UNIVERSITY OF CALIFORNIA, SAN DIEGO

Constructing Parsers by Example via Interactive Program Synthesis

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Alan Leung

2017

The Dissertation of Alan Leung is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

DEDICATION

To my loving wife and family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my Ph.D. advisor, Sorin Lerner, whose advice over the years has helped me mature as both a researcher and a person. I also give special thanks to Ranjit Jhala for being an enthusiastic mentor during the early formative years of my Ph.D. To the other members of my committee, Samuel Buss, Ryan Kastner, and Todd Millstein, I am grateful for the patience and generosity with which they gave me their time.

Next, I must thank the members of the UCSD Programming Systems Group for their camaraderie over the years. Thanks to Dimitar Bounov, John Sarracino, Manish Gupta, and Ross Tate for being first-class collaborators and friends for whom I will always have the utmost respect. Thanks also to Alexander Bakst and Panagiotis Vekris, my officemates for several years, for their good nature and insightful conversation in good times, and heartfelt commiseration in hard times. Thanks to every Progsys member past and present with whom I had the good fortune to share a meal, a coffee, or a conversation.

Thanks to the members of the Rita cooking collective: Bridgette Wilson, Dan Moeller, Greg Long, Jason Greco, Karyn Benson, Marlena Fecho, Ming Wang, and Ryan Kanoknukulchai. Making friends in a new place can be hard, but you made it feel effortless. To the members of 24th Street Labs, Kai Wang and Karyn Benson: thanks for all the surfing, barbecues, and impromptu parties. It was a slice of the California beach lifestyle that I will always remember fondly. Thanks to Sam Kwak for the thousands of laps at Canyonview Pool. Thanks to the other members of my cohort that I had the good fortune to befriend – to name a few, Dan Ricketts, Matt Der, Sheeraz Ahmad, and Wilson Lian. You're all class acts.

Finally, I must thank Wai-San, my wife and best friend, for giving me endless support and strength, for having unyielding confidence in my ability, and for reminding me that there's more to life than work.

Chapter 1, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

Chapter 1, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 2, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

Chapter 2, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 3, in full, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

Chapter 4, in full, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 5, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation,

2015. The dissertation author was the primary investigator and author on this paper.

Chapter 5, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

VITA

2004	Bachelor of Science, Cornell University
2004–2009	Component Design Engineer, Intel
2012	Research Intern, Microsoft Research, Cambridge
2010–2017	Research Assistant, University of California, San Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015.

Leung, Alan; Bounov, Dimitar; Lerner, Sorin. “C-to-Verilog Translation Validation,” 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign, 2015.

Leung, Alan; Gupta, Manish; Agarwal, Yuvraj; Gupta, Rajesh; Jhala, Ranjit; Lerner, Sorin. “Verifying GPU Kernels by Test Amplification,” Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2012.

Tate, Ross; Leung, Alan; Lerner, Sorin. “Taming Wildcards in Java’s Type System,” Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011.

ABSTRACT OF THE DISSERTATION

Constructing Parsers by Example via Interactive Program Synthesis

by

Alan Leung

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Sorin Lerner, Chair

Parsers – programs that extract structure from strings – are fundamental components of many software systems. Although parsing theory may have found its roots in early work on programming languages, the growth of computing during the intervening decades has expanded the role of parsers into many systems one might not immediately expect: email clients, video games, spreadsheet programs, and relational databases are only a few among myriad examples of systems that extract structured information from input text. As a result, the construction of parsers has become a ubiquitous programming task that is performed by developers across a large spectrum of domains. It is not just a task for programming language experts anymore.

Unfortunately, despite over forty years of research on parsing, writing parsers remains a painstaking, manual process that is prone to subtle bugs and pitfalls. Existing tools for generating parsers assume a great deal of background knowledge in parsing and formal language theory, so the learning curve is high.

In this dissertation, we argue that it is possible to make parsing more accessible by combining interactive visual feedback with the programming-by-example paradigm, wherein users synthesize programs simply by providing example inputs and outputs demonstrating the result of the intended computation. Towards this aim, we present novel algorithms for (1) constructing syntactic specifications by example, (2) constructing lexical analyses by example, and (3) visualizing progress toward parser completion. We instantiate these algorithms in two graphical development environments we have implemented, Parsify and its successor Parsimony, whose central user interaction paradigm is that of programming-by-example. Finally, via user study we demonstrate that non-expert users indeed show significantly better performance when using our system.

Chapter 1

Introduction

A parser, at its most fundamental level, is a program that extracts semantic information from strings. Given the breadth of this definition, it should be no surprise that parsers are ubiquitous in software systems. Most obviously, parsers serve a vital role in the implementation of programming languages – the execution of a compiler or interpreter, no matter how sophisticated, often starts with the conversion of strings, mere sequences of bytes, into formats more amenable to further analysis and transformation. Indeed, the development of parsing theory has been a keystone success in programming language research, with a history that spans nearly half a century.

However, parsing is not simply the domain of programming language specialists. With the explosive growth of computing, the role of parsing has also grown as a core component in many of the software systems on which we rely. Consider only a small selection of activities that at their core, employ parsers: importing a CSV file into spreadsheet software, converting a blog post in Markdown into HTML, extracting a query string from a request URL, mining Apache server logs for anomalous behavior, inspecting the fields of a TCP packet, reading a configuration file on application startup, extracting the arguments from a command-line invocation, or interpreting a JSON-formatted string from a web-based API. One would be hard-pressed to find any sophisticated software toolchain that does not need to extract information from strings.

Thus, parsing is a ubiquitous programming task that developers across a large spectrum of domains need to understand to accomplish their goals. Parsing is no longer a specialized discipline to be left to those with specialized skills (e.g., programming language and compiler developers). It follows that we should seek to make parsing accessible to a wider audience.

Unfortunately, the current state-of-the-art in parsing leaves something to be desired when it comes to its accessibility. Despite decades of research on parsing, the construction of parsers remains a painstaking, manual process that requires specialized knowledge to avoid its subtle pitfalls. Consider Bison, one of the most popular parser generators in common use today. The following is extracted directly from its user manual:

> *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

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. Although we mention Bison first, Bison is not alone when in comes to its high learning curve.

More modern incantations of parser technologies such as ANTLR [59] and Packrat parsing [21, 25] seemingly 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 [57]. 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 [64] and GLR [70] 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,” [53] 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

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 [42], spreadsheet table transformations [29], and data extraction from ad-hoc logs [20]. 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).

1.1 Outline of this work

This dissertation argues that it is possible to make parsing more accessible by using a combination of program synthesis and interactive visual feedback. To support this argument, we build two graphical development environments, Parsify and Parsimony, whose central user interaction paradigm is that of programming-by-example.

We first discuss Parsify, an interactive, graphical development environment for incrementally synthesizing and testing parsers. In Parsify, the user does not write a single line of code. Instead, the user provides input/output examples demonstrating the result of a correct derivation with respect to a context-free grammar to be inferred. Parsify's underlying synthesis engine then infers a refined grammar consistent with each given example. The key component of this engine is an iterative algorithm for synthesizing and refining the grammar one production and one example at a time. In response to any such inference, Parsify's interface updates with immediate visual feedback displaying the result of the change induced by that inference. For ease of use, Parsify provides a graphical mechanism for specifying example parse trees using only textual selections – the user need not manually input examples, which is both tedious and error-prone. We show the effectiveness of Parsify in practice by conducting a series of case studies in which a co-author successfully implemented the parsers for several input languages from different domains, each in less than a day of work.

We next describe Parsimony, the spiritual successor to Parsify that makes several key improvements over the previous system. In particular, Parsimony reframes parser synthesis as satisfaction of a constraint system derived from user-provided examples. In this more general setting, it is possible to solve for many examples simultaneously, opening the door to the inference of much more complex solutions consisting of systems of mutually-dependent productions, rather than just one production at a time. Using this improved machinery, we design a parametric, extensible framework capable of inferring entire subgrammars, such as that for algebraic expressions, with only the single up-front cost of instantiating the framework with a concrete heuristic. Another major improvement over Parsify is the ability to infer not only context-free grammars, but also regular expressions for lexer definitions. We describe an algorithm for inferring such regular expressions from a corpus of existing definitions and show that it has several nice theoretical properties with regard to the size and quality of the inferred solution. Finally, we conduct a controlled user study in which 18 participants with no previous experience using either Parsify or Parsimony were asked to accomplish a series of parser implementation tasks. The results of our study show that Parsimony is effective at increasing the participants' speed at making progress, while also decreasing the number of mistakes they make.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a brief overview of language and parsing theory. Chapter 3 describes Parsify, our first programming-by-example framework for synthesizing parsers. Chapter 4 describes Parsimony, the spiritual successor to Parsify, and its various improvements. Chapter 5 surveys related work. Finally, Chapter 6 summarizes this dissertation and presents areas for future progress.

1.3 Acknowledgements

This chapter, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

This chapter, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 2

Preliminaries

We begin with a preliminary overview of lexical analysis, context-free grammars, parsing, and disambiguating filters. The definitions in this chapter are referenced in Chapters 3 and 4.

2.1 Lexical Analysis

2.1.1 Regular Expressions

A language is a set of strings and a string is a sequence of symbols. A regular expression (regex) is an algebraic notation for defining a language, the syntax and semantics of which is defined in Figure 2.1. We denote the language of a regex r by $\mathscr{L}(r)$, and we denote the set of all regexes by $\mathfrak{R}$. We say regex r *matches* string s iff $s \in \mathscr{L}(r)$. For any regex r, there exists an equivalent deterministic finite automaton (DFA) D such that the language of the automaton $\mathscr{L}(D) = \mathscr{L}(r)$. Language containment $\mathscr{L}(D_1) \subset \mathscr{L}(D_2)$ on DFAs is decidable. Thus $\forall r_1, r_2 \in \mathfrak{R}.\ \mathscr{L}(r_1) \subset \mathscr{L}(r_2)$ is decidable.

2.1.2 Lexers

A lexer is a 3-tuple $L = (\Sigma, \Gamma, Q)$ where Σ is a set of symbols called terminals, Γ is a set of symbols distinct from Σ called the alphabet of L, and $Q \in [(\Sigma \times \mathfrak{R})]$ is a sequence of lexer rules. Suppose $(\tau, s') = \textsc{lex}(L, s)$, where s is a string of symbols drawn from Γ,

$$\begin{array}{rcll} r & ::= & \texttt{a} & \text{symbol} \\ & | & rr & \text{concatenation} \\ & | & r \mid r & \text{union} \\ & | & r? & \text{optional} \\ & | & r^* & \text{Kleene closure} \\ & | & r^+ & \text{Kleene plus} \end{array}$$

$$\begin{array}{rcl} \mathscr{L}(\texttt{a}) & = & \{\texttt{a}\} \\ \mathscr{L}(r_1 r_2) & = & \{ss' \mid s \in \mathscr{L}(r_1) \wedge s' \in \mathscr{L}(r_2)\} \\ \mathscr{L}(r_1 \mid r_2) & = & \{s \mid s \in \mathscr{L}(r_1) \vee s \in \mathscr{L}(r_2)\} \\ \mathscr{L}(r?) & = & \mathscr{L}(r) \cup \{\varepsilon\} \\ \mathscr{L}(r^*) & = & \cup_{i=0}^{\infty} \mathscr{L}(r^i) \\ \mathscr{L}(r^+) & = & \cup_{i=1}^{\infty} \mathscr{L}(r^i) \end{array}$$

Figure 2.1. Syntax and semantics of regular expressions.

and LEX is as defined in the reference implementation depicted in Figure 2.2. Informally, τ is a string of terminals computed by greedily matching subsequences of s with lexer rules defined by L. s' is the suffix of s that could not be matched in this way. We say L fails to lex s if $s' \neq \varepsilon$. If $s' = \varepsilon$ we say that L lexes s and that τ is the *token stream* of s with respect to L. When clear from context, we omit reference to L and simply say τ is the token stream of s.

2.2 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 dissertation: 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 letters w and τ are (possibly empty) strings of terminals, and productions

```
function LEX($L, s$)
    let $(\cdot, \cdot, Q) = L$
    $(\tau, s') \leftarrow ([], s)$
    while $|s'| > 0$ do
        let $(t, s'') =$ NEXT-TOKEN$(Q, s')$
        if $t \neq \bot$ then
            $(\tau, s') \leftarrow (\tau +\!\!+ [t], s'_{[|s''|\ldots]})$
        else
            return $(\tau, s')$
        end if
    end while
    return $(\tau, \varepsilon)$
end function

function NEXT-TOKEN($Q, s$)
    $(t, s') \leftarrow (\bot, \varepsilon)$
    for $(t', r)$ in $Q$ do
        let $s'' =$ MAX-MATCH$(r, s)$
        if $|s''| > |s'|$ then
            $(t, s') \leftarrow (t', s'')$
        end if
    end for
    return $(t, s')$
end function
```

Figure 2.2. Reference lexer algorithm. MAX-MATCH(r, s) is the longest prefix of s matched by r.

$(A, \beta) \in P$ are written equivalently as $A \rightarrow \beta$. We write $G(A)$ to mean the grammar G with start nonterminal replaced by A.

2.2.1 String Indexing

String indices begin at 0. We write $\alpha_{[i]}$ to mean the symbol at index i of string α. The notation $\alpha_{[i\ldots]}$ denotes the suffix of α starting at index i, and the notation $\alpha_{[i\ldots j]}$ denotes the substring of α starting at index i with length $j - i$. We write $|\alpha|$ to mean the length of α and $\alpha\beta$ or $\alpha \cdot \beta$ for concatenation of α and β.

2.2.2 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 $(\alpha, \beta) \in \Rightarrow^*$, and a *derivation* of β from α is a sequence $\alpha \Rightarrow ... \Rightarrow \beta$ that witnesses $\alpha \Rightarrow^* \beta$.

2.2.3 Sentential Forms

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 $\mathscr{L}(G)$, 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.

2.2.4 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 $A \rightarrow v_1 ... v_n \in P$. 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., $sig([A\ [Ba] b]) = A \rightarrow Bb$). 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 $\mathit{tree}(d)$ be the map from derivations to their parse trees. We can now define the set of parse trees of a sentence w as follows:

$$\mathit{trees}(w) = \{\mathit{tree}(d) \mid d \in D(w)\}$$

which we extend to grammars naturally:

$$\mathit{trees}(G) = \bigcup_{w \in \mathscr{L}(G)} \mathit{trees}(w)$$

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 $\mathit{index}(t)$ to be the map from parse trees to their index trees. The *span* of index tree $\hat{t}$, written $\mathit{span}(\hat{t})$, 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

$$\exists\ d_1, d_2 \in D_G(w).\, \mathit{tree}(d_1) \neq \mathit{tree}(d_2)$$

For brevity, we say w is *ambiguous* when G is clear from context. A grammar G is ambiguous iff $\mathscr{L}(G)$ contains an ambiguous sentence.

2.2.5 Parsers

A *parser* p is a function of type $G \times \Sigma^* \to \mathscr{P}(\mathit{trees}(G) \times \Sigma^*)$ 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 $w = \mathit{yield}(t) \cdot w'$. 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 $w \notin \mathscr{L}(G)$ then $\overline{p}(G,w) = \emptyset$.)

Generalized Parsers

A parser p is generalized iff

$$\forall w \in \mathscr{L}(G), w' \in \Sigma^*.\, p(G, w \cdot w') \supseteq \mathit{trees}(w) \times \left\{w'\right\}$$

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 [32] or Elkhound [50], respectively. In the remainder of this dissertation, let p_{GLL} be such a generalized parser.

2.3 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:

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

2.3.1 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:

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

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. The following are the analogous definitions for right-associativity and non-associativity filters, respectively.

$$f_{\pi_R}(R) = \lambda t.\exists\, t' \in nodes(t).$$
$$\begin{cases} \bigvee_{i=0}^{n-1} \big(sig(t') \in R \wedge sig(t_i) \in R\big) & \text{if } t' = [A\ t_0 ... t_n] \\ false & \text{otherwise} \end{cases}$$

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

2.3.2 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.

2.3.3 Consistency

Because filters are simply predicates on trees, it is possible that a composition of filters gives rise to a trivially satisfiable predicate $\big(\pi(t) \vee \pi'(t)\big) \leftrightarrow \mathit{true}$. 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*.

2.3.4 Filter Specification Syntax

Having established the semantics of disambiguating filters, we now describe the corresponding syntax.

Associativity Filter Syntax

The following two syntactic forms are two different ways to specify a left-associativity filter of one production: $f_{\pi_L}(\{A \rightarrow \beta\})$. To specify right-associativity or non-associativity, we replace `left` with `right` or `nonassoc`, respectively.

$$\texttt{left \{ } A \rightarrow \beta \texttt{ \}}$$

$$A \rightarrow \beta \texttt{ \{ left \}}$$

To specify a left-associativity filter over multiple productions, we use the following syntax, which corresponds to the filter $f_{\pi_L}(\{A_1 \rightarrow \beta_1,\, A_2 \rightarrow \beta_2,\, \ldots,\, A_n \rightarrow \beta_n\})$.

$$\texttt{left \{} A_1 \rightarrow \beta_1 \quad A_2 \rightarrow \beta_2 \quad \ldots \quad A_n \rightarrow \beta_n \texttt{ \}}$$

The forms for `right` and `nonassoc` are immediately analogous.

Priority Filter Syntax

The following syntactic form specifies the priority filter $f_{\pi_>}(A_1 \to \beta_1\ , A_2 \to \beta_2)$.

$$A_1 \to \beta_1 > A_2 \to \beta_2$$

For syntactic clarity, a sequence of such forms may be enclosed in a `priorities` block. Enclosure in a `priorities` block has no effect on the semantics of the enclosed filters.

```
priorities {
  $A_1 \to \beta_1 > A_2 \to \beta_2$
  $A_3 \to \beta_3 > A_4 \to \beta_4$
  ...
  $A_{n-1} \to \beta_{n-1} > A_n \to \beta_n$
}
```

2.4 Acknowledgements

This chapter, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

This chapter, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for

Example-Guided Synthesis of Lexers and Parsers." The dissertation author was the primary investigator and author on this paper.

