that consume more text, and (b) when the yield of two parse trees are of the same length, prefer the tree that subsumes the other. Subsumption is determined by constructing a preorder $\sqsubseteq^*_G$ on the nonterminals of the grammar such that tree t subsumes tree t' iff $rootSymbol(t') \sqsubseteq^*_G rootSymbol(t) \wedge rootSymbol(t) \not\sqsubseteq^*_G rootSymbol(t')$. Intuitively, $A \sqsubseteq^*_G B$ if there may exist a parse tree with root symbol B that contains a node with symbol A. Formally, let $G = (N, \Sigma, P, S)$. We define $\sqsubseteq^*_G$ as the transitive closure of binary relation $\sqsubseteq_G$, where $A \sqsubseteq_G B$ iff $A = B \vee (B \to \alpha A \beta \in P)$. We can then sort parses according to $better$ and choose the root symbol of the highest rated parse tree to be part of a member of the new coloring (in the case of ties, we simply choose one).

The COLOR function that actually computes a new coloring is a simple greedy algorithm that performs a linear scan, accepting the best parse found at each examined position.

```
1:  function COLOR(σ)
2:      let (G, Π, M, w_0, _) = σ
3:      let (N, _, _, _) = G
4:      C ← ∅; w ← w_0; n ← 0
5:      while |w| > 0
6:          let X = {(t, w') |
7:                  ∃ A ∈ N. (t, w') ∈ p_GLL|_Π(G(A), w) ∧
8:                  ∀ t̂ ∈ nodes(index(t)).LABEL(t̂, n) ∉ M}
9:          if X ≠ ∅
10:             let (t, w') = first(sortBy(better, X))
11:             C ← C ∪ { LABEL(index(t), n) }
12:             w ← w'; n ← n + |yield(t)|
13:         else
14:             w ← w[1...]; n ← n + 1
15:     return C
16: function LABEL(t̂, n)
17:     let (i, j) = span(t̂)
18:     return (rootSymbol(t̂), i + n, j + 1 + n)
```

With COLOR defined, we can now define the operation DRAW, which simply threads a new coloring into the session state:

$$[\![\text{DRAW}]\!](\sigma = (G, \Pi, M, w, \mathcal{C})) = (G, \Pi, M, w, \text{COLOR}(\sigma))$$

An important consideration is that it is possible for the computed coloring to be incorrect, in the sense that the user does not agree with the label assigned to some part of the text. (Recall from Section 2.4 that this occurred when `fun` keywords were incorrectly identified as instances of **`expr`**.) In such cases, it is important that the user be permitted to inform Parsify that it has made a mistake. The NEGATE operation, which we define next, allows the user to do exactly that.

Negate The NEGATE operation is the user's mechanism for specifying that a coloring is incorrect. In particular, negation of a label tells Parsify that in subsequent DRAW operations, (a) that label cannot appear in a coloring again, and (b) no parse tree whose subtrees induce the negated label may be considered when computing a new coloring. Line 8 of function COLOR performs this check. The definition of the NEGATE operation is then almost trivial: we simply add the negated label to the set of negative labels M in the session state.

$$[\![\text{NEGATE}(A, i, j)]\!](\sigma = (G, \Pi, M, w, \mathcal{C})) = \\ (G, \Pi, M \cup \{(A, i, j)\}, w, \mathcal{C})$$

Returning to our running example, consider the situation from Section 2.4 in which the user wished to tell Parsify that the substring "`fun`" at indices 0 through 3 was incorrectly identified to be an **`expr`**. In the UI, the user applied a red box to the offending parse tree node, which caused the interface to immediately refresh with a corrected coloring. Under the hood, Parsify actually performed the operation NEGATE(`expr`, 0, 3), followed immediately by a DRAW operation to regenerate a new coloring respecting the new constraint.

Annotate When the user selects a region of text and applies a name to the selection, the underlying operation is an ANNOTATE operation that generates a new production using the selected region as a template for the body of the production.

```
1:  function GEN-PROD(σ, A, i, j)
2:      let (_, _, _, w, C) = σ
3:      idx ← i; β ← []
4:      while idx < j
5:          let X = {(A', i', j') ∈ C | idx = i' ∧ j' < j}
6:          if |X| = 1
7:              let {(A', _, j')} = X
8:              β ← β · A'
9:              idx ← j'
10:         else if |X| = 0
11:             β ← β · w[idx]
12:             idx ← idx + 1
13:         else // unreachable
14:     return A → β
```

Informally, GEN-PROD scans the selected range from left to right looking for labels in $\mathcal{C}$ that fit within the selected range. Intuitively, in the user interface this corresponds to a textual selection in the File View – for every colored region contained within the selection, Parsify adds the corresponding nonterminal to the production body being inferred (Lines 6–9). If Parsify finds a terminal that is uncolored, then Parsify simply appends the terminal to the inferred production body (Lines 10–12). The branch body on Line 13 is unreachable because its corresponding branch predicate is satisified when $|X| > 1$, which can only happen when the coloring contains overlapping labels. However, COLOR never produces overlapping labels due to the increment on Line 12 of function COLOR.

The definition of ANNOTATE generates a new production with GEN-PROD, then simply threads the production into the grammar.

$$[\![\text{ANNOTATE}(A, i, j)]\!]\sigma = (G', \Pi, M, w, \mathcal{C})$$

$$\begin{aligned} \text{where } (G, \Pi, M, w, \mathcal{C}) &= \sigma \\ (N, \Sigma, P, S) &= G \\ P' &= P \cup \{\text{GEN-PROD}(A, i, j)\} \\ G' &= (N \cup A, \Sigma, P', S) \end{aligned}$$

Generalize The GENERALIZE operation provides Parsify the ability to expand the grammar by permitting arbitrary repetition of strings in a controlled fashion. For this purpose, we extend our grammars with a standard meta-syntax for repetitions borrowed from Extended Backus-Naur Form's regular expressions: α^+ (and α^*) for 1 or more (and 0 or more) repetitions of α. Note that these constructs do not increase expressive power beyond context-free grammars and can be desugared into forms without explicit repetition [14].

Given two production bodies, the function GEN attempts to find a *compatible partition* of both bodies. Informally, a compatible partition of two strings $\alpha, \beta \in V^*$ is a 1-to-1 correspondence between non-empty substrings of α and β such that corresponding substrings are either (a) exactly equal, or (b) both consistent with some number of repetitions of the same sequence of symbols, possibly separated by occurrences of a single delimiter symbol. For example, suppose $\alpha = BAAC; C$ and $\beta = BAAAC$. Then a compatible partition of α and β would be $B, AA, C; C$ and B, AAA, C because B equals B, AA equals AAA modulo repetitions, and $C; C$ equals C modulo repetitions with delimiter ";". The result of generalization would be a new body $BA^+C(;C)^*$. The algorithm uses brute force search of all partitions with 4 or fewer non-empty substrings to find a compatible partition.

```
1:  function PARTITION(α, n)
2:      return { Δ | concat(Δ) = α ∧ |Δ| = n ∧
                     ∀ δ ∈ Δ. |δ| > 0 }
3:  function COMPATIBLE(Δ_α, Δ_β)
4:      return ∀ (α, β) ∈ zip(Δ_α, Δ_β). α = β ∨
          REP-EQ(α, β) ≠ ⊥ ∨ DELIM-EQ(α, β) ≠ ⊥
5:  function EXTRACT(α, β)
6:      if α = β
7:          return α
8:      else if REP-EQ(α, β) ≠ ⊥
9:          return regex REP-EQ(α, β)^+
10:     else if DELIM-EQ(α, β) ≠ ⊥
11:         let (A, b) = DELIM-EQ(α, β)
12:         return regex A(bA)^*
13:     else return α
14: function GEN(α, β)
15:     for 1 ≤ n < 5
16:         for (Δ_α, Δ_β) ∈ PARTITION(α, n) × PARTITION(β, n)
17:             if COMPATIBLE(Δ_α, Δ_β)
18:                 γ ← []
19:                 for (α', β') ∈ zip(Δ_α, Δ_β)
20:                     γ ← γ · EXTRACT(α', β')
21:                 return γ
22:     return ⊥
```

The algorithm uses two helper functions: (a) REP-EQ returns nonterminal A if its two arguments match regex pattern A^+ (the same nonterminal repeated 1 or more times), (b) DELIM-EQ returns the pair (A, b) if its two arguments match regex pattern $A(bA)^*$ (the same nonterminal A repeated 1 or more times, with repetitions separated by the delimiter b), and both functions return $\perp$ if no match is found. We also use two functional programming primitives: *zip*, which given two sequences returns a sequence of pairs of corresponding input elements, and $concat$, which concatenates the elements of a sequence.