Chapter 3

Parsify

This chapter presents Parsify, our first instantiation of programming-by-example in the context of parser construction. To achieve a high-level of interactivity and user-accessibility, we architect Parsify to adhere to the following three design principles:

- We should minimize the amount of manual coding asked of the user.
- The user should have intuitive, real-time feedback in response to incremental changes to the parser.
- The user should be able to explore the design space of possible parser implementations easily and quickly.

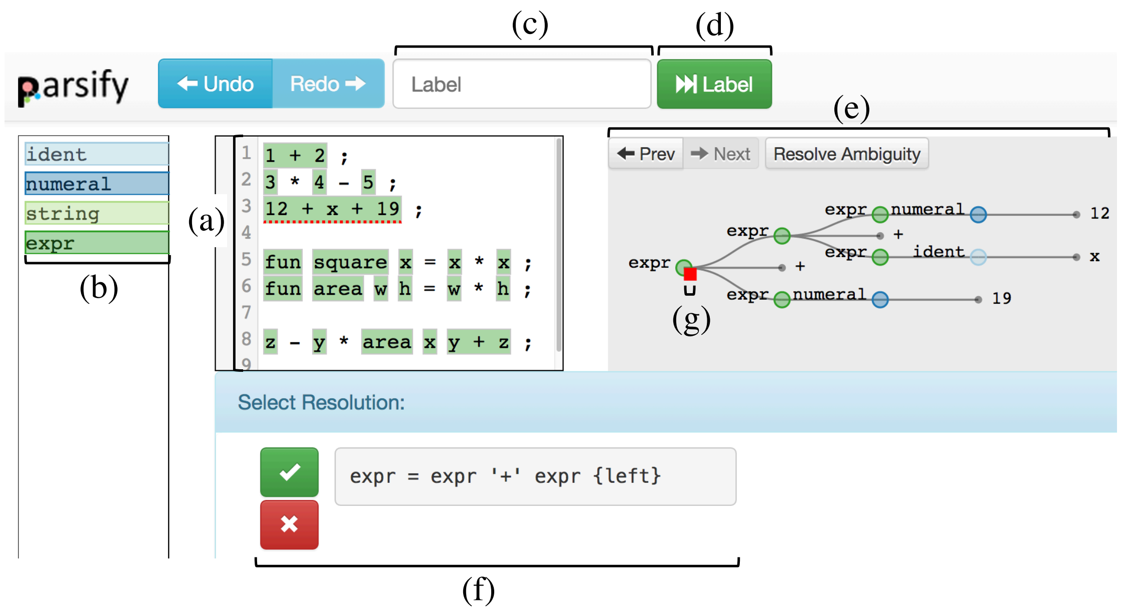
To achieve these principles, Parsify presents an exploratory interface in which the user poses examples that induce the shape of productions in the underlying grammar being inferred. These examples are presented in the form of text selections (i.e., using a mouse) in an editor window presenting the file 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.

This chapter covers both the user interaction model and the underlying algorithms developed to implement the Parsify development environment. To show the usefulness of Parsify as a development tool, we additionally discuss the results of several case studies in which we used Parsify to implement parsers for a variety of languages. In summary, this chapter details the following contributions:

1. We present a novel application of programming-by-example to the domain of parsers for context-free languages.
2. We present techniques and algorithms for efficiently visualizing progress, inferring productions, and synthesizing disambiguating filters by example, including novel uses of generalized parsers and A^*-search.
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.

3.1 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 3.3. 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.



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

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

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

```
expr = expr '+' expr {left}
```

Figure 3.1. 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

3.1.1 User Interface Overview

Figure 3.1 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.

3.1.2 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 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 3.1. 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 to add to the grammar: `expr` $\rightarrow$ `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 production 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 3.2 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` $\rightarrow$ `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 ;
```

3.1.3 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 `expr` $\rightarrow$ `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` production 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 3.1, 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 3.1 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 disambiguating filters to remove the unwanted parses. More specifically, in this case Parsify automatically synthesizes the *left-associativity filter* `expr` $\rightarrow$ `expr + expr {left}`, which disallows derivation of trees with form `expr + [expr + expr]`. The filter is displayed in the Resolution Pane at the bottom of the UI, as shown in Figure 3.1.

Priority

Now the user proceeds similarly for the `*` and `-` operators, teaching Parsify the productions `expr` $\rightarrow$ `expr * expr` and `expr` $\rightarrow$ `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` $\rightarrow$ `expr * expr` $>$ `expr` $\rightarrow$ `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` $\rightarrow$ `expr * expr ; expr` $\rightarrow$ `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 3.1. 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-associativity filter.

3.1.4 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 exprs. 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 3.1. 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 exprs 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 productions of the form

```
fundef → ‘fun’ ident ident = expr ;
fundef → ‘fun’ ident ident ident = expr ;
```

Although these productions handle 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.

3.1.5 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.

3.1.6 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. Our solution formulates synthesis as an instantiation of A^*-search to avoid unlikely candidates.

Section 3.2 details our solutions to these challenges.

3.2 Algorithm

In this section we describe the user interactions and core algorithms employed by Parsify for inferring context-free grammars. We formalize the model of user interaction by defining a core set of operations as transitions between *session states* that represent a snapshot of the system's state at any point in time. Then, we define various user-visible actions as compositions of these operations.

3.2.1 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

$(G, \Pi, M, w, \mathscr{C})$ 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 $\mathscr{C}$ 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* $\mathscr{C} \subseteq \mathbb{L}$ 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 $(\texttt{expr}, 3, 10) \in \mathscr{C}$, then Parsify colors the seven character substring, starting at index 3, with the color corresponding to `expr`.

3.2.2 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 $\rightarrow$ GENERALIZE $\rightarrow$ DRAW
2. *Reject Label:* NEGATE $\rightarrow$ DRAW

3. *Disambiguate:* RESOLVE $\rightarrow$ 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.

3.2.3 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 $\mathit{rootSymbol}(t') \sqsubseteq_G^* \mathit{rootSymbol}(t) \wedge \mathit{rootSymbol}(t) \not\sqsubseteq_G^* \mathit{rootSymbol}(t')$. 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 $A = B \vee (B \rightarrow \alpha A \beta \in P)$. We can then sort parses according to *better* and choose the root symbol of the highest rated parse tree to be part

```
function COLOR($\sigma$)
    let $(G,\Pi,M,w_0,\_) = \sigma$
    let $(N,\_,\_,\_) = G$
    $\mathscr{C} \leftarrow \emptyset; w \leftarrow w_0; n \leftarrow 0$
    while $|w| > 0$ do
        let $X = \{(t,w') \mid$
                $\exists\, A \in N.\ (t,w') \in p_{GLL}|_\Pi(G(A),w) \wedge$
                $\forall \hat{t} \in nodes(index(t)).\textsc{label}(\hat{t},n) \notin M\}$
        if $X \neq \emptyset$ then
            let $(t,w') = first(sortBy(better,X))$
            $\mathscr{C} \leftarrow \mathscr{C} \cup \{\,\textsc{label}(index(t),n)\,\}$
            $w \leftarrow w'; n \leftarrow n + |yield(t)|$
        else
            $w \leftarrow w_{[1\ldots]}; n \leftarrow n+1$
        end if
    end while
    return $\mathscr{C}$
end function

function LABEL($\hat{t},n$)
    let $(i,j) = span(\hat{t})$
    return $(rootSymbol(\hat{t}), i+n, j+1+n)$
end function
```

Figure 3.2. The COLOR algorithm.

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. The algorithm is shown in Figure 3.2.

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

$$[\![\textsc{draw}]\!](\sigma = (G,\Pi,M,w,\mathscr{C})) = (G,\Pi,M,w,\textsc{color}(\sigma))$$

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 3.1.4 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.

3.2.4 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.

$$\begin{gathered}[\![\textsc{negate}(A,i,j)]\!](\sigma = (G,\Pi,M,w,\mathscr{C})) = \\ (G,\Pi,M\cup\left\{(A,i,j)\right\},w,\mathscr{C})\end{gathered}$$

Returning to our running example, consider the situation from Section 3.1.4 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 $\textsc{negate}(\texttt{expr},0,3)$, followed immediately by a DRAW operation to regenerate a new coloring respecting the new constraint.

```
1:  function GEN-PROD($\sigma, A, i, j$)
2:      let $(\_,\_,\_,w,\mathscr{C}) = \sigma$
3:      $idx \leftarrow i; \beta \leftarrow []$
4:      while $idx < j$ do
5:          let $X = \{(A', i', j') \in \mathscr{C} \mid idx = i' \land j' < j\}$
6:          if $|X| = 1$ then
7:              let $\{(A', \_, j')\} = X$
8:              $\beta \leftarrow \beta \cdot A'$
9:              $idx \leftarrow j'$
10:         else if $|X| = 0$ then
11:             $\beta \leftarrow \beta \cdot w_{[idx]}$
12:             $idx \leftarrow idx + 1$
13:         else // unreachable
14:         end if
15:     end while
16:     return $A \rightarrow \beta$
17: end function
```

Figure 3.3. The GEN-PROD algorithm.

3.2.5 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. The algorithm for generating this production, called GEN-PROD, is shown in Figure 3.3.

Informally, GEN-PROD scans the selected range from left to right looking for labels in $\mathscr{C}$ 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 satisfied 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.

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

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

3.2.6 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 (EBNF): 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 [26].

Given two production bodies, the function GEN, shown in Figure 3.4, 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.

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:

```
function PARTITION($\alpha, n$)
    return $\{\Delta \mid concat(\Delta) = \alpha \wedge |\Delta| = n \wedge$
                $\forall \delta \in \Delta . |\delta| > 0\}$
end function

function COMPATIBLE($\Delta_\alpha, \Delta_\beta$)
    return $\forall (\alpha, \beta) \in zip(\Delta_\alpha, \Delta_\beta) . \alpha = \beta \vee$
        REP-EQ($\alpha, \beta$) $\neq \perp \vee$ DELIM-EQ($\alpha, \beta$) $\neq \perp$
end function

function EXTRACT($\alpha, \beta$)
    if $\alpha = \beta$ then
        return $\alpha$
    else if REP-EQ($\alpha, \beta$) $\neq \perp$ then
        return regex REP-EQ($\alpha, \beta$)$^+$
    else if DELIM-EQ($\alpha, \beta$) $\neq \perp$ then
        let $(A, b)$ = DELIM-EQ($\alpha, \beta$)
        return regex $A(bA)^*$
    else return $\alpha$
    end if
end function

function GEN($\alpha, \beta$)
    for $1 \leq n < 5$ do
        for $(\Delta_\alpha, \Delta_\beta) \in$ PARTITION($\alpha, n$) $\times$ PARTITION($\beta, n$) do
            if COMPATIBLE($\Delta_\alpha, \Delta_\beta$) then
                $\gamma \leftarrow []$
                for $(\alpha', \beta') \in zip(\Delta_\alpha, \Delta_\beta)$ do
                    $\gamma \leftarrow \gamma \cdot$ EXTRACT($\alpha', \beta'$)
                end for
                return $\gamma$
            end if
        end for
    end for
    return $\perp$
end function
```

Figure 3.4. The GEN algorithm.

$$[\![\textsc{generalize}(A,\alpha,\beta)]\!]\sigma$$

$$= \begin{cases} (G',\Pi,M,w,\mathscr{C}) & \text{if } \textsc{gen}(\alpha,\beta) \neq \bot \\ \sigma & \text{otherwise} \end{cases}$$

$$\begin{aligned} \text{where } (G,\Pi,M,w,\mathscr{C}) &= \sigma \\ (N,\Sigma,P,S) &= G \\ P' &= P - \{A \to \alpha\} \\ & \quad - \{A \to \beta\} \\ & \quad \cup \{A \to \textsc{gen}(\alpha,\beta)\} \\ G' &= (N,\Sigma,P',S) \end{aligned}$$

Now let us return to the running example from Section 3.1.4, 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`

3.2.7 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 $\mathscr{P}(\Pi_{tpl})$,

3. define the *successors* relation on candidates, and

4. perform A^*-search [30] 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 $T = \{t' \in nodes(t) \mid yield(t') \text{ is ambiguous w.r.t. } G(rootSymbol(t'))\}$. We define a candidate set of productions $R_{ambig} = \{sig(t) \mid t \in T\} \cup \{sig(t') \mid t \in T \wedge t' \in children(t)\}$, and from R_{ambig} construct our template set Π_{tpl} as follows:

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

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.

```
1:  function HEURISTIC($\Pi, t$)
2:      let $ps = \overline{p_{GLL}}|_{\Pi}(G(rootSymbol(t)), yield(t))$
3:      if $\neg\exists(t', s') \in ps.\ t' = t$ then
4:          return $\infty$
5:      end if
6:      if $|ps| = 1$ then
7:          return 0
8:      else
9:          return $min(|ps|, 10)$
10:     end if
11: end function
```

Figure 3.5. The HEURISTIC algorithm.

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. The algorithm is shown in Figure 3.5. 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 7, 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 9, 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).

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. The algorithm for computing the

```
function SUCCESSORS($\Pi$)
    for $\pi \in \Pi_{tpl}$ do
        let $\Pi' = \text{MERGE-FILTERS}(\Pi \cup \{\pi\})$
        if $\neg\text{CONSISTENT}(\Pi')$ then
            next
        else
            yield $\Pi'$
        end if
    end for
end function
```

Figure 3.6. The SUCCESSORS algorithm.

successors of candidate Π, called SUCCESSORS, is shown in Figure 3.6. 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. $\text{A*-SEARCH}(\Pi, t)$ employs a graph search algorithm that in each iteration picks the candidate Π' in its frontier with minimum value of $d(\Pi') + \text{HEURISTIC}(\Pi', t)$, 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.

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.

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

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

3.3 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

Table 3.1. Benchmark suite.

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

as effective as possible? To examine these questions, we ran several case studies in which a co-author built several parsers from benchmarks drawn from different languages. To demonstrate versatility, our chosen benchmarks come from different programming paradigms and styles. Table 3.1 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 [56], and the large dataset comes from a public NASA repository [6]. 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 [12].

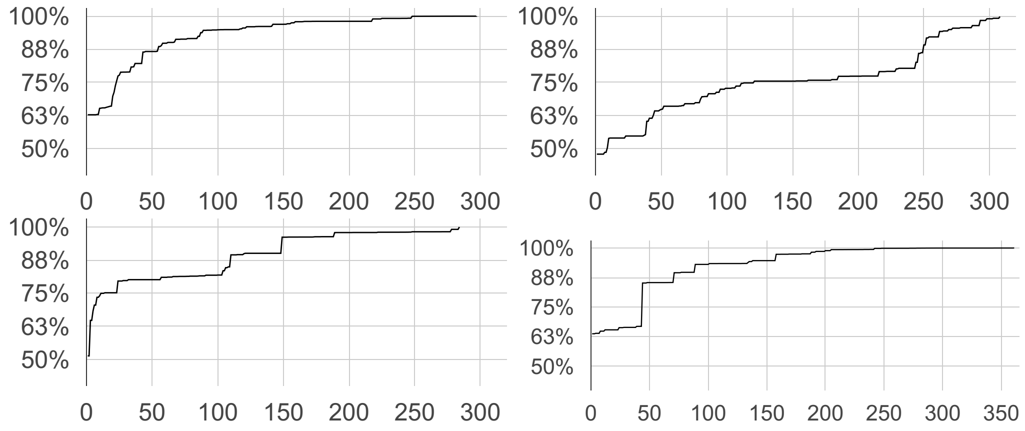


Figure 3.7. Progression plots for Verilog (top left), Tiger (top right), Apache Logs (bottom left), and SQL (bottom right).

3.3.1 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.

3.3.2 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 3.7 shows 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 author and his collaborators 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.

3.3.3 Best Practices

Build bottom-up

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

3.4 Acknowledgements

This chapter, in full, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

Chapter 4

Parsimony

In the previous chapter, we presented Parsify, our initial programming-by-example (PBE) system for constructing parsers by example. Recall the three overarching design principles of Parsify. First, we should minimize the amount of manual coding. Second, the user should receive real-time feedback. Third, we should allow the user to quickly iterate on designs. Despite the merits of this design philosophy and its embodiment in Parsify, over the course of the dissertation author's accumulated experience, several limitations became apparent. This chapter addresses several of Parsify's limitations via new synthesis algorithms and their implementation in a second PBE system called Parsimony. Here we enumerate these limitations and the key algorithmic insights for overcoming each.

Lack of Lexer Capability

Despite the fact that lexical analysis is a key component of the parsing workflow, Parsify had no ability to infer lexer definitions (i.e., named regular expressions). Thus, the user either had to write a custom lexer prior to using Parsify, or hope that that Parsify's internally hardcoded lexer was sufficient.

Parsimony implements a new lexer synthesis engine that synthesizes lexer definitions from example tokens. Our key insight is that there already exists a large corpus of useful, curated regular expressions in the real world: the lexers for existing programming languages. We frame the task of synthesizing lexers as the task of querying a data structure called an R-DAG built from these existing lexers. In contrast to previously known approaches to regular expression inference [3], 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.

Sensitivity to Order

Parsify was highly sensitive to the order in which the user presented examples – the underlying algorithm employed a heuristic to decide the most likely candidate production corresponding to a labeled example – unfortunately, this heuristic prematurely threw away all but the highest scoring candidate from consideration, even when other candidates might have been applicable. In cases where this heuristic was incorrect, the user's only path forward was, unintuitively, to first undo several steps, then present examples in a different order so as to induce different heuristic decisions. The root cause of this behavior was two-fold: (a) Parsify always synthesizes productions one example at a time, thus enforcing a strict ordering on inference steps; and (b) after each inference, Parsify cannot retain multiple candidates simultaneously.

To address this issue, we present a fresh perspective on a classical parsing algorithm first described over 50 years ago, the Cocke-Younger-Kasami (CYK) parsing algorithm [75]. We propose a novel graph data structure built from CYK tables called the *CYK automaton* that efficiently keeps track of 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 worst-case space only linear in the length of the example. In this setting, 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. This formulation also provides a natural way to solve for multiple user-labeled examples simultaneously: by *composing* multiple CYK automata, one for each example.

Ad-hoc Generalization

Parsify had ad-hoc support for generalizing from instances of common patterns such as unbounded repetitions (e.g., Kleene plus), but the mechanism was specialized for repetitions, and it was not obvious how we might extend the approach to other important patterns.

With Parsimony, we make the key insight that many design patterns can in fact 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 new approach by implementing the repetition patterns from Parsify and more (such as recursively defined infix algebraic expressions), simply by defining a CYK automaton for each pattern, along with a schema for the productions to generate based on that pattern.

This chapter covers each of the key improvements described above. In addition, we examine the utility of Parsimony over a wider audience: in contrast to the case study approach presented in Chapter 3, we perform an evaluation of Parsimony's effectiveness via a controlled user study in which 18 programmers previously unfamiliar with Parsimony were asked to complete a series of parser implementation tasks using experimental and control variants of Parsimony. The results of that study are described in this chapter.

In summary, this chapter covers the following contributions:

1. We describe a lexer synthesis algorithm that converges on useful lexer definitions

with only a small number of examples.

2. We present a parser synthesis algorithm that frames synthesis as satisfaction of constraint systems derived from user-provided examples; in this vein, we formalize the notion of parser synthesis constraint systems, then define a novel data structure, the CYK automaton, for efficiently solving these systems.

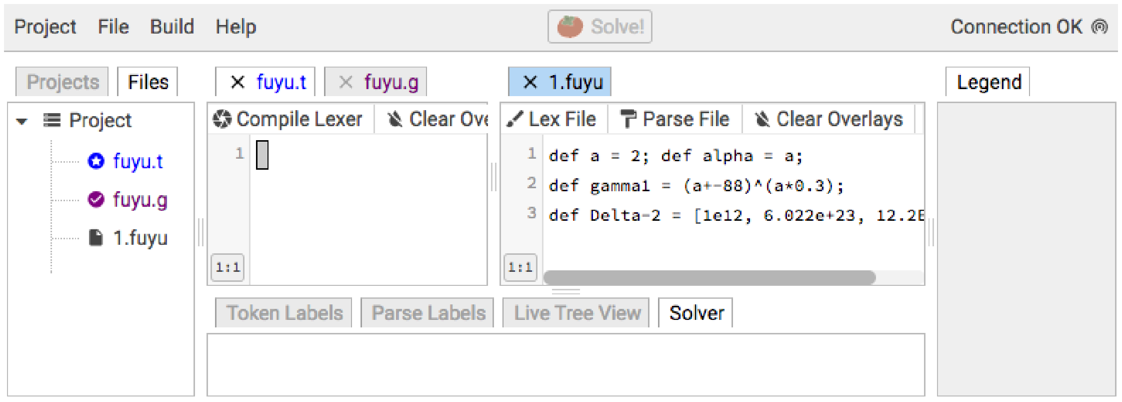
3. We present an extensible framework for detecting and generalizing from examples of common parser design patterns, thus allowing for synthesis of tricky productions that are correct by construction.

4. We present the results of a controlled user study in which 18 Computer Science students, previously unfamiliar with Parsimony, were asked to complete two realistic parser implementation projects using either Parsimony or a control variant stripped of synthesis features. Our results show that Parsimony improves user outcomes as measured by both time to completion and number of mistakes.

4.1 Overview

Parsimony's user interface is shown in Figure 4.1. Its basic functionality includes standard features ubiquitous amongst integrated development environments: (a) a customizable workspace consisting of resizable panes and draggable tabs, (b) a file browser for viewing and managing the contents of a project, and (c) text editors for viewing and editing files. Parsimony also borrows the graphical features first shown in Parsify, such as parse tree visualizations and live coloring of text files based on grammar changes.

Unique to Parsimony, however, are two tabs for interacting with the lexer and parser synthesis engines:



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

Figure 4.1. The Parsimony user interface.

1. *The Token Labels Tab* allows the user to synthesize regular expressions and lexer rules by providing example strings and their intended tokenization.

2. *The Solver Tab* provides rich functionality for synthesizing and previewing grammar productions derived from strings labeled with syntactic categories (i.e., nonterminals).

A Parsimony session starts with a project containing a lexer definition file and parser definition file (both of which start empty), and one or more sample text files containing source code examples of the language being implemented. The user interacts with the Token Labels Tab to add lexer rules to the lexer definition, interacts with the Solver Tab to add productions to the parser definition, and iterates on the design until all sample files parse correctly according to the user's requirements. This iterative process is guided by Parsify-style colored boxes (i.e., colorings), that Parsimony draws over text editor panes to show current progress: as with Parsify, the size, shape, and placement of these boxes indicate how much of the file is covered by the rules written so far.

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 is shown in Figure 4.2.

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

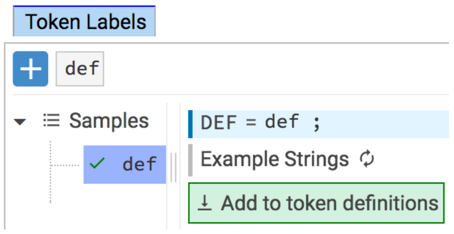
Figure 4.2. Sample of Fuyu source code: `1.fuyu`.

4.1.1 Constructing the Lexer

To begin, we open the lexer definition `fuyu.t` and sample source `1.fuyu` shown at left and right in Figure 4.1, respectively. Since we have just started, `fuyu.t` is empty. The goal will be to fill `fuyu.t` with lexer rules for tokenizing `1.fuyu`.

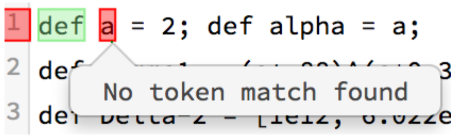
Keywords

We start by synthesizing a lexer rule for the `def` keyword via following three steps: (a) click the Token Labels Tab to activate it, (b) select the substring "`def`" in `1.fuyu`, then (c) add it as an example token by clicking the blue plus sign that appears. The Token Labels Tab then updates its contents, as shown in Figure 4.3. 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*.



```
DEF = def ;
```

Figure 4.3. Token Labels Tab after adding the "`def`" example token.



```
def a = 2; def alpha = a;
No token match found
```

Figure 4.4. Text editor after accepting inferred lexer rule `DEF = def`.

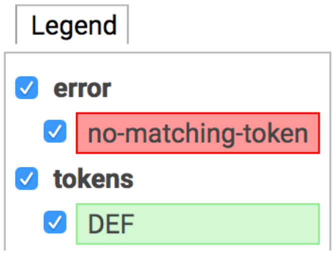


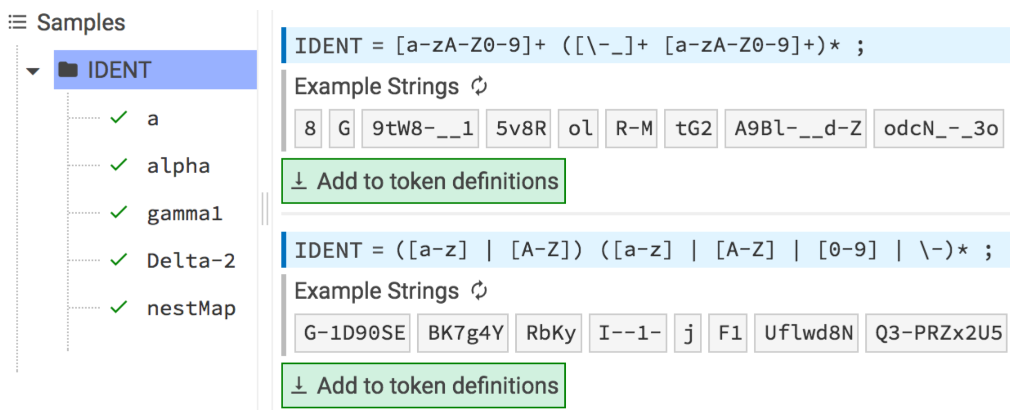
Figure 4.5. Legend after adding new lexer rule for `DEF`.

Parsimony immediately recompiles the lexer and colors `1.fuyu` in response. The coloring, shown in Figure 4.4, 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, shown in Figure 4.5. 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 seems easy enough to write by hand without needing to synthesize it. Identifiers, however, are a more complex sort of token whose lexer rule is correspondingly more challenging to specify. In particular, suppose our language specification dictates that identifiers consist only of alphanumeric and hyphen characters; additionally, the first character must be alphabetical. We want Parsimony to automatically synthesize a lexer rule that meets that specification.



```
IDENT = [a-zA-Z0-9]+ ([\-_]+ [a-zA-Z0-9]+)* ;
```

Example Strings

`8` `G` `9tW8-__1` `5v8R` `ol` `R-M` `tG2` `A9Bl-__d-Z` `odcN_-_3o`

```
IDENT = ([a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9] | \-)* ;
```

Example Strings

`G-1D90SE` `BK7g4Y` `RbKy` `I--1-` `j` `F1` `Uflwd8N` `Q3-PRZx2U5`

Figure 4.6. Token Labels Tab after adding five examples of the `IDENT` token.

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: we are trying to “teach" Parsimony what identifiers look like, but one example is simply 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 can drag and drop multiple samples into their own folder. Shown in Figure 4.6 is a folder labeled `IDENT` into which we have added five example identifiers; the right-hand side of the figure shows that Parsimony has inferred two different candidate rules from the examples contained in that folder.

To gain some intuition about the meaning of these rules, we can ask Parsimony to show us some examples of strings matched by each rule. When we click the *Example Strings* button, Parsimony updates the view with the strings shown in gray. It is clear from them that the first rule permits the presence of underscores, which violates our specification. The second candidate rule, however, seems to be correct based on 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

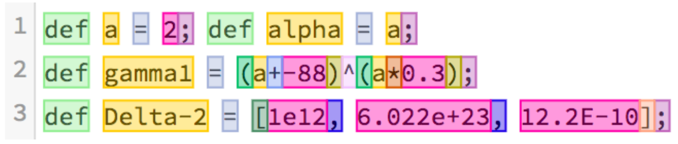
Numeric literals in Fuyu take the form of integers or floating point numbers specified in decimal or scientific notation. The desired lexer should assign such literals the token name `NUMBER`. Shown below are the six numeric literals from `1.fuyu`.

```
2  -88  0.3  1e12  6.022e+23 12.2E-10
```

This is the most complex lexical form in Fuyu, and it would likely take a seasoned veteran of lexical analysis to correctly implement its regular expression on the first try:

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

Parsimony synthesizes this regular expression from just those six examples. In fact, it is the only candidate Parsimony chooses to show 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 regular expression `.*` also matches the examples, too, but it is clearly much more general, and thus an inferior candidate – we define this notion of quality formally in Section 4.2. After accepting the inference, our lexer is complete. `1.fuyu`, with all tokens properly colored, is shown in Figure 4.7.



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

Figure 4.7. `1.fuyu` with all tokens properly colored.

4.1.2 Constructing the Parser

With lexer in hand, we proceed to the parser. In this section, we will 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 (a) selecting "`def a = 2;`", (b) typing "`assign`" into the textbox that appears, then (c) clicking the Solve button. The Solver Tab responds by presenting the following stylized candidate production, depicted in Figure 4.8, that Parsimony has synthesized from the example.

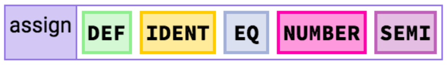


Figure 4.8. Candidate synthesized from one example.

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, shown in Figure 4.9.

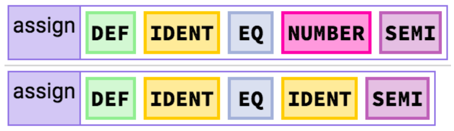


Figure 4.9. Candidates synthesized from two examples.

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 in Figure 4.10.

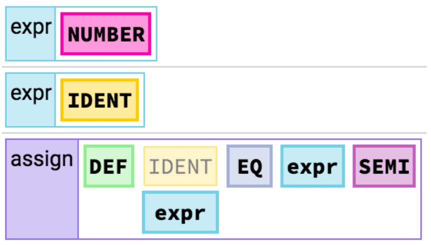


Figure 4.10. Candidates synthesized from four examples.

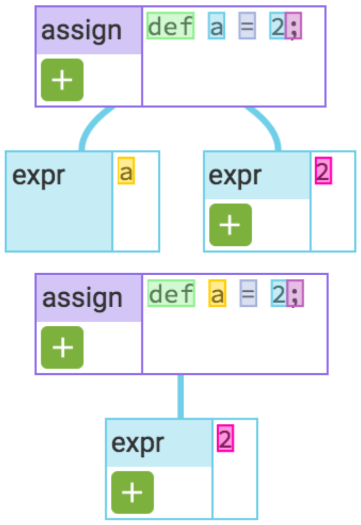
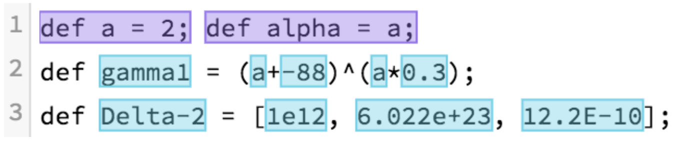


Figure 4.11. Parse tree visualizations.

The first two candidates match our expectation: to parse `NUMBER` and `IDENT` tokens as expressions, add the two productions `expr` $\rightarrow$ `NUMBER` and `expr` $\rightarrow$ `IDENT`. The third candidate has a special form. It indicates that the 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, as depicted in Figure 4.11. 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` $\longrightarrow$ `NUMBER`, `expr` $\longrightarrow$ `IDENT`, and `expr` $\longrightarrow$ `DEF IDENT EQ expr SEMI`. Parsimony automatically recompiles the parser, then colors `1.fuyu` accordingly. The result is shown in Figure 4.12. Note that the first two assignments are now surrounded by colored boxes corresponding to the nonterminal `assign`.



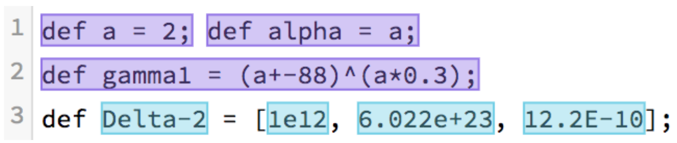
```
def a = 2; def alpha = a;
def gamma1 = (a+-88)^(a*0.3);
def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

Figure 4.12. `1.fuyu` after accepting solution.

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 (a) if this is indeed an algebraic expression, (b) for each operator (`+`,`*`,`^`) whether that operator is left- or right-associative, and (c) 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` $\longrightarrow$ `expr + expr {left}`, `expr` $\longrightarrow$ `expr * expr {left}`, `expr` $\longrightarrow$ `expr ^ expr {right}`, and `expr` $\longrightarrow$ `( expr )`. Additionally, Parsimony synthesizes the following precedence annotation: `priorities { expr` $\longrightarrow$ `expr ^ expr > expr` $\longrightarrow$ `expr * expr ; expr` $\longrightarrow$ `*`

`expr > expr` $\rightarrow$ `expr + expr ; }`. Under the hood, these annotations compile to disambiguating filters 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. The result is shown in Figure 4.13.

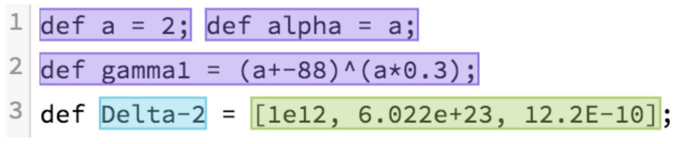


```
1 def a = 2; def alpha = a;
2 def gamma1 = (a+-88)^(a*0.3);
3 def Delta-2 = [1e12, 6.022e+23, 12.2E-10];
```

Figure 4.13. `1.fuyu` after accepting inferences for `expr`.

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 (a) each element of the list is a `NUMBER` or `expr`, (b) the separator between elements is a comma, and (c) the list is surrounded by a pair of square brackets. After confirming, Parsimony synthesizes an idiomatic subgrammar for delimited lists of `expr`s: `array` $\rightarrow$ `[ -array-inner ]`, `-array-inner` $\rightarrow$ `expr`, `-array-inner` $\rightarrow$ `expr COMMA -array-inner`.



```
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];
```

Figure 4.14. `1.fuyu` after accepting inferences for `array`.

The new coloring, shown in Figure 4.14, shows that the array literal parses correctly. However, the parent assignment is still not surrounded by a box for `assign`. The reason is that we never 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` $\to$ `array`, shown in Figure 4.15.

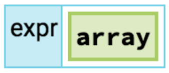


Figure 4.15. Inferred candidate `expr` $\to$ `array`.

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` $\to$ `assign` and `program` $\to$ `assign program`. Our parser is now complete.

4.2 Lexer Synthesis

In this section we formalize Parsimony's algorithms for synthesizing lexers.

4.2.1 A Data Structure for Sets of Regular Expressions

In this section we describe our data structure, called an R-DAG, for representing sets of regular expressions. The structure is designed to support efficient queries utilized by our regex inference algorithm as described in Section 4.2.2.

R-DAG Definition

We first define R-DAG*, a partially ordered set (poset) with properties similar to an R-DAG. We then define R-DAG via reduction from an R-DAG*.

Definition 1 (R-DAG*)**.** An R-DAG* is a poset $\mathscr{D}^* = (R, \sqsubset^*)$ such that

1. $R \subseteq \mathfrak{R}$ is a set of regexes,

2. $\sqsubset^* \subseteq (R \times R)$ is the language containment relation over R such that $\forall r_1, r_2 \in R. \mathscr{L}(r_1) \subset \mathscr{L}(r_2) \Leftrightarrow (r_1, r_2) \in \sqsubset^*$,

3. R contains a designated regex $\top_{\mathscr{D}}$ such that $\mathscr{L}(\top_{\mathscr{D}})$ is the set of all strings, and

4. $\forall r_1, r_2 \in R. \mathscr{L}(r_1) = \mathscr{L}(r_2) \Rightarrow r_1 = r_2$.

By 2 we have that $r_1 \sqsubset^* r_2$ if and only if the language of r_1 is strictly contained in the language of r_2. By 3 we have that R always has a topmost regex $\top_{\mathscr{D}}$. By 4 we have that R contains no two redundant regexes whose languages are identical. An R-DAG* can be viewed equivalently as a directed acyclic graph such that R is its vertex set and $\sqsubset^*$ 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.