With the specifics defined, we can now define our GENERALIZE operation, which takes a nonterminal and two production bodies to be generalized, and replaces the corresponding productions with a generalized variant if found:

$$[\![\text{GENERALIZE}(A, \alpha, \beta)]\!]\sigma$$
$$= \begin{cases} (G', \Pi, M, w, \mathcal{C}) & \text{if GEN}(\alpha, \beta) \neq \bot \\ \sigma & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \text{where } (G, \Pi, M, w, \mathcal{C}) & = & \sigma \\ (N, \Sigma, P, S) & = & G \\ P' & = & P - \{A \to \alpha\} \\ & & - \{A \to \beta\} \\ & & \cup \{A \to \text{GEN}(\alpha, \beta)\} \\ G' & = & (N, \Sigma, P', S) \end{array}$$

Now let us return to the running example from Section 2.4, in which we wished to generalize two productions specifying the syntax for function definitions. The compatible partition discovered by GEN is depicted visually in the following tables: each column of the upper table contains a corresponding pair of substrings, and the final row depicts the generalized production body returned by GEN:

`'fun'`	`ident ident`	`= expr`
`'fun'`	`ident ident ident`	`= expr`
`'fun'`	`ident+`	`= expr`

Resolve The RESOLVE operation enables Parsify to synthesize a set of disambiguating filters given an ambiguous sentence w and its correct parse tree t. The goal is to find a set of filters that reject all but the correct parse tree on the example. We use the following strategy for defining and searching the space of possible disambiguations:

1. Identify a set of possible filters Π_{tpl} based on the structure of the provided ambiguous example,
2. define a heuristic cost function h that assigns a score to each candidate drawn from $\mathcal{P}(\Pi_{tpl})$,
3. define the *successors* relation on candidates, and
4. perform A^*-search [18] on the directed graph induced by *successors* to find a low-cost set of filters that correctly disambiguates the example.

We wish to minimize the number of candidates considered in order to reduce the space of filters to search. The main intuition is that even though a parse tree may be large, we consider only those subtrees whose yield is ambiguous, which may be small. Let $T = \{t' \in \mathit{nodes}(t) \mid \mathit{yield}(t') \text{ is ambiguous w.r.t. } G(\mathit{rootSymbol}(t'))\}$. We define a candidate set of productions rules $R_{ambig} = \{\mathit{sig}(t) \mid t \in T\} \cup \{\mathit{sig}(t') \mid t \in T \wedge t' \in \mathit{children}(t)\}$, and from R_{ambig} construct our template set Π_{tpl} as follows:

$$\begin{aligned} \Pi_{tpl} = &\{f_{\pi_>}(r, r') \mid r, r' \in R_{ambig} \wedge r \neq r'\} \cup \\ &\{f_{\pi_L}(\{\, r \,\}), f_{\pi_R}(\{\, r \,\}), f_{\pi_N}(\{\, r \,\}) \mid r \in R_{ambig}\} \cup \\ &\{f_{\pi_L}(\{\, r, r' \,\}), f_{\pi_R}(\{\, r, r' \,\}), f_{\pi_N}(\{\, r, r' \,\}) \mid \\ &\quad r, r' \in R_{ambig} \wedge r \neq r'\} \end{aligned}$$

In other words, Π_{tpl} consists of all possible priority and associativity filters that mention two or fewer productions in R_{ambig}. Although this may seem like a large set, we rely on heuristic-guided search to avoid evaluating many poor candidates.