Definition 2 (R-DAG)**.** Let $\mathscr{D}^* = (R, \sqsubset^*)$ be an R-DAG*. We define its corresponding R-DAG $\mathscr{D} = (R, \sqsubset)$ to be the transitive reduction of $\mathscr{D}^*$. That is, $\mathscr{D}$ is the graph with the same vertex set R as $\mathscr{D}^*$, but with edge set $\sqsubset$, the unique minimum size relation whose transitive closure is $\sqsubset^*$.

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. Because of this structure, one may design graph algorithms to efficiently answer questions of the sort “What is the *most specific* set of regexes that match strings s_1, s_2, ...?" 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 notions of “most specific" and “largest" are made formal in the following section.

4.2.2 Regular Expression Inference via R-DAG Queries

In this section, we 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 $\mathscr{D} = (R, \sqsubset)$ and a string s. First, we wish to find a set of regexes $\mathscr{H} \subseteq R$ such that every regex in $\mathscr{H}$ matches s. Second, we require $\mathscr{H}$ to be succinct: no two regexes in $\mathscr{H}$ should be related by $\sqsubset^*$. Third, we require $\mathscr{H}$ 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 $\top_{\mathscr{D}}$). These three conditions are captured by the notions of *consistency*, *succinctness*, and *maximality*, defined formally here.

Definition 3 (Consistent)**.** Let S be a set of strings. $\mathscr{H}$ is consistent with S iff $\forall r \in \mathscr{H}, s \in S. s \in \mathscr{L}(r)$.

Definition 4 (Succinct)**.** Let $\mathscr{D} = (R, \sqsubset)$. $\mathscr{H}$ is succinct with respect to $\mathscr{D}$ iff $\forall r, r' \in \mathscr{H}. r \not\sqsubset^* r'$.

Definition 5 (Maximal)**.** Let $\mathscr{D} = (R, \sqsubset)$. Let S be a set of strings. $\mathscr{H}$ is maximal with respect to $(\mathscr{D}, S)$ iff no regex $r \in R$ exists such that:

1. $r \notin \mathscr{H}$,
2. $\forall s \in S. s \in \mathscr{L}(r)$, and
3. $\forall r' \in \mathscr{H}. (r \sqsubset^* r' \vee r' \not\sqsubset^* r)$.

Definition 6 (Horizon)**.** The set $\mathscr{H}$ is a horizon of $(\mathscr{D}, S)$ iff $\mathscr{H}$ is consistent with S, succinct with respect to $\mathscr{D}$, and maximal with respect to $(\mathscr{D}, S)$.

```
function HORIZON($\mathscr{D}, S$)
    $\mathscr{W} \leftarrow \{\top_{\mathscr{D}}\}$ ; $\mathscr{H} \leftarrow \emptyset$
    while $|\mathscr{W}| > 0$ do
        let $r = removeAny(\mathscr{W})$
        let $R_p = \{r' \in predecessors(\mathscr{D}, r) \mid \forall s \in S . s \in \mathscr{L}(r')\}$
        if $|R_p| > 0$ then
            $\mathscr{W} \leftarrow (\mathscr{W} - r) \cup R_p$
        else
            $\mathscr{W} \leftarrow \mathscr{W} - r$ ; $\mathscr{H} \leftarrow \mathscr{H} \cup \{r\}$
        end if
    end while
    return $\mathscr{H}$
end function
```

Figure 4.16. The HORIZON algorithm.

We now define the query HORIZON$(\mathscr{D}, S)$, which computes the horizon of $(\mathscr{D}, S)$. The algorithm is shown in Figure 4.16. Intuitively, HORIZON maintains a worklist $\mathscr{W}$ of vertices to inspect. The worklist initially contains only $\top_{\mathscr{D}}$, the topmost regex that matches any string, and is thus guaranteed to be reachable from any other vertex of $\mathscr{D}$. 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 $\mathscr{H}$. We continue this process until the worklist is exhausted. At completion, $\mathscr{H}$ is a set of regexes that is consistent with S, succinct with respect to $\mathscr{D}$, and maximal with respect to $(\mathscr{D}, S)$. In other words, it is the horizon of $(\mathscr{D}, S)$. 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 $\mathscr{H}$ is succinct with respect to $\mathscr{D}$ and maximal with respect to $(\mathscr{D}, S)$, we know there are no “better" candidates that we have missed.

Theorem 1 (Consistency)**.** HORIZON$(\mathscr{D}, S)$ *is consistent with S.*

Proof. We first prove the following loop invariant: $\forall r \in \mathscr{W}, s \in S. s \in \mathscr{L}(r)$.

Base case

On loop entry, the worklist $\mathscr{W}$ contains only $\top_{\mathscr{D}}$, which is guaranteed to match all strings.

Inductive step

On each iteration of the loop, a regex is added to the worklist $\mathscr{W}$ (line 7) only if it belongs to the set R_p. By construction, every member of R_p matches every string in S, as guaranteed by the predicate $\forall s \in S. s \in \mathscr{L}(r')$ on line 5. Thus, the invariant holds after each iteration.

Since every regex added to $\mathscr{H}$ (line 9) is drawn from $\mathscr{W}$ (line 4), our loop invariant implies that every regex in $\mathscr{H}$ is guaranteed to match every string in S. □

Lemma 1 (Descendant Matching)**.** *Let $\mathscr{D} = (R, \sqsubset)$ be an* R-DAG*, s be a string, and r be a regex in R. If $s \in \mathscr{L}(r)$, then for every descendant r' of r in $\mathscr{D}$, $s \in \mathscr{L}(r')$.*

Proof. For every descendant r' of r, we have that $r \sqsubset^* r'$. By the definition of $\sqsubset^*$, $\mathscr{L}(r) \subset \mathscr{L}(r')$. Thus, if $s \in \mathscr{L}(r)$ then $s \in \mathscr{L}(r')$. □

Theorem 2 (Succinctness)**.** HORIZON$(\mathscr{D}, S)$ *is succinct with respect to $\mathscr{D}$.*

Proof. Let $\mathscr{H} =$ HORIZON$(\mathscr{D}, S)$. Suppose for contradiction that there exist $r, r' \in \mathscr{H}$ such that $r \sqsubset^* r'$. Then there must exist a path p in $\mathscr{D}$ from r to r' such that every vertex in p matches s (Lemma 1). However, r' could not have been added to $\mathscr{H}$ unless it had no predecessors matching s (lines 5-6). This implies that p cannot exist, as p must necessarily go through such a predecessor. □

Theorem 3 (Maximality)**.** HORIZON$(\mathscr{D}, S)$ *is maximal with respect to $(\mathscr{D}, S)$.*

Proof. Let $\mathscr{D} = (R, \sqsubset)$ and $\mathscr{H} = \textsc{horizon}(\mathscr{D}, S)$. Assume for contradiction that there exists $r \in R$ such that

1. $\forall s \in S. s \in \mathscr{L}(r)$ and

2. $r \notin \mathscr{H} \wedge \forall r' \in \mathscr{H}. (r \sqsubset^* r' \vee r' \not\sqsubset^* r)$.

There are two cases to consider:

1. r was excluded from $\mathscr{H}$ because at least one of its predecessors r' also matches s. If so, then some ancestor of r must exist in $\mathscr{H}$, which contradicts assumption 2.

2. r was never inspected (i.e., never added to the worklist), which implies there exists no matching path from $\top_{\mathscr{D}}$ to r. By assumption 1 and Lemma 1, however, such a path must exist, which is a contradiction.

$\square$

4.2.3 Example of Token Inference

To better illustrate the connection between R-DAGs and inference, we now show a small example. Note that although the example we use employs only a small corpus of regular expressions, the actual implementation of Parsimony utilizes an R-DAG constructed from a much larger corpus of thousands of regular expressions.

Consider the set of lexer rules shown in Figure 4.17. To reduce verbosity and improve readability, we have written the lexer rules such that a reference to a terminal name represents a substitution of the terminal's corresponding regex into the referencing body. For example, `HEX-LETTER | DEC-DIGIT` is shorthand for the regex `[a-f]|[0-9]`. We have also used the common syntactic shorthand `[abc]` that desugars to `a|b|c`, as well as the shorthand `.` that represents a single instance of any symbol.

```
          (TOP, .*)
   (HEX-LETTER, [a-f])
    (OCT-DIGIT, [0-7])
    (DEC-DIGIT, [0-9])
    (HEX-DIGIT, HEX-LETTER | DEC-DIGIT)
        (OCTAL, 0 OCT-DIGIT*)
      (DECIMAL, 0 | [1-9] DEC-DIGIT*)
  (HEXADECIMAL, HEX-DIGIT*)
        (FLOAT, -? DECIMAL (\. DEC-DIGIT+)? ([Ee] [+-]? DECIMAL)?)
```

Figure 4.17. Lexer rules for constructing R-DAG.

The R-DAG constructed from this set of of lexer rules is shown in Figure 4.18. For ease of visualization, we have labeled each vertex with the terminal name of the regex rather than regex itself. From the figure, a few features are apparent. First, any two elements connected by a path are also related by language containment. For example, since OCT-DIGIT reaches FLOAT, we know that any string that matches the OCT-DIGIT regex also matches the FLOAT regex. Second, the unique uppermost element of the R-DAG is TOP. This corresponds to the designated element $\top_{\mathscr{D}}$ required by Definition 1. TOP is reachable from every element of the R-DAG, as expected, since the TOP regex matches any string. Notice, also, a particular quirk of this R-DAG: OCTAL is not an ancestor of DECIMAL, even though intuitively, we might think that every octal string also looks like a decimal string. This turns out not to be true in this particular case because the OCTAL regex in fact always expects a 0 digit to appear as the first digit, whereas the regex for DECIMAL also permits non-zero digits in that position.

Finally, Table 4.1 shows several examples of the horizons computed from sets of example strings using this R-DAG.

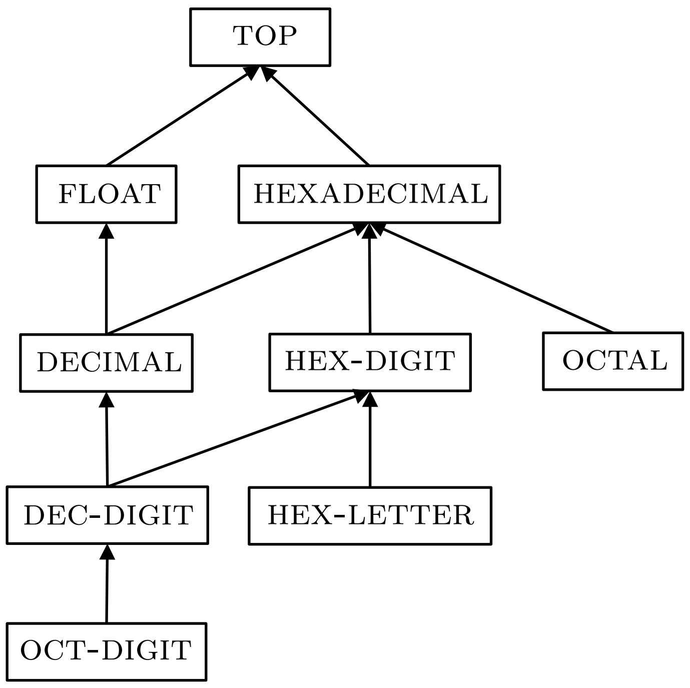


Figure 4.18. Example R-DAG constructed from lexer rules shown in Figure 4.17.

Table 4.1. Example HORIZON queries.

Example Strings	Horizon
"0"	OCT-DIGIT, OCTAL
"5"	OCT-DIGIT
"8"	DEC-DIGIT
"a"	HEX-LETTER
"x"	TOP
"-1.2"	FLOAT
"1e12"	FLOAT, HEXADECIMAL
"0" "5"	OCT-DIGIT
"0" "5" "8"	DEC-DIGIT
"0" "5" "8" "a"	HEX-DIGIT
"1e12" "a"	HEXADECIMAL
"1e12" "-1.2"	FLOAT

4.3 Parser Synthesis

In this section we formalize Parsimony's algorithms for synthesizing parsers. We begin with a preliminary overview of CYK parsing.

4.3.1 Preliminaries: CYK Parsing Algorithm

The Cocke-Younger-Kasami (CYK) parsing algorithm [75] is a dynamic programming algorithm for computing whether a string τ is a member of the language of a grammar G. In other words, it computes whether τ has a full derivation with respect to G. The crucial distinguishing feature of the CYK algorithm is that the algorithm constructs a two-dimensional table M that records, for every substring τ' of τ, the set of nonterminals that derive τ'. The algorithm is shown in Figure 4.19.

We describe the CYK algorithm informally here. We build M from smaller substrings of τ up to larger ones. For each substring of length 1 at each index i, (i.e., single tokens), we check whether some nonterminal A derives that token in a single step via a unit rule of form $A \rightarrow a$. If so, we add A to the set at table element $M_{i,1}$. We do this for every such nonterminal. Having examined every substring of length 1, we proceed to those of length $l = 2$: for each length 2 substring τ' at each index j, we check whether some nonterminal A' derives τ' in a single step via a rule of the form $A' \rightarrow BC$. To do this, we try to split τ' into a prefix and suffix such that M contains B for the prefix and C for the suffix. If we find such a split, we add A' to the set at table element $M_{j,l}$. We iterate this procedure for every nonterminal and for each value of l from 2 up to $|\tau|$. At termination, each element of M is a (possibly empty) set of nonterminals that derive the corresponding substring of τ:

$$A \in M_{i,l} \iff A \Rightarrow^* \tau_{[i\ldots i+l]} \tag{4.1}$$

To determine membership of τ in $\mathscr{L}(G)$, we simply check whether $M_{0,|\tau|}$ contains the start symbol of G. One important remark is that the algorithm makes use of a function $\textsc{cnf}(G)$ that we have not defined. The CYK algorithm as described requires the grammar on which it operates to be in Chomsky normal form (CNF): every production must either be of the form $A \to a$ or $A \to BC$. Well-known algorithms exist for converting any context-free grammar to CNF [26], so we omit the details here and simply take as a given that we have access to the function $\textsc{cnf}(G)$ that takes an arbitrary context-free grammar G and returns its CNF conversion.

```
function CYK($G, \tau$)
    let $(\cdot,\cdot,P,\cdot) = \textsc{cnf}(G)$
    for $i$ in $[0,|\tau|)$ do
        $M_{i,1} = \left\{A \mid A \to a \in P \wedge \tau_{[i]} = a\right\}$
    end for
    for $l$ in $[2,|\tau|]$ do
        for $j$ in $[0,|\tau|-l]$ do
            $M_{j,l} = \{A \mid \exists\, k.A \to BC \in P$
                        $\wedge B \in M_{j,k}$
                        $\wedge C \in M_{j+k,l-k}\}$
        end for
    end for
    return $M$
end function
```

Figure 4.19. The CYK algorithm.

4.3.2 Parser Synthesis Constraint Systems

In this section, we make precise the parser synthesis problem by framing it as satisfaction of constraints over labeled strings.

Definition 7 (Parser Synthesis Constraint System)**.** A *parser synthesis constraint system* is a tuple $\mathbb{C} = (G, \mathbb{F}, \mathbb{L})$ where

1. G is a grammar,
2. $\mathbb{F} = \{\tau^{f_1}, \tau^{f_2}, ..., \tau^{f_k}\}$ is a set of strings called *files* with unique labels $f_1, f_2, ..., f_k$ 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:

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

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 $\mathbb{L}$.

Trivial Solutions

Note that there always exist trivial solutions to any parser synthesis constraint system: for every constraint $\langle A, i, l \rangle^f \in \mathbb{L}$ generate the production $A \rightarrow \tau_{[i \dots i+l]}$. In other words, simply use each sequence of selected terminals as the production body. However, such a solution is usually unprofitable, in the sense that it is unlikely to be the user's intention. In the following sections, we describe our mechanism for finding non-trivial solutions.

4.3.3 A Data Structure for 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 $\tau_{[2 \dots 7]}$. 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 sentential form 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 8 (CYK Automaton)**.** A CYK automaton is a directed graph $Y = (I, E, i_0, I_f, U, \Lambda)$ where $I \subseteq \mathbb{Z}^*$ is a set of vertices, $E \subseteq I \times I$ is a set of edges, $i_0 \in I$ 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 $\lambda x.$ 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, shown in Figure 4.20. 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

```
function BUILD-CYK-AUTOMATON($G, \tau, i_0, i_f$)
    let $(N, \Sigma, \cdot, \cdot) = G$
    let $M = \text{CYK}(G, \tau)$
    $(I, E, \Lambda) \leftarrow (\{0 \leq i \leq |\tau|\}, \emptyset, \lambda x.\emptyset)$
    for $i$ in $[i_0, i_f - 1)$ do
        $E \leftarrow E \cup \{(i, i+1)\}$
        $\Lambda \leftarrow \Lambda[(i, i+1) \mapsto \{\tau_{[i]}\}]$
    end for

    for $i$ in $[i_0, i_f)$ do
        for $l$ in $[1, i_f - i]$ do
            if $M_{i,l} \neq \emptyset$ then
                $E \leftarrow E \cup \{(i, i+l)\}$
                $\Lambda \leftarrow \Lambda[(i, i+l) \mapsto \Lambda(i, i+l) \cup M_{i,l}]$
            end if
        end for
    end for
    return $(I, E, i_0, \{i_f\}, N \cup \Sigma, \Lambda)$
end function
```

Figure 4.20. The BUILD-CYK-AUTOMATON algorithm.

at that position. By construction, any path between vertex 0 and vertex $|\tau|$ corresponds to a set of sentential forms that derive τ. That is, for a path $p = i_0, i_1, ..., i_n$ such that i_0 is the start vertex and i_n is a final vertex, its set of corresponding sentential forms is given by the n-ary Cartesian product

$$\Lambda(i_0, i_1) \times \Lambda(i_1, i_2) \times ... \times \Lambda(i_{n-1}, i_n) \tag{4.2}$$

4.3.4 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-FULL.

Case 1: A Single Parse Constraint

We first consider synthesis constraint systems with only one parse constraint. Let us revisit the example from Section 4.3.3, in which we have the string *ab*, a partially implemented grammar G_1 with productions $A \to a$ and $B \to b$, but wish for *ab* to derive from a new nonterminal S that has yet to be implemented. As we saw in Section 4.1, to do this using Parsimony we simply highlight the text *ab*, type the label S into the textbox that appears, then press the Solve button. Under the hood, this sequence of user operations constructs the following parser synthesis constraint system:

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

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 $\tau = ab, i_0 = 0, I_f = \{2\}$:

$$\begin{tikzcd} 0 \arrow[r, "\{A, a\}"] & 1 \arrow[r, "\{B, b\}"] & 2 \end{tikzcd}$$

The shortest (and only) path is $0, 1, 2$. Taking the n-ary Cartesian product of edge attributes along the path, as defined in (4.2), we have $\Lambda(0,1) \times \Lambda(1,2) =$

$$\{A, a\} \times \{B, b\} = \{(A,B), (A,b), (a,B), (a,b)\}.$$

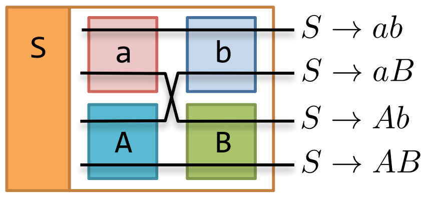


Figure 4.21. Candidate matrix $S[\![\{a,A\}\,\{b,B\}]\!]$.

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 $\mathbb{C}_1$. Given the options, the question is “which solution should we choose?" In the absence of more information, Parsimony cannot answer this question. Rather than make an arbitrary choice, Parsimony displays all the possibilities and leaves the choice to the user.

Candidate Matrices

To succinctly represent a potentially large space of choices, Parsimony uses a graphical representation called a *candidate matrix*, an example of which is shown in Figure 4.21.

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 $X[\![N_1 N_2 \ldots N_k]\!]$ 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. Figure 4.21 explicitly enumerates the four productions encoded by the 2-column candidate matrix $S[\![\{a,A\}\,\{b,B\}]\!]$.

Each production encoded by a candidate matrix $\mathbb{M}$ is called a *valuation* of $\mathbb{M}$. To

```
1: function $\textsc{parsynth}_1^{\triangleleft}(\mathbb{C})$
2:     let $(G, \{\tau^f\}, \{\langle A, i, l\rangle^f\}) = \mathbb{C}$
3:     let $Y = \textsc{build-cyk-automaton}(G, \tau^f, i, i+l)$
4:     return $\Psi_\Lambda^A(\textsc{shortest-path}(Y, i, i+l))$
5: end function
```

Figure 4.22. The $\textsc{parsynth}_1^{\triangleleft}$ algorithm.

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 as follows:

$$\Psi_\Lambda^X(i_0, ..., i_n) = X[\![\Lambda(i_0, i_1) \ldots \Lambda(i_{n-1}, i_n)]\!]$$

We additionally define the following *enumeration function* ENUM, which given a candidate matrix $\mathbb{M}$, returns the set of all valuations of $\mathbb{M} = X[\![N_j]\!]_{j=0}^n$:

$$\textsc{enum}(\mathbb{M}) = \{X \rightarrow x_0 x_1 ... x_n \mid (x_0, x_1, ..., x_n) \in N_0 \times N_1 \times ... \times N_n\}$$

Algorithm $\textsc{parsynth}_1^{\triangleleft}$

The algorithm just sketched, called $\textsc{parsynth}_1^{\triangleleft}$, is shown in Figure 4.22. The primary lines of interest are lines 3-4 in which we construct a CYK automaton Y then construct and return the candidate matrix corresponding to the shortest path through Y.

Case 2: Non-Overlapping Parse Constraints

Algorithm $\textsc{parsynth}_1^{\triangleleft}$ can only handle constraint systems with a single parse constraint. As a first step towards generalizing our algorithm to handle multiple constraints, we consider the simplest case in which no two parse constraints overlap. Two parse constraints $\langle A, i, l\rangle^f$ and $\langle B, j, k\rangle^{f'}$ *overlap* when

$$\left(f = f'\right) \wedge \left(\{m \mid i \leq m < i + l\} \cap \{n \mid j \leq n < j + k\} \neq \emptyset\right)$$

```
function PARSYNTH$_2^{\lhd}(\mathbb{C})$
    let $(G, \{\tau^{f_j}\}_{j=0}^{n}, \mathbb{L}) = \mathbb{C}$
    $\widetilde{\mathbb{M}} \leftarrow \emptyset$
    for $\langle A, i, l\rangle^{f_k} \in \mathbb{L}$ do
        let $Y =$ BUILD-CYK-AUTOMATON$(G, \tau^{f_k}, i, i+l)$
        let $\mathbb{M} = \Psi_{\Lambda}^{A}($SHORTEST-PATH$(Y, i, i+l))$
        $\widetilde{\mathbb{M}} \leftarrow \widetilde{\mathbb{M}} \cup \{\mathbb{M}\}$
    end for
    return $\widetilde{\mathbb{M}}$
end function
```

Figure 4.23. The PARSYNTH$_2^{\lhd}$ algorithm.

where $f = f'$ stipulates that both constraints reference the same file, and the second conjunct stipulates that both constraints reference some of the same indices into that file.

A straightforward approach to handle this more general case is to construct more than one candidate matrix – in particular, one per parse constraint. The revised algorithm, PARSYNTH$_2^{\lhd}$, appears in Figure 4.23. Note that on line 7 we accumulate each additional candidate matrix, then return the entire set $\widetilde{\mathbb{M}}$ on line 9.

Example

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 "`id` = `id`" and "`id` = 1" with the nonterminal S (statements). The synthesis task is to infer one or more productions for S.

$$\mathbb{C}_2 = (G_2, \mathbb{F}_2, \mathbb{L}_2)$$

$$G_2 = (\{E, S\}, \{\texttt{id}, 1, =\}, \{E \to \texttt{id}, E \to 1\}, S)$$

$$\mathbb{F}_2 = \{\texttt{id} = \texttt{id}^{f_1}, \texttt{id} = 1^{f_2}\}$$

$$\mathbb{L}_2 = \{\langle S, 0, 3\rangle^{f_1}, \langle S, 0, 3\rangle^{f_2}\}$$

$$\begin{aligned}\textsc{ParSynth}_2^{\triangleleft}(\mathbb{C}_2) = \{\mathbb{M}_1, \mathbb{M}_2\}& \\ \mathbb{M}_1 &= S[\![\{E,\texttt{id}\}\,\{=\}\,\{E,\texttt{id}\}]\!] \\ \mathbb{M}_2 &= S[\![\{E,\texttt{id}\}\,\{=\}\,\{E,\texttt{1}\}]\!]\end{aligned}$$

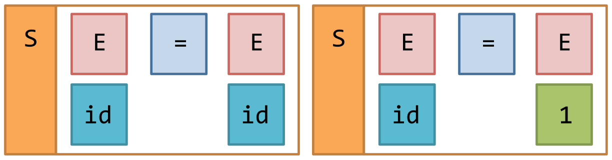


Figure 4.24. Solution to $\mathbb{C}_2$ via $\textsc{ParSynth}_2^{\triangleleft}$.

The computed solution is shown in Figure 4.24, where we have depicted $\mathbb{M}_1$ and $\mathbb{M}_2$ textually at top and visually at bottom. 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 $S \rightarrow \texttt{id} = E$. 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 $\mathbb{M}_2$ in Figure 4.24 are redundant – we need only one of the two since both share the desired valuation $S \rightarrow \texttt{id} = E$. 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.

Definition 9 (CYK Automaton Intersection)**.** Let $Y = (I, E, i_0, I_f, U, \Lambda)$ and $Y' = (I', E', i'_0, I'_f, U', \Lambda')$. The intersection of Y and Y', written $Y \mathbin{\tilde{\cap}} Y'$, is defined as follows:

$$\begin{aligned}
Y \mathbin{\tilde{\cap}} Y' &= (I \times I', E^{\cap}, (i_0, i'_0), I_f \times I'_f, \Lambda^{\cap}) \\
\Lambda^{\cap} &= \lambda \left((x,x'),(y,y')\right) . \Lambda\left((x,y)\right) \cap \Lambda'\left((x',y')\right) \\
E^{*} &= \left\{ \left((x,x'),(y,y')\right) \mid (x,y) \in E \wedge (x',y') \in E' \right\} \\
E^{\cap} &= \left\{ e \in E^{*} \mid \Lambda^{\cap}(e) \neq \emptyset \right\}
\end{aligned}$$

We say Y and Y' are compatible, written $\textsc{compatible}(Y, Y')$, iff the intersection $Y \mathbin{\tilde{\cap}} Y'$ contains a path from start vertex (i_0, i'_0) to final vertex $i_f \in I_f \times I'_f$.

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 $Y \mathbin{\tilde{\cap}} Y'$ corresponds to a common set of sentential forms shared by both Y and Y'. If $\neg\textsc{compatible}(Y, Y')$, then there exists no such shared sentential form.

Partitioning

Our partitioning algorithm is shown in Figure 4.25. 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 $\textsc{partition}$. 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 $\textsc{score}_{\mathbb{Y}}$ 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 $\textsc{partition}$ should be a set $\mathbb{Y}'$ such that $\left|\mathbb{Y}'\right| \leq |\mathbb{Y}|$, 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 in Figure 4.26, 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.

The revised algorithm, $\textsc{parsynth}_3^{\triangleleft}$, is shown in Figure 4.27. The main revision from $\textsc{parsynth}_2^{\triangleleft}$ is that we first partition the set of automata on line 7 before constructing candidate matrices from them.

The computed solution $\textsc{parsynth}_3^{\triangleleft}(\mathbb{C}_2)$ is

$$\{\mathbb{M}_3\} = \left\{S[\![\{E, \texttt{id}\}\,\{=\}\,\{E\}]\!]\right\}.$$

There are two key features of this solution to note. First, there is only one candidate matrix, not two as with $\textsc{parsynth}_2^{\triangleleft}$. 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 $S \to E = E$ and $S \to \texttt{id} = E$. 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.

```
function PARTITION($\mathbb{Y}$)
    let $\mathtt{ays} = \{((A_1,Y_1),(A_2,Y_2)) \mid ((A_1,Y_1),(A_2,Y_2)) \in \mathbb{Y}^2 \wedge$
            $A_1 = A_2 \wedge \text{COMPATIBLE}(Y_1,Y_2)\}$
    if ($\mathtt{ays} \neq \emptyset$) then
        let $(\mathtt{ay}_1, \mathtt{ay}_2) = \mathit{first}(\mathit{sortDescBy}(\text{SCORE}_{\mathbb{Y}}, \mathtt{ays}))$
        let $((A_1,Y_1),(A_2,Y_2)) = (\mathtt{ay}_1, \mathtt{ay}_2)$
        let $Y_{12} = Y_1 \,\widetilde{\cap}\, Y_2$
        return PARTITION($\mathbb{Y} - \{\mathtt{ay}_1, \mathtt{ay}_2\} \cup \{(A_1, Y_{12})\}$)
    else return $\mathbb{Y}$
    end if
end function

function SCORE$_{\mathbb{Y}}$($(A_1,Y_1),(A_2,Y_2)$)
    if ($A_1 \neq A_2$) then return 0
    end if
    return $\sum_{(A_3,Y_3)\in\mathbb{Y} \text{ s.t. } A_3=A_1}$ SCORE-ONE-TRIPLET($Y_1,Y_2,Y_3$)
end function

function SCORE-ONE-TRIPLET($Y_1,Y_2,Y_3$)
    if (COMPATIBLE($Y_1,Y_3$) $\wedge$ COMPATIBLE($Y_2,Y_3$) $\wedge$ COMPATIBLE($Y_1 \,\widetilde{\cap}\, Y_2, Y_3$))
    then
        return 1
    else return 0
    end if
end function
```

Figure 4.25. The PARTITION algorithm. $\mathbb{Y}$ is a set of pairs (A,Y) where Y is a CYK automaton and A is its corresponding nonterminal.

Case 4: Overlapping Parse Constraints

An additional complication occurs when constraints are permitted to 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

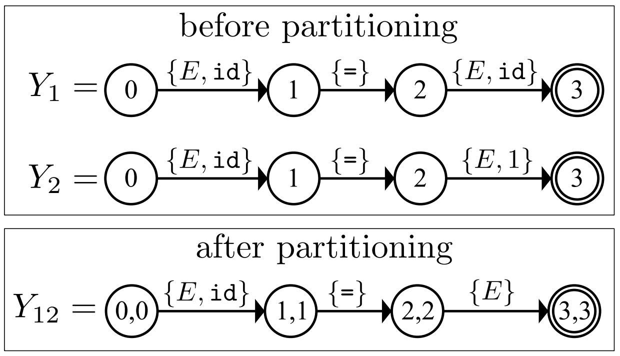


Figure 4.26. Before and after partitioning CYK automata for $\mathbb{C}_2$.

```
1: function PARSYNTH$_3^{\lhd}(\mathbb{C})$
2:     let $(G, \{\tau^{f_j}\}_{j=0}^{n}, \mathbb{L}) = \mathbb{C}$
3:     $\mathbb{Y} \leftarrow \emptyset$
4:     for $\langle A, i, l\rangle^{f_k} \in \mathbb{L}$ do
5:         $\mathbb{Y} \leftarrow \mathbb{Y} \cup \{(A, \text{BUILD-CYK-AUTOMATON}(G, \tau^{f_k}, i, i+l))\}$
6:     end for
7:     for $(A, Y = (\cdot, \cdot, i_0, \{i_f\}, \cdot, \Lambda)) \in \text{PARTITION}(\mathbb{Y})$ do
8:         let $\mathbb{M} = \Psi_{\Lambda}^{A}(\text{SHORTEST-PATH}(Y, i_0, i_f))$
9:         $\widetilde{\mathbb{M}} \leftarrow \widetilde{\mathbb{M}} \cup \{\mathbb{M}\}$
10:     end for
11:     return $\widetilde{\mathbb{M}}$
12: end function
```

Figure 4.27. The PARSYNTH$_3^{\lhd}$ algorithm.

parse constraints like so: $\underbrace{\texttt{id=}\underbrace{\texttt{id+id}}_{E}}_{S}$.

$$\mathbb{C}_4 = (G_4, \mathbb{F}_4, \mathbb{L}_4)$$

$$G_4 = G_2$$

$$\mathbb{F}_4 = \{\texttt{id} = \texttt{id} + \texttt{id}^{f_1}\}$$

$$\mathbb{L}_4 = \{\langle S, 0, 5\rangle^{f_1}, \langle E, 2, 3\rangle^{f_1}\}$$

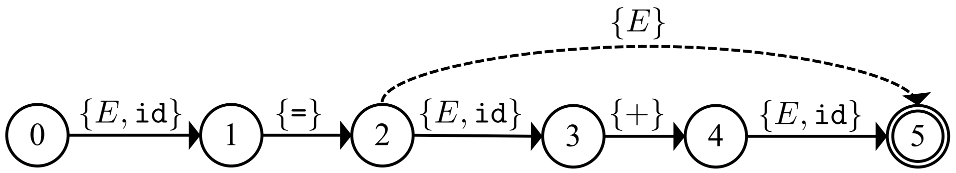
The user's intention is to specify that the enclosing context `id=id+id` is an example of a statement S, and that nested within it `id+id` is an example of an expression E. Unfortunately, PARSYNTH$_3^{\triangleleft}$ ignores this nested relationship:

$$\text{PARSYNTH}_3^{\triangleleft}(\mathbb{C}_4) = \{\mathbb{M}_4, \mathbb{M}_5\}$$

$$\mathbb{M}_4 = S[\![\{E,\texttt{id}\}\,\{=\}\,\{E,\texttt{id}\}\{+\}\{E,\texttt{id}\}]\!]$$

$$\mathbb{M}_5 = E[\![\{E,\texttt{id}\}\,\{+\}\,\{E,\texttt{id}\}]\!]$$

To see the problem, examine the CYK automaton corresponding to $\mathbb{M}_4$ (ignore the dotted edge for now):



The subpath 2,3,4,5 corresponds to the substring `id+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 $\mathbb{M}_4' = S[\![\{E,\texttt{id}\}\,\{=\}\,\{E\}]\!]$ 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,

```
function PARSYNTH$_4^{\lhd}(\mathbb{C})$
    for $(A, Y = (\cdot,\cdot,i_0,\{i_f\},\cdot,\Lambda)) \in$ PARTITION(APPLY-NESTING($\mathbb{C}$)) do
        let $\mathbb{M} = \Psi_{\Lambda}^{A}($SHORTEST-PATH$(Y, i_0, i_f))$
        $\widetilde{\mathbb{M}} \leftarrow \widetilde{\mathbb{M}} \cup \{\mathbb{M}\}$
    end for
    return $\widetilde{\mathbb{M}}$
end function
```

Figure 4.28. The PARSYNTH$_4^{\lhd}$ algorithm.

```
function APPLY-NESTING($\mathbb{C}$)
    let $(G, \{\tau^{f_j}\}_{j=0}^{n}, \mathbb{L}) = \mathbb{C}$
    $\mathbb{Y} \leftarrow \emptyset$
    for $L = \langle A, i, l\rangle^{f_k} \in \mathbb{L}$ do
        let $\mathbb{L}' = \{L' \in \mathbb{L} \mid L$ contains $L'\}$
        $Y \leftarrow$ BUILD-CYK-AUTOMATON$(G, \tau^{f_k}, i, i+l)$
        for $L' \in \mathbb{L}'$ do
            $Y \leftarrow$ SUMMARIZE$(Y, L')$
        end for
        $\mathbb{Y} \leftarrow \mathbb{Y} \cup \{(A, Y)\}$
    end for
    return $\mathbb{Y}$
end function
```

Figure 4.29. The APPLY-NESTING algorithm. SUMMARIZE takes an automaton and parse constraint and returns the automaton with constrained paths replaced by a summary edge.

one possible valuation for $\mathbb{M}_4'$ and $\mathbb{M}_5$ is $S \rightarrow \texttt{id} = E$ and $E \rightarrow E + E$.