For our heuristic, we wish to assign higher cost to candidates that are less likely to correctly disambiguate the given example. Our heuristic is simple: prefer candidates that invalidate more parse trees, but reject a candidate if it rejects the correct tree. More precisely, we define a HEURISTIC function that takes a candidate set of disambiguating filters Π, a correct tree t, and returns the score for Π. There are three particular features to note: (a) on line 4, we return ∞ if the resulting disambiguation removes the intended parse tree from the set of parses, because the candidate cannot possibly be a solution, (b) on line 6, we return 0 if the candidate has rejected all but the intended parse tree, meaning this candidate is indeed a solution, and (c) on line 8, we otherwise cap our heuristic cost at 10 such that we need not enumerate parse trees beyond the first 10 returned by the parser (the parser computes parse forests lazily).

```
1: function HEURISTIC(Π, t)
2:     let ps = overline(p_GLL)|_Π(G(rootSymbol(t)), yield(t))
3:     if ¬∃(t', s') ∈ ps. t' = t
4:         return ∞
5:     if |ps| = 1
6:         return 0
7:     else
8:         return min(|ps|, 10)
```

To define the successors of a candidate Π, the naive approach would be to simply append each member of Π_{tpl} to Π in turn. In other words, a successor is just a candidate with one more filter than before. Unfortunately, with such a construction many of the successors would be *inconsistent* and thus useless. Thus, we define a more refined notion of successor that excludes any inconsistent candidate:

```
1: function SUCCESSORS(Π)
2:     for π ∈ Π_tpl
3:         let Π' = MERGE-FILTERS(Π ∪ {π})
4:         if ¬CONSISTENT(Π')
5:             next
6:         else
7:             yield Π'
```

We define helper function MERGE-FILTERS(Π) as follows:

1. If there exist in Π two priority filters $\pi_>(r, r')$ and $\pi_>(r', r'')$, then add the filter $\pi_>(r, r'')$ to Π if it does not already exist.
2. If there exist in Π two priority filters $\pi_>(r, r')$ and $\pi_>(r'', r''')$ and also an associativity filter $\pi_L(R)$, $\pi_R(R)$, or $\pi_N(R)$ such that $r', r'' \in R$, then add the filter $\pi_>(r, r''')$ to Π if it does not already exist.
3. If there exist in Π two left-associativity filters $\pi_L(R)$ and $\pi_L(R')$ such that $R \cap R' \neq \emptyset$, then replace them in Π with $\pi_L(R \cup R')$, essentially combining two left-associativity filters into one. Do analogously for right- and non-associativity filters.
4. Repeat the previous steps until no more additions can be made.

The intuition behind MERGE-FILTERS(Π) is that it is natural to view priorities and associativities as a partially ordered set of sets of operators. As such, steps 1 and 2 transitively close the priorities, and step 3 expands equivalence classes of associativities. Finally, the function CONSISTENT(Π) simply checks that a set of filters is consistent. In particular, it checks that (a) Π contains no cycle of priority filters such that $\pi_>(r, r')$ and $\pi_>(r', r)$, and (b) Π contains no conflicting associativity filters $\pi_X(R)$ and $\pi_Y(R')$ such that $X \neq Y \wedge R \cap R' \neq \emptyset$.

We are now ready to define our instantiation of A^*-search. A*-SEARCH(Π, t) employs a graph search algorithm that in each iteration picks the candidate Π' in its frontier with minimum value of $d(\Pi')$ + HEURISTIC(Π', t), where d is a measure of distance from the initial candidate. In our case the initial candidate is simply the existing set of disambiguation filters Π from our session state. The distance metric d is a weighted sum $l + 2r + 3n + 1.5p$ where l, r, n, p are the number of additional productions in left-associativity, right-associativity, non-associativity, and priority filters, respectively. Intuitively, this choice of coefficients encodes the fact that left-associativity is most preferred, followed by priority, right-associativity, and non-associativity filters.