The revised algorithm PARSYNTH$_4^{\lhd}$, which incorporates summarization, is shown in Figure 4.28. The key revision is the call to APPLY-NESTING on line 2 that takes as input constraint system $\mathbb{C}$ and returns a set $\mathbb{Y}$ with summary edges inserted. The implementation of APPLY-NESTING is shown in Figure 4.29.

Case 5a: Constrained Patterns – Lists

Consider the constraint system $\mathbb{C}_5$, which models the scenario in which the user has selected and labeled the string `[1;id;1]` with the nonterminal A (arrays).

$$\mathbb{C}_5 = (G_5, \mathbb{F}_5, \mathbb{L}_5)$$

$$G_5 = (\{E\}, \{\texttt{[}, \texttt{]}, \texttt{;}, 1, \texttt{id}\}, \{E \to 1, E \to \texttt{id}\}, E)$$

$$\mathbb{F}_5 = \{\texttt{[1;id;1]}^{f_1}\}$$

$$\mathbb{L}_5 = \{\langle A, 0, 7\rangle^{f_1}\}$$

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, the solution $\textsc{parsynth}^{\lhd}_4(\mathbb{C}_5)$ simply hardcodes the fact that the literal array must contain exactly 3 elements: $\textsc{parsynth}^{\lhd}_4(\mathbb{C}_5) =$

$$\{\mathbb{M}_6\} = \left\{A[\![\{\texttt{[}\}\,\{E,1\}\,\{\texttt{;}\}\,\{E,\texttt{id}\}\,\{\texttt{;}\}\,\{E,1\}\,\{\texttt{]}\}]\!]\right\}$$

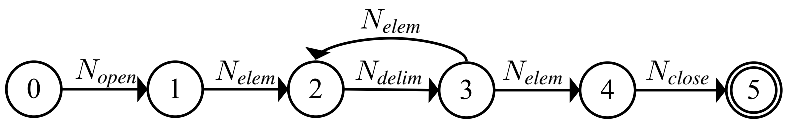
Our strategy for handling this case is two-fold. First, we detect instances of common grammar design patterns using the machinery for intersection of CYK automata. Second, for each pattern we predefine carefully crafted schema for productions; the schema contain holes to be instantiated with symbols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

Pattern Detection

Suppose Y_6 is the CYK automaton, shown below, from which we computed $\mathbb{M}_6$.

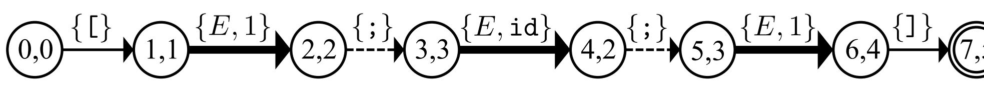
$$\begin{tikzcd}
0 \arrow[r, "\{\texttt{[}\}"] & 1 \arrow[r, "\{E,1\}"] & 2 \arrow[r, "\{\texttt{;}\}"] & 3 \arrow[r, "\{E,\texttt{id}\}"] & 4 \arrow[r, "\{\texttt{;}\}"] & 5 \arrow[r, "\{E,1\}"] & 6 \arrow[r, "\{\texttt{]}\}"] & 7
\end{tikzcd}$$

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 $Y^{\bullet}_{edlist}$:



where $N_{open}, N_{close}, N_{elem},$ and 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 encloser terminals (`[]`, `()`, `{}`), 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 $Y^{\bullet}_{edlist}$. If $\text{COMPATIBLE}(Y_6, Y^{\bullet}_{edlist})$, then we have detected a match.

The intersection $Y_6 \mathbin{\tilde{\cap}} Y^{\bullet}_{edlist}$ 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 $\{$`;`$\}$ and gives us the set of possible delimiters $N^{\cap}_{delim}$. Analogously, the set intersection of all bolded edge attributes is $\{E\}$ and gives us the set of possible elements $N^{\cap}_{elem}$. Finally, the first and last edge attributes are $\{$`[`$\}$ and $\{$`]`$\}$, which give us the sets of possible open and closing enclosers $N^{\cap}_{open}$ and $N^{\cap}_{close}$.

Schema Instantiation

The last step is to instantiate productions. Parsimony contains the following pre-defined schema, named $P^{\bullet}_{edlist}$, 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.

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

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

$$\bullet_{open} \in N^{\cap}_{open} \qquad \bullet_{close} \in N^{\cap}_{close} \qquad \bullet_{elem} \in N^{\cap}_{elem} \qquad \bullet_{delim} \in N^{\cap}_{delim}$$

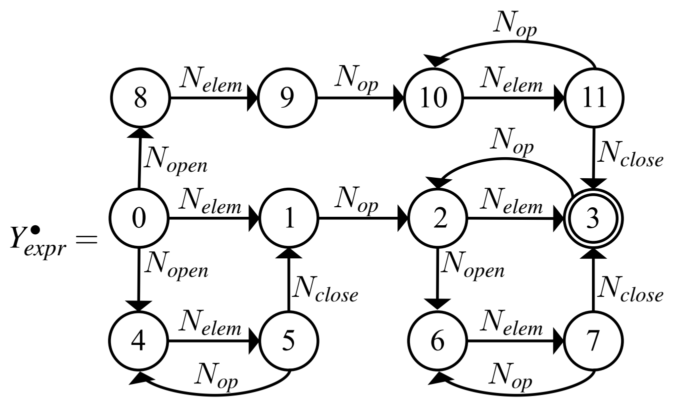
Additionally, $\bullet_{lhs}$ is a special placeholder instantiated with A, the nonterminal from the originating parse constraint $\langle A, 0, 7\rangle^{f_1}$. Fully instantiated, our solution is

$$\left\{ A \rightarrow \texttt{[}A_{fresh}\texttt{]}\ \ ,\ A_{fresh} \rightarrow E\ \ ,\ A_{fresh} \rightarrow E\,\texttt{;}A_{fresh} \right\}$$

As already discussed in Section 4.1, 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 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 and defer their treatment to Appendix B. We encapsulate list pattern detection and schema instantiation in procedure LIST-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.

Case 5b: Constrained Patterns - Algebraic Expressions

We now describe a more complex instantiation of our pattern framework: algebraic expressions. The core approach is the same as for list patterns: we define a CYK automaton that encodes the pattern, then specify schema with placeholders to be filled by symbols captured during pattern matching. The pattern automaton $Y^{\bullet}_{expr}$ is shown below.



The sets N_{open}, N_{close}, and N_{elem} are defined the same as for list patterns. N_{op} is a statically predefined set of common binary operator terminals (`+`,`-`,`*`,`/`, etc.). Together, $Y^{\bullet}_{expr}$ encodes the shape of algebraic expressions with at most one level of parenthetical nesting (e.g., "`a+b`", "`1*(2+3)`", or "`(1/2)`").

To detect an instance of the pattern, as before, we compute the intersection of $Y^{\bullet}_{expr}$ with respect to the automaton being matched. The sets $N^{\cap}_{open}$, $N^{\cap}_{close}$, and $N^{\cap}_{elem}$ are computed, as before, via set intersection of all corresponding edge attributes. However, the set $N^{\cup}_{op}$ is instead computed via set union, not set intersection, of corresponding edge attributes. The result is that $N^{\cup}_{op}$ contains the set of all plausible binary operator terminals seen during pattern matching. Finally, the production schema $P^{\bullet}_{expr}$ is shown below:

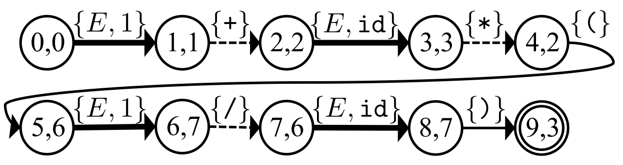
$$P^{\bullet}_{expr} = \begin{Bmatrix} \forall \star \in N^{\cup}_{op} \,.\, \bullet_{elem} \to \bullet_{elem} \star \bullet_{elem} \\ \\ \bullet_{elem} \to \bullet_{open} \bullet_{elem} \bullet_{close} \end{Bmatrix}$$

The special form $\forall \star \in N^{\cup}_{op} \,.\, \bullet_{elem} \to \bullet_{elem} \star \bullet_{elem}$ instantiates to multiple productions: one production per binary operator as captured by $N^{\cup}_{op}$. The result is that an instantiation of $P^{\bullet}_{expr}$ contains a production for each detected binary operator.

Consider the following constraint system, which models the scenario in which the user has selected and labeled the substring `1+id*(1/id)` with the nonterminal E (expressions).

$$\begin{aligned}
&\mathbb{C}_6 = (G_6, \mathbb{F}_6, \mathbb{L}_6) \\
&G_6 = (\{E\}, \{\texttt{(}, \texttt{)}, \texttt{1}, \texttt{id}, \texttt{+}, \texttt{*}, \texttt{/}\}, \{E \to \texttt{1}, E \to \texttt{id}\}, E) \\
&\mathbb{F}_6 = \{\texttt{1+id*(1/id)}^{f_1}\} \\
&\mathbb{L}_6 = \{\langle E, 0, 9\rangle^{f_1}\}
\end{aligned}$$

Suppose Y_7 is the CYK automaton computed from the parse constraint $\langle E, 0, 9\rangle^{f_1}$. Then the intersection $Y_7 \,\widetilde{\cap}\, Y^{\bullet}_{expr}$ is



where the intersection of bolded edge attributes is $N^{\cap}_{elem} = \{E\}$, the union of dashed edge attributes is $N_{op} = \{\texttt{+}, \texttt{*}, \texttt{/}\}$, and the normally drawn edges give us $N^{\cap}_{open} = \{\texttt{(}\}$ and $N^{\cap}_{close} = \{\texttt{)}\}$.

The instantiation of $P^{\bullet}_{expr}$ with these parameters is

$$\{E \to E\texttt{+}E\ ,\ E \to E\texttt{*}E\ ,\ E \to E\texttt{/}E\ ,\ E \to \texttt{(}E\texttt{)}\}$$

which indeed satisfies $\mathbb{C}_6$. Notice, however, that this solution is somewhat unsatisfactory: the resulting grammar is ambiguous because it fails to specify the relative associativities and precedences with respect to the binary operators. To handle this situation, we additionally query the user for information necessary to resolve the ambiguity. In particular, a graphical wizard asks the user (a) to assign an associativity to each operator, and (b) to rank those operators by their precedence. From these user responses, we construct a set of disambiguating filters that resolve the ambiguities with respect to those operators:

1. For each left-associative (resp. right-associative) set O of operators with equal precedence rank, instantiate schema `left{` $\forall \star \in O.\ \bullet_{elem} \to \bullet_{elem} \star \bullet_{elem}$ `}` (resp. `right{` $\forall \star \in O.\ \bullet_{elem} \to \bullet_{elem} \star \bullet_{elem}$ `}`) that permits only derivations respecting the specified associativity.

2. For each set O_{high} of precedence rank i and set O_{low} of precedence rank $i-1$, instantiate schema

$$\texttt{priorities}\ \{\forall \star \in O_{high}, \star' \in O_{low}.$$
$$\bullet_{elem} \to \bullet_{elem} \star \bullet_{elem} > \bullet_{elem} \to \bullet_{elem} \star' \bullet_{elem}\ \}$$

that permits only derivations respecting the specified precedence relationship.

We encapsulate the described expression pattern detection and schema instantiation in procedure EXPR-HEURISTIC($\mathbb{Y}$), which returns a 3-tuple $(P, \Pi, \mathbb{L})$ where P is an instantiated set of production schemas (i.e., a set of productions), Π is an instantiated set of disambiguating filters, and $\mathbb{L}$ is the set of parse constraints from which they derive.

The Algorithm PARSYNTH-FULL

The parser synthesis algorithm, PARSYNTH-FULL, is shown in Figure 4.30. We make three key modifications from $\text{PARSYNTH}_4^{\triangleleft}$. First, we repeatedly attempt to match patterns (lines 3-21), accumulating all instantiated productions and disambiguation filters until no more matches are found. Second, instead of returning just a set of candidate matrices, we return a 3-tuple $(G', \Pi', \widetilde{\mathbb{M}})$ containing the modified grammar, instantiated disambiguating filters, and computed candidate matrices. Finally, we perform two passes of the loop on lines 25-28. With almost no additional machinery, this tweak provides the advantage that solutions computed in the second pass can take advantage of those produced by the first. In particular, the operation $G' \,\widetilde{\uplus}\, \widetilde{\mathbb{M}}$ on line 23 inserts into G' productions equivalent to the EBNF form $X \to (A_1|...|A_m)...(Z_1|...|Z_n)$ for each candidate matrix $X[\![\{A_1,...,A_m\},...,\{Z_1,...,Z_n\}]\!]$ in $\widetilde{\mathbb{M}}$. To see why this is valuable, recall Section 4.1.2, in which we synthesized the following candidate matrices:

$$\texttt{expr}[\![\{\texttt{IDENT}\}]\!]$$

$$\texttt{expr}[\![\{\texttt{NUMBER}\}]\!]$$

$$\texttt{assign}[\![\{\texttt{DEF}\}\{\underline{\texttt{expr}},\texttt{IDENT}\}\{\texttt{EQ}\}\{\texttt{expr}\}\{\texttt{SEMI}\}]\!]$$

The underlined nonterminal `expr` is computed in the second pass by making use of the candidate production $\texttt{expr} \to \texttt{IDENT}$, which was computed in the first pass.

```
function $\textsc{parsynth-full}(\mathbb{C} = (G, \mathbb{F}, \mathbb{L}))$
    $G' \leftarrow G$ ; $\mathbb{L}' \leftarrow \mathbb{L}$ ; $\Pi' \leftarrow \emptyset$ ; $\widetilde{\mathbb{M}} \leftarrow \emptyset$ ; $\texttt{change?} \leftarrow \mathit{true}$
    while $\texttt{change?}$ do
        $\mathbb{Y} \leftarrow \textsc{partition}(\textsc{apply-nesting}((G', \mathbb{F}, \mathbb{L}')))$
        let $(P, \mathbb{L}'') = \textsc{list-heuristic}(\mathbb{Y})$
        if $P \neq \emptyset$ then
            $G' \leftarrow G' \uplus P$
            $\mathbb{L}' \leftarrow \mathbb{L}' - \mathbb{L}''$
            $\texttt{change?} \leftarrow \mathit{true}$
            continue
        end if
        let $(P, \Pi, \mathbb{L}'') = \textsc{expr-heuristic}(\mathbb{Y})$
        if $P \neq \emptyset$ then
            $G' \leftarrow G' \uplus P$
            $\Pi' \leftarrow \Pi' \cup \Pi$
            $\mathbb{L}' \leftarrow \mathbb{L}' - \mathbb{L}''$
            $\texttt{change?} \leftarrow \mathit{true}$
            continue
        end if
        $\texttt{change?} \leftarrow \mathit{false}$
    end while
    for $i$ in $[0, 1]$ do
        $\mathbb{Y} \leftarrow \textsc{partition}(\textsc{apply-nesting}((G' \mathbin{\widetilde{\uplus}} \widetilde{\mathbb{M}}, \mathbb{F}, \mathbb{L}')))$
        $\widetilde{\mathbb{M}} \leftarrow \emptyset$
        for $(A, Y = (\cdot, \cdot, i_0, \{i_f\}, \cdot, \Lambda)) \in \mathbb{Y}$ do
            let $\mathbb{M} = \Psi^A_\Lambda(\textsc{shortest-path}(Y, i_0, i_f))$
            $\widetilde{\mathbb{M}} \leftarrow \widetilde{\mathbb{M}} \cup \{\mathbb{M}\}$
        end for
    end for
    return $(G', \Pi', \widetilde{\mathbb{M}})$
end function
```

Figure 4.30. The PARSYNTH-FULL algorithm.

4.4 Implementation

We have implemented Parsimony as a web application consisting of a graphical frontend, implemented primarily in ClojureScript, that communicates with a backend server implemented in Clojure and Java. The complete implementation consists of ~38K lines of code. We describe the salient features of our implementation here.

4.4.1 Backend Design

The backend's primary purpose is to service HORIZON requests in response to user interaction with the Token Labels Tab. In particular, every time the user adds an example token string, the backend receives a request to compute a new horizon with respect to the newly added string. If the user has modified a folder (e.g., by adding or removing a string), the backend receives a HORIZON query for the set all strings belonging to the folder. The result of each query is used directly to update the set of candidate lexer rules presented to the user. All queries operate on an R-DAG that we precompute offline.

Offline R-DAG Construction

To precompute the R-DAG, we scrape all patterns contained in a corpus of 81 existing lexer implementations provided in the ANTLR project's open-source grammar repository [24]. We use the BRICS finite-state automata library [16] to compile all compatible regexes (a total of 3018 unique regexes), then compute the language containment relation $\sqsubset^*$ by exhaustively querying BRICS whether $\mathscr{L}(a_1) \subset \mathscr{L}(a_2)$ for every pair a_1, a_2 of compiled automata. To ensure the existence of $\top_{\mathscr{D}}$, we seed the corpus a priori with the regex `.*` that matches all strings. To remove redundant regexes, we additionally query BRICS whether $\mathscr{L}(a_1) = \mathscr{L}(a_2)$; when such a pair is found, we simply discard one of the two from the corpus. The resulting regexes form the vertex set of an R-DAG* with corresponding edge set given by the $\sqsubset^*$ relation just computed. Finally, we compute the transitive reduction of this R-DAG* using the algorithm of Aho et al. for transitive reduction via adjacency matrix multiplication [1].

4.4.2 Frontend Design

In addition to all graphical portions of Parsimony, the frontend contains (a) an implementation of the CYK algorithm, including CNF conversion, (b) a library for constructing parse forests from CYK tables, including implementation of disambiguating filters, and (c) an implementation of the PARSYNTH-FULL algorithm, including utilities for construction and manipulation of CYK automata.

User Interactions

Every time the user executes the Solver, Parsimony constructs a parser synthesis constraint system consisting of (a) the current grammar, (b) all unsatisfied parse constraints the user has provided via textual selections, and (c) all files in the current project. Parsimony then executes PARSYNTH-FULL on the constructed constraint system, pausing for user interaction whenever required for forward progress. In particular, Parsimony requests user input for one of two reasons:

1. To request instantiations of schema placeholders during heuristic pattern matching.
2. To request the selection of valuations for each computed candidate matrix.