Language	Paradigm	Source	LOC
Verilog	Imp	HLS tools	10,184
Tiger	Imp/Func	textbook	362
Apache [small]	Ad-hoc	online repo	1,546
SQL [small]	Query	census-postgres	1,492
Apache [big]	Ad-hoc	NASA	3.5M
SQL [big]	Query	census-postgres	228K

Figure 2. Benchmark suite

On every iteration, if the chosen candidate Π' has heuristic score 0 we know it is a possible disambiguation for our example and we add it to the set of solutions. Otherwise, we add the successors of Π' to our frontier and continue. Crucially, because the user may reject a given candidate, A*-SEARCH returns a lazily computed sequence of solutions by continuing to search for more candidates, even when a solution has already been found.

$$[\![\text{RESOLVE}(t)]\!](\sigma = (G, \Pi, M, w, \mathcal{C})) = (G, \Pi \cup \Pi', M, w, \mathcal{C})$$

where Π' is the first element of A*-SEARCH(Π, t) accepted by the user, otherwise $\emptyset$.

4. Evaluation

We evaluate Parsify along two dimensions: (a) **versatility**: can Parsify handle the complexities of a wide variety of languages from different language paradigms? and (b) **usability**: how easy is the tool to use, and what are best practices to make usage as effective as possible? To examine these questions, we ran several case studies in which one of the authors built several parsers from benchmarks drawn from different languages. To demonstrate versatility, our chosen benchmarks come from different programming paradigms and styles. Figure 2 shows the different benchmarks for which we constructed parsers. For each, we also list the language paradigm, the source of the benchmark, and the number of lines of code. We divide our benchmarks into two sets: (a) the first 4, which we call the *breadth* set, (b) and the last 2, much larger benchmarks, which we call the *depth* set. During each case study, we used Parsify with examples drawn from the breadth set to build a parser for the given language. Then, in the case for Apache and SQL, the constructed parsers were applied to the the depth set, without modification, to test for overfitting.

We now describe each of the languages in our benchmark set. **Verilog** is a popular hardware description language used to define digital circuits. Our goal was to parse the Verilog output of two high-level synthesis tools: Xilinx Vivado and C-to-Verilog, which compile C code to Verilog. The benchmarks come from an unpublished suite. **Tiger** is a textbook imperative language [4] with functional idioms (e.g., control statements as expressions). **Apache** logs come from the Apache web server. The small dataset was downloaded from an online repository [30], and the large dataset comes from a public NASA repository [5]. Log entries encode the requesting server, requested URL, server return code and size of the reply. Our goal was to perform a deep parse (e.g., parse URLs fully by matching CGI parameters separately, rather than parsing URLs as opaque strings). **SQL** is a ubiquitous database query language. We picked SQL because its syntax is drastically different from above languages. The queries were mined from the census-postgres open source project [6].

4.1 Versatility

We were able to build parsers to successfully parse 100% of each breadth benchmark. Additionally, with no modification to the parsers generated for Apache and SQL, we were able to achieve 97% and 86% coverage, respectively, on the large Apache and SQL benchmarks in our depth set. We measured coverage by splitting input files into individual top-level entities (a single log entry for Apache, and a single query for SQL). We then ran our inferred parsers against each entity. We report coverage as the number of lines successfully parsed in this way.

To diagnose the lower coverage achieved for the SQL benchmark, we examined a randomly sampled selection of code that failed to parse to determine the reason for failure. In all cases, we determined that the cause for failure was the presence of a syntactic form that did not exist in our smaller breadth sets. After allowing Parsify to learn on one more example, we were able to achieve 97% coverage on the large SQL benchmark.

Language features across the four benchmarks included: for and while loops; named records; function declarations and calls; conditionals (e.g., branches and switches); regular expressions; and various unary, binary, and ternary arithmetic operators.

4.2 Usability

To understand the level of interaction required to build a parser, we use *progression plots*. A progression plot shows the cumulative progress made as a function of number of UI actions taken. In particular, the x-axis of a progression plot shows the number of actions taken, where an action is any single UI interaction: applying a label, applying a negative label, resolving an ambiguity, undoing, or redoing. For each x-value, a progression plot displays the *cumulative progress*: the percentage of all the code in the project that is fully and successfully parsed after the first x actions. To compute progress, rather than count the number of characters parsed, we count the positions *between characters* that are part of some coloring. We do this for an important reason: if we were to count characters, then suppose two separate colored regions (labels) were directly adjacent. This would appear to achieve 100% when in fact a better parse would encompass both. Figure 3 show the progression plots for each of our breadth benchmarks: Verilog, Tiger, Apache, and SQL. Note that we are able to build each of these parsers with fewer than 400 UI interactions.