In both cases, Parsimony allows the user to preview the result of her choices by visualizing the parse forest underlying the relevant parse constraints. In this way, the user can better understand the impact of each choice before committing. After the user accepts all candidates, Parsimony inserts the relevant productions and disambiguating filters into the project's syntax specification, stored as a file with `.g` extension.

4.5 Evaluation

To evaluate the effectiveness of Parsimony, we conducted a user study in which 18 participants 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.

4.5.1 Hypotheses

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.

4.5.2 Participants

We targeted our user study at programmers who had some familiarity with parsing, but who were not experts. Computer Science students were recruited by an open call to internal student mailing lists within the UCSD Computer Science Department. The only requirement for participation was previous experience writing a parser, whether through classwork or personal use. We additionally asked respondents to self-rate their level of proficiency writing parsers, for the purpose of excluding respondents with high expertise. Our final sample pool consisted of 18 Computer Science students (9 undergraduate and 9 graduate) who were split by random assignment into a control group and an experimental group, each with 9 students. The control group consisted of 5 undergraduates and 4 graduate students, with the experimental group containing the remainder.

4.5.3 Interface Differences

The interface seen by the experimental group consisted of all features described in this paper: The Token Labels Tab and Solver Tab, along with parse tree visualizations and colorings. The interface seen by the control group had none of these features, but did retain the tabbed workspace, file browser, and text editors. Due to the lack of synthesis features, control participants had to manually code their definitions via text editor. The interface additionally provided buttons to compile and run these definitions – any errors were reported in a textual console (e.g., "Syntax error on line 1, found `foo` but expected `bar`"). The control interface was designed to closely model a traditional parsing workflow in which the user employs a text editor and relies on command-line compiler feedback.

4.5.4 Methodology

Both the control and experimental groups were first asked to read a brief introduction. Participants were then given a tutorial project to introduce them to the basic features of the interface. The content of this tutorial was tailored to the particular interface that the participant would see (either control or experimental), although the actual parsing and lexing tasks in each tutorial were the same.

After completing the tutorial, participants were given up to 2 hours to complete two projects asking them to implement lexers and parsers for two toy languages designed specifically for the experiment. We chose to use synthetic toy languages in order to minimize bias that might stem from participants' previous familiarity with existing languages – although synthetic, these languages were designed to contain syntactic constructs that occur commonly in real languages, such as literal primitives and data structures, loops, branches, and various statements.

The first project comprised a sequence of 7 focused tasks that together implemented the lexer and parser for the Fuyu language introduced in Section 4.1. Each of the seven tasks asked the student to implement a single syntactic construct of the language in isolation (e.g., keyword, numeric literal, expression, or statement). In order to finish a task, participants ran a battery of tests (provided by us) to determine whether their solution was correct. Participants were not allowed to proceed to the next task until these tests passed.

The second project consisted of two open-ended tasks in which users were asked to develop the parser for a toy language called Hachiya. These tasks were significantly more open-ended, as the participant was given a large sample source file and asked to implement the parser without any prescribed order. The project was not deemed complete until the participant's solution passed the provided tests.

After completing the experiment, participants were given a survey asking them to rate various aspects of their experience via 5-point Likert scales. They were additionally given an opportunity to provide open-ended feedback.

4.5.5 Quantitative Results

We discuss our quantitative results with respect to the two hypotheses.

Hypothesis 1

Based on our measurements, we find that Parsimony significantly improves the speed at which users are able to construct parsers. To show this, we describe two related measures of time-based participant performance: average time to completion per task, and total progress made. We discuss total progress first.

Figure 4.31 shows the number of participants able to complete each task, with tasks ordered sequentially from first to last. In the experimental condition, five participants were able to progress through all 9 tasks in the time allotted. In the control condition, only two participants were able to complete all tasks. The drop-off in the control condition begins at task 4, which was one of the more complex tasks, as it involved construction of a grammar for mathematical expressions built from variables and literals. Indeed, our measurements show that control participants took on average nearly 45 minutes to complete this task, while participants in our experimental group

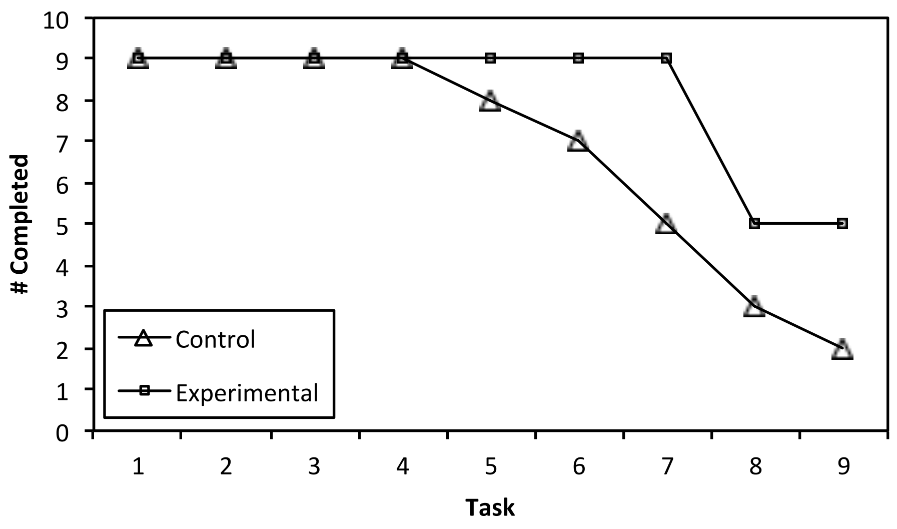


Figure 4.31. Number of participants completing each task.

took only half that time. By contrast, the drop-off in the experimental group begins much later at task 7 – in fact, all experimental group participants were able to complete the entirety of the Fuyu project.

Figure 4.32 shows average time to completion for each task. We average across only those users who successfully completed that task. The figure shows that Parsimony either matches, or significantly improves performance in the three most difficult tasks in the Fuyu project: 2, 4, and 5. Task 2 asked users to construct a lexer rule for numeric literals (recall the motivating example from Section 4.1). Task 4, as already mentioned, asked users to write productions for algebraic expressions. Finally, task 5 asked users to write productions for literal arrays that may contain an unbounded number of elements. In all three cases, control participants took at least twice as long to complete the task. We do note, however, that our results appear to show no significant speed advantage with Parsimony in tasks 8 and 9 – the averages for those tasks are biased toward the control group because they contain only the highest-performing minority from the control condition, compared against a larger group from the experimental condition.

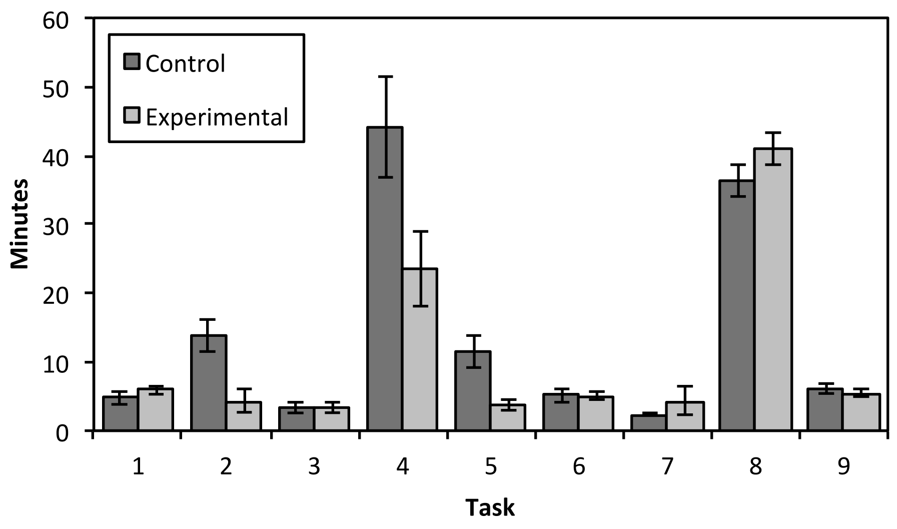


Figure 4.32. Average time to completion in minutes. Breakdown by task. Error bars show standard error.

Hypothesis 2

Based on our measurements, we find that Parsimony also significantly reduces the number of mistakes made by users. We measure two kinds of mistakes: compile errors, which occur when the participant specifies a bad rule that causes a compile failure; and reasoning errors, which occur when the participant specifies a rule, then later modifies or removes it. We additionally break these results down by the type of task performed: lexing tasks, which comprised the first three tasks in the Fuyu project, and parsing tasks, which were the remainder of the tasks.

Lexing Tasks: Compile Errors

Figure 4.33 shows the per-user average number of compile errors encountered during each of the three lexer tasks – in all three tasks, the experimental group significantly outperformed the control group. Additionally, note that both groups encountered the most compile errors in the first task, which likely stems from the fact that participants had just

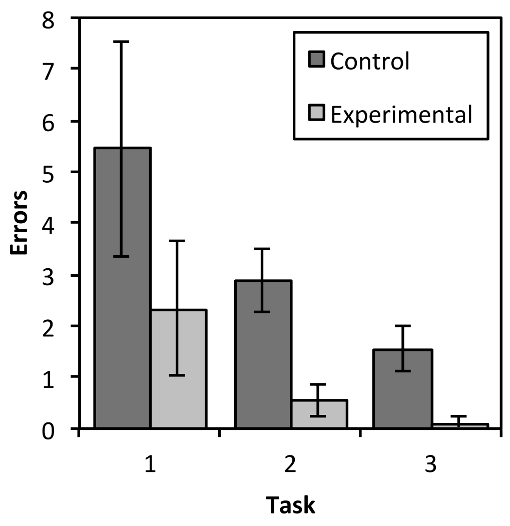


Figure 4.33. Average lexer compile errors per participant. Breakdown by task. Error bars show standard error.

finished the tutorial and were still getting used to the interface. Even so, the experimental group saw fewer than half as many compile errors as the control group in the first task, indicating that it was easier for participants to grow accustomed to Parsimony.

Lexing Tasks: Reasoning Errors

Next, we discuss reasoning errors. To measure reasoning errors, we recorded every lexer rule written by the participant. We call this the cumulative lexer rule set C_{lex}. We also recorded the set of lexer rules that appeared in the participant'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 participant had to edit and massage their result in response to errors in reasoning. The experimental group produced an average churn ratio of 1.2 while the control group saw a ratio of 3.1. In other words, the control group kept only 32% of rules and threw away more than two-thirds of attempted rules. By comparison, the experimental group kept nearly 81% of attempted rules.

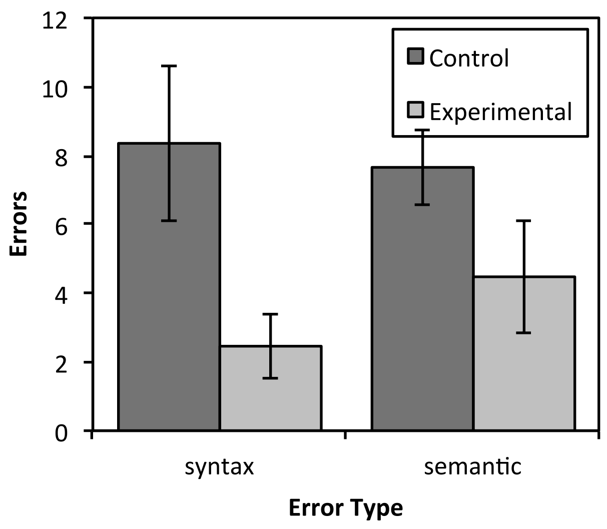


Figure 4.34. Average parser compile errors per participant. Breakdown by type. Error bars show standard error.

Parsing Tasks: Compile Errors

Figure 4.35 shows the per-user average number of compile errors encountered during each of the six parser tasks – again, the experimental group significantly outperformed the control group. To more finely distinguish the contributing factors for this trend, we additionally classify these errors into two different bins: syntax errors and semantic errors. The average breakdown per participant is shown in Figure 4.34. Syntax errors are self-explanatory. Semantic errors occur when the compiler fails a semantic check, which occur when (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). We see a significant reduction in both syntax and semantic errors, indicating 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.

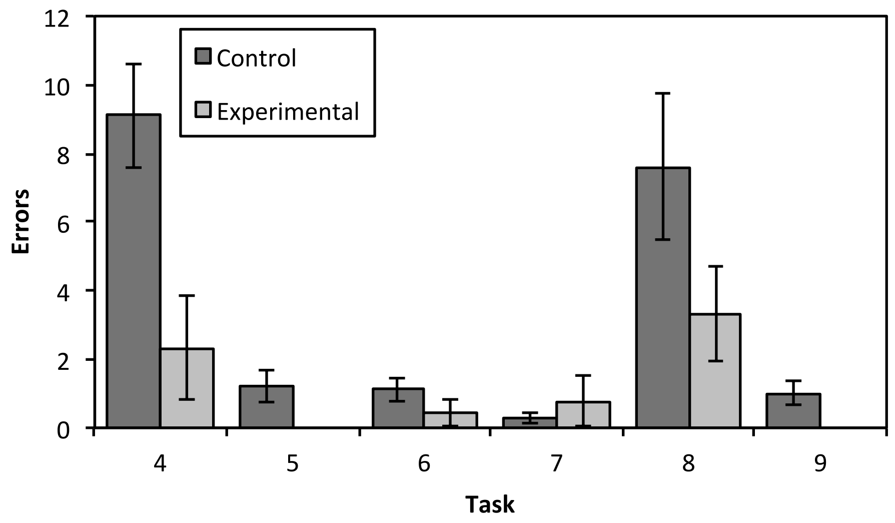


Figure 4.35. Average parser compile errors per participant. Breakdown by task. Error bars show standard error.

Parsing Tasks: Reasoning Errors

To measure reasoning errors in parsing tasks, we use a similar methodology to that for lexer errors. We recorded production sets C_{parse} and F_{parse} for each participant, which contained every production written by the participant and every production in the participant's final answer, respectively. The ratio $|C_{parse}| / |F_{parse}|$ is the parser churn ratio, analogous to the lexer churn ratio. In the experimental condition, the average churn ratio was 1.31, which tells us that participants 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, indicating that nearly half of all productions they wrote were eventually discarded.

4.5.6 Qualitative Results

We now discuss the qualitative results gathered from our exit surveys. User responses were uniformly positive across all metrics.

Numerical User Responses

The exit survey contained a series of Likert scale questions on a scale of 1-5, with 5 being most affirmative. The results are summarized in Table 4.2. The first six questions were only given to the experimental group, as they reference features only visible in the experimental interface. The purpose of these questions was to get feedback on the usefulness of each feature of Parsimony individually. In order, the questions asked if the following were useful: (a) the colorings applied to text editors, (b) the parse tree visualizations, (c) the Token Labels Tab for synthesizing lexer rules, (d) the Solver for synthesizing productions, (e) the heuristic pattern detection for sequence-like constructs, and (f) the heuristic pattern detection for algebraic expressions. The last three questions were given to both groups for the purpose of comparing responses between groups. In order, these questions asked (g) if the tool was useful overall, (h) if the tool was easier to use than software they had previously used to construct parsers, and (i) if they would recommend the tool to others. Answers to questions (a)-(f) were uniformly positive, with the average rating for each above 4.5. In other words, users seemed to agree that every feature of Parsimony was valuable. Responses to questions (g)-(i) show that participants in the experimental group seemed to favor Parsimony more than those of the control group, across all three metrics.

Other Responses

Participants were also asked to provide general comments on their most negative and positive impressions of the tool. We summarize a few of the most common sentiments expressed by participants in the experimental group.

Table 4.2. Numerical user responses to exit survey. Higher is better. E and C correspond to the experimental and control group, respectively.

	Question	E	C
(a)	Coloring	4.78	n/a
(b)	Live Parse View	4.67	n/a
(c)	Token Labels Tab	4.56	n/a
(d)	Solver	4.78	n/a
(e)	Sequence Heuristic	4.56	n/a
(f)	Expression Heuristic	4.67	n/a
(g)	Overall	4.78	4.00
(h)	Easier to Use	4.56	4.00
(i)	Recommend to Others	4.78	4.00

Speed and ease of development

Six of the participants mentioned that they liked most the speed and ease of development. One participant, for instance, wrote that the "tool took three week's worth of work into three hours."

High-level focus

Five participants mentioned that they enjoyed that Parsimony seems to eliminate much of the detailed work involved in writing a parser, thus allowing them to focus on high-level design instead. For example, one participant wrote that he could "focus on the high level or overall organization of the language rather than on the brute force aspect."

Quality of feedback

Four participants stated that they thought highly of the strong feedback loop and the quality of the feedback. A participant wrote, "The graphical representation was much better/faster feedback than normal tools." Another wrote, "Animations with visual tree and colors helped me understand what was happening very easily."

Participants also mentioned several areas for improvement. These tended to focus on various usability problems with the interface.

Organization

Three participants mentioned that they thought the organization of the user interface was confusing, especially with regard to the dichotomy between the Token Labels Tab and Solver Tab. One participant wrote that “lexing and parsing examples could have a single UI.”

Graphical Undo

Five participants mentioned that they would have liked to see a way to graphically remove or undo inferences, instead of having to delete them using the text editor: “It would be better to rename or remove some parser rules graphically.”

Keyboard Shortcuts

Three participants suggested that the process would have been much smoother if they had access to more keyboard shortcuts for performing common operations. One participant lamented the “lack of [a] hotkey for addition of selected text to [the] Token Labels Tab."

4.6 Acknowledgements

This chapter, in full, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 5

Related Work

5.1 Program Synthesis

The topic of program synthesis [37] focuses on the problem of inferring a program from specifications. Such specifications can take many forms, such as input/output examples [27], logical constraints [68], or even partial reference implementations [67]. We focus here on example-based synthesis, into which the work of this dissertation falls.

Programming-by-Example

The class of algorithms for program synthesis via input/output examples is broadly termed *programming-by-example* (PBE) [13, 28, 48, 61]. Some of the earliest examples of PBE systems focused on inference of simple Lisp programs from example executions [47, 65]. More recently, programming-by-example has become re-energized by the promise to give non-expert users the ability to generate useful programs: a primary benefit of PBE systems is that when successfully realized, they reduce the amount of domain knowledge required to produce a relatively complex computational artifact. In general, this level of interaction is achieved by deep domain-specific insights on the part of the algorithm designers. Examples of such proposed programming-by-example techniques include synthesis of string transformations [5, 27, 51], spreadsheet manipulations [7, 29], input specifications for fuzzers [8], number transformations [66], and data extraction from unstructured or semi-structured inputs [44]. One particularly relevant instantiation of PBE is the PADS system [18, 19, 20] for automatically inferring the structure of ad-hoc data formats such as log files. In contrast to our system, PADS makes several simplifying assumptions about its inputs due to its focus on ad-hoc formats: for example, their system assumes 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.

Interactive Graphical Program Synthesis

Some program synthesis systems have a visual component similar to our own. For example, LAUNCHPADS [14] is the visual frontend for the PADS system described above. Much like Parsify and Parsimony, LAUNCHPADS provides an interface into which a user highlights various regions of a sample file according to the type of data contained within. However, the system is essentially a visual system for performing edits to PADS specifications and does not provide access to example-based inference features. The LAPIS [54] and SMARTedit [42] systems support repetitive edits by example, but do not export any hierarchical structure from the underlying text, as their aim is to aid repetitive text editing tasks, not to produce structured output. The STEPS [74] system also provides a facility for highlighting and labeling regions similar to Parsify and Parsimony, but with the narrower aim of facilitating more limited text transformations akin to that accomplished by “a short Perl/AWK script.” In all cases, the key feature that distinguishes our work from previous work on visual program synthesis environments is the broader scope of the task: parsers for full context-free languages that are capable of producing meaningfully structured parse trees.

Version Space Algebra

The CYK automaton-based formulation of constraint satisfaction presented in Chapter 4 is inspired by the notion of *version space algebras* [43], in which the space of possible solutions (the hypothesis space) is encoded by a special structure called a version space. By composing version spaces representing individual examples, we produce a set of hypotheses consistent with the *set of examples* provided. This formulation was developed as an extension to Mitchell's original version space framework [55] in concert with the development of the SMARTedit [41] system described above. In our setting, a CYK automaton represents a hypothesis space consisting of a set of possible productions. Intersection of such automata produces a new hypothesis space that remains consistent with the given parse constraints.

5.2 Grammatical Inference

The topic of *grammatical inference* [15, 72], 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* [23], 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). Inference of probabilistic syntactic rules for natural languages [34, 71], as studied by computational linguists, is a well known field of study within this genre. In non-probabilistic settings, steady progress has been made on inferring language specifications for regular languages [3, 58], reversible languages [2], reversible context-free grammars with structural descriptions [62], and even context-free languages [45, 60]. 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 [38] found known efforts to apply grammatical inference to programming language parsers [17, 52] 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 those described in this dissertation, that allow more fine-grained control of the form of inferred productions.

5.3 Parsing

The literature on parsing and its applications is enormous, spanning several decades. We focus here on work towards making parsing more user-friendly: parsing algorithms for easier syntax specification, and algorithms for analyzing, debugging, and testing parsers.

Recent work [46] has employed natural-language processing to generate parsers from English descriptions of input formats. Generalized parsers of the GLL [35, 64] and GLR [70] families have recently gained popularity due to their ability to accept any context-free grammar, in contrast to well-known parser families such as LR(k) [40] whose difficulties are well-studied [36], due in large part to the restricted form of its grammars. Advances in this line of research have led to the development of generalized parser generators that offer high performance, such as the Elkhound [50] parser generator. Unfortunately, the ability to specify ambiguous grammars remains a significant disadvantage in comparison to parser generators based on more restricted grammars. More so, many such parsers offer no mechanism for disambiguation besides refactoring productions. Disambiguating filters [39, 69] offer a declarative mechanism for disambiguation without modifications to the grammar. This is the disambiguation approach taken by the work of this dissertation.

Another line of research focuses on tooling and analyses for detecting problems in existing syntax specifications. For example, significant work has gone into the development of techniques for detecting ambiguities statically [9, 11, 63]. However, no sound and complete method can exist, as detecting ambiguity in context-free grammars is undecidable in general [31]. To get around this issue, tools make use of additional information such as example parse trees [10], or restrict themselves to a particular subset of grammars, such as LALR [33]. The work described above can be seen as complementary to the work presented in this dissertation: Parsify and Parsimony focus primarily on methods for aiding developers to construct parsers, whereas the methods described above focus on aiding developers to mend existing parsers.

5.4 Acknowledgements

This chapter, in part, is adapted from material as it appears in Leung, Alan; Sarracino, John; Lerner, Sorin. “Interactive Parser Synthesis by Example,” Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015. The dissertation author was the primary investigator and author on this paper.

This chapter, in part, is adapted from material currently being prepared for submission for publication. Leung, Alan; Lerner, Sorin. “Parsimony: An IDE for Example-Guided Synthesis of Lexers and Parsers.” The dissertation author was the primary investigator and author on this paper.

Chapter 6

Conclusion

Parsing is a ubiquitous programming task that remains a challenge for non-expert programmers. In this dissertation, we have presented two systems in support of our thesis that it is possible to make writing parsers easier through a combination of program synthesis and visual feedback. In Chapter 3, we presented Parsify, our first system for synthesizing parsers by example. The primary interaction model of Parsify is that of the input/output example – the user provides labeled textual selections, and the system infers productions that satisfy the intent of the labels. We provided evidence of Parsify's effectiveness by performing a series of case studies demonstrating that it is possible to construct useful parsers using Parsify. In Chapter 4, we identified three limitations of Parsify, namely its lack of lexing support, its inability to learn from multiple examples, and its use of ad-hoc generalization. We then presented Parsimony, which addressed Parsify's limitations with additional capabilities for lexer inference, multiple-example learning, and parametric heuristic pattern detection. Finally, we presented the results of a controlled user study with non-expert programmers, which showed that Parsimony does indeed help users write parsers more quickly and with fewer mistakes.

We close by mentioning avenues for future research. A natural extension of this work follows from considering that a parser is generally only the first part of a much larger workflow. One important, and also non-trivial task, is that of transforming concrete parse trees, as generated by a parser, into abstract syntax trees that have been stripped of their syntactic sugar and any semantically unimportant bits (e.g., punctuation). For parser generators that allow it, the traditional way of performing this transformation is to write code inline in the syntax specification (so-called semantic actions) that perform this transformation at parse time. As with any non-trivial programming task, this is prone to error. It also conflates the task of language specification (e.g., the grammar) with the task of program translation. A fresh take on this task could be to view this as yet another instance of a synthesis problem: given example concrete parse trees and their abstract syntax counterparts, can one synthesize the program that performs this transformation? Although some work has already been done on synthesizing tree transformations on hierarchical structures [73], perhaps domain-specific insights from parsing theory can provide for additional expressivity and more efficient inference while still keeping the task tractable.

Another avenue for future work presents itself when we consider that, for a development environment to be useful, it must be compatible with the downstream consumers of its outputs. For whatever reason, it may be the case that a developer must have an LALR grammar due to external constraints. In this more restricted setting, it would be interesting to discover how far synthesis can aid in translating grammars to this more restricted form, and how much of the groundwork laid in this dissertation can be extended to that setting.

Finally, consider that much of a parser's development lifecycle is spent in maintenance. Once constructed, unless quite trivial the parser will likely require changes over its lifetime. Program synthesis techniques have already shown progress in the domain of patch generation for imperative programs [49]. Perhaps it would be fruitful to view refactoring of parsers in a similar light?

Appendix A

CYK-based Coloring

This appendix describes the modified coloring algorithm used by Parsimony. The primary benefit of this algorithm is its easy integration with an existing CYK parser – simply compute colorings at parse time – because the algorithm exploits similar optimal substructure to that of the CYK algorithm: an optimal coloring is the composition of optimal subcolorings.

The algorithm is defined in Figure A.1. We describe the algorithm informally here. Let τ be a string on which we wish to compute a coloring, and M be a CYK table computed from τ. Assume we have optimal colorings for all substrings of τ. First, we check $M_{0,|\tau|}$ for a non-empty entry. If so, then that means the entire string is derivable from nonterminals in $M_{0,|\tau|}$, so we just construct a coloring directly from the table elements and return. If not, then that means the string is not derivable from any nonterminal of the grammar. Instead, we attempt to construct a new coloring by splitting τ into two, then taking the union of the coloring for the left-hand part and the coloring for the right-hand part. We do this for every possible 2-way splitting and return the best coloring found in this way. The procedure just outlined is performed for every substring of length 1, then 2, and so on until $|\tau|$. The score of a coloring is given by the 3-tuple $(cov, span, num)$, where *cov* is the number of colored terminals, *num* is the number of colored boxes (i.e., labels), and *span* is size of the largest such colored box. The comparison relation $\sqsupset$ simply gives preference to the score, in lexicographic order, with highest *cov*, then highest *span*, then lowest *num*. Finally, REMOVE-SUBSUMES$(G,\mathscr{C})$ removes any element of the coloring $\mathscr{C}$ subsumed by another element.

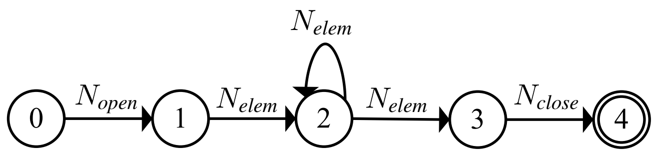
```
function CYK-COLOR($G, \tau, M$)
    let $(N, \cdot, \cdot, \cdot) = G$
    let $w$ = $width\ of\ M$
    for $i$ in $[0, w)$ do
        $\mathbb{C}_{i,1} \leftarrow \{(A, i, i+1) \mid A \in (M_{i,1} \cap N \cup \{\tau_{[i]}\})\}$
        $\mathbb{S}_{i,1} \leftarrow (1, 1, 1)$
    end for
    for $l$ in $[2, w]$ do
        for $i$ in $[0, w-l]$ do
            COLOR-ONE($N, M, i, l, \mathbb{C}, \mathbb{S}$)
        end for
    end for
    return REMOVE-SUBSUMES($G, \mathbb{C}_{0,w}$)
end function

function COLOR-ONE($N, M, i, l, \mathbb{C}, \mathbb{S}$)
    let $nts = M_{i,l} \cap N$
    if $|nts| > 0$ then
        $\mathbb{C}_{i,l} \leftarrow \{(A, i, i+l) \mid A \in nts\}$
        $\mathbb{S}_{i,l} \leftarrow (l, l, 1)$
        return
    end if
    $best \leftarrow \emptyset$
    $bestScore \leftarrow (-\infty, -\infty, \infty)$
    for $k$ in $[1, l)$ do
        let $(cov_L, span_L, num_L) = \mathbb{S}_{i,k}$
        let $(cov_R, span_R, num_R) = \mathbb{S}_{i+k,l-k}$
        let $score = (cov_L + cov_R, max(span_L, span_R), num_L + num_R)$
        if $score \sqsupset bestScore$ then
            $best \leftarrow \mathbb{C}_{i,k} \cup \mathbb{C}_{i+k,l-k}$
            $bestScore \leftarrow score$
        end if
    end for
    $\mathbb{C}_{i,l} \leftarrow best$
    $\mathbb{S}_{i,l} \leftarrow bestScore$
end function
```

Figure A.1. The CYK-COLOR algorithm.

Appendix B

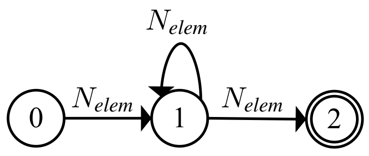
Additional Patterns and Schemas

This appendix shows the CYK automata and production schemas used by Parsimony's pattern detection heuristics to detect the following three syntactic constructs: (a) enclosed, undelimited lists; (b) unenclosed, delimited lists; and (c) unenclosed, undelimited lists. We show here only the automaton and corresponding schema. See Section 4.3.4 for actual discussion of the algorithm.



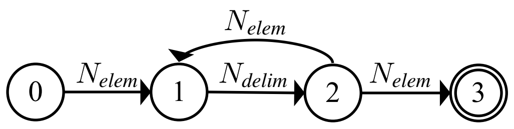
$$P^{\bullet}_{eulist} = \left\{ \begin{array}{l} \quad \bullet_{lhs} \rightarrow \bullet_{open}\, A_{fresh}\, \bullet_{close} \\ A_{fresh} \rightarrow \bullet_{elem} \\ A_{fresh} \rightarrow \bullet_{elem}\, A_{fresh} \end{array} \right\}$$

Figure B.1. Enclosed and undelimited list.



$$P^{\bullet}_{ulist} = \left\{ \begin{array}{l} \bullet_{lhs} \rightarrow \bullet_{elem} \\ \bullet_{lhs} \rightarrow \bullet_{elem} \bullet_{lhs} \end{array} \right\}$$

Figure B.2. Unenclosed and undelimited list.



$$P^{\bullet}_{dlist} = \left\{ \begin{array}{l} \bullet_{lhs} \rightarrow \bullet_{elem} \\ \bullet_{lhs} \rightarrow \bullet_{elem} \bullet_{delim} \bullet_{lhs} \end{array} \right\}$$

Figure B.3. Unenclosed and delimited list.

Bibliography

[1] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] Dana Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, July 1982.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[4] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[5] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *Proceedings of the VLDB Endowment*, 2(1):514–525, August 2009.

[6] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 126–137, New York, NY, USA, 1996. ACM.

[7] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin 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*, PLDI '15, pages 218–228, New York, NY, USA, 2015. ACM.

[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *CoRR*, abs/1608.01723, 2016.

[9] H. J. S. Basten. *Tracking Down the Origins of Ambiguity in Context-Free Grammars*, pages 76–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Hendrikus J. S. Basten and Jurgen J. Vinju. *Parse Forest Diagnostics with Dr. Ambiguity*, pages 283–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176 – 191, 2010.

[12] census-postgres. https://github.com/leehach/census-postgres, 2014.

[13] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[14] Mark Daly, Mary F. Fernández, Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. LAUNCHPADS: A system for processing ad hoc data. In *PLAN-X 2006 Informal Proceedings, Charleston, South Carolina, January 14, 2006*, pages 90–91, 2006.

[15] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38(9):1332–1348, September 2005.

[16] dk.brics.automaton. http://www.brics.dk/automaton/index.html, 2016.

[17] A. Dubey, S.K. Aggarwal, and P. Jalote. A technique for extracting keyword based rules from a set of programs. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 217–225, March 2005.

[18] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[19] Kathleen Fisher, David Walker, and Kenny Q. Zhu. LearnPADS: Automatic tool generation from ad hoc data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1299–1302, New York, NY, USA, 2008. ACM.

[20] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.

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

[22] GNU Software Foundation. *GNU Bison manual*.

[23] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[24] grammars v4. https://github.com/antlr/grammars-v4, 2016.

[25] Robert Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[26] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[27] Sumit 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*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[28] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14, Sept 2012.

[29] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 317–328, New York, NY, USA, 2011. ACM.

[30] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[31] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[32] instaparse. https://github.com/Engleberg/instaparse, 2014.

[33] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 555–564, New York, NY, USA, 2015. ACM.

[34] Mark Johnson. PCFG models of linguistic tree representations. *Comput. Linguist.*, 24(4):613–632, December 1998.

[35] Adrian Johnstone and Elizabeth Scott. *Modelling GLL Parser Implementations*, pages 42–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[36] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of the ACM International*

Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 918–932, New York, NY, USA, 2010. ACM.

[37] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming*, pages 50–73. Springer, Berlin, Heidelberg, Berlin, Heidelberg, September 2009.

[38] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005.

[39] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, 1994.

[40] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[41] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture*, K-CAP '03, pages 36–43, New York, NY, USA, 2003. ACM.

[42] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, October 2003.

[43] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[44] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM.

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

[46] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 1294–1303, 2013.

[47] Henry Lieberman. An example based environment for beginning programmers. *Instructional Science*, 14(3):277–292, 1986.

[48] Henry Lieberman. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[49] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[50] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer Berlin Heidelberg, 2004.

[51] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–187–I–195. JMLR.org, 2013.

[52] Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett R. Bryant. Can a parser be generated from examples? In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1063–1067, New York, NY, USA, 2003. ACM.

[53] Matthew Might and David Darais. Yacc is dead. *CoRR*, abs/1010.5023, 2010.

[54] Robert C. Miller and Brad A. Myers. Lightweight structured text processing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 10–10, Berkeley, CA, USA, 1999. USENIX Association.

[55] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203 – 226, 1982.

[56] MonitorWare. Apache (unix) log samples. http://www.monitorware.com/en/logsamples/apache.php, 2014.

[57] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, NAACL 2000, pages 249–255, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[58] Jose Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*, volume 5, pages 99–108. World Scientific, 1992.

[59] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[60] Georgios Petasis, Georgios Paliouras, Constantine D. Spyropoulos, and Constantine Halatsis. eg-GRIDS: Context-free grammatical inference from positive examples using genetic search. In *Grammatical Inference: Algorithms and Applications: 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004. Proceedings*, pages 223–234. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[61] Oleksandr Polozov and Sumit 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*, OOPSLA 2015, pages 107–126, New York, NY, USA, 2015. ACM.

[62] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23 – 60, 1992.

[63] Sylvain Schmitz. *Conservative Ambiguity Detection in Context-Free Grammars*, pages 692–703. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[64] Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177 – 189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[65] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pages 260–267, San Francisco, CA, USA, 1975. Morgan Kaufmann Publishers Inc.

[66] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

[67] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[68] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[69] Mikkel Thorup. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica*, 33(5):511–522, 1996.

[70] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[71] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 104–, New York, NY, USA, 2004. ACM.

[72] Enrique Vidal. Grammatical inference: An introductory survey. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin Heidelberg, 1994.

[73] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 508–521, New York, NY, USA, 2016. ACM.

[74] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 495–504, New York, NY, USA, 2013. ACM.

[75] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.