Completion Time A second important metric is the amount of *time* taken to reach a solution. We used Verilog as the first large case study, and not surprisingly it uncovered a variety of bugs in the implementation of Parsify. As a result, our experiment with Verilog was interspersed with several bug fixes and restarts, so we do not have an accurate measure of how much time Verilog took. On the other hand, the parsers for Tiger, Apache and SQL took between 6-8 hours. The authors of this paper had prior experience building manually written parsers for Verilog and Tiger: those prior efforts took nearly an order of magnitude longer than the corresponding Parsify effort – in fact much of the inspiration for Parsify is the wish to avoid previous difficulties.

To better understand where the time is spent when using Parsify, we analyzed recordings of each case study. A particularly interesting observation is that in some cases, an ambiguity was encountered that could not be resolved by synthesizing a disambiguating filter. The only course of action was to undo several actions. Because undo operations are counted as actions in our progression plots, these situations often correspond to some of the "plateaus" we see in the progression plots, during which no progress is made. After carefully analyzing the underlying reasons, we have formulated several "best practice" guidelines for building parsers even more quickly by avoiding these problems. Completion times were reduced significantly when following these practices, with times ranging from 30 minutes to 2 hours for each of the 4 languages.

4.3 Best Practices

Build bottom-up: It is important to employ a bottom-up approach when building parsers with Parsify. Consider two nonterminals, one

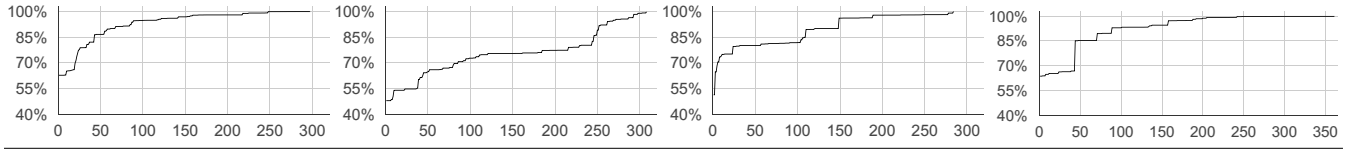

Figure 3. Progression plots for Verilog, Tiger, Apache Logs, and SQL (left to right)

of which always occurs higher in parse trees than the other (e.g., `stmt` and `expr`, where `stmt` always occurs higher than `expr`). In this case it is best to give as many examples as possible for `expr`, making it as complete as possible, before moving to `stmt`. This ordering is preferable because it allows detection and resolution of ambiguities earlier, on smaller examples, which makes it easier for *both* the human and Parsify. For instance, consider a simple ambiguity that occurs on a small expression when adding a new operator. This ambiguity is simple to visualize for the user, and easy to resolve for Parsify because the search space is small. In contrast this ambiguity becomes much more difficult to resolve if the ambiguity is discovered after statements have been parsed, because the ambiguity could occur deeply nested in a large statement. This not only makes the parse tree hard to visualize, but it also makes it harder for Parsify to resolve, because the search space of possible disambiguations can be much larger.

Consistency for Generalization We have determined two "styles" for using Parsify's generalization feature. Parsify works best when the user consistently uses one or the other, but *not both*, for the same syntactic entity. Consider the simple example of parsing literal arrays: suppose we have two arrays `[a]` and `[a,b]`, where the expressions `a` and `b` have already been correctly identified as `expr`s. There are two styles we can use, depending on whether we use an intermediate nonterminal or not.

Style 1: don't use an intermediate nonterminal for sequences. Apply label `array` to `[a]` and `[a,b]`, and Parsify generalizes the `array` rule to produce: "`[`", followed by a comma-separated list of `expr`s, followed by "`]`".

Style 2: use an intermediate nonterminal for sequences. We start by applying label `eseq` to both `a,b` and `a`, at which point Parsify generalizes the `eseq` rule to match a comma-separated sequence of `expr`s. The two expressions we are trying to parse have now been recolored and look as follows: `[a]` and `[a,b]`. At this point, we just need to label one of these two expressions as `array`.

Both styles work individually, but Parsify does not generalize well when the two styles are mixed for a given syntactic category (in the above example, arrays). In future work, we plan to investigate ways to augment our generalization technique to detect "mixed-style" usage and appropriately modify grammars to generalize these cases.

5. Related Work

Grammatical Inference The topic of *grammatical inference* [38], broadly focused on the challenge of inferring latent structure from text, has been studied extensively for decades. The classical problem, termed *identification in the limit*, is concerned with the ability to infer a correct language specification given a set of *positive* and *negative* examples (strings inside and outside the language, respectively). Steady progress has been made on inferring language specifications for regular languages [3], reversible languages [2], reversible context-free grammars with structural descriptions [33], and ultimately context-free languages [24].

We distinguish ourselves from this line of prior work in two ways: (a) prior work focuses more on the theoretical problem of correct grammar inference, whereas we focus on practical techniques that scale to software engineering problems; and (b) the ability in prior work to infer *some* grammar, does not mean the inferred grammar is meaningful to a human engineer: the parser developer also has an expectation that the corresponding syntax trees are easily comprehensible. In particular, a recent survey [21] found known efforts to apply grammatical inference to programming language parsers [9, 27] to be limited in nature, and it remains unclear whether these techniques scale to nontrivial software engineering problems. The second limitation points to the need for tools such as Parsify that allow more fine-grained control of the form of inferred productions.

Programming-by-example PBE techniques [7, 16] are concerned with constructing programs that implement an informal specification provided in the form of input and output examples. For example, researchers have proposed PBE techniques for domains such as text editing [15], spreadsheet table transformations [17], number transformations [35], and extraction from semi-structured sources [23]. A particularly relevant instantiation of PBE is the PADS system [10] for automatically inferring the structure of ad-hoc data formats such as log files. Due to its focus on such formats, PADS makes several simplifying assumptions, for example that log entries are chunked line-by-line or file-by-file, and that format specifications are not recursive. As a result it is unclear whether their techniques can generalize to context-free languages.

Some PBE systems have a visual component similar to our own: for example, LAUNCHPADS [8], LAPIS [29], SMARTedit [22], and STEPS [39]. LAUNCHPADS is a visual frontend for the PADS system described above. SMARTedit and LAPIS support repetitive edits by example, but do not export the hierarchical structure of the underlying text. STEPS provides a facility for highlighting and labeling regions similarly to Parsify, but for the purpose of facilitating more limited text transformations akin to that accomplished by "a short Perl/AWK script."

Parser Generators We focus here on recent advances towards more *user-friendly* parsing. Recent work [25] has employed natural-language processing to generate parsers from English descriptions of input formats. Generalized parsers of the GLL [34] and GLR [37] families have recently gained popularity due to their ability to accept any context-free grammar while still offering good performance. Unfortunately, the ability to specify ambiguous grammars is a significant disadvantage in comparison to parser generators based on more restricted grammars. More so, such parsers offer no mechanism for disambiguation besides refactoring productions. Disambiguating filters [20, 36] offer a declarative mechanism for disambiguation without modifications to the grammar. Parsify uses the instaparse GLL parser, which we augmented with disambiguating filters.

6. Future Work and Conclusion

We have presented a novel application of programming-by-example to parsers and implemented a tool, Parsify, that embodies this approach. Our evaluation shows that the approach is feasible and works well on a variety of languages and paradigms.

Still, there are many lines of future work left to be pursued. As mentioned in Section 4.3, one direction would be to improve our generalization technique to handle scenarios where the two styles of generalization are mixed. Another direction would be to improve the disambiguation algorithm to handle more cases. One particular kind of disambiguation that Parsify does not support well is what we term

context-sensitive disambiguation (not to be confused with context-sensitive grammars), where the disambiguation is done based on the context in which the ambiguity occurs. Consider the following situation, adapted from our Apache log benchmark. We wish to parse a sequence of id-value pairs of the form `id=expr`, where `expr` matches both floating point values and regular expressions. The string `x=1.37` is ambiguous because `1.37` can be parsed as either a floating point value or a regular expression (with "." meaning any character). Proper disambiguation requires knowing the context in which the `expr` occurs: `expr` can be a regular expression if and only if `id` is the string "`search`". Parsify cannot automatically provide this kind of context-sensitive disambiguation. Although an unambiguous grammar that handles this construct can be built in Parsify today, this does require that the user avoid inferring the offending productions from the start. It would be interesting to investigate how one can enhance Parsify to handle these context-sensitive disambiguations in the future.

A key limitation of our current prototype deals with token structure: we rely on predefined rules for parsing initial tokens and collapsing whitespace. In future iterations we aim to add token inference to allow constructs such as syntactically significant whitespace (e.g., as in Python).

Finally, two interesting avenues for future work involve systematic evaluation of our tool across a larger audience. First, we seek to develop a methodology for more systematic evaluation of Parsify's heuristics. For example, the set of coefficients used in our distance metric have worked well for the benchmarks we evaluated in Section 4, but it is possible that other coefficients may be more appropriate for languages with different characteristics that we have not tested. Second, we aim to perform a broader user study to see how other programmers make use of our tool. The website for Parsify is available at `http://goto.ucsd.edu/parsify`.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Science Foundation through grants 1228967, 1219172 and 1423517, and a generous gift from Google.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, 1972.

[2] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3), 1982.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987.

[4] A. W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[5] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *SIGMETRICS*, 1996.

[6] census-postgres, 2014. URL `https://github.com/leehach/census-postgres`.

[7] A. Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, 1993.

[8] M. Daly, M. F. Fernández, K. Fisher, Y. Mandelbaum, and D. Walker. LAUNCHPADS: A system for processing ad hoc data. In *PLAN-X*, 2006.

[9] A. Dubey, S. Aggarwal, and P. Jalote. A technique for extracting keyword based rules from a set of programs. In *CSMR*, 2005.

[10] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL*, 2008.

[11] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *POPL*, 2004.

[12] *GNU Bison manual*. GNU Software Foundation. URL `http://www.gnu.org/software/bison/manual/`.

[13] R. Grimm. Better extensibility through modular syntax. In *PLDI*, 2006.

[14] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[16] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

[17] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[18] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2), 1968.

[19] instaparse, 2014. URL `https://github.com/Engelberg/instaparse`.

[20] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *ASMICS*, 1994.

[21] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM TOSEM*, 14(3), 2005.

[22] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2), 2003.

[23] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *PLDI*, 2014.

[24] L. Lee. Learning of context-free languages: A survey of the literature. Technical Report TR-12-96, Harvard University, 1996.

[25] T. Lei, F. Long, R. Barzilay, and M. C. Rinard. From natural language specifications to program input parsers. In *ACL*, 2013.

[26] S. McPeak and G. Necula. Elkhound: A fast, practical GLR parser generator. In *CC*, 2004.

[27] M. Mernik, G. Gerlič, V. Žumer, and B. R. Bryant. Can a parser be generated from examples? In *SAC*, 2003.

[28] M. Might and D. Darais. Yacc is dead. *CoRR*, abs/1010.5023, 2010.

[29] R. C. Miller and B. A. Myers. Lightweight structured text processing. In *USENIX ATC*, 1999.

[30] MonitorWare. Apache (Unix) log samples, 2004. URL `http://www.monitorware.com/en/logsamples/apache.php`.

[31] R. C. Moore. Removing left recursion from context-free grammars. In *NAACL*, 2000.

[32] T. Parr and K. Fisher. LL(*): The foundation of the antlr parser generator. In *PLDI*, 2011.

[33] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1), 1992.

[34] E. Scott and A. Johnstone. GLL parsing. *ENTCS*, 253(7), 2010.

[35] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.

[36] M. Thorup. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica*, 33(5), 1996.

[37] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[38] E. Vidal. Grammatical inference: An introductory survey. In *Grammatical Inference and Applications*, LNCS. 1994.

[39] 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 *UIST*, 2013